

Advanced Qt Programming

Creating Great Software
with C++ and Qt 4

Mark Summerfield

H I G H T E C H

Qt

Профессиональное программирование

Разработка кроссплатформенных
приложений на C++

Марк Саммерфилд



Санкт-Петербург — Москва
2011

Серия «High tech»
Марк Саммерфилд

Qt. Профессиональное программирование **Разработка кроссплатформенных приложений на C++**

Перевод А. Слинкина

Главный редактор	<i>А. Галунов</i>
Зав. редакцией	<i>Н. Макарова</i>
Выпускающий редактор	<i>П. Щеголев</i>
Научный редактор	<i>В. Силицын</i>
Редактор	<i>Е. Тульсанова</i>
Корректор	<i>С. Минин</i>
Верстка	<i>К. Чубаров</i>

Саммерфилд М.

Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 560 с., ил.
ISBN 978-5-93286-207-0

Qt представляет собой мощную библиотеку разработки кроссплатформенных приложений – настольных, сетевых и для мобильных устройств. Но нередко даже опытные Qt-программисты используют лишь малую часть заложенных в ней возможностей.

Книга Марка Саммерфилда открывает путь к овладению разнообразными паттернами и приемами создания приложений с использованием Qt. Разработчики узнают, как в полной мере воспользоваться самыми ценными из новых API, появившихся в последних версиях. Основное внимание уделено подходам, обладающим наибольшей эффективностью и гибкостью, но при этом не создающим дополнительных сложностей.

Основной акцент сделан на создании моделей, графических представлений и гибридных приложений «рабочий стол + Интернет», на многопоточной обработке данных и приложениях, содержащих мультимедийные объекты и форматированный текст. Представлено подробное введение в подсистемы анимации и конечных автоматов, включенные в версию Qt 4.6.

В книге приведены примеры кода, протестированные на платформах Windows, Mac OS X и Linux с использованием Qt 4.6 (а многие работают также с версией Qt 4.5) и написанные с ориентацией на будущие версии Qt.

ISBN 978-5-93286-207-0
ISBN 978-0-321-63590-7 (англ)

© Издательство Символ-Плюс, 2011
Authorized translation of the English edition © 2011 Qttrac Ltd. This translation is published and sold by permission of Qttrac Ltd., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛПН N 000054 от 25.12.98.
Подписано в печать 31.03.2011. Формат 70×100¹/₁₆. Печать офсетная.
Объем 35 печ. л. Тираж 1200 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

*Книга посвящается
Анне Ребекке Патерсон*

Оглавление

Предисловие	11
Глава 1. Гибридные приложения: рабочий стол + Интернет	19
Виджеты с поддержкой Интернета	21
Использование WebKit	36
Универсальный компонент, содержащий окно веб-браузера	38
Создание приложений для работы с конкретным сайтом	46
Внедрение виджетов Qt в веб-страницы	60
Глава 2. Аудио и видео	69
Использование классов QSound и QMovie.....	70
Мультимедийная библиотека Phonon.....	76
Воспроизведение музыки	81
Воспроизведение видео	98
Глава 3. Архитектура модель/представление: табличные модели	105
Архитектура модель/представление в Qt	106
Применение класса QStandardItemModel для таблиц.....	108
Изменение табличной модели с помощью пользовательского интерфейса.....	110
Подкласс QStandardItemModel для таблиц	121
Удаление строк-дубликатов с помощью класса QSortFilterProxyModel	126
Применение класса QSortFilterProxyModel для включения нужных строк	128
Создание пользовательских табличных моделей	132
Изменение табличной модели с помощью пользовательского интерфейса.....	133
Пользовательский подкласс QAbstractItemModel для таблиц	135
Глава 4. Архитектура модель/представление: древовидные модели	149
Деревья на основе модели QStandardItemModel	151
Изменение древовидной модели с помощью пользовательского интерфейса.....	152

Подкласс QStandardItem	
для представления элементов дерева	162
Подкласс QStandardItemModel	
для представления дерева	163
Создание пользовательских древовидных моделей	171
Изменение древовидной модели	
с помощью пользовательского интерфейса.....	173
Специализированный класс элемента дерева	176
Специализированный подкласс	
QAbstractItemModel для деревьев.....	179
Глава 5. Архитектура модель/представление: делегаты	207
Редакторы для конкретных типов данных	208
Делегаты, ориентированные на конкретный тип данных	210
Делегат для строки или столбца, допускающих только чтение.....	211
Делегат для редактируемой строки или столбца	216
Делегаты для конкретной модели	223
Глава 6. Архитектура модель/представление: представления	229
Подклассы QAbstractItemView	230
Визуализация моделей	
с помощью специальных представлений.....	247
Виджет визуализатора.....	248
Агрегированный заголовок визуализатора	254
Агрегированное представление визуализатора	258
Глава 7. Многопоточное программирование	
с модулем QtConcurrent	269
Выполнение функции в потоке.....	273
Функция QtConcurrent::run()	277
Класс QRunnable.....	282
Фильтрация и отображение в потоках	286
Использование QtConcurrent для фильтрации.....	296
Применение модуля QtConcurrent для фильтрации с редукцией ...	303
Применение модуля QtConcurrent для отображения	308
Глава 8. Многопоточное программирование	
с классом QThread	313
Обработка независимых элементов	314
Обработка разделяемых данных	329
Глава 9. Создание редакторов форматированного текста	345
Введение в класс QTextDocument.....	346
Создание специализированных текстовых редакторов.....	348
Автозавершение в однострочных редакторах	
и комбинированных списках	349

Автозавершение и подсветка синтаксиса в текстовых редакторах	351
Однострочный редактор форматированного текста	373
Многострочный редактор форматированного текста	383
Глава 10. Создание форматированных текстовых документов	389
Качество файла, в который экспортирован QTextDocument	391
Создание объектов QTextDocument	395
Создание QTextDocument с помощью HTML	395
Создание QTextDocument с помощью QTextCursor	398
Экспорт и печать документов	402
Экспорт QTextDocument	403
Печать и предварительный просмотр документа QTextDocument	408
Рисование страниц	410
Рисование PDF- и PostScript-документов	418
Рисование SVG-документов	418
Рисование растровых изображений	419
Глава 11. Подсистема Graphics View: создание окон	421
Архитектура Graphics View	422
Графические представления: виджеты и компоновщики	425
Введение в графические элементы	433
Глава 12. Подсистема Graphics View: создание сцен	443
Сцены, элементы и действия	445
Создание главного окна	446
Сохранение, загрузка, печать и экспорт сцен	449
Манипулирование графическими элементами	457
Расширение класса QGraphicsView	474
Создание стыкуемого виджета с набором инструментов	475
Создание пользовательских графических элементов	482
Расширение класса QGraphicsTextItem	483
Расширение существующего класса графических элементов	491
Создание пользовательского графического элемента с нуля	494
Глава 13. Анимация и конечные автоматы	505
Введение в подсистему анимации	506
Введение в подсистему конечных автоматов	510
Сочетание анимации с конечными автоматами	517
Эпилог	528
Список литературы	532
Алфавитный указатель	535

Список таблиц

Таблица 1.1. Основные классы, входящие в состав WebKit	37
Таблица 1.2. Глобальные вспомогательные функции Qt	61
Таблица 2.1. Основные классы Phonon	80
Таблица 3.1. API класса QAbstractItemModel	137
Таблица 3.2. Перечисление Qt::ItemDataRole	139
Таблица 3.3. Перечисление Qt::ItemFlag	140
Таблица 4.1. API перетаскивания в классе QAbstractItemModel	180
Таблица 5.1. API класса QStyledItemDelegate	216
Таблица 6.1. API класс QAbstractItemView	231
Таблица 9.1. API класса QTextCursor (часть 1)	356
Таблица 9.2. API класса QTextCursor (часть 2)	357
Таблица 9.3. API класса QTextCursor (часть 3)	358
Таблица 9.4. Перечисление QTextCursor::MoveOperation)	362
Таблица 11.1. API класса QGraphicsItem, избранные методы (часть 1)	436
Таблица 11.2. API класса QGraphicsItem, избранные методы (часть 2)	437
Таблица 11.3. API класса QGraphicsItem, избранные методы (часть 3)	439
Таблица 11.4. API класса QGraphicsItem, избранные методы (часть 4)	440
Таблица 11.5. Перечисление Qt::ItemSelectionMode	440
Таблица 11.6. Перечисление QGraphicsItem::GraphicsItemFlag (часть 1)	441
Таблица 11.7. Перечисление QGraphicsItem::GraphicsItemFlag (часть 2)	441

Предисловие

В далеком 1991 году сидел я как-то на скамеечке в парке норвежского городка Тронхейм в компании Хаварда Норда. Мы проходили в местной больнице альтернативную воинскую службу и должны были разработать программу для хранения и анализа изображений, полученных в процессе УЗИ. В больнице был настоящий компьютерный зоопарк, поэтому требовалось, чтобы система работала в операционных системах UNIX, Mac и Windows. Задача была непростая, поэтому мы принялись разыскивать на рынке подходящие библиотеки классов. И пришли в ужас от качества того, что удалось найти. На той скамеечке мы решили написать свою библиотеку.

Мы были молоды, амбициозны и наивны. Устав от бесплодных попыток разобраться в использовании интуитивно непонятных инструментов и библиотек, мы задумались над тем, как улучшить положение. Мы хотели всего-навсего изменить мир разработки ПО, облегчив тем самым жизнь программистов. Чтобы они могли сосредоточиться на том, что, как все мы знаем, и составляет привлекательную сторону программирования: творческий подход и хорошо написанный код на выходе. Вот так мы и создали первые, еще далекие от совершенства версии Qt и спустя несколько лет основали компанию Trolltech.

Думаю, что хотя бы отчасти мы достигли поставленной цели. С момента выхода первой версии в 1995 году Qt добилась оглушительного успеха.

В 2008 году компанию Trolltech приобрела Nokia, а в апреле 2009 года я решил, что пора уходить. Итак, проработав в компании 15 лет и 27 дней, я покинул ее.

Продукт находится в хороших руках, и вся команда трудится над ним так же усердно и увлеченно, как и раньше. «Тролли» в Nokia делают все возможное для того, чтобы Qt и дальше сохраняла ту непробиваемую надежность, которую от нее ждут. Команда Ларса Кнолла (Lars Knoll) (известного разработкой KHTML и WebKit) ныне насчитывает почти 150 инженеров, занимающихся только Qt. Nokia добавила к ранее существовавшим условиям лицензирования еще и LGPL, что сделало Qt доступной более широкому кругу разработчиков.

Этой осенью Nokia пригласила меня в качестве почетного гостя на Дни разработчиков Qt (Qt Developer Days) в немецком городе Мюнхене. Эта конференция пользователей (проводимая также в США) – знаменательное событие для поклонников Qt, причем масштаб ее с каждым годом растет. Было просто великолепно побеседовать с пользователями Qt со

всей Европы. Многие разработчики говорили мне, что Qt по-настоящему изменила их работу. Приятные слова для старины-программиста.

Но тот факт, что Qt – хороший инструмент и удачная библиотека классов, – лишь половина успеха. Не обойтись также без хорошей документации и учебных пособий. В конце концов, целью-то было облегчить жизнь разработчиков.

Вот почему тогда, в 2003 году, я с готовностью воспринял предложенную мне идею. Я был президентом Trolltech, и в один прекрасный день ко мне зашел Марк Саммерфилд, начальник отдела документации. Он хотел совместно с Жасмином Бланшетом написать книгу о Qt. По-настоящему хорошую книгу, авторы которой близко знакомы с продуктом и горят желанием разложить все по полочкам. Кто лучше подходит для такой работы, как не глава отдела документации в компании с одним из лучших разработчиков Qt?

Результатом стала замечательная книга о Qt, которая с тех пор не раз переиздавалась и дополнялась.

А теперь Марк завершил еще один важный проект.

В арсенале программистов на Qt не хватало книги по продвинутому использованию этой библиотеки. Я очень рад, что Марк написал ее. Он великолепный технический писатель и обладает всеми знаниями, необходимыми для авторитетного рассказа о программировании на Qt. Меня всегда впечатляло собственное ему внимание к деталям и умение ясно и понятно излагать свои мысли. В общем, вы не пожалеете о времени, потраченном на чтение!

Сейчас вы держите в руках книгу, дающую вам отличную возможность расширить свои знания о том, какие потрясающие вещи можно делать с помощью Qt.

Удачного программирования!

*Эрик Чамбе-Инг
Южные Альпы, Франция,
24 декабря 2009 года*

Введение

Я уже давно хотел написать книгу по Qt, в которой освещались бы темы, слишком сложные для «C++ GUI Programming with Qt 4»¹, хотя и она показалась некоторым читателям отнюдь не тривиальной. Кроме того, имеются кое-какие специализированные вопросы – не все из них сложные, – которым просто не нашлось места в первой книге по программированию с помощью Qt. И наконец, сам размер библиотеки Qt таков, что в одной книге вряд ли реально рассказать обо всех ее возможностях, так что понадобилось еще одно издание для изложения нового материала.

В этой книге я решил рассмотреть ряд модулей и классов и показать, как ими лучше всего пользоваться. Темы выбраны с учетом моих собственных интересов и тех вопросов, которые вызывают наиболее оживленное обсуждение в списке рассылки `qt-interest`. Что-то в эту книгу не вошло, а что-то относится к хорошо знакомой тематике, например архитектуре модель/представление. Во всех случаях я стремился к тому, чтобы изложение было более полным, чем где бы то ни было.

Итак, цель этой книги – помочь программистам, использующим Qt, расширить и углубить свои знания, показав, какие еще задачи можно решать с помощью этой библиотеки. Слово «профессиональный» во многих случаях относится скорее к тому, чего можно добиться, а не к средствам достижения цели. Связано это с тем, что Qt – как всегда – насколько возможно изолирует разработчика от несущественных деталей и нижележащей сложности, предоставляя простые и удобные для использования API. Например, мы увидим, как создать аудиопроигрыватель, ничего не зная о внутренних механизмах его работы; достаточно будет воспользоваться предлагаемым Qt высокоуровневым API. С другой стороны, даже при наличии высокоуровневого модуля *QtConcurrent* многопоточное программирование остается сложным делом.

В этой книге предполагается, что читатель владеет основами программирования на языке C++ и знает хотя бы, как создавать простейшие приложения Qt; например, уже прочитал какую-нибудь хорошую книгу по Qt 4 и приобрел некоторый практический опыт. Также читатель должен быть знаком с организацией справочной документации по Qt и способен, по крайней мере, найти в ней сведения об API интересующих его классов. Кроме того, в некоторых главах предполагаются опре-

¹ Бланшет Ж., Саммерфилд М. «Qt 4: программирование GUI на C++», Кудиц-Пресс, 2007.

деленные знания в конкретной области; например, в главе 1 – знания о языке JavaScript и веб-программировании, а в главах о потоках – знакомство с базовыми концепциями многопоточного программирования и классами потоков в Qt. Такие допущения позволяют не останавливаться на многих деталях и классах, уже знакомых программистам, работающим с Qt (менеджеры компоновки, создание действий, соединение сигналов и слотов и т. д.), и уделить внимание менее известным вещам.

Разумеется, в одной книге невозможно отдать должное всем открытым классам Qt, которых больше 700 (в Qt 4.6 почти 800) и документация по которым содержит свыше миллиона слов. Мы даже пытаться не будем. Вместо этого в книге на примерах объясняется использование некоторых наиболее мощных функций Qt, так что ее можно рассматривать как дополнение к справочной документации, но не ее замену.

Я стремился сделать главы книги по возможности самостоятельными, чтобы не нужно было читать ее последовательно от корки до корки. Если какой-то прием используется в нескольких главах, то объяснение приводится только в одном месте, а из других мест на него даются ссылки. Тем не менее если вы собираетесь читать избранные главы не по порядку, я рекомендую все же сначала бегло просмотреть всю книгу целиком, поскольку главы и разделы, посвященные какой-то одной теме, могут содержать материал, относящийся к другим темам. Кроме того, я старался включать множество мелких деталей, касающихся Qt API, чтобы обогатить содержание книги и показать использование различных средств в контексте, поэтому полезная информация разбросана повсюду.

Как и в предыдущих моих книгах, включенные в текст фрагменты кода – это «живой код», который был автоматически извлечен из исходного текста примеров и вставлен в предоставленный издательству PDF-файл, поэтому никаких ошибок из-за копирования и вставки быть не может, код заведомо работоспособный. Примеры выложены по адресу www.qtrac.eu/aqpbook.html и предоставляются на условиях лицензии GPL (GNU General Public License) версии 3. В книге более 25 примеров, охватывающих свыше 150 *hpp*- и *cpp*-файлов. В совокупности они насчитывают более 20 000 строк кода. Хотя почти все важные части кода оформлены в виде листингов и снабжены пояснениями, многие мелкие детали в книге не обсуждаются, поэтому я рекомендую скачать примеры и почитать исходный код хотя бы тех из них, что относятся к интересующим вас вопросам. Помимо примеров предлагаются также некоторые модули, содержащие часто употребляемую функциональность. Весь их код помещен в пространство имен AQP, чтобы его было проще использовать повторно, а пояснения даются в первых двух главах и далее предполагаются известными.

Все примеры, за исключением приведенных в последней главе, где применяются возможности из версии Qt 4.6, были протестированы в вер-

сиях Qt 4.5 и 4.6 на платформах Linux, Mac OS X и Windows. Приложения, собранные с Qt 4.5, будут работать без изменения с Qt 4.6 и более поздними версиями 4.x, поскольку в рамках одной главной версии Qt гарантируется обратная совместимость. Однако там, где между этими двумя версиями имеются различия, я рассказываю только о подходе, принятом в Qt 4.6, а в исходном коде используется директива `#if QT_VERSION`, чтобы при компиляции для каждой версии выбирались подходящие для нее средства. Некоторые примеры могут работать и с предыдущими версиями Qt 4.x, в частности 4.4, а для других возможен обратный перенос на более ранние версии, но в целом книга ориентирована на Qt 4.5 и 4.6, поэтому об обратном переносе я специально нигде не говорю.

В книге демонстрируются рекомендуемые подходы к работе с Qt 4.6, но, несмотря на многочисленные нововведения, появившиеся в Qt 4.6, на коде это отразилось мало. Одно тривиальное различие состоит в том, что в Qt 4.6 появилась комбинация клавиш для действия «quit» (выйти), которой не было в Qt 4.5; поэтому в исходном коде для Qt 4.6 используется эта комбинация, а для Qt 4.5 с помощью директивы `#if QT_VERSION` включен эквивалентный код. Гораздо важнее другое различие: в Qt 4.6 добавлен класс `QGraphicsObject` и изменилось поведение графических элементов в части уведомления об изменении геометрии. В книге отличия описываются во врезках, а приводимые фрагменты кода относятся к версии Qt 4.6, но в самом исходном коде употребляется директива `#if QT_VERSION` и показано, как добиться одного и того же эффекта в Qt 4.6 и 4.5 (и более ранних версиях), применяя в каждом случае оптимальную технику. В последней главе представлены средства, имеющиеся только в Qt 4.6, причем два из трех примеров являются переработкой показанного ранее кода с использованием появившихся в Qt 4.6 средств анимации и конечных автоматов. Модификация предшествующих примеров позволяет лучше понять, как перейти от традиционного для Qt подхода к использованию новых подсистем.

Следующая версия – Qt 4.7¹ – будет ориентирована на повышение стабильности и скорости работы. Если не считать технологии Qt Quick (которая позволяет декларативно создавать графические интерфейсы пользователя на языке, напоминающем JavaScript), то в ней будет меньше новых возможностей, чем в предыдущих версиях. Так или иначе, несмотря на быстрое и непрекращающееся развитие библиотеки Qt и постоянное расширение спектра ее возможностей, эта книга еще несколько лет будет полезна для изучения наиболее важных технологий, применяемых в версиях серии Qt 4.x, а особенно в Qt 4.5, 4.6 и более поздних.

¹ На момент подготовки русскоязычного издания книги данная версия Qt уже была выпущена и являлась актуальной. – *Прим. науч. ред.*

Благодарности

Прежде всего выражаю благодарность своему другу Трентону Шульцу (Trenton Schulz), бывшему старшему разработчику в подразделении Qt Development Frameworks компании Nokia (ранее Trolltech), который ныне работает научным сотрудником в Норвежском вычислительном центре. Трентон проявил себя как ответственный, вдумчивый и активный рецензент. Его внимательность, высочайшие стандарты качества и многочисленные предложения способствовали значительному улучшению книги.

Благодарю также еще одного своего друга, Жасмина Бланшета, который раньше был старшим разработчиком в подразделении Qt Development Frameworks и моим соавтором по книге «*C++ GUI Programming with Qt 4*», а теперь работает над докторской диссертацией в Мюнхенском техническом университете. Некоторое время назад нам обоим пришла в голову мысль написать эту книгу, и лишь нехватка времени в связи с напряженной работой стала причиной того, что в этой книге Жасмин оказался отличным – и требовательным – рецензентом, а не соавтором.

Хочу также выразить признательность многим людям, работающим (или работавшим) в подразделении Qt Development Frameworks, которые прочли отдельные части этой книги и прислали полезные отзывы, или отвечали на технические вопросы, или то и другое вместе. Это Андреас Ардал Хансен (Andreas Aardal Hanssen) (особенно ценны его отзывы и предложения, касающиеся глав о графических представлениях; он же написал для меня первоначальный текст врезки о внеэкранном рендеринге), Энди Шоу (Andy Shaw), Бьёрн Эрик Нильсен (Bjørn Erik Nilsen), Дэвид Бодди (David Boddie), Хенрик Харц (Henrik Hartz), Кавиндра Дэви Палараджа (Kavindra Devi Palaraja), Райнер Шмид (Rainer Schmid) (теперь работает в компании froglogic), Саймон Хаусман (Simon Hausmann), Тьерри Бастиан (Thierry Bastian) и Фольке Хильшаймер (Volker Hilsheimer).

Итальянская компания www.develer.com любезно предоставила мне бесплатный хостинг репозитория, чтобы я спокойно чувствовал себя в течение всего – довольно долгого – процесса работы над книгой. А некоторые сотрудники этой фирмы прислали полезные замечания, в том числе по поводу некоторых примеров в первых главах. Особенно я благодарен Джанни Валдамбрини (Gianni Valdambrini), Джованни Баджо (Giovanni Bajo), Лоренцо Манчини (Lorenzo Mancini) (который настроил для меня репозиторий) и Томмазо Массими (Tommaso Massimi).

Отдельная благодарность Алексею Смирнову, который прочитал черновик, нашел несколько ошибок и убедил меня добавить поддержку прокси в некоторых примерах, касающихся сетевого программирования.

Хочу также поблагодарить основателей компании froglogic Реджинальда Штадльбауэра (Reginald Stadlbauer) и Харри Портена (Harri Porten) – они предложили мне внештатную работу, которая давала средства к существованию во время работы над книгой, а также ознакомили с некоторыми неизвестными мне программными технологиями. Они же сделали из меня пламенного приверженца своего инструментария для тестирования приложений с графическим интерфейсом, *Squish*.

Заслуживает благодарности и мой друг Бен Томпсон (Ben Thompson) – за то, что напомнил некоторые подзабытые мной математические методы и терпеливо объяснял их снова и снова, пока я не начал понимать.

Эта книга (и ряд других написанных мной) была бы невозможна без библиотеки Qt. Поэтому я благодарен Эрику Чамби-Ингу и Хаварду Норду (Haavard Nord) за создание Qt, а особенно Эрику за то, что, когда я работал в Trolltech, он разрешил мне писать свою первую книгу в рабочее время, а также за то, что любезно согласился написать предисловие к этой книге.

Особая благодарность моему редактору, Дебре Уильямс Коули (Debra Williams Cauley), которая тоже советовала мне написать эту книгу и оказывала практическую помощь во время работы над ней. Еще спасибо Дженнифер Линднер (Jennifer Lindner), которая сделала ряд полезных замечаний о структуре книги и высказала другие мысли, учтенные мной в тексте. Благодарю также Анну Попик (Anna Popick), эффективно руководившую производственным процессом, и корректора Барбару Буд (Barbara Wood), проделавшую отличную работу. И спасибо моей жене Андреа, которая переносила вместе со мной все взлеты и падения во время работы над книгой, – за терпение, любовь и поддержку.

1

Гибридные приложения: рабочий стол + Интернет

- Виджеты с поддержкой Интернета
- Использование WebKit

Повсеместное распространение «облачных вычислений», широкая доступность мобильных телефонов, небольших нетбуков и смартбуков с возможностью выхода в Интернет, не говоря уже о файловом хранилище Google Doc и очень низкой стоимости развертывания веб-приложений, – все это может навести на мысль, будто персональные приложения для рабочего стола – это динозавры, которые еще не знают, что обречены на вымирание.

Но прежде чем откладывать в сторону C++ и Qt и переходить на веб-программирование, отдаваясь изысканным радостям JavaScript и HTML, стоит все же поразмыслить о некоторых достоинствах персональных приложений.

- *Доступность* – если не считать специальных критических областей, можно с уверенностью сказать, что временами (и, как всегда, в самый неудобный момент) Интернет будет недоступен – из-за сбоев в сети, ошибок интернет-провайдера и т. д. А в этом случае веб-приложение оказывается бесполезным¹.
- *Доступ к ресурсам* – персональное приложение имеет полный доступ к компьютеру пользователя, без ограничений, которые прихо-

¹ См., например, opencloudcomputing.info/trends/cloud-computing-downtime или базу данных об отказах облачных вычислений (Cloud Computing Incidents Database).

дится накладывать на веб-приложения для обеспечения безопасности.

- *Внешний облик* – у персонального приложения нет излишних (и только запутывающих) меню и панелей инструментов браузера в дополнение к собственным; оно располагает своими комбинациями клавиш, не вступающими в конфликт с комбинациями, определенными в браузере; внешний вид его интерфейса всегда один и тот же – тот, что был запрограммирован, – и не зависит от конкретного браузера.
- *Специальные виджеты* – персональное приложение может предложить пользователю нестандартные виджеты, разработанные специально под данную задачу, обеспечивая тем самым такой уровень удобства работы, который веб-приложениям и не снился.

В идеале хотелось бы иметь все плюсы персональных приложений и в то же время наслаждаться преимуществами доступа к Интернету, когда он имеется. Благодаря модулю *QtWebKit*, появившемуся в версии Qt 4.4, это стало возможно, потому что *QtWebKit* позволяет создавать гибридные приложения, способные работать как в сети, так и автономно.

Основной недостаток по сравнению с веб-приложениями – это развертывание; приложение Qt должно быть установлено на компьютер пользователя. На случай, когда требуется минимизировать затраты на развертывание или трафик сети, имеется несколько вариантов действий. Например, можно поместить значительную часть функциональности приложения в относительно небольшие подключаемые модули, обновляемые независимо. Или воспользоваться механизмом сценариев и реализовать как можно больше функций на языке JavaScript (ECMAScript) с применением модуля *QtScript* – или стороннего модуля, если вы предпочитаете другой язык сценариев, – и при необходимости обновлять либо добавлять отдельные сценарии. Или перенести как можно больше функций на сервер и веб-страницы, тем самым существенно уменьшив необходимость в обновлении клиента.

В этой главе нас будут интересовать ключевые вопросы, связанные с поддержкой гибридных приложений со стороны Qt. В первом разделе мы применим удобный класс *QNetworkAccessManager*, включенный в Qt 4.4, для создания виджетов с возможностью доступа в Интернет. Во втором разделе мы будем работать с модулем *QtWebKit*, начав с разработки универсального компонента, реализующего функции веб-браузера. Благодаря функциональности, предоставляемой модулем *QtWebKit*, эта задача окажется очень простой. Затем мы воспользуемся только что написанным компонентом, скажем, для создания приложения, обращающегося к конкретному сайту, и с помощью модуля *QtWebKit* получим доступ к объектной модели (DOM) загружаемых незаметно для пользователя страниц, чтобы извлечь из них информацию для дальнейшей обработки. А затем мы увидим, как внедрить виджеты Qt, в том числе разработанные нами, в веб-страницы и тем самым получать возможности, не предоставляемые стандартными элементами управления HTML.

Виджеты с поддержкой Интернета

Мы будем определять виджет с поддержкой Интернета как виджет, который автоматически получает данные из Интернета либо один раз в момент создания, либо через регулярные промежутки времени.

Простейший способ реализовать виджет с поддержкой Интернета заключается в том, чтобы создать подкласс `QWidget`, в котором используется объект `QNetworkAccessManager`. Такие объекты умеют посылать запросы типа HEAD, POST, GET и PUT по протоколу HTTP (или HTTPS), а также поддерживают куки (с помощью объекта типа `QNetworkCookieJar`) и аутентификацию (с помощью `QAuthenticator`).

В этом разделе мы рассмотрим пример использования одного объекта `QNetworkAccessManager` для периодического чтения данных из Интернета, и другого – для скачивания изображений по запросу. Этого достаточно для знакомства с принципами работы с объектами `QNetworkAccessManager`. На рис. 1.1 показано, как объект `QNetworkAccessManager` обращается к внешнему веб-сайту. Отметим, что класс `QNetworkAccessManager` является частью модуля *QtNetwork*, и значит, для любого приложения, в котором этот класс используется, *pro*-файл должен содержать строку `QT += network`.

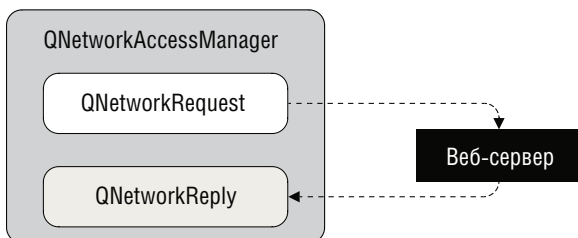


Рис. 1.1. Взаимодействие между `QNetworkAccessManager` и веб-сайтом

В этом разделе рассматривается пример приложения, пиктограмма которого представлена в системном лотке. Обычно такие приложения служат для реализации часто выполняемых задач, например для управления громкостью звука или, скажем, для отображения информации об объеме занятой памяти, текущей даты и времени. Мы же разработаем приложение *Weather Tray Icon* (*weathertrayicon*), пиктограмма которого будет выводить сведения о погоде в указанном аэропорту США, полученные от Национальной метеорологической службы США (*www.weather.gov*).

На снимке экрана в левой части рис. 1.2 показано приложение *Weather Tray Icon* со всплывающей подсказкой – пиктограмма расположена под правым нижним углом окна подсказки. В правой части рисунка показано контекстное меню приложения. Каждый час приложение обновляет данные о погоде для выбранного аэропорта и отображает соответствующую пиктограмму.

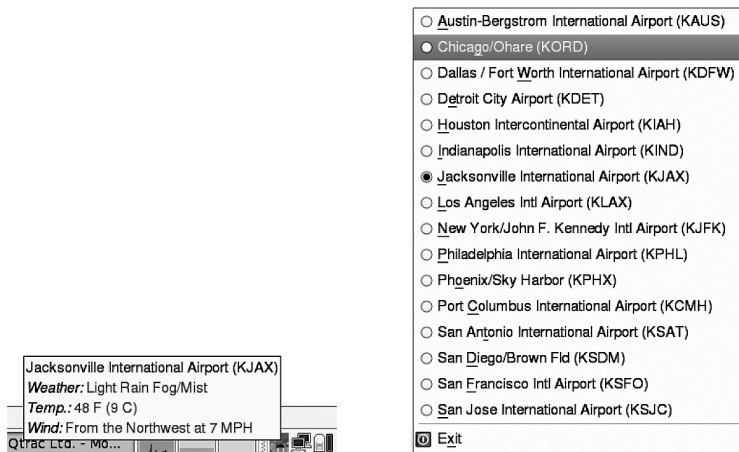


Рис. 1.2. Приложение Weather Tray Icon и его контекстное меню

Приложения, представленные пиктограммами в системном лотке, работают на всех платформах, поддерживаемых Qt. Например, снимки экрана на рис. 1.2 были сделаны в системе Linux с рабочим столом GNOME. В Windows и Mac OS X в окне всплывающей подсказки отображается простой текст, потому что на этих платформах всплывающие подсказки Qt не поддерживают форматированный текст (HTML-разметку). И разумеется, в случае Mac OS X пиктограмма будет присутствовать в строке меню, как и ожидается.

В большинстве примеров в этой книге мы не будем приводить код функции `main()`, потому что он почти всегда простой и стандартный. Однако в данном случае есть два существенных отклонения от нормы, поэтому мы покажем, как выглядит функция `main()` для приложения Weather Tray Icon.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    app.setApplicationName(app.translate("main",
                                         "Weather Tray Icon"));
    app.setOrganizationName("Qtrac Ltd.");
    app.setOrganizationDomain("qtrac.eu");
    app.setQuitOnLastWindowClosed(false);
    if (int error = enableNetworkProxying())
        return error;

    WeatherTrayIcon weatherTrayIcon;
    weatherTrayIcon.show();
    return app.exec();
}
```

Функция начинается, как обычно в Qt, созданием объекта `QApplication`. Мы задаем имя приложения, которое сможем использовать позже, например в заголовках диалоговых окон, обращаясь к функции `QApplication::applicationName()`. Затем задается название организации и домен. Это означает, что в дальнейшем мы сможем создавать объекты `QSettings`, не задумываясь о передаче им аргументов.

У этой функции есть два необычных аспекта. Во-первых, мы говорим Qt, что приложение не следует завершать после закрытия последнего окна. Дело в том, что, как правило, приложение в системном лотке не имеет окна (оно представлено пиктограммой), а все окна, которые оно открывает (к примеру, всплывающая подсказка или контекстное меню), обычно временные, так что их закрытие не должно неявно приводить к завершению приложения.

Во-вторых, мы обращаемся к специальной функции `enableNetworkProxying()`. Она обсуждается во врезке «Поддержка сетевых прокси-серверов» ниже. Если эта функция возвращает ненулевой код, значит, произошла какая-то ошибка, и мы должны завершить приложение.

Поддержка сетевых прокси-серверов

Для машин, подключенных к Интернету напрямую (например, с помощью широкополосного модема или маршрутизатора), приведенные в этой книге примеры сетевых программ работают без каких-либо проблем. Но если сеть защищена брандмауэром (так обычно бывает в корпоративных сетях), то попытка программы выйти в Интернет может завершиться неудачно. В большинстве защищенных брандмауэрами сетей имеется прокси-сервер, через который устанавливаются соединения с Интернетом. Qt поддерживает работу с прокси-серверами, поэтому мы включили соответствующие возможности в примеры `browserwindow`, `nyrbviewer`, `rsspanel` и `weathertrayicon`. Суть приема в том, чтобы из `main()` вызвать специальную функцию `enableNetworkProxying()`.

Функция `enableNetworkProxying()` вызывает функцию `AQP::OptionParser` (имеется в примерах, прилагаемых к этой книге, в каталоге `option_parser`, обратите внимание на пространство имен `AQP`), чтобы разобрать заданные в командной строке аргументы для настройки прокси-сервера. Приложения с поддержкой прокси понимают следующие аргументы командной строки:

<code>-h --help</code>	вывести эту справку и завершиться
<code>-H --host=STRING</code>	имя хоста, например <code>www.example.com</code>
<code>-P --password=STRING</code>	пароль
<code>-p --port=INTEGER</code>	номер порта, например <code>1080</code>

```
-t --type=STRING      (http, socks5; по умолчанию socks5) тип прокси
-u --username=STRING имя пользователя
```

Доступ к прокси-серверу активируется только в том случае, когда задано имя хоста. Вот как выглядит код; переменная `parser` имеет тип `AQP::OptionParser`.

```
if (parser.hasValue("host")) {
    QNetworkProxy proxy;
    proxy.setType(parser.string("type") == "socks5"
        ? QNetworkProxy::Socks5Proxy
        : QNetworkProxy::HttpProxy);
    proxy.setHostName(parser.string("host"));
    if (parser.hasValue("port"))
        proxy.setPort(parser.integer("port"));
    if (parser.hasValue("username"))
        proxy.setUser(parser.string("username"));
    if (parser.hasValue("password"))
        proxy.setPassword(parser.string("password"));
    QNetworkProxy::setApplicationProxy(proxy);
}
```

Если аргумент `host` задан, то при настройке прокси-сервера устанавливается этот хост, заданный или подразумеваемый по умолчанию тип прокси и прочие параметры, указанные пользователем. Тем самым доступ к сети через прокси-сервер настраивается глобально для всего приложения. Можно также организовать доступ через прокси для одного сокета, воспользовавшись функцией `QAbstractSocket::setProxy()`.

Данные о погоде можно представлять в различных форматах, но мы решили остановиться на формате XML. Сам по себе формат простой, данные представлены в виде пар ключ–значение, где ключом является имя тега, а значением — текст между открывающим и закрывающим тегом. Например:

```
<weather>Fair</weather>
<temperature_string>49 F (9 C)</temperature_string>
<temp_f>49</temp_f>
<temp_c>9</temp_c>
<wind_string>From the Northeast at 5 MPH</wind_string>
<visibility_mi>9.00</visibility_mi>
<icon_url_base>http://weather.gov/weather/images/fcicons/</icon_url_base>
<icon_url_name>nskc.jpg</icon_url_name>
```

Сразу после запуска приложение выбирает аэропорт: тот, что был ранее установлен пользователем, или — при первом запуске — подразумеваемый по умолчанию. Затем для получения данных о погоде применяется

объект `QNetworkAccessManager`. Двумя элементами этих данных являются URL и имя файла пиктограммы, представляющей погодные условия в аэропорту. Для получения пиктограммы, отображаемой в системном лотке, приложение пользуется вторым объектом `QNetworkAccessManager`. Впрочем, как мы вскоре увидим, для экономии трафика приложение кэширует ранее полученные пиктограммы.

```
class WeatherTrayIcon : public QSystemTrayIcon
{
    Q_OBJECT
public:
    explicit WeatherTrayIcon();

private slots:
    void requestXml();
    void readXml(QNetworkReply *reply);
    void readIcon(QNetworkReply *reply);
    void setAirport(QAction *action);

private:
    ...
    QMenu menu;
    QNetworkAccessManager *networkXmlAccess;
    QNetworkAccessManager *networkIconAccess;
    QString airport;
    QCache<QUrl, QIcon> iconCache;
    int retryDelaySec;
};
```

Чуть ниже мы поясним все используемые здесь методы, включая и не показанные закрытые, но сначала поговорим о некоторых закрытых данных-членах. В строке `airport` хранится название текущего аэропорта, например «Chicago/Ohare (KORD)». В кэше `iconCache` находятся ключи типа `QUrl` и *указатели* на значения типа `QIcon`. Остальные данные-члены мы рассмотрим, когда будем обсуждать методы, в которых они используются.

Принцип кэширования в классе `QCache` основан на понятии «стоимости». По умолчанию максимальная стоимость кэша равна 100; это означает, что сумма стоимостей отдельных элементов всегда меньше или равна этому максимуму. По умолчанию стоимость каждого элемента равна 1, поэтому если мы не изменим величину максимума или не зададим стоимости элементов самостоятельно, то в кэше в любой момент будет находиться не более 100 элементов. Если при добавлении нового элемента суммарная стоимость всех элементов превысит максимально допустимую, то из кэша удаляются один или несколько элементов, к которым дольше всего не было обращений, пока суммарная стоимость оставшихся не окажется меньше или равна максимуму.

«За кулисами» для обеспечения очень быстрого поиска по ключу в `QCache` применяется класс `QHash`. Однако без дополнительных действий с нашей

стороны `QHash` не может использовать в качестве ключей объекты типа `QUrl`, так как в `Qt` отсутствует функция `qHash(QUrl)`¹. Впрочем, это легко поправить, написав такую функцию-однострочник:

```
inline uint qHash(const QUrl &url) {return qHash(url.toString());}
```

Здесь мы просто перепоручаем работу встроенной функции `qHash(QString)`. Теперь мы готовы приступить к рассмотрению методов и начнем с конструктора.

```
WeatherTrayIcon::WeatherTrayIcon()
    : QSystemTrayIcon(), retryDelaySec(1)
{
    setIcon(QIcon(":/rss.png"));
    createContextMenu();

    networkXmlAccess = new QNetworkAccessManager(this);
    networkIconAccess = new QNetworkAccessManager(this);
    connect(networkXmlAccess, SIGNAL(finished(QNetworkReply*)),
            this, SLOT(readXml(QNetworkReply*)));
    connect(networkIconAccess, SIGNAL(finished(QNetworkReply*)),
            this, SLOT(readIcon(QNetworkReply*)));

    QTimer::singleShot(0, this, SLOT(requestXml()));
}
```

Мы связываем с приложением начальную пиктограмму, которая будет отображаться, пока не загрузится первая пиктограмма погоды. Затем мы создаем контекстное меню с опциями для смены аэропорта и завершения приложения.

Большая часть конструктора посвящена настройке доступа к Интернету, точнее, созданию двух объектов `QNetworkAccessManagers`. Один используется для загрузки данных о погоде, другой – для получения пиктограммы, ассоциированной с текущими погодными условиями. Два менеджера доступа к сети применяются для того, чтобы они могли работать независимо друг от друга. В обоих случаях мы создаем только одно соединение между сигналом и слотом, так как интересует нас лишь момент окончания загрузки.

Наконец мы вызываем слот `requestXml()` с помощью однократного таймера. Этот метод пользуется менеджером `networkXmlAccess` для получения данных о погоде в текущем аэропорту.

Можно было бы вызвать `requestXml()` напрямую, но ради единства стиля мы предпочитаем вызывать методы «создания», которые вносят вклад в конструирование объекта, в конструкторах, а все методы инициализации, логически не относящиеся к конструированию, – через однократ-

¹ Функция `qHash(QUrl)` появилась в `Qt 4.7`.

ный таймер. Тем самым гарантируется, что к моменту вызова метода инициализации объект уже полностью сконструирован. Поэтому метод инициализации может обращаться к любой переменной-члену и к любому методу, что на этапе конструирования может быть небезопасно.

Прежде чем переходить к слоту `requestXml()`, давайте посмотрим, как создается контекстное меню. Это поможет понять, как пользователь задает аэропорт, данные о котором нужно загрузить.

```
void WeatherTrayIcon::createContextMenu()
{
    QStringList airports;
    airports << "Austin-Bergstrom International Airport (KAUS)"
        ...
        << "San Jose International Airport (KSJC)";
    QSettings settings;
    airport = settings.value("airport", QVariant(airports.at(0)))
        .toString();

    QActionGroup *group = new QActionGroup(this);
    foreach (const QString &anAirport, airports) {
        QAction *action = menu.addAction(anAirport);
        group->addAction(action);
        action->setCheckable(true);
        action->setChecked(anAirport == airport);
        action->setData(anAirport);
    }
    connect(group, SIGNAL(triggered(QAction*)),
        this, SLOT(setAirport(QAction*)));
    menu.addSeparator();
    menu.addAction(QIcon(":/exit.png"), tr("E&xit"), qApp,
        SLOT(quit()));
    AQP::accelerateMenu(&menu);
    setContextMenu(&menu);
}
```

Здесь список аэропортов «зашит» в код, но можно было бы без труда прочитать их из файла или из ресурса. (Если бы мы захотели включить в список все аэропорты США, то можно было бы их сгруппировать, например, включив штаты в качестве пунктов меню верхнего уровня, а аэропорты – как подменю.) Для задания текущего аэропорта мы пользуемся классом `QSettings`, а при первом запуске приложения делаем текущим первый аэропорт в списке.

Для каждого аэропорта создается объект `QAction`, и тот из них, который соответствует текущему аэропорту, помечается как выбранный. Действия, ассоциированные с аэропортами, включаются в группу `QActionGroup`. По умолчанию свойство `exclusive` в объекте `QActionGroup` равно `true`, поэтому действия представляются в виде набора переключателей, а не флажков. В результате в любой момент времени отмеченным может быть только один аэропорт.

Кроме того, мы добавляем действие для выхода из программы и ассоциируем с ним клавишу-акселератор. Затем вызываем функцию `AQP::accelerateMenu()`, которая назначит акселераторы максимально возможному количеству аэропортов, и напоследок устанавливаем только что созданное меню в качестве контекстного меню приложения. Если приложение собирается для Mac OS X, то обращение к `AQP::accelerateMenu()` обычно не дает никакого эффекта, потому что Mac OS X не поддерживает акселераторы. Дополнительные сведения об автоматическом задании клавиш-акселераторов см. во врезке «Клавиши-акселераторы» ниже.

Нам необходимо соединить каждое действие со слотом `setAirport()` и параметризовать вызовы слота, чтобы тот мог понять, какой аэропорт выбран. Проще всего добиться этого путем вызова метода `QObject::sender()` внутри слота, чтобы узнать, из какого действия он был вызван, а затем получить от действия текст и тем самым определить аэропорт. Альтернативный способ – воспользоваться классом `QSignalMapper`. Но в данном случае есть даже более простое решение – вместо того чтобы соединять со слотом действия `QAction` каждого аэропорта, мы можем соединить объект `QActionGroup`. Сигнал `QActionGroup::triggered()` передает выбранный объект `QAction` в качестве параметра.

```
void WeatherTrayIcon::requestXml()
{
    QString airportId = airport.right(6);
    if (airportId.startsWith("(") && airportId.endsWith(")")) {
        QString url = QString("http://www.weather.gov/xml/"
                               "current_obs/%1.xml").arg(airportId.mid(1, 4));
        networkXmlAccess->get(QNetworkRequest(QUrl(url)));
    }
}
```

XML-данные о погоде в конкретном аэропорту хранятся в файле, имя которого определяется четырехбуквенным кодом аэропорта. Мы поместили этот код в конце названия аэропорта, заключив его в скобки. Теперь для его извлечения мы пользуемся методами класса `QString`. Построив URL, мы должны обратиться к нему с GET-запросом, для чего воспользуемся менеджером доступа к сети. Метод отправки запроса возвращает указатель на объект типа `QNetworkReply`, который служит для получения результатов от сервера. Объекты этого типа генерируют сигналы, индицирующие этапы взаимодействия с сервером, например `downloadProgress()` и `uploadProgress()`. А по завершении запроса объект, представляющий ответ, генерирует сигнал `finished()`.

Менеджер доступа к сети, инициировавший запрос, также генерирует сигнал `finished()`, а поскольку отслеживать ход взаимодействия мы не собираемся, то только к этому сигналу и подключаемся. Именно поэтому мы и проигнорировали значение, возвращенное методом `QNetworkAccessManager::get()`. После того как загрузка данных завершит-

ся (успешно или нет), установленное соединение между сигналом и слотом приведет к вызову метода `readXml()`, которому будет передан единственный аргумент – указатель на объект-ответ:

```
void WeatherTrayIcon::readXml(QNetworkReply *reply)
{
    if (reply->error() != QNetworkReply::NoError) {
        setToolTip(tr("Failed to retrieve weather data:\n%1")
            .arg(reply->errorString()));
        QTimer::singleShot(retryDelaySec * 1000,
            this, SLOT(requestXml()));
        retryDelaySec <= 1;
        if (retryDelaySec > 60 * 60)
            retryDelaySec = 1;
        return;
    }
    retryDelaySec = 1;
    QDomDocument document;
    if (document.setContent(reply))
        populateToolTip(&document);
    QTimer::singleShot(60 * 60 * 1000, this, SLOT(requestXml()));
}
```

Если запрос завершился неудачно, мы помещаем во всплывающую подсказку пиктограммы сообщение об ошибке и через некоторое время повторяем попытку. Величина интервала между попытками хранится в закрытой переменной `retryDelaySec` и инициализируется значением 1. (Необходимо умножить это значение на 1000, поскольку методу `QTimer::singleShot()` передается тайм-аут, выраженный в миллисекундах.) После каждой неудачной попытки мы удваиваем интервал (сдвиг целого числа влево на один бит как раз и означает удвоение). Так, после второй подряд неудачной попытки мы будем ждать две секунды, затем четыре секунды и т. д. При таком алгоритме после дюжины попыток интервал превысит час, и в этот момент мы сбрасываем величину интервала назад в 1.

Интервал между попытками увеличивается для того, чтобы наши запросы по ошибке не приняли за атаку типа «отказ в обслуживании», а заодно чтобы не тратить зря время на неудачные запросы.

Как только запрос завершается успешно, мы сбрасываем переменную `retryDelaySec` в 1 и приступаем к разбору XML-данных. Так как `QNetworkReply` – подкласс `QIODevice`, то, помимо сигналов о ходе взаимодействия с сервером, он может, например, генерировать сигнал `readyRead()`. Кроме того, из любого объекта типа `QIODevice` (например, файла) можно читать данные, которые по умолчанию возвращаются в виде массива байтов `QByteArray`. А поскольку метод `QDomDocument::setContent()` умеет получать XML-данные из объектов `QByteArray`, `QString` и `QIODevice`, то мы можем сразу передать ему объект `QNetworkReply` для разбора.

Клавиши-акселераторы (для всех платформ, кроме Mac OS X)

Клавиши-акселераторы важны для тех пользователей, которые быстро печатают и не могут или не хотят пользоваться мышью. Так называются комбинации клавиш вида Alt+*x* (где *x* обычно буква или цифра), как правило, они служат для выбора пункта главного меню (например, Alt+F часто ассоциируется с пунктом File). Из появившегося меню конкретный пункт можно выбрать путем нажатия подчеркнутой в его названии буквы или цифры; так, для создания нового файла надо последовательно нажать Alt+F, N.

В диалоговых окнах акселераторы применяются для передачи фокуса конкретному виджету; например, если с меткой Total: ассоциирован акселератор Alt+T, то при его нажатии в фокусе окажется виджет, связанный с этой меткой (buddy). Акселераторы также часто назначаются флажкам и переключателям, в этом случае одновременное нажатие соответствующих клавиш изменяет состояние флажка или переключателя.

И в коде, и на экране принято записывать акселераторы заглавными буквами (например, Alt+E). (Клавиши-акселераторы отличаются от горячих клавиш, под которыми понимаются произвольные комбинации клавиш, ассоциированные с действиями; например, Ctrl+N для создания нового файла.)

Для коротких меню или диалоговых окон, содержащих всего несколько виджетов, акселераторы легко задать вручную (включив символ & в текст). Но когда количество пунктов или виджетов превышает 15, выбрать оптимальные акселераторы становится тяжело. В идеале хотелось бы поручить задачу назначения акселераторов компьютеру, именно так мы и поступаем в этой книге.

В каталоге aqr имеется модуль alt_key.{hpp,cpp}, предоставляющий API, и модуль kuhn_munkres.{hpp,cpp}, в котором реализован алгоритм быстрого вычисления оптимальных назначений, намного превосходящий различные наивные алгоритмы¹. Например, чтобы выбрать акселераторы для всех меню главного окна, мы можем поместить в конце его конструктора такую строку:

```
AQP::accelerateMenu(menuBar());
```

И больше ничего не нужно. Алгоритм Куна–Манкреса вычисляет оптимальные акселераторы, учитывая те элементы, которым акселераторы уже назначены (например, бывает, что мы хотим

¹ Полную документацию по нему можно найти по адресу http://qtrac.eu/alt_key_api.html – Прим. науч. ред..

назначить некоторым пунктам меню или меткам вполне определенные акселераторы самостоятельно). Для диалоговых окон применяется столь же простой подход, в конце конструктора окна пишется такая строка:

```
AQP::accelerateWidget(this);
```

Функции назначения акселераторов не учитывают скрытые виджеты, например расположенные на вкладке, которая не показывается. Но можно обработать и этот случай, достаточно один или несколько раз вызвать функцию `AQP::accelerateWidgets()`, каждый раз передавая ей список интересующих нас виджетов.

Если метод `QDomDocument::setContent()` возвращает `true`, значит, разбор завершился удачно, и мы можем записать во всплывающую подсказку новые данные, причем по ходу дела может понадобиться загрузить новую пиктограмму. Если же разобрать XML-данные не удалось, то содержание подсказки не изменяется. В конце устанавливается однократный таймер, который повторит всю процедуру через час.

Класс `QDomDocument` является частью модуля *QtXml*, поэтому для его использования необходимо включить в *pro*-файл проекта строку `QT += xml`.

На рис. 1.3 показана блок-схема данного приложения. Выполнение начинается с GET-запроса к менеджеру доступа к сети для получения XML-данных, затем загружаются сами данные и пиктограммы, и весь процесс повторяется каждый час.

Для полноты картины приведем закрытый метод `populateToolTip()`, который мы разобьем на три части, а также два вызываемых из него закрытых вспомогательных метода.

```
void WeatherTrayIcon::populateToolTip(QDomDocument *document)
{
    QString toolTipText = tr("<font color=darkblue>%1</font><br>")
                          .arg(airport);
    QString weather = textForTag("weather", document);
    if (!weather.isEmpty())
        toolTipText += toolTipField("Weather", "green", weather);
    ...
}
```

Здесь мы извлекаем те текстовые элементы, которые хотим включать в состав всплывающей подсказки. Большая часть кода опущена, поскольку мало чем отличается от обработки элемента `weather`.

Закрытый вспомогательный метод `textForTag()` извлекает текст, обрамленный указанным тегом. Этот подход работает, потому что мы точно знаем, что все теги в данных о погоде различны и вложенных тегов не существует.

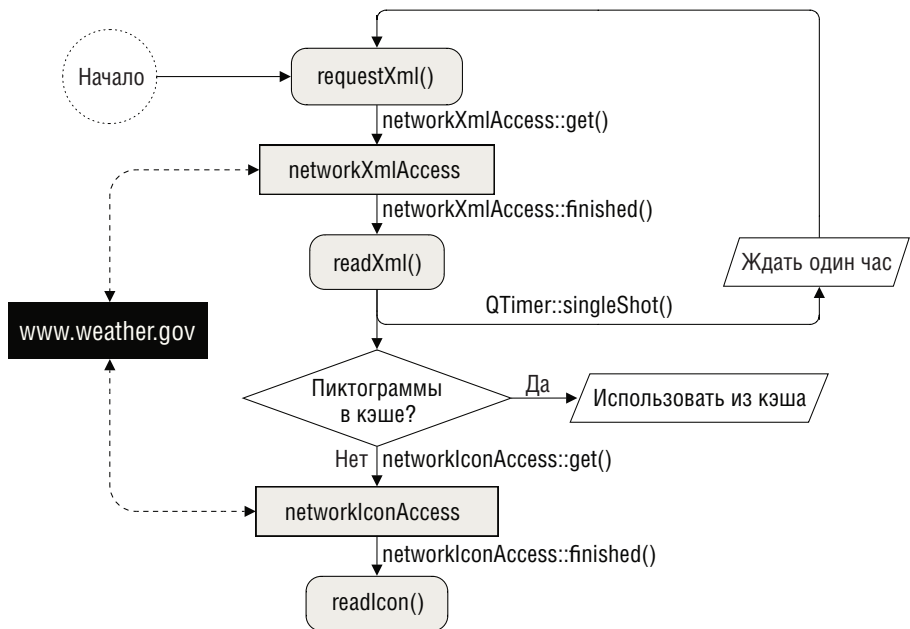


Рис. 1.3. Блок-схема приложения Weather Tray Icon

```

QString iconUrl = textForTag("icon_url_base", document);
if (!iconUrl.isEmpty()) {
    QString name = textForTag("icon_url_name", document);
    if (!name.isEmpty()) {
        iconUrl += name;
        QUrl url(iconUrl);
        QIcon *icon = iconCache.object(url);
        if (icon && !icon->isNull())
            setIcon(*icon);
        else
            networkIconAccess->get(QNetworkRequest(url));
    }
}
}

```

Пиктограмма, соответствующая текущим погодным условиям, идентифицируется двумя XML-элементами, `icon_url_base` и `icon_url_name`. Мы пытаемся получить тексты обоих элементов и составить из них URL-адрес. Затем мы пробуем найти в кэше пиктограмму по ключу, совпадающему с этим URL. Метод `QCache::object()` возвращает 0, если в кэше нет элемента с указанным ключом.

Если в кэше уже имеется нужный нам указатель `QIcon`, то его и используем. На самом деле мы получаем копию (это дешево, потому что в Qt применяется схема копирования при записи, и полезно, так как `QCache`

в любой момент может удалить хранящиеся в кэше элементы). В противном случае загружаем пиктограмму с помощью менеджера доступа к сети. И если была запущена процедура загрузки пиктограммы, то установленное ранее соединение сигнала со слотом гарантирует вызов слота `readIcon()` (мы рассмотрим его ниже) по завершении загрузки.

```
#ifndef Q_WS_X11
    toolTipText = QTextDocumentFragment::fromHtml(toolTipText).toPlainText();
#endif
    setToolTip(toolTipText);
}
```

К сожалению, Qt поддерживает форматированный текст (HTML-разметку) во всплывающих подсказках только на платформе X11, поэтому для Windows и Mac OS X приходится преобразовывать HTML-код подсказки в обычный текст. Для этого мы пользуемся статическим методом `QTextDocumentFragment::fromHtml()`, чтобы получить объект `QTextDocumentFragment`, а затем — методом `QTextDocumentFragment::toPlainText()` для получения простого текста. Пользоваться методом `QTextDocumentFragment` удобнее, чем преобразовывать разметку вручную, потому что он не только корректно преобразует HTML-компоненты в соответствующие символы Unicode и убирает теги HTML, но и заменяет теги `
` символами новой строки.

```
QString WeatherTrayIcon::textForTag(const QString &tag,
                                     QDomDocument *document)
{
    QDomNodeList nodes = document->elementsByTagName(tag);
    if (!nodes.isEmpty()) {
        const QDomNode &node = nodes.item(0);
        if (!node.isNull() && node.hasChildNodes())
            return node.firstChild().nodeValue();
    }
    return QString();
}
```

Применение класса `QDomDocument` идеально для небольших XML-файлов, поскольку он разбирает файл целиком, сохраняет все данные в памяти и предоставляет ряд полезных методов доступа.

Мы начинаем с получения списка всех узлов `QDomNodes`, в которых используется указанный тег. Если список не пуст, берем первый узел. В данных о погоде все теги уникальны, поэтому для любого тега должно найтись не более одного узла. В DOM API текст, расположенный между открывающим и закрывающим тегом, представлен дочерним узлом, поэтому для его получения нужно взять первого потомка узла, преобразовать его в текстовый узел, а затем извлечь текстовые данные, вот так: `node.firstChild().toText().data()`. Но Qt предлагает и более короткий путь — метод `QDomNode::nodeValue()`, который возвращает строку, зависящую от типа узла. В случае текстового узла это будет сам текст.


```

QString WeatherTrayIcon::toolTipField(const QString &name,
    const QString &htmlColor, const QString &value, bool appendBr)
{
    return QString("<i>%1:</i>&nbsp;<font color=\"%2\">%3</font>%4")
        .arg(name).arg(htmlColor).arg(value)
        .arg(appendBr ? "<br>" : "");
}

```

Закрытый вспомогательный метод `toolTipField()` позволяет вынести форматирование отдельных строк текста всплывающей подсказки. Это немного сокращает и упрощает код метода `populateToolTip()`. Кроме того, код форматирования будет проще модифицировать, если возникнет такая необходимость.

Если предстоит загружать пиктограмму, то сразу после готовности объекта `QNetworkReply` будет вызван слот `readIcon()` (поскольку мы ранее уже настроили его подключение к сигналу). Рассмотрение кода этого слота разобьем на две части.

```

void WeatherTrayIcon::readIcon(QNetworkReply *reply)
{
    QUrl redirect = reply->attribute(
        QNetworkRequest::RedirectionTargetAttribute).toUrl();
    if (redirect.isValid())
        networkIconAccess->get(QNetworkRequest(redirect));
}

```

Этот метод вызывается по завершении запроса на загрузку пиктограммы. Для начала мы проверяем, не получили ли вместо ожидаемого ответа перенаправление на какой-нибудь другой адрес. Если так, то отправляем новый GET-запрос для загрузки пиктограммы с указанного в перенаправлении URL-адреса. По соображениям безопасности объект `QNetworkAccessManager` не выполняет перенаправление автоматически, но в данном случае мы решили, что сайту можно доверять. Если же безопасность под вопросом, то следовало бы проверить новый URL, например убедиться, что он принадлежит тому же домену, что и исходный, и не ведет на какой-нибудь вредоносный JavaScript-сценарий. В типичном случае, когда никакого перенаправления нет, вызов `redirect.isValid()` вернет `false`, и можно приступать к чтению полученных данных.

```

else {
    QByteArray ba(reply->readAll());
    QPixmap pixmap;
    if (pixmap.loadFromData(ba)) {
        QIcon *icon = new QIcon(pixmap);
        setIcon(*icon);
        iconCache.insert(reply->request().url(), icon);
    }
}
}
}

```

Если ответ не подразумевает перенаправления, то либо мы получили данные пиктограммы, либо произошла ошибка. Мы считываем все при-

шедшие данные в массив `QByteArray`, после чего загружаем их в объект `QPixmap`. Если метод `QPixmap::loadFromData()` возвращает `false`, то данные либо неполны, либо повреждены, либо представлены в неизвестном формате, либо имела место ошибка и данных вообще нет. В любом случае мы отказываемся от попытки получить пиктограмму и оставляем текущую пиктограмму без изменения.

Если загрузка прошла успешно, то объект `QPixmap` преобразуется в `QIcon` и результат становится новой пиктограммой, отображаемой в системном лотке. Затем эта пиктограмма помещается в кэш с ключом, равным его URL-адресу, причем мы точно знаем, что в кэше никогда не окажется более 100 пиктограмм и что необходимости загружать уже присутствующую в кэше пиктограмму не возникнет.

```
void WeatherTrayIcon::setAirport(QAction *action)
{
    airport = action->data().toString();
    QSettings settings;
    settings.setValue("airport", airport);
    requestXml();
}
```

Этот слот вызывается, когда пользователь выбирает из контекстного меню новый аэропорт. Мы получаем название аэропорта, делаем его текущим, а затем вызываем метод `requestXml()`, чтобы скачать данные о погоде для вновь выбранного аэропорта.

Отметим, что при завершении приложения мы не сохраняем текущие параметры (в данном случае всего один); сохранение происходит в момент изменения параметров. У такого подхода есть то преимущество, что сохраненные параметры всегда актуальны, даже во время работы приложения или в случае аварийного завершения. Однако имеется и недостаток – код сохранения параметров может оказаться размазанным по всему приложению, что усложняет его сопровождение.

Мы закончили рассмотрение небольшого приложения, в котором используется удобный высокоуровневый класс `QNetworkAccessManager`, позволяющий выполнить простые операции загрузки данных из Интернета. В дополнение к приложению `Weather Tray Icon` в состав примеров к данной книге входит приложение `RssPanel` (`rsspanel`), показанное на рис. 1.4.

В приложении `RssPanel` демонстрируется виджет с поддержкой Интернета `RssComboBox`, который автоматически отображает данные из RSS-ленты (XML-документа), находящейся по указанному URL-адресу¹,

¹ Занятно, что, в отличие от `weathertrayicon`, приложение `rsspanel` не обрабатывает перенаправление, и поэтому попытка скачать RSS-ленту, установленную в нем по умолчанию, завершается с ошибкой (обусловленной изменениями, произошедшими в сервисе BBC). Реализация поддержки перенаправления в `rsspanel` остается читателю русскоязычного издания в качестве упражнения. – *Прим. науч. ред.*



Рис. 1.4. Приложение RssPanel

и периодически обновляет себя. Мы не станем рассматривать код этого примера, потому что структурно он очень похож на приложение Weather Tray Icon. Однако в качестве отправной точки для создания собственных виджетов с поддержкой Интернета он может показаться даже проще. Кроме того, для разбора полученных RSS-данных здесь применяется класс `QXmlStreamReader`, а не `QDomDocument`, как в примере `weathertrayicon`¹.

В оставшейся части этой главы мы продолжим рассмотрение вопроса о получении данных из Интернета, но для отображения этих данных (то есть HTML-страниц) и выполнения операций над ними будем использовать компонент WebKit.

Использование WebKit

WebKit – это компонент с открытым исходным кодом, предназначенный для отображения и редактирования веб-контента. Его первая версия была написана разработчиками среды KDE ('K' Desktop Environment). Сейчас WebKit лежит в основе многих браузеров, в том числе Google Chrome, KDE Konqueror и Mac OS X Safari, а также применяется в большинстве мобильных устройств с возможностью выхода в Интернет. WebKit стремится соответствовать имеющимся стандартам и поддерживает все стандартные веб-технологии, включая HTML5, SVG (Scalable Vector Graphics), CSS (Cascading Style Sheets, в том числе CSS 3 Web Fonts) и JavaScript. Входящий в состав Qt модуль `QtWebKit` предлагает интерфейс к WebKit в стиле Qt и предоставляет в распоряжение работающих с Qt программистов всю функциональность WebKit, а также немало дополнительных возможностей. Для использования этого модуля необходимо включить в *pro*-файл приложения строку `QT += webkit`.

Наиболее важные из входящих в состав модуля `QtWebKit` классов показаны в табл. 1.1, а отношения между некоторыми из них – на рис. 1.5.

¹ Не исключено, что в будущем от модуля `QtXml`, предоставляющего поддержку синтаксических анализаторов DOM и SAX, откажутся в пользу гораздо более быстрых классов `QXmlStreamReader` и `QXmlStreamWriter`, являющихся частью модуля `QtCore`.

Таблица 1.1. Основные классы, входящие в состав WebKit

Класс	Описание
QWebElement	Класс для доступа и редактирования элементов DOM, входящих в QWebFrame, с помощью API, аналогичного имеющемуся в jQuery (Qt 4.6)
QWebFrame	Объект, представляющий один фрейм внутри веб-страницы
QWebHistory	История посещенных ссылок, ассоциированная с объектом QWebPage
QWebHistoryItem	Объект, представляющий одну посещенную ссылку из QWebHistory
QWebPage	Объект, представляющий веб-страницу
QWebSettings	Объект, в котором хранятся параметры, используемые в данном объекте QWebFrame или QWebPage
QWebView	Виджет для визуализации QWebPage

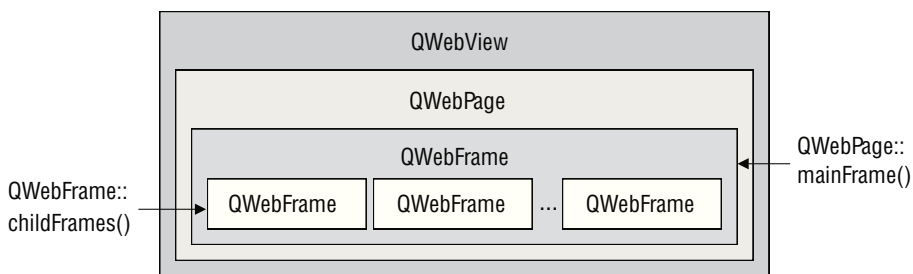


Рис. 1.5. Некоторые классы QtWebKit в контексте

Отметим, что в модуле *QtWebKit* есть всего один виджет¹ — *QWebView*. А отображаемая в нем страница *QWebPage* и один или несколько фреймов *QWebFrame* — это классы данных. С помощью класса *QWebPage* можно загрузить из сети веб-контент, обработать его и представить результат в пользовательском интерфейсе по собственному усмотрению — нечто подобное мы придумаем во втором подразделе этого раздела.

Теперь, получив представление о том, что предлагает модуль *QtWebKit*, рассмотрим некоторые примеры его использования. В первом подразделе мы создадим компонент, содержащий окно веб-браузера, во втором — приложение для работы с конкретным сайтом, которое незаметно для пользователя будет читать и обрабатывать его страницы. А в третьем

¹ Строго говоря, упомянутый ниже *QWebInspector* также наследует *QWidget*, но, конечно, он не является средством для отображения веб-страниц. — Прим. науч. ред.

подразделе покажем, как можно внедрить готовые и написанные пользователем виджеты Qt в веб-страницу.

Универсальный компонент, содержащий окно веб-браузера

В следующих двух подразделах мы воспользуемся модулем WebKit для разработки двух гибридных приложений. А сейчас создадим компонент, содержащий окно браузера (`browserwindow`) (рис. 1.6), который пригодится нам в дальнейших примерах.

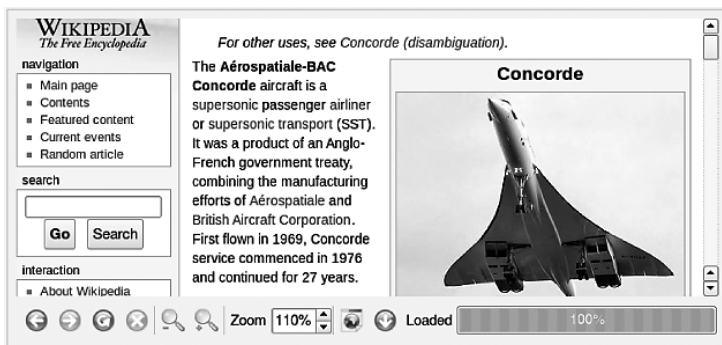


Рис. 1.6. Компонент, содержащий окно веб-браузера

Этот компонент поддерживает стандартные функции браузера: перемещение вперед и назад, обновление, отмена загрузки, изменение масштаба, открытие указанной страницы и возможность возврата к любой странице, хранящейся в истории просмотра. У него также есть контекстное меню и панель инструментов (которые можно скрыть).

Помимо этого, если определен символ `DEBUG` (например, с помощью строки `DEFINES += DEBUG` в *pro*-файле), то в контекстном меню браузера появляется дополнительный пункт `Inspect`, при выборе которого запускается инспектор WebKit, показанный на рис. 1.7. Это очень полезный инструмент отладки, предоставляющий разнообразную информацию о веб-странице, в частности о ее объектной модели (DOM) и используемых ресурсах (к примеру, таблицах стилей, графических изображениях и JavaScript-сценариях) с указанием размеров и времени загрузки. И это еще далеко не все.

Начиная с версии Qt 4.6 веб-инспектор можно вызывать более традиционным способом: создать объект `QWebInspector`, передав ему объект `QWebPage`, а затем обратиться к его методу `show()`.

Чтобы получить доступ к веб-инспектору (и чтобы в версии Qt 4.6 заработал класс `QWebInspector`), необходимо включить параметр `QWebSettings`

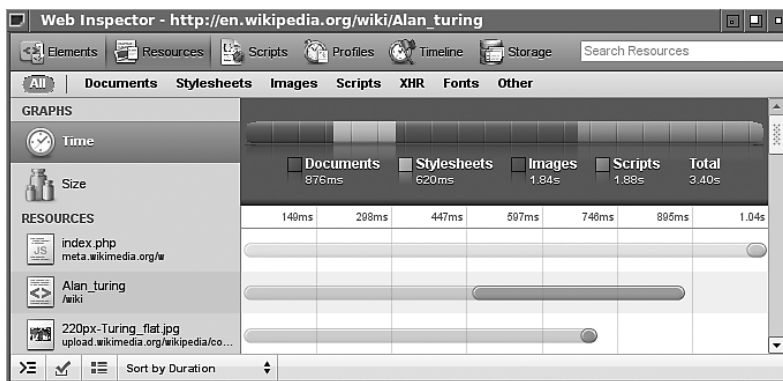


Рис. 1.7. Веб-инспектор

::DeveloperExtrasEnabled. Этот и другие параметры включаются в функции `main()`, для чего следует воспользоваться методами глобального объекта `QWebSettings`, как показано в следующем фрагменте:

```
QWebSettings *webSettings = QWebSettings::globalSettings();
webSettings->setAttribute(QWebSettings::AutoLoadImages, true);
webSettings->setAttribute(QWebSettings::JavascriptEnabled,
                        true);
webSettings->setAttribute(QWebSettings::PluginsEnabled, true);
webSettings->setAttribute(QWebSettings::ZoomTextOnly, true);
#ifdef DEBUG
webSettings->setAttribute(QWebSettings::DeveloperExtrasEnabled,
                        true);
#endif
```

Параметры, установленные в глобальном объекте `QWebSettings`, наследуются всеми объектами `QWebPage` и `QWebView`, созданными в приложении, хотя при желании для отдельных объектов их можно переопределить. Атрибут `QWebSettings::ZoomTextOnly` появился в версии Qt 4.5 и влияет на коэффициент масштабирования. Если этот атрибут равен `true`, то изображения не будут масштабироваться (растровые изображения при масштабировании искажаются), а увеличиваться и уменьшаться будет только текст.

Когда готовилась к печати эта книга, в документации по Qt не были описаны параметры сетевых настроек, подразумеваемые по умолчанию, поэтому на разных платформах и в более старых версиях серии 4.x они могут различаться. Однако мы всегда можем проверить значение любого параметра, вызвав метод `QWebSettings::testAttribute()`, который принимает аргумент перечисляемого типа, определяющий атрибут, и возвращает значение типа `bool`.

Чтобы получить представление об API окна браузера, рассмотрим открытую и защищенную секции определения класса в заголовочном файле:

```
class BrowserWindow : public QFrame
{
    Q_OBJECT
public:
    explicit BrowserWindow(const QString &url=QString(),
                           QWebPage *webPage=0, QWidget *parent=0,
                           Qt::WindowFlags flags=0);
    QString toHtml() const
    { return webView->page()->mainFrame()->toHtml(); }
    QString toPlainText() const
    { return webView->page()->mainFrame()->toPlainText(); }

signals:
    void loadFinished(bool ok);
    void urlChanged(const QUrl &url);

public slots:
    void load(const QString &url);
    void setHtml(const QString &html) { webView->setHtml(html); }
    void showToolBar(bool on) { toolBar->setVisible(on); }
    void enableActions(bool enable);

protected:
    void focusInEvent(QFocusEvent*) { webView->setFocus(); }
```

Значительная часть функциональности, особенно в части действий панели инструментов и контекстного меню, реализована в закрытых slots (не показаны), которые мы будем обсуждать по мере встречи с ними в последующих фрагментах кода. Метод `QWidget::focusInEvent()` переопределен так, что при передаче фокуса окну браузера из программы (путем вызова его метода `QWidget::setFocus()`) в фокусе реально оказывалось веб-представление. В классе имеются также некоторые закрытые переменные (не показаны), обеспечивающие доступ к его виджетам.

Благодаря изначально богатой функциональности класса `QWebView` класс `BrowserWindow` оказывается совсем маленьким, причем большая часть кода сосредоточена в конструкторе и методах с префиксом `create`. Вот как выглядит конструктор.

```
BrowserWindow::BrowserWindow(const QString &url, QWebPage *webPage,
                              QWidget *parent, Qt::WindowFlags flags)
    : QFrame(parent, flags)
{
    setFrameStyle(QFrame::Box|QFrame::Raised);
    webView = new QWebView;
    if (webPage)
        webView->setPage(webPage);
```

```

        load(url);
        createActions();
        createToolBar();
        createLayout();
        createConnections();
    }

```

Мы сделали окно браузера подклассом `QFrame`, чтобы его можно было заключить в рамку. Это полезно, так как веб-страницы часто сами содержат виджеты (кнопки, текстовые поля и т. д.), поэтому, обрамляя окно браузера, мы проводим четкую границу между веб-страницей и приложением, в которое она встроена.

Мы разрешаем клиенту класса передавать конструктору страницу `QWebPage`; если же он этого не сделает, то `QWebView` сам создаст для себя объект `QWebPage`. Это полезно, когда возникает необходимость создать подкласс `QWebPage`; мы еще встретимся с этой техникой в разделе «Внедрение виджетов Qt в веб-страницы» (стр. 60).

Метод `createActions()` немного необычен, так как самостоятельно нам приходится создавать всего несколько действий. Вот фрагмент этого метода, в котором создание действий `zoomInAction`, `setUrlAction` и `historyAction` опущено, поскольку они создаются точно так же, как представленное ниже действие `zoomOutAction`.

```

void BrowserWindow::createActions()
{
    zoomOutAction = new QAction(QIcon(":/zoomout.png"),
                                tr("Zoom Out"), this);
    zoomOutAction->setShortcuts(QKeySequence::ZoomOut);
    ...
    QList<QAction*> actions;
    actions << webView->pageAction(QWebPage::Back)
             << webView->pageAction(QWebPage::Forward)
             << webView->pageAction(QWebPage::Reload)
             << webView->pageAction(QWebPage::Stop)
             << zoomOutAction << zoomInAction << setUrlAction
             << historyAction;

#ifdef DEBUG
    actions << webView->pageAction(QWebPage::InspectElement);
#endif
    AQP::accelerateActions(actions);
    webView->addActions(actions);
    webView->setContextMenuPolicy(Qt::ActionsContextMenu);
}

```

Мы создаем список действий, которые должны присутствовать в контекстном меню окна браузера, включая и инспектирование, если макрос `DEBUG` определен, используя всюду, где возможно, предопределенные действия `QWebView`. Затем функция `AQP::accelerateActions()` назначает клавиши-акселераторы (то есть подчеркнутые буквы в названиях пун-

ктов меню, а не горячие клавиши типа Ctrl+X) на всех платформах, кроме Mac OS X (стр. 30). После этого мы просто добавляем действия в объект QWebView и просим его создать содержащее их контекстное меню.

Обратите внимание, что нам не понадобилось соединять сигналы со слотами или заводить слоты для действий, предоставляемых QWebView, поскольку все это уже встроено.

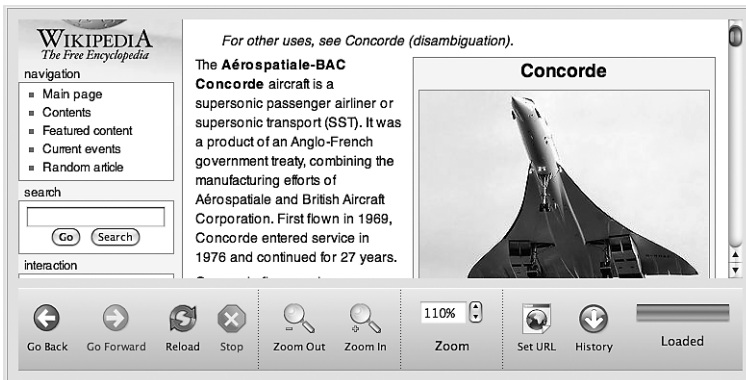


Рис. 1.8. Компонент, содержащий окно браузера, на платформе Mac OS X

Мы не привели текст метода `createToolBar()`, поскольку во всех приложениях C++/Qt он более-менее стандартный. Однако, как видно на рис. 1.8 (сравните с рис. 1.6), панель инструментов в Mac OS X скомпонована иначе, чем на других платформах. Так сделано потому, что в Mac OS X принято отображать на кнопках инструментов значок и ниже пояснительный текст. И чтобы сохранить единый стиль оформления панели инструментов, мы поместили метки счетчика (spinbox) и индикатора хода выполнения под элементами, а не слева от них, как на других платформах. Для этого нам пришлось включить в файл `browserwindow/browserwindow.cpp` директивы `#ifdef` и воспользоваться тем фактом, что макрос `Q_WS_MAC` определен только на платформе Mac OS X¹.

Метод `createLayout()` небольшой и вполне стандартный, так что мы его пропустим и перейдем к методу `createConnections()`.

```
void BrowserWindow::createConnections()
{
    connect(webView, SIGNAL(loadProgress(int)),
            progressBar, SLOT(setValue(int)));
    connect(webView, SIGNAL(urlChanged(const QUrl&)),
            this, SLOT(urlChange(const QUrl&)));
    connect(webView, SIGNAL(loadFinished(bool)),
```

¹ Как отмечалось во введении, полный исходный код можно скачать со страницы www.qtrac.eu/aqpbook.html.

```
        this, SLOT(loadFinish(bool)));
connect(setUrlAction, SIGNAL(triggered()),
        this, SLOT(setUrl()));
connect(historyAction, SIGNAL(triggered()),
        this, SLOT(popUpHistoryMenu()));
connect(zoomOutAction, SIGNAL(triggered()),
        this, SLOT(zoomOut()));
connect(zoomInAction, SIGNAL(triggered()),
        this, SLOT(zoomIn()));
connect(zoomSpinBox, SIGNAL(valueChanged(int)),
        this, SLOT(setZoomFactor(int)));
    }
```

Первые три соединения позволяют следить за ходом загрузки и изменением страницы, а значит, своевременно обновлять индикатор хода выполнения и связанную с ним метку. Остальные соединения нужны, чтобы обрабатывать действия, которые мы создали сами, и реагировать на изменение пользователем счетчика, управляющего масштабированием.

Теперь мы уже достаточно знаем об окне браузера и можем ознакомить-ся с методами, которые отвечают за его поведение.

```
void BrowserWindow::load(const QString &url)
{
    if (url.isEmpty())
        return;
    QString theUrl = url;
    if (!theUrl.contains("://"))
        theUrl.prepend("http://");
    webView->load(theUrl);
}
```

Если переданная строка URL-адреса пуста, мы ничего не делаем. В противном случае в качестве любезности добавляем в начало «http://», если никакая другая схема не указана, и просим объект `QWebView` загрузить страницу. (Мы используем `QString`, а не `QUrl`, потому что корректно добавить схему проще в строку, чем в `QUrl`.)

```
void BrowserWindow::setUrl()
{
    load(QInputDialog::getText(this, tr("Set URL"), tr("&URL:")));
}
```

Этот метод вызывается, если пользователь выполняет действие «Задать URL». Если же пользователь производит отмену, то методу `load()` будет передан пустой URL, и он просто ничего не сделает.

```
void BrowserWindow::urlChange(const QUrl &url)
{
    emit urlChanged(url);
    progressLabel->setText(tr("Loading"));
}
```

При каждом изменении URL-адреса страницы (в результате щелчка по ссылке, выполнения действия «Задать URL» или выбора ссылки из истории просмотра) вызывается этот слот. Мы генерируем собственный сигнал `urlChanged()`, который может быть полезен клиентам класса `BrowserWindow`, и обновляем метку хода выполнения, уведомляя о том, что началась загрузка.

```
void BrowserWindow::loadFinish(bool ok)
{
    emit loadFinished(ok);
    progressBar->setText(ok ? tr("Loaded") : tr("Canceled"));
}
```

Этот слот вызывается по окончании загрузки, причем булевый аргумент `ok` показывает, завершилась она успешно или нет. Опять же, для удобства клиентов `BrowserWindow` мы генерируем сигнал, после чего обновляем метку хода выполнения так, чтобы она отражала сложившуюся ситуацию.

Заботиться об актуальности индикатора хода выполнения нет нужды, так как в конце конструктора¹ мы соединили сигнал `loadProgress()` веб-представления со слотом индикатора `setValue()`.

```
void BrowserWindow::setZoomFactor(int zoom)
{
    webView->setZoomFactor(zoom / 100.0);
}
```

Этот слот вызывается, когда пользователь как-то изменяет счетчик, управляющий масштабированием. В результате изменяется размер текста в окне представления. (Если мы захотим, чтобы масштабировались также изображения, то должны будем присвоить атрибуту `QWebSettings::ZoomTextOnly` значение `false`.)

```
const int ZoomStepSize = 5;
void BrowserWindow::zoomOut()
{
    zoomSpinBox->setValue(zoomSpinBox->value() - ZoomStepSize);
}
```

С этим слотом соединено действие `zoomOutAction`. Имеются, хотя и не показаны, также действие `zoomInAction` и соответствующий ему слот. Когда вызываются эти слоты, происходит обращение к методу `setValue()` и счетчик генерирует сигнал `valueChanged()`, а это приводит к вызову слота `setZoomFactor()` из-за наличия рассмотренного выше соединения.

```
void BrowserWindow::enableActions(bool enable)
{
    foreach (QAction *action, webView->actions())
```

¹ Точнее, в начале метода `CreateConnections()`, вызываемого в конце конструктора. — *Прим. науч. ред.*

```

        action->setEnabled(enable);
        toolBar->setEnabled(enable);
        webView->setContextMenuPolicy(enable ? Qt::ActionsContextMenu
                                          : Qt::NoContextMenu);
    }

```

Иногда приложение, в которое встраивается окно браузера, не позволяет пользователю задействовать какую-нибудь функциональность браузера, помимо просмотра и взаимодействия с показанной страницей. Этот метод позволяет включать или отключать все действия в окне браузера.

```

const int MaxHistoryMenuItems = 20;
const int MaxMenuWidth = 300;

void BrowserWindow::popUpHistoryMenu()
{
    QFontMetrics fontMetrics(font());
    QMenu menu;
    QSet<QUrl> uniqueUrls;
    QListIterator<QWebHistoryItem> i(webView->history()->items());
    i.toBack();
    while (i.hasPrevious() &&
           uniqueUrls.count() < MaxHistoryMenuItems) {
        const QWebHistoryItem &item = i.previous();
        if (uniqueUrls.contains(item.url()))
            continue;
        uniqueUrls << item.url();
        QString title = fontMetrics.elidedText(item.title(),
                                                Qt::ElideRight, MaxMenuWidth);
        QAction *action = new QAction(item.icon(), title, &menu);
        action->setData(item.url());
        menu.addAction(action);
    }
    AQP::accelerateMenu(&menu);
    if (QAction *action = menu.exec(QCursor::pos()))
        webView->load(action->data().toUrl());
}

```

Этот метод отображает меню, пункты которого соответствуют просмотренным пользователем страницам. Данные о посещенных ссылках запрашиваются у объекта `QWebHistory`, принадлежащего `QWebView`. В нем находится список объектов `QWebHistoryItem`, в каждом из которых хранится URL-адрес страницы, ее пиктограмма (Qt предоставляет пиктограмму по умолчанию, если она не определена веб-сервером) и еще кое-какие данные.

В этом меню ссылки перечислены в обратном порядке, то есть страница, посещенная последней, находится сверху, а самая старая – внизу. Кроме того, количество пунктов меню ограничено, а дубликаты устранены, поэтому порядок посещения воспроизведен не совсем точно. У некоторых страниц заголовки очень длинные, тогда мы обрезаем текст заго-

ловка справа, заменяя отброшенную часть многоточием. Для этой цели используется метод `QFontMetrics::elidedText()`. Можно также обрезать текст слева или вырезать фрагмент из середины, для этого достаточно передать в качестве второго аргумента `Qt::ElideLeft` или `Qt::ElideMiddle` соответственно.

Выше мы отмечали, что до выхода версии 4.7 в Qt отсутствовала функция `qHash(QUrl)`, поэтому просто так взять и сохранить объект `QUrl` в контейнере `QSet` не получится. Но поскольку класс `QSet` реализован с помощью `QHash`, то решить проблему можно, как в приложении `Weather Tray Icon`, добавив однострочную функцию `qHash(QUrl)`.

Если пользователь закроет меню, ничего не выбрав (нажав клавишу Esc или щелкнув мышью в любом месте вне меню), то метод `QMenu::exec()` вернет 0, в противном случае – указатель на объект `QAction`, соответствующий выбранному пункту меню. Если возвращен указатель на `QAction`, то мы извлекаем из объекта хранящийся в нем URL. А зная URL, мы можем попросить веб-представление загрузить соответствующую страницу.

На этом мы завершаем знакомство с компонентом «окно браузера». У браузера есть и другие стандартные действия, которые можно было бы добавить. Некоторые из них реализуются легко, поскольку необходимая функциональность уже встроена в WebKit. Например, можно было бы включить функцию поиска в тексте, основанную на методе `QWebView::findText()`, и функцию печати, основанную на одном из методов `QWebFrame::print()` или `QWebFrame::render()`.

В следующих двух подразделах мы воспользуемся окном браузера как основной составной частью гибридных приложений. Мы также научимся загружать веб-контент незаметно для пользователя и внедрять в веб-страницы JavaScript-сценарий с целью извлечения из них информации. А кроме того, мы узнаем, как усовершенствовать окно браузера, чтобы в нем можно было отображать стандартные и написанные пользователем виджеты Qt. И не только отображать, но и взаимодействовать с ними.

Создание приложений для работы с конкретным сайтом

Если человек проводит много времени на каком-то одном сайте, то почему бы не предложить ему больше функциональности в более удобной упаковке, написав приложение, ориентированное на этот сайт и на потребности данного пользователя. Конечно, такие приложения подвержены определенному риску – они могут перестать работать при изменении структуры сайта, но иногда экономия времени, достигаемая за счет удобной организации работы (особенно если у сайта много пользователей), перевешивает риск. К тому же влияние изменений можно ограничить, так что модифицировать придется только работающий «за кулисами» JavaScript-сценарий, не трогая исходный код.

При наличии приложения, ориентированного на конкретный сайт, можно также гарантировать, что пользователи смогут обращаться к этому сайту исключительно из данного приложения. Пожалуй, самый известный пример подобного рода – музыкальный магазин iTunes Music Store компании Apple, в который (во время работы над этой книгой) невозможно зайти из стандартного браузера.

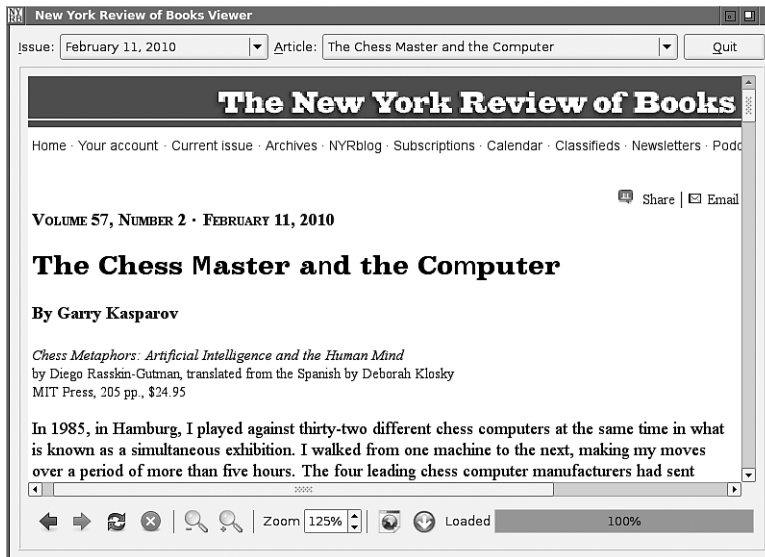


Рис. 1.9. Программа просмотра сайта *The New York Review of Books*

В этом подразделе мы рассмотрим приложение New York Review of Books Viewer (nyrbviewer), показанное на рис. 1.9. Оно показывает страницы сайта NYRB (*New York Review of Books*) с помощью компонента «окно браузера» из предыдущего подраздела. Это приложение удобнее обычного браузера тем, что показывает два комбинированных списка: с номерами выпусков и перечнем статей в выбранном выпуске. Это позволяет быстро узнать, какие номера доступны, и выбрать интересующую пользователя статью.

Большая часть функциональность уже имеется в компоненте «окно браузера», нам осталось только заполнить списки данными и организовать работу с ними. Начнем с конструктора:

```
const QString NYRBUrL("http://www.nybooks.com");

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    createWidgets();
    createLayout();
    issueLinkFetcher = new LinkFetcher(NYRBUrL,
```

```

        scriptPathAndName("fetch_issue_links.js"), this);
articleLinkFetcher = new LinkFetcher(NYRBUrl,
        scriptPathAndName("fetch_article_links.js"), this);
createConnections();
AQP::accelerateWidget(this);
issueComboBox->setFocus();
issueLinkFetcher->load(NYRBUrl);
setWindowTitle(QApplication::applicationName());
QTimer::singleShot(1000 * 60, this, SLOT(networkTimeout()));
}

```

Как обычно, мы создаем виджеты и менеджеры компоновки. Функция `AQP::accelerateWidget()` вызывает `QObject::findChildren()`, чтобы найти все метки `QLabel` и виджеты типа `QAbstractButton` (и его подклассов, отличных от `QToolButton`), являющиеся потомками окна, и назначает им клавиши-акселераторы (стр. 30).

Новым для нас является создание двух объектов типа `LinkFetcher`. Каждый из них создает объект `QWebPage` (напомним, что `QWebPage` — информационный объект, то есть не связан с окном) и просит его загрузить указанную страницу. Затем в эту страницу внедряется переданный при создании JavaScript-сценарий, который находит все интересующие нас ссылки. Переданный конструктору URL-адрес сайта применяется для того, чтобы преобразовать относительные ссылки вида `articles/22273` в абсолютные, например `http://www.nybooks.com/articles/22273`.

Создав сборщики ссылок и соединив их сигналы со слотами, мы запускаем процедуру: просим объект `issueLinkFetcher` загрузить главную страницу сайта NYRB и с помощью сценария `fetch_issue_links.js` извлекаем ссылки на все выпуски; затем эти ссылки используются для заполнения комбинированного списка выпусков. После того как выпуск выбран, мы просим объект `articleLinkFetcher` загрузить страницу этого выпуска и с помощью сценария `fetch_article_links.js` извлекаем ссылки на статьи; они заносятся в комбинированный список статей. Класс `LinkFetcher` мы рассмотрим в конце этого раздела.

Мы завели метод `scriptPathAndName()` для того, чтобы обеспечить себе гибкость при работе со сценариями, чуть ниже мы расскажем об этом подробнее.

И напоследок отметим использование однократного таймера. Он срабатывает через минуту и вызывает слот `networkTimeout()`; это позволит вывести сообщение об ошибке, если окажется, что Интернет недоступен.

```

QString MainWindow::scriptPathAndName(const QString &filename)
{
    QString name = filename;
    QString path = AQP::applicationPathOf() + "/";
    if (QFile::exists(path + name))
        return path + name;
    return QString(":/%1").arg(name);
}

```

Этот метод ищет указанный сценарий в каталоге, содержащем исполняемый файл приложения, и, обнаружив его, возвращает полный путь и имя, которые передаются сборщику ссылок. Если же в файловой системе не удалось найти сценарий, то будет использоваться сценарий, включенный в исполняемый файл в виде ресурса. Это означает, что по умолчанию приложение работает со встроенными в него сценариями, но если возникает необходимость заменить сценарий (например, потому, что изменилась структура сайта), то достаточно поместить новую версию в каталог с исполняемым файлом. Приложение автоматически обнаружит новый сценарий и будет использовать его вместо встроенного. (Разумеется, если вы озабочены безопасностью, то можете принимать только файлы в двоичном формате с контрольной суммой или еще как-то проверять законность локального установленного сценария, но эти вопросы выходят за рамки данной книги.)

Функция `AQP::applicationPathOf()` из модуля `aqp.{hpp,cpp}` возвращает путь к приложению (в виде объекта `QString`) или путь к подкаталогу внутри каталога приложения, если передано имя этого подкаталога. Мы не можем напрямую воспользоваться функцией `QApplication::applicationDirPath()`, потому что она не учитывает того факта, что исполняемый файл может находиться в разных каталогах в зависимости от того, выпущено приложение или находится в процессе разработки. (Например, в процессе разработки на платформе Windows исполняемый файл обычно находится в подкаталоге *debug* или *release*.)

```
const QString InitialMessage(
    QObject::tr("Attempting to connect to the network..."));
const QString FailMessage(
    QObject::tr("No issues or articles available"));

void MainWindow::networkTimeout()
{
    const QString text = browser->toPlainText().trimmed();
    if (text == InitialMessage || text == FailMessage)
        browser->setHtml("<h3><font color=red>Failed to connect "
            "to the network</font></h3>Perhaps the proxy "
            "settings are wrong, or maybe a proxy is needed. "
            "Try:<br><tt>nyrbviewer --help</tt>");
}
```

Этот слот вызывается через минуту после запуска приложения. Если подключиться к Интернету не удалось, то в окне браузера будет находиться строка `InitialMessage` или `FailMessage`, — тем самым мы попытались немного помочь пользователю (см. врезку «Поддержка сетевых прокси-серверов» на стр. 23).

В заголовочном файле определена закрытая переменная `namesForUrlsForIssueCache`:

```
QHash<int, QMap<QString, QString> > namesForUrlsForIssueCache;
```


Ключами этого кэша являются индексы элементов в комбинированном списке выпусков, а значениями – объекты QMap с URL-адресами в качестве ключей и названиями статей в качестве значений. Как он используется, мы увидим, когда займемся методами главного окна, а пока отметим, что сборщик ссылок может возвращать отображение URL на название статьи для тех ссылок, которые он нашел на загруженной странице. (Мы не используем QCache, потому что не хотим, чтобы кэшированные данные вытеснялись из кэша.)

Мы не приводим методы `createWidgets()` и `createLayout()`, потому что они абсолютно стандартны, но методу `createConnections()` уделим внимание.

```
void MainWindow::createConnections()
{
    connect(issueLinkFetcher, SIGNAL(finished(bool)),
            this, SLOT(populateIssueComboBox(bool)));
    connect(articleLinkFetcher, SIGNAL(finished(bool)),
            this, SLOT(populateCache(bool)));
    connect(issueComboBox, SIGNAL(currentIndexChanged(int)),
            this, SLOT(currentIssueIndexChanged(int)));
    connect(articleComboBox, SIGNAL(currentIndexChanged(int)),
            this, SLOT(currentArticleIndexChanged(int)));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
}
```

После того как сборщик ссылок загрузил страницу и нашел в ней все интересующие его ссылки, он генерирует сигнал `finished(bool)`, передавая ему булевый параметр, который показывает, как завершилась операция: успешно или нет. Сборщик ссылок на выпуски соединяется непосредственно со слотом `populateIssueComboBox()`, однако сборщик ссылок на статьи мы соединяем со слотом `populateCache()`, а тот уже вызывает `populateArticleComboBox()` после того, как заполнит кэш.

Когда пользователь выбирает выпуск, мы должны обновить список статей, а когда он выбирает статью, мы должны загрузить ее в окно браузера. То и другое поведение задается в третьем и четвертом вызовах функции `connect`. Блок-схема работы приложения приведена на рис. 1.10.

Теперь обратимся к методам главного окна. Начнем с тех, что касаются списка выпусков, а потом перейдем к методам, работающим со списком статей.

```
void MainWindow::populateIssueComboBox(bool ok)
{
    if (ok)
        populateAComboBox(tr("- no issue selected -"),
                           issueLinkFetcher->namesForUrls(), issueComboBox);
    else {
        issueComboBox->clear();
        issueComboBox->addItem(tr("- no issues available -"));
    }
}
```

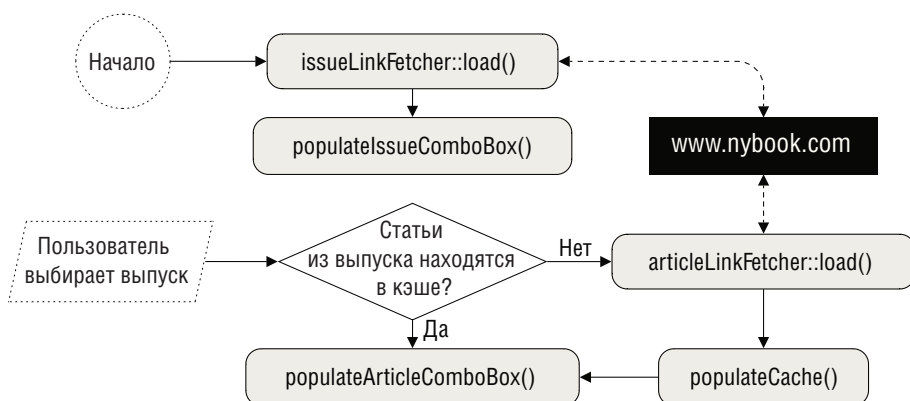


Рис. 1.10. Блок-схема приложения *The New York Review of Books Viewer*

Этот слот вызывается только один раз после того, как метод `LinkFetcher::load()` объекта `issueLinkFetcher`, вызванный в конструкторе главного окна, сгенерирует сигнал `LinkFetcher::finished()`. Сборщик ссылок на выпуски находит названия и URL-адреса всех выпусков (начиная с 2000 года). Найденные результаты представлены в виде отображения `QMap<QString, QString>`, в котором ключами служат URL-адреса, а значениями – названия выпусков. Это отображение передается методу `populateAComboBox()`, который переносит хранящиеся в нем данные в комбинированный список.

```

void MainWindow::populateAComboBox(const QString &statusText,
    const QMap<QString, QString> &namesForUrls,
    QComboBox *comboBox)
{
    comboBox->clear();
    comboBox->addItem(statusText);
    QMapIterator<QString, QString> i(namesForUrls);
    i.toBack();
    while (i.hasPrevious()) {
        i.previous();
        comboBox->addItem(i.value(), i.key());
    }
    if (comboBox->count() > 1)
        comboBox->setCurrentIndex(1);
}
  
```

Этот метод вызывается из слота `populateIssueComboBox()` и из метода `populateArticleComboBox()`.

Сначала мы очищаем список и помещаем в качестве первого элемента строку, переданную в аргументе `statusText`. Затем мы обходим в обратном порядке отображение URL-адресов на названия выпусков (или на названия статей, если метод вызван для заполнения списка статей), которое было построено и передано сборщиком ссылок. В случае сайта

NYRB в URL-адресе выпуска и статьи закодирована дата, например /contents/20090115, поэтому мы обходим элементы, начиная с самого позднего по времени выпуска. Названия выпусков – это просто даты в формате, принятом в США (например, «Jan 15, 2009»), они и помещаются в комбинированный список в качестве текста элементов, а ассоциированными данными служат URL-адреса. В случае статей названия – это настоящие заголовки статей.

Если в списке больше одного элемента, то есть имеется хотя бы один выпуск или статья, то первый элемент (самый последний выпуск или статья) делается текущим, что приводит к вызову одного из слотов `currentIssueIndexChanged()` или `currentArticleIndexChanged()`.

```
void MainWindow::currentIssueIndexChanged(int index)
{
    articleComboBox->clear();
    if (index == 0) {
        articleComboBox->addItem(tr("- no issue selected -"));
        return;
    }
    if (namesForUrlsForIssueCache.contains(index))
        populateArticleComboBox();
    else {
        articleComboBox->addItem(
            tr("+ fetching the list of articles +"));
        browser->setHtml(tr("<h3><font color=red>"
            "Fetching the list of articles...</font></h3>"));
        QString url = issueComboBox->itemData(index).toString();
        articleLinkFetcher->load(url);
    }
}
```

При выборе нового выпуска мы сначала очищаем *список статей*. Если выбран конкретный выпуск, то смотрим, присутствует ли отображение URL на названия статей для него в кэше. Если да, то можно сразу вызывать метод `populateArticleComboBox()`, который всегда получает данные из кэша.

Если же данных о только что выбранном выпуске в кэше еще нет, то мы помещаем в список статей и в окно браузера сообщение о состоянии. Затем получаем URL-адрес выпуска (он хранится в поле данных выбранного элемента списка) и просим сборщик ссылок на статьи загрузить страницу выпуска и извлечь из нее ссылки на статьи. По завершении загрузки вызывается слот `populateCache()` – благодаря установленному ранее соединению между сигналом и слотом.

```
void MainWindow::populateCache(bool ok)
{
    if (!ok || issueComboBox->count() == 1) {
        articleComboBox->setItemText(0,
            tr("- no articles available -"));
    }
}
```

```

        browser->setHtml(tr("<h3><font color=red>%1</font></h3>")
                        .arg(FailMessage));

        return;
    }
    QTextDocument document;
    QMap<QString, QString> namesForUrls = articleLinkFetcher->namesForUrls();
    QMapIterator<QString, QString> i(namesForUrls);
    while (i.hasNext()) {
        i.next();
        document.setHtml(i.value());
        i.setValue(document.toPlainText());
    }
    namesForUrlsForIssueCache[issueComboBox->currentIndex()] = namesForUrls;
    populateArticleComboBox();
}

```

Если формирование списка статей завершилось неудачно, мы информируем об этом пользователя, помещая соответствующий текст в комбинированный список статей и в окно браузера. Если же все прошло успешно, то мы получаем от сборщика ссылок на статьи отображение URL-адресов на названия статей, добавляем новую информацию в кэш и вызываем метод `populateArticleComboBox()`.

Названия статей нам нужны в виде простого текста, потому что они будут отображаться в списке `QComboBox` (который не умеет обрабатывать разметку). Вместо того чтобы каждый раз создавать и уничтожать объект `QTextDocumentFragment` (то есть вызывать `i.setValue(QTextDocumentFragment::fromHtml(i.value()).toPlainText());`), мы один раз создадим `QTextDocument` и на каждой итерации цикла будем устанавливать для него HTML-текст, а затем вызывать метод `toPlainText()`.

```

void MainWindow::populateArticleComboBox()
{
    int index = issueComboBox->currentIndex();
    if (index > 0)
        populateAComboBox(tr("- no article selected -"),
                           namesForUrlsForIssueCache[index], articleComboBox);
}

```

Этот метод вызывается только в том случае, когда в кэше уже имеются данные о статьях в выбранном выпуске. Мы заполняем комбинированный список с помощью метода `populateAComboBox()`, показанного на стр. 51, передавая ему текст сообщения о состоянии, которое станет первым элементом списка, взятое из кэша отображение URL на названия статей и сам заполняемый список. Если пользователь выбрал из списка выпусков первый элемент («no issue selected»), то мы ничего не делаем.

```

void MainWindow::currentArticleIndexChanged(int index)
{
    if (index == 0)
        return;
}

```

```

        QString url = articleComboBox->itemData(index).toString();
        browser->load(url);
        browser->setFocus();
    }

```

Когда пользователь выбирает элемент из списка статей (любой, кроме первого, «no article selected»), мы извлекаем ассоциированный с этой статьей URL-адрес из поля данных выбранного элемента и просим окно браузера загрузить соответствующую страницу. Затем мы передаем окну браузера фокус (который в действительности оказывается у виджета `QWebView`, принадлежащего этому компоненту), чтобы пользователь мог сразу же приступить к прокрутке содержимого с помощью курсорных клавиш (со стрелками).

Итак, мы полностью рассмотрели реализацию приложения `New York Review of Books Viewer` (за исключением заголовочного файла и методов `createWidgets()` и `createLayout()`). Теперь займемся классом `LinkFetcher` и посмотрим, как он реализует загрузку страниц и сбор ссылок. Вот как выглядит его заголовочный файл:

```

class LinkFetcher : public QObject
{
    Q_OBJECT

public:
    explicit LinkFetcher(const QString &site_,
                        const QString &scriptOrScriptName_, QObject *parent=0);
    void load(const QString &url);
    QMap<QString, QString> namesForUrls() const
    { return m_namesForUrls; }
    void clear() { m_namesForUrls.clear(); }

signals:
    void finished(bool);

public slots:
    void addUrlAndName(const QString &url, const QString &name);

private slots:
    void injectJavaScriptIntoWindowObject();
    void fetchLinks(bool ok);

private:
    QWebPage page;
    QMap<QString, QString> m_namesForUrls;
    const QString site;
    const QString scriptOrScriptName;
};

```

В этом классе для загрузки страницы с указанным URL-адресом применяется класс `QWebPage`. Затем переданный в конструкторе `JavaScript-`

сценарий находит на странице нужные ссылки и сохраняет информацию о них – URL и текст ссылки – в отображении `m_namesForUrls`.

Мы можем сделать доступным JavaScript-сценарию объект любого класса, производного от `QObject`. Это означает, что сценарий, внедренный в загруженную страницу, может обращаться не только к элементам самой страницы (с помощью ее объектной модели DOM), но и к любым объектам `QObject`, которые мы сделали доступными. В частности, JavaScript-сценарий сможет вызывать открытые слоты `QObject` и обращаться к свойствам `QObject`, если таковые имеются. Для этого мы передаем ссылку на экземпляр `LinkFetcher`, чтобы внедренный сценарий мог взаимодействовать с нашим кодом на C++, вызывая открытые слоты сборщика ссылок.

```
LinkFetcher::LinkFetcher(const QString &site_,
                        const QString &scriptOrScriptName_, QObject *parent)
    : QObject(parent), site(site_),
      scriptOrScriptName(scriptOrScriptName_)
{
    QWebSettings *webSettings = page.settings();
    webSettings->setAttribute(QWebSettings::AutoLoadImages, false);
    webSettings->setAttribute(QWebSettings::PluginsEnabled, false);
    webSettings->setAttribute(QWebSettings::JavaEnabled, false);
    webSettings->setAttribute(QWebSettings::JavascriptEnabled, true);
    webSettings->setAttribute(QWebSettings::PrivateBrowsingEnabled, true);
    connect(page.mainFrame(), SIGNAL(javascriptWindowObjectCleared()),
            this, SLOT(injectJavaScriptIntoWindowObject()));
    connect(&page, SIGNAL(loadFinished(bool)),
            this, SLOT(fetchLinks(bool)));
}
```

Поскольку мы загружаем страницу незаметно для пользователя и интересуют нас только находящиеся на ней ссылки, то изменим параметры `QWebPage` для этой страницы. Отключаем загрузку изображений, деактивируем подключаемые модули и Java, активируем JavaScript, а также включаем режим `QWebSettings::PrivateBrowsingEnabled`, в котором не сохраняется ни история посещения страниц, ни связанные с ними пиктограммы. Все это нам не нужно и только будет понапрасну потреблять память и процессорное время.

Когда начинается загрузка страницы, JavaScript-объекты `window`¹ во всех принадлежащих ей фреймах `QWebFrame` очищаются и подготавливаются к обработке новых JavaScript-сценариев, которые могут присутствовать во фреймах данной страницы. Поскольку мы хотим внедрить собственный сценарий в оконный объект главного фрейма загруженной

¹ Данный объект является так называемым глобальным, то есть все другие объекты и функции, доступные сценариям JavaScript, на самом деле являются его полями. – *Прим. науч. ред.*

страницы, то должны позаботиться о том, чтобы повторить внедрение после очистки. В этом и состоит задача первого соединения сигнала со слотом.

Второе соединение необходимо для того, чтобы сразу после завершения загрузки мы могли приступить к извлечению ссылок.

```
void LinkFetcher::injectJavaScriptIntoWindowObject()
{
    page.mainFrame()->addToJavaScriptWindowObject("linkFetcher", this);
}
```

Метод `QWebFrame::addToJavaScriptWindowObject()` позволяет добавить произвольный объект `QObject` к полям JavaScript-объекта `window` в данном фрейме `QWebFrame`. Его первый аргумент – строка, содержащая имя, по которому JavaScript-сценарий сможет найти объект (в данном случае `linkFetcher`), а второй аргумент – ссылка на сам объект `QObject` (в данном случае на текущий экземпляр класса `LinkFetcher`).

```
void LinkFetcher::load(const QString &url)
{
    clear();
    page.mainFrame()->load(QUrl(url));
}
```

Когда сборщику ссылок передается URL-адрес подлежащей загрузке страницы, он сначала очищает отображение `m_namesForUrls`, а затем просит главный фрейм `QWebPage` выполнить загрузку. По завершении загрузки вызывается слот `fetchLinks()` – благодаря ранее установленному соединению его с сигналом.

```
void LinkFetcher::fetchLinks(bool ok)
{
    if (!ok) {
        emit finished(false);
        return;
    }
    QString javaScript = scriptOrScriptName;
    if (scriptOrScriptName.endsWith(".js")) {
        QFile file(scriptOrScriptName);
        if (!file.open(QIODevice::ReadOnly)) {
            emit finished(false);
            return;
        }
        javaScript = QString::fromUtf8(file.readAll());
    }
    QWebFrame *frame = page.mainFrame();
    frame->evaluateJavaScript(javaScript);
    emit finished(true);
}
```

Если загрузка завершилась неудачно, то мы уведомляем об этом все связанные объекты. В противном случае смотрим, что хранится в за-

крытой переменной `scriptOrScriptName` типа `QString`: сам сценарий или только его имя. В последнем случае мы пытаемся прочитать текст сценария. Вместо того чтобы создавать объект `QTextStream`, мы открываем файл в двоичном режиме и преобразуем возвращенный методом `QFile::readAll()` массив байтов `QByteArray` в строку, пользуясь статическим методом `QString::fromUtf8()`. Подготовив сценарий, мы просим главный фрейм страницы `QWebPage` выполнить его, а затем уведомляем все подсоединенные объекты об успешном завершении.

Метод `QWebFrame::evaluateJavaScript()` возвращает объект типа `QVariant`, который содержит значение последнего вычисленного выражения в JavaScript-сценарии; мы его игнорируем, потому что выбрали более гибкий подход: взаимодействие между JavaScript и C++. Поскольку в качестве объекта, доступного JavaScript-сценарию, мы установили сам сборщик ссылок, то сценарий может вызывать любые открытые слоты сборщика и обращаться к его свойствам. В данном случае мы создали слот `addUrlAndName()`, предназначенный специально для JavaScript-сценариев.

```
void LinkFetcher::addUrlAndName(const QString &url, const QString &name)
{
    if (url.startsWith("http://"))
        m_namesForUrls[url] = name;
    else
        m_namesForUrls[site + url] = name;
}
```

Всякий раз, как сценарий находит ссылку, он вызывает этот слот, передавая ему URL-адрес и название, и эти данные заносятся в отображение `m_namesForUrls`. URL-адрес может быть как абсолютным, так и относительным, поэтому в начало относительных адресов мы дописываем адрес сайта, переданный конструктору сборщика ссылок.

На рис. 1.11 показано, как в результате добавления на страницу объекта `QObject`, принадлежащего приложению, сценарий, исполняемый в контексте страницы, получает возможность обращаться как к HTML-элементам страницы (посредством DOM), так и к любым объектам приложения, к которым у страницы есть доступ.

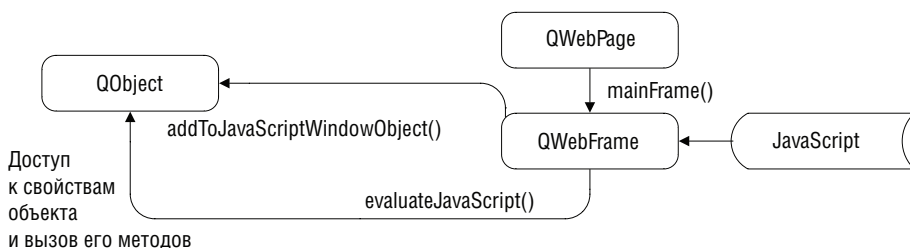


Рис. 1.11. Внедрение JavaScript-сценария для доступа к HTML-элементам и объектам `QObject` приложения

На этом мы завершаем обзор класса `LinkFetcher`, но, чтобы вы лучше поняли, как все это работает, приведем код сценария `fetch_article_links.js` (поскольку он самый короткий из двух).

```
Java | var links = document.getElementsByTagName("a");  
Script | for (var i = 0; i < links.length; ++i) {  
        |     if (links[i].href.search("/articles/") != -1) {  
        |         linkFetcher.addUrlAndName(links[i].href, links[i].innerHTML);  
        |     }  
        | }  
        | }
```

Метод JavaScript `getElementsByTagName()` находит все теги `<a>`, содержащие ссылки, — сама ссылка содержится в свойстве `href`, а текст между тегами `<a>` и `` доступен в виде свойства `innerHTML`.

Наибольший интерес представляет последнее выражение, где с помощью ссылки `linkFetcher` на сборщик ссылок, написанный на C++, вызывается его слот `LinkFetcher::addUrlAndName()`.

Первоначальная версия сценария `fetch_issue_links.js` была очень похожа на `fetch_article_links.js`, но чуть длиннее (около 20 строк), поскольку мы читали ссылки на выпуски из комбинированного списка, присутствовавшего на главной странице сайта.

За те девять месяцев, что писалась эта книга, в структуре главной страницы сайта NYRB произошли значительные изменения, и комбинированный список, из которого сценарий `fetch_issue_links.js` читал перечень выпусков, приказал долго жить. Такого списка нет и на других страницах, перечень выпусков в какой бы то ни было форме вообще отсутствует. Однако веб-мастер сообщил, что для каждого года имеется страница, на которой представлен список выпусков за этот год. Вооружившись этой информацией, мы смогли без труда написать новый сценарий `fetch_issue_links.js` — длиной около 40 строк — и поместили его в один каталог с исполняемым файлом. А поскольку мы заложили в приложение возможность искать сценарии в этом каталоге и исполнять их вместо сценариев, внедренных как ресурсы, то не пришлось даже ничего перекомпилировать; программа автоматически «подцепила» новую версию `fetch_issue_links.js`.

Поскольку новый сценарий `fetch_issue_links.js` должен прочитать по одной странице на каждый год (мы сконфигурировали его для чтения информации о выпусках за последние пять лет), то работает он медленнее первоначального, так как отправляет синхронные GET-запросы на загрузку каждой страницы¹. Впрочем, решение оказалось несложным в реализации и работает вполне прилично. Однако в перспективе было бы желательно уйти от синхронной загрузки страниц JavaScript-сценарием, потому что при этом блокируется цикл обработки сообще-

¹ В новом сценарии используется объект `XMLHttpRequest`, и основан он на идеях, почерпнутых из книги Дэвида Флэнагана «JavaScript. Подробное руководство», 5-е издание, Символ-Плюс, 2008.

ний в графическом интерфейсе. Так что в идеале следовало бы перепроектировать программу, так, чтобы для получения списка выпусков использовались классы Qt для работы с сетью, а сценарий `fetch_article_links.js` оставить только для поиска статей. Оставляем это читателю в качестве несложного упражнения.

Прежде чем закончить разговор о JavaScript в этом подразделе, полезно будет обсудить вопрос об отладке приложений, в которых применяется внедрение JavaScript-сценариев, поскольку это отнюдь не простая задача! Простой способ решить ее – писать все сценарии так, чтобы они возвращали некоторое значение, и затем проверять его. Но в данном примере мы выбрали более гибкий подход – добавили еще один открытый слот в заголовочный файл сборщика ссылок:

```
void debug(const QString &value)
{ qDebug("%s", qPrintable(value)); }
```

В процессе разработки мы вызывали этот слот из JavaScript-сценариев (вызовы `debug()` остались в исходном коде, но закомментированы). Например, в цикле `for` в сценарии `fetch_article_links.js` имеется такая строка (не показанная в приведенном выше листинге):

```
linkFetcher.debug(links[i].href + " * " + links[i].innerHTML);
```

Java
Script

Такое добавление вызовов `debug()` бывает очень полезно. (На платформе Windows в *pro*-файл, конечно, следует добавить строку `CONFIG += console`, чтобы был виден отладочный вывод.)

К сожалению, эта техника бесполезна, если в сценарии имеется синтаксическая ошибка, потому что тогда он просто не запустится. Убедиться в отсутствии синтаксических ошибок можно, например, запустив поставляемую в комплекте с Qt программу *qscript* (она находится в каталоге *examples*) и указав в командной строке имя файла сценария в качестве аргумента. Если ошибок нет, то *qscript* попытается выполнить сценарий, в противном случае напечатает сообщение с указанием номера строки, в которой была обнаружена первая ошибка.

В рассмотренном приложении The New York Review of Books Viewer используются довольно простые JavaScript-сценарии (хотя последняя версия `fetch_issue_links.js` уже не столь тривиальна). Но сценарии могут быть и гораздо более изощренными, особенно если они обращаются к объектной модели (DOM) загруженной страницы. Кроме того, для доступа к веб-страницам через DOM (в том числе и для модификации) можно использовать появившийся в версии Qt 4.6 класс `QWebElement`, что очень удобно при работе с сайтами, которые заведомо стабильны или которые мы можем контролировать. В общем, вам решать, чему отдать предпочтение: простоте и удобству JavaScript-сценариев, которые можно оперативно изменять при изменении структуры сайта, или мощи и возможности асинхронного доступа, присущей классам Qt для работы с сетью. В последнем случае интерфейс пользователя не будет зависеть, а все веб-программирование можно вести на чистом C++.

Внедрение виджетов Qt в веб-страницы

Набор элементов управления в языке HTML довольно ограничен. Решать эту проблему можно по-разному: путем размещения контента в патентованных форматах типа Flash, или за счет применения нестандартных расширений браузера, имеющихся, например, для Internet Explorer, или путем внедрения Java-апплетов. Во всех случаях требуется поддержка со стороны браузера, а при разработке кросс-платформенных приложений это не всегда возможно. Другой недостаток нестандартных форматов или расширений состоит в том, что мы ограничены реализованной в них функциональностью.

Альтернативное решение – внедрить виджеты Qt. Преимущества подобного решения в том, что мы получаем полный контроль над поведением и внешним видом внедряемых виджетов, и ничто не мешает нам реализовать в них любую необходимую функциональность. Но у этого подхода есть и недостаток – браузер должен поддерживать внедрение виджетов Qt.

В этом подразделе мы рассмотрим игру Matrix Quiz (Викторина с матрицами), показанную на рис. 1.12. Эта страница внедрена в компонент «окно браузера» и является составной частью приложения Matrix Quiz (matrixquiz).

Рис. 1.12. Страница с игрой Matrix Quiz

На странице показано две случайным образом сгенерированных матрицы 3×3, а пользователю предлагается найти их сумму, вписав правильные значения в третью матрицу, которая первоначально заполнена нулями. При нажатии кнопки New генерируется новая пара матриц, а при нажатии Submit введенные в третью матрицу значения проверяются и неправильные выделяются красным цветом, как, например, клетки, содержащие значения 181 и 187 на рис. 1.12. Текущая клетка отобража-

ется инверсными цветами (например, белый на черном фоне), и вокруг нее рисуется прямоугольник, как вокруг любого виджета, имеющего фокус; на рисунке текущей является клетка, содержащая число 116.

На этой странице встречаются как HTML-элементы, так и виджеты Qt, стандартные и написанные нами. Текст заголовка, метка «Name:», а также крупные знаки «+» и «=» – это стандартные HTML-элементы. Поле для ввода имени, кнопки и метку, в которой выводится результат, тоже можно было бы сделать обычными HTML-элементами, но в педагогических целях мы воспользовались стандартными виджетами Qt. Матрицы представлены пользовательскими виджетами (простым подклассом `QTableWidget`), так как чистого HTML для их реализации недостаточно¹.

Компонент «окно браузера» в своем первоначальном виде не поддерживает внедрение виджетов Qt. Чтобы добавить такую поддержку, мы просто создали подкласс `QWebPage`, обладающий необходимым функционалом, и передали его окну браузера, не позволяя ему самостоятельно создавать экземпляр стандартного класса `QWebPage`.

Рассмотрение кода мы начнем с группы из трех фрагментов, взятых из функции `main()` приложения `Matrix Quiz`.

```
qsrand(static_cast<uint>(time(0)));

QWebSettings *webSettings = QWebSettings::globalSettings();
webSettings->setAttribute(QWebSettings::AutoLoadImages, true);
webSettings->setAttribute(QWebSettings::JavascriptEnabled, true);
webSettings->setAttribute(QWebSettings::PluginsEnabled, true);

QString url = QUrl::fromLocalFile(AQP::applicationPathOf() +
                                   "/matrixquiz.html").toString();
BrowserWindow *browser = new BrowserWindow(url, new WebPage);
browser->showToolBar(false);
browser->enableActions(false);
```

Глобальная функция `qsrand()` необходима для инициализации генератора случайных чисел. Без нее вызовы `qrand()` всегда возвращали бы одну и ту же последовательность случайных чисел (поскольку начальное значение равно 1, если явно не переустановить его). Мы предпочитаем пользоваться для генерации псевдослучайных чисел функциями Qt, поскольку альтернативы реализованы не на всех платформах (в частности, отсутствуют в некоторых встраиваемых системах). Кроме того, хотя сейчас это несущественно, включенные в Qt функции потокобезопасны. (Глобальные функции, в том числе `qsrand()`, перечислены в табл. 1.2.)

¹ При этом, конечно, их можно реализовать средствами HTML, JavaScript и CSS. – *Прим. науч. ред.*

Таблица 1.2. Глобальные вспомогательные функции Qt

Функция и пример	Описание
<code>u = qAbs(n);</code>	Возвращает абсолютное значение <i>n</i>
<code>x = qBound(min, n, max);</code>	Возвращает <i>n</i> , если $\text{min} \leq n \leq \text{max}$; иначе возвращает <i>min</i> , если $n < \text{min}$; иначе возвращает <i>max</i>
<code>qDebug("%d: %s", integer, qPrintable(string));</code>	Выводит на консоль простые типы данных C++, следуя синтаксису функции <code>printf()</code> ; не понимает типов Qt (на платформе Windows <i>pro</i> -файл должен содержать строку <code>CONFIG += console</code>)
<code>QDebug() << number << string << hash << stringlist << map << variant << object;</code>	Выводит на консоль простые типы данных C++ и произвольные объекты <code>QObject</code> , в том числе коллекции, например <code>QHash</code> и <code>QMap</code> ; требует наличия директивы <code>#include <QDebug></code> (на платформе Windows <i>pro</i> -файл должен содержать строку <code>CONFIG += console</code>)
<code>b = qFuzzyCompare(f, g);</code>	Возвращает <code>true</code> , если числа с плавающей точкой (или объекты типа <code>QTransform</code> в Qt 4.6) <i>f</i> и <i>g</i> можно считать равными
<code>x = qMax(n, m);</code>	Возвращает наибольшее из чисел <i>n</i> и <i>m</i>
<code>x = qMin(n, m);</code>	Возвращает наименьшее из чисел <i>n</i> и <i>m</i>
<code>const char *s = qPrintable(qstring);</code>	Возвращает для заданного объекта <code>QString</code> значение типа <code>char*</code> (в локальной 8-битной кодировке), пригодное для передачи в <code>printf()</code> или <code>QDebug()</code>
<code>x = qRound(f);</code>	Возвращает значение <i>f</i> , округленное до ближайшего целого (в виде <code>int</code>)
<code>x = qRound64(f);</code>	Возвращает значение <i>f</i> , округленное до ближайшего целого (в виде <code>qint64</code>)
<code>x = qrand();</code>	Возвращает псевдослучайное число в диапазоне от 0 до <code>RAND_MAX</code> (эта константа определена в <code><stdlib></code>). Функция потокобезопасна, а в качестве начального значения используется 1; для установки другого начального значения следует вызвать <code>qsrand()</code>
<code>qsrand(u);</code>	Инициализирует генератор псевдослучайных чисел значением <code>uint u</code>
<code>s = qVersion();</code>	Возвращает строку типа <code>const char*</code> , содержащую номер версии Qt, с которой собрано приложение (например, "4.6.2")

Чтобы можно было пользоваться языком JavaScript, мы должны активировать его. Точно так же, чтобы иметь возможность работать с внед-

ренными виджетами, следует активировать поддержку подключаемых модулей (плагинов).

Обычно класс `QUrl` применяется для создания URL-адресов веб-страниц в Интернете, но здесь мы с его помощью создали URL страницы в локальной файловой системе, используя схему `file://`.

При создании экземпляра `BrowserWindow` мы передаем не только URL нужной нам страницы (в виде строки), но еще и экземпляр написанного нами класса `WebPage`. Кроме того, мы скрываем панель инструментов окна и отключаем для него все действия – теперь это окно невозможно использовать в качестве браузера общего назначения, оно позволяет взаимодействовать только с указанной страницей.

Рассмотрение остального кода мы начнем с небольшого класса `WebPage`, который предоставляет поддержку для встраивания виджетов Qt. Затем мы рассмотрим класс `MatrixWidget` и напоследок на примере страницы `matrixquiz.html` покажем, как внедряются виджеты и как можно использовать JavaScript для реализации некоторой части функциональности страницы.

Класс `WebPage` является подклассом `QWebPage`. Его конструктор (не показан) просто передает необязательный аргумент `parent` конструктору базового класса, само же тело пусто. В классе переопределен только один защищенный метод `createPlugin()`, но перед тем, как заняться им, приведем фрагмент файла `matrixquiz.html`, в котором показано создание одной из кнопок.

```
<object type="application/x-qt-plugin" classid="QPushButton"
      id="newButton" height="40" width="100">
  Can't load QPushButton plugin!</object>
```

Всякий раз, как на странице, хранящейся в `QWebPage`, встречается HTML-тег `<object>`, в котором атрибут `type` равен `application/x-qtplugin`, вызывается метод `createPlugin()`. Эта процедура изображена на рис. 1.13.

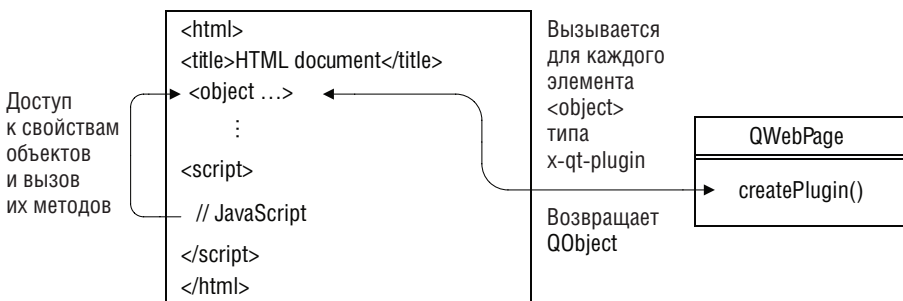


Рис. 1.13. Доступ к объектам `QObject` из JavaScript-сценария, внедренным в HTML-документ

```

QObject* WebPage::createPlugin(const QString &classId,
                               const QUrl&, const QStringList &parameterNames,
                               const QStringList &parameterValues)
{
    QWidget *widget = 0;
    if (classId == "MatrixWidget") {
        widget = new MatrixWidget(view());
        int index = parameterNames.indexOf("readonly");
        if (index > -1)
            static_cast<MatrixWidget*>(widget)->setReadOnly(
                static_cast<bool>(parameterValues[index].toInt()));
    }
    else {
        QUiLoader loader;
        widget = loader.createWidget(classId, view());
    }
    if (widget) {
        int index = parameterNames.indexOf("width");
        if (index > -1)
            widget->setMinimumWidth(parameterValues[index].toInt());
        index = parameterNames.indexOf("height");
        if (index > -1)
            widget->setMinimumHeight(parameterValues[index].toInt());
    }
    return widget;
}

```

Реализация этого метода в классе `QWebPage` возвращает `0`, и тогда выводится не объект, а текст внутри тега `<object>` (если таковой имеется).

В аргументе `classId` передается имя класса объекта, заданное в атрибуте `classid` тега `<object>`. Все остальные атрибуты этого тега передаются в двух параллельных списках строк: в первом – имена атрибутов, во втором – соответствующие значения. Так, для показанного выше тега `<object>`, описывающего кнопку, список `parameterNames` будет содержать `["id", "height", "width"]`, а список `parameterValues` – `["newButton", "40", "100"]`.

Сначала метод создает запрошенный виджет, делая его дочерним для объекта `QWebView`, связанного с `WebPage`. Все отличные от `null` виджеты, возвращенные методом `createPlugin()`, отображаются на HTML-странице объектом `QWebView`, ассоциированным с `QWebPage`.

Если класс запрошенного виджета – `MatrixWidget`, то мы создаем экземпляр этого класса, и если в списке `parameterNames` имеется элемент `"readonly"`, то вызываем метод `MatrixWidget::readOnly()`, передавая ему булево значение. (В HTML-разметке первые две матрицы сделаны доступными только для чтения, а третья – для чтения и записи.)

Для любого другого запрошенного виджета, то есть для всех стандартных виджетов Qt, можно было бы применить ту же технику, что и для `MatrixWidgets`. Но это привело бы к очень длинной последовательности

операторов `if ... else` для проверки значений `classId`, и сопровождать такой код было бы трудно.

К счастью, в Qt уже есть класс, который умеет создавать экземпляры стандартных виджетов Qt по их именам классов; этот класс называется `QUiLoader`. Изначально он предназначался для поддержки динамической загрузки и отображения *ui*-файлов, создаваемых конструктором интерфейсов *Qt Designer*, но здесь мы воспользовались входящим в него методом `QUiLoader::createWidget()`, чтобы создать запрошенный в теге `<object>` виджет и вернуть указатель на него. (Отметим, что для использования класса `QUiLoader` необходимо включить в *pro*-файл строку `CONFIG += uitools`.)

Создав виджет, мы устанавливаем его минимальную ширину и высоту, если они указаны в атрибутах тега `<object>`. В коде на C++ мы не используем атрибут `id`, но он будет нужен в JavaScript-сценарии, реализующем те аспекты поведения веб-страницы, которые выходят за рамки функциональности `MatrixWidget` и других классов Qt.

Мы хотим, чтобы наш класс `MatrixWidget` можно было программировать из кода на JavaScript. Как и в случае `QtScript`, JavaScript-сценарию доступны все свойства класса типа `Q_PROPERTY` – в виде свойств JavaScript-объектов, а также все открытые слоты – в виде методов JavaScript-объектов. Вот как выглядит определение `MatrixWidget` в заголовочном файле:

```
class MatrixWidget : public QTableWidgetItem
{
    Q_OBJECT

public:
    explicit MatrixWidget(QWidget *parent=0);

public slots:
    void clearMatrix();
    void repopulateMatrix();
    QString valueAt(int row, int column) const
        { return item(row, column)->text(); }
    void setValueAt(int row, int column, const QString &value)
        { item(row, column)->setText(value); }
    void setHighlighted(int row, int column, bool highlight=true)
        { item(row, column)->setBackground(highlight ? Qt::red : Qt::white); }
    void setReadOnly(bool read_only);
};
```

Мы определили несколько открытых слотов – тех, что будут доступны из JavaScript-сценария, но создавать какие-то специальные свойства не понадобилось, потому что определенные в базовом классе свойства `rowCount` и `columnCount` и так уже доступны JavaScript-сценарию, а больше нам ничего и не нужно. Метод `setHighlighted()` устанавливает цвет фона элементов таблицы, он используется для выделения клеток с неправильными значениями.


```

const int ColumnWidth = 40;

MatrixWidget::MatrixWidget(QWidget *parent)
    : QTableWidget(3, 3, parent)
{
    verticalHeader()->hide();
    horizontalHeader()->hide();
    for (int row = 0; row < rowCount(); ++row) {
        for (int column = 0; column < columnCount(); ++column) {
            QTableWidgetItem *item = new QTableWidgetItem("0");
            item->setTextAlignment(Qt::AlignCenter);
            setItem(row, column, item);
            if (row == 0)
                setColumnWidth(column, ColumnWidth);
        }
    }
}

```

В конструкторе мы создаем объект `QTableWidget` с фиксированным числом строк и столбцов и с фиксированной шириной столбца. Все клетки инициализируются строкой «0», которая выравнивается по центру. Заголовки строк и столбцов скрыты, поэтому виджет выглядит несколько иначе, чем стандартная таблица `QTableWidget`.

```

void MatrixWidget::setReadOnly(bool read_only)
{
    setEditTriggers(read_only ? QAbstractItemView::NoEditTriggers
                     : QAbstractItemView::AllEditTriggers);
    setFocusPolicy(read_only ? Qt::NoFocus : Qt::WheelFocus);
}

```

Если для виджета задан режим «только для чтения», то мы отключаем все триггеры редактирования. Кроме того, мы изменяем политику захвата фокуса, так что виджет в режиме чтения не может получить фокус клавиатуры; это означает, что при нажатии клавиши `Tab` в момент, когда фокусом владеет виджет, предшествующий матрице в режиме чтения, фокус будет передан следующему за ней виджету, способному его принять.

В этом примере фокус в начале принадлежит полю ввода «name»; когда пользователь нажимает клавишу `Tab`, фокус передается матрице в режиме чтения-записи, куда следует вводить ответы, а обе промежуточные матрицы, находящиеся в режиме чтения, пропускаются.

```

void MatrixWidget::repopulateMatrix()
{
    for (int row = 0; row < rowCount(); ++row) {
        for (int column = 0; column < columnCount(); ++column)
            item(row, column)->setText(
                QString::number(qrand() % 100));
    }
}

```

Этот метод заполняет виджет случайными целыми числами (представленными в виде строк) в промежутке от 0 до 99. Метод `clearMatrix()` (не показан) очень похож на этот, но записывает в каждую клетку строку «0», а фон делает белым.

Мы рассмотрели весь представляющий интерес код на C++. В файле *matrixquiz.html* находится HTML-разметка, содержащая стандартные HTML-элементы и теги `<object>` для всех виджетов Qt, включая и `MatrixWidget`. Все они устроены по одному и тому же принципу, но, просто чтобы продемонстрировать отсутствие различий между описаниями стандартных и пользовательских виджетов Qt, приведем еще тег `<object>` для первой матрицы `MatrixWidget`:

```
<object type="application/x-qt-plugin" classid="MatrixWidget"
      id="leftMatrix" width="124" height="94" readonly="1">
  Can't load MatrixWidget plugin!</object>
```

Файл *matrixquiz.html* управляет поведением страницы с помощью JavaScript-сценария. Тег `<script>` помещен в конец файла, поскольку сценарию необходим доступ к ранее созданным объектам (то есть элементам `<object>`). В коде определены две функции (мы рассмотрим их ниже), а затем идет завершающий блок:

```
newButton.text = "&New";
submitButton.text = "&Submit";
resultLabel.text = "Enter the answer and click Submit";
repopulateMatrices();
newButton.clicked.connect(repopulateMatrices);
submitButton.clicked.connect(checkAnswer);
nameEdit.setFocus();
```

Java
Script

Одно интересное отличие от C++/Qt заключается в том, что в JavaScript и QtScript соединения между сигналами и слотами задаются с помощью одной из следующих двух синтаксических конструкций:

```
object.signalName.connect(functionName)
object.signalName.connect(otherObject.methodName)
```

Java
Script

Мы не приводим функцию `repopulateMatrices()`, потому что она всего лишь вызывает метод `repopulateMatrix()` для обеих матриц `MatrixWidget`¹, но для полноты картины покажем код функции `checkAnswer()`.

```
function checkAnswer()
{
    var allCorrect = true;
    for (var row = 0; row < leftMatrix.rowCount; ++row) {
        for (var column = 0; column < leftMatrix.columnCount;
            ++column) {
            var highlight = false;
```

Java
Script

¹ И метод `clearMatrix()` для третьей матрицы `MatrixWidget`. – Прим. науч. ред.

Java
Script

```

        if (Number(leftMatrix.valueAt(row, column)) +
            Number(rightMatrix.valueAt(row, column)) !=
            Number(answerMatrix.valueAt(row, column))) {
            highlight = true;
            allCorrect = false;
        }
        answerMatrix.setHighlighted(row, column, highlight);
    }
}
name = nameEdit.text == "" ? "mystery person" : nameEdit.text;
if (allCorrect)
    resultLabel.text = "Yes, " + name + ", that's right!";
else
    resultLabel.text = "No, " + name + ", that's not right.";
}

```

Эта функция обходит все клетки во всех трех матрицах. Если значение в клетке матрицы-ответа правильно, то мы снимаем с нее выделение (устанавливаем белый цвет фона), в противном случае подсвечиваем ее (делаем фон красным). Если хотя бы одно значение неправильно, то в переменную `allCorrect` записывается `false`.

В самом конце мы получаем имя пользователя из виджета `nameEdit` типа `QLineEdit` и помещаем в метку `resultLabel` типа `QLabel` текст, зависящий от того, правильно ли пользователь ответил.

На этом мы завершаем рассмотрение страницы `Matrix Quiz` и кода на `C++/Qt` и `JavaScript`, отвечающего за ее внешний вид и поведение.

Внедрение виджетов `Qt` в веб-страницу – это мощный и эффективный способ расширить пределы возможностей веб-страниц, но для его применения необходимо, чтобы браузер пользователя поддерживал `Qt` или же было установлено специальное приложение с компонентом, подобным разработанному в этой главе, – `browserwindow`. Существует много подходов к созданию гибридных приложений, у каждого из них есть свои плюсы и минусы. В версии `Qt 4.4` спектр возможностей расширился в связи с появлением модуля *QtXmlPatterns*, который поддерживает языки `XQuery` и `XPath`. Новые средства были добавлены и в `Qt 4.6`: класс `QGraphicsWebView` (вариант `QWebView` на основе `QGraphicsItem`, оптимизированный для показа внутри `QGraphicsView`) и класс `QWebElement`, предоставляющий элегантный API по образцу библиотеки `jQuery` для доступа и редактирования элементов DOM во фрейме `QWebFrame`. `Qt` позволяет вам выбрать тот подход, который лучше отвечает конкретной задаче, а это значит, что не придется жертвовать удобством работы, функциональностью или «родным» внешним обликом ради того, чтобы включить в приложение возможность доступа к Интернету.

2

Аудио и видео

- Использование классов `QSound` и `QMovie`
- Мультимедийная библиотека `Phonon`

Библиотека Qt 4 всегда умела воспроизводить звук из *wav*-файлов и показывать анимированные изображения из *gif*- и *mng*-файлов. В Qt 4.4 была добавлена интеграция с мультимедийной библиотекой `Phonon`, которая умеет воспроизводить аудио- (музыку) и видеофайлы (фильмы). Таким образом, интеграция существенно расширила мультимедийные возможности Qt.

Короткие звуки могут давать пользователям полезные звуковые подсказки, хотя полагаться только на них нельзя, так как у некоторых людей есть проблемы со слухом, а другие приглушают или отключают звук в системе. А бывает, что в системе вообще нет ни звуковой карты, ни динамиков. Аналогично, движущиеся изображения могут дать визуальное подтверждение того, что некоторый процесс идет и приложение не зависло. Еще одно преимущество движущихся изображений состоит в том, что с их помощью можно визуально продемонстрировать результат выбора некоторого действия или установки того или иного параметра. Но некоторые пользователи считают движущиеся изображения отвлекающими внимание и даже утомляющими, поэтому использовать их следует осторожно, обязательно давая пользователю возможность в любой момент остановить движение.

В первом разделе этой главы мы расскажем, как с помощью классов `QSound` и `QMovie` добавить в приложение звук и движущиеся изображения.

Для приложений, единственное назначение которых – воспроизводить музыку и видео, мультимедийная библиотека `Phonon` предоставляет гораздо более мощные и гибкие возможности, чем простенькие классы

QSound и QMovie. Во втором разделе мы посмотрим, как можно создать аудио- и видеоплеер с помощью этой библиотеки.

Поддержка мультимедиа в Qt реализована на всех платформах с одним и тем же API. Однако для воспроизведения мультимедийных файлов часто нужны кодеки и сторонние библиотеки, которые могут быть не установлены на компьютере конечного пользователя, например из-за мешающих свободной конкуренции и связывающих по рукам и ногам условий патентов на программное обеспечение¹. В настоящее время у этой проблемы нет приемлемого решения, поэтому конкретный аудио- или видеофайл может воспроизводиться на одной платформе и не воспроизводиться на другой. Самый правильный путь – использовать открытые мультимедийные форматы типа *ogg* (www.xiph.org/ogg), хотя в настоящее время на платформах, составляющих собственность какой-то компании (например, Windows), они поддерживаются хуже, чем на открытых платформах типа Linux или BSD.

Использование классов QSound и QMovie

Для воспроизведения коротких звуков и показа анимированных изображений (например, в формате *gif* или *mng*) проще всего воспользоваться классами QSound и QMovie. В том и в другом среди прочих имеются методы `play()` (в случае QMovie он называется `start()` и `stop()`). Кроме того, QMovie генерирует ряд сигналов, включая `stateChanged()` (с аргументом, принимающим значение `QMovie::NotRunning`, `QMovie::Paused` или `QMovie::Running`), а также `frameChanged()` и `finished()`.

Чтобы продемонстрировать использование этих классов в контексте, рассмотрим приложение *Movie Jingle* (`moviejingle`), показанное на рис. 2.1. Оно умеет загружать и отображать анимированные изображения и захватывать отдельные кадры, которые можно сохранить в любом из поддерживаемых Qt растровых форматов. Помимо этого, приложение воспроизводит короткий звук (*jingle* – звяканье) при вызове любого действия и позволяет включать и отключать звуковое сопровождение.

Для ассоциирования звуков с действиями мы создадим подкласс `QAction` под названием `JingleAction`, в котором в дополнение к базовому классу будут храниться путь к аудиофайлу и статическая булевая переменная, показывающая, надо воспроизводить звук или нет. Основное преимущество наследования от `QAction` в данном случае состоит в том, что звук воспроизводится (если не отключен) вне зависимости от того,

¹ Если вы хотите узнать больше о патентах на ПО, то можете посмотреть видео на сайте patentabsurdity.com, посетить посвященную патентам страницу Лиги за свободу программирования по адресу progfree.org/Patents/patents.html или прочитать выступление Ричарда Стеллмена «*The Dangers of Software Patents*» (Опасность патентов на ПО) по адресу www.ifso.ie/documents/rms-2004-05-24.html.



Рис. 2.1. Приложение *Movie Jingle*

как именно вызвано действие (нажатием горячей клавиши, выбором пункта меню или нажатием кнопки на панели инструментов).

Класс `JingleAction` предлагает те же конструкторы, что и `QAction`, и еще два, принимающих имя аудиофайла. Оба они реализованы непосредственно в заголовочном файле *jingleaction.hpp*:

```
QString jingleFile() const { return m_jingleFile; }
static void setMute(bool mute) { s_mute = mute; }
```

Закрывающая переменная `m_jingleFile` имеет тип `QString`. Закрывающая статическая переменная `s_mute` имеет тип `bool` и инициализируется значением `false` в файле *jingleaction.cpp*.

Конструкторы, не принимающие имя файла, просто передают свои аргументы конструктору базового класса, их тела пусты. Что же касается остальных конструкторов, то они тоже передают аргумент конструктору базового класса и еще вызывают один метод в своем теле. Вот пример, взятый из файла *jingleaction.cpp*:

```
JingleAction::JingleAction(const QString &jingleFile,
                           const QString &text, QObject *parent)
    : QAction(text, parent)
{
    setJingleFile(jingleFile);
}
```

Помимо этого реализован новый метод `setJingleFile()` и закрытый слот `play()`.

```
void JingleAction::setJingleFile(const QString &jingleFile)
{
    if (!m_jingleFile.isEmpty())
        disconnect(this, SIGNAL(triggered(bool)), this, SLOT(play()));
    m_jingleFile = jingleFile;
    if (!m_jingleFile.isEmpty())
```

```

        connect(this, SIGNAL(triggered(bool)), this, SLOT(play()));
    }

```

Разрывать и повторно устанавливать соединение действия с сигналом `triggered()`, строго говоря, необязательно, но мы предпочитаем не генерировать сигнал, если в этом нет необходимости. Метод `clearJingle()` не предоставляется, так как вполне достаточно вызова `setJingleFile(QString())`.

```

void JingleAction::play()
{
    if (!s_mute && !m_jingleFile.isEmpty())
        QSound::play(m_jingleFile);
}

```

Здесь мы проигрываем звук, если воспроизведение не отключено и задан аудиофайл. Длительность звучания лучше делать очень короткой — не более секунды, — чтобы звук был слышен не дольше того времени, которое необходимо для выполнения запрошенного действия.

Ознакомившись с основными частями реализации `JingleAction`, рассмотрим некоторые фрагменты кода, взятые из класса главного окна приложения `Movie Jingle`. Это поможет понять, как используются действия и как обращаться с классом `QMovie`. Начнем с двух перечислений, определенных в заголовочном файле класса главного окна:

```

enum ReloadMode {DontReload, Reload};
enum MovieState {NoMovie, Stopped, Playing};

```

Как эти перечисления применяются, мы увидим далее при обсуждении методов главного окна. Но сначала покажем конструктор главного окна:

```

const int StatusTimeout = AQP::MSecPerSecond * 5;

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), movieState(NoMovie)
{
    movie = new QMovie(this);
    createActions();
    createMenusAndToolBar();
    createWidgets();
    createLayout();
    createConnections();
    AQP::accelerateMenu(menuBar());
    updateUi();
    statusBar()->showMessage(tr("Open a Movie file to start..."),
                             StatusTimeout);
    setWindowTitle(QApplication::applicationName());
}

```

Сначала мы создаем закрытый объект `QMovie` с именем `movie`, с помощью которого будем загружать и воспроизводить анимированные изображения; нам понадобится всего один такой объект. Методы `createMenusAndToolBar()`, `createWidgets()`, `createLayouts()` и `createConnections()` мы не

приводим, потому что они стандартны для Qt. Какие создаются действия, и с какими слотами они связываются, станет понятно, когда мы перейдем к рассмотрению методов приложения; например, действие открытия файла соединяется со слотом `fileOpen()` и т. д.

Однако мы все же приведем фрагмент метода `createActions()`, чтобы показать его отличия от аналогичных методов в других приложениях.

После того как виджеты созданы и соединения установлены, мы автоматически добавляем клавиши-акселераторы ко всем меню и их пунктам (стр. 30), а затем инициализируем пользовательский интерфейс, готовя приложение к взаимодействию с пользователем. Хотя код компоновки не показан, отметим, что в главном окне имеются две расположенные бок о бок метки: в левой показывается анимация, а в правой – ее кадры.

```
void MainWindow::createActions()
{
    jinglePath = AQP::applicationPathOf("jingles");
    imagePath = AQP::applicationPathOf("images");
    fileOpenAction = new JungleAction(
        jinglePath + "/fileopen.wav",
        QIcon(imagePath + "/fileopen.png"), tr("Open..."), this);
    fileOpenAction->setShortcuts(QKeySequence::Open);
    ...
}
```

Мы показали создание только первого действия, потому что почти все остальные устроены аналогично. Исключение составляет действие отключения звука, которое сделано переключаемым и первоначально отключено.

Сначала определяем, в каких каталогах следует искать звуковые файлы и изображения. Обратите внимание, что в путях (они хранятся в закрытых переменных экземпляра, доступных другим методам) употребляются разделители, принятые в UNIX, поскольку они работают на всех платформах.

Звуковые файлы должны располагаться в файловой системе. Они не могут быть Qt-ресурсами, потому что во время работы над этой книгой класс QSound не поддерживал ресурсы. Поэтому мы сохраняем звуковые файлы, а заодно уж – единообразия ради – и изображения в файловой системе, а не komponуем их с приложением в виде ресурсов. (Функцию `AQP::applicationPathOf()` мы рассматривали выше, стр. 49.) Но раз так, то мы должны убедиться, что приложение может найти свои звуковые файлы и изображения во время работы.

Теперь обратимся к реализации тех методов приложения, которые отвечают за открытие, воспроизведение и остановку воспроизведения файла с анимированным изображением, вырезание и сохранение кадра и включение/выключение звука.


```

void MainWindow::fileOpen()
{
    QString fileFormats = AQP::filenameFilter(tr("Movies"),
        QMovie::supportedFormats());
    QString path(movie && !movie->fileName().isEmpty()
        ? QFile::FileInfo(movie->fileName()).absolutePath() : ".");
    QString filename = QFileDialog::getOpenFileName(this,
        tr("%1 - Choose a Movie File")
        .arg(QApplication::applicationName()), path, fileFormats);
    if (filename.isEmpty())
        return;

    movie->setFileName(filename);
    statusBar()->showMessage(tr("Loaded %1").arg(filename), StatusTimeout);
    movieState = Stopped;
    startOrStop(DontReload);
}

```

Статический метод `QMovie::supportedFormats()` **возвращает список** `QList<QByteArray>` **файловых суффиксов, например** `["gif", "mng"]`. **Функция** `AQP::filenameFilter()` **из модуля** `aqp.{hpp,cpp}` **возвращает построенную по такому списку строку** `QString`, **например** `"Movies (*.gif *.mng)"`, **пригодную для передачи функции** `QFileDialog::getOpenFileName()` **в качестве фильтра файлов.**

Когда пользователь выбирает файл, мы копируем его имя и устанавливаем его в качестве имени файла ролика, а потом обновляем строку состояния. Затем устанавливаем состояние анимации и вызываем слот `startOrStop()`, **чтобы сразу же начать воспроизведение. Параметр метода** `startOrStop()` **по умолчанию равен** `Reload`, **но мы явно передали значение** `DontReload`, **потому что ролик уже загружен.**

```

void MainWindow::startOrStop(ReloadMode reloadMode)
{
    if (movieState == Stopped) {
        if (reloadMode == Reload)
            movie->setFileName(movie->fileName());
        movie->start();
        movieState = Playing;
    }
    else {
        movie->stop();
        movieState = Stopped;
    }
    updateUi();
}

```

Метод `QMovie::setFileName()` **следует вызывать, чтобы повторно воспроизвести уже загруженный и проигранный ролик, а для только что загруженного ролика вызывать его не нужно. В зависимости от значения**

переменной movieState ролик либо запускается, либо останавливается, а текущее состояние индицируется с помощью метода updateUi().

```
void MainWindow::updateUi()
{
    if (movieState == Playing) {
        startOrStopAction->setText(tr("&Stop"));
        startOrStopAction->setIcon(QIcon(imagePath + "/editstop.png"));
        startOrStopAction->setJingleFile(jinglePath + "/editstop.wav");
    }
    else {
        startOrStopAction->setText(tr("&Start"));
        startOrStopAction->setIcon(QIcon(imagePath + "/editstart.png"));
        startOrStopAction->setJingleFile(jinglePath + "/editstart.wav");
    }
    startOrStopAction->setEnabled(movieState != NoMovie);
    takeSnapshotAction->setEnabled(movieState != NoMovie);
}
```

В этом методе мы обновляем текст, пиктограмму и файл, ассоциированные с действием startOrStopAction, превращая его либо в действие запуска, либо в действие останова. Если ролика еще нет, например, когда приложение только что запущено, то действия запуска, останова и захвата кадра деактивированы.

```
void MainWindow::takeSnapshot()
{
    snapshot = movie->currentPixmap();
    fileSaveAction->setEnabled(!snapshot.isNull());
    snapshotLabel->setPixmap(snapshot);
}
```

После того как ролик загружен (неважно, воспроизводится он или остановлен), пользователь может сделать копию текущего кадра, для чего служит метод QMovie::currentPixmap(). Если результат отличен от null (а так и должно быть), то мы активируем действие сохранения и в любом случае копируем результат в метку snapshot, чтобы показать изображение из захваченного кадра.

```
void MainWindow::fileSave()
{
    if (snapshot.isNull())
        return;
    QString fileFormats = AQP::filenameFilter(tr("Images"),
        QImageWriter::supportedImageFormats());
    QString filename = QFileDialog::getSaveFileName(this,
        tr("%1 - Save Snapshot")
        .arg(QApplication::applicationName()),
        QFile::FileInfo(movie->fileName()).absolutePath(),
        fileFormats);
    if (filename.isEmpty())
```

```

        return;
    if (!snapshot.save(filename))
        AQP::warning(this, tr("Error"), tr("Failed to save snapshot image"));
    else
        statusBar()->showMessage(tr("Saved %1").arg(filename), StatusTimeout);
}

```

Если захваченный кадр имеется, то мы открываем диалоговое окно `QFileDialog`, установив в качестве фильтра перечень растровых форматов, поддерживаемых Qt. Если пользователь укажет имя файла, то мы попытаемся сохранить кадр в файле с таким именем, поручив методу `QPixmap::save()` самостоятельно определить формат в соответствии с суффиксом. Если произошла ошибка при сохранении, мы выводим ее текст в диалоговом окне, в противном случае информация об успешном завершении операции отображается в строке состояния. (О том, почему для вывода сообщений пользователю мы применяем функцию `AQP::warning()` и ей подобные вместо стандартных статических функций `QMessageBox`, см. врезку «Избегайте статических вспомогательных функций `QMessageBox`» на стр. 77.) Еще один плюс использования функций из модуля AQP состоит в том, что они сами помещают имя приложения (получая его с помощью `QApplication::applicationName()`) в строку заголовка, а нам остается только передать конец заголовка. (Сравните вызовы `QFileDialog::getSaveFileName()` и `AQP::warning()` и почувствуйте разницу.)

```

void MainWindow::muteJingles(bool mute)
{
    JingleAction::setMute(mute);
}

```

Среди прочих действий, создаваемых в конструкторе главного окна, есть и `fileMuteJinglesAction` – переключаемое действие, первоначально выключенное. Оно добавляется в меню и на панель инструментов и соединяется со слотом, а назначение его – позволить пользователю управлять звуковым сопровождением.

Хотя классы `QSound` и `QMovie` весьма полезны для предоставления звуковых и визуальных подсказок, развитыми средствами работы с мультимедиа они не располагают. Но, к счастью, такой функциональностью обладает мультимедийная библиотека `Phonon`, которую мы и рассмотрим в следующем разделе.

Мультимедийная библиотека Phonon

Мультимедийная библиотека `Phonon` была создана разработчиками среды KDE ('K' Desktop Environment) с целью упростить написание мультимедийных приложений для KDE. `Phonon` состоит из двух частей – фронтальной, которая предоставляет обобщенный API для работы с мультимедиа, и движков (backend), которые, собственно, и реализуют мультимедийные сервисы, предоставляемые API. Входящий в со-

став Qt модуль *Phonon* – это тонкая обертка вокруг Phonon API, преобразующая его к виду, принятому в Qt. Для получения доступа к этому модулю необходимо добавить в *pro*-файл проекта строку `QT += phonon`.

Одна из важнейших особенностей библиотеки Phonon заключается в ее кроссплатформенности благодаря поддержке разнообразных движков. В Linux в качестве движка обычно используются библиотеки GStreamer, в Mac OS X – QuickTime, а в Windows – DirectX и DirectShow. Допускается также использование других движков, например VLC и MPlayer (code.google.com/p/phonon-vlc-mplayer). Перед началом сборки Qt платформенно-зависимый движок уже должен быть установлен, с инструкциями можно ознакомиться по адресу qt.nokia.com/doc/phonon-overview.html¹.

Избегайте статических вспомогательных функций QMessageBox

Класс QMessageBox предоставляет несколько статических функций для отображения модальных диалоговых окон, содержащих указанные пользователем сообщение и кнопки. Однако пользователи Mac OS X привыкли к тому, что такого рода сообщения появляются в *листочках*, а не в обычных диалоговых окнах. Листок – это разновидность модального диалогового окна, которое «выплывает» из строки заголовка родительского окна и располагается прямо под ней по центру (как будто это часть окна, причем перемещаемая), тогда как обычное диалоговое окно всплывает поверх родительского и его можно перемещать и изменять его размер.

Чтобы приложения были по-настоящему кросс-платформенными, мы написали собственные вспомогательные функции для вывода сообщений: `AQP::information()`, `AQP::warning()` и `AQP::question()`, а также функции `AQP::okToDelete()`, которая выводит диалоговое окно с кнопками Delete (Удалить) и Do Not Delete (Не удалять) (см. стр. 121), и `AQP::okToClearData()`, выводящую диалоговое окно с кнопками Save (Сохранить), Discard (Не сохранять) и Cancel (Отмена) (см. стр. 156). Обе функции, `AQP::okToDelete()` и `AQP::okToClearData()`, возвращают булево значение, а функция `AQP::okToClearData()` еще и вызывает переданный ей метод сохранения, если пользователь пожелал сохранить данные. Ниже приведен код функции `AQP::warning()`, все остальные аналогичны.

```
void warning(QWidget *parent, const QString &title,
             const QString &text, const QString &detailedText)
```

¹ В настоящее время сборка Qt с движком Windows DirectShow возможна только при использовании коммерческого компилятора.

```
{
    QScopedPointer<QMessageBox> messageBox(new QMessageBox(parent));
    if (parent)
        messageBox->setWindowModality(Qt::WindowModal);
    messageBox->setWindowTitle(QString("%1 - %2")
        .arg(QApplication::applicationName()).arg(title));
    messageBox->setText(text);
    if (!detailedText.isEmpty())
        messageBox->setInformativeText(detailedText);
    messageBox->setIcon(QMessageBox::Warning);
    messageBox->addButton(QMessageBox::Ok);
    messageBox->exec();
}
```

Окно сообщения должно существовать только на протяжении работы этой функции, поскольку после возврата из блокирующего вызова `exec()` оно закрывается. Мы храним указатель на окно в переменной типа `QScopedPointer` (или `QSharedPointer` в версии Qt 4.5); тем самым гарантируется, что указатель будет удален сразу по выходе из области видимости, так что мы можем опустить явный вызов `delete`, не опасаясь утечки памяти. (См. врезку «Интеллектуальные указатели в Qt» ниже.)

Задавать модальность окна `Qt::WindowModal` необходимо для того, чтобы в Mac OS X окно выглядело, как листок, при том что на других платформах это будет обычное диалоговое окно.

Интеллектуальные указатели в Qt

В версии Qt 4.0 появился тип охраняемого указателя `QPointer`, в Qt 4.5 – типы интеллектуальных указателей `QSharedPointer` и `QWeakPointer`, а в Qt 4.6 – тип интеллектуального указателя `QScopedPointer`. Все эти типы обертывают обычные указатели и, как правило, потребляют дополнительную память и замедляют доступ по указателю. Тем не менее интеллектуальные указатели настолько полезны и удобны (и помогают избежать утечек памяти), что применять их имеет смысл.

При программировании с помощью Qt благодаря иерархии владения родитель–потомок нам редко приходится прибегать к интеллектуальным указателям просто потому, что мы почти никогда не применяем оператор `delete`. Однако всякий раз, как возникает необходимость в вызове `delete` или есть шанс, что указатель где-то в другом месте может быть обнулен, стоит подумать об использовании интеллектуального указателя.

Самым употребительным и гибким из всех интеллектуальных указателей является `QSharedPointer`; он ведет себя аналогично классу `std::shared_ptr`, который входит в библиотеку `Boost` и появится в следующей версии стандарта `C++`. К сожалению, API класса `QSharedPointer` не полностью совпадает с `std::shared_ptr`; в частности, для получения обычного указателя, обернутого интеллектуальным, служат методы `QSharedPointer::data()` и `std::shared_ptr::get()` соответственно. Интеллектуальность `QSharedPointer` проявляется в том, что с ним можно обращаться, как с самым обычным указателем, в частности копировать (так что образуются два и более `QSharedPointer`, указывающих на один и тот же объект), но как только последний (или единственный) `QSharedPointer` выйдет из области видимости, для обернутого им простого указателя автоматически будет вызван оператор `delete`.

Если для объекта, выделенного в куче и не являющегося частью иерархии `Qt` родитель–потомок, не использовать интеллектуальный указатель, то за удаление переставшего быть нужным указателя отвечаем мы сами. Поставить оператор `delete` в конце кода, где используется указатель, недостаточно потому, что в случае возникновения исключения мы выйдем из функции, не дойдя до `delete`, и значит, произойдет утечка памяти. Можно, конечно, воспользоваться конструкцией `try ... catch` и поместить `delete` внутри блока `catch`. Но тогда требуется перехватывать все возможные исключения, а применить блок с перехватом всех вообще исключений нельзя, так как мы не хотим, чтобы случайно – и молча – перехватывались неожиданные исключения, скрывая тем самым ошибки программирования.

Правильнее всего применить идиому `RAII` (`Resource Acquisition Is Initialization` – захват ресурса есть инициализация). На практике это означает, что указатель создается внутри конструктора ограниченного (`scoped`) или разделяемого (`shared`) указателя. Тогда нам не придется самим беспокоиться ни об удалении указателя, ни о преждевременном выходе из функции вследствие исключения, поскольку, как только ограниченный указатель (или последний из разделяемых указателей, указывающих на объект) выйдет из области видимости, он автоматически удалит объект, на который указывает.

Применение иерархии `Qt` родитель–потомок наряду с интеллектуальными указателями означает, что в некоторых приложениях от использования оператора `delete` можно полностью отказаться.

Таблица 2.1. Основные классы Phonon

Класс	Описание
Phonon::AudioOutput	Медиаузел приемника аудио, управляющий звуковой картой или наушниками
Phonon::Effect	Медиаузел-процессор, который умеет преобразовывать аудиопоток
Phonon::EffectWidget	Виджет, управляющий параметрами процессора эффектов
Phonon::MediaNode	Базовый класс для всех типов медиаузлов
Phonon::MediaObject	Медиаузел, управляющий воспроизведением мультимедийных объектов
Phonon::MediaSource	Объект, поставляющий мультимедийные данные медиаобъекту, представляющему узел-источник
Phonon::Path	Информационный тракт от медиаобъекта, представляющего узел-источник, к медиаобъекту, представляющему узел-приемник
Phonon::SeekSlider	Виджет для отображения и изменения временной точки воспроизведения медиаобъекта
Phonon::VideoPlayer	Виджет, который умеет загружать и воспроизводить видео и автоматически управляет созданием медиаузлов и трактов
Phonon::VideoWidget	Виджет для воспроизведения видео
Phonon::VolumeSlider	Виджет для отображения и изменения уровня громкости медиаобъекта

В библиотеке Phonon есть три основных вида объектов: источники медийных данных, медиаузлы и медиаустройства. В табл. 2.1 перечислены классы, представляющие эти объекты.

Источники медийных данных представлены объектами `Phonon::MediaSource`, при создании которых указывается файл, URL или объект `QIODevice`, из которого можно получить медийные данные.

Это не медиаузлы как таковые, их можно использовать, только если передать медиаобъекту, представляющему узел-источник.

Медиаузлы бывают трех видов: узлы-источники (не путать с источниками медийных данных), узлы-процессоры и узлы-приемники (sink). Узлы-источники представляются объектами `Phonon::MediaObject` и обеспечивают интерфейс для воспроизведения мультимедиа. Они имеют текущий объект `Phonon::MediaSource`; кроме того, с ними также может быть связана очередь других объектов-источников медийных данных, готовых к последовательному воспроизведению по мере завершения воспроизведения предыдущего объекта.

Выход объекта `Phonon::MediaObject` должен поступать приемнику по одному или нескольким трактам. Тркт представляется объектом `Phonon::Path` и должен иметь узел-источник (`Phonon::MediaObject`) и узел-приемник, например `Phonon::AudioOutput` или `Phonon::VideoWidget`. Тркт может быть прямым или содержать промежуточные узлы-процессоры, которые добавляют специальные эффекты.

Модуль *Phonon* не поддерживает прямую манипуляцию данными в медиапотоках. Однако он предоставляет косвенные средства манипулирования аудиопотоками: процессоры эффектов. Такие процессоры, представленные объектами класса `Phonon::Effect`, можно включать в тркт между источником и приемником, и они будут преобразовывать данные, передаваемые от одного к другому. Набор доступных эффектов зависит от используемого движка *Phonon*, информацию о них можно получить от функции `Phonon::BackendCapabilities::availableAudioEffects()`. К числу эффектов относятся усиление, позиционирование потоков на стереопанораме, выравнивание амплитудно-частотной характеристики (эквалайзер) и повторное сэмплирование.

Чтобы применить эффект, мы создаем экземпляр класса `Phonon::Effect` для нужного эффекта. Затем в предположении, что был сохранен указатель на `Phonon::Path`, возвращенный методом `Phonon::createPath()`, мы вызываем метод `Phonon::Path::insertEffect()`, передавая ему только что созданный объект `Phonon::Effect`.

Узел-приемник — это то место, откуда данные в конечном итоге посылаются физическому устройству вывода, например звуковой карте, наушникам или виджету, для воспроизведения видео. Он может быть представлен медиаузлом `Phonon::AudioOutput` либо виджетами `Phonon::VideoPlayer` или `Phonon::VideoWidget`, которые также играют роль медиаузлов. Общая архитектура показана на рис. 2.2.

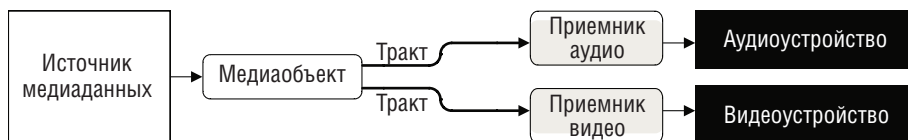


Рис. 2.2. Архитектура *Phonon*

Теперь, разобравшись с теоретическими основами работы библиотеки *Phonon*, посмотрим, как она используется на практике. В следующих подразделах мы рассмотрим два примера: воспроизведение музыки и видео (в том числе и звуковых дорожек, если они присутствуют).

Воспроизведение музыки

На примере программы *Play Music* (`playmusic`), рассматриваемой в этом подразделе и показанной на рис. 2.3, мы продемонстрируем создание

приложения для воспроизведения музыки. Обычно в таких приложениях для организации композиций используются списки воспроизведения, база данных или файловая система; мы остановились на файловой системе. Сразу после запуска центральная область пуста, и программа предлагает пользователю указать каталог, содержащий музыкальные файлы. Получив каталог, программа ищет все музыкальные файлы, находящиеся как в нем самом, так и в его подкаталогах, и заполняет дерево `QTreeWidget`, помещая в него информацию об исполнителе, альбоме, названии и продолжительности композиции.

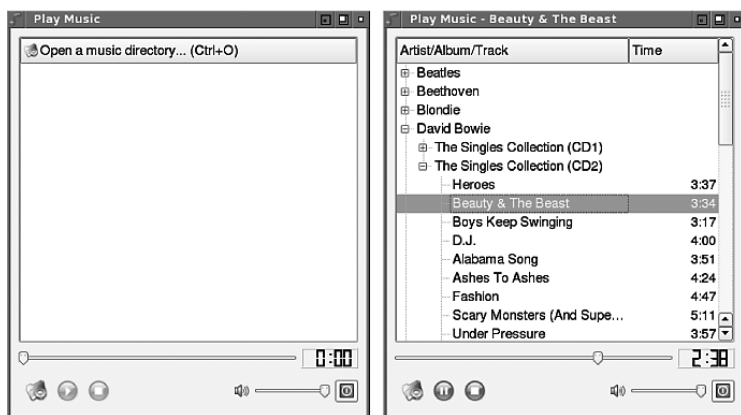


Рис. 2.3. Приложение Play Music сразу после запуска и с заполненным деревом музыкальных композиций

Пользователь может перейти к конкретной композиции с помощью клавиатуры или мыши, воспроизвести или приостановить ее, нажав кнопку Play/Pause на панели инструментов или клавишу пробела. Когда текущая композиция заканчивается, программа автоматически начинает воспроизводить следующую, если только пользователь не нажал кнопку Stop.

Данные приложения хранятся в объекте `QTreeWidget`. Для исполнителей и альбомов сохраняются только имена и названия, а для композиций – название и имя соответствующего файла. Все эти данные представлены объектами класса `TreeWidgetItem`, простенького подкласса `QTreeWidgetItem`, который мы рассмотрим ниже. (`QTreeWidget` – это вспомогательный класс, являющийся частью реализованной в Qt архитектуры модель/представление – представление, хранящее собственную модель. Архитектура модель/представление, в том числе создание и использование пользовательских моделей, рассматривается в главах 3–6.)

Начнем с конструктора главного окна, который поможет нам понять, как устроено приложение.

```
const int FilenameRole = Qt::UserRole;
const int OneSecond = AQP::MSecPerSecond;

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), nextItem(0)
{
    playIcon = QIcon(":/play.png");
    pauseIcon = QIcon(":/pause.png");

    mediaObject = new Phonon::MediaObject(this);
    mediaObject->setTickInterval(OneSecond);
    audioOutput = new Phonon::AudioOutput(Phonon::MusicCategory,
                                          this);
    Phonon::createPath(mediaObject, audioOutput);

    createActions();
    createToolBar();
    createWidgets();
    createLayout();
    createConnections();

    setWindowTitle(QApplication::applicationName());
}
```

Здесь `nextItem` — объект типа `TreeWidgetItem`, представляющий следующую композицию. Значение ему присваивается, когда пользователь выбирает композицию и когда текущая композиция завершается. Элементы дерева могут содержать несколько столбцов, а в каждом столбце могут храниться пользовательские данные в виде списка объектов `QVariant`. Ниже мы воспользуемся этим для того, чтобы хранить имя файла композиции в пользовательских данных первого столбца (с индексом 0), указывая в качестве индекса в пользовательских данных константу `FilenameRole`.

Закрытая переменная `mediaObject` используется при воспроизведении выбранной пользователем композиции, а приемник, в который посылаются аудиоданные, представлен переменной `audioOutput`. Для аудиоданных необходимо указать категорию устройства вывода. Например, в случае телефонного приложения, работающего по протоколу VoIP (Voice over Internet Protocol), звук направляется в наушники, тогда как для музыкального проигрывателя — на выход звуковой карты.

Имея медиаобъект и приемник, мы можем создать соединяющий их тракт, а поскольку мы не планируем включать в этот тракт процессоры для добавления специальных эффектов, то ссылку на созданный тракт можно не сохранять. Кроме того, мы настраиваем медиаобъект так, чтобы он генерировал сигнал `tick()` один раз в секунду (1000 мс); это понадобится для обновления цифрового дисплея, на котором показывается положение текущей точки воспроизведения во времени.

На методах `createActions()`, `createToolBar()` и `createLayout()` мы задерживаться не станем, так как они не таят никаких сюрпризов. Однако приведем фрагмент метода `createWidgets()`, в котором создаются виджеты `Phonon`, а также остановимся на методе `createConnections()`.

```
void MainWindow::createWidgets()
{
    seekSlider = new Phonon::SeekSlider(this);
    seekSlider->setToolTip(tr("Playback Position"));
    seekSlider->setMediaObject(mediaObject);
    volumeSlider = new Phonon::VolumeSlider(this);
    volumeSlider->setToolTip(tr("Volume Control"));
    volumeSlider->setAudioOutput(audioOutput);
    volumeSlider->setSizePolicy(QSizePolicy::Maximum,
                               QSizePolicy::Maximum);
    ...
}
```

Ползунок позиционирования (`seekSlider`) — это верхний ползунок на рис. 2.3. Он служит для того, чтобы дать визуальное представление о том, какая часть композиции уже воспроизведена. Кроме того, он позволяет осуществлять перемотку вперед и назад. Этот ползунок связан с медиа-объектом и потому может показывать и изменять положение точки воспроизведения. Ползунок управления громкостью (`volumeSlider`) вместе с кнопкой выключения звука служит для регулирования громкости звука. Он связан с приемником аудиоданных.

Метод `createConnections()` мы рассмотрим в два приема.

```
void MainWindow::createConnections()
{
    connect(mediaObject, SIGNAL(tick(qint64)), this, SLOT(tick(qint64)));
    connect(mediaObject,
            SIGNAL(stateChanged(Phonon::State, Phonon::State)),
            this, SLOT(stateChanged(Phonon::State)));
    connect(mediaObject, SIGNAL(aboutToFinish()),
            this, SLOT(aboutToFinish()));
    connect(mediaObject,
            SIGNAL(currentSourceChanged(const Phonon::MediaSource&)),
            this, SLOT(currentSourceChanged()));
}
```

Первые четыре соединения связывают медиаобъект со слотами главного окна. Сигнал `tick()` служит для обновления виджета, имитирующего цифровой дисплей, в котором показывается текущее положение точки воспроизведения во времени. Соединение с сигналом `stateChanged()` используется для того, чтобы реагировать на изменения состояния; например, оно позволяет активировать и деактивировать элементы управления. Этот сигнал несет с собой как новое (текущее), так и старое (предыдущее) состояние, но в данном приложении нас интересует лишь новое состояние. Соединение с сигналом `aboutToFinish()` используется для того, чтобы добавить следующую композицию в очередь медиаобъекта и тем

самым обеспечить плавный переход от одной композиции к другой. Сигнал `currentSourceChanged()` генерируется, когда изменяется источник данных для медиаобъекта, то есть загружена новая композиция. Мы используем его для обновления пользовательского интерфейса.

```
connect(setMusicDirectoryAction, SIGNAL(triggered()),
        this, SLOT(setMusicDirectory()));
connect(playOrPauseAction, SIGNAL(triggered()),
        this, SLOT(playOrPause()));
connect(stopAction, SIGNAL(triggered()), this, SLOT(stop()));
connect(treeWidget,
        SIGNAL(currentItemChanged(QTreeWidgetItem*, QTreeWidgetItem*)),
        this, SLOT(currentItemChanged(QTreeWidgetItem*)));
connect(treeWidget,
        SIGNAL(itemDoubleClicked(QTreeWidgetItem*, int)),
        this, SLOT(playTrack(QTreeWidgetItem*)));
connect(quitAction, SIGNAL(triggered()), this, SLOT(close()));
}
```

Остальные соединения обеспечивают общую функциональность пользовательского интерфейса: задать каталог, в котором ищутся музыкальные композиции, начать, приостановить или прекратить воспроизведение, выбрать новую композицию или завершить работу приложения.

Теперь обратимся к слотам, так как все они имеют отношение к работе с библиотекой Phonon. Начнем с самого сложного, `setMusicDirectory()`, поскольку он показывает, как заполняется дерево и как можно использовать временный медиаобъект для извлечения информации о композиции. Мы рассмотрим его код в три приема, а затем ознакомимся с используемыми в нем вспомогательными методами.

```
void MainWindow::setMusicDirectory()
{
    QString path = QFileDialog::getExistingDirectory(this,
        tr("Choose a Music Directory"),
        QDesktopServices::storageLocation(
            QDesktopServices::MusicLocation));
    if (path.isEmpty())
        return;
}
```

Сначала мы просим пользователя указать каталог с музыкальными композициями, по умолчанию предлагая тот, который возвращает статический метод `QDesktopServices::storageLocation()`. Например, в Windows это может быть каталог `%HOMEPATH%\My Documents\My Music` или `%USERPROFILE%\My Documents\My Music` (вместо переменной окружения подставляется конкретный путь).

```
QApplication::setOverrideCursor(QCursor(Qt::WaitCursor));
QSet<QString> validSuffixes = getSuffixes();
treeWidget->clear();
treeWidget->headerItem()->setIcon(0, QIcon());
treeWidget->setHeaderLabels(QStringList()
```

```

    << tr("Artist/Album/Track") << tr("Time"));
    QHash<QString, TreeWidgetItem*> itemForArtist;
    QHash<QString, TreeWidgetItem*> itemForArtistAlbum;
    QDirIterator i(path, QDirIterator::Subdirectories);
    while (i.hasNext()) {
        const QString filename = i.next();
        if (!QFileInfo(filename).isFile() ||
            !validSuffixes.contains(QFileInfo(filename).suffix()))
            continue;
        addTrack(filename, &itemForArtist, &itemForArtistAlbum);
    }

```

Первым делом мы получаем набор файловых суффиксов, соответствующих форматам музыкальных файлов, которые поддерживаются движком Phonon. Создаем два хеша: один – для сопоставления элемента дерева, описывающего исполнителя, с его именем, второй – для сопоставления элемента дерева, описывающего альбом, с именем исполнителя и названием альбома. (Необходимо объединять имя исполнителя и название альбома во избежание конфликтов в тех редких случаях, когда альбомы разных исполнителей называются одинаково.)

Перед тем как приступить к систематизации музыкальных файлов, мы создаем итератор `QDirIterator`; подобные объекты дают очень удобный способ перебора всех файлов в каталоге, они автоматически рекурсивно просматривают подкаталоги и сообщают об имени каждого обнаруженного файла, каталога и прочих объектов файловой системы. Для каждого объекта, являющегося файлом (а не каталогом, ссылкой или еще чем-то) с подходящим суффиксом, мы вызываем метод `addTrack()`, который добавляет информацию об этом файле в дерево.

В классе `QFileInfo` имеются такие методы, как `isFile()` и `isDir()`, позволяющие определить вид объекта файловой системы с данным именем, а также методы, возвращающие компоненты пути к файлу: `absolutePath()`, `fileName()` и `suffix()`. Есть также методы, возвращающие сведения о правах доступа, например `isReadable()` и `isWritable()`, и о различных аспектах состояния: размере (`size()`), дате и времени создания (`created()`), дате и времени последней модификации (`lastModified()`).

```

foreach (QTreeWidgetItem *item, itemForArtistAlbum)
    if (!item->childCount())
        delete item;
foreach (QTreeWidgetItem *item, itemForArtist)
    if (!item->childCount())
        delete item;
treeWidget->sortItems(0, Qt::AscendingOrder);
treeWidget->resizeColumnToContents(0);
stop();
QApplication::restoreOverrideCursor();
}

```

В конце производится очистка. Например, если в альбоме нет ни одной композиции, то альбом удаляется. И аналогично, если у исполнителя нет ни одного альбома, то удаляется исполнитель. Затем мы сортируем элементы дерева и устанавливаем ширину первого столбца. Кроме того, мы прекращаем воспроизведение, так как дерево полностью перестроено, а значит, может оказаться, что композиция, которую мы перед этим слушали, более не присутствует в пользовательском интерфейсе.

По умолчанию, если дерево отсортировано по первому столбцу, то сортировка применяется на каждом его уровне, при этом строки сортируются с учетом регистра. Но мы хотим отсортировать элементы иначе и поэтому заполнили дерево объектами класса `TreeWidgetItem`, а не стандартного класса `QTreeWidgetItem` (которому `TreeWidgetItem` наследует). Конструктор `TreeWidgetItem` (не показан) просто передает свои аргументы конструктору базового класса, его тело пусто. Мы изменили только реализацию встраиваемой функции-члена `operator<()`:

```
bool operator<(const QTreeWidgetItem &other) const
{
    QString left = data(0, FilenameRole).toString();
    QString right;
    if (!left.isEmpty())
        right = other.data(0, FilenameRole).toString();
    else {
        left = text(0);
        right = other.text(0);
    }
    return QString::compare(left, right, Qt::CaseInsensitive) < 0;
}
```

Метод `operator<()` *обязан* определять полное упорядочение, иначе дерево будет работать некорректно. На первый взгляд кажется, что приведенный код не удовлетворяет этому условию, однако это не так. На каждом уровне дерева сортировка потомков родительского узла производится независимо (узлы верхнего уровня расположены под невидимым корневым узлом). Следовательно, в данном случае сначала сортируются все исполнители, затем для каждого исполнителя – его альбомы и, наконец, для каждого альбома – вошедшие в него композиции. Для исполнителя и альбома никаких пользовательских данных нет, поэтому сравниваются тексты, а для композиции сравниваются пользовательские данные (имена файлов). Эта схема дает прекрасные результаты, если файлы именуются так, что в начале идет номер композиции – *01-Space Oddity.ogg*, *02-Changes.ogg*, *03-Starman.ogg* и т. д., – и при этом обеспечивает полное упорядочение на каждом уровне иерархии.

Сделаем еще два замечания, касающихся эффективности. Мы проверяем, что пользовательские данные отсутствуют только для одного из сравниваемых элементов, поскольку если сравниваются исполни-

тели или альбомы, то данные в другом элементе заведомо отсутствуют. И вместо того чтобы выполнять сравнение в виде `return left.toLower() < right.toLower()`, мы пользуемся гораздо более быстрой функцией `QString::compare()` (которая возвращает `int`) и преобразуем результат к требуемому типу `bool`.

Теперь рассмотрим два вспомогательных метода, вызываемых из `setMusicDirectory()`.

```
QSet<QString> MainWindow::getSuffixes()
{
    QStringList mimeTypeTypes;
    foreach (const QString &mimeType,
        Phonon::BackendCapabilities::availableMimeTypes())
    if (mimeType.startsWith("audio/"))
        mimeTypeTypes << mimeType;
    return AQP::suffixesForMimeTypes(mimeTypeTypes);
}
```

В библиотеке Phonon формат данных опознается по типу MIME. Функция `AQP::suffixesForMimeTypes()` из модуля `aqp.{hpp,cpp}` принимает список типов MIME и возвращает набор соответствующих файловых суффиксов. Выглядит это просто и элегантно, но в действительности все несколько сложнее (см. врезку «Типы MIME, файловые суффиксы и магические числа» чуть ниже).

Рассмотрение метода `addTrack()` мы проведем в два приема.

```
void MainWindow::addTrack(const QString &filename,
    QHash<QString, TreeWidgetItem*> *itemForArtist,
    QHash<QString, TreeWidgetItem*> *itemForArtistAlbum)
{
    Phonon::MediaObject localMediaObject;
    Phonon::MediaSource source(filename);
    localMediaObject.setCurrentSource(source);
    if (!waitForMediaObjectToLoad(&localMediaObject, OneSecond))
        return;
    QString artist = localMediaObject.metadata(
        Phonon::ArtistMetaData).join("/").trimmed();
    QString album = localMediaObject.metadata(
        Phonon::AlbumMetaData).join("/").trimmed();
    QString artistAlbum = artist + "\\t" + album;
    QString track = localMediaObject.metadata(
        Phonon::TitleMetaData).join("/").trimmed();
    qint64 msec = localMediaObject.totalTime();
}
```

Локальный медиаобъект `localMediaObject` нужен исключительно для получения информации (метаданных) о композициях. Мы начинаем с того, что указываем переданное нам имя файла в качестве источника для данного медиаобъекта. Затем медиаобъект начнет загружать музыкальный файл. На получение метаданных из файла уходит пусть небольшое, но все же ненулевое время, а в некоторых случаях метаданные

нельзя извлечь вовсе – например, если файл поврежден. Для обработки таких ситуаций мы решили организовать локальный цикл сообщений с тайм-аутом; как работает этот механизм, мы рассмотрим ниже при обсуждении метода `waitForMediaObjectToLoad()`.

Мы извлекаем из музыкального файла метаданные и объединяем списки строк `QStringList`, возвращаемые каждым вызовом `QMediaObject::metaData()`, в одну строку `QString`, разделяя их косой чертой. Метод `QMediaObject::totalTime()` возвращает длительность звучания каждой композиции в миллисекундах. Чего мы не извлекаем, так это метаданных о номере композиции (`Phonon::TracknumberMetaData`); зачастую это поле пусто, поэтому мы решили сортировать композиции по именам файлов, о чем уже было сказано выше.

Типы MIME, файловые суффиксы и магические числа

Существует три общеупотребительных метода распознавания типа файла или фрагмента данных, полученного по электронной почте либо загруженного из Интернета. Первый способ – по суффиксу файла; например, суффикс `.png` соответствует файлам в формате Portable Network Graphics (PNG). Другой способ – анализ магического числа, последовательности из одного или нескольких байтов, расположенной, как правило, в начале файла; например, в начале `png`-файла находятся байты `0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A`.

К сожалению, оба эти метода страдают общим недостатком: они официально не стандартизированы, поэтому не исключено, что один и тот же суффикс или магическое число будет относиться к совершенно разным типам данных. И такое дублирование на практике действительно имеет место, особенно в части файловых суффиксов.

Третий метод распознавания файлов или данных заключается в анализе типа MIME, например данным в формате `png` соответствует тип `image/png`. Типы MIME стандартизированы Комитетом по цифровым адресам в Интернете (IANA), поэтому каждый такой тип уникально идентифицирует формат данных.

В модуле *Phonon* (и, кстати, в подсистеме перетаскивания мышью и работы с буфером обмена в Qt) типы MIME используются для идентификации форматов файлов и данных. Этот механизм хорошо работает для данных, полученных по электронной почте или из Интернета. Однако для обработки файлов типы MIME менее удобны, потому что необходимо каким-то образом отобразить типы MIME на суффиксы или магические числа. Но если сами типы MIME стандартизированы, то никакого официального отображения их на суффиксы и магические числа не существует. На платформе

Linux приемлемое – хотя и неполное – отображение обычно хранится в текстовом файле */etc/mime.types*.

В модуле `aqr.{hpp,cpp}`, включенном в примеры к этой книге, имеется функция `QSet<QString> suffixesForMimeTypes(const QStringList &mimeTypes)`, которая использует файл *mime.types*, если он имеется (наряду с зашитыми в нее данными), для отображения типов MIME на суффиксы. Эта функция работает на всех платформах, и ее легко расширить, добавив дополнительные типы MIME, – достаточно дополнить файл *mime.types* или подправить код.

Примечание. Компания Apple Inc. разработала четвертый способ идентификации типов данных (как файлов, так и фрагментов данных) – унифицированные идентификаторы типов (Uniform Type Identifiers – UTI). Они призваны решить проблемы, преследующие другие методы идентификации, но пока идея UTI не получила широко признания.

Приведенный выше код оказывается хрупким в случае, когда метаданные неполны или отсутствуют. Одно из возможных решений – подставить вместо отсутствующего имени исполнителя или названия альбома имя файла, а в качестве названия композиции использовать порядковый номер.

```
TreeWidgetItem *artistItem = itemForArtist->value(artist);
if (!artistItem) {
    artistItem = new TreeWidgetItem(
        treeWidget->invisibleRootItem(),
        QStringList() << artist);
    itemForArtist->insert(artist, artistItem);
}
TreeWidgetItem *albumItem = itemForArtistAlbum->value(artistAlbum);
if (!albumItem) {
    albumItem = new TreeWidgetItem(artistItem, QStringList() << album);
    itemForArtistAlbum->insert(artistAlbum, albumItem);
}
TreeWidgetItem *trackItem = new TreeWidgetItem(albumItem,
    QStringList() << track
    << minutesSecondsAsStringForMSec(msec));
trackItem->setData(0, FilenameRole, filename);
trackItem->setTextAlignment(1, Qt::AlignVCenter|Qt::AlignRight);
}
```

Узел любой композиции в дереве должен быть потомком узла альбома, а узел любого альбома – потомком узла исполнителя. Поэтому сначала мы ищем в хеше `itemForArtist` объект `TreeWidgetItem`, соответствующий исполнителю текущей композиции. Если это первая композиция в первом альбоме данного исполнителя, то такой объект не будет най-

ден, и тогда `QHash::value()` вернет значение объекта `TreeWidgetItem`, созданное конструктором по умолчанию (`default-constructed value`), то есть нулевой указатель. В таком случае мы создаем элемент как дочерний узел невидимого корня дерева, то есть получаем элемент верхнего уровня. Неважно, создали мы узел или нашли его в хеше, все равно в итоге получается узел дерева, представляющий исполнителя композиции. А если мы создали новый элемент, то добавляем его в хеш `itemForArtist`, чтобы его можно было найти в следующий раз.

Аналогичная процедура применяется, когда нужно получить элемент, описывающий альбом. Отличие лишь в том, что при необходимости создать элемент (поскольку это первая композиция в первом альбоме исполнителя данной композиции) мы делаем элемент-альбом потомком элемента-исполнителя. А вновь созданный элемент добавляется в хеш `itemForArtistAlbum`.

И наконец мы создаем элемент, описывающий композицию, и делаем его потомком элемента-альбома, который сам является потомком элемента-исполнителя. В первый столбец этого элемента-композиции мы записываем название композиции, а во второй – продолжительность звучания в минутах и секундах. Кроме того, в пользовательских данных для первого столбца элемента-композиции сохраняется имя файла.

Для полноты картины приведем текст метода `minutesSecondsAsStringForMSec()`, тем более что он еще понадобится нам позже.

```
QString MainWindow::minutesSecondsAsStringForMSec(qint64 msec)
{
    int minutes;
    int seconds;
    AQP::hoursMinutesSecondsForMSec(msec, 0, &minutes, &seconds);
    return QString("%1:%2").arg(minutes, 2, 10, QChar(' '))
        .arg(seconds, 2, 10, QChar('0'));
}
```

Функция `AQP::hoursMinutesSecondsForMSec()` принимает количество миллисекунд и вычисляет, сколько в нем часов, минут и секунд. Полученные величины присваиваются трем целым числам, переданным по указателям. Если функции `QString::arg()` в качестве первого аргумента передано целое число, то остальными необязательными аргументами являются ширина поля, основание системы счисления и символ-заполнитель.

В методе `addTrack()` вызывается метод `waitForMediaObjectToLoad()` для получения метаданных из музыкального файла. Простейший способ решить эту задачу – написать цикл `while`, внутри которого проверять, готовы уже метаданные или нет, и установить таймер, по истечении которого мы выйдем из цикла, если ждали результата слишком долго. Но такое активное ожидание без нужды потребляет процессорное время, поэтому мы выбрали более эффективный подход с применением локального цикла сообщений.

```

bool MainWindow::waitForMediaObjectToLoad(
    Phonon::MediaObject *mediaObject, int timeoutMSec)
{
    QEventLoop eventLoop;
    QTimer timer;
    timer.setSingleShot(true);
    timer.setInterval(timeoutMSec);
    connect(&timer, SIGNAL(timeout()), &eventLoop, SLOT(quit()));
    connect(mediaObject, SIGNAL(stateChanged(Phonon::State, Phonon::State)),
        &eventLoop, SLOT(quit()));
    timer.start();
    eventLoop.exec();
    return mediaObject->state() == Phonon::StoppedState;
}

```

Сначала создается цикл сообщений и однократный таймер с заданным тайм-аутом. Оба установленных соединения сигналов со слотами осуществляют выход из цикла сообщений, первое – в результате истечения тайм-аута, второе – вследствие изменения состояния. Настроив соединения, мы активируем таймер и запускаем цикл сообщений, а затем ждем выхода из цикла.

Пока медиаобъект загружается, он находится в состоянии `Phonon::LoadingState`. Как только состояние медиаобъекта изменяется, мы тут же выходим из цикла. Если загрузка оказалась успешной, то медиаобъект перейдет в состояние `Phonon::StoppedState` и мы можем безопасно извлекать из него метаданные, поэтому в данном случае возвращаем `true`. Если же цикл событий прерван сигналом от таймера, то мы считаем, что файл поврежден или не может быть прочитан по какой-то другой причине, и потому возвращаем `false` – для вызывающей программы это означает, что файл следует пропустить, не добавляя в дерево.

Итак, мы просмотрели весь каталог с музыкальными файлами и поместили в дерево данные обо всех обнаруженных композициях. Теперь пользователь может «гулять» по дереву, и при изменении текущего элемента будет вызываться слот `currentItemChanged()`.

```

void MainWindow::currentItemChanged(QTreeWidgetItem *item)
{
    if (!playOrPauseAction->isEnabled()) {
        QString filename = item->data(0, FilenameRole).toString();
        if (!filename.isEmpty())
            playOrPauseAction->setEnabled(true);
    }
}

```

Если действие `playOrPauseAction` деактивировано, но текущим элементом стала композиция, то мы активируем это действие, чтобы пользователь мог нажать кнопку Play/Pause. При нажатии этой кнопки вызывается слот `playOrPause()`.

```
void MainWindow::playOrPause()
{
    switch (mediaObject->state()) {
        case Phonon::PlayingState:
            mediaObject->pause();
            playOrPauseAction->setIcon(playIcon);
            break;
        case Phonon::PausedState:
            mediaObject->play();
            playOrPauseAction->setIcon(pauseIcon);
            break;
        default:
            playTrack(treeWidget->currentItem());
            break;
    }
}
```

Если какая-то композиция уже воспроизводится, то кнопка Play/Pause выступает в роли Pause, поэтому мы приостанавливаем воспроизведение медиаобъекта и изменяем пиктограмму кнопки, показывая, что теперь это кнопка Play. Если же воспроизведение уже приостановлено, то поступаем в точности наоборот: возобновляем воспроизведение и превращаем кнопку в Pause.

Если ни одна композиция не воспроизводится и не приостановлена, значит, пользователь нажал кнопку Play, выбрав новый элемент. В таком случае мы вызываем слот `playTrack()`, передавая ему текущий элемент. Тот же слот вызывается в случае двойного щелчка по элементу.

```
void MainWindow::playTrack(QTreeWidgetItem *item)
{
    Q_ASSERT(item);
    QString filename = item->data(0, FilenameRole).toString();
    if (filename.isEmpty())
        return;
    if (!QFile::exists(filename)) {
        AQP::warning(this, tr("Error"),
            tr("File %1 appears to have been moved or deleted")
                .arg(filename));
        return;
    }
    nextItem = item;
    mediaObject->clearQueue();
    mediaObject->setCurrentSource(filename);
    mediaObject->play();
}
```

Сначала мы извлекаем имя файла композиции, но ничего не делаем, если текущий элемент соответствует исполнителю или альбому (и, следовательно, с ним не ассоциировано имя файла) или если файл был

удален либо переименован с момента чтения каталога. Правда, в последнем случае мы еще и выводим сообщение об ошибке. (См. врезку на стр. 77, где объяснено, почему мы пользуемся функцией `AQP::warning()`, а не `QMessageBox::warning()`.) В переменную `nextItem` записывается указатель на текущий элемент, после чего мы очищаем очередь источников медийных данных для медиаобъекта, устанавливаем указанный файл в качестве текущего источника и запускаем воспроизведение. В результате вызывается слот `currentSourceChanged()`.

```
const QString ZeroTime(" 0:00");

void MainWindow::currentSourceChanged()
{
    if (nextItem) {
        playOrPauseAction->setIcon(pauseIcon);
        timeLcd->display(ZeroTime);
        setWindowTitle(tr("%1 - %2")
            .arg(QApplication::applicationName())
            .arg(nextItem->text(0)));
        treeWidget->setCurrentItem(nextItem);
        nextItem = 0;
    }
}
```

Если следующий элемент существует (как будет в случае, когда данный слот вызван в результате нажатия кнопки Play при выделении подсветкой какой-либо композиции или при двойном щелчке мышью по ней), то мы обновляем интерфейс — сбрасываем время на дисплее в значение 0:00 и показываем название композиции в строке заголовка окна. Затем в переменную `nextItem` записывается 0.

В любой момент во время воспроизведения пользователь может приостановить его, нажав кнопку Pause, или вообще остановить кнопкой Stop. При изменении состояния медиаобъекта вызывается слот `stateChanged()`, а при каждом тике таймера (в данном случае каждую секунду) вызывается слот `tick()`. Если композиция проиграна до конца, то непосредственно перед завершением вызывается слот `aboutToFinish()`.

Далее рассмотрим реализацию этих слотов.

```
void MainWindow::stop()
{
    nextItem = 0;
    mediaObject->stop();
    mediaObject->clearQueue();
    playOrPauseAction->setIcon(playIcon);
    timeLcd->display(ZeroTime);
    setWindowTitle(QApplication::applicationName());
}
```

Если пользователь нажимает кнопку Stop, мы обнуляем указатель на следующий элемент, чтобы предотвратить автоматический переход к следующей композиции в случае, когда текущая почти завершилась, а затем останавливаем воспроизведение. Кроме того, очищается очередь медиаобъекта на случай, если в нее уже добавлена следующая композиция; об управлении очередью мы расскажем при обсуждении слота `aboutToFinish()` ниже. Затем текущая ситуация отражается в пользовательском интерфейсе.

```
void MainWindow::tick(qint64 msec)
{
    timeLcd->display(minutesSecondsAsStringForMSec(msec));
}
```

Этот слот вызывается при каждом тике (в данном случае один раз в секунду) и обновляет текущую временную позицию точки воспроизведения на цифровом дисплее.

```
void MainWindow::stateChanged(Phonon::State newState)
{
    switch (newState) {
        case Phonon::ErrorState:
            AQP::warning(this, tr("Error"),
                        mediaObject->errorString());
            playOrPauseAction->setEnabled(false);
            stopAction->setEnabled(false);
            break;
        case Phonon::PlayingState:
            playOrPauseAction->setEnabled(true);
            playOrPauseAction->setIcon(pauseIcon);
            stopAction->setEnabled(true);
            break;
        case Phonon::PausedState:
            playOrPauseAction->setEnabled(true);
            playOrPauseAction->setIcon(playIcon);
            stopAction->setEnabled(true);
            break;
        case Phonon::StoppedState:
            playOrPauseAction->setEnabled(true);
            playOrPauseAction->setIcon(playIcon);
            stopAction->setEnabled(false);
            timeLcd->display(ZeroTime);
            break;
        default:
            playOrPauseAction->setEnabled(false);
            break;
    }
}
```

При всяком изменении состояния медиаобъекта мы обновляем пользовательский интерфейс: активируем или деактивируем соответствующие новому состоянию действия, а в случае ошибки выводим сообщение с объяснением причины.

```
void MainWindow::aboutToFinish()
{
    QTreeWidgetItem *item = nextItem ? nextItem :
                                   treeWidget->currentItem();

    if (!item)
        return;
    item = treeWidget->itemBelow(item);
    if (!item) // Текущая композиция последняя в дереве
        return;
    QString filename = item->data(0, FilenameRole).toString();
    if (filename.isEmpty()) { // item соответствует исполнителю или альбому

        item = item->child(0);
        if (!item)
            return;
        else {
            filename = item->data(0, FilenameRole).toString();
            if (filename.isEmpty()) // item соответствует альбому
                item = item->child(0);
            if (!item)
                return;
            filename = item->data(0, FilenameRole).toString();
            if (filename.isEmpty())
                return;
        }
    }
    nextItem = item;
    Phonon::MediaSource source(filename);
    mediaObject->enqueue(source);
}
```

Слот `aboutToFinish()` предназначен для того, чтобы приложение могло подготовиться к воспроизведению следующей композиции в случае, когда пользователь не остановил воспроизведение текущей и оно дошло до конца. Мы должны рассмотреть четыре случая. Самый простой – когда ниже текущей композиции в дереве ничего нет; это означает, что мы дошли до конца альбома, поэтому слот ничего не делает и сразу же возвращает управление. В остальных случаях под текущим элементом есть какие-то другие. Если следующий элемент – исполнитель, то необходимо найти первую композицию в первом альбоме этого исполнителя. Аналогично, если следующий элемент – альбом, то нужно найти первую композицию в нем. В противном случае следующий элемент соответствует композиции. Найдя композицию, которая будет воспроиз-

водиться следующей, мы записываем в переменную `nextItem` указатель на соответствующий ей элемент дерева и добавляем источник медийных данных, то есть имя файла, хранящееся в этом элементе, в очередь медиаобъекта.

Если пользователь не нажимал кнопку `Stop`, то по завершении текущей композиции медиаобъект выберет в качестве своего источника медийных данных следующую композицию из очереди и сгенерирует сигнал `currentSourceChanged()`. Этот сигнал связан с рассмотренным выше слотом `currentSourceChanged()` главного окна, который обновляет пользовательский интерфейс – показывает, что началось воспроизведение новой композиции.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    nextItem = 0;
    mediaObject->stop();
    mediaObject->clearQueue();
    event->accept();
}
```

Когда пользователь закрывает приложение, важно остановить проигрыватель, иначе он будет и дальше воспроизводить музыку, хотя окна приложения уже нет.

На этом мы завершаем рассмотрение приложения `Play Music`. В него можно было бы внести два очевидных усовершенствования. Во-первых, запоминать последний выбранный каталог с музыкальными композициями (например, с помощью класса `QSettings`) и по умолчанию делать его начальным при следующем запуске. Во-вторых, на этапе инициализации читать только перечень подкаталогов (это существенно ускорит запуск, особенно если количество композиций велико – сотни и тысячи), а список композиций в подкаталоге заполнять только в момент, когда пользователь раскрывает соответствующую ветвь дерева. Обе эти доработки оставляем читателю в качестве упражнения.

Написанное приложение вполне работоспособно, хотя мы использовали не весь арсенал средств, предлагаемых `Phonon API` для воспроизведения музыки; мы рассмотрели лишь наиболее важные аспекты. Большая часть остальных возможностей касается различных тонких деталей. Например, можно управлять промежутком времени между окончанием одной композиции и началом воспроизведения следующей в очереди. Для этого служит свойство `transitionTime`. По умолчанию оно равно 0, то есть никакой паузы нет вовсе. Отрицательное значение можно использовать для организации перекрестного затухания, а положительное означает, что между двумя композициями будет пауза заданной длительности (в миллисекундах). Кроме того, можно программно перейти в конкретную временную точку (если источник медийных дан-

ных допускает такую операцию) и подключиться к другим сигналам, генерируемым медиаобъектом, например к сигналу `finished()`; это мы рассмотрим в следующем разделе.

Воспроизведение видео

Воспроизведение видео и музыки – очень похожие процессы, по крайней мере принципиально. Основное различие состоит в том, что помимо тракта, идущего от медиаобъекту к аудиовыходу, создается также тракт, идущий к видеовыходу. Как это выглядит на практике, мы продемонстрируем на примере приложения `Play Video (playvideo)`, показанного на рис. 2.4.



Рис. 2.4. Приложение `Play Video`: воспроизводится смоделированное видео

Программа `Play Video` загружает и воспроизводит одновременно один видеофайл, поэтому вместо того, чтобы открывать каталог, как в приложении `Play Music` из предыдущего раздела, пользователь должен открыть видеофайл. Отсюда, в частности, следует, что создавать очередь видео не нужно, потому что в каждый момент времени используется только один файл. Тем не менее приложение `Play Video` по своей структуре очень похоже на `Play Music`, поэтому мы ограничимся лишь рассмотрением аспектов, относящихся непосредственно к `Phonon`, и, в особенности, его отличий от приложения `Play Music`. Начнем с конструктора главного окна.

```
const int OneSecond = AQP::MSecPerSecond;

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    playIcon = QIcon(":/play.png");
    pauseIcon = QIcon(":/pause.png");
}
```

```

mediaObject = new Phonon::MediaObject(this);
mediaObject->setTickInterval(OneSecond);
videoWidget = new Phonon::VideoWidget(this);
Phonon::createPath(mediaObject, videoWidget);
audioOutput = new Phonon::AudioOutput(Phonon::VideoCategory, this);
Phonon::createPath(mediaObject, audioOutput);

createActions();
createToolBar();
createWidgets();
createLayout();
createConnections();

setWindowTitle(QApplication::applicationName());
}

```

Здесь создается медиаобъект для управления видео и два тракта: от медиаобъекта к видеовиджету (который одновременно является и видеовыходом) и к аудиовыходу. Создавать тракт, идущий к аудиовыходу, имеет смысл только для видео со звуковой дорожкой, но поскольку данное приложение рассчитано на любые видеофайлы, мы хотели бы всегда иметь возможность воспроизвести звуковую дорожку, если таковая присутствует. Следует отметить, что в качестве категории аудио теперь нужно задавать `Phonon::VideoCategory`, тогда как для воспроизведения музыки мы задавали `Phonon::MusicCategory`.

Метод `createActions()` и прочие методы `createXXX` (кроме `createConnections()`) мы опустим, потому что они мало чем отличаются от рассмотренных ранее. Например, метод `createWidgets()` почти дословно совпадает с одноименным методом из приложения `Play Music`, именно здесь создаются виджеты `Phonon::SeekSlider` и `Phonon::VolumeSlider`. Отметим, однако, что никто не заставляет нас пользоваться виджетами из библиотеки `Phonon`; с равным успехом можно было бы создать собственные виджеты для управления перемещением по медиаданным и громкостью — с помощью композиции существующих виджетов или написав с нуля.

```

void MainWindow::createConnections()
{
    connect(mediaObject, SIGNAL(tick(qint64)), this, SLOT(tick(qint64)));
    connect(mediaObject,
            SIGNAL(stateChanged(Phonon::State, Phonon::State)),
            this, SLOT(stateChanged(Phonon::State)));
    connect(mediaObject, SIGNAL(finished()), this, SLOT(stop()));
    connect(fullScreenAction, SIGNAL(triggered()),
            videoWidget, SLOT(enterFullScreen()));
    connect(stopAction, SIGNAL(triggered()), this, SLOT(stop()));
    connect(playOrPauseAction, SIGNAL(triggered()),
            this, SLOT(playOrPause()));
}

```

```

connect(chooseVideoAction, SIGNAL(triggered()),
        this, SLOT(chooseVideo()));
connect(quitAction, SIGNAL(triggered()), this, SLOT(close()));
}

```

Первые три соединения устанавливаются между медиаобъектом и слотами главного окна. Сигнал `tick()` используется для обновления цифрового дисплея, на котором показана позиция текущей точки воспроизведения во времени. Соединение с сигналом `stateChanged()` служит для реагирования на изменение состояния, а сигнал `finished()` позволяет обнаружить момент окончания воспроизведения.

Остальные соединения применяются для реализации функций, свойственных всем приложениям такого рода: начало и возобновление воспроизведения, приостановка, прекращение, выбор видеофайла и выход из программы.

Одно из соединений установлено с кнопкой на панели инструментов, нажатие которой приводит к переходу в полноэкранный режим. В этом режиме отображается только видео, никаких виджетов для управления приложением нет. Поэтому необходимо предоставить какой-то способ возврата к оконный режим. Для этого мы зарезервировали клавишу (Esc) и настроили фильтр событий так, что при щелчке мышью по видеовиджету тоже происходит возврат в оконный режим. Ниже приведены строки из кода `createActions()`, в которых настраивается фильтр событий и горячая клавиша:

```

videoWidget->installEventFilter(this);
(void) new QShortcut(QKeySequence("Escape"),
                    videoWidget, SLOT(exitFullScreen()));

```

А это код самого фильтра событий:

```

bool MainWindow::eventFilter(QObject *target, QEvent *event)
{
    if (target == videoWidget &&
        event->type() == QEvent::MouseButtonPress &&
        videoWidget->isFullScreen()
        videoWidget->exitFullScreen();
    return QMainWindow::eventFilter(target, event);
}

```

Этот фильтр обеспечивает возврат в оконный режим по щелчку мышью.

Того же эффекта можно было бы достичь, создав подкласс класса `Phonon::VideoWidget` и переопределив в нем метод `mousePressEvent()`. Реализация небольшого изменения в поведении виджета с помощью фильтра события может оказаться разумным выбором, но если создается несколько экземпляров виджета или его более сложная модификация, то лучше использовать подкласс, поскольку наличие многочисленных фильтров событий может привести к снижению производительности.

Далее рассмотрим слоты, отличающиеся от одноименных методов в приложении **Play Music**, и начнем с самого длинного, `chooseVideo()`, который разобьем на три части.

```
void MainWindow::chooseVideo()
{
    QString filename = QFileDialog::getOpenFileName(this,
        tr("Choose Video"), QDesktopServices::storageLocation(
            QDesktopServices::MoviesLocation), getFileFormats());
    if (filename.isEmpty())
        return;
```

Прежде всего мы просим пользователя выбрать видеофайл, установив в качестве начального каталог «Видео» (или домашний каталог пользователя, если каталог фильмов не сконфигурирован). Если пользователь нажмет кнопку отмены, то метод ничего не делает и просто возвращает управление.

```
stop();
playOrPauseAction->setEnabled(false);
stopAction->setEnabled(false);
mediaObject->setCurrentSource(filename);
if (!mediaObject->hasVideo()) {
    QEventLoop eventLoop;
    QTimer timer;
    timer.setSingleShot(true);
    timer.setInterval(3 * OneSecond);
    connect(&timer, SIGNAL(timeout()), &eventLoop,
        SLOT(quit()));
    connect(mediaObject, SIGNAL(hasVideoChanged(bool)),
        &eventLoop, SLOT(quit()));
    timer.start();
    eventLoop.exec();
}
```

После того как видеофайл выбран, мы останавливаем воспроизведение текущего видео и деактивируем соответствующие действия. Как и в случае аудио, между загрузкой видео методом `Phonon::MediaObject::setCurrentSource()` и началом его воспроизведения может быть заметный временной интервал. Если видеоданные медиаобъекта сразу недоступны, то мы используем локальный цикл событий почти так же, как в приложении **Play Music**. Различие в том, что выход из цикла производится либо при изменении состояния видеопотока, то есть когда видеоданные становятся доступны, либо по тайм-ауту (в данном случае мы устанавливаем больший тайм-аут, три секунды).

```
if (mediaObject->hasVideo()) {
    QString title(mediaObject->metaData(Phonon::TitleMetaData)
        .join("/") .trimmed());
    if (title.isEmpty())
```

```

        title = QFileInfo(filename).baseName();
        setWindowTitle(tr("%1 - %2")
            .arg(QApplication::applicationName()).arg(title));
        mediaObject->play();
    }
    else {
        setWindowTitle(QApplication::applicationName());
        AQP::warning(this, tr("Error"),
            tr("Cannot play video from %1").arg(filename));
    }
}

```

После выхода из локального цикла событий либо стали доступны видеоданные, и в этом случае мы извлекаем кое-какие метаданные и начинаем воспроизведение, либо произошел тайм-аут, и тогда мы сообщаем пользователю о проблеме. Явно никакие действия не активируются, это происходит в слоте `stateChanged()`, а состояние изменяется сразу, как только начинается воспроизведение.

```

QString MainWindow::getFileFormats()
{
    QStringList mimeTypeTypes;
    foreach (const QString &mimeType,
        Phonon::BackendCapabilities::availableMimeTypes())
    if (mimeType.startsWith("video/"))
        mimeTypeTypes << mimeType;
    return AQP::filenameFilter(tr("Video"), mimeTypeTypes);
}

```

Этот метод приведен только для полноты картины. Он создает список MIME-типов видеоформатов, поддерживаемых движком Phonon, а затем на его основе формирует список файловых суффиксов, обращаясь к функции `AQP::filenameFilter()` из модуля `aqp.{hpp,cpp}`. Об отображении типов MIME на файловые суффиксы см. врезку «Типы MIME, файловые суффиксы и магические числа» на стр. 89.

Если не считать небольшого обходного маневра, слот `stateChanged()` ничем не отличается от одноименного слота в приложении `Play Music`, поэтому мы покажем только различия:

```

case Phonon::PlayingState:
    videoWidget->setAspectRatio(Phonon::VideoWidget::AspectRatioWidget);
    videoWidget->setAspectRatio(Phonon::VideoWidget::AspectRatioAuto);
    playOrPauseAction->setEnabled(true);
    playOrPauseAction->setIcon(pauseIcon);
    stopAction->setEnabled(true);
    break;

```

В начале воспроизведения видео необходимо масштабировать в соответствии с размером видеовиджета, учтя пропорции. Когда писалась эта

книга, данная операция не производилась автоматически при воспроизведении определенных видеофайлов в некоторых системах, поэтому мы на всякий случай *дважды* вызываем метод `Phonon::VideoWidget::setAspectRatio()` с разными параметрами: один раз для изменения масштаба, другой – для корректировки пропорций.

```
void MainWindow::playOrPause()
{
    switch (mediaObject->state()) {
        case Phonon::PlayingState:
            mediaObject->pause();
            playOrPauseAction->setIcon(playIcon);
            break;
        case Phonon::PausedState: // Проход
        case Phonon::StoppedState:
            mediaObject->play();
            playOrPauseAction->setIcon(pauseIcon);
            break;
        default:
            break;
    }
}
```

Слот `playOrPause()` гораздо проще, чем в приложении *Play Music*. Слоты `stop()`, `tick()` и `closeEvent()` мы опускаем, потому что они мало чем отличаются от того, что мы уже видели в *Play Music*.

Вот мы и закончили рассмотрение приложения *Play Video*. Можно было бы поступить проще и воспользоваться классом `Phonon::VideoPlayer` вместо `Phonon::VideoWidget`. Виджет видеоплеера удобнее – не нужно создавать источник, медиаобъект, тракт и приемник, но за это приходится расплачиваться утратой средств более точного управления, имеющихся при работе с видеовиджетом.

В модуль *Phonon* включен также класс `Phonon::MediaController`, предназначенный для контроля над дополнительными возможностями, которые предоставляют некоторые мультимедийные носители, например названия CD-дисков, главы DVD и ракурсы DVD. Во время работы над этой книгой ни один из движков Qt Phonon не поддерживал эти возможности.

На этом мы заканчиваем рассмотрение модуля *Phonon*. В настоящее время он не поддерживает ни захват мультимедийных данных – например, вырезание аудио или видеоклипов, – ни сохранение мультимедиа для последующего воспроизведения. Нет также средств для манипулирования потоками, например редактирования потока или микширования нескольких источников. Все эти недостатки, скорее всего, будут устранены в будущем, когда модуль *Phonon* станет более зрелым.

В версии Qt 4.6 появился новый низкоуровневый мультимедийный модуль *QtMultimedia*. Он умеет читать и воспроизводить аудио- и видеоданные так же, как и модуль *Phonon*, но уровень предлагаемого интерфейса ниже, поэтому работать с ним труднее, чем с *Phonon*. Например, чтобы воспроизвести аудиофайл с помощью *QtMultimedia*, необходимо создать объект `QAudioFormat`, содержащий различные технические детали аудиоформата, в частности частоту дискретизации, число каналов, квантование и MIME-тип кодека, а затем передать его объекту `QAudioOutput` для воспроизведения вместе с объектом `QFile`, представляющим аудиофайл, открытый в двоичном режиме. Напротив, модуль *Phonon* разбирается со всеми этими низкоуровневыми деталями автоматически и требует только задания имени файла. Поэтому лучше начать с использования *Phonon*, а к *QtMultimedia* прибегать только тогда, когда необходим низкоуровневый контроль, не предоставляемый *Phonon*, но имеющийся в *QtMultimedia*.

3

Архитектура модель/представление: табличные модели

- Архитектура модель/представление в Qt
- Применение класса `QStandardItemModel` для таблиц
- Создание пользовательских табличных моделей

Среди усовершенствований, появившихся в версии Qt 4.0, следует особо отметить архитектуру модель/представление для элементов данных. Она позволяет программисту легко отделить данные от их представления: в предыдущих версиях Qt решение подобной задачи доставляло некоторые неудобства. По мере развития серии Qt 4.x эта архитектура пополнялась все новой функциональностью и стала гораздо более совершенной, полезной и надежной по сравнению с тем, что представляла собой вначале¹.

Это первая из четырех глав, посвященных различным аспектам архитектуры модель/представление в Qt. Здесь мы рассмотрим табличные модели, а в последующих главах обратимся к древовидным моделям, делегатам и представлениям.

У каждого представления имеется делегат по умолчанию (вместо которого можно подставить собственный), который служит для отображения элементов, а в случае редактируемых элементов предоставляет

¹ Сейчас разрабатывается архитектура модель/представление нового поколения, но когда она станет достаточно зрелой и войдет в Qt, да и войдет ли вообще, можно только гадать. О том, что делается в этом направлении, можно узнать по адресу gitorious.org/qt-labs/itemviews-ng.

также подходящий редактор. Из всех имеющихся встроенных представлений мы ограничимся только классом `QComboBox`, а также теми виджетами-представлениями, которые требуют задания модели и в полной мере задействуют мощь и гибкость архитектуры модель/представление, в частности `QTableView` и `QTreeView`. И разумеется, в главе, посвященной представлениям, мы расскажем о создании пользовательских представлений.

Qt также поддерживает списковые модели, но специально мы их рассматривать не будем, поскольку, по существу, это не что иное, как табличные модели с одним столбцом. (Однако мы встретимся со списковой моделью в главе 6, когда займемся созданием программы просмотра для пользовательской списковой модели.) Древовидные модели будут обсуждаться в следующей главе.

В первом разделе этой главы приводится очень краткое введение в архитектуру модель/представление. Затем, во втором разделе, мы создадим подкласс `QStandardItemModel`, добавив в него возможность загружать и сохранять объекты типа `QStandardItem` в файл. А в третьем разделе мы создадим пользовательскую табличную модель, которую можно будет подставить вместо подкласса стандартной модели элементов, созданного во втором разделе.

И во втором, и в третьем разделе мы покажем, как добавлять строки (с редактированием на месте с помощью делегата), удалять и редактировать их (опять-таки с редактированием на месте). Мы также создадим два подкласса `QSortFilterProxyModel`: один – для фильтрации всех строк, кроме тех, которые интересуют пользователя, а другой – для фильтрации строк-дубликатов. Начнем изложение с общего описания архитектуры модель/представление в Qt, чтобы стало понятно, о чем пойдет речь.

Архитектура модель/представление в Qt

На протяжении этой и следующей главы модели будут применяться для хранения элементов данных. Qt предлагает несколько виджетов для просмотра данных, хранящихся в моделях. Это виджеты-представления: `QListView`, `QTableView`, `QColumnView` (показывает древовидную иерархию в виде горизонтальной последовательности списков – идиома, используемая в Mac OS X) и `QTreeView`. Для каждого из них необходимо указать модель – либо написанную нами, либо одну из предопределенных в Qt. Существуют также готовые виджеты (convenience widgets), названные так потому, что они уже содержат встроенную модель и, стало быть, могут использоваться непосредственно. К их числу относятся `QListWidget`, `QTableWidget` и `QTreeWidget`. И есть виджет `QComboBox`, который можно использовать либо как готовый, поскольку у него уже есть собственная модель, либо как виджет-представление для внешней модели, которую

мы должны предоставить сами. Виджеты-представления рассматриваются в главе 6.

Для всех стандартных представлений имеется создаваемый по умолчанию делегат `QStyledItemDelegate` – класс, который отображает элементы внутри представления, а для редактируемых элементов еще и предоставляет подходящий редактор. Естественно, мы можем написать собственный делегат и тем самым получить полный контроль над представлением и редактированием элементов. Делегаты рассматриваются в главе 5.

Отношения между моделями, представлениями, делегатами и собственными данными показаны на рис. 3.1.

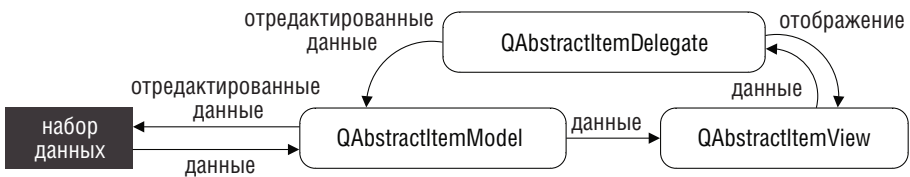


Рис. 3.1. Архитектура модель/представление в Qt

В некоторых контекстах использование готовых виджетов вполне оправданно – особенно для небольших наборов данных (от сотен до нескольких тысяч элементов), при условии, что содержащиеся в них данные в каждый момент времени показываются только в одном виджете. В предыдущих главах мы уже встречались с готовыми представлениями, например с объектами `QTableWidget` в примере Matrix Quiz из главы 1 (стр. 60) и `QTreeWidget` в примере Play Music из главы 2 (стр. 81).

Пожалуй, главное, что нужно понимать при обсуждении моделей Qt, – тот факт, что, несмотря на наличие единого унифицированного API, существуют два совершенно разных рода моделей: табличные, формулируемые в терминах строк и столбцов, и древовидные, в которых речь идет о родителях и потомках. (Списковые модели – это частный случай табличных с одним-единственным столбцом.) В этой главе мы будем рассматривать табличные модели, а в главе 4 – древовидные.

Помимо двух родов моделей, существует три типа моделей. Первый – это предопределенные модели, включенные в Qt, например `QStringListModel`, `QDirModel` и пришедшая ей на смену `QFileSystemModel`, – их можно использовать непосредственно с минимальными дополнительными усилиями. Второй тип – `QStandardItemModel`; это обобщенная модель, которую можно использовать как списковую, табличную или древовидную. Она предоставляет API, ориентированный на элементы данных, очень похожий на API готовых виджетов-представлений (например, `QTableWidget`). Класс `QStandardItemModel` идеален для случая, когда данные точно соответствуют идиоме списка, таблицы или дерева, состоя-

щего из элементов; при этом его можно применять в готовом виде или с минимальной адаптацией. И последний тип – написанные нами модели, производные от класса `QAbstractItemModel` (а также от `QAbstractListModel` либо `QAbstractTableModel`). Их рекомендуется применять, когда нужно обеспечить максимальную производительность или модель не вполне укладывается в рамки идиомы набора элементов. Часть иерархии моделей в Qt показана на рис. 3.2.

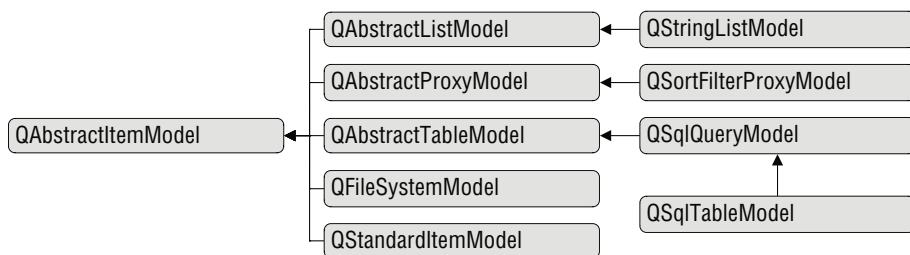


Рис. 3.2. Некоторые классы из иерархии моделей в Qt

Теоретически при использовании класса `QStandardItemModel` создавать собственную модель необязательно, однако во время работы над этой книгой у объектов типа `QStandardItemModel` было два потенциальных недостатка. Во-первых, для загрузки объемных наборов данных может потребоваться заметно больше времени, чем в случае использования пользовательской модели, а, во-вторых, предоставляемый API для древовидных моделей не обладает той полнотой функциональности, которую можно получить в пользовательских моделях. Тем не менее в большинстве случаев можно начать с использования класса `QStandardItemModel`, а впоследствии, если понадобится, реализовать замещающую его пользовательскую модель. Краткое сравнение `QStandardItemModel` с пользовательскими моделями приведено во врезке «`QStandardItemModel` и пользовательские модели» (стр. 131).

Применение класса `QStandardItemModel` для таблиц

Табличные модели оперируют в терминах строк и столбцов, а в качестве родителя каждого элемента выступает недопустимый объект типа `QModelIndex`. Между списковыми и табличными моделями нет принципиальной разницы; списковая модель – это просто табличная модель с одним столбцом.

В этом разделе мы покажем, как создать подкласс `QStandardItemModel`, который сможет загружать и сохранять пользовательские данные и будет хранить их в объектах типа `QStandardItem`. В следующем разделе мы подменим класс `QStandardItemModel` написанным нами подклассом класса

`QAbstractTableModel`, где для хранения данных будем использовать наши собственные облегченные объекты. Рассматриваемые в обоих разделах приложения обладают одинаковыми функциями, но на тех компьютерах, где они тестировались, второй вариант (на основе пользовательской модели) неизменно загружал данные значительно быстрее, чем первый.

В обоих примерах для отображения и редактирования элементов применяется пользовательский делегат `ItemDelegate`. Эта тема рассматривается в главе 5.

Приложения `Zipcodes` – `zipcodes1` и `zipcodes2` – умеют загружать и сохранять двоичные файлы, содержащие данные о почтовых индексах: сам индекс, название почтового отделения, округа и штата. Визуально приложения ничем не отличаются, функциональность их также одинакова. Внешний вид одного из них показан на рис. 3.3. Оба приложения предлагают стандартные операции: загрузку и сохранение, удаление строк, добавление и редактирование строк на месте. Кроме того, они позволяют различными способами выбирать и фильтровать строки, так что попутно мы узнаем, как создавать подклассы `QSortFilterProxyModel` и манипулировать моделью отбора данных для представления.



Рис. 3.3. Редактирование штата сотрудников в приложении `Zipcodes`

В приложении `zipcodes1` для загрузки и сохранения данных используется простой подкласс `QStandardItemModel`. Объект `QTableView`, в котором отображаются данные, получает к ним доступ с помощью подкласса `QSortFilterProxyModel`, умеющего фильтровать данные согласно заданному пользователем критерию. Комбинированные списки, применяемые для фильтрации (или выборки) строк, заполняются с помощью еще одного подкласса, `QSortFilterProxyModel`, который удаляет дубликаты. Само главное окно диалоговое (то есть вместо меню используются

кнопки), но в остальном вполне традиционное. Как обычно, мы будем уделять внимание деталям, относящимся к рассматриваемой теме (архитектуре модель/представление), не останавливаясь на создании окон и менеджеров компоновки и многих других методах.

Изменение табличной модели с помощью пользовательского интерфейса

Мы начнем с рассмотрения главного окна, это даст нам общее представление о программе, а затем перейдем к подклассам моделей, лежащих в основе приложения. Прежде всего – глобальные константы:

```
const int MinZipcode = 0;
const int MaxZipcode = 99999;
const int InvalidZipcode = MinZipcode - 1;
enum Column {Zipcode, PostOffice, County, State};
```

В пояснениях они не нужны. Далее приведем определение класса `MainWindow` из заголовочного файла, опустив все закрытые методы и почти все закрытые данные-члены (то есть большую часть виджетов).

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent=0);

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void load();
    void load(const QString &filename);
    bool save();
    void addZipcode();
    void deleteZipcode();
    void setDirty() { setWindowModified(true); }
    void updateUi();
    void radioButtonClicked();
    void selectionChanged();

private:
    ...
    QTableView *tableView;
    StandardTableModel *model;
    ProxyModel *proxyModel;
    bool loading;
};
```

Методы `load()` и `save()` служат для загрузки и сохранения данных приложения; мы их не приводим, хотя покажем вызываемые из них методы `StandardTableModel::load()` и `StandardTableModel::save()`, когда будем рассматривать подкласс `StandardTableModel` ниже.

Слоты `addZipcode()` и `deleteZipcode()` нужны для добавления и удаления строки данных, ниже мы рассмотрим оба.

Большая часть остальных закрытых слотов вызывается, когда пользователь взаимодействует с виджетами для фильтрации и выборки, например при выборе округа из комбинированного списка `County` (Округ). В секции `private:` расположено несколько закрытых методов и почти все виджеты (по большей части они не показаны, так как связаны с пользовательским интерфейсом, а не с программированием моделей и представлений). Когда пользователь устанавливает режим фильтрации или выборки либо задает с помощью комбинированных списков критерии (например, минимальный почтовый индекс или конкретный штат), вызывается слот `updateUi()`, который, в свою очередь, вызывает метод `restoreFilters()` для фильтрации или `performSelection()` для выборки. Все остальные методы – не более чем стандартное программирование ГИП (графического интерфейса пользователя, GUI) на C++/Qt.

В переменной `tableView` хранится представление приложения, а в переменной `model` – его модель. Однако представление взаимодействует с моделью не напрямую, а с помощью объекта `proxyModel`, который оставляет только строки, отвечающие заданному пользователем критерию фильтрации.

```
const int StatusTimeout = AQP::MSecPerSecond * 10;

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), loading(false)
{
    model = new StandardTableModel(this);
    proxyModel = new ProxyModel(this);
    proxyModel->setSourceModel(model);

    createWidgets();
    createComboBoxModels();
    createLayout();
    createConnections();

    AQP::accelerateWidget(this);
    setWindowTitle(tr("%1 (QStandardItemModel)[*]")
        .arg(QApplication::applicationName()));
    statusBar()->showMessage(tr("Ready"), StatusTimeout);
}
```

В начале конструктора мы создаем модель и прокси-модель, с которой будет работать представление. (Обе модели – экземпляры подклассов,

которые мы рассмотрим ниже.) Последующий код устроен по уже знакомому нам образцу. Мы рассмотрим лишь небольшой фрагмент в конце метода `createWidgets()`, а метод `createLayout()` вообще опустим. (О функции `AQP::accelerateWidget()` мы рассказывали выше, на стр. 30.)

```
void MainWindow::createWidgets()
{
    ...
    tableView = new QTableView;
    tableView->setModel(proxyModel);
    tableView->setItemDelegate(new ItemDelegate(this));
    tableView->verticalHeader()->setDefaultAlignment(
        Qt::AlignVCenter|Qt::AlignRight);
}
```

Обратите внимание, что в качестве модели табличного представления задается `proxyModel`, а не сама модель, содержащая данные. Пользовательский делегат будет рассмотрен позже (стр. 223). Мы задаем выравнивание вертикального заголовка (номера строк), традиционно принятое для чисел.

```
void MainWindow::createComboBoxModels()
{
    createComboBoxModel(countyComboBox, County);
    createComboBoxModel(stateComboBox, State);
}

void MainWindow::createComboBoxModel(QComboBox *comboBox,
                                     int column)
{
    delete comboBox->model();
    UniqueProxyModel *uniqueProxyModel = new UniqueProxyModel(
        column, this);
    uniqueProxyModel->setSourceModel(model);
    uniqueProxyModel->sort(column, Qt::AscendingOrder);
    comboBox->setModel(uniqueProxyModel);
    comboBox->setModelColumn(column);
}
```

Оба комбинированных списка, необходимые для фильтрации и выбора строк, должны получать данные от той же модели. Однако округа и штаты для разных почтовых индексов повторяются, что ясно видно на рис. 3.3. Поэтому в качестве модели мы установим для этих списков не `model`, а специальную прокси-модель, устраняющую дубликаты.

Перед тем как создавать модель для комбинированного списка, мы удаляем старую модель. Делается так потому, что каждый раз при загрузке нового файла мы вызываем метод `createComboBoxModels()`, поскольку хотим, чтобы фильтрация и выборка основывались на данных, которые присутствуют в загруженной модели. Например, в файле могут храниться индексы только для штатов Коннектикут и Делавэр, и тог-

да у пользователя не должно быть возможности фильтровать по штату Монтана, так как заведомо не будет найдено ни одной строки.

В пользовательском классе `UniqueProxyModel` используются те же столбцы, что в исходной модели, и при создании прокси-модели следует указать, в каком столбце должны быть уникальные значения. В качестве модели-источника для прокси задается исходная модель, а в качестве модели для комбинированного списка – прокси-модель. При этом столбец, который должен отображаться в списке, задается в методе `QComboBox::setModelColumn()`. (Класс `UniqueProxyModel` мы рассмотрим ниже, на стр. 126.)

В методе `createConnections()` создается около двадцати соединений между сигналами и слотами. Мы разобьем их на четыре группы и рассмотрим последовательно.

```
connect(model, SIGNAL(itemChanged(QStandardItem*)),
        this, SLOT(setDirty()));
connect(model, SIGNAL(rowsRemoved(const QModelIndex&,int,int)),
        this, SLOT(setDirty()));
connect(model, SIGNAL(modelReset()), this, SLOT(setDirty()));
```

Каждый из этих сигналов связан со слотом `setDirty()` (который, в свою очередь, вызывает метод `setWindowModified()`). Они уведомляют о том, что имеются несохраненные изменения, и этот факт индицируется звездочкой (*) в строке заголовка окна или точкой на кнопке закрытия в случае Mac OS X.

```
connect(countyGroupBox, SIGNAL(toggled(bool)),
        this, SLOT(updateUi()));
connect(countyComboBox,
        SIGNAL(currentIndexChanged(const QString&)),
        this, SLOT(updateUi()));
...
foreach (QRadioButton *radioButton, QList<QRadioButton*>())
    << dontFilterOrSelectRadioButton << filterRadioButton
    << selectByCriteriaRadioButton)
    connect(radioButton, SIGNAL(clicked()),
            this, SLOT(radioButtonClicked()));
```

Если пользователь отмечает (или сбрасывает) флажок `County` (Округ), мы должны узнать об этом, чтобы начать осуществлять фильтрацию или выборку по округу (либо, наоборот, прекратить это делать). Если при отмеченном флажке пользователь изменяет округ, то нужно заново произвести фильтрацию или выборку. (Имеются также соединения для флажка `State` (Штат) и комбинированного списка штатов, но они не показаны, так как ничем не отличаются от обработки округа.) Кроме того, установлены соединения между сигналом `valueChanged()` от счетчиков, задающих минимальное и максимальное значения почтового индекса, и слотом `updateUi()`; они тоже не показаны.

С помощью переключателя пользователь может установить один из трех режимов: фильтрацию, выборку или ни то ни другое. Соединения с сигналами переключателя используются для того, чтобы выполнять заказанную операцию (или не выполнять никакой).

```
connect(tableView, SIGNAL(clicked(const QModelIndex&)),
        this, SLOT(selectionChanged()));
connect(tableView->selectionModel(),
        SIGNAL(currentChanged(const QModelIndex&, const QModelIndex&)),
        this, SLOT(selectionChanged()));
connect(tableView->horizontalHeader(),
        SIGNAL(sectionClicked(int)), tableView, SLOT(sortByColumn(int)));
```

Первые два соединения в этой группе нужны для того, чтобы обновлять пользовательский интерфейс, когда пользователь переходит к какому-то элементу таблицы (щелчком мыши или с помощью клавиатуры), а третье – для сортировки. (Как работает сортировка, мы объясним ниже, на стр. 130.)

Отметим, что представления фактически ассоциированы с *двумя* моделями – той, что поставляет данные, и внутренней, которая отслеживает выборку. Модель выборки возвращает метод `QAbstractItemView::selectionModel()`, это объект класса `QItemSelectionModel`, наследующего непосредственно `QObject`, то есть не являющегося подклассом `QAbstractItemModel`.

```
connect(loadButton, SIGNAL(clicked()), this, SLOT(load()));
connect(saveButton, SIGNAL(clicked()), this, SLOT(save()));
connect(addButton, SIGNAL(clicked()), this, SLOT(addZipcode()));
connect(deleteButton, SIGNAL(clicked()), this, SLOT(deleteZipcode()));
connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
```

Соединения из этой последней группы не содержат ничего принципиально нового. Они отвечают за действия, выполняемые, когда пользователь просит загрузить или сохранить данные, добавить или удалить строки либо выйти из приложения.

Итак, мы описали общую структуру приложения и соединения, реализующие его поведение. Большинство закрытых слотов и методов относятся к пользовательскому интерфейсу, и с программированием моделей и представлений как таковым не связаны. Поэтому мы их опустим, а покажем только те фрагменты кода, которые имеют отношения к архитектуре модель/представление.

```
void MainWindow::radioButtonClicked()
{
    if (dontFilterOrSelectRadioButton->isChecked()) {
        proxyModel->clearFilters();
        QItemSelectionModel *selectionModel = tableView->selectionModel();
        selectionModel->clearSelection();
    }
    else
```

```
        updateUi();  
    }
```

Если пользователь выбирает переключатель Don't Filter or Select (Без фильтрации и выборки), то фильтры и селекторы очищаются, поэтому показываются все строки. В противном случае вызывается метод `updateUi()`, который производит фильтрацию или выборку в соответствии с заданным критерием.

```
void MainWindow::updateUi()  
{  
    if (loading || dontFilterOrSelectRadioButton->isChecked())  
        return;  
    if (filterRadioButton->isChecked())  
        restoreFilters();  
    else  
        performSelection();  
}
```

Когда пользователь изменяет состояние какого-либо флажка, списка или счетчика, вызывается этот метод. Если в данный момент происходит загрузка данных или фильтрация и выборка отключены, то мы ничего не делаем и сразу возвращаем управление. В противном случае фильтруем или выбираем значения в зависимости от установленного пользователем режима.

Метод `performSelection()` довольно длинный, поэтому рассмотрим его в два приема.

```
void MainWindow::performSelection()  
{  
    proxyModel->clearFilters();  
    int minimumZipcode = minimumZipSpinBox->value();  
    int maximumZipcode = maximumZipSpinBox->value();  
    QString county = countyGroupBox->isChecked()  
        ? countyComboBox->currentText() : QString();  
    QString state = stateGroupBox->isChecked()  
        ? stateComboBox->currentText() : QString();
```

Сначала очищаются фильтры, чтобы никакие строки не отфильтровывались. Затем получаем минимальное и максимальное значения почтового индекса и указанные пользователем округ и штат (или пустую строку, если соответствующий параметр не задан).

```
    QItemSelection selection;  
    int firstSelectedRow = -1;  
    for (int row = 0; row < proxyModel->rowCount(); ++row) {  
        QModelIndex index = proxyModel->index(row, Zipcode);  
        int zipcode = proxyModel->data(index).toInt();  
        if (zipcode < minimumZipcode || zipcode > maximumZipcode)  
            continue;
```

```

        if (!matchingColumn(county, row, County))
            continue;
        if (!matchingColumn(state, row, State))
            continue;
        if (firstSelectedRow == -1)
            firstSelectedRow = row;
        QTableWidgetItem rowSelection(index, index);
        selection.merge(rowSelection, QTableWidgetItem::Select);
    }
    QTableWidgetItem *selectionModel = tableView->selectionModel();
    selectionModel->clearSelection();
    selectionModel->select(selection, QTableWidgetItem::Rows|
        QTableWidgetItem::Select);
    if (firstSelectedRow != -1)
        tableView->scrollTo(proxyModel->index(firstSelectedRow, 0));
}

```

Выделение строится построчно. Начинаем мы с пустого объекта `QItemSelection` и обходим все строки в прокси-модели. (Это означает, что мы просматриваем все без исключения строки исходной модели, так как фильтры прокси-модели очищены.) Если строка отвечает критерию, то мы создаем содержащий ее объект `QItemSelection` и помещаем его в набор `selection` всех отображенных строк.

После того как все строки просмотрены, мы очищаем текущее выделение и заменяем его только что отображенными строками (которых, конечно, может и не быть). И напоследок, если есть хотя бы одна отображенная строка, мы прокручиваем таблицу-представление так, чтобы она стала видимой.

```

bool MainWindow::matchingColumn(const QString &value, int row, int column)
{
    if (value.isEmpty())
        return true;
    QModelIndex index = proxyModel->index(row, column);
    return value == proxyModel->data(index).toString();
}

```

Этот вспомогательный метод возвращает true, если переданное ему значение `value` совпадает с текстом, который находится в ячейке на пересечении заданных строки и столбца, а также если значение пусто. Так, если пользователь не ввел, например, округ, этот метод будет возвращать `true` для любого округа.

```

void MainWindow::restoreFilters()
{
    proxyModel->setMinimumZipcode(minimumZipSpinBox->value());
    proxyModel->setMaximumZipcode(maximumZipSpinBox->value());
    proxyModel->setCounty(countyGroupBox->isChecked()
        ? countyComboBox->currentText() : QString());
}

```

```

        proxyModel->setState(stateGroupBox->isChecked()
            ? stateComboBox->currentText() : QString());
        reportFilterEffect();
    }

```

Если пользователь устанавливает переключатель в положение Filter или изменяет состояние любого комбинированного списка либо счетчика, когда выбран переключатель Filter, то вызывается этот метод. Он обращается к методам нашей прокси-модели с целью установить в фильтре критерий согласно текущему состоянию интерфейса, а это действие, в свою очередь, приведет к обновлению представления.

```

void MainWindow::reportFilterEffect()
{
    if (loading)
        return;
    statusBar()->showMessage(tr("Filtered %L1 out of %Ln zipcode(s)",
        "", model->rowCount()).arg(proxyModel->rowCount()), StatusTimeout);
}

```

После того как пользователь выбрал переключатель Filter или изменил критерий фильтрации, вызывается этот метод, который показывает, сколько из общего числа строк в наборе данных отфильтровано.

Мы используем спецификаторы %L1, %L2 и т. д. вместо обычных %1, %2, чтобы в представлении групп цифр учитывались местные соглашения. Например, в США группы из трех цифр разделяются запятыми. Так удобнее воспринимать большие числа. Кроме того, мы хотим, чтобы переводчик правильно переводил форму множественного числа (например, «...out of one zipcode» или «...out of %Ln zipcodes»), но при необходимости мог откатиться к разумному умолчанию («...out of %Ln zipcode(s)»); эта тема обсуждается во врезке «Использование tr() с тремя аргументами» на стр. 302.

```

void MainWindow::addZipcode()
{
    dontFilterOrSelectRadioButton->click();
    QList<QStandardItem*> items;
    QStandardItem *zipItem = new QStandardItem;
    zipItem->setData(MinZipcode, Qt::EditRole);
    items << zipItem;
    for (int i = 0; i < model->columnCount() - 1; ++i)
        items << new QStandardItem(tr("(Unknown)"));
    model->appendRow(items);
    tableView->scrollToBottom();
    tableView->setFocus();
    QModelIndex index = proxyModel->index(proxyModel->rowCount() - 1, Zipcode);
    tableView->setCurrentIndex(index);
    tableView->edit(index);
}

```

Если пользователь хочет добавить новый почтовый индекс, то мы прежде всего отключаем фильтрацию и выборку. Это крайне важно, потому что если новый индекс не отвечает критерию фильтрации, то он сразу же будет отфильтрован, и у пользователя не останется никакого шанса не то что отредактировать, но даже увидеть добавляемые данные.

Поскольку мы используем простой подкласс `QStandardItemModel` и храним данные в объектах `QStandardItem`, то добавление нового индекса с редактированием на месте сводится к добавлению новой строки правильно инициализированных объектов `QStandardItem`. Новые данные добавляются в исходную модель, прокси-модель, ассоциированная с представлением, обнаруживает изменение и соответственно подстраивает себя. Есть один тонкий момент – для нестроковых данных, например почтового индекса, имеет смысл явно задать в качестве роли, сопоставляемой с хранимыми данными, `Qt::EditRole` и поручить Qt самостоятельно создать строковое представление, когда будут запрашиваться данные для роли `Qt::DisplayRole`. (Все существующие роли перечислены в табл. 3.2 на стр. 139.)

Добавив данные, мы перемещаемся в конец табличного представления (поскольку новый индекс добавлен в конец таблицы) и переходим в режим редактирования первого столбца в строке нового индекса, то есть пользователь сразу же оказывается в поле счетчика `QSpinBox` (на самом деле слегка модифицированного подкласса `QSpinBox`).

Мы не вызывали слот `setDirty()` и не устанавливали соединение с сигналом `QStandardItemModel::rowsInserted()`. Это означает, что после добавления к только что загруженному или сохраненному набору данных приложение *не* считает, что в нем есть несохраненные изменения. Однако если пользователь изменит хотя бы одну из ячеек в строке вновь добавленного почтового индекса (или вообще любую ячейку), то будет сгенерирован сигнал `itemChanged()`, а вот он уже соединен со слотом `setDirty()`.

Метод `deleteZipcode()` тоже довольно объемный, поэтому рассмотрим его в два приема.

```
void MainWindow::deleteZipcode()
{
    QItemSelectionModel *selectionModel =
        tableView->selectionModel();
    if (!selectionModel->hasSelection())
        return;
    QModelIndex index = proxyModel->mapToSource(
        selectionModel->currentIndex());
    if (!index.isValid())
        return;
    int zipcode = model->data(model->index(index.row(), Zipcode)).toInt();
    if (!AQP::okToDelete(this, tr("Delete Zipcode"),
        tr("Delete Zipcode %1?").arg(zipcode, 5, 10, QChar('0'))))
        return;
```

Если пользователь желает удалить строку, то сначала мы смотрим, есть ли в таблице выбранная ячейка, и если да, то преобразуем ее модельный индекс в соответствующий индекс исходной модели, для чего вызываем метод `QSortFilterProxyModel::mapToSource()`.

Зная строку, которую пользователь собрался удалить (то есть индекс строки, содержащей выбранную ячейку, в исходной модели), мы запрашиваем подтверждение, и если пользователь передумал, то ничего не делаем.

```
bool filtered = filterRadioButton->isChecked();
bool selected = selectByCriteriaRadioButton->isChecked();
QString county = countyGroupBox->isChecked()
    ? countyComboBox->currentText() : QString();
QString state = stateGroupBox->isChecked()
    ? stateComboBox->currentText() : QString();
dontFilterOrSelectRadioButton->click();
model->removeRow(index.row(), index.parent());
createComboBoxModels();
if (!county.isEmpty())
    countyComboBox->setCurrentIndex(
        countyComboBox->findText(county));
if (!state.isEmpty())
    stateComboBox->setCurrentIndex(
        stateComboBox->findText(state));
if (filtered)
    filterRadioButton->click();
else if (selected)
    selectByCriteriaRadioButton->click();
}
```

Перед тем как удалять строку, мы запоминаем состояние переключателей Filter и Select by Criteria, а также комбинированных списков, задающих критерий, затем отключаем фильтрацию и выборку, «щелкая» по переключателю Don't Filter or Select. После этого мы удаляем строку и восстанавливаем тот режим фильтрации или выборки, который действовал перед удалением. Строго говоря, сохранять и восстанавливать состояние фильтрации/выборки необязательно, но вот заново создать модели комбинированных списков необходимо, потому что в них теперь может стать на одну строку меньше – например, если пользователь удалил ту единственную строку, которая содержала какой-то округ или штат.

Поскольку в разных приложениях очень часто приходится что-то удалять, то для удобства мы написали функцию `AQP::okToDelete()`, которую приводим здесь для полноты:

```
bool okToDelete(QWidget *parent, const QString &title,
                const QString &text, const QString &detailedText)
{
    QScopedPointer<QMessageBox> messageBox(new QMessageBox(parent));
    if (parent)
        messageBox->setWindowModality(Qt::WindowModal);
}
```

```

messageBox->setIcon(QMessageBox::Question);
messageBox->setWindowTitle(QString("%1 - %2")
    .arg(QApplication::applicationName()).arg(title));
messageBox->setText(text);
if (!detailedText.isEmpty())
    messageBox->setInformativeText(detailedText);
QAbstractButton *deleteButton = messageBox->addButton(
    QObject::tr("&Delete"), QMessageBox::AcceptRole);
messageBox->addButton(QObject::tr("Do &Not Delete"),
    QMessageBox::RejectRole);
messageBox->setDefaultButton(
    qobject_cast<QPushButton*>(deleteButton));
messageBox->exec();
return messageBox->clickedButton() == deleteButton;
}

```

Если не считать типа возвращаемого значения, сигнатура этой функции такая же, как у функций `AQP::information()` и `AQP::warning()`, а создание и конфигурирование окна сообщений тоже очень похоже на то, что делается в этих функциях. У параметра `detailedText` есть значение по умолчанию – пустая строка, поэтому при вызове его можно опускать. (О том, почему мы используем нестандартные функции для вывода окон сообщений, см. врезку «Избегайте статических вспомогательных функций `QMessageBox`» на стр. 77.) Приведение `qobject_cast<>()` необходимо потому, что метод `QMessageBox::setDefaultButton()` ожидает получить указатель на `QPushButton`, а мы использовали для `deleteButton` указатель на `QAbstractButton`, чтобы упростить сравнение `QMessageBox::clickedButton()`, так как этот метод возвращает указатель на `QAbstractButton`. (С классом `QScopedPointer`, появившимся в версии **Qt 4.6**, мы ознакомились на врезке «Интеллектуальные указатели в Qt» на стр. 78.)¹

Теоретически можно было бы гарантировать удаление диалогового окна, обратившись к методу `QWidget::setAttribute(Qt::WA_DeleteOnClose)`, но на практике мы предпочитаем, чтобы об удалении заботился интеллектуальный указатель. Это означает, что окно сообщения при нажатии любой кнопки просто закрывается, а не удаляется, то есть после вызова `QMessageBox::exec()` окно все еще существует, что крайне важно, если мы хотим проверить, какая кнопка была нажата, а мы этого хотим.

Имейте в виду, что метод `QDialog::exec()` выходит из моды, по крайней мере в крупных проектах. Проблема в том, что, хотя с точки зрения пользователя он является блокирующим (то есть не дает взаимодействовать с другими окнами приложения), на самом деле он не приостанавливает обработку событий. Это означает, что состояние приложения может существенно измениться между моментом обращения к `exec()` и временем, когда пользователь примет решение о подтверждении или отмене. Вполне может статься, что и само диалоговое окно будет случай-

¹ В исходном коде имеется директива `#if QT_VERSION`, так что при компиляции для Qt 4.5 используется класс `QSharedPointer`.

но удалено. Поэтому безопаснее пользоваться методом `QDialog::open()` (или `QDialog::show()` для немодальных диалоговых окон), а для реагирования на подтверждение устанавливать соединение между сигналом и слотом. Тем не менее, хотя мы понимаем, какие проблемы и опасности связаны с использованием метода `exec()`, ни одному из примеров в этой книге они не угрожают, поэтому мы по-прежнему пользуемся им, поскольку он удобен и кода нужно писать меньше, чем в случае `open()`¹.

Мы рассмотрели все методы, относящиеся к манипулированию моделями через пользовательский интерфейс. В следующем разделе мы будем говорить о подклассе `QStandardItemModel`, который используется для загрузки, редактирования и сохранения данных приложения. А в двух разделах, следующих за ним, рассмотрим подклассы `QSortFilterProxyModel`, применяемые для фильтрации данных и обеспечения уникальности значений в комбинированных списках, в которых задаются критерии фильтрации и выборки.

Подкласс `QStandardItemModel` для таблиц

Класс `QStandardItemModel` предоставляет всю функциональность, необходимую для манипулирования табличными данными и взаимодействия с представлениями. Единственное, что нужно добавить для работы с реальными данными, — это возможность загружать данные из файла и сохранять их в файле, а также средства для очистки данных — чтобы пользователь мог создать новый набор данных.

Начнем с рассмотрения конструктора и метода `clear()`, а затем посмотрим, как данные загружаются и сохраняются.

```
StandardTableModel::StandardTableModel(QObject *parent)
    : QStandardItemModel(parent)
{
    initialize();
}
void StandardTableModel::initialize()
{
    setHorizontalHeaderLabels(QStringList() << tr("Zipcode")
                                << tr("Post Office") << tr("County") << tr("State"));
}
```

Мы вынесли код инициализации в отдельный метод `initialize()`, потому что он используется также в методе `clear()`.

```
void StandardTableModel::clear()
{
    QStandardItemModel::clear();
    initialize();
}
```

¹ См. статью «New Ways of Using Dialogs» в журнале *Qt Quarterly*, выпуск 30, по адресу qt.nokia.com/doc/qq/.

Обратите внимание, что этот метод, да и весь класс `StandardTableModel`, ничего не знает о том, изменились данные модели или нет, поэтому обработка несохраненных изменений – задача клиентов этого класса, в данном случае класса главного окна.

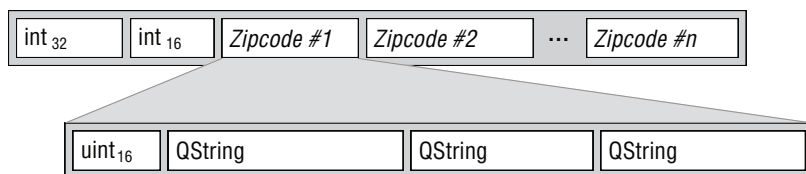


Рис. 3.4. Формат файла почтовых индексов

В этом подклассе всего два дополнительных метода: `load()` и `save()`. Данные о почтовых индексах хранятся в двоичном файле, структура которого показана на рис. 3.4. В начале файла находится сигнатура (магическое число) и номер версии формата, затем нуль или более записей. В каждой записи хранится почтовый индекс, название почтового отделения, округ и штат.

```
const quint32 MagicNumber = 0x5A697043;
const quint16 FormatNumber = 100;

void StandardTableModel::save(const QString &filename)
{
    if (!filename.isEmpty())
        m_filename = filename;
    if (m_filename.isEmpty())
        throw AQP::Error(tr("no filename specified"));
    QFile file(m_filename);
    if (!file.open(QIODevice::WriteOnly))
        throw AQP::Error(file.errorString());

    QDataStream out(&file);
    out << MagicNumber << FormatNumber;
    out.setVersion(QDataStream::Qt_4_5);
    for (int row = 0; row < rowCount(); ++row) {
        out << static_cast<quint16>(<
            item(row, Zipcode)->data(Qt::EditRole).toUInt())
            << item(row, PostOffice)->text()
            << item(row, County)->text() << item(row, State)->text();
    }
}
```

Сохранение файла не вызывает никаких сложностей. Сначала открываем файл, указывая имя, использовавшееся в последний раз, или новое имя, если пользователь его задал; по умолчанию параметр `filename` равен пустой строке. Если имя файла не указано при вызове или не удалось открыть файл, возбуждается исключение. Его перехватит ме-

тод `MainWindow::save()`, вызвавший данный метод, и выведет сообщение об ошибке.

Открыв файл, мы записываем в него магическое число и номер версии формата. Мы используем (надемся) уникальное магическое число для идентификации формата нашего файла почтовых индексов, а номер версии уточняет особенности его конкретной версии. (Магические числа описаны во врезке «Типы MIME, файловые суффиксы и магические числа» на стр. 89.) Наличие номера версии сильно упрощает последующее внесение изменений в формат, поскольку программа может проанализировать номер и подстроиться под нужную версию. Далее мы устанавливаем версию `Qt_4_5` для класса `QDataStream` (чтобы данные могли быть прочитаны программами, собранными с библиотекой Qt версии 4.5 и более поздних) и приступаем к записи данных. Поскольку данные содержат только числа и строки, то можно было бы установить и более раннюю версию сериализатора Qt (например, `Qt_4_0`). Плюсы использования самой свежей версии состоят в том, что она, возможно, поддерживает больше встроенных в Qt типов, чем предыдущие, быстрее загружает или сохраняет данные либо представляет их более компактно. Мы всегда предпочитаем использовать версию сериализатора, работающую с самой старой версией Qt, которую мы собираемся поддерживать на уровне сборки приложения.

Формат фундаментальных типов данных, в частности целых, не зависит от версии `QDataStream`, поэтому их можно без опаски записывать еще до установки версии сериализатора. Однако представление собственных типов Qt и чисел с плавающей точкой в разных версиях может различаться, поэтому нужно следить за тем, чтобы при записи и чтении таких типов использовались одинаковые версии `QDataStream`.¹ Отметим также, что ни при записи, ни при чтении чисел с плавающей точкой *ни в коем случае* не следует использовать тип `qreal`, потому что его размер зависит от платформы; всегда нужно явно указывать тип `float` или `double`.

Для вывода строковых данных можно использовать текст из объекта `QStandardItem`, но для нестроковых (например, самого почтового индекса) необходимо указывать ту роль, в которой хранятся данные. Обычно – и в нашей программе тоже – это роль `Qt::EditRole`. А при записи целых чисел важно указывать точный тип, в Qt они представлены во всем многообразии – от `qint8` и `quint8` до `qint64` и `quint64`.

Еще один способ сохранить и загрузить объекты `QStandardItem` – воспользоваться потоками непосредственно, потому что эти объекты под-

¹ Представление чисел с плавающей точкой изменилось при переходе от Qt 4.5 к Qt 4.6. Но если устанавливать номер версии `QDataStream` с помощью метода `setVersion()`, то такие изменения не вызовут никаких проблем и не окажут влияния ни на код, ни на данные.

держивают глобальные функции Qt `operator<<()` и `operator>>()`, которые принимают в качестве первого аргумента `QDataStream`. А если требуется добиться более точного контроля над тем, что записывается и читается, то можно создать подкласс `QStandardItem` и переопределить в нем методы `QStandardItem::read()` и `QStandardItem::write()`, которые вызывают операторы потокового ввода/вывода. Но в данном случае мы решили воспользоваться немодифицированным классом `QStandardItem`, а чтением и записью управлять в нашем подклассе `QStandardItemModel`.

Класс `Error`, которым мы пользуемся для возбуждения исключений, находится в модуле `aqp.{hpp,cpp}`; приведем его для полноты картины.

```
class Error : public std::exception
{
public:
    explicit Error(const QString &message) throw()
        : message(message.toUtf8()) {}
    ~Error() throw() {}
    const char *what() const throw() { return message; }

private:
    const char *message;
};
```

Ничего необычного в этом классе нет, но мы предпочитаем передавать сообщение об ошибке в виде `QString`, а не `char*`. Чтобы получить указатель на текст сообщения, следует воспользоваться методом `QString::fromUtf8(error.what())`.

Метод загрузки `load()` несколько длиннее, чем метод сохранения, поэтому мы разобьем его на четыре коротеньких фрагмента.

```
void StandardTableModel::load(const QString &filename)
{
    ...
    QDataStream in(&file);
    qint32 magicNumber;
    in >> magicNumber;
    if (magicNumber != MagicNumber)
        throw AQP::Error(tr("unrecognized file type"));
    qint16 formatVersionNumber;
    in >> formatVersionNumber;
    if (formatVersionNumber > FormatNumber)
        throw AQP::Error(tr("file format version is too new"));
    in.setVersion(QDataStream::Qt_4_5);
    clear();
}
```

Строки, касающиеся обработки имени файла (употреблявшегося в последний раз или вновь заданного), почти такие же, как в `save()`, поэтому мы их опустили. Открыв файл (на этот раз в режиме `QIODevice::ReadOnly`), мы считываем магическое число, дабы убедиться, что это именно файл почтовых индексов, и номер версии формата. В данном случае мы толь-

ко проверяем версию, но именно здесь можно было бы адаптироваться к конкретному формату, если бы возникла такая необходимость. Затем мы устанавливаем правильную версию сериализатора и очищаем модель, удаляя из нее все данные.

```
quint16 zipcode;
QString postOffice;
QString county;
QString state;
QMap<quint16, QList<QStandardItem*> > itemsForZipcode;
```

Обычно мы заносим в модель данные в том порядке, в котором читаем записи из файла. Но в данном случае мы хотим, чтобы данные были с самого начала упорядочены, поэтому предварительно считываем их в локальный объект QMap с почтовым индексом в качестве ключа (в виде списка QList указателей на QStandardItem), а в самом конце копируем в модель.

```
while (!in.atEnd()) {
    in >> zipcode >> postOffice >> county >> state;
    QList<QStandardItem*> items;
    QStandardItem *item = new QStandardItem;
    item->setData(zipcode, Qt::EditRole);
    items << item;
    foreach (const QString &text, QStringList() << postOffice
                                                    << county << state)
        items << new QStandardItem(text);
    itemsForZipcode[zipcode] = items;
}
```

Для каждой прочитанной записи создается список QList<QStandardItem*>. Строковые данные можно передавать конструктору QStandardItem непосредственно, но нестроковые *крайне важно* представлять в конкретной роли, обычно Qt::EditRole; именно так мы и поступаем для почтовых индексов. Сформировав список элементов, представляющих одну строку данных, мы вставляем его в QMap, указывая в качестве ключа почтовый индекс. Поскольку QMap – упорядоченная структура данных, то индексы будут храниться в ней в отсортированном виде (по возрастанию).

```
QMapIterator<quint16, QList<QStandardItem*> >
    i(itemsForZipcode);
while (i.hasNext())
    appendRow(i.next().value());
}
```

Прочитав все данные, мы обходим QMap и добавляем все хранящиеся в нем списки объектов QStandardItem в виде новых строк таблицы. Уведомлять ассоциированные представления о том, что данные модели изменились, нет необходимости, так как об этом позаботится метод базового класса appendRow().

Выше мы отмечали, что загрузка данных в модель `QStandardItemModel` на всех протестированных нами машинах неизменно оказывалась медленнее, чем загрузка в специально написанную модель. Это справедливо даже для случая, когда данные загружаются непосредственно (без промежуточного отображения `QMap`, обеспечивающего упорядоченность), так что определяющим фактором, по-видимому, является создание отдельных объектов `QStandardItem`.

На этом мы завершаем обзор подкласса `StandardTableModel`. Поскольку мы добавили лишь возможности загрузки, сохранения и очистки данных, а ответственность за обработку ошибок и несохраненных изменений переложили на клиентов класса, то реализация оказалась несложной. В следующем разделе мы создадим подкласс `QAbstractTableModel`, который можно будет просто подставить вместо `StandardTableModel`, и сравним оба решения. Но сначала поговорим о подклассах `QSortFilterProxyModel`, которые использовались в нашем приложении при работе с объектами `QTableView` и `QComboBox`.

Удаление строк-дубликатов с помощью класса `QSortFilterProxyModel`

В приложении `Zipcodes` есть два комбинированных списка для отображения округов и штатов, с помощью которых пользователь может задавать критерий фильтрации или выборки. Оба списка должны получать данные из исходной модели, в которой хранятся сведения обо всех почтовых индексах. Однако к одному округу или штату может относиться много индексов, а мы не хотим, чтобы в списках присутствовали дубликаты, поэтому использовать исходную модель непосредственно не можем.

Мы решили эту проблему, создав специализированный подкласс класса `QSortFilterProxyModel`, который отфильтровывает строки модели, содержащие повторяющиеся значения в указанном столбце. Для фильтрации строк необходимо реализовать защищенный метод `filterAcceptsRow()`, а для фильтрации столбцов – метод `filterAcceptsColumn()`. (Для сортировки можно реализовать защищенный метод `QSortFilterProxyModel::lessThan()`, хотя и сам класс `QSortFilterProxyModel` уже обеспечивает сортировку и, как будет объяснено ниже, мы воспользовались этим в приложении `Zipcodes`.) В данном конкретном случае необходимо также переопределить метод `setSourceModel()`, но об этом чуть позже. Сначала приведем определение из заголовочного файла.

```
class UniqueProxyModel : public QSortFilterProxyModel
{
    Q_OBJECT
public:
    explicit UniqueProxyModel(int column, QObject *parent=0)
        : QSortFilterProxyModel(parent), Column(column) {}
```

```

void setSourceModel(QAbstractItemModel *sourceModel);

protected:
    bool filterAcceptsRow(int sourceRow,
                          const QModelIndex &sourceParent) const;

private slots:
    void clearCache() { cache.clear(); }

private:
    const int Column;
    mutable QSet<QString> cache;
};

```

Мы включили в класс изменяемый кэш, где храним все уникальные строки, которые объект UniqueProxyModel уже встречал в указанном столбце.

```

void UniqueProxyModel::setSourceModel(QAbstractItemModel *sourceModel)
{
    connect(sourceModel, SIGNAL(modelReset()), this, SLOT(clearCache()));
    QSortFilterProxyModel::setSourceModel(sourceModel);
}

```

Если модель-источник переустанавливается, то мы очищаем кэш, поскольку имеем совершенно новые данные. Надо ли создавать соединения, которые очищают кэш при вставке и удалении строк? В случае вставки в этом нет необходимости, потому что новые строки автоматически обрабатываются правильно в силу выбранного нами способа реализации метода `filterAcceptsRow()`. Что же касается удаленных строк, то удалять их из кэша или нет, зависит от того, хотим ли мы, чтобы пользователь мог фильтровать по значениям, которых уже, возможно, нет в исходной модели. В данном случае мы решили, что вреда от этого не будет.

```

bool UniqueProxyModel::filterAcceptsRow(int sourceRow,
    const QModelIndex &sourceParent) const
{
    QModelIndex index = sourceModel()->index(sourceRow, Column, sourceParent);
    const QString &text = sourceModel()->data(index).toString();
    if (cache.contains(text))
        return false;
    cache << text;
    return true;
}

```

Для каждого комбинированного списка устанавливается свой экземпляр UniqueProxyModel, в котором задан нужный столбец (County или State); это делалось в методе createComboBoxModel() (стр. 112). Когда комбинированному списку требуется доступ к строкам (например, при

раскрытии его содержимого), прокси-модель вызывает этот метод, чтобы отфильтровать ненужные строки.

Алгоритм очень простой. Получаем модельный индекс исходной модели, соответствующий строке прокси-модели, и извлекаем из модели текст того столбца, который был задан при создании экземпляра прокси-модели. Если этот текст уже имеется в кэше, то есть мы встречали его ранее, то возвращаем `false`, извещая о том, что эту строку следует отфильтровать. В противном случае добавляем текст в кэш (в результате строки, в которых он снова встретится в указанном столбце, будут отфильтрованы) и возвращаем `true`, разрешая тем самым включить данные из этой строки – в первый и последний раз.

Применение класса `QSortFilterProxyModel` для включения нужных строк

Рассмотренный выше класс `UniqueProxyModel` весьма полезен, но специализирован для конкретного случая. В приложениях `Zipcodes` нам была необходима гораздо более развитая прокси-модель, позволяющая фильтровать данные на основе комбинации критериев, заданных пользователем: минимального и максимального почтового индекса и, возможно, округа и штата. Для этой цели мы создали класс `ProxyModel`; как и в классе `UniqueProxyModel`, фильтрация реализована в методе `filterAcceptsRow()`.

Подкласс `ProxyModel`, помимо конструктора, предоставляет методы чтения и установки критерия фильтрации, а также методы `filterAcceptsRow()` и `clearFilters()`. Мы рассмотрим их все, но для методов чтения (getter) и установки (setter) ограничимся только одной парой, потому что все они структурно однотипны.

```
ProxyModel::ProxyModel(QObject *parent)
    : QSortFilterProxyModel(parent)
{
    m_minimumZipcode = m_maximumZipcode = InvalidZipcode;
}
```

Закрытым переменным, содержащим минимальный и максимальный почтовые индексы, присваиваются недопустимые значения. Такое значение служит методу `filterAcceptsRow()` указанием на то, что сравнение с почтовым индексом можно пропустить.

```
QString state() const { return m_state; }
```

Определение этого метода чтения находится в заголовочном файле; существуют аналогичные методы для получения округа и минимального и максимального индексов.

```
void ProxyModel::setState(const QString &state)
{
    if (m_state != state) {
```

```

        m_state = state;
        invalidateFilter();
    }
}

```

Все методы установки устроены аналогично: если значение изменилось, то сначала сохраняем новое значение, а затем вызываем защищенный метод `QSortFilterProxyModel::invalidateFilter()`, чтобы прокси-модель уведомила все ассоциированные с ней представления о том, что они должны обновить показанные на экране данные.

Если передается пустая строка, то фильтрация по штату отключается. То же самое касается и метода установки округа.

```

void ProxyModel::clearFilters()
{
    m_minimumZipcode = m_maximumZipcode = InvalidZipcode;
    m_county.clear();
    m_state.clear();
    invalidateFilter();
}

```

Строки вообще не отфильтровываются, если одновременно выполнены следующие условия: минимальный и максимальный почтовые индексы имеют недопустимое значение, округ и штат – пустые строки. Поэтому вызов этого метода, по сути, означает отключение фильтрации.

```

bool ProxyModel::filterAcceptsRow(int sourceRow,
    const QModelIndex &sourceParent) const
{
    if (m_minimumZipcode != InvalidZipcode ||
        m_maximumZipcode != InvalidZipcode) {
        QModelIndex index = sourceModel()->index(sourceRow, Zipcode,
                                                    sourceParent);

        if (m_minimumZipcode != InvalidZipcode &&
            sourceModel()->data(index).toInt() < m_minimumZipcode)
            return false;

        if (m_maximumZipcode != InvalidZipcode &&
            sourceModel()->data(index).toInt() > m_maximumZipcode)
            return false;
    }

    if (!m_county.isEmpty()) {
        QModelIndex index = sourceModel()->index(sourceRow, County,
                                                    sourceParent);

        if (m_county != sourceModel()->data(index).toString())
            return false;
    }

    if (!m_state.isEmpty()) {
        QModelIndex index = sourceModel()->index(sourceRow, State,
                                                    sourceParent);

        if (m_state != sourceModel()->data(index).toString())
            return false;
    }
}

```



```
    }  
    return true;  
}
```

Фильтр прокси-модели возвращает `true` (оставить строку) для всех вообще строк, если минимальный и максимальный почтовые индексы принимают недопустимое значение, а округ и штат – пустые строки.

Почтовый индекс считается допустимым, если он больше или равен `MinZipcode`. Если минимальный индекс допустим и в рассматриваемой строке значение индекса меньше, то возвращаем `false`, чтобы отфильтровать эту строку. Аналогично, если максимальный индекс допустим и в рассматриваемой строке значение индекса больше, тоже возвращаем `false`. Если название округа не пусто, то отфильтровываются все строки, в которых округ имеет другое значение; аналогично для штатов. Если мы доходим до конца метода, то возвращаем `true` (оставить строку).

Как видно на примере этого и рассмотренного в предыдущем разделе фильтра, создание подклассов класса `QSortFilterProxyModel` для реализации фильтрации не вызывает сложностей. И разумеется, фильтры можно сцеплять – ценой небольшой потери производительности.

Создание подклассов – не единственный способ использования класса `QSortFilterProxyModel`. Можно создавать его экземпляры непосредственно и с помощью метода `setFilterKeyColumn()` задавать столбец, по которому производится фильтрация, а с помощью `setFilterRegExp()` – регулярное выражение, которому должно соответствовать значение в этом столбце, чтобы строка не была отфильтрована. (Можно также использовать сравнение с фиксированными строками и шаблонами с метасимволами¹.)

Что касается сортировки, то и тут есть разные подходы. Когда представление получает требование отсортировать данные, оно вызывает метод `QAbstractItemModel::sort()`, поэтому один из способов реализовать сортировку – переопределить этот метод, поскольку его версия в базовом классе вообще ничего не делает. В приложениях `Zipcodes` представление вызывает метод `QSortFilterProxyModel::sort()`, поскольку в качестве модели мы установили прокси-модель. Содержащаяся в этом классе реализация умеет сортировать целые числа и некоторые простые типы `Qt`, например `QString` и `QDateTime`. Именно поэтому для сортировки модели `Zipcodes` оказалось достаточно создать объект `QSortFilterProxyModel` и установить соединение между сигналом `sectionClicked()` от горизонтального заголовка и слотом `sortByColumn()` представления.

Можно организовать и более точный контроль над прокси-моделью с помощью методов `QSortFilterProxyModel::setSortCaseSensitivity()` и `QSortFilterProxyModel::setSortLocaleAware()`. Или создать подкласс `QSortFilterProxyModel` и переопределить в нем метод `lessThan()`.

¹ Методы называются `setFilterFixedString()` и `setFilterWildcard()` соответственно. – *Прим. науч. ред.*

Еще один способ сортировки – для подклассов `QStandardItemModel` – состоит в том, чтобы воспользоваться методом `QStandardItemModel::setSortRole()`, например указать, что сортировать следует данные, ассоциированные с ролью `Qt::UserRole`. Чтобы этот подход заработал, мы должны сообщить представлению, что оно должно поддерживать сортировку, обратившись к методу `setSortingEnabled(true)`, и позаботиться о том, чтобы в каждом элементе, помимо обычных данных, используемых в режиме отображения или редактирования, хранились еще данные для пользовательской роли (или той, которую мы назначаем ролью для сортировки). Предположим, к примеру, что имеются текстовые элементы на английском языке. В роли `Qt::DisplayRole` можно было бы хранить истинные данные, а в роли `Qt::UserRole` – те же данные, но в нижнем регистре и без начального артикля (The, An, A), чтобы сортировка выглядела более естественно.

QStandardItemModel и пользовательские модели

Когда мы используем для представления данных – будь то список, таблица или дерево – класс `QStandardItemModel` (или его подкласс), все элементы данных хранятся в виде объектов класса `QStandardItem` (или производного от него подкласса).

Концептуально класс `QStandardItemModel` занимает промежуточное положение между комбинацией виджета-представления и модели и готовым виджетом, в который модель уже встроена. Использовать `QStandardItemModel` проще, чем создавать пользовательскую модель, потому что зачастую мы можем работать с ним непосредственно, а даже если приходится создавать подкласс, все равно, как правило, достаточно добавить всего несколько методов, например для загрузки и сохранения данных. Еще одна причина, по которой разработчики предпочитают класс `QStandardItemModel`, заключается в том, что он предоставляет знакомый API, основанный на работе с элементами, а не с индексами, как в пользовательских моделях.

Класс `QStandardItem` предлагает развитый API, благодаря которому использовать его в готовом виде легко и удобно. Чаще всего применяют методы для получения и установки цвета фона, флажков, редактируемости, шрифта, цвета текста, пиктограммы, информации о состоянии, самого текста и его выравнивания, всплывающей подсказки. Дополнительные данные можно хранить в ролях, которые сам класс `QStandardItem` не использует, например `Qt::UserRole`, `Qt::UserRole + 1` и т. д. Кроме того, данные можно записывать в поток и читать из потока типа `QDataStream`.

Но за все это богатство возможностей и удобство приходится расплачиваться, по крайней мере теоретически, объемом памяти и быстродействием. В пользовательской модели можно не хранить самостоятельные элементы вообще или хранить только простые данные, например числа или строки.

Но в любом случае проще и быстрее всего начинать с классов `QStandardItemModel` и `QStandardItem` (или их подклассов). С их помощью можно быстро создать работающий прототип. Если впоследствии мы обнаружим, что потребление памяти и/или скорость работы неприемлемы, то сможем подумать о подстановке пользовательской модели вместо стандартной.

Пользовательская модель может принести ощутимую выгоду в ситуациях, когда сами элементы очень просты, когда нам не нужно большинство функций, встроенных в `QStandardItem`, или когда количество элементов велико (тысячи и больше). Кроме того, в случае древовидных моделей API класса `QStandardItemModel` предлагает меньше возможностей, чем можно получить при работе с пользовательской моделью.

Создание пользовательских списковых и табличных моделей не вызывает особых сложностей, поэтому при работе с большими наборами данных они потенциально могут дать заметный выигрыш без особых затрат. Написание древовидной пользовательской модели требует больших усилий, и добиться ее правильной работы не всегда просто, но бывает, что это единственный способ получить требуемую функциональность, например возможность произвольно перемещать элементы (вместе с потомками) в дереве, при условии, конечно, что для набора данных, для которого создана модель, это имеет смысл.

Создание пользовательских табличных моделей

Использование класса `QStandardItemModel` — обычно самая простая и удобная отправная точка в программе, где требуется визуально представить данные в Qt. Однако объекты класса `QStandardItem`, из которых состоит модель `QStandardItemModel`, часто более тяжеловесны (в частности, потребляют много памяти), чем необходимо в конкретной задаче, потому что рассчитаны на общий случай.

В этом разделе мы заменим простой подкласс `QStandardItemModel`, который применяли в приложении `zipcodes1`, пользовательской моделью и получим в результате приложение `zipcodes2`. У обоих приложений одинаковый внешний вид и сходное поведение, но `zipcodes2` показывала более высокую скорость загрузки данных на всех компьютерах, на которых тестировалась.

В новом приложении появилось всего три новых файла: *zipcodeitem.hpp*, *tablemodel.hpp* и *tablemodel.cpp*. Все остальные взяты из приложения *zipcodes1* (за исключением, разумеется, файлов *standardtablemodel.{hpp,cpp}*, которые уже не нужны). В файл проекта *zipcodes2.pro* добавлена строка `DEFINES += CUSTOM_MODEL`, а в файлы *zipcodes1* помещены директивы `#ifdef` в тех местах, где код отличается от *zipcodes2*.

Поскольку методы главного окна в приложениях *zipcodes1* и *zipcodes2* по большей части одинаковы, мы рассмотрим только те, где есть отличия: конструктор, `createConnections()` и `addZipcode()`. Директивы `#ifdef` мы не показываем, а приводим лишь код, который видит компилятор, когда определена константа препроцессора `CUSTOM_MODEL`. (В предыдущем разделе мы поступали точно так же: показывали код, который компилятор видит, когда константа `CUSTOM_MODEL` не определена.)

Изменение табличной модели с помощью пользовательского интерфейса

Конструктор почти не отличается от конструктора в приложении *zipcodes1*, но взглянуть на него стоит, потому что здесь присутствует нюанс, с которым мы раньше не встречались.

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), loading(false)
{
    model = new TableModel(this);
#ifdef MODEL_TEST
    (void) new ModelTest(model, this);
#endif
    proxyModel = new ProxyModel(this);
    proxyModel->setSourceModel(model);

    createWidgets();
    createComboBoxModels();
    createLayout();
    createConnections();

    AQP::accelerateWidget(this);
    setWindowTitle(tr("%1 (Custom Model)[*]")
        .arg(QApplication::applicationName()));
    statusBar()->showMessage(tr("Ready"), StatusTimeout);
}
```

Единственное отличие от *zipcodes1* состоит в использовании класса `TableModel` (подкласса абстрактного класса `QAbstractTableModel`) вместо подкласса `QStandardItemModel`. Ну еще заголовок окна другой.

В обоих приложениях мы создаем также объект `ModelTest` (хотя раньше его не показывали). Он служит для тестирования моделей и взят с сайта Qt Development Frameworks по адресу developer.qt.nokia.com/wiki/Model_Test. Использовать его очень просто. Скачав его код и поместив,

например, в параллельный каталог, вы должны будете добавить его в *pro*-файл своего проекта. Ниже приведены соответствующие строки из файла *zipcodes2.pro* (точно такие же есть и в *zipcodes1.pro*):

```
.pro- | exists(..../modeltest-0.2/modeltest.pri) {
файл |     DEFINES += MODEL_TEST
      |     include(..../modeltest-0.2/modeltest.pri)
      | }
      |
```

Мы используем реализованную в программе `qmake` функцию `exists()`: определяем макрос `MODEL_TEST` и включаем *pri*-файл только в случае, когда он действительно существует. Кроме того, необходимо включить в *pro*-файл строчку `CONFIG += debug`, иначе все утверждения `Q_ASSERT`, встречающиеся в тесте модели, будут удалены компилятором.

Необходимо еще включить заголовочный файл в тот исходный файл, где создается объект `ModelTest`, в нашем случае *mainwindow.cpp*:

```
#ifdef MODEL_TEST
#include <modeltest.h>
#endif
```

Все относящиеся к тестированию модели строки находятся между директивами `#ifdef`, так что приложение соберется даже, если код теста модели отсутствует.

Последний шаг – создание экземпляра `ModelTest` и передача ему модели, подлежащей тестированию; это тоже делается в конструкторе главного окна. Больше писать ничего не надо, но если теперь мы допустим при работе с пользовательской моделью какую-нибудь ошибку из числа распознаваемых `ModelTest`, то последний заявит о наличии проблемы. Для табличных пользовательских моделей типа той, что мы сейчас создадим, все достаточно просто, поэтому `ModelTest` может и не понадобится, но в случае древовидных моделей, о которых речь пойдет в следующей главе, использование `ModelTest` может принести реальную пользу.

Между методами `createConnections()` в обоих приложениях `Zipcodes` есть лишь одно отличие. Если в *zipcodes1* мы соединяли сигнал `itemChanged()` от модели `QStandardItemModel` со слотом `setDirty()`, то в *zipcodes2* слот `setDirty()` соединяется с сигналом `dataChanged()` от пользовательской модели `TableModel`.

Единственный метод, в котором имеются существенные отличия, – это `addZipcode()`; в приложении *zipcodes2* он несколько проще. (Реализация в приложении *zipcodes1* приведена на стр. 118.)

```
void MainWindow::addZipcode()
{
    dontFilterOrSelectRadioButton->click();
    if (!model->insertRow(model->rowCount()))
        return;
    tableView->scrollToBottom();
    eableView->setFocus();
}
```

```
QModelIndex index = proxyModel->index(proxyModel->rowCount() - 1, Zipcode);  
tableView->setCurrentIndex(index);  
tableView->edit(index);  
}
```

В отличие от `zipcodes1`, здесь мы не обязаны создавать все элементы для новой строки. Вместо этого вызываем метод `QAbstractTableModel::insertRow()`, который, в свою очередь, вызывает `insertRows()` со значением счетчика, равным 1. Для пользовательских моделей, допускающих вставку, мы в любом случае должны переопределить метод `insertRows()`, так что никакой добавочной работы нет.

Все остальные классы и методы, общие для обоих приложений, в точности одинаковы, в частности ни пользовательские делегаты, ни обе пользовательские прокси-модели (`UniqueProxyModel` и `ProxyModel`) не поменялись. Равным образом и поведения приложений не претерпели существенных изменений при переходе от подкласса `QStandardItemModel` к подклассу `QAbstractTableModel`. Конечно, за кулисами для создания подкласса `QAbstractTableModel` нам предстоит проделать заметно больше работы (так как `QStandardItemModel` изначально предоставляет весьма обширную функциональность), но взамен мы получаем гораздо более точный контроль и потенциально более высокую производительность.

Эта возможность подставить одну модель вместо другой (при условии, что обе обрабатывают одни и те же данные) без внесения изменений в представления и делегаты есть одно из главных преимуществ архитектуры модель/представление в Qt.

Пользовательский подкласс `QAbstractItemModel` для таблиц

В этом разделе мы опишем создание подкласса класса `QAbstractTableModel`. Как и для всех подклассов `QAbstractItemModel` в архитектуре модель/представление в Qt, мы должны переопределить ряд методов так, чтобы API нашего подкласса модели был совместим с архитектурой и мог использоваться в любом контексте, где требуется модель.

В табл. 3.1 (стр. 137) перечислены методы, которые должны быть переопределены в различных ситуациях. Например, во всех моделях нужно переопределять методы, необходимые для поддержки чтения (`flags()`, `data()` и т. д.), в редактируемых моделях – также методы поддержки редактирования и т. д. (Методы, которые следует переопределять для поддержки перетаскивания мышью, здесь не упоминаются; мы рассмотрим эту тему в следующей главе на стр. 180.)

Для некоторых моделей, допускающих изменение размера, эта операция имеет смысл только при определенных обстоятельствах. Например, табличная модель может разрешать вставку и удаление строк, но не столбцов. В таких случаях мы должны переопределить методы `insertRows()` и `removeRows()`, но не методы `insertColumns()` и `removeColumns()`.

Класс `TableModel`, рассматриваемый в этом разделе, – это подкласс `QAbstractTableModel`, хранящий свои данные в списке `QList<ZipcodeItem>`. Мы выбрали `QList`, а не `QVector`, потому что в общем случае он обеспечивает более высокую производительность, в особенности при вставках и удалениях из середины списка. (Отметим, что для очень больших списков, если приходится выполнять операции вставки и удаления в середине списка, лучше воспользоваться классом `QLinkedList`.)

Тип данных элементов, хранящихся в `QList`, должен допускать присваивание, то есть в нем должны быть определены конструктор по умолчанию, конструктор копирования и оператор присваивания. Кроме того, для использования некоторых методов могут понадобиться дополнительные операторы. Например, чтобы можно было обращаться к методам `QList::contains()`, `QList::count()` (при задании подсчитываемого значения в качестве аргумента), `QList::indexOf()`, `QList::lastIndexOf()`, `QList::removeAll()`, `QList::removeOne()` и `QList::operator!=()`, в классе, описывающем элемент, должен быть определен метод `operator==()`. А для поддержки сортировки с помощью функции `qSort()` в классе элемента должен быть метод `operator<()`. Ниже приведен код класса `ZipcodeItem`, который целиком содержится в заголовочном файле:

```
struct ZipcodeItem
{
    explicit ZipcodeItem(int zipcode_=InvalidZipcode,
                        const QString &postOffice_=QString(),
                        const QString &county_=QString(),
                        const QString &state_=QString())
        : zipcode(zipcode_), postOffice(postOffice_),
          county(county_), state(state_) {}
    bool operator<(const ZipcodeItem &other) const
        { return zipcode != other.zipcode ? zipcode < other.zipcode
          : postOffice < other.postOffice; }

    int zipcode;
    QString postOffice;
    QString county;
    QString state;
};
```

В этом классе имеется конструктор по умолчанию – все аргументы могут быть опущены, поскольку для них заданы значения по умолчанию. Мы предоставили компилятору C++ сгенерировать конструктор копирования и оператор присваивания, поскольку все данные-члены – значения, а не указатели. Метод `operator==()` не реализован, поэтому мы не сможем ни искать объекты `ZipcodeItem` в списке `QList`, ни выполнять еще какие-либо операции, требующие наличия оператора сравнения на равенство. Однако метод `operator<()` реализован – путем сравнения почтовых индексов или названий почтовых отделений, если индексы совпадают, поэтому сортировать по индексу мы сможем.

В более развитом классе мы, наверное, реализовали бы методы чтения и установки свойств, предоставили бы методы `isValid()` и `operator==()`, но сейчас нам все это не нужно, поскольку ничто из перечисленного не будет использоваться в тех задачах, для решения которых предназначен класс `TableModel`.

В классе `TableModel` реализованы все методы для поддержки доступа на чтение (`flags()`, `data()`, `headerData()`, `rowCount()`, `columnCount()`), оба метода для поддержки редактирования (`setData()` и `setHeaderData()`), но только два метода изменения размера (`insertRows()` и `removeRows()`); следовательно, в модели `TableModel` количество столбцов фиксированно. Кроме того, имеются методы `load()` и `save()` и метод чтения (getter) `filename()`, обеспечивающие возможность сохранения и загрузки данных из файла.

В классе `TableModel` есть всего две закрытых переменных: `zipcodes` типа `QList<ZipcodeItem>` и `m_filename` типа `QString`. Конструктор просто вызывает конструктор базового класса `QAbstractTableModel`, передавая ему аргумент `parent`; его тело пусто.

Теперь перейдем к рассмотрению методов `TableModel`. Хотя данные, которыми манипулируют методы, специфичны для приложений `Zipcodes`, структуру кода можно обобщить на любой подкласс `QAbstractTableModel` или `QAbstractItemModel`. Это не означает, что из него можно бездумно копировать фрагменты в свой код, но образцом для реализации методов модели в пользовательских табличных моделях он служить может.

Методы `QAbstractItemModel` API для таблиц

Начнем с рассмотрения методов `QAbstractItemModel`, которые должны быть переопределены для создания пользовательской модели, допускающей редактирование элементов и изменение своего размера (в части строк, но не столбцов).

Таблица 3.1. API класса `QAbstractItemModel`

Метод	Описание
Все модели	
<code>data(index, role)</code>	Возвращает значение типа <code>QVariant</code> для заданного модельного индекса <code>index</code> и роли <code>role</code> (типа <code>Qt::ItemDataRole</code> ; стр. 139)
<code>flags(index)</code>	Возвращает результат применения операции поразрядного ИЛИ с одним или несколькими значениями перечисления <code>Qt::ItemFlag</code> (стр. 139), показывающий, является ли элемент в позиции с данным модельным индексом <code>index</code> активным, допускающим отметку флажком, редактирование, выделение и т. д.

Таблица 3.1 (продолжение)

Метод	Описание
headerData (sect, orient, role)	Возвращает значение типа QVariant, описывающее роль <i>role</i> заголовка секции <i>sect</i> и ориентации <i>orient</i>
rowCount(index)	Возвращает количество строк ниже родителя с заданным модельным индексом <i>index</i> , то есть количество детей этого родителя

Табличные и древовидные модели

columnCount(index)	Возвращает количество столбцов ниже родителя с заданным модельным индексом <i>index</i> , часто является постоянным для всей модели
--------------------	---

Все редактируемые модели

setData (index, value, role)	Задаёт значение <i>value</i> для указанной роли <i>role</i> в элементе с указанным модельным индексом <i>index</i> . В случае успеха возвращает true и генерирует сигнал dataChanged()
setHeaderData (sect, orient, value, role)	Задаёт значение <i>value</i> для указанной роли <i>role</i> в заголовке секции <i>sect</i> с ориентацией <i>orient</i> . В случае успеха возвращает true и генерирует сигнал headerDataChanged()

Все модели с изменяемым размером

insertRows (row, count, index)	Вставляет значение <i>count</i> для строк в позицию <i>row</i> ниже родителя с модельным индексом <i>index</i> и возвращает true в случае успеха; при переопределении обязательно вызывать beginInsertRows() и endInsertRows()
removeRows (row, count, index)	Удаляет значение <i>count</i> для строк, начиная с позиции <i>row</i> ниже родителя с модельным индексом <i>index</i> , и возвращает true в случае успеха; при переопределении обязательно вызывать beginRemoveRows() и endRemoveRows()

Табличные и древовидные модели с изменяемым размером

insertColumns(column, count, index)	Вставляет значение <i>count</i> для столбцов в позицию <i>column</i> ниже родителя с модельным индексом <i>index</i> и возвращает true в случае успеха; при переопределении обязательно вызывать beginInsertColumns() и endInsertColumns()
removeColumns (column, count, index)	Удаляет значение <i>count</i> для столбцов, начиная с позиции <i>column</i> ниже родителя с модельным индексом <i>index</i> , и возвращает true в случае успеха; при переопределении обязательно вызывать beginRemoveColumns() и endRemoveColumns()

Метод	Описание
Древовидные модели	
<code>index</code> (<i>row</i> , <i>column</i> , <i>index</i>)	Возвращает объект <code>QModelIndex</code> для элемента в позиции с заданной строкой <i>row</i> , столбцом <i>column</i> и модельным индексом родителя <i>index</i>
<code>parent(index)</code>	Возвращает объект <code>QModelIndex</code> для родителя элемента с модельным индексом <i>index</i>

Таблица 3.2. Перечисление `Qt::ItemDataRole`

Флаг	Описание
<code>Qt::AccessibleDescriptionRole</code>	Описание элемента для поддержки специальных возможностей
<code>Qt::AccessibleTextRole</code>	Текст, используемый инструментами поддержки специальных возможностей, например программой чтения с экрана
<code>Qt::BackgroundRole</code>	Кисть для закрашивания фона при отображении данных
<code>Qt::CheckStateRole</code>	Состояние отмеченности элемента флажком
<code>Qt::DecorationRole</code>	Пиктограмма, ассоциированная с данными
<code>Qt::DisplayRole</code>	Текст, отображаемый для представления данных
<code>Qt::EditRole</code>	Текст в форме, пригодной для редактирования
<code>Qt::FontRole</code>	Шрифт для отображения данных в текстовом виде
<code>Qt::ForegroundRole</code>	Кисть для рисования текста при отображении данных
<code>Qt::SizeHintRole</code>	Предпочтительный размер
<code>Qt::StatusTipRole</code>	Текст информации о состоянии
<code>Qt::TextAlignmentRole</code>	Выравнивание текста при представлении данных в текстовом виде
<code>Qt::ToolTipRole</code>	Текст всплывающей подсказки
<code>Qt::UserRole</code>	Роль, которую можно использовать для хранения дополнительных пользовательских данных; данные можно хранить также в роли <code>Qt::UserRole + 1</code> и т. д.
<code>Qt::WhatsThisRole</code>	Текст, отображаемый для данного элемента в режиме «Что это?»

Таблица 3.3. Перечисление `Qt::ItemFlag`

Флаг	Описание
<code>Qt::ItemIsDragEnabled</code>	Элемент можно перетаскивать
<code>Qt::ItemIsDropEnabled</code>	На элемент можно перетаскивать
<code>Qt::ItemIsEditable</code>	Элемент можно редактировать
<code>Qt::ItemIsEnabled</code>	Пользователь может взаимодействовать с элементом
<code>Qt::ItemIsSelectable</code>	Элемент можно выделять
<code>Qt::ItemIsTristate</code>	У элемента имеется не два, а три возможных состояния (отмечен, не отмечен, не изменялся)
<code>Qt::ItemIsUserCheckable</code>	У элемента есть флажок, которым может манипулировать пользователь
<code>Qt::NoItemFlags</code>	Если установлен только этот флаг, то элемент нельзя отмечать, выделять, редактировать и т. д.

```

Qt::ItemFlags TableModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags theFlags = QAbstractTableModel::flags(index);
    if (index.isValid())
        theFlags |= Qt::ItemIsSelectable|Qt::ItemIsEditable|
                    Qt::ItemIsEnabled;
    return theFlags;
}

```

Если переданный модельный индекс допустим, то в элементе устанавливаются флаги, разрешающие выделение и редактирование и, разумеется, взаимодействие с ним. На самом деле, реализация в базовом классе уже устанавливает флаги `Qt::ItemIsSelectable|Qt::ItemIsEnabled`, так что достаточно было бы добавить к ним только `Qt::ItemIsEditable`; но мы предпочитаем выражать свои намерения явно. (Все имеющиеся флаги перечислены в табл. 3.3.)

Метод `data()` позволяет получить доступ ко всем данным и метаданным, ассоциированным с элементом. Хотя он не такой уж длинный, мы все же разобьем его на четыре части, чтобы было проще объяснять.

```

const int MaxColumns = 4;

QVariant TableModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid() ||
        index.row() < 0 || index.row() >= zipcodes.count() ||
        index.column() < 0 || index.column() >= MaxColumns)
        return QVariant();
    const ZipcodeItem &item = zipcodes.at(index.row());

```

Самая необычная особенность этого метода (а также метода `headerData()`) заключается в том, что мы *не* вызываем метод базового класса для обработки случаев, которые не обрабатываем сами. Вместо этого мы *обязаны* вернуть объект `QVariant`, обозначающий недопустимые данные, в тех случаях, которые не захотели или не смогли обработать. Ничего другого, кроме «недопустимого `QVariant`», возвращать в такой ситуации нельзя!

Архитектура модель/представление в Qt предполагает, что методы `QAbstractItemModel::data()` и `QAbstractItemModel::headerData()` возвращают либо допустимый `QVariant`, который далее можно использовать, либо недопустимый `QVariant`, и в таком случае Qt вычисляет подходящее значение самостоятельно. Если вернуть допустимый `QVariant`, представляющий непригодное для нас значение, например пустую строку или 0 для обозначения необработанного случая, то Qt воспользуется этим значением (поскольку это допустимый `QVariant`), что приведет к сбою в работе.

Сначала мы проверяем, что модельный индекс корректен и что хранящиеся в нем строка и столбец принадлежат допустимому диапазону. Если все проверки успешно прошли, то мы получаем ссылку на элемент в списке `zipcodes`, с которым можем работать дальше.

Мы решили рассмотреть два случая: запрос предпочтительного размера (роль `Qt::SizeHintRole`) и запрос данных элемента (роль `Qt::DisplayRole` или `Qt::EditRole`, которые в данном случае синонимы).

```
if (role == Qt::SizeHintRole) {
    QStyleOptionComboBox option;
    switch (index.column()) {
        case Zipcode: {
            option.currentText = QString::number(MaxZipcode);
            const QString header = headerData(Zipcode,
                Qt::Horizontal, Qt::DisplayRole).toString();
            if (header.length() > option.currentText.length())
                option.currentText = header;
            break;
        }
        case PostOffice: option.currentText = item.postOffice;
            break;
        case County: option.currentText = item.county; break;
        case State: option.currentText = item.state; break;
        default: Q_ASSERT(false);
    }
    QFontMetrics fontMetrics(data(index, Qt::FontRole).value<QFont>());
    option.fontMetrics = fontMetrics;
    QSize size(fontMetrics.width(option.currentText),
        fontMetrics.height());
    return qApp->style()->sizeFromContents(QStyle::CT_ComboBox,
        &option, size);
}
```

Для каждого столбца получаем соответствующий ему текст, за исключением столбца с почтовым индексом, для которого в качестве ширины используется длина максимально возможного числа или длина текста заголовка в зависимости от того, что больше. Затем создается объект, представляющий метрики шрифта, с помощью которого вычисляется размер, необходимый для представления текста. Обратите внимание на рекурсивное обращение к методу `data()` для получения шрифта. Мы не обрабатываем роль `Qt::FontRole` сами – Qt сделает это за нас, так как для всех необработанных случаев мы возвращаем недопустимый `QVariant`.

Нам понадобится дополнительное пространство, потому что, когда пользователь захочет отредактировать почтовый индекс на месте, он получит виджет-счетчик, которому необходимо место для кнопок прокрутки. Аналогично, для изменения штата используется комбинированный список, которому тоже необходимо место для кнопки раскрытия. Если это место не зарезервировать, то при редактировании данных элемента с помощью счетчика или комбинированного списка часть текста может оказаться невидимой. Для названия почтового отделения и округа дополнительное место оставлять не надо, так как они редактируются с помощью виджета `QLineEdit`, которому нужно лишь чуть-чуть места для рамки, тем не менее, если мы зарезервируем немного лишнего места, это не повредит, а код станет короче, потому что для всех столбцов производятся одни и те же вычисления.

Чтобы узнать фактически необходимый размер, мы берем объект `QStyleOptionComboBox`, устанавливаем в качестве `fontMetrics` метрики шрифта элемента, а в качестве `currentText` – текст элемента. Затем передаем этот объект вместе с размером, необходимым для размещения самого текста, методу `QStyle::sizeFromContents()` (получив указатель на объект `QStyle` приложения от глобального объекта `qApp` типа `QApplication`). Мы сообщаем этому методу, что хотим вычислить размер области, достаточной для отображения переданного текста в комбинированном списке (на что указывает первый аргумент `CT_ComboBox`), и возвращаем полученный от него размер. (Аналогичной константы для виджета-счетчика не предусмотрено, но поскольку комбинированный список в любом случае занимает больше места, чем счетчик, то можно использовать вышеупомянутую константу в обоих случаях.)

```
if (role == Qt::DisplayRole || role == Qt::EditRole) {
    switch (index.column()) {
        case Zipcode: return item.zipcode;
        case PostOffice: return item.postOffice;
        case County: return item.county;
        case State: return item.state;
        default: Q_ASSERT(false);
    }
}
```

Данные в режимах отображения и редактирования обрабатываются одинаково (довольно распространенная, но ни в коем случае не обяза-

тельная практика¹). Хотя типы данных, хранящихся в отдельных полях объекта `ZipcodeItem`, различаются (к примеру, `int` и `QString`), возвращаем мы всегда объект типа `QVariant`.

```
    return QVariant();
}
```

Для всех необработанных ситуаций (в данном случае ролей, которые мы не стали обрабатывать) возвращается недопустимый `QVariant`, чтобы `Qt` могла обработать их самостоятельно.

```
QVariant TableModel::headerData(int section,
                                Qt::Orientation orientation, int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();
    if (orientation == Qt::Horizontal) {
        switch (section) {
            case Zipcode: return tr("Zipcode");
            case PostOffice: return tr("Post Office");
            case County: return tr("County");
            case State: return tr("State");
            default: Q_ASSERT(false);
        }
    }
    return section + 1;
}
```

У табличных представлений обычно имеются два заголовка – горизонтальный и вертикальный, – поэтому в табличных моделях предусмотрены тексты для обоих. Нас интересует только роль отображения, поэтому запросы на данные во всех остальных ролях мы переадресуем `Qt`, возвращая недопустимый `QVariant`.

Секция является строкой, если задана ориентация `Qt::Vertical`, и столбцом, если задана ориентация `Qt::Horizontal`. Для заголовков столбцов мы возвращаем соответствующий текст, а для заголовков строк – номер секции плюс 1, чтобы номера начинались с единицы.

```
int TableModel::rowCount(const QModelIndex &index) const
{
    return index.isValid() ? 0 : zipcodes.count();
}
```

Этот метод возвращает количество строк (фактически число потомков) для указанного модельного индекса. В табличных (и списковых) моделях у всех элементов индекс родителя принимает недопустимое значение, поэтому если индекс содержит именно такое значение, то мы воз-

¹ Например, в электронной таблице данными в режиме редактирования может быть формула, а в режиме отображения – текущий результат ее вычисления. – *Прим. науч. ред.*

возвращаем общее количество строк в таблице (или списке). Если бы количество строк было фиксировано (поскольку не реализованы методы `insertRows()` и `removeRows()`), то в качестве его значения можно было бы возвращать константу.

Если индекс допустим, значит, у нас запрашивают количество строк (то есть число потомков) для элемента – действие, имеющее смысл только для древовидных моделей, – поэтому для списковых и табличных моделей мы в ответ должны вернуть 0.

```
int TableModel::columnCount(const QModelIndex &index) const
{
    return index.isValid() ? 0 : MaxColumns;
}
```

Этот метод возвращает количество столбцов для указанного модельного индекса. Если значение индекса допустимо, то мы возвращаем общее количество столбцов в таблице. (Для списковых моделей мы унаследовали бы классу `QAbstractListModel` и не стали бы реализовывать этот метод, потому что реализации из базового класса достаточно.) В данном примере количество столбцов фиксировано (так как мы не реализовали методы `insertColumns()` и `removeColumns()`), но если бы соответствующие методы изменения размера были реализованы, то количество столбцов могло бы изменяться.

Если индекс допустим, значит, у нас запрашивают количество столбцов для элемента (действие, которое редко имеет смысл), поэтому в нашем случае следует вернуть 0.

```
bool TableModel::setData(const QModelIndex &index,
                        const QVariant &value, int role)
{
    if (!index.isValid() || role != Qt::EditRole ||
        index.row() < 0 || index.row() >= zipcodes.count() ||
        index.column() < 0 || index.column() >= MaxColumns)
        return false;
    ZipcodeItem &item = zipcodes[index.row()];
    switch (index.column()) {
        case Zipcode: {
            bool ok;
            int zipcode = value.toInt(&ok);
            if (!ok || zipcode < MinZipcode ||
                zipcode > MaxZipcode)
                return false;
            item.zipcode = zipcode;
            break;
        }
        case PostOffice: item.postOffice = value.toString(); break;
        case County: item.county = value.toString(); break;
        case State: item.state = value.toString(); break;
        default: Q_ASSERT(false);
    }
}
```

```

        emit dataChanged(index, index);
        return true;
    }

```

Этот метод начинается так же, как `data()`, но для индикации успешного или неудачного завершения редактирования возвращает значение типа `bool`. Если модельный индекс прошел все проверки, то мы получаем неконстантную (то есть изменяемую) ссылку на соответствующий объект `ZipcodeItem` и записываем в указанный столбец переданные данные.

Мы не стали реализовывать контроль данных в столбцах, содержащих строковые данные, хотя было бы несложно отбрасывать пустые строки — надо лишь добавить еще одно условие в предложение `if` в начале метода, например `|| (index.column() != Zipcode && value.toString.isEmpty())`. Для почтовых индексов мы принимаем только допустимые значения.

Если не произошло никаких изменений, мы *обязаны* вернуть `false`. Напротив, если изменение завершилось успешно, мы *обязаны* сгенерировать сигнал `dataChanged()`, указав диапазон индексов изменившихся ячеек, и вернуть `true`.

Может потребоваться, чтобы одно редактирование оказывало каскадный эффект, и архитектура модель/представление в какой-то мере поддерживает эту возможность, позволяя указать в сигнале `dataChanged()` индексы левого верхнего и правого нижнего угла прямоугольной области, содержащей изменившиеся данные. Но обычно изменяются данные только в одной ячейке, поэтому в обоих аргументах мы передаем один и тот же индекс, как в коде выше.

```

bool setHeaderData(int, Qt::Orientation, const QVariant&,
                  int=Qt::EditRole) { return false; }

```

Этот метод мы реализовали прямо в заголовочном файле. Мы не хотим, чтобы пользователь мог редактировать заголовки строк и столбцов. Чтобы воспрепятствовать этому, достаточно возвращать `false` при любых входных аргументах.

Если мы разрешаем редактировать заголовки, то *обязаны* генерировать сигнал `headerDataChanged()`, передавая в нем ориентацию и индексы первой и последней изменившихся секций (строки или столбца), и вернуть `true`.

```

bool TableModel::insertRows(int row, int count, const QModelIndex&)
{
    beginInsertRows(QModelIndex(), row, row + count - 1);
    for (int i = 0; i < count; ++i)
        zipcodes.insert(row, ZipcodeItem());
    endInsertRows();
    return true;
}

```

Чтобы размер модели можно было изменять, необходимо реализовать методы `insertRows()` и `removeRows()` (или `insertColumns()` и `removeColumns()`),

или все четыре). Если аргумент `row` равен 0, то новые строки вставляются в начало, а если `row == rowCount()`, то в конец.

Структурно переопределенный метод `insertRows()` всегда устроен одинаково: *перед* внесением изменений в модель вызывается `beginInsertRows()`, затем идет код вставки, а *после* изменения модели – вызов `endInsertRows()`. Если какие-то изменения были произведены, то метод должен возвращать `true`.

Показанные здесь обращения к `beginInsertRows()` и `endInsertRows()` можно в точно таком же виде повторить в любом подклассе списковой или табличной модели. Древовидные модели несколько сложнее; о том, как обстоит дело с ними, мы поговорим в следующей главе.

В приложениях `Zipcodes` мы добавляем один пустой элемент `ZipcodeItem` для каждой вставляемой строки. На самом деле приложение `zipcodes2` не вызывает этот метод напрямую, однако оно обращается к `insertRow()` из `addZipcode()`, а уже этот метод (реализованный в базовом классе) вызывает полиморфный вариант `insertRows()`, передавая ему строку и счетчик 1.

```
bool TableModel::removeRows(int row, int count, const QModelIndex&)
{
    beginRemoveRows(QModelIndex(), row, row + count - 1);
    for (int i = 0; i < count; ++i)
        zipcodes.removeAt(row);
    endRemoveRows();
    return true;
}
```

Этот метод аналогичен `insertRows()` и устроен точно так же, только в начале и в конце вызываются соответственно `beginRemoveRows()` и `endRemoveRows()`. Точно такие же обращения можно делать в любой списковой или табличной модели.

В данном случае мы воспользовались методом `QList::removeAt()`, который удаляет и не учитывает значение в указанной строке; этот метод *требует*, чтобы строка принадлежала допустимому диапазону. Обратите внимание на одну тонкость: мы всегда удаляем «одну и ту же» строку – после удаления строки все следующие за ней сдвигаются на одну позицию, так что последующий вызов `removeAt()` удалит следующую строку.

Как и раньше, в приложении `zipcodes2` методы `insertRows()`, `removeRows()` не вызываются напрямую; вместо этого метод `deleteZipcode()` обращается к `removeRow()`, а реализация в базовом классе вызывает `removeRows()`, передавая указанную строку и счетчик 1.

Мы закончили рассмотрение всех переопределенных методов, необходимых для реализации редактируемой таблицы (или списка) с изменяемым размером (в смысле добавления и удаления строк).

Методы для сохранения и загрузки элементов таблицы

В этом разделе мы рассмотрим методы `save()` и `load()`, которые служат для сохранения и загрузки элементов таблицы из файла. В обоих методах подразумевается такой же формат файла почтовых индексов, как и в предыдущем разделе, с тем же магическим числом, тем же номером версии формата и той же версией `QDataStream` (стр. 122).

```
void TableModel::save(const QString &filename)
{
    if (!filename.isEmpty())
        m_filename = filename;
    if (m_filename.isEmpty())
        throw AQP::Error(tr("no filename specified"));
    QFile file(m_filename);
    if (!file.open(QIODevice::WriteOnly))
        throw AQP::Error(file.errorString());
    QDataStream out(&file);
    out << MagicNumber << FormatNumber;
    out.setVersion(QDataStream::Qt_4_5);
    QListIterator<ZipcodeItem> i(zipcodes);
    while (i.hasNext())
        out << i.next();
}
```

Этот метод начинается так же, как рассмотренный ранее метод `StandardTableModel::save()`, — используется либо переданное имя файла, либо имя, хранящееся в закрытой переменной `m_filename`, если новое имя не указано. Как и раньше, выводятся магическое число и номер версии формата, а затем устанавливается версия сериализатора. Далее с помощью перегруженного метода `QDataStream::operator<<()` выводятся все элементы данных.

```
QDataStream &operator<<(QDataStream &out, const ZipcodeItem &item)
{
    out << static_cast<quint16>(item.zipcode) << item.postOffice
        << item.county << item.state;
    return out;
}
```

Как обычно, при записи целого в поток `QDataStream` необходимо точно указать знаковую и разрядность.

```
void TableModel::load(const QString &filename)
{
    ...
    QDataStream in(&file);
    quint32 magicNumber;
    in >> magicNumber;
    if (magicNumber != MagicNumber)
        throw AQP::Error(tr("unrecognized file type"));
    quint16 formatVersionNumber;
```

```

in >> formatVersionNumber;
if (formatVersionNumber > FormatNumber)
    throw AQP::Error(tr("file format version is too new"));
in.setVersion(QDataStream::Qt_4_5);
zipcodes.clear();

ZipcodeItem item;
while (!in.atEnd()) {
    in >> item;
    zipcodes << item;
}
qSort(zipcodes);
reset();
}

```

Этот метод аналогичен рассмотренному ранее методу `StandardTableModel::load()`. Мы опустили обработку имени файла, поскольку она мало чем отличается от имеющейся в методе `save()`. В основном коде мы считываем и проверяем магическое число и номер версии формата, затем устанавливаем версию сериализатора – все точно так же, как в `StandardTableModel::load()`.

Подготовившись к чтению из потока, мы очищаем прежние данные, читаем все элементы `ZipcodeItem` – с помощью перегруженного метода `QDataStream::operator>>()` – и по одному добавляем их в список `zipcodes`. В конце список сортируется по почтовому индексу и вызывается метод `reset()`, который уведомляет все ассоциированные представления о том, что модель существенно изменилась.

```

QDataStream &operator>>(QDataStream &in, ZipcodeItem &item)
{
    quint16 zipcode;
    in >> zipcode >> item.postOffice >> item.county >> item.state;
    item.zipcode = static_cast<int>(zipcode);
    return in;
}

```

Считывание объектов `ZipcodeItem` с помощью этого оператора, а не по отдельности с последующей передачей конструктору, способствует четкому разделению обязанностей. По большей части данные можно читать непосредственно в `ZipcodeItem&`, но в случае целых чисел мы должны сначала прочитать значение в переменную правильного размера и знаковости.

На этом мы полностью завершили рассмотрение табличных моделей. Мы научились создавать подкласс `QStandardItemModel` для хранения табличных данных и пользовательскую модель в виде подкласса `QAbstractTableModel`, который предоставляет тот же API `QAbstractItemModel`, что и любая другая модель. В следующей главе мы обратимся к древовидным моделям, а затем поговорим о делегатах и представлениях.

4

Архитектура модель/представление: древовидные модели

- Деревья на основе модели `QStandardItemModel`
- Создание пользовательских древовидных моделей

В этой главе рассматриваются древовидные модели в архитектуре модель/представление. Предполагается знакомство с основами архитектуры модель/представление в Qt в объеме предыдущей главы (стр. 106).

В этой главе будут рассмотрены три модели. В первом разделе мы поговорим об использовании подкласса `QStandardItemModel` с хранением элементов в подклассе `QStandardItem`. (В предыдущей главе мы применяли для этих целей класс `QStandardItem` как таковой.) А во втором заменим класс `QStandardItemModel` пользовательской моделью. Как и для табличной модели из предыдущей главы, мы покажем способы добавления и редактирования элементов на месте и удаления их. Для пользовательской модели мы также реализуем перетаскивание мышью, вырезание и вставку элементов, перемещение вниз и вверх по дереву, увеличение и уменьшение уровня вложенности – во всех случаях элементы будут перемещаться вместе со своими потомками.

Древовидные модели формулируются в терминах родителей и потомков, то есть строка каждого элемента находится в некоторой позиции списка потомков его родителя. (Теоретически древовидную модель можно рассматривать как рекурсивное дерево таблиц, но ни одно из представлений Qt не поддерживает такую идею.)

У многих деревьев либо фиксирована структура, либо их элементы разнотипны – тогда перемещение элементов по дереву обычно не имеет смысла. Но если все элементы дерева (и их потомки) однотипны и перемещение любого элемента (вместе с потомками) в любое другое место дерева оправдано, то мы хотели бы дать пользователю возможность выполнять эту операцию. Эту задачу можно без особого труда решить с помощью специализированных подклассов класса `QAbstractItemModel`. Но мы не стали реализовывать перемещение элементов в подклассах `QStandardItemModel`, потому что, хотя метод `QStandardItemModel::insertRow()` и позволяет вставлять строки в любое место иерархии дерева, метод `QStandardItemModel::takeRow()` работает только со строками верхнего уровня, так как не принимает аргумент `parent` типа `QModelIndex`, описывающий родителя. Это означает, что при необходимости переместить строки в дереве, представленном подклассом `QStandardItemModel`, нам пришлось бы заняться утомительным копированием.

В первом разделе этой главы мы увидим, как создать подкласс `QStandardItemModel`, который умеет загружать и сохранять данные. Каждый элемент будет храниться в экземпляре пользовательского подкласса `QStandardItem`. Во втором разделе мы заменим подкласс `QStandardItemModel` пользовательским подклассом `QAbstractItemModel` и элементы также будем хранить в объектах специализированного класса. Пример из второго раздела предоставляет также важную дополнительную функциональность: возможность перемещать потомков в другое место дерева, в том числе путем перетаскивания мышью.

Приложения `Timelog – timelog1` и `timelog2` – загружают и сохраняют *xml*-файлы с данными о «задачах». У каждой задачи имеется имя, состояние «выполнена» и одна или несколько пар объектов типа дата/время, представляющих начальный и конечный моменты. С именами задач могут быть связаны простые атрибуты стиля шрифта, применяемые при отображении: полужирный, курсив и цвет. Глубина вложенности задач произвольна, а полное время выполнения задачи складывается из времени, заданной для нее самой, а также всех ее потомков любого уровня. В каждый момент времени активной (то есть хронометрируемой) может быть только одна задача.

Следует отметить, что индивидуальные пары начало–окончание *не* представлены в пользовательском интерфейсе сами по себе. Каждая задача представлена своим именем, состоянием выполнения и двумя агрегатами объектов дата/время: один показывает полное время работы над задачей за текущий день, другое – общее полное время. Это означает, что под каждую задачу в дереве отведена одна строка.

По аналогии с табличными моделями мы покажем, как в разных реализациях дерева выполняются одни и те же операции: загрузка и сохранение элементов, удаление элементов (и рекурсивно их потомков), добавление и редактирование элементов на месте. Кроме того, в прило-

жении `timelog2`, основанном на пользовательской модели, мы рассмотрим перемещение задач (вместе с потомками) и предоставим пользователю возможность делать это с помощью клавиатуры, меню, кнопок на панели инструментов и перетаскивания мышью. Заодно мы увидим, как можно скрывать и вновь показывать задачи, – в приложениях `Timelog` это будет делаться в зависимости от состояния выполнения задачи. Однако мы по возможности будем стараться уходить от обсуждения тех особенностей приложения, которые не связаны с архитектурой модель/представление, – в частности, опустим большую часть кода, относящегося к запуску и остановке хронометража задач и анимации его пиктограмм.

В обоих примерах для отображения и редактирования имен задач применяется специализированный делегат для форматированного текста (rich text). Эта тема рассматривается в главе 5 (стр. 216).

Деревья на основе модели QStandardItemModel

В приложении `timelog1` для загрузки, редактирования и сохранения данных используется подкласс класса `QStandardItemModel`, а для отображения и редактирования – класс `QTreeView` со специализированным делегатом. Пользовательский интерфейс традиционный: со строкой меню и панелью инструментов. Как обычно, основное внимание мы будем уделять вопросам моделей и представлений, опуская большую часть кода создания и компоновки виджетов, а также многие методы.

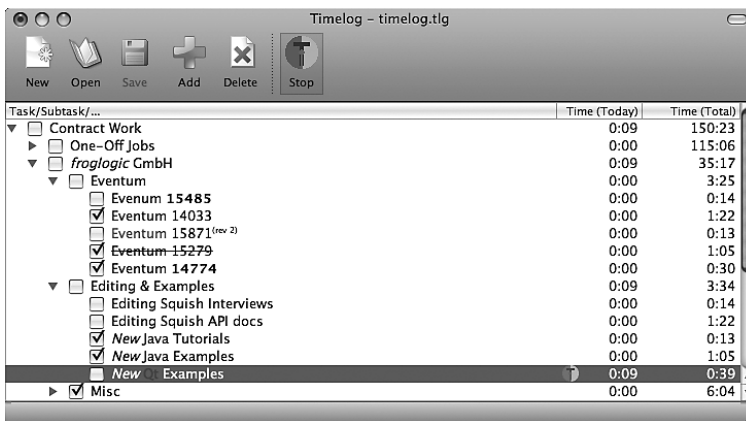


Рис. 4.1. Приложение `timelog1`

Внешний вид приложения показан на рис. 4.1. Видно, что в именах некоторых задач используются разные шрифты и цвета. В приложении `timelog1` позиция задачи в дереве фиксируется в момент добавления, а в следующем разделе при рассмотрении приложения `timelog2` мы по-

кажем, как перемещать задачи в произвольное место дерева (вместе с потомками).

Интерфейс `timelog1` поддерживает типичные для документо-ориентированных приложений функции: создание нового файла, открытие существующего файла и сохранение файла (в данном случае файл содержит дерево задач в формате XML), а также добавление новых и удаление существующих элементов. Кроме того, пользователь может запустить и остановить хронометраж задачи (при этом в список пар моментов начала и окончания данной задачи добавляется новый временной интервал), а также скрыть или показать «выполненные» задачи (отмеченные флажком).

Изменение древовидной модели с помощью пользовательского интерфейса

В этом подразделе мы рассмотрим скелет приложения и его пользовательского интерфейса, заложив основу для обсуждения связанных с моделями особенностей и фрагментов кода в последующих подразделах. Мы начнем с выдержки из определения класса главного окна, а затем рассмотрим его конструктор, в том числе код для создания древовидной модели и виджета `TreeView` наряду с наиболее существенными соединениями сигнал-слот. Из прочих методов мы рассмотрим только добавление и удаление задач, а также сокрытие (и показ) «выполненных» задач. (Методы для манипулирования структурой дерева, то есть для перемещения задач, реализованы только в версии с пользовательской моделью, а она будет обсуждаться в следующем разделе.)

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent=0);

public slots:
    void stopTiming();

protected:
    void closeEvent(QCloseEvent*);

private slots:
    void fileNew();
    void fileOpen();
    bool fileSave();
    bool fileSaveAs();
    void editAdd();
    void editDelete();
    void editHideOrShowDoneTasks(bool hide);
```

```

void setDirty(bool dirty=true) { setWindowModified(dirty); }
void load(const QString &filename,
          const QStringList &taskPath=QStringList());

private:
    ...
    QTreeView *treeView;
    StandardTreeModel *model;
};

```

Класс главного окна не таит никаких сюрпризов. Некоторые закрытые методы и данные-члены мы опустили, но вернемся к ним, когда это понадобится при рассмотрении того или иного метода. Мы не показали методов для работы с файлами, но упомянем метод load(), когда будем обсуждать следующие три фрагмента конструктора.

```

const QString FilenameSetting("Filename");
const QString GeometrySetting("Geometry");
const QString CurrentTaskPathSetting("CurrentTaskPath");
const int FirstFrame = 0;
const int LastFrame = 4;

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    createModelAndView();
    createActions();
    createMenusAndToolBar();
    createConnections();

    AQP::accelerateMenu(menuBar());
    setWindowTitle(tr("%1 (QStandardItemModel)[*]")
                  .arg(QApplication::applicationName()));
}

```

Начало конструктора вполне обычное: мы создаем модель и представление, затем действия, меню и панель инструментов, потом переходим к настройке соединений между сигналами и слотами. Описание создания действий, меню и панели инструментов мы опускаем, так как все это хорошо знакомо и не относится к программированию моделей и представлений.

```

timer.setInterval(333);
iconTimeline.setDuration(5000);
iconTimeline.setFrameRange(FirstFrame, LastFrame + 1);
iconTimeline.setLoopCount(0);
iconTimeline.setCurveShape(QTimeLine::LinearCurve);

```

Эта часть конструктора приведена здесь для того, чтобы был понятен контекст. В приложении используются два таймера: объект timer типа QTimer, который служит для актуализации отметок времени в элементах, и объект iconTimeline типа QTimeLine, необходимый для создания

анимированной пиктограммы в хронометрируемой задаче. Таймер `timer` обновляет отметку времени три раза в секунду, а `iconTimeline` в бесконечном (поскольку в методе `setLoopCount` задан нулевой счетчик цикла) цикле раз в пять секунд перебирает все пять кадров анимации (0–4). Форма кривой (*curve shape*) определяет интервал между кадрами: константа `QTimeline::LinearCurve` означает, что все интервалы одинаковы. (В версии Qt 4.6 появилась новая подсистема анимации, более высокого уровня и не ограничивающаяся одним лишь использованием таймеров или временных шкал `QTimeline`; мы рассмотрим ее в главе 13.)

```
QSettings settings;
restoreGeometry(settings.value(GeometrySetting).toByteArray());
QString filename = settings.value(FilenameSetting).toString();
if (filename.isEmpty())
    QTimer::singleShot(0, this, SLOT(fileNew()));
else
    QMetaObject::invokeMethod(this, "load", Qt::QueuedConnection,
                              Q_ARG(QString, filename), Q_ARG(QStringList,
                              settings.value(
                                  CurrentTaskPathSetting).toStringList()));
}
```

В самом конце конструктора мы пытаемся либо загрузить данные из того файла задач, который использовался при последнем запуске приложения, либо создать новый файл, подготовив его для добавления задач. Мы всегда начинаем загружать файл после того, как главное окно уже сконструировано, чтобы окно появилось на экране как можно быстрее, даже если загружаемый файл (или файлы) очень велик. Использовать однократный таймер для загрузки файла не получится, потому что при вызове слота необходимо задавать параметры.

Метод `QMetaObject::invokeMethod()` применяется, когда мы хотим вызвать слот с помощью очереди сообщений, то есть поместить вызов в очередь, так что он будет выполнен после всех предшествующих ему вызовов – в данном случае по завершении конструктора. Первым аргументом является объект, от имени которого вызывается метод, вторым – имя вызываемого слота, третьим – требуемый тип соединения, а остальные аргументы описывают параметры вызова. (На самом деле этот метод можно использовать и для немедленного вызова слота, если указать как требуемый тип соединения `Qt::DirectConnection`; эта возможность может пригодиться в случае, когда пользователь прямо или косвенно задает вызываемый метод и его аргументы, например, в диалоговом окне. Пример использования `Qt::DirectConnection` мы увидим в главе 12, стр. 457.)

Слоту `load()` передается два аргумента: имя файла задач и *путь к задаче*. Путь к задаче представляет собой список строк (имен задач), определяющий конкретную задачу. Например, на рис. 4.1 (стр. 151) путь к выделенной задаче таков: ["Contract Work", "<i>froglogic</i> GmbH", "Editing & Examples", "<i>New</i> Qt Examples"]. Имя

задачи может содержать небольшое подмножество HTML для задания эффектов шрифта; эта тема будет обсуждаться при рассмотрении делегата для форматированного текста в главе 5 (стр. 216). Вот фрагмент метода `load()` для показа пути к задаче:

```
model->load(filename);
if (!taskPath.isEmpty()) {
    if (QStandardItem *item = model->itemForPath(taskPath))
        setCurrentIndex(item->index());
}
```

Этот код помещен внутрь блока `try ... catch` на случай, если файл не удастся загрузить. Метод `load()` вызывается как из конструктора (для восстановления файла, который использовался при последнем запуске приложения), так и из метода `fileOpen()` (в этом случае путь к задаче – пустая строка). Метод `QStandardItem::index()` возвращает модельный индекс элемента. Пользовательский метод `StandardItemModel::itemForPath()` (и связанный с ним метод `StandardItemModel::pathForIndex()`) мы рассмотрим ниже (стр. 164).

```
void MainWindow::setCurrentIndex(const QModelIndex &index)
{
    if (index.isValid()) {
        treeView->scrollTo(index);
        treeView->setCurrentIndex(index);
    }
}
```

Этот коротенький вспомогательный метод мы создали для удобства, поскольку включенный в него код нужен в нескольких местах. Метод обеспечивает, чтобы элемент с заданным модельным индексом был виден и выделен, для чего при необходимости осуществляет прокрутку и раскрывает ветви дерева.

```
void MainWindow::createModelAndView()
{
    model = new StandardTreeModel(this);
    treeView->setAllColumnsShowFocus(true);
    treeView->setItemDelegateForColumn(0, new RichTextDelegate);
    treeView->setModel(model);
    setCentralWidget(treeView);
}
```

Здесь мы создаем экземпляр класса `StandardItemModel` (подкласса `QStandardItemModel`), а для представления данных используем стандартный виджет `QTreeView`.

В дереве имеется три столбца: имена задач, время, потраченное на задачу в текущий день, и общее время. Отображать столбцы с датой и временем мы поручили встроенному в `QTreeView` делегату, но для отображения столбца с именами задач должны написать собственный де-

легат, который будет интерпретировать HTML-разметку, а не выводить ее в виде обычного текста. Одно из достоинств специализированных делегатов для столбцов заключается в том, что все элементы данных в одном столбце обычно имеют одинаковый тип, поэтому у делегата, написанного для конкретного типа данных в столбце, больше шансов на повторное использование, чем у делегата, ориентированного на конкретную модель. (Делегаты рассматриваются в следующей главе.)

Хотя в тексте это и не показано, в реальном исходном коде имеются директивы `#ifdef`, обрамляющие код создания объекта `ModelTest` в случае, когда подключен модуль тестирования модели, — точно так же, как это было сделано в приложениях `Zipcodes` из предыдущей главы (стр. 131). Хотя подкласс модели `QStandardItemModel` добавляет только возможность загрузки и сохранения, не модифицируя встроенную функциональность, объект тестирования (версии 0.2) все равно сообщает об ошибке. Тест диагностирует две проблемы. Первая (строка 106 в файле `modeltest.cpp`) выглядит как безвредная ошибка в классе `QStandardItemModel` (флаги для недопустимого модельного индекса отличны от 0)¹. Вторая — это скорее признак чрезмерного рвения, чем реальная проблема (над строкой 341 в исходном коде имеется комментарий, говорящий о том, что строки, вызывающие сообщение об ошибке, можно спокойно закомментировать). В обоих случаях мы закомментировали строки, вызвавшие сообщения об ошибках, потому что это, похоже, ложные срабатывания.

```
void MainWindow::createConnections()
{
    connect(treeView->selectionModel(),
           SIGNAL(currentChanged(const QModelIndex&,
                                const QModelIndex&)), this, SLOT(updateUi()));
    connect(model, SIGNAL(itemChanged(QStandardItem*)),
           this, SLOT(setDirty()));
    connect(model, SIGNAL(rowsRemoved(const QModelIndex&,int,int)),
           this, SLOT(setDirty()));
    connect(model, SIGNAL(modelReset()), this, SLOT(setDirty()));
```

Мы показали только первые несколько соединений. Слот `updateUi()` (не показан) служит для актуализации пользовательского интерфейса; он активирует или деактивирует некоторые действия в зависимости от состояния приложения. Все сигналы, генерируемые моделью, косвенно устанавливают свойство `windowModified`, чтобы предупредить пользователя о наличии несохраненных изменений.

¹ Номера строк были действительны на момент написания книги — для версии, взятой из репозитория `Subversion`, которая не совпадает с распространяемой в виде дистрибутива, — но могут отличаться для того теста модели, который вы скачаете с сайта.

Большинство остальных соединений просто связывают сигнал, генерируемый действием, с соответствующим слотом – например, действие `fileNewAction` соединяется со слотом `fileNew()`, а действие `editAddAction()` – со слотом `editAdd()`. Имеются также два соединения, относящихся к таймерам: для хронометража задачи и для анимации связанной с ней пиктограммы.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    stopTiming();
    if (okToClearData()) {
        QSettings settings;
        settings.setValue(GeometrySetting, saveGeometry());
        settings.setValue(FilenameSetting, model->filename());
        settings.setValue(CurrentTaskPathSetting,
            model->pathForIndex(treeView->currentIndex()));
        event->accept();
    }
    else
        event->ignore();
}
```

Этот обработчик события вызывается, когда пользователь закрывает приложение. Метод `stopTiming()` (не показан) делает именно то, на что намекает его имя, – останавливает хронометраж. Для индикации наличия изменений в документе используется слот `setDirty()`, а устанавливаемый им признак состояния проверяется в методе `okToClearData()`. Если `okToClearData()` возвращает `true`, то мы сохраняем пользовательские настройки – геометрию главного окна, имя файла задач, путь к выделенной в данный момент задаче, – а затем подтверждаем событие закрытия, разрешая приложению завершиться. Для полноты приведем также код метода `okToClearData()` и вспомогательной функции `AQP::okToClearData()`, к которой он обращается.

```
bool MainWindow::okToClearData()
{
    if (isWindowModified())
        return AQP::okToClearData(&MainWindow::fileSave, this,
            tr("Unsaved changes"), tr("Save unsaved changes?"));
    return true;
}
```

Поскольку вывод диалогового окна с предложением сохранить изменения – распространенная задача, мы написали для этого вспомогательную функцию `AQP::okToClearData()`. Она принимает указатель на метод, который следует вызвать для сохранения; указатель на окно, поверх которого должно появиться диалоговое окно (в данном случае это главное окно) и которое, собственно, является объектом, требующим вызова метода сохранения; текст заголовка диалогового окна; текст сообщения

в окне и дополнительный текст (который мы в данном случае не передаем). Первый аргумент должен указывать на метод окна, переданного во втором аргументе, причем этот метод должен возвращать булево значение, равное `true`, если сохранение прошло успешно, и `false` в противном случае. Синтаксически передача указателя на метод записывается в виде `&ClassName::MethodName`.

```
template<typename T>
bool okToClearData(bool (T::*saveData>(), T *parent,
                  const QString &title, const QString &text,
                  const QString &detailedText=QString())
{
    QScopedPointer<QMessageBox> messageBox(new
        QMessageBox(parent));
    messageBox->setWindowModality(Qt::WindowModal);
    messageBox->setIcon(QMessageBox::Question);
    messageBox->setWindowTitle(QString("%1 - %2")
        .arg(QApplication::applicationName()).arg(title));
    messageBox->setText(text);
    if (!detailedText.isEmpty())
        messageBox->setInformativeText(detailedText);
    messageBox->addButton(QMessageBox::Save);
    messageBox->addButton(QMessageBox::Discard);
    messageBox->addButton(QMessageBox::Cancel);
    messageBox->setDefaultButton(QMessageBox::Save);
    messageBox->exec();
    if (messageBox->clickedButton() ==
        messageBox->button(QMessageBox::Cancel))
        return false;
    if (messageBox->clickedButton() ==
        messageBox->button(QMessageBox::Save))
        return (parent->*saveData());
    return true;
}
```

В сигнатуре функции параметр типа функции-члена без аргументов синтаксически записывается в виде `returnType (Type::method)()`, где *returnType* — тип возвращаемого методом значения (возможно, `void`), *Type* — класс, которому принадлежит метод (например, `MainWindow`), и *method* — имя, под которым этот метод фигурирует внутри тела функции, — оно может быть любым и не обязано совпадать с реальным именем метода. (Однако при вызове функции `okToClearData()` следует передавать имя вполне определенного класса и метода, например, `&MainWindow::fileSave`.)

Мы сделали `okToClearData()` шаблонной функцией, чтобы не записывать в ее код имя класса. Следовательно, ее можно использовать с любым классом, в котором есть метод, возвращающий `bool`, например `MyWindow::save()`. В теле функции этот метод называется `saveData()` вне зависимости от его истинного имени.

Конфигурирование объекта `QMessageBox` и применение класса `QScopedPointer` в версии Qt 4.6 объяснялось во врезках «Избегайте статических вспомогательных функций `QMessageBox`» (стр. 77) и «Интеллектуальные указатели в Qt» (стр. 78)¹.

Если пользователь хочет сохранить изменения, то мы возвращаем результат вызова переданного нам метода. Синтаксически вызов метода, переданного по указателю и не принимающего аргументов, записывается в виде `(object->*method)()`. В данном случае вызывается метод `MainWindow::fileSave()`. Если имя файла неизвестно и пользователь нажал в диалоговом окне сохранения кнопку отмены, то метод `fileSaveAs()` (вызванный из `fileSave()`) вернет `false`, и это же значение будет возвращено методом `fileSave()`. В таком случае `okToClearData()` тоже вернет `false`, и, значит, хотя изменения не были сохранены, они и не потерялись. Единственный способ избежать сохранения в неименованном (то есть новом) файле – явно нажать кнопку отказа от изменений в диалоговом окне.

В приложении `timelog1` позиция задачи в дереве фиксируется в момент ее создания (это ограничение не распространяется на `timelog2`, как мы убедимся в следующем разделе). Поэтому при добавлении новой задачи мы должны дать пользователю возможность указать, в какое место дерева она должна быть помещена. Этим занимается метод `editAdd()`, который мы рассмотрим в три приема.

```
void MainWindow::editAdd()
{
    QModelIndex index = treeView->currentIndex();
    StandardTreeModel::Insert insert = StandardTreeModel::AtTopLevel;
```

Получаем индекс текущего элемента (он может оказаться недопустимым, если текущего элемента нет, например, когда пользователь только что выполнил операцию `File → New`). Кроме того, вначале мы предполагаем, что новый элемент будет вставлен на верхнем уровне, задав соответствующую константу из перечисления `StandardTreeModel`; при отсутствии элементов в дереве это вообще единственная возможность.

```
if (index.isValid()) {
    QStandardItem *item = model->itemFromIndex(index);
    QScopedPointer<QMessageBox> messageBox(new QMessageBox(this));
    messageBox->setWindowModality(Qt::WindowModal);
    messageBox->setIcon(QMessageBox::Question);
    messageBox->setWindowTitle(tr("%1 - Add Task")
        .arg(QApplication::applicationName()));
    messageBox->setText(tr("<p>Add at the top level or as a "
        "sibling or child of\n'%1'").arg(item->text()));
    messageBox->addButton(tr("&Top Level"),
```

¹ В исходном коде имеется директива `#if QT_VERSION`, позволяющая компилировать его в версии Qt 4.5 с заменой `QScopedPointer` на `QSharedPointer`.

```

QMessageBox::AcceptRole);
QAbstractButton *siblingButton = messageBox->addButton(
    tr("&Sibling"), QMessageBox::AcceptRole);
QAbstractButton *childButton = messageBox->addButton(
    tr("C&hild"), QMessageBox::AcceptRole);
messageBox->setDefaultButton(
    qobject_cast<QPushButton*>(childButton));
messageBox->addButton(QMessageBox::Cancel);
messageBox->exec();
if (messageBox->clickedButton() ==
    messageBox->button(QMessageBox::Cancel))
    return;
if (messageBox->clickedButton() == childButton)
    insert = StandardTreeModel::AsChild;
else if (messageBox->clickedButton() == siblingButton)
    insert = StandardTreeModel::AsSibling;
}

```

Если текущий элемент существует, то пользователю предлагается на выбор три варианта: добавить новый элемент на верхний уровень, на один уровень с текущим или сделать его дочерним элементом текущего. Если пользователь не нажал кнопку отмены и не выбрал верхний уровень, то мы соответствующим образом изменяем переменную `insert`.

Обратите внимание, что в начале текста сообщения мы поставили HTML-тег `<p>`, обозначающий начало абзаца. В результате текст сообщения будет интерпретироваться как HTML и отображаться правильно. Это существенно, так как имя текущей задачи может содержать HTML-разметку.

```

if (QStandardItem *item = model->insertNewTask(insert,
    tr("New Task"), index)) {
    QModelIndex index = item->index();
    setCurrentIndex(index);
    treeView->edit(index);
    setDirty();
    updateUi();
}
}

```

Мы просим модель вставить новую задачу в указанную позицию, задавая ее имя по умолчанию и родителя (при вставке на верхний уровень родитель игнорируется). Если вставка завершилась успешно (как оно и должно быть), то мы запрашиваем модельный индекс нового элемента, делаем этот индекс текущим и иницилируем редактирование, чтобы пользователь мог заменить текст «New Task» осмысленным именем задачи.

```

void MainWindow::editDelete()
{
    QModelIndex index = treeView->currentIndex();

```

```

    if (!index.isValid())
        return;
    QStandardItem *item = model->itemFromIndex(index);
    if (item == timedItem)
        stopTiming();
    QString name = item->text();
    int rows = item->rowCount();
    QString message;
    if (rows == 0)
        message = tr("<p>Delete '%1'").arg(name);
    else if (rows == 1)
        message = tr("<p>Delete '%1' and its child (and "
            "grandchildren etc.)").arg(name);
    else if (rows > 1)
        message = tr("<p>Delete '%1' and its %2 children (and "
            "grandchildren etc.)").arg(name).arg(rows);
    if (!AQP::okToDelete(this, tr("Delete"), message))
        return;
    model->removeRow(index.row(), index.parent());
    setDirty();
    updateUi();
}

```

Если ни один элемент не выделен, этот метод возвращает управление, ничего не делая. В противном случае он запрашивает у модели элемент, который соответствует индексу выбранной строки, и возвращает его текст и количество потомков. Чтобы упростить жизнь переводчикам, мы используем три разных строки для трех возможных случаев (нет потомков, один потомок, много потомков).

Затем мы просим пользователя подтвердить удаление, и если он нажимает кнопку Delete, то просим модель выполнить удаление. (Функцию `AQP::okToDelete()` мы разбирали выше, стр. 119.) В «самом базовом» классе `QAbstractItemModel` метод `removeRow()` не делает ничего и возвращает `false`, но его реализация в непосредственном базовом классе `StandardItemModel` корректно удаляет переданную строку (и рекурсивно всех ее потомков) и возвращает `true`.

```

void MainWindow::editHideOrShowDoneTasks(bool hide)
{
    hideOrShowDoneTask(hide, model->invisibleRootItem());
}

void MainWindow::hideOrShowDoneTask(bool hide, QStandardItem *item)
{
    QModelIndex index = item->parent() ? item->parent()->index()
        : QModelIndex();
    bool hideThisOne = hide && (item->checkState() == Qt::Checked);
    treeView->setRowHidden(item->row(), index, hideThisOne);
    if (!hideThisOne) {
        for (int row = 0; row < item->rowCount(); ++row)

```



```

        hideOrShowDoneTask(hide, item->child(row, 0));
    }
}

```

Действие `editHideOrShowDoneTasksAction` соединено со слотом `editHideOrShowDoneTasks()`. Из этого слота производится начальное обращение к рекурсивному методу `hideOrShowDoneTask()`.

В методе `hideOrShowDoneTask()` мы сначала выясняем, нужно ли скрыть текущий элемент, а затем вызываем `QTreeView::setRowHidden()`, чтобы показать или скрыть строку текущего элемента в зависимости от ее текущего состояния. Если мы скрываем строку, то о ее потомках можно не беспокоиться, потому что они будут скрыты автоматически; но если строка не скрыта, то следует рекурсивно пробежаться по всем потомкам и в зависимости от состояния показать или скрыть их.

На этом рассмотрение пользовательского интерфейса приложения `Time-log` в части, касающейся древовидной модели, можно считать законченным. Теперь у нас достаточно контекста, чтобы разобраться в подклассе `QStandardItem`, который рассматривается в следующем разделе, и в подклассе `QStandardItemModel`, который описывается сразу вслед за ним.

Подкласс `QStandardItem` для представления элементов дерева

В приложении `timelog1` мы решили реализовать специализированный подкласс класса `QStandardItem`, так как хотим добавить пользовательские данные и методы, относящиеся к моментам начала и окончания работы над задачей. Вот полное определение этого класса:

```

class StandardItem : public QStandardItem
{
public:
    explicit StandardItem(const QString &text, bool done);
    QStandardItem *todayItem() const { return m_today; }
    QStandardItem *totalItem() const { return m_total; }
    void addDateTime(const QDateTime &start, const QDateTime &end)
        { m_dateTimes << qMakePair(start, end); }
    QList<QPair<QDateTime, QDateTime> > dateTimes() const
        { return m_dateTimes; }
    void incrementLastEndTime(int msec);
    QString todaysTime() const;
    QString totalTime() const;

private:
    int minutesForTask(bool onlyForToday) const;
    QStandardItem *m_today;
    QStandardItem *m_total;
    QList<QPair<QDateTime, QDateTime> > m_dateTimes;
};

```

Не вполне обычным является хранение указателей на два объекта `QStandardItem`, которые используются для отображения времени, затраченного на задачу в текущий день и в целом. Здесь мы пожертвовали памятью ради удобства. В списке `QList` хранятся все пары начало–окончание для одной задачи; элемент добавляется в этот список, когда пользователь запускает хронометраж для задачи.

Из всех методов мы рассмотрим только конструктор, остальные не-встроенные методы служат для увеличения времени, когда задача хронометрируется, а также для вычисления значений продолжительности и возврата их в виде строки или объекта `QDateTime`, поэтому к программированию моделей и представлений они отношения не имеют (хотя в исходном коде, прилагаемом к книге, они содержатся).

```
StandardItem::StandardItem(const QString &text, bool done)
    : QStandardItem(text)
{
    setCheckable(true);
    setCheckState(done ? Qt::Checked : Qt::Unchecked);
    setFlags(Qt::ItemIsSelectable|Qt::ItemIsEnabled|
             Qt::ItemIsEditable|Qt::ItemIsUserCheckable);
    m_today = new QStandardItem;
    m_today->setFlags(Qt::ItemIsSelectable|Qt::ItemIsEnabled);
    m_today->setTextAlignment(Qt::AlignVCenter|Qt::AlignRight);
    m_total = new QStandardItem;
    m_total->setFlags(Qt::ItemIsSelectable|Qt::ItemIsEnabled);
    m_total->setTextAlignment(Qt::AlignVCenter|Qt::AlignRight);
}
```

Элемент, представляющий задачу, сделан отмечаемым (чтобы отражать состояние флага «задача выполнена»), а также допускающим выделение, редактирование и отметку пользователем (чтобы пользователь мог поставить и сбросить флажок, щелкнув по нему мышью или нажав клавишу пробела, когда соответствующая задача выделена).

Подкласс `StandardItemModel` – единственное место в коде, где мы создаем объекты `StandardItem`, по одному для каждой задачи. А эти объекты, в свою очередь, создают экземпляры `QStandardItem` для хранения моментов начала и окончания задачи. Элементы, представляющие время, допускают только выделение – мы не хотим, чтобы пользователь мог их редактировать, и флажки им тоже не нужны.

Теперь, когда мы знаем, какие данные хранятся в классе `StandardItem` и какие методы он предоставляет, можно взглянуть на класс `StandardItemModel`, в котором хранятся все задачи.

Подкласс QStandardItemModel для представления дерева

StandardTreeModel — это подкласс класса QStandardItemModel, предназначенный для представления дерева задач. Помимо методов для работы с файлами — load() и save() — этот класс предоставляет метод insertNewTask() для добавления новой задачи; мы уже видели, как он вызывается из метода MainWindow::editAdd() (стр. 159). Кроме того, в нем есть методы pathForIndex() и itemForPath() для работы с путями к задачам; с ними мы тоже встречались (стр. 154 и 156) и подробно рассмотрим их в этом разделе. Единственный дополнительный закрытый член в этом классе — имя файла типа QString.

Данные приложения хранятся в *xml*-файле на диске. Фрагмент такого файла показан на рис. 4.2.

Имя и статус выполнения задачи хранятся в атрибутах тега TASK. Имена задач могут содержать HTML-разметку, которая должна быть надлежащим образом экранирована, чтобы не конфликтовать с XML-разметкой файла данных. Флаг выполнения представляется значением 0 (false) или 1 (true). Пары начало–окончание хранятся в виде строк в формате ISO 8601 в атрибутах START и END тегов WHEN, вложенных в тег TASK соответствующей задачи. Иерархическая организация задач осуществляется естественным образом: посредством вложенных тегов TASK с произвольной глубиной вложенности.

```
<TASK NAME="Editing & Examples" DONE="0">
  <WHEN START="2009-03-18T13:23:59" END="2009-03-18T13:25:15"/>
  <TASK NAME="Editing Squish Interviews" DONE="0">
    <WHEN START="2009-02-03T16:19:35" END="2009-02-03T16:34:02"/>
  </TASK>
  <TASK NAME="Editing Squish API docs" DONE="0">
    <WHEN START="2008-11-14T07:46:31" END="2008-11-14T08:25:55"/>
    <WHEN START="2008-11-14T09:55:26" END="2008-11-14T10:21:11"/>
    <WHEN START="2009-02-25T10:50:20" END="2009-02-25T10:51:29"/>
    <WHEN START="2009-02-25T10:52:02" END="2009-02-25T11:09:33"/>
  </TASK>
</TASK>
...
```

Рис. 4.2. Фрагмент файла данных для приложения *Timelog*

В этом разделе мы рассмотрим все методы класса StandardTreeModel, поскольку все они относятся к программированию моделей и представлений.

```
StandardTreeModel::StandardTreeModel(QObject *parent)
: QStandardItemModel(parent)
```

```

    {
        initialize();
    }

void StandardTreeModel::initialize()
{
    setHorizontalHeaderLabels(QStringList() << tr("Task/Subtask/...")
        << tr("Time (Today)") << tr("Time (Total)"));
    for (int column = 1; column < columnCount(); ++column)
        horizontalHeaderItem(column)->setTextAlignment(
            Qt::AlignVCenter|Qt::AlignRight);
}

```

Мы вынесли инициализацию в отдельный метод, поскольку он вызывается из двух мест.

```

void StandardTreeModel::clear()
{
    QStandardItemModel::clear();
    initialize();
}

```

Метод базового класса clear() уничтожает не только все элементы модели, но и их заголовки. Поэтому после очистки мы заново создаем заголовки, обращаясь к initialize().

```

void StandardTreeModel::save(const QString &filename)
{
    if (!filename.isEmpty())
        m_filename = filename;
    if (m_filename.isEmpty())
        throw AQP::Error(tr("no filename specified"));
    QFile file(m_filename);
    if (!file.open(QIODevice::WriteOnly|QIODevice::Text))
        throw AQP::Error(file.errorString());

    QDomStreamWriter writer(&file);
    writer.setAutoFormatting(true);
    writer.writeStartDocument();
    writer.writeStartElement("TIMELOG");
    writer.writeAttribute("VERSION", "2.0");
    writeTaskAndChildren(&writer, invisibleRootItem());
    writer.writeEndElement(); // TIMELOG
    writer.writeEndDocument();
}

```

Этот метод начинается так же, как встречавшийся в приложениях Zip-codes: если имя нового файла задано, используем его, в противном случае берем имя уже существующего файла. Ожидается, что исключение перехватит и обработает вызывающий метод. Класс AQP::Error мы уже рассматривали выше (стр. 124).

Класс QXmlStreamWriter умеет выводить всю нужную нам разметку, так что никакие подклассы не нужны. Если свойство autoFormatting равно true, то XML-разметка выводится в виде, удобном для восприятия человеком, – с отступами и разбиением на строки; в противном случае вывод будет максимально компактным, без лишних пробелов. Метод writeStartDocument() выводит строку `<?xml version="1.0" encoding="UTF-8"?>` в начале файла; разумеется, атрибут encoding (и сама кодировка) будут иными, если мы вызовем метод QXmlStreamWriter::setCodec(), указав нужную нам кодировку.

Вся иерархия тегов TASK помещается в тег TIMELOG (не показан), у которого имеется атрибут VERSION, позволяющий впоследствии без труда изменить формат файла. Метод QXmlStreamWriter::writeAttribute() принимает имя и значение атрибута – экранировать их необязательно, об этом позаботится сам класс QXmlStreamWriter. Затем вызывается метод writeTaskAndChildren(), который выводит данные обо всех задачах.

```
const QString TaskTag("TASK");
const QString NameAttribute("NAME");
const QString DoneAttribute("DONE");
const QString WhenTag("WHEN");
const QString StartAttribute("START");
const QString EndAttribute("END");

void StandardTreeModel::writeTaskAndChildren(QXmlStreamWriter *writer,
                                             QStandardItem *root)
{
    if (root != invisibleRootItem()) {
        StandardItem *item = static_cast<StandardItem*>(root);
        writer->writeStartElement(TaskTag);
        writer->writeAttribute(NameAttribute, item->text());
        writer->writeAttribute(DoneAttribute,
                               item->checkState() == Qt::Checked ? "1" : "0");
        QListIterator<
            QPair<QDateTime, QDateTime> > i(item->dateTimes());
        while (i.hasNext()) {
            const QPair<QDateTime, QDateTime> &dateTime = i.next();
            writer->writeStartElement(WhenTag);
            writer->writeAttribute(StartAttribute,
                                   dateTime.first.toString(Qt::ISODate));
            writer->writeAttribute(EndAttribute,
                                   dateTime.second.toString(Qt::ISODate));
            writer->writeEndElement(); // WHEN
        }
    }
    for (int row = 0; row < root->rowCount(); ++row)
        writeTaskAndChildren(writer, root->child(row, 0));
    if (root != invisibleRootItem())
        writer->writeEndElement(); // TASK
}
```

Обратите внимание, что имена тегов и атрибутов *не* заключены в функцию `tr()`, так как это элементы форматирования файла и для чтения человеком они, строго говоря, не предназначены.

Этот метод выводит одну задачу и рекурсивно всех ее потомков, пропуская, однако, невидимый корневой узел. Сначала выводится тег `TASK` с атрибутами `name` и `done`. Затем мы обходим все пары начало–окончание, выводя тег `WHEN` для каждой пары. А далее переходим к выводу дочерних задач (которые выведут *свои* дочерние задачи и т. д.). Иерархическая структура отражается корректно, потому что закрывающий тег `TASK` каждой задачи выводится *после* вывода потомков (любого уровня).

Ознакомившись с тем, как сохраняются задачи, посмотрим, как они загружаются. Для простоты разобьем код метода `load()` на две части:

```
void StandardTreeModel::load(const QString &filename)
{
    ...
    clear();
    QStack<QStandardItem*> stack;
    stack.push(invisibleRootItem());
    QDomStreamReader reader(&file);
    while (!reader.atEnd()) {
        reader.readNext();
        if (reader.isStartElement()) {
            if (reader.name() == TaskTag) {
                const QString name = reader.attributes()
                    .value(NameAttribute).toString();
                bool done = reader.attributes().value(DoneAttribute) == "1";
                StandardItem *nameItem = createNewTask(stack.top(),
                                                         name, done);

                stack.push(nameItem);
            }
            else if (reader.name() == WhenTag) {
                const QDateTime start = QDateTime::fromString(
                    reader.attributes().value(StartAttribute)
                        .toString(), Qt::ISODate);
                const QDateTime end = QDateTime::fromString(
                    reader.attributes().value(EndAttribute)
                        .toString(), Qt::ISODate);
                StandardItem *nameItem = static_cast<StandardItem*>(
                    stack.top());
                nameItem->addDateTime(start, end);
            }
        }
        else if (reader.isEndElement()) {
            if (reader.name() == TaskTag)
                stack.pop();
        }
    }
}
```

Метод `load()` начинается почти так же, как метод `save()`, – обработкой имени файла, так что этот код мы опускаем. Единственное отличие состоит в том, что режим открытия файла теперь равен `QIODevice::ReadOnly`, поэтому файл открывается в двоичном режиме только для чтения. Объект `QXmlStreamReader` читает тег `<?xml?>` и определяет из него, какую использовать кодировку; если кодировка не задана, по умолчанию подразумевается UTF-8.

Успешно открыв файл, мы стираем существующие элементы, готовясь заполнить метод данными, прочитанными из *xml*-файла.

Создавать подкласс класса `QXmlStreamReader` не нужно, поскольку предоставляемой им функциональности и так достаточно.

При создании новой задачи необходимо указать ее родителя – либо реального, либо *невидимый корневой узел*, предоставляемый базовым классом, если задача расположена на верхнем уровне дерева. Для хранения родителей мы используем стек `QStack<QStandardItem*>`, в который по мере необходимости помещаем задачи, а затем извлекаем их.

Встретив тег `TASK`, мы создаем представляющий задачу объект `StandardItem`, передавая его конструктору в качестве родителя задачу, взятую с вершины стека, а в качестве имени и флага выполнения – атрибуты, прочитанные из тега. Вновь созданная задача помещается на вершину стека.

Метод `QXmlStreamReader::attributes()` возвращает атрибуты текущего элемента в виде объекта `QXmlStreamAttributes`. Метод `QXmlStreamAttributes::value()` принимает имя атрибута и возвращает его значение в виде объекта `QStringRef`, убрав из текста все XML-экранирование. Например, имя задачи «Editing & Examples» в *xml*-файле хранится в виде «Editing & Examples», но метод `value()` возвращает его в исходном виде.

Хотя стандартные операторы сравнения позволяют сравнивать объекты `QString` и `QStringRef`, получая разумные результаты, но если нам необходим текст, хранящийся в объекте `QStringRef`, то необходимо вызвать его метод `toString()`. Например, мы заносим имя задачи в строку `QString` методом `QStringRef::toString()`, но для определения того, выполнена ли задача, достаточно сравнить значение атрибута «done» со строкой «1».

Встретив тег `WHEN`, мы считываем время начала и время окончания работы и добавляем их в конец списка моментов начала–окончания для задачи, находящейся на вершине стека. А встретив закрывающий тег `TASK`, мы выталкиваем верхний элемент из стека.

```
if (reader.hasError())
    throw AQP::Error(reader.errorString());
if (stack.count() != 1 || stack.top() != invisibleRootItem())
    throw AQP::Error(tr("loading error: possibly corrupt file"));
calculateTotalsFor(invisibleRootItem());
}
```

Если при разборе *xml*-файла возникнет ошибка, метод `QXmlStreamReader::atEnd()` вернет `true` (в результате чего произойдет выход из цикла `while`), а метод `QXmlStreamReader::hasError()` также вернет `true`, информируя нас о неполадках. Если в конце цикла в стеке окажется не в точности один элемент (невидимый корень дерева), значит, что-то не в порядке. При обнаружении любой ошибки мы возбуждаем исключение, считая, что с ним разберется вызывающий метод.

Если файл загрузился успешно, мы вызываем метод `calculateTotalsFor()`, который позаботится о правильном отображении времени для всех задач.

```
StandardItem *StandardItemModel::createNewTask(QStandardItem *root,
        const QString &name, bool done)
{
    StandardItem *nameItem = new StandardItem(name, done);
    root->appendRow(QList<QStandardItem*>() << nameItem
        << nameItem->todayItem() << nameItem->totalItem());
    return nameItem;
}
```

Выше мы видели, что конструктор объекта `StandardItem` создает два объекта `QStandardItem` для отображения времени, затраченного на задачу в текущий день и в целом (стр. 162). Создав задачу, мы добавляем представляющий ее объект `StandardItem` вместе с ассоциированными объектами `QStandardItem` в виде новой строки, являющейся потомком переданного элемента `root` (которым будет невидимый корневой узел, если это задача верхнего уровня). Начиная с этого момента владение всеми тремя элементами принимает на себя модель, как это обычно и происходит в Qt, поэтому мы можем не заботиться об их удалении.

```
void StandardTreeModel::calculateTotalsFor(QStandardItem *root)
{
    if (root != invisibleRootItem()) {
        StandardItem *item = static_cast<StandardItem*>(root);
        item->todayItem()->setText(item->todayTime());
        item->totalItem()->setText(item->totalTime());
    }
    for (int row = 0; row < root->rowCount(); ++row)
        calculateTotalsFor(root->child(row, 0));
}
```

Этот рекурсивный метод служит для задания текста во всех элементах, представляющих время. Мы не показываем методы `todayTime()` и `totalTime()`, так как они не касаются программирования моделей и представлений, но в исходном коде они, конечно, присутствуют.

```
enum Insert {AtTopLevel, AsSibling, AsChild};

QStandardItem *StandardItemModel::insertNewTask(Insert insert,
        const QString &name, const QModelIndex &index)
```



```

{
    QStandardItem *parent;
    if (insert == AtTopLevel)
        parent = invisibleRootItem();
    else {
        if (index.isValid()) {
            parent = itemFromIndex(index);
            if (!parent)
                return 0;
            if (insert == AsSibling)
                parent = parent->parent() ? parent->parent()
                                         : invisibleRootItem();
        }
        else
            return 0;
    }
    return createNewTask(parent, name, false);
}

```

Этот метод вызывается из `MainWindow::editAdd()` для добавления новой задачи. В начале путем сравнения с элементами перечисления `Insert` определяется, кто должен быть родителем новой задачи, а затем с помощью того же метода `createNewTask()`, что и в методе `load()`, создается новая задача с указанным именем (метод `editAdd()` передает в качестве имени строку «New Task») и флагом выполнения `false` (флажок не отмечен).

```

QStringList StandardTreeModel::pathForIndex(const QModelIndex &index)
const
{
    QStringList path;
    if (index.isValid()) {
        QStandardItem *item = itemFromIndex(index);
        while (item) {
            path.prepend(item->text());
            item = item->parent();
        }
    }
    return path;
}

```

Этот метод возвращает путь к задаче; как он применяется, мы уже видели (стр. 154, 156). Сначала мы помещаем текст переданного элемента в путь `path`, представленный объектом `QStringList`, а затем добавляем в начало списка текст родителя, его родителя и т. д., пока не дойдем до верхнего уровня. Имейте в виду, что вызов метода `QStandardItem::parent()` для элемента верхнего уровня возвращает `0` (несмотря на то, что владеет элементом объект `QStandardItemModel`).

```

QStandardItem *StandardTreeModel::itemForPath(const QStringList &path)
const
{

```

```

        return itemForPath(invisibleRootItem(), path);
    }

    QStandardItem *StandardItemModel::itemForPath(QStandardItem *root,
        const QStringList &path) const
    {
        Q_ASSERT(root);
        if (path.isEmpty())
            return 0;
        for (int row = 0; row < root->rowCount(); ++row) {
            QStandardItem *item = root->child(row, 0);
            if (item->text() == path.at(0)) {
                if (path.count() == 1)
                    return item;
                if ((item = itemForPath(item, path.mid(1))))
                    return item;
            }
        }
        return 0;
    }
}

```

Эти методы противоположны `pathForIndex()` — они принимают путь к задаче и возвращают соответствующий элемент. Открытый (первый) метод получает путь в виде аргумента и вызывает закрытый метод, передавая ему невидимый корневой элемент и путь. Закрытый метод обходит непосредственных потомков переданного элемента и ищет среди них тот, текст которого совпадает с первым текстом в пути. Обнаружив искомое, метод рекурсивно вызывает себя, передавая в качестве нового корня найденный элемент, а в качестве пути — исходный путь, из которого исключена первая (уже сопоставленная) строка. В конечном итоге либо будут найдены соответствия для всех компонентов пути и возвращен соответствующий этому пути элемент, либо какой-то компонент не найдется, и тогда метод вернет 0.

На этом мы завершаем рассмотрение приложения `timelog1` и подклассов `QStandardItem` и `QStandardItemModel`, которые в нем используются. В следующем разделе мы создадим пользовательскую древовидную модель с дополнительной функциональностью и сможем использовать ее вместо только что рассмотренной.

Создание пользовательских древовидных моделей

При обсуждении класса `QStandardItemModel` в контексте таблиц мы говорили, что обычно его использование — самый простой и быстрый способ получить работающую реализацию.

Однако в настоящее время класс `QStandardItemModel` предлагает для древовидных моделей меньше возможностей, чем можно достичь за счет написания пользовательской модели. Тем не менее почти всегда лучше начинать с `QStandardItemModel` – возможно, дополнительная функциональность, которую могут дать пользовательские модели, ни к чему, а писать такие модели сложнее, чем списковые или табличные. Так что применение `QStandardItemModel` поможет сэкономить немало времени. Но если все-таки необходимо предоставить пользователям возможность произвольно перемещать элементы в дереве – а это обычно имеет смысл только тогда, когда все элементы дерева однотипны и допускают произвольную вложенность (как наши задачи), – то не остается ничего другого, кроме как прибегнуть к пользовательской древовидной модели.

В этом разделе мы займемся созданием приложения `timelog2`, показанного на рис. 4.3. При его рассмотрении мы будем поступать так же, как для приложений `Zipcodes` из предыдущей главы, – одни файлы выкинем, другие добавим и воспользуемся директивами `#ifdef`, чтобы обеспечить возможность совместного использования как можно большей части кода. В данном случае мы заменим файлы `standarditem.{hpp,cpp}` и `standardtreemodel.{hpp,cpp}` файлами `taskitem.{hpp,cpp}` и `treemodel.{hpp,cpp}` и добавим в файл `timelog2.pro` строку `DEFINES += CUSTOM_MODEL`.



Рис. 4.3. Приложение `timelog2`

Как и для приложений `Zipcodes`, мы не станем показывать директивы `#ifdef`, а будем приводить код в том виде, в каком его видит компилятор, когда определена константа препроцессора `CUSTOM_MODEL`. (В предыдущем разделе мы поступали точно так же, только показывали тот код, который компилятор видит, когда константа `CUSTOM_MODEL` не определена.)

В первом подразделе мы рассмотрим различия между `timelog1` и `timelog2` в части, относящейся к программированию моделей и представлений, – большинство из них касаются поддержки перемещения элементов в пользовательском интерфейсе с помощью вырезания/вставки, перетаскивания мышью или посредством одного из действий перемещения. Далее будет рассмотрен класс `TaskItem`, представляющий задачи, а затем – класс `TreeModel`, являющийся подклассом `QAbstractItemModel` и предназначенный для хранения всех данных о задачах.

Изменение древовидной модели с помощью пользовательского интерфейса

Между данными-членами в классе главного окна приложений `timelog1` и `timelog2` есть только одно отличие: в `timelog2` упоминается пользовательский класс `TreeModel`, а не `StandardTreeModel`. Что касается методов, то в `timelog2` используется другой метод `hideOrShowDoneTask()` и предоставляется шесть новых методов для поддержки перемещения элементов: `editCut()`, `editPaste()`, `editMoveUp()`, `editMoveDown()`, `editPromote()` и `editDemote()`. В пользовательском интерфейсе также имеются действия, с помощью которых эти методы вызываются.

Конструктор `timelog2` отличается только заголовком окна приложения, но в методе `createModelAndView()` есть парочка существенных отличий, поэтому приведем его полностью.

```
void MainWindow::createModelAndView()
{
    model = new TreeModel(this);
    treeView->setDragDropMode(QAbstractItemView::InternalMove);
    treeView->setAllColumnsShowFocus(true);
    treeView->setItemDelegateForColumn(0, new RichTextDelegate);
    treeView->setModel(model);
    setCentralWidget(treeView);
}
```

Самое важное отличие состоит в том, что теперь используется класс `TreeModel`. Отметим также, что мы просим `QTreeView` поддерживать перетаскивание, – но лишь для перемещения элементов внутри самого дерева. (Как и в случае `timelog1`, мы опустили директиву `#ifdef`, подключающую тесты `ModelTest`, если соответствующий модуль доступен, хотя в исходном коде она есть.) Делегат рассматривается в главе 5.

Методы `createActions()` и `createMenusAndToolBar()`, которые вызываются из конструктора, отличаются лишь тем, что в `timelog2` создаются дополнительные действия.

Что касается соединений сигнал–слот, то слот `setDirty()` теперь соединен не с сигналом `QStandardItemModel::itemChanged()`, а с `QAbstractItemModel::dataChanged()`. Ну и, конечно, новые действия соединены с соот-

ветствующими им слотами – например, сигнал `triggered()`, генерируемый действием `editCutAction`, соединен со слотом `editCut()`.

Фрагмент кода из метода `MainWindow::load()`, показанный на стр. 154, в приложении `timelog2` оказывается несколько проще:

```
model->load(filename);
if (!taskPath.isEmpty()) {
    setCurrentIndex(model->indexOfPath(taskPath));
}
```

В приложении `timelog1` мы должны были искать элемент с помощью метода `StandardItemModel::itemForPath()`, а затем получить его модельный индекс, чтобы передать его в `setCurrentIndex()`; здесь же у нас есть метод `TreeModel::indexOfPath()`, которым можно воспользоваться напрямую.

Другие рассмотренные ранее методы также претерпели изменения: `editAdd()` стал гораздо короче и проще, а `hideOrShowDoneTask()` изменился лишь немного – мы покажем оба метода. Но метод `editDelete()` приводить не станем, так как единственное отличие от прежней версии состоит в том, что он может непосредственно работать с модельными индексами для получения имени и числа потомков удаляемого элемента, а не запрашивать эту информацию у элемента; само же удаление с помощью метода `removeRow()` осталось таким же, как раньше. Мы также покажем несколько новых методов, опуская те, что почти идентичны рассмотренным ранее.

```
void MainWindow::editAdd()
{
    QModelIndex index = treeView->currentIndex();
    if (model->insertRow(0, index)) {
        index = model->index(0, 0, index);
        setCurrentIndex(index);
        treeView->edit(index);
        setDirty();
        updateUi();
    }
}
```

В приложении `timelog1` мы должны были спрашивать пользователя, куда он хочет поместить новую строку: на верхний уровень, на тот же уровень, где и выделенная строка, или сделать ее потомком выделенной строки. Но поскольку `timelog2` позволяет без труда перемещать задачу по дереву, то мы можем всегда делать новую задачу дочерним элементом выделенной, считая, что пользователь впоследствии при необходимости переместит ее куда угодно.

Создав новую задачу (путем вставки строки перед первым потомком текущей задачи или на верхнем уровне, если дерево пусто), мы получаем модельный индекс элемента, содержащего имя новой задачи, и прокру-

чиваем дерево так, чтобы он появился на экране. Затем мы входим в режим редактирования (чтобы не заставлять пользователя лишний раз нажимать клавишу F2 – или Return в Mac OS X – либо дважды щелкать мышью), так что пользователь может сразу же заменить текст «New Task» более осмысленным.

Еще одно, не столь очевидное, преимущество добавления нового элемента в качестве потомка текущего состоит в том, что сразу после создания он невидим (если только не является задачей верхнего уровня), так что представление не обязано обращаться к методу `data()`, пока пользователь не перешел к новому элементу. В данном случае это не столь существенно, но, вообще говоря, все изменения в дереве лучше производить, когда элементы невидимы (то есть являются потомками свернутого родителя), так как это может избавить представление от необходимости вызывать метод `data()` для элементов, которые перемещены или удалены.

```
void MainWindow::editHideOrShowDoneTasks(bool hide)
{
    hideOrShowDoneTask(hide, QModelIndex());
}

void MainWindow::hideOrShowDoneTask(bool hide, const QModelIndex &index)
{
    bool hideThisOne = hide && model->isChecked(index);
    if (index.isValid())
        treeView->setRowHidden(index.row(), index.parent(), hideThisOne);
    if (!hideThisOne) {
        for (int row = 0; row < model->rowCount(index); ++row)
            hideOrShowDoneTask(hide, model->index(row, 0, index));
    }
}
```

Эти методы очень похожи на те, что мы уже видели в приложении `time-log1`, и логика у них такая же. Основное различие в том, что сейчас мы работаем в терминах модельных индексов, а не элементов, потому что пользуемся API класса `QAbstractItemModel` (дополненного нашими расширениями типа метода `isChecked()`, стр. 190), а не API класса `QStandardItemModel`, ориентированным на элементы.

```
void MainWindow::editCut()
{
    QModelIndex index = treeView->currentIndex();
    if (model->isTimedItem(index))
        stopTiming();
    setCurrentIndex(model->cut(index));
    editPasteAction->setEnabled(model->hasCutItem());
}
```

Этот слот, как и все слоты, относящиеся к перемещению, перепоручает основную часть работы модели. Это необходимо, так как при удалении

и перемещении элементов изменяется структура дерева, что следует отразить во всех ассоциированных с ним представлениях.

Если вырезанная задача хронометрируется, то мы останавливаем хронометраж, потому что нет смысла отсчитывать время для задачи, которая невидима и, возможно, будет удалена (если пользователь не вставит ее обратно в дерево). Затем мы просим модель вырезать задачу с указанным модельным индексом (вместе со всеми ее потомками) и делаем задачу, индекс которой возвращен¹, выделенной. Кроме того, мы активируем действие вставки, чтобы пользователь видел, что операция вставки сейчас доступна.

```
void MainWindow::editPaste()
{
    setCurrentIndex(model->paste(treeView->currentIndex()));
    editHideOrShowDoneTasks(
        editHideOrShowDoneTasksAction->isChecked());
}
```

Структура этого слота типична почти для всех слотов приложения `time-log2`, имеющих отношение к перемещению: он просит модель выполнить операцию, затем прокручивает дерево так, чтобы задача с возвращенным моделью индексом стала видна на экране, и делает эту задачу выделенной. В самом конце он скрывает или показывает выполненные задачи в зависимости от состояния флажка `editHideOrShowDoneTasksAction`.

Когда задача вставляется обратно в дерево, вместе с ней рекурсивно вставляются и все ее потомки, однако узел дерева, соответствующий вставленной задаче, оказывается свернутым, поэтому потомки не видны, пока пользователь явно не раскроет узел.

Слот `editMoveUp()` отличается только тем, что вместо метода `TreeModel::paste()` вызывается `TreeModel::moveUp()`. То же самое относится к слоту `editMoveDown()`, из которого вызывается `TreeModel::moveDown()`. Слоты `editPromote()` и `editDemote()` (вызывающие методы модели `TreeModel::promote()` и `TreeModel::demote()`) также очень похожи, только в начале останавливают хронометраж, если для выделенной задачи он был включен (как в слоте `editCut()`). И во всех случаях метод применяется к выделенной задаче и ее потомкам любого уровня.

Мы подробно рассмотрели относящиеся к пользовательскому интерфейсу методы и слоты и теперь обладаем достаточной информацией, чтобы разобраться в классе `TreeModel`, который используется для хранения данных о задачах. Но перед тем, как переходить к его реализации, мы должны ознакомиться с классом `TaskItem`, которым `TreeModel` используется для представления отдельных задач.

¹ Это элемент, расположенный выше по списку: либо «брат», либо родитель вырезанной задачи. — *Прим науч. ред.*

Специализированный класс элемента дерева

Нам необходим некий класс для представления одного элемента дерева. Работая с классом `QStandardItem`, мы были вынуждены разделять элементы для хранения имени задачи и времени, затраченного на ее выполнение в текущий день и в целом. Но в пользовательской модели никто не мешает нам хранить все данные в одном объекте `TaskItem` и возвращать из модели данные конкретного столбца, как того требует запрос.

Класс `TaskItem` предоставляет двоякую функциональность: методы для работы с данными элемента – именем задачи, состоянием ее выполнения и временем начала и окончания – и методы для управления дочерними элементами. Как мы увидим в следующем подразделе, все дерево представлено одним указателем на корневой элемент (безымянный объект `TaskItem`, который в `TreeModel` является эквивалентом невидимого корневого узла в `QStandardItemModel`), а все остальные задачи являются либо непосредственными потомками этого узла, либо потомками его потомков и т. д.

Большинство методов `TaskItem` определены в заголовочном файле, который мы рассмотрим в три приема: две группы методов и закрытые данные-члены, причем начнем с данных, чтобы создать контекст для обсуждения методов.

```
private:
    int minutesForTask(bool onlyForToday) const;
    QString m_name;
    bool m_done;
    QList<QPair<QDateTime, QDateTime> > m_dateTimes;
    TaskItem *m_parent;
    QList<TaskItem*> m_children;
};
```

Метод для вычисления количества минут, затраченных на задачу, мы рассматривать не будем, потому что к программированию моделей и представлений он отношения не имеет. У каждой задачи имеются данные-члены, отражающие содержимое *xml*-файла: имя, флаг выполнения и список моментов начала и окончания работы. Кроме того, для поддержки иерархической организации в каждой задаче есть указатель на родителя и список дочерних задач. Безымянный корневой элемент (который не сохраняется и не загружается, а заведен только для удобства программирования) – единственный элемент, для которого указатель на родителя равен 0.

Сравните данные, хранящиеся в `TaskItem`, с теми, что хранятся в `StandardItem` и двух подклассах `QStandardItem`, необходимых в приложении `timelog1`, которое основано на использовании класса модели `QStandardItemModel`. В обоих случаях хранятся одни и те же пользовательские данные (имя задачи, флаг выполнения и список моментов начала и окончания работы), но в `TaskItem` непроизводительные издержки сводятся

к указателю и списку указателей `QList`, тогда как в трех объектах `QStandardItem`, необходимых модели `QStandardItemModel`, приходится дополнительно хранить девять указателей, девять чисел типа `int`, три вектора `QVector` указателей и три вектора `QVector` значений (причем в каждом значении хранится `int` и `QVariant`) – это в версии `Qt 4.5.0`. Впрочем, вопрос о том, так ли существенно потребление дополнительной памяти, решается для каждого приложения индивидуально, и мы по-прежнему настаиваем, что почти всегда лучше начинать с `QStandardItemModel` и реализовывать пользовательский подкласс `QAbstractItemModel` только в случае, когда это необходимо из соображений производительности или для удовлетворения функциональных требований.

```
class TaskItem
{
public:
    explicit TaskItem(const QString &name=QString(),
                     bool done=false, TaskItem *parent=0);
    ~TaskItem() { qDeleteAll(m_children); }
    QString name() const { return m_name; }
    void setName(const QString &name) { m_name = name; }
    bool isDone() const { return m_done; }
    void setDone(bool done) { m_done = done; }
    QList<QPair<QDateTime, QDateTime> > dateTimes() const
    { return m_dateTimes; }
    void addDateTime(const QDateTime &start, const QDateTime &end)
    { m_dateTimes << qMakePair(start, end); }
    QString todaysTime() const;
    QString totalTime() const;
    void incrementLastEndTime(int msec);
```

Конструктор мы рассмотрим ниже, но опустим методы для вычисления времени, потраченного на задачу в текущий день и в целом, а также для инкремента момента окончания работы.

Деструктор реализовывать необходимо, потому что `TaskItem` не является подклассом `QObject`, поэтому управлять временем жизни объектов, представляющих задачи, мы должны самостоятельно. При удалении элемента мы удаляем всех его прямых потомков, те, в свою очередь, удаляют своих потомков и т. д., поэтому, чтобы удалить из дерева все задачи, достаточно удалить только корневой элемент.

```
TaskItem *parent() const { return m_parent; }
TaskItem *childAt(int row) const
{ return m_children.value(row); }
int rowOfChild(TaskItem *child) const
{ return m_children.indexOf(child); }
int childCount() const { return m_children.count(); }
bool hasChildren() const { return !m_children.isEmpty(); }
QList<TaskItem*> children() const { return m_children; }
void insertChild(int row, TaskItem *item)
{ item->m_parent = this; m_children.insert(row, item); }
```

```
void addChild(TaskItem *item)
{ item->m_parent = this; m_children << item; }
void swapChildren(int oldRow, int newRow)
{ m_children.swap(oldRow, newRow); }
TaskItem* takeChild(int row);
```

В методе `childAt()` мы предусмотрительно вызываем `QList::value()`, а не `QList::operator[]()`; тогда при запросе строки, находящейся вне диапазона, возвращается значение, созданное конструктором по умолчанию (то есть 0), — вместо аварийного завершения программы.

Когда новый элемент вставляется в конкретной строке или добавляется в конец, мы *обязаны* заново установить его родителя. Дело в том, что переданный объект `TaskItem` мог быть вырезан или перемещен из другого места, и тогда его родитель уже установлен, но, как правило, указывает не на тот элемент, что нужно.

В файле *taskitem.cpp* есть всего два метода, относящихся к программированию моделей и представлений: конструктор и метод `takeChild()`. Их мы и рассмотрим.

```
TaskItem::TaskItem(const QString &name, bool done, TaskItem *parent)
    : m_name(name), m_done(done), m_parent(parent)
{
    if (m_parent)
        m_parent->addChild(this);
}
```

Если для вновь созданной задачи родитель отличен от нуля, то необходимо добавить ее в конец списка потомков ее родителя.

```
TaskItem* TaskItem::takeChild(int row)
{
    TaskItem *item = m_children.takeAt(row);
    Q_ASSERT(item);
    item->m_parent = 0;
    return item;
}
```

Если задача исключается из дерева, то есть удаляется из списка потомков своего родителя, то мы должны обнулить указатель на родителя, отразив тот факт, что теперь у нее нет владельца. Это означает, что с этого момента за возвращенный указатель отвечаем мы сами и, значит, должны либо удалить его, либо как можно быстрее вставить элемент обратно в дерево, иначе произойдет утечка памяти.

Специализированный подкласс QAbstractItemModel для деревьев

Чтобы реализовать подкласс QAbstractItemModel для деревьев, который допускал бы редактирование элементов и изменение размера модели, мы обычно должны переопределить все или почти все методы, перечисленные в табл. 3.1 (стр. 137). Однако в случае приложения Timelog – да и вообще довольно часто для деревьев – переопределять insertColumns() и removeColumns() не нужно, так как количество столбцов фиксировано.

Для поддержки перемещения элементов (вместе с потомками) путем перетаскивания необходимо также реализовать методы, относящиеся к перетаскиванию, которые перечислены в табл. 4.1. API перетаскивания подразумевает сериализацию и десериализацию элементов модели, и, как станет понятно позже, мы можем воспользоваться теми же методами, что и для загрузки и сохранения. Поскольку мы хотим предоставить пользователю и другие способы перемещения элементов, то расширили API класса QAbstractItemModel, добавив несколько своих методов:

```
QModelIndex moveUp(const QModelIndex &index);
QModelIndex moveDown(const QModelIndex &index);
QModelIndex cut(const QModelIndex &index);
QModelIndex paste(const QModelIndex &index);
QModelIndex promote(const QModelIndex &index);
QModelIndex demote(const QModelIndex &index);
```

Таблица 4.1. API перетаскивания в классе QAbstractItemModel

Метод	Описание
dropMimeData(mimeData, dropAction, row, column, parent)	Этот метод вызывается в момент бросания; он должен десериализовать <i>mimeData</i> и воспользоваться этими данными для выполнения указанного действия <i>dropAction</i> для элемента с данной строкой <i>row</i> , столбцом <i>column</i> и родителем <i>parent</i>
mimeData(indexes)	Возвращает объект типа QMimeData, который содержит сериализованные данные, соответствующие переданному списку модельных индексов <i>indexes</i> ; вызывается самой моделью для подготовки данных к бросанию
mimeTypes()	Возвращает список QStringList типов MIME, которые описывают список модельных индексов
supportedDragActions()	Возвращает результат поразрядного ИЛИ одного или нескольких элементов перечисления Qt::DropActions (перечисления для операции перетаскивания не существует)
supportedDropActions()	Возвращает результат поразрядного ИЛИ одного или нескольких элементов перечисления Qt::DropActions (Qt::CopyAction, Qt::MoveAction и т. д.)

В версии Qt 4.6 появилось несколько новых защищенных методов, упрощающих перемещение элементов в модели: `beginMoveColumns()`, `endMoveColumns()`, `beginMoveRows()` и `endMoveRows()`. Мы не стали их использовать, чтобы сохранить совместимость с версией Qt 4.5. Для проектов, ориентированных на версии не ниже Qt 4.6, новые методы могут оказаться полезными, однако внимательно читайте документацию, поскольку у них есть ряд ограничений.

Помимо дополнительных пользовательских методов, в классе `TreeModel` есть методы, относящиеся к хронометражу (мы их не рассматриваем), и методы, аналогичные добавленным в класс `QStandardItemModel`: `clear()`, `load()`, `save()`, `pathForIndex()` и `indexForPath()`.

Кроме того, в классе `TreeModel` имеется ряд закрытых методов, которые мы рассмотрим тогда, когда это будет наиболее уместно, а также некоторые закрытые данные-члены.

```
private:
    QString m_filename;
    QIcon m_icon;
    TaskItem *timedItem;
    TaskItem *rootItem;
    TaskItem *cutItem;
```

Что касается переменных типа `TaskItems`, то `timedItem` — указатель на элемент, имеющий родителя, поэтому нам удалять его не следует. `rootItem` — корень дерева, и его необходимо удалить, когда того требует ситуация. `cutItem` — элемент, который вырезан, но еще не вставлен; если такой элемент существует, то его необходимо удалить в подходящий момент, например при открытии нового файла или выходе из приложения.

Все сказанное означает, что класс `TreeModel` можно подставить вместо `StandardTreeModel`, но при этом он обладает дополнительной функциональностью, в частности поддерживает вырезание и вставку элементов, перетаскивание мышью и перемещение.

Теперь мы перейдем к рассмотрению всех методов `TreeModel`, так или иначе связанных с архитектурой модель/представление. Начнем с конструктора и деструктора, затем поговорим о методах, реализующих API класса `QAbstractItemModel`, и о методе `isChecked()` (просто для полноты картины, потому что мы уже видели, как он используется), затем о методах, реализующих API перетаскивания, определенный в классе `QAbstractItemModel`, далее о методах перемещения и, наконец, о методах работы с файлом и путем к задаче.

```
explicit TreeModel(QObject *parent=0)
    : QAbstractItemModel(parent), timedItem(0), rootItem(0),
      cutItem(0) {}

~TreeModel() { delete rootItem; delete cutItem; }
```

Конструктор должен всего лишь инициализировать нулем данные-члены, являющиеся указателями, и вызвать конструктор базового класса, передав ему аргумент `parent`. Деструктор должен удалить корневой и вырезанный элементы (последний может быть равен 0). Мы полагаемся на то, что деструктор класса `TaskItem` рекурсивно удалит всех потомков задачи.

API класса `QAbstractItemModel` для деревьев

В этом подразделе мы рассмотрим методы, образующие API класса `QAbstractItemModel`, которые в классе `TreeModel` реализованы для поддержки возможности редактирования и изменения размера дерева (в части количества строк, но не столбцов). Они перечислены в табл. 3.1 (стр. 137).

```
enum Column {Name, Today, Total};

Qt::ItemFlags TreeModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags theFlags = QAbstractItemModel::flags(index);
    if (index.isValid()) {
        theFlags |= Qt::ItemIsSelectable|Qt::ItemIsEnabled;
        if (index.column() == Name)
            theFlags |= Qt::ItemIsUserCheckable|Qt::ItemIsEditable|
                Qt::ItemIsDragEnabled|Qt::ItemIsDropEnabled;
    }
    return theFlags;
}
```

Этот метод реализуется похожим образом в моделях всех типов. В данном случае мы делаем все элементы активными и допускающими выделение, но отметку флажком, редактирование и перетаскивание разрешаем только для имен задач. Ниже мы увидим, что перетаскивание реализовано следующим образом: если начать перетаскивание имени задачи, то перемещается вся задача (включая и элементы, относящиеся к времени) вместе со своими потомками. Однако мы предпочитаем перетаскивать имя, а не информацию о времени, поскольку так пользователю становится понятнее, что происходит.

```
const int ColumnCount = 3;

QVariant TreeModel::data(const QModelIndex &index, int role) const
{
    if (!rootItem || !index.isValid() || index.column() < 0 ||
        index.column() >= ColumnCount)
        return QVariant();
    if (TaskItem *item = itemForIndex(index)) {
        if (role == Qt::DisplayRole || role == Qt::EditRole) {
            switch (index.column()) {
                case Name: return item->name();
```

```

        case Today: return item->todaysTime();
        case Total: return item->totalTime();
        default: Q_ASSERT(false);
    }
}
if (role == Qt::CheckStateRole && index.column() == Name)
    return static_cast<int>(item->isDone() ? Qt::Checked
                               : Qt::Unchecked);
if (role == Qt::TextAlignmentRole) {
    if (index.column() == Name)
        return static_cast<int>(Qt::AlignVCenter|Qt::AlignLeft);
    return static_cast<int>(Qt::AlignVCenter|Qt::AlignRight);
}
if (role == Qt::DecorationRole && index.column() == Today &&
    timedItem && item == timedItem && !m_icon.isNull())
    return m_icon;
}
return QVariant();
}

```

Метод `data()` – ключевой для архитектуры модель/представление в Qt, поскольку он предоставляет доступ ко всем данным и метаданным. При обсуждении класса `TableModel` (стр. 140) мы говорили, что этот метод не предполагает обращения к реализации в базовом классе; мы *обязаны* вернуть недопустимый объект `QVariant`, если не обрабатываем какой-то случай.

Метод `itemForIndex()` возвращает указатель на хранящийся в дереве объект `TaskItem` по его модельному индексу; мы рассмотрим его чуть ниже.

Мы решили обрабатывать роли `Qt::DisplayRole` и `Qt::EditRole` как синонимы, поэтому возвращаем одни и те же данные для обеих ролей. Наши данные, строго говоря, не разбиты на столбцы, мы просто храним в дереве элементы, представляющие задачи, однако столбцы виджета-представления отображаются на поля данных или, как в случае столбцов, относящихся к времени, на вычисляемые значения. Мы также обрабатываем роль `Qt::CheckStateRole`, возвращая элемент перечисления, соответствующий текущему состоянию выполнения задачи.

Мы решили выравнивать имя задачи по левому краю, а время – по правому. Кроме того, мы возвращаем пиктограмму, если запрошены данные для роли `Qt::DecorationRole`, но только если запрос относится к столбцу `Today` и при этом элемент хронометрируется. Во всех остальных случаях и для всех прочих ролей мы возвращаем недопустимый объект `QVariant` и оставляем все дальнейшие заботы Qt.

```

TaskItem *TreeModel::itemForIndex(const QModelIndex &index) const
{
    if (index.isValid()) {
        if (TaskItem *item = static_cast<TaskItem*>(
            index.internalPointer()))

```

```

        return item;
    }
    return rootItem;
}

```

При создании объекта `QModelIndex` (методом `QAbstractItemModel::createIndex()`) мы можем, помимо строки и столбца, передать конструктору указатель (или числовой идентификатор). Для древовидных моделей очень часто передают указатель на соответствующий элемент в дереве, и, как будет видно далее, именно так мы и поступаем в классе `TreeModel`, когда создаем модельные индексы. Это упрощает получение указателя по данному индексу – достаточно вернуть внутренний указатель, хранящийся в модельном индексе. Если указателя нет или индекс недопустимый, то мы возвращаем указатель на корневой элемент (который будет равен 0, если в дереве нет ни одного элемента).

```

QVariant TreeModel::headerData(int section,
                                Qt::Orientation orientation, int role) const
{
    if (orientation == Qt::Horizontal && role == Qt::DisplayRole) {
        if (section == Name)
            return tr("Task/Subtask/...");
        else if (section == Today)
            return tr("Time (Today)");
        else if (section == Total)
            return tr("Time (Total)");
    }
    return QVariant();
}

```

Класс `QTreeView` в `Qt` поддерживает только горизонтальные заголовки, поэтому мы возвращаем их имена, когда это требуется. А поскольку метод `headerData()` следует тем же соглашениям, что и `data()`, то есть требует возвращать значение, а не вызывать метод базового класса, то мы возвращаем недопустимый `QVariant` во всех случаях, которые не обрабатываем сами.

```

int TreeModel::rowCount(const QModelIndex &parent) const
{
    if (parent.isValid() && parent.column() != 0)
        return 0;
    TaskItem *parentItem = itemForIndex(parent);
    return parentItem ? parentItem->childCount() : 0;
}

```

Счетчик строк для элемента дерева – это количество его *непосредственных* потомков (без рекурсии). Если родитель указан, но номер столбца отличен от 0, то мы должны вернуть 0, поскольку разрешаем иметь потомков только элементам в первом столбце. В противном случае мы находим задачу, соответствующую индексу родителя. Если индекс ро-

дителя недопустим, то метод `itemForIndex()` корректно вернет корневой элемент (который может быть равен 0, если в дереве нет элементов).

```
int TreeModel::columnCount(const QModelIndex &parent) const
{
    return parent.isValid() && parent.column() != 0 ? 0 : ColumnCount;
}
```

Пользовательская модель `TreeModel`, как и многие другие древовидные модели, имеет фиксированное количество столбцов, что упрощает реализацию метода и делает ее похожей на ту пользовательскую табличную модель, которую мы уже видели в приложении `Zipcodes` (стр. 143). Если индекс допустим и столбец не первый (имя задачи), значит, нас спрашивают о счетчике столбцов для столбца с данными о времени; в данной модели это не имеет смысла, поэтому мы возвращаем 0.

```
QModelIndex TreeModel::index(int row, int column,
                              const QModelIndex &parent) const
{
    if (!rootItem || row < 0 || column < 0 || column >= ColumnCount
        || (parent.isValid() && parent.column() != 0))
        return QModelIndex();
    TaskItem *parentItem = itemForIndex(parent);
    Q_ASSERT(parentItem);
    if (TaskItem *item = parentItem->childAt(row))
        return createIndex(row, column, item);
    return QModelIndex();
}
```

Этот метод возвращает модельные индексы пользователям модели и также используется внутри самой модели.

Помимо очевидных проверок корректности входных данных, мы проверяем еще и номер столбца родителя. Мы не возвращаем модельный индекс для тех элементов, у которых столбец родителя не равен 0, потому что потомки могут быть только у элементов в нулевом столбце.

Модельный индекс состоит из строки, столбца и указателя (или числового идентификатора). Для списковых и табличных моделей указатель обычно равен 0, но в древовидной модели это поле зачастую содержит указатель на соответствующий элемент дерева (или его числовой идентификатор). Далее мы находим родительскую задачу, обращаясь к методу `itemForIndex()` (стр. 183), и получаем дочерний элемент родителя с порядковым номером `row`. Затем вызываем метод `QAbstractItemModel::createIndex()`, передавая ему заданные строку и столбец и указатель на элемент, представляющий задачу, поскольку именно на этот элемент ссылается создаваемый модельный индекс. Указатель на задачу и становится внутренним указателем, хранящимся в модельном индексе.

Если мы не можем создать индекс, то должны вернуть недопустимый объект `QModelIndex`. Отметим, что в классе `QModelIndex` есть только два открытых конструктора: конструктор копирования и конструктор без ар-

гументов, который, следовательно, можно использовать для создания недопустимых модельных индексов. Таким образом, можно создать допустимый индекс, если вызвать метод `createIndex()` или воспользоваться конструктором копирования для копирования существующего индекса.

Структура этого метода – и большинства прочих показанных здесь методов древовидной модели, реализующих API класса `QAbstractItemModel`, – может быть повторена в любой древовидной модели, где используются указатели на объекты-элементы дерева, причем методы для управления потомками реализованы в классе, представляющем эти элементы (например, `TaskItem` со своими методами или его аналог), а в самой модели есть метод `itemForIndex()` (или эквивалентный ему). Таким образом, приведенный выше код можно при необходимости без труда адаптировать.

```
QModelIndex TreeModel::parent(const QModelIndex &index) const
{
    if (!index.isValid())
        return QModelIndex();
    if (TaskItem *childItem = itemForIndex(index)) {
        if (TaskItem *parentItem = childItem->parent()) {
            if (parentItem == rootItem)
                return QModelIndex();
            if (TaskItem *grandParentItem = parentItem->parent()) {
                int row = grandParentItem->rowOfChild(parentItem);
                return createIndex(row, 0, parentItem);
            }
        }
    }
    return QModelIndex();
}
```

Вернуть модельный индекс родителя элемента не так просто, как может показаться, несмотря на то что в объектах `TaskItem` хранится указатель на родителя. Дело в том, что мы не можем напрямую отобразить указатель `TaskItem` на модельный индекс. Поэтому требуется найти родительский элемент для данного элемента, а затем отыскать строку, которую этот элемент занимает в списке потомков *своего* родителя (то есть строку родителя в списке потомков деда). Зная эту строку и имея указатель на родителя, мы вызываем метод `createIndex()`, чтобы создать модельный индекс, соответствующий родителю элемента.

Отметим, что если родительский элемент – корень дерева, то мы возвращаем недопустимый модельный индекс – в архитектуре модель/представление в Qt действует соглашение о том, что родитель элемента верхнего уровня имеет недопустимый модельный индекс (а не является

корневым элементом, если в модели таковой предусмотрен), и мы следуем этому соглашению в своем коде.

На рис. 4.4 показаны отношения между элементами, родителями и строками. Здесь родителем элемента **А** является элемент **Р**, то есть **А** – это первый потомок **Р** (строка 0), а родителем элемента **Р** является элемент **GP**, то есть **Р** – второй потомок **GP** (строка 1).

```
bool setHeaderData(int, Qt::Orientation, const QVariant&,
    int=Qt::EditRole) { return false; }
```

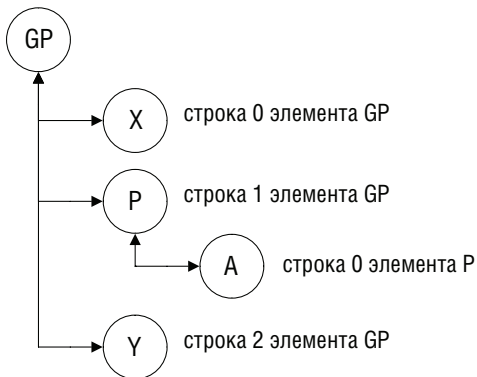


Рис. 4.4. Родитель элемента – это элемент в списке потомков деда с порядковым номером *row*

Мы решили, что заголовки нельзя изменять, поэтому такой тривиальной реализации в заголовочном файле будет достаточно.

```
bool TreeModel::setData(const QModelIndex &index,
    const QVariant &value, int role)
{
    if (!index.isValid() || index.column() != Name)
        return false;
    if (TaskItem *item = itemForIndex(index)) {
        if (role == Qt::EditRole)
            item->setName(value.toString());
        else if (role == Qt::CheckStateRole)
            item->setDone(value.toBool());
        else
            return false;
        emit dataChanged(index, index);
        return true;
    }
    return false;
}
```

Этот метод используется для поддержки редактирования элементов задач – в частности, имени и флага выполнения задачи. Нам не нужно забивать себе голову деталями редактирования – пользовательский делегат для форматированного текста занимается как редактированием имен задач, так и переключением состояния выполнения с помощью флажка (стр. 216).

Если редактирование произведено, то мы *обязаны* сгенерировать сигнал `dataChanged()`, передав ему модельные индексы, определяющие диапазон измененных элементов, и вернуть `true`; в противном случае мы должны вернуть `false`. В первом аргументе передается индекс левого верхнего, а во втором – индекс правого нижнего угла прямоугольной области, охватывающей элементы модели. В данном случае мы редактируем только один модельный индекс, поэтому в обоих аргументах передается один и тот же индекс.

```
bool TreeModel::insertRows(int row, int count, const QModelIndex &parent)
{
    if (!rootItem)
        rootItem = new TaskItem;
    TaskItem *item = parent.isValid() ? itemForIndex(parent) : rootItem;
    beginInsertRows(parent, row, row + count - 1);
    for (int i = 0; i < count; ++i) {
        TaskItem *item = new TaskItem(tr("New Task"), false);
        parentItem->insertChild(row, item);
    }
    endInsertRows();
    return true;
}
```

API архитектуры модель/представление *требует* вызывать метод `QAbstractItemModel::beginInsertRows()` перед началом вставки строк в модель и метод `QAbstractItemModel::endInsertRows()` – по завершении вставки. Обращение к `beginInsertRows()` можно дословно копировать в другие реализации – аргументами этого метода являются модельный индекс родителя, номер строки, начиная с которого вставляются новые строки, и номер последней из вставленных строк.

Сначала мы проверяем, что существует корневой элемент; его не будет, если, например, пользователь только что выполнил операцию `File → New`. Затем мы получаем элемент, который будет родителем для всех вставленных строк – это либо элемент, хранящийся в переданном индексе родителя (если этот индекс допустим), либо корневой элемент (тогда все вставленные строки будут располагаться на верхнем уровне). Далее мы создаем `count` новых элементов-задач, задавая для каждой имя и состояние выполнения по умолчанию и делая каждую потомком найденного ранее родительского элемента в заданной строке. (Напомним, конструктор `TaskItem` устроен так, что если задан ненулевой родитель, то создаваемый элемент добавляет себя в конец списка потомков этого родителя; стр. 179.)

Этот метод вызывается, когда некоторая задача (вместе со своими потомками) перетаскивается и бросается в другое место дерева, – чтобы вставить брошенные строки. Отметим, что есть еще метод `insertRow()` (который мы использовали в `MainWindow::editAdd()`; стр. 174), но его переопределять не нужно, поскольку имеющаяся в базовом классе реализация и так вызывает `insertRows()` со значением `count`, равным 1.

```
bool TreeModel::removeRows(int row, int count, const QModelIndex &parent)
{
    if (!rootItem)
        return false;
    TaskItem *item = parent.isValid() ? itemForIndex(parent) : rootItem;
    beginRemoveRows(parent, row, row + count - 1);
    for (int i = 0; i < count; ++i)
        delete item->takeChild(row);
    endRemoveRows();
    return true;
}
```

API архитектуры модель/представление *требует* вызывать метод `QAbstractItemModel::beginRemoveRows()` перед началом удаления строк из модели и метод `QAbstractItemModel::endRemoveRows()` – по завершении удаления. Обращение к `beginRemoveRows()` можно дословно копировать в другие реализации.

Если корневого элемента нет, значит, дерево пусто, так что и удалять нечего – в таком случае мы сразу возвращаем `false`. В противном случае, чтобы удалить `count` строк, начиная со строки с номером `row`, мы просто удаляем элемент, находящийся в указанной строке, `count` раз. В первый раз будет удален элемент с номером `row`, во второй – с номером `row + 1` (поскольку после предыдущего удаления он оказался в позиции `row`) и т. д. Естественно, потомки удаляемых элементов тоже рекурсивно удаляются. Отметим, что метод `TaskItem::takeChild()` (стр. 179) «отрывает» заданный элемент от его родителя, устанавливает в нем значение указателя на родителя 0 и возвращает этот элемент (у которого теперь нет ни родителя, ни владельца), после чего мы его незамедлительно удаляем.

Этот метод вызывается, когда некоторая задача (вместе с потомками) перетаскивается и бросается, – чтобы удалить перемещенные и уже брошенные строки, ведь брошенные строки заново вставляются методом `insertRows()`.

На этом мы завершаем рассмотрение переопределенных методов, необходимых для реализации редактируемой древовидной модели, допускающей изменение размера (то есть добавление и удаление строк).

Напоследок мы рассмотрим один крохотный метод, который дополняет API класса `QAbstractItemModel`, поскольку он встречался нам по ходу

изложения. А затем перейдем к группам дополнительных методов для поддержки перетаскивания и перемещения элементов, а также их загрузки и сохранения в файле на диске.

```
bool TreeModel::isChecked(const QModelIndex &index) const
{
    if (!index.isValid())
        return false;
    return data(index, Qt::CheckStateRole).toInt() == Qt::Checked;
}
```

Это вспомогательный метод, который мы добавили только для того, чтобы код метода `MainWindow::hideOrShowDoneTask()` было проще читать (стр. 175). Хотя мы обязаны переопределить некоторые части API класса `QAbstractItemModel`, никто не мешает нам вводить дополнительные методы, если мы сочтем это разумным.

API класса `QAbstractItemModel` для перетаскивания

В этом подразделе мы рассмотрим те методы из API класса `QAbstractItemModel`, которые реализованы в классе `TreeModel` для поддержки перетаскивания мышью. Все они перечислены в табл. 4.1 (стр. 180). В Qt перетаскивание устроено следующим образом: перетаскиваемые элементы сначала нужно сериализовать, а в момент бросания – десериализовать¹.

```
Qt::DropActions supportedDragActions() const
{ return Qt::MoveAction; }

Qt::DropActions supportedDropActions() const
{ return Qt::MoveAction; }
```

Реализация обоих методов тривиальна и находится в заголовочном файле. Здесь мы говорим, что единственное поддерживаемое действие перетаскивания – это перемещение. Для нашего примера с задачами это совершенно естественное предположение, но в других случаях может понадобиться поддерживать копирование или одновременно копирование и перемещение (в последнем случае следовало бы вернуть `Qt::MoveAction|Qt::CopyAction`).

```
const QString MimeType = "application/vnd.qtrac.xml.task.z";

QStringList TreeModel::mimeTypes() const
{
    return QStringList() << MimeType;
}
```

Механизм перетаскивания (и работы с буфером обмена) в Qt основан на использовании типов MIME для определения характера данных.

¹ Отметим, что этот код работает в версии Qt 4.5 и старше, но, похоже, в Qt 4.4 и более ранних версиях он нестабилен.

(Краткое введение в типы MIME приведено на стр. 89.) Мы должны переопределить метод `mimeType()` так, чтобы он возвращал список типов MIME, поддерживаемых нашей моделью. Для идентификации данных о задаче мы создали специальный тип MIME. Как станет ясно чуть ниже, мы выбрали для перетаскивания задачи тот же XML-формат, что для сохранения и загрузки данных о ней. Это решение требует больше памяти, чем компактный двоичный формат, зато позволяет повторно использовать тот же код сериализации и десериализации, что и для загрузки и сохранения.

```
const int MaxCompression = 9;

QMimeType *TreeModel::mimeType(const QModelIndexList &indexes) const
{
    Q_ASSERT(indexes.count());
    if (indexes.count() != 1)
        return 0;
    if (TaskItem *item = itemForIndex(indexes.at(0))) {
        QMimeType *mimeType = new QMimeType;
        QByteArray xmlData;
        QXmlStreamWriter writer(&xmlData);
        writeTaskAndChildren(&writer, item);
        mimeType->setData(MimeType, qCompress(xmlData, MaxCompression));
        return mimeType;
    }
    return 0;
}
```

Этот метод автоматически вызывается в начале перетаскивания, ему передается список модельных индексов перетаскиваемых элементов. В случае древовидной модели, если перетаскивается некоторый элемент, то в списке будет присутствовать его индекс, но *не* индексы его потомков, хотя ответственность за их перетаскивание вместе с родителем (и с их потомками любого уровня) с нас никто не снимает.

Метод `mimeType()` можно использовать для поддержки перетаскивания нескольких элементов (поскольку он принимает список модельных индексов). Однако мы решили ограничиться только одним элементом (разумеется, вместе со всеми его потомками). Мы получаем указатель на этот элемент и создаем массив байтов `QByteArray`, в который записываем все данные задачи в том же XML-формате, который применяется для хранения задач на диске. Затем с помощью метода `QXmlStreamWriter` выводим задачу вместе с ее потомками в виде XML-данных (как это выглядит, показано на рис. 4.2 на стр. 164). Выведенные данные сжимаются с максимальной плотностью (то есть результат занимает меньше места, но ценой большего времени сжатия), чтобы уменьшить расход памяти, поскольку XML-данные сильно избыточны. То, что получилось, мы сохраняем как данные объекта `QMimeType`. Ответственность за

возвращенный методом setData объект QMimeData принимает на себя Qt, так что нам удалять его не нужно.

```
void TreeModel::writeTaskAndChildren(QXmlStreamWriter *writer,
                                     TaskItem *task) const
{
    if (task != rootItem) {
        writer->writeStartElement(TaskTag);
        writer->writeAttribute(NameAttribute, task->name());
        writer->writeAttribute(DoneAttribute, task->isDone() ? "1" : "0");
        QListIterator<
            QPair<QDateTime, QDateTime> > i(task->dateTimes());
        while (i.hasNext()) {
            const QPair<QDateTime, QDateTime> &dateTime = i.next();
            writer->writeStartElement(WhenTag);
            writer->writeAttribute(StartAttribute,
                                   dateTime.first.toString(Qt::ISODate));
            writer->writeAttribute(EndAttribute,
                                   dateTime.second.toString(Qt::ISODate));
            writer->writeEndElement(); // WHEN
        }
    }
    foreach (TaskItem *child, task->children())
        writeTaskAndChildren(writer, child);
    if (task != rootItem)
        writer->writeEndElement(); // TASK
}
```

Это метод применяется для записи XML-представления элемента с помощью переданного объекта QXmlStreamWriter. Он мало чем отличается от одноименного метода в классе StandardTreeModel (стр. 167).

Мы никогда не записываем безымянный корневой элемент, потому что он нужен только для удобства программирования и не является частью данных. Записав данные самого элемента, мы далее рекурсивно записываем данные всех его потомков, так что если этот метод вызван для корневого элемента, то будет записано все дерево, хотя в данной программе он применяется только для записи перетаскиваемого элемента (с потомками).

```
bool TreeModel::dropMimeData(const QMimeData *mimeData,
                             Qt::DropAction action, int row, int column,
                             const QModelIndex &parent)
{
    if (action == Qt::IgnoreAction)
        return true;
    if (action != Qt::MoveAction || column > 0 ||
        !mimeData || !mimeData->hasFormat(MimeType))
        return false;
    if (TaskItem *item = itemForIndex(parent)) {
        emit stopTiming();
```

```
        QByteArray xmlData = qUncompress(mimeData->data(MimeType));
        QXmlStreamReader reader(xmlData);
        if (row == -1)
            row = parent.isValid() ? parent.row() : rootItem->childCount();
        beginInsertRows(parent, row, row);
        readTasks(&reader, item);
        endInsertRows();
        return true;
    }
    return false;
}
```

Этот метод автоматически вызывается в момент бросания. Если действие допустимо – в данном примере допустимым является перемещение, – то мы первым делом выясняем, на какой элемент брошены данные. Механизм перетаскивания устроен так, что перетаскиваемые элементы удаляются, а затем заново создаются точно в таком же виде, только в точке бросания. Это означает, что после завершения перетаскивания все указатели на перемещенные элементы становятся недействительными. Чтобы учесть этот факт, мы генерируем сигнал `stopTiming()`, который сообщает всем подключенным к нему объектам `QObject` (в данном случае `MainWindow`) о необходимости прекратить хронометраж элемента после бросания. (Конечно, можно было бы проверить, находится ли хронометрируемый элемент в числе перемещенных, и сгенерировать сигнал `stopTiming()` только в этом случае, но наше решение быстрее и обеспечивает согласованное поведение.)

Зная элемент, под которым должны располагаться перемещенные элементы, мы распаковываем данные, хранящиеся в `QMimeData`, и получаем их XML-представление. Затем вызываем метод `readTasks()`, который повторно создает брошенную задачу (и рекурсивно ее потомков) в виде дочернего элемента родителя, на который она была брошена.

Если бросание завершилось успешно, мы *обязаны* вернуть `true`, иначе `false`. В случае перемещения Qt автоматически вызывает метод `removeRows()`, чтобы удалить исходные элементы, которые теперь находятся в новом месте.

Мы не знаем, сколько именно строк будет вставлено, потому что не видим, описывает ли XML-представление элемент с потомками. Но это и неважно, потому что с точки зрения представления бросание добавляет ровно один элемент – видимый или невидимый, а сколько элементов находится под ним, не имеет значения, так как все они свернуты. Если переданный номер строки допустим, то вставка происходит после этой строки, иначе после строки родителя, а если и она недопустима, то на верхнем уровне. Обращения к методам `beginInsertRows()` и `endInsertRows()` обязательны для правильной работы представления.

Отметим, что модуль тестирования моделей проявляет чрезмерное рвение при проверке строк, вставляемых в результате бросания. Нам при-

шлось закомментировать две строки (468 и 477), чтобы предотвратить ложные сообщения об ошибке.

К сожалению, на некоторых платформах механизм перетаскивания внутри деревьев ведет себя не очень стабильно. Например, при работе с версией Qt 4.5 в Linux совсем несложно вызвать сбой программы. А в Mac OS X (для Qt 4.5 и Qt 4.6) перетаскивание в большинстве случаев работает нормально, но иногда программа отказывается бросать данные на первые несколько элементов. К счастью, эти проблемы не проявляются в Windows, да и в любом случае перемещение, равно как и уменьшение/увеличение уровня элемента с помощью кнопок на панели инструментов или клавиатуры, дает пользователю такие же возможности, как перетаскивание мышью.

```
void TreeModel::readTasks(QXmlStreamReader *reader, TaskItem *task)
{
    while (!reader->atEnd()) {
        reader->readNext();
        if (reader->isStartElement()) {
            if (reader->name() == TaskTag) {
                const QString name = reader->attributes()
                    .value(NameAttribute).toString();
                bool done = reader->attributes().value(DoneAttribute) == "1";
                task = new TaskItem(name, done, task);
            }
            else if (reader->name() == WhenTag) {
                const QDateTime start = QDateTime::fromString(
                    reader->attributes().value(StartAttribute)
                        .toString(), Qt::ISODate);
                const QDateTime end = QDateTime::fromString(
                    reader->attributes().value(EndAttribute)
                        .toString(), Qt::ISODate);
                Q_ASSERT(task);
                task->addDateTime(start, end);
            }
        }
        else if (reader->isEndElement()) {
            if (reader->name() == TaskTag) {
                Q_ASSERT(task);
                task = task->parent();
                Q_ASSERT(task);
            }
        }
    }
}
```

Этот метод служит для считывания данных задачи из XML-представления и размещения ее в качестве потомка переданного элемента. Он работает рекурсивно, считывая и размещая также всех потомков данной задачи. Структурно код очень похож на ранее написанный метод `StandardTreeModel::load()` (стр. 169).

Этот метод можно использовать для загрузки всего *xml*-файла задач, если вызвать его для корневого элемента, но здесь мы применяем его только для воссоздания перемещенного элемента (с потомками) в нужном месте дерева.

На этом заканчивается рассмотрение переопределенных методов класса `QAbstractItemModel`, необходимых для поддержки перетаскивания. На самом деле написанный код будет работать также для списковых и табличных моделей, потому что во всех трех случаях Qt применяет один и тот же подход. Таким образом, адаптировать код к пользовательским списковым и табличным моделям не составит труда.

Методы для перемещения элементов в дереве

Для редактируемых древовидных моделей, в которых все элементы однотипы, а глубина вложенности произвольна, имеет смысл предоставить и другие способы перемещения элементов, помимо перетаскивания мышью. Возможность перемещать элементы с помощью клавиатуры очень пригодилась бы людям, которые не могут или не хотят пользоваться мышью. Ну а поскольку все обсуждаемые ниже методы перемещения вызываются в результате действий `QAction`, то, конечно же, любители мыши смогут воспользоваться ими путем выбора из меню или нажатия кнопок на панели инструментов.

Мы реализовали три группы методов: для перемещения вверх и вниз в пределах одного уровня, для вырезания элемента и вставки его в другое место и для увеличения или уменьшения уровня элемента в дереве. Последнее означает операцию, в результате которой потомок оказывается на одном уровне со своим родителем или превращается в потомка своего «брата». Естественно, что эти методы, как и в случае перетаскивания, распространяются не только на выделенный элемент, но и рекурсивно на всех его потомков.

```
QModelIndex TreeModel::moveUp(const QModelIndex &index)
{
    if (!index.isValid() || index.row() <= 0)
        return index;
    TaskItem *item = itemForIndex(index);
    Q_ASSERT(item);
    TaskItem *parent = item->parent();
    Q_ASSERT(parent);
    return moveItem(parent, index.row(), index.row() - 1);
}
```

Элемент можно переместить вверх при условии, что он не первый на своем уровне, то есть порядковый номер его строки больше 0. Если это так, то мы вызываем вспомогательный метод `moveItem()`, передавая ему родителя элемента, номер текущей (прежней) строки элемента и номер новой строки, который в случае перемещения вверх на одну позицию на единицу меньше прежнего номера.

Метод `moveDown()` (не показан) отличается только тем, что перемещаемый элемент не должен быть последним на своем уровне, а номер новой строки получается на единицу больше номера текущей.

```
QModelIndex TreeModel::moveItem(TaskItem *parent, int oldRow, int newRow)
{
    Q_ASSERT(0 <= oldRow && oldRow < parent->childCount() &&
             0 <= newRow && newRow < parent->childCount());
    parent->swapChildren(oldRow, newRow);
    QModelIndex oldIndex = createIndex(oldRow, 0, parent->childAt(oldRow));
    QModelIndex newIndex = createIndex(newRow, 0, parent->childAt(newRow));
    emit dataChanged(oldIndex, newIndex);

    return newIndex;
}
```

Этот метод вызывается из `moveUp()` и `moveDown()` для выполнения перемещения. Метод `TaskItem::swapChildren()` обращается к `QList::swap()`, чтобы переставить два элемента списка потомков данной задачи. Закончив перемещение, `moveItem()` генерирует сигнал `dataChanged()`, чтобы уведомить все представления о том, что два элемента модели изменились. Возвращается модельный индекс перемещенного элемента в его новой позиции.

Как обычно, при вызове `createIndex()` мы передаем номер строки и столбца – последний для данной модели всегда равен 0 – и указатель на объект `TaskItem`, представляющий элемент.

На рис. 4.5 изображен процесс перемещения элемента – можно считать, что он иллюстрирует как перемещение **А** вниз, так и перемещение **В** вверх, поскольку конечный результат один и тот же. Закрашенными кружочками на правом рисунке представлены элементы, затронутые перемещением: **А** и **В**, которые, собственно, и перемещались, и их родитель **Р** – у него изменился список его потомков.

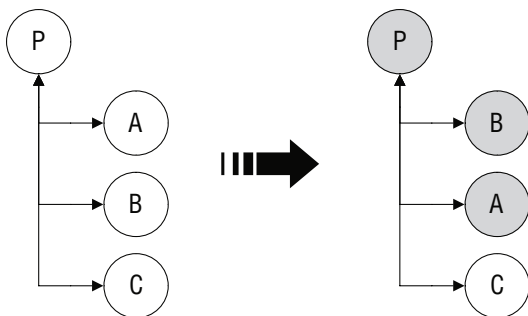


Рис. 4.5. Перемещение элемента вверх и вниз посредством обмена местами со своим братом

Этот метод, как и большинство относящихся к перемещению, возвращает модельный индекс. Как правило, и в данном случае в том числе, речь идет о модельном индексе перемещенного элемента. Индекс возвращается для того, чтобы вызывающая программа могла прокрутить дерево до данного элемента и выделить его. Это особенно удобно пользователю при перемещении элемента вверх и вниз по списку, потому что, один раз выделив элемент, он может снова и снова выполнять действие Up (или нажимать комбинацию клавиш Ctrl+↑ либо ⌘+↑ в Mac OS X), перемещая тем самым элемент вверх, пока тот не станет первым потомком родителя. Точно так же можно, наоборот, выполнять действие Down (или нажимать комбинацию клавиш Ctrl+↓ либо ⌘+↓), перемещая элемент вниз, пока тот не станет последним потомком родителя.

```
QModelIndex TreeModel::cut(const QModelIndex &index)
{
    if (!index.isValid())
        return index;
    delete cutItem;
    cutItem = itemForIndex(index);
    Q_ASSERT(cutItem);
    TaskItem *parent = cutItem->parent();
    Q_ASSERT(parent);
    int row = parent->rowOfChild(cutItem);
    Q_ASSERT(row == index.row());
    beginRemoveRows(index.parent(), row, row);
    TaskItem *child = parent->takeChild(row);
    endRemoveRows();
    Q_ASSERT(child == cutItem);

    if (row > 0) {
        --row;
        return createIndex(row, 0, parent->childAt(row));
    }
    TaskItem *grandParent = parent->parent();
    Q_ASSERT(grandParent);
    return createIndex(grandParent->rowOfChild(parent), 0, parent);
}
```

Этот метод устроен так же, как большинство относящихся к перемещению: сначала выполняется действие, затем создается модельный индекс, который возвращается вызывающей программе, чтобы та могла сделать элемент видимым на экране и выделить его.

Сначала мы удаляем элемент, на который указывает `cutItem`; это безопасно, даже если указатель равен 0, если же нет, то старый вырезанный элемент следует удалить (вместе со всеми потомками), поскольку вставить его уже никуда нельзя. Далее мы получаем указатель на элемент-задачу, который должен быть вырезан, и сохраняем его в `cutItem`; кроме того, мы получаем родителя этого элемента и номер строки в списке потомков родителя. Затем мы вызываем метод `beginRemoveRows()`, чтобы

уведомить инфраструктуру о том, что собираемся удалить элемент, после чего исключаем вырезанный элемент из списка потомков его родителя. Теперь элемент `cutItem` остался сиротой, и мы должны будем удалить его, когда наступит подходящий момент. (На самом деле он удаляется в трех методах: этом, `clear()` и в деструкторе класса `TreeModel`.) Исключив элемент из списка, мы вызываем метод `endRemoveRows()`, чтобы уведомить инфраструктуру о завершении удаления. (Утверждения `Q_ASSERT`, в которых сравниваются номера строк, а также `child` и `cutItem` – просто проверки на всякий случай.)

Когда элемент вырезается из дерева, все ассоциированные с моделью представления автоматически делают текущим «ближайший» к нему элемент. Представление считает ближайшим предыдущего брата, если таковой имеется, или следующего брата, если он существует, или родителя. Мы предпочитаем выбирать предыдущего брата, а если его нет, то родителя, и возвращаем модельный индекс того элемента, который хотим сделать текущим, ожидая, что вызывающая программа передаст его написанному нами методу `setCurrentIndex()`. Если же поведение по умолчанию нас устраивает, то можно было бы объявить этот метод как `void`, то есть не возвращать ничего, а просто завершить его обращением к `endRemoveRows()`. Однако для других относящихся к перемещению методов такой свободы выбора у нас нет – они обязаны сообщать ассоциированным представлениям, какой элемент сделать текущим.

```
QModelIndex TreeModel::paste(const QModelIndex &index)
{
    if (!index.isValid() || !cutItem)
        return index;
    TaskItem *sibling = itemForIndex(index);
    Q_ASSERT(sibling);
    TaskItem *parent = sibling->parent();
    Q_ASSERT(parent);
    int row = parent->rowOfChild(sibling) + 1;
    beginInsertRows(index.parent(), row, row);
    parent->insertChild(row, cutItem);
    TaskItem *child = cutItem;
    cutItem = 0;
    endInsertRows();
    return createIndex(row, 0, child);
}
```

Этот метод вставляет элемент (вместе с потомками) обратно в дерево. Мы решили при вставке вырезанного элемента делать его братом выделенного элемента, размещая его в строке, следующей за выделенным элементом.

Следствием такого решения является тот факт, что пользователь никогда не сможет сделать вырезанный элемент первым в списке потомков – для этого сначала придется вставить его после первого, а потом поднять на одну позицию. С другой стороны, чтобы сделать вырезанный элемент

последним, достаточно вставить его после текущего последнего элемента. Если бы мы приняли противоположное решение – вставлять вырезанный элемент над текущим, то пользователь легко смог бы вставлять в первую позицию, но для вставки в конец списка пришлось бы выполнить две операции. Наконец, можно было бы показать всплывающее меню с такими пунктами: *Paste Before Current Item* (Вставить перед текущим элементом) и *Paste After Current Item* (Вставить после текущего элемента).

Мы получаем элемент по модельному индексу выделенного элемента – он станет братом вставленного элемента. Затем мы находим родителя этого брата и выясняем, какую позицию в списке потомков своего родителя занимает брат. Далее вызываем метод `beginInsertRows()`, чтобы уведомить инфраструктуру о том, что собираемся вставить строку, и вставляем вырезанный элемент в следующую позицию.

Завершив вставку, мы обнуляем указатель `cutItem`, поскольку повторно вставлять уже помещенный в дерево элемент не следует. Затем вызываем метод `endInsertRows()`, чтобы уведомить инфраструктуру о том, что вставка завершена. И наконец, возвращаем модельный индекс только что вставленного элемента, чтобы представление могло сделать его видимым на экране и выделить.

```
QModelIndex TreeModel::promote(const QModelIndex &index)
{
    if (!index.isValid())
        return index;
    TaskItem *item = itemForIndex(index);
    Q_ASSERT(item);
    TaskItem *parent = item->parent();
    Q_ASSERT(parent);
    if (parent == rootItem)
        return index; // всегда элемент верхнего уровня

    int row = parent->rowOfChild(item);
    TaskItem *child = parent->takeChild(row);
    Q_ASSERT(child == item);
    TaskItem *grandParent = parent->parent();
    Q_ASSERT(grandParent);
    row = grandParent->rowOfChild(parent) + 1;
    grandParent->insertChild(row, child);
    QModelIndex newIndex = createIndex(row, 0, child);
    emit dataChanged(newIndex, newIndex);
    return newIndex;
}
```

Уменьшение уровня, или повышение, означает, что элемент делается сыном своего деда, занимая строку после своего бывшего родителя. И разумеется, эта операция рекурсивно затрагивает всех потомков элемента. На рис. 4.6 показан результат повышения элемента В. Закрашенными кружочками показаны элементы, затронутые этой операцией: элемент

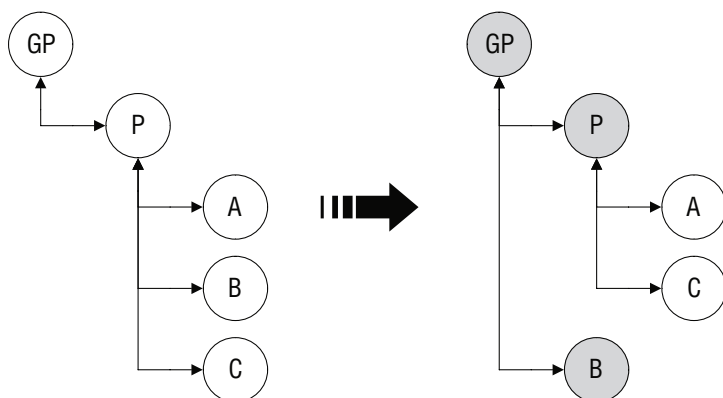


Рис. 4.6. Повышаемый элемент становится сыном своего деда

GP, дед элемента B, стал его родителем, а из списка потомков исходного родителя B этот элемент исключен.

Сначала мы получаем модельные индексы повышаемого элемента и его родителя. Если родителем повышаемого элемента является корневой элемент, значит, он уже находится на верхнем уровне и дальше его повысить нельзя, поэтому мы ничего не делаем и просто возвращаем его модельный индекс. В противном случае находим порядковый номер строки повышаемого элемента в списке потомков его родителя и отсоединяем его от родителя – в этот момент элемент оказывается сиротой и им никто не владеет, поскольку метод `TaskItem::takeChild()` (стр. 179) исключает элемент из списка потомков родителя и обнуляет указатель на родителя.

С помощью утверждения `Q_ASSERT` мы проверяем, что отсоединенный от родителя элемент действительно тот, который мы собираемся повысить. Затем мы находим деда и строку, которую родитель занимает в списке потомков деда. Далее повышаемый элемент вставляется после своего бывшего родителя; метод `TaskItem::insertChild()` изменяет родителя вставляемого им элемента, так что после его выполнения элемент снова находится в дереве и имеет родителя.

В самом конце мы создаем модельный индекс повышенного элемента и генерируем сигнал `dataChanged()`, чтобы уведомить ассоциированные представления о том, что модель изменилась. Наконец, модельный индекс повышенного элемента возвращается вызывающей программе.

```
QModelIndex TreeModel::demote(const QModelIndex &index)
{
    if (!index.isValid())
        return index;
    TaskItem *item = itemForIndex(index);
    Q_ASSERT(item);
```

```

TaskItem *parent = item->parent();
Q_ASSERT(parent);
int row = parent->rowOfChild(item);
if (row == 0)
    return index; // Нет предыдущего брата, под который можно
                  // было бы вставить этот элемент
TaskItem *child = parent->takeChild(row);
Q_ASSERT(child == item);
TaskItem *sibling = parent->childAt(row - 1);
Q_ASSERT(sibling);
sibling->addChild(child);
QModelIndex newIndex = createIndex(sibling->childCount() - 1, 0, child);
emit dataChanged(newIndex, newIndex);
return newIndex;
}

```

Увеличение уровня, или понижение, означает, что элемент делается сыном находящегося над ним брата. И разумеется, эта операция рекурсивно затрагивает всех потомков элемента. Понижаемый элемент можно было бы поместить в любое место списка потомков его бывшего брата (ставшего отцом), но мы решили всегда помещать его в конец списка. На рис. 4.7 изображено понижение элемента В. Закрашенными кружочками показаны элементы, затронутые этой операцией: элемент А, бывший братом В, становится его родителем, причем В оказывается последним сыном А и исключается из списка потомков своего бывшего родителя Р.

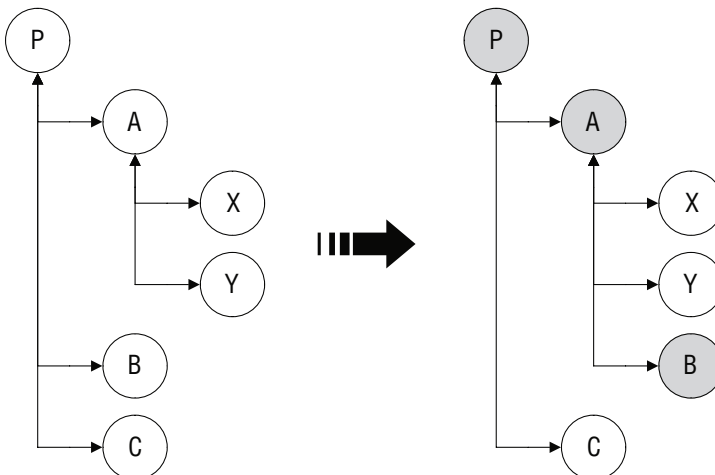


Рис. 4.7. Понижаемый элемент становится последним сыном расположенного над ним брата

Сначала мы получаем элементы, соответствующие модельному индексу понижаемого элемента и его родителя. Если элемент является пер-

вым потомком своего родителя, то выше него братьев нет, так что перемещать его некуда, поэтому мы ничего не делаем и просто возвращаем его модельный индекс. В противном случае отсоединяем элемент от родителя – в этот момент элемент оказывается сиротой и им никто не владеет, поскольку метод `TaskItem::takeChild()` (стр. 179) исключает элемент из списка потомков родителя и обнуляет указатель на родителя.

Так же как в случае повышения, мы с помощью утверждения `Q_ASSERT` проверяем, что отсоединенный от родителя элемент – именно тот, который мы собираемся понизить. Затем мы находим предшествующего брата понижаемого элемента и добавляем отсоединенный элемент в конец списка потомков своего бывшего брата. Метод `TaskItem::addChild()` (стр. 178) изменяет родителя добавляемого элемента, так что после его выполнения элемент снова находится в дереве и имеет родителя.

В заключение, как и в методе `promote()`, мы создаем модельный индекс для пониженного элемента и генерируем сигнал `dataChanged()`, чтобы уведомить ассоциированные представления о том, что модель изменилась. Наконец модельный индекс пониженного элемента возвращается вызывающей программе.

На этом мы заканчиваем рассмотрение методов, добавленных в API класса `QAbstractItemModel` для поддержки перемещения элементов в дереве. При условии, что в классе элемента хранится список элементов-потомков и присутствуют использованные нами методы (`addChild()`, `insertChild()`, `takeChild()` и т. д. – или эквивалентные им), показанная реализация легко адаптируется к другим пользовательским древовидным моделям. Такая возможность повторного использования очень кстати, поскольку, как видно из многочисленных утверждений `Q_ASSERT`, рассыпанных по всему коду, большинство методов, хотя они и выглядят обманчиво простыми, на самом деле не так-то легко написать правильно.

Методы для сохранения и загрузки элементов дерева

В этом разделе мы рассмотрим методы, необходимые для поддержки загрузки и сохранения элементов дерева в файле, в том числе и методы для работы с путями к задачам. Хотя приложение `timelog2` обладает более широкой функциональностью, чем `timelog1` (в частности, поддерживает перетаскивание, вырезание и вставку и перемещение элементов), XML-формат для хранения дерева (стр. 164) ничуть не изменился, так что один и тот же файл можно использовать в обоих приложениях.

```
void TreeModel::clear()
{
    delete rootItem;
    rootItem = 0;
    delete cutItem;
    cutItem = 0;
    timedItem = 0;
    reset();
}
```

Этот метод нужен для поддержки действия `fileNewAction`. Мы должны удалить корневой и вырезанный элементы, потому что отвечаем за них; однако хронометрируемый элемент находится в дереве, так что им уже владеет корень. (Больше мы не будем говорить о хронометрируемом элементе, потому что он важен лишь в контексте хронометража и не имеет ни малейшего отношения к задаче программирования моделей и представлений.) В заключение мы вызываем метод `QAbstractItemModel::reset()`, чтобы уведомить ассоциированные представления о том, что данные модели существенно изменились, поэтому необходимо заново запросить все элементы, которые требуется показать.

```
void TreeModel::save(const QString &filename)
{
    ...
    QDomStreamWriter writer(&file);
    writer.setAutoFormatting(true);
    writer.writeStartDocument();
    writer.writeStartElement("TIMELOG");
    writer.writeAttribute("VERSION", "2.0");
    writeTaskAndChildren(&writer, rootItem);
    writer.writeEndElement(); // TIMELOG
    writer.writeEndDocument();
}
```

Этот метод начинается так же, как `StandardTreeModel::save()`, поэтому повторяющаяся часть кода опущена. Мы используем имя ранее открывавшегося файла или переданное имя и возбуждаем исключение, если не удастся открыть файл (стр. 165). Вся содержательная работа производится в методе `writeTaskAndChildren()`, который мы уже рассматривали, когда занимались реализацией перетаскивания и должны были сериализовывать элемент и его потомков в формате XML (стр. 192).

```
void TreeModel::load(const QString &filename)
{
    ...
    clear();
    rootItem = new TaskItem;
    QDomStreamReader reader(&file);
    readTasks(&reader, rootItem);
    if (reader.hasError())
        throw AQP::Error(reader.errorString());
    reset();
}
```

Этот метод начинается так же, как метод `save()`, в части обработки имени файла и возбуждения исключения, если файл не удастся открыть, поэтому код снова опущен. Очистив имеющиеся задачи, мы создаем новый корневой элемент и с помощью метода `readTasks()` заполняем дерево с известным корнем, читая данные из объекта `QDomStreamReader`. Этот тот

же самый метод `readTasks()`, которым мы пользовались раньше для воссоздания данных, брошенных на некоторый элемент дерева (стр. 194).

```
QStringList TreeModel::pathForIndex(const QModelIndex &index) const
{
    QStringList path;
    QModelIndex thisIndex = index;
    while (thisIndex.isValid()) {
        path.prepend(data(thisIndex).toString());
        thisIndex = thisIndex.parent();
    }
    return path;
}
```

Этот метод возвращает список `QStringList`, представляющий элемент дерева; его логика такая же, как в варианте, написанном для класса `StandardTreeModel`.

Первым делом мы помещаем в список `path` текст переданного элемента, затем добавляем в начало списка текст родителя, потом текст его родителя и т. д., пока не дойдем до верхнего уровня. Не забывайте, что для элемента верхнего уровня модельным индексом родителя является *не* индекс корневого элемента (то есть родителя всех элементов верхнего уровня), а недопустимый модельный индекс.

Этот метод вызывается обработчиком события закрытия главного окна, чтобы запомнить в объекте `QSettings`, какой элемент был текущим в момент выхода из приложения.

```
QModelIndex TreeModel::indexPathForPath(const QStringList &path) const
{
    return indexPathForPath(QModelIndex(), path);
}

QModelIndex TreeModel::indexPathForPath(const QModelIndex &parent,
                                         const QStringList &path) const
{
    if (path.isEmpty())
        return QModelIndex();
    for (int row = 0; row < rowCount(parent); ++row) {
        QModelIndex thisIndex = index(row, 0, parent);
        if (data(thisIndex).toString() == path.at(0)) {
            if (path.count() == 1)
                return thisIndex;
            thisIndex = indexPathForPath(thisIndex, path.mid(1));
            if (thisIndex.isValid())
                return thisIndex;
        }
    }
    return QModelIndex();
}
```

Эти методы противоположны `pathForIndex()`, то есть принимают модельный индекс и путь к задаче, а возвращают модельный индекс элемента-задачи. Открытый (первый) метод получает в качестве аргумента путь к задаче и вызывает закрытый метод, передавая ему в качестве индекса родителя недопустимый модельный индекс (который в архитектуре модель/представление в Qt обозначает корень) и полученный путь к задаче.

Закрытый метод перебирает модельные индексы потомков родителя с указанным индексом и ищет среди них такого потомка, у которого текст совпадает с текстом в первом компоненте пути к задаче. Если такой потомок найден, то метод рекурсивно вызывает себя, передавая модельный индекс потомка в качестве нового родителя и путь к задаче, из которого удален первый компонент (уже сопоставленная строка). В конечном итоге либо все компоненты пути к задаче будут сопоставлены с потомками и метод вернет соответствующий модельный индекс, либо на каком-то шаге не удастся найти потомка, и тогда будет возвращен недопустимый модельный индекс.

В момент запуска приложение `Timelog` с помощью объекта `QSettings` находит в настройках имя последнего загруженного файла и путь к задаче, соответствующей выделенному в последний раз элементу дерева. Если файл еще существует, то после его загрузки программа, зная путь к задаче, находит модельный индекс, делает элемент с этим индексом видимым на экране и выделяет его, то есть дерево оказывается в том же состоянии, в каком было перед выходом из приложения.

На самом деле состояние восстанавливается не полностью, поскольку мы раскрываем лишь те ветви дерева, которые необходимы для того, чтобы показать выделенный элемент, в то время как перед выходом из приложения могли быть раскрыты еще какие-то ветви. В принципе, можно было бы восстановить состояние дерева в полном объеме, но для этого потребовалось бы неоправданно много места в объекте `QSettings` (который на платформе Windows хранится в реестре) в том случае, если было полностью раскрыто много узлов верхнего уровня с большим количеством потомков. Поэтому если такое требование выдвигается, то, наверное, было бы лучше сохранять состояние в самих данных (например, в виде атрибута «`expanded`» или «`visible`») либо завести отдельный файл, потому что размер реестра в Windows ограничен.

Итак, мы полностью завершили рассмотрение древовидных моделей и научились создавать пользовательский подкласс класса `QStandardItemModel` для хранения дерева элементов. Кроме того, мы видели, как можно создать подкласс `QAbstractItemModel`, который предоставляет тот же API, что и любая другая модель, и его расширения для поддержки перемещения элементов в дереве, а также сохранения и загрузки дерева из файла. В следующей главе мы уделим внимание пользовательским делегатам, а в последней из четырех глав, посвященных архитектуре модель/представление, рассмотрим пользовательские представления.

5

Архитектура модель/представление: делегаты

- Редакторы для конкретных типов данных
- Делегаты, ориентированные на конкретный тип данных
- Делегаты для конкретной модели

В этой главе рассматриваются делегаты для моделей и представлений. Предполагается, что читатель знаком с основными принципами архитектуры модель/представление в Qt, изложенными в начале главы 3 (стр. 106).

Все стандартные классы представлений в Qt — `QListView`, `QTableView`, `QColumnView`, `QTreeView` и `QComboBox` — предоставляют делегат `QStyledItemDelegate` для отображения и (в случае редактируемых моделей) редактирования данных, к которым они обращаются¹.

В этой главе мы расскажем, как написать и использовать специализированный делегат, который позволяет точно контролировать внешний вид элементов, отображаемых в представлении, или реализовать собственный редактор для редактируемых элементов, или то и другое вместе. Вообще говоря, существует три подхода к решению этой задачи, и мы рассмотрим каждый из них.

¹ В Qt имеется также класс `QItemDelegate`, который начиная с версии Qt 4.4 заменен классом `QStyledItemDelegate`.

Встроенные в Qt делегаты пользуются определенными виджетами для редактирования элементов разных типов. В первом разделе мы покажем, как изменить виджет, подразумеваемый Qt по умолчанию, другим встроенным или пользовательским виджетом. Но такая операция распространяется на все редактируемые виджеты для соответствующих типов данных во всех представлениях, поэтому из всех решений это оказывается наименее гибким, особенно по сравнению с пользовательскими делегатами.

Во втором разделе мы увидим, как можно создать пользовательский делегат, ориентированный на конкретные типы данных, и задать его для определенных строк или столбцов. Таким делегаты обладают высокой гибкостью и могут повторно использоваться для разных моделей. Кроме того, в отличие от простой подмены виджета редактора, пользовательский делегат позволяет одновременно контролировать отображение и редактирование элементов. Мы рассмотрим два примера. В первом будет использоваться простой делегат, который умеет только отображать дату и время в специальном формате. Второй, более сложный, делегат предназначен для отображения и редактирования форматированного текста, как, например, тот, что использовался в приложениях Timelog из главы 4.

В некоторых ситуациях удобнее не прибегать к ориентированным на определенный тип данных делегатам в строках или столбцах модели, а создать делегат для конкретной модели, который будет применяться ко всем ее элементам. В последнем разделе этой главы мы покажем, как реализуются такие делегаты, взяв в качестве примера делегат для приложения Zipcodes из главы 3.

Редакторы для конкретных типов данных

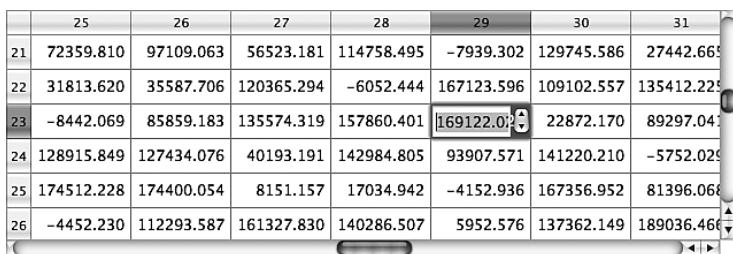
Если нам нужен глобальный редактор, зависящий только от типа элемента, то можно создать объект класса `QItemEditorFactory` и зарегистрировать конкретный виджет, который будет использоваться в качестве редактора для интересующего нас типа данных (или нескольких типов). Например:

```
QItemEditorFactory *editorFactory = new QItemEditorFactory;
QItemEditorCreatorBase *numberEditorCreator = new
    QStandardItemEditorCreator<SpinBox>();
editorFactory->registerEditor(QVariant::Double, numberEditorCreator);
editorFactory->registerEditor(QVariant::Int, numberEditorCreator);
QItemEditorFactory::setDefaultFactory(editorFactory);
```

Здесь мы говорим, что для любого редактируемого элемента со значением типа `double` или `int` в любом представлении, встречающемся в данном приложении, в качестве редактора будет применяться пользовательский виджет-счетчик `SpinBox`.

```
explicit SpinBox(QWidget *parent=0)
    : QDoubleSpinBox(parent)
{
    setRange(-std::numeric_limits<double>::max(),
             std::numeric_limits<double>::max());
    setDecimals(3);
    setAlignment(Qt::AlignVCenter|Qt::AlignRight);
}
```

Мы реализовали только конструктор – задали для счетчика диапазон от наименьшего отрицательного до наибольшего положительного представимого числа типа `double` и определили, что нужно выводить три знака после запятой и выравнивать число по правому краю¹. На рис. 5.1 показано, как выглядит такой редактор. Зарегистрировав этот счетчик, мы добились единообразного отображения и редактирования чисел типа `double` и `int` во всех представлениях, встречающихся в данном приложении.



	25	26	27	28	29	30	31
21	72359.810	97109.063	56523.181	114758.495	-7939.302	129745.586	27442.665
22	31813.620	35587.706	120365.294	-6052.444	167123.596	109102.557	135412.225
23	-8442.069	85859.183	135574.319	157860.401	169122.032	22872.170	89297.041
24	128915.849	127434.076	40193.191	142984.805	93907.571	141220.210	-5752.029
25	174512.228	174400.054	8151.157	17034.942	-4152.936	167356.952	81396.068
26	-4452.230	112293.587	161327.830	140286.507	5952.576	137362.149	189036.466

Рис. 5.1. Зарегистрированный в фабрике класс `SpinBox` в действии

К сожалению, подобные фабрики плохо стыкуются с объектами типа `QStandardItem` для нестроковых данных, по крайней мере без дополнительной доработки. Дело в том, что в объектах `QStandardItem` данные хранятся в виде `QString`, поэтому, когда начинается редактирование, именно `QString` и передается редактору. Но тогда не будет вызываться наш редактор `SpinBox`, потому что он зарегистрирован только для типов `double` и `int`. Решение простое: нужно всегда сохранять данные в `QStandardItem-Model` и извлекать их оттуда, указывая роль `Qt::EditRole`, и использовать для хранения данных подкласс `QStandardItem`. Вот пример такого подкласса для хранения данных типа `double`:

```
class StandardItem : public QStandardItem
{

```

¹ Мы не можем воспользоваться функцией `std::numeric_limits<T>::min()`, потому что для типов с плавающей точкой она возвращает наименьшее представимое значение, большее 0 (очень маленькую дробь), а не наибольшее по абсолютной величине отрицательное значение (как для целых типов).


```

public:
    explicit StandardItem(const double value) : QStandardItem()
    { setData(value, Qt::EditRole); }
    QStandardItem *clone() const
    { return new StandardItem(data(Qt::EditRole).toDouble()); }
    QVariant data(int role=Qt::UserRole+1) const
    {
        if (role == Qt::DisplayRole)
            return QString("%1").arg(QStandardItem::data(Qt::EditRole)
                                     .toDouble(), 0, 'f', 3);
        if (role == Qt::TextAlignmentRole)
            return static_cast<int>(Qt::AlignVCenter|Qt::AlignRight);
        return QStandardItem::data(role);
    }
};

```

В конструкторе мы сохраняем число типа `double` в роли `Qt::EditRole`. Кроме того, мы написали метод `clone()` так, чтобы при создании копии такого элемента создавался именно объект типа `StandardItem`, а не `QStandardItem`.

Метод `QStandardItem::data()` ведет себя *не* так, как метод `QAbstractItemModel::data()`; это обычный метод C++, пользующийся версией из базового класса, то есть для случаев, которые не обрабатываем мы сами, он должен вернуть результат, полученный от метода базового класса, а не недопустимый объект `QVariant`. В данном случае мы должны обработать только роли для отображения и выравнивания текста, чтобы получить нужное форматирование, — для роли `Qt::EditRole` число типа `double` (в составе `QVariant`) правильно возвращает метод базового класса.

Приведенные выше фрагменты кода, демонстрирующие использование фабрики редакторов, взяты из приложения `Number Grid` (`numbergrid`), описанного в главе 7.

Регистрация виджета-редактора для элементов конкретного типа данных — недостаточно избирательное решение, оно распространяется на все приложение в целом. На практике, однако, мы чаще хотим настраивать внешний вид и поведение элементов для каждой модели в отдельности — и этого легко добиться с помощью пользовательских делегатов, как мы увидим в следующих двух разделах.

Делегаты, ориентированные на конкретный тип данных

Если мы создаем много пользовательских моделей и, как правило, задаем для каждой пользовательский делегат, то можем столкнуться с проблемой дублирования кода. Например, пусть с каждой из нескольких моделей ассоциирован пользовательский делегат. Эти делегаты будут неодинаковыми, потому что в разных моделях типы данных в различных столбцах отличаются друг от друга, хотя код обработки некоего

конкретного типа данных – скажем, специального редактора дат для столбца, содержащего даты, – один и тот же.

Один из способов избежать дублирования и повысить степень повторной используемости кода – причем в разных приложениях, а не только в данном, – состоит в том, чтобы создать делегат, ориентированный на конкретный тип данных, и связать его с определенными строками или столбцами. Например, создав делегат для дат, мы можем назначить его для столбца во всех моделях, где имеются столбцы с датами. Это устранил дублирование кода и позволит без труда использовать тот же делегат в новых моделях, содержащих столбцы с датами. И никто не мешает создать другие делегаты, например, для величин углов, цветов, денежных единиц, времени и т. д., избавившись тем самым от дублирования и обеспечив единообразный внешний вид всех наших приложений.

В этом разделе мы рассмотрим два таких делегата. И тот и другой будут связываться со столбцами, но с тем же успехом их можно было бы ассоциировать со строками модели. Первый пример – простой делегат для показа даты и времени в специальном формате, второй – делегат для отображения и редактирования форматированного текста, то есть такого, в котором допускается узкое подмножество HTML-тегов для выделения полужирным шрифтом, курсивом, цветом и т. д. Этим делегатом мы пользовались в приложении *Timelog* из предыдущей главы.

Делегат для строки или столбца, допускающих только чтение

На рис. 5.2 показано приложение *Folder View* (*folderview*) с двумя виджетами *QTreeView*. В правом виджете применен пользовательский делегат *DateTimeDelegate*, который показывает время в виде циферблата часов, а дату – в формате ISO 8601. Данные берутся из модели *QFileSystemModel*, а чтобы снимок экрана поместился на странице, мы скрыли парочку столбцов.

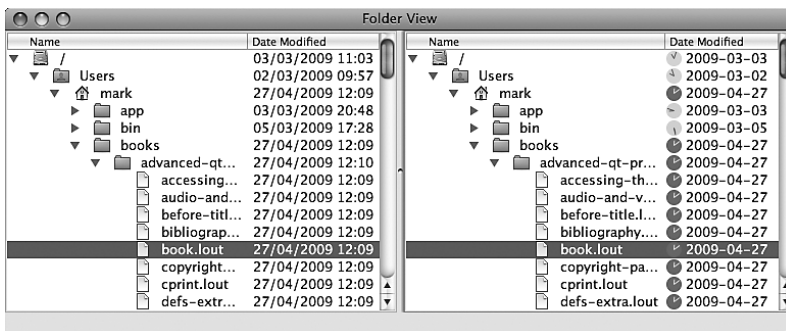


Рис. 5.2. Делегат *DateTimeDelegate* для отображения неизменяемых данных (в правом дереве)

Делегат `DateTimeDelegate` показывает дату последней модификации в виде аналоговых часов, причем время до полудня отображается на светлом фоне, а после полудня – на темном, а если дата модификации раньше текущей, то фон дополнительно осветляется.

Перед тем как переходить к коду самого делегата, вкратце рассмотрим код создания модели, представления и делегата – просто, чтобы иметь какой-то контекст.

```
QFileSystemModel *model = new QFileSystemModel;
model->setRootPath(QDir::homePath());
QModelIndex index = model->index(QDir::homePath());
QTreeView *view = new QTreeView;
view->setItemDelegateForColumn(3, new DateTimeDelegate);
view->setModel(model);
view->setColumnHidden(1, true);
view->setColumnHidden(2, true);
view->scrollTo(index);
view->expand(index);
view->setCurrentIndex(index);
```

Статический метод `QDir::homePath()` возвращает путь к домашнему каталогу пользователя. В классе `QDir` есть и другие подобные методы, например `QDir::currentPath()`, `QDir::rootPath()` и `QDir::tempPath()`, – все они возвращают строки. Имеются также парные методы, возвращающие объекты `QDir` (`QDir::home()` и т. д.).

Модель `QFileSystemModel` допускает редактирование, поэтому ее можно положить в основу файлового менеджера. Но в данном случае мы просто воспользовались ею как источником данных для представления `QTreeView`, с четвертым столбцом которого связали делегат `DateTimeDelegate`, тогда как для прочих столбцов оставили встроенный в дерево делегат `QStyledItemDelegate`. Вызов `QFileSystemModel::setRootPath()` не устанавливает текущий выделенный элемент (это делает метод `setCurrentIndex()`), а просто задает каталог, за которым будет следить объект `QFileSystemWatcher`. Любые изменения, происходящие с файлами и подкаталогами, которые находятся в отслеживаемом каталоге (например, файл удален или обновлен), отражаются в модели, а затем во всех ассоциированных с ней представлениях.

В классе `DateTimeDelegate` необходимо реализовать только конструктор и метод `paint()`, так как мы собираемся изменить лишь внешний вид обрабатываемых этим делегатом элементов, а не их поведение.

```
class DateTimeDelegate : public QStyledItemDelegate
{
    Q_OBJECT
public:
    explicit DateTimeDelegate(QObject *parent=0)
        : QStyledItemDelegate(parent) {}
    void paint(QPainter *painter, const QStyleOptionViewItem
        &option, const QModelIndex &index) const;
```

```
private:
    ...
};
```

Конструктор просто перепоручает всю работу конструктору базового класса. А вот метод `paint()` и закрытые вспомогательные методы, к которым он обращается, мы рассмотрим, чтобы понять, как производится отображение элемента.

```
void DateTimeDelegate::paint(QPainter *painter,
    const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    const QFileSystemModel *model =
        qobject_cast<const QFileSystemModel*>(index.model());
    Q_ASSERT(model);
    const QDateTime &lastModified = model->lastModified(index);
    painter->save();
    painter->setRenderHints(QPainter::Antialiasing|
        QPainter::TextAntialiasing);

    if (option.state & QStyle::State_Selected)
        painter->fillRect(option.rect, option.palette.highlight());
    const qreal diameter = qMin(option.rect.width(), option.rect.height());
    const QRectF rect = clockRect(option.rect, diameter);
    drawClockFace(painter, rect, lastModified);
    drawClockHand(painter, rect.center(), diameter / 3.5,
        ((lastModified.time().hour() +
            (lastModified.time().minute() / 60.0))) * 30);
    drawClockHand(painter, rect.center(), diameter / 2.5,
        lastModified.time().minute() * 6);
    drawDate(painter, option, diameter, lastModified);
    painter->restore();
}
```

Вначале мы получаем дату и время последней модификации для того элемента – файла, каталога или иного объекта файловой системы, – который собираемся отображать. Затем мы сохраняем состояние объекта `painter` и включаем режим сглаживания (антиалиасинг)¹.

Если элемент выделен, то мы закрашиваем фон соответствующим цветом, выбранным из текущей палитры. Затем вычисляется диаметр циферблата, который понадобится позже для различных вычислений, а также прямоугольник, описанный вокруг окружности циферблата.

Все подготовив, мы рисуем циферблат, минутную и часовую стрелку, затем дату (в текстовом виде) и, наконец, восстанавливаем объект

¹ Во время написания этой книги в документации по Qt ничего не говорилось о том, какие флаги отображения установлены по умолчанию, поэтому на всякий случай мы явно указываем те, что нужны нам.

`painter` в том состоянии, в котором он находился перед началом рисования, чтобы его можно было снова использовать для рисования следующего элемента.

```
QRectF DateTimeDelegate::clockRect(const QRectF &rect,
                                   const qreal &diameter) const
{
    QRectF rectangle(rect);
    rectangle.setWidth(diameter);
    rectangle.setHeight(diameter);
    return rectangle.adjusted(1.5, 1.5, -1.5, -1.5);
}
```

Этот метод возвращает квадрат со стороной, чуть меньшей диаметра циферблата, и слегка смещенный относительно прямоугольника данного элемента, чтобы при отображении образовались небольшие поля.

Метод `QRect::adjusted()` (и его аналог в классе `QRectF`) возвращает новый прямоугольник, в котором координаты левого верхнего и правого нижнего угла смещены на указанные величины. В данном случае левый верхний угол сдвинут вправо вниз (поскольку ось *Y* направлена вниз) на 1,5 пиксела, а правый нижний угол – тоже на 1,5 пиксела, но влево вверх.

```
void DateTimeDelegate::drawClockFace(QPainter *painter,
                                     const QRectF &rect, const QDateTime &lastModified) const
{
    const int Light = 120;
    const int Dark = 220;
    int shade = lastModified.date() == QDate::currentDate() ? Light : Dark;
    QColor background(shade, shade,
                     lastModified.time().hour() < 12 ? 255 : 175);
    painter->setPen(background);
    painter->setBrush(background);
    painter->drawEllipse(rect);
    shade = shade == Light ? Dark : Light;
    painter->setPen(QColor(shade, shade,
                     lastModified.time().hour() < 12 ? 175 : 255));
}
```

Этот метод рисует циферблат – эллипс оказывается окружностью, если описанный прямоугольник – квадрат, как в данном случае. Большая часть кода касается раскрашивания с использованием цветовой схемы RGB (красный, зеленый, синий); значения отдельных компонентов цвета должны находиться в диапазоне от 0 до 255, в нашем примере цвет зависит от того, находится ли время в первой или второй половине суток, а также от того, является ли дата сегодняшним днем или более ранним. В конце мы устанавливаем цвет пера, которым будем рисовать стрелки часов, поскольку этот цвет зависит от цвета циферблата (для обеспечения контраста).

```
void DateTimeDelegate::drawClockHand(QPainter *painter,
    const QPointF &center, const qreal &length,
    const qreal &degrees) const
{
    const qreal angle = AQP::radiansFromDegrees(
        (qRound(degrees) % 360) - 90);
    const qreal x = length * std::cos(angle);
    const qreal y = length * std::sin(angle);
    painter->drawLine(center, center + QPointF(x, y));
}
```

Этот метод вызывается дважды — для рисования минутной и часовой стрелки. В аргументе `center` передается центр циферблата, в аргументе `length` — доля диаметра (часовая стрелка короче, минутная — длиннее), а в аргументе `degrees` — угол, пропорциональный времени, указываемому стрелкой. Мы приводим угол к диапазону от 0 до 360° и вычитаем 90°, чтобы значение 0° соответствовало не востоку, как принято в геометрии, а северу — как в аналоговых часах. Затем мы вычисляем координаты конца стрелки и проводим прямую линию из центра в конечную точку. (Функция `qRound()` описана в табл. 1.2 на стр. 62, а функция `AQP::radiansFromDegrees()` имеется в модуле `aqp.{hpp,cpp}` из кода, прилагаемого к книге.)

```
void DateTimeDelegate::drawDate(QPainter *painter,
    const QStyleOptionViewItem &option, const qreal &diameter,
    const QDateTime &lastModified) const
{
    painter->setPen(option.state & QStyle::State_Selected
        ? option.palette.highlightedText().color()
        : option.palette.windowText().color());
    QString text = lastModified.date().toString(Qt::ISODate);
    painter->drawText(option.rect.adjusted(
        qRound(diameter * 1.2), 0, 0, 0), text,
        QTextOption(Qt::AlignVCenter|Qt::AlignLeft));
}
```

Дата выводится цветом текста в окне — или цветом выделенного текста, если данный элемент выделен. А в качестве прямоугольника берется прямоугольник элемента, сдвинутый по оси X, чтобы освободить место для циферблата и оставить небольшой зазор.

Класс `QTextOption` служит для хранения метода выравнивания, режима переноса строк, позиций табуляции и различных флагов форматирования произвольного форматированного текста. Чаще всего он используется при рисовании текста для задания выравнивания, а в случае многострочных текстов — для задания режима переноса строк по словам.

Итак, мы видели, что отображение в специальном формате даты и времени, равно как и других элементов данных, легко осуществляется путем создания пользовательского делегата с переопределением его метода `paint()`. Как и для большинства задач рисования в Qt, работа

сводится в основном к выбору подходящих цветов и выполнению математических вычислений, связанных с установкой размера и позиционированием. Но когда речь идет о редактируемых элементах, иногда приходится создавать собственные виджеты-редакторы, и в этом случае нужно переопределять больше методов. Как это делается, мы увидим в следующем подразделе.

Делегат для редактируемой строки или столбца

Пользовательские делегаты можно применять как для отображения, так и для редактирования элементов. Если в случае отображения мы должны переопределить только метод `paint()`, то для поддержки редактирования необходимо также реализовать и некоторые методы из API класса `QStyledItemDelegate`, перечисленные в табл. 5.1, – как минимум `createEditor()`, `setEditorData()` и `setModelData()`.

Таблица 5.1. API класса *QStyledItemDelegate*

Метод	Описание
<code>createEditor</code> (<i>parent</i> , <i>styleOption</i> , <i>index</i>)	Возвращает виджет, пригодный для редактирования элемент с модельным индексом <i>index</i>
<code>paint</code> (<i>painter</i> , <i>styleOption</i> , <i>index</i>)	Рисует элемент с модельным индексом <i>index</i> . Редко переопределяется для простых данных – текста, дат и чисел
<code>setEditorData</code> (<i>editor</i> , <i>index</i>)	Передаёт редактору <i>editor</i> данные из элемента модели с индексом <i>index</i>
<code>setModelData</code> (<i>editor</i> , <i>model</i> , <i>index</i>)	Записывает в элемент модели с индексом <i>index</i> данные из редактора <i>editor</i>
<code>sizeHint</code> (<i>styleOption</i> , <i>index</i>)	Возвращает размер, необходимый делегату для отображения или редактирования элемента с модельным индексом <i>index</i>
<code>updateEditorGeometry</code> (<i>editor</i> , <i>styleOption</i> , <i>index</i>)	Устанавливает размер и положение редактора <i>editor</i> для редактирования на месте – переопределяется редко

Реализации методов `sizeHint()` и `updateEditorGeometry()` в базовом классе почти всегда достаточно, так что переопределять их приходится редко. Кроме того, часто можно обойтись без переопределения метода `paint()`, особенно если данные представляют собой простой текст, дату, время или число.

В приложениях *Timelog*, которые мы рассматривали в главе 4, использовался форматированный текст – узкое подмножество HTML-тегов для поддержки простейших шрифтовых выделений: полужирным шрифтом, курсивом, цветом. Кроме того, элементы, представленные в виде форматированного текста, содержали флажок. Чтобы поддержать все

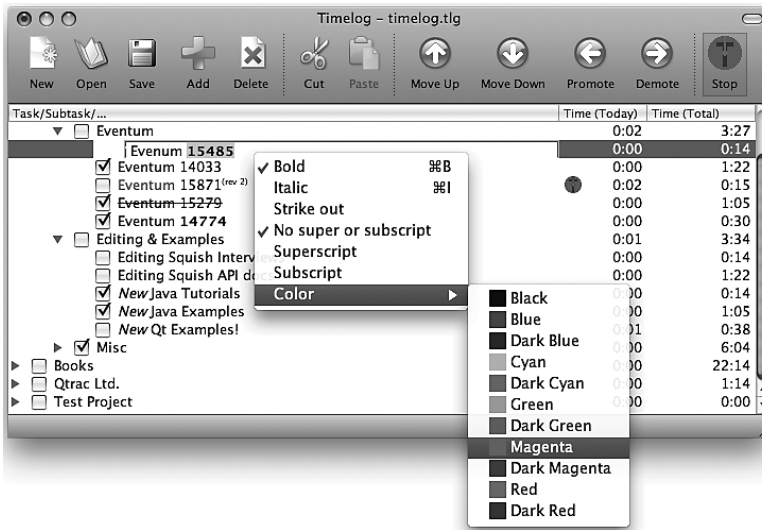


Рис. 5.3. Делегат *RichTextDelegate* в действии

это, мы создали делегат *RichTextDelegate*, контекстное меню которого показано на рис. 5.3.

Класс *RichTextDelegate* обеспечивает и отображение, и редактирование. В нем реализованы почти все методы базового класса *QStyledItemDelegate*, точнее: *paint()*, *sizeHint()*, *createEditor()*, *setEditorData()* и *setModelData()*. Помимо этого, в нем имеется закрытый слот *closeAndCommitEditor()* и две закрытых переменных-члена – указатели на *QCheckBox* и *QLabel*. Мы рассмотрим все эти методы, начиная с конструктора.

```
RichTextDelegate::RichTextDelegate(QObject *parent)
    : QStyledItemDelegate(parent)
{
    checkbox = new QCheckBox;
    checkbox->setFixedSize(
        qRound(1.3 * checkbox->sizeHint().height()),
        checkbox->sizeHint().height());
    label = new QLabel;
    label->setTextFormat(Qt::RichText);
    label->setWordWrap(false);
}
```

Существует три подхода к отображению элементов в пользовательских делегатах. Во-первых, можно рисовать все вообще самостоятельно – так мы поступали при реализации делегата *DateTime* (стр. 211); недостаток такого подхода в том, что нам же и приходится учитывать особенности разных платформ. Во-вторых, можно воспользоваться методами класса *QStyle*, например *QStyle::drawControl()*, *QStyle::drawComplexControl()* и др.

Это мощный, но низкоуровневый подход, который требует аккуратности и написания большого объема кода. Мы же пойдем по более простому и самому высокоуровневому пути: будем рисовать виджеты, в данном случае флажок и метку, оставляя Qt заботы о межплатформенных различиях и стремясь сохранить чистоту и простоту своего кода.

Если бы мы хотели вывести лишь простой текст, то хватило бы виджета `QCheckBox`, потому что он рисует как флажок, так и ассоциированный с ним текст. Но поскольку нам нужно выводить форматированный текст, то этот виджет будет использоваться только для рисования самого флажка (мы не будем передавать его конструктору текст), а для показа форматированного текста применим метку `QLabel`.

Мы отводим под флажок чуть больше места, чем ему реально необходимо, чтобы оставить промежуток между флажком и меткой. А метке назначаем, чтобы она интерпретировала получаемый текст как форматированный (HTML) и не разбивала его на строки.

```
~RichTextDelegate() { delete checkbox; delete label; }
```

Вместе с делегатом мы должны уничтожить принадлежащие ему флажок и метку — это тривиально и делается во встраиваемом деструкторе прямо в заголовочном файле.

Для простоты объяснения рассмотрим метод `paint()` в четыре приема.

```
void RichTextDelegate::paint(QPainter *painter,
    const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    bool selected = option.state & QStyle::State_Selected;
    QPalette palette(option.palette);
    palette.setColor(QPalette::Active, QPalette::Window,
        selected ? option.palette.highlight().color()
                : option.palette.base().color());
    palette.setColor(QPalette::Active, QPalette::WindowText,
        selected
        ? option.palette.highlightedText().color()
        : option.palette.text().color());
```

Первым делом мы создаем новую палитру. За основу берем палитру из параметра `option` (типа `QStyleOptionViewItem`), а затем задаем для нее цвета фона `QPalette::Window` и текста `QPalette::WindowText`, взяв их из параметра `option` и приняв во внимание, выделен элемент или нет. (Мы решили в качестве цвета фона задать `QPalette::base()`, а не `QPalette::window()`, потому что это дает лучшие результаты на тех платформах, где мы тестировали программу.)

```
int yOffset = checkbox->height() < option.rect.height()
    ? (option.rect.height() - checkbox->height()) / 2 : 0;
QRect checkboxRect(option.rect.x(), option.rect.y() + yOffset,
    checkbox->width(), checkbox->height());
```

```
checkbox->setPalette(palette);
bool checked = index.model()->data(index, Qt::CheckStateRole)
    .toInt() == Qt::Checked;
checkbox->setChecked(checked);
```

Сначала настраиваем флажок. Создаем прямоугольник `checkboxRect`, который понадобится позже, во время рисования флажка. Если прямоугольник, заданный в параметре `option`, выше флажка, то создаваемый прямоугольник центрируется по вертикали в отведенном для него пространстве. Затем в качестве палитры прямоугольника задается только что созданная палитра, а состояние флажка устанавливается в соответствии с состоянием элемента.

```
QRect labelRect(option.rect.x() + checkbox->width(),
    option.rect.y(), option.rect.width() - checkbox->width(),
    option.rect.height());
label->setPalette(palette);
label->setFixedSize(qMax(0, labelRect.width()), labelRect.height());
QString html = index.model()->data(index, Qt::DisplayRole).toString();
label->setText(html);
```

Для флажка в конструкторе задан фиксированный размер, и мы его никогда не изменяем, но размер метки (хотя он тоже фиксирован) задаваться должен для каждого элемента в отдельности. Создаваемый для метки `labelRect` прямоугольник тоже основан на прямоугольнике из параметра `option`, но сдвинут вправо (его ширина уменьшена на величину сдвига), чтобы освободить место для флажка. При таком сужении ширина может оказаться отрицательной – например, если пользователь сделал окно очень узким, – поэтому мы корректируем результат с помощью функции `qMax()`. Задав палитру и размер метки, мы получаем текст элемента (в формате HTML) и копируем его в текст метки.

Теперь у флажка и у метки правильно установлены палитра, размер и содержимое, и у нас есть прямоугольники, в которых можно рисовать.

```
QString checkboxKey = QString("CHECKBOX:%1.%2").arg(selected)
    .arg(checked);
paintWidget(painter, checkboxRect, checkboxKey, checkbox);
QString labelKey = QString("LABEL:%1.%2.%3x%4").arg(selected)
    .arg(html).arg(labelRect.width()).arg(labelRect.height());
paintWidget(painter, labelRect, labelKey, label);
}
```

Мы вынесли весь код рисования виджетов во вспомогательный закрытый метод `paintWidget()`. Кроме того, мы воспользовались глобальным кэшем `QPixmapCache`, чтобы не рисовать одинаковые растры снова и снова. В кэше каждый растр идентифицируется строкой, в качестве которой мы выбрали конкатенацию состояния и содержимого (состояние отмеченности флажка и текст метки). Таким образом, для флажков в кэше будет не более четырех растров: (выделен, не отмечен), (выделен,

отмечен), (не выделен, не отмечен) и (не выделен, отмечен). Сформировав строку-ключ кэша, мы вызываем метод `paintWidget()`.

```
void RichTextDelegate::paintWidget(QPainter *painter,
    const QRect &rect, const QString &cacheKey,
    QWidget *widget) const
{
    QPixmap pixmap(widget->size());
    if (!QPixmapCache::find(cacheKey, &pixmap)) {
        widget->render(&pixmap);
        QPixmapCache::insert(cacheKey, pixmap);
    }
    painter->drawPixmap(rect, pixmap);
}
```

Сначала создается пустой растр нужного размера. Затем с помощью метода `QPixmapCache::find()` мы ищем в кэше растр с данным ключом. Если такой ключ существует, то метод возвращает `true` и копирует в объект `QPixmap`, переданный по указателю – или неконстантной ссылке (то есть без знака `&`) для версии Qt 4.5 и более ранних, – иначе `false`. В первый раз, когда мы запрашиваем нужный растр, он отсутствует в кэше, поэтому мы рисуем переданный виджет в пустом растре и помещаем растр в кэш. В самом конце растр рисуется в переданном прямоугольнике. (Другой способ получить растр виджета – вызывать метод `QPixmap::grabWidget()`, передав ему виджет в качестве параметра.)

Основное достоинство этого подхода в том, что почти вся работа по рисованию и применению стилей перепоручается Qt, что делает наш код гораздо проще, чем при других вариантах. И эффективнее – за счет использования кэша растров.

```
QSize RichTextDelegate::sizeHint(const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    QString html = index.model()->data(index, Qt::DisplayRole).toString();
    document.setDefaultFont(option.font);
    document.setHtml(html);
    return QSize(document.idealWidth(), option.fontMetrics.height());
}
```

В большинстве случаев переопределять метод `QStyledItemDelegate::sizeHint()` нет необходимости, но наша ситуация необычна. Предположим, к примеру, что есть такой `QTextDocument`- и `QTextEditHTML`-текст: `bold blue bear`. Он содержит 50 символов, но отображается только 18. Стандартная реализация `sizeHint()` вычислила бы ширину, необходимую для размещения всех 50 символов, поэтому мы должны переопределить ее для получения более точного результата.

Самый очевидный способ вычислить ширину состоит в том, чтобы преобразовать разметку в обычный текст и вызвать для него `QFontMetrics::width()`. К сожалению, при этом не учитываются такие тонкости, как наличие надстрочных и подстрочных индексов, почти всегда

отображаемых более мелким шрифтом, или полужирных и курсивных символов, которые обычно шире нормальных, а также возможность смены шрифтов. Но выход есть – точные вычисления с учетом вышеупомянутых и ряда других деталей производит метод `QTextDocument::idealWidth()`, которым мы и воспользовались.

На некоторых платформах создание и уничтожение объекта `QTextDocument` для каждого вызова `sizeHint` обходится довольно дорого, поэтому в закрытой секции класса мы объявили изменяемый член `QTextDocument document`, который можем использовать повторно.

```
QWidget *RichTextDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem &option, const QModelIndex&) const
{
    RichTextLineEdit *editor = new RichTextLineEdit(parent);
    editor->viewport()->setFixedHeight(option.rect.height());
    connect(editor, SIGNAL(returnPressed()),
        this, SLOT(closeAndCommitEditor()));
    return editor;
}
```

Этот метод создает подходящий редактор для элемента с переданным модельным индексом. Поскольку наш делегат ассоциируется со столбцом, в котором хранится форматированный текст, нам не нужно знать, для какого конкретно элемента понадобился редактор. (Позже мы покажем другую реализацию этого метода, в которой тип редактора зависит от некоторого аспекта элемента, например, от того, в каком столбце элемент находится, см. стр. 223.)

Мы создали виджет `RichTextLineEdit` (рассматривается в главе 9, стр. 373) для редактирования HTML-данных, с которыми имеет дело данный делегат. В данном случае *существенно*, чтобы высота области просмотра редактора была фиксирована – равна высоте прямоугольника из параметра `option`, – тогда виджет `RichTextLineEdit` не станет прыгать вверх-вниз в процессе ввода текста. (Это происходит потому, что, хотя класс `RichTextLineEdit` рассчитан на редактирование одной строки текста, он является производным от класса `QTextEdit`, который предназначен для редактирования многострочных текстов.)

Нажатие пользователем клавиши `Return` (или `Enter`) воспринимается как конец редактирования, поэтому мы соединяем сигнал `returnPressed()` от `RichTextLineEdit` (название выбрано по аналогии с одноименным сигналом в классе `QLineEdit`) с закрытым слотом `closeAndCommitEditor()`, который мы рассмотрим чуть ниже.

```
void RichTextDelegate::setEditorData(QWidget *editor,
    const QModelIndex &index) const
{
    QString html = index.model()->data(index, Qt::DisplayRole).toString();
    RichTextLineEdit *lineEdit = qobject_cast<RichTextLineEdit*>(editor);
    Q_ASSERT(lineEdit);
```

```

        lineEdit->setHtml(html);
    }

```

Создав редактор, делегат вызывает метод `setEditorData()`, чтобы передать в него данные из модели. Мы извлекаем текст (в формате HTML), получаем указатель на объект `RichTextLineEdit`, созданный в методе `createEditor()`, и копируем текст элемента в текст редактора.

```

void RichTextDelegate::closeAndCommitEditor()
{
    RichTextLineEdit *lineEdit = qobject_cast<RichTextLineEdit*>(sender());
    Q_ASSERT(lineEdit);
    emit commitData(lineEdit);
    emit closeEditor(lineEdit);
}

```

Реализация подобного слота часто бывает полезна в редакторах, которые генерируют сигнал, означающий успешное завершение редактирования, такой, например, как сигнал `returnPressed()` в классе `QLineEdit`.

Мы применяем функцию `qobject_cast<>()` в сочетании с методом `QObject::sender()`, чтобы получить указатель на `RichTextLineEdit`, а затем генерируем два сигнала: первый сообщает делегату о том, что нужно сохранить данные редактора, то есть скопировать их в модель; второй, — что делегат должен закрыть редактор, поскольку тот больше не нужен.

```

void RichTextDelegate::setModelData(QWidget *editor,
    QAbstractItemModel *model, const QModelIndex &index) const
{
    RichTextLineEdit *lineEdit = qobject_cast<RichTextLineEdit*>(editor);
    Q_ASSERT(lineEdit);
    model->setData(index, lineEdit->toSimpleHtml());
}

```

Этот метод вызывается, когда пользователь обозначает конец редактирования, — щелкнув мышью вне редактора, нажав клавишу `Tab` для перехода к другому элементу или `Return` либо `Enter`. (Пользователь может и отменить редактирование клавишей `Esc`.) Здесь мы получаем указатель на объект `RichTextLineEdit` и копируем HTML-код из редактора в текст элемента модели. Как мы увидим в главе 9 (стр. 373), метод `toSimpleHtml()` порождает более простой и компактный HTML-код, чем, скажем, метод `QTextEdit::toHtml()`, правда, за счет того, что умеет обрабатывать только очень ограниченное подмножество HTML.

В методах `setEditorData()`, `closeAndCommitEditor()` и `setModelData()` мы вставили утверждение `Q_ASSERT()`, проверяющее успешность приведения `qobject_cast<>()`. Обычно мы используем утверждения, когда согласно логике приложения некое условие *должно* быть истинным в некоторой точке (то есть его несоблюдение свидетельствует о логической ошибке), а в противном случае прибегаем к оператору `if`. (В этой книге вы встретите многочисленные примеры того и другого рода.)

На этом мы завершаем реализацию класса `RichTextDelegate`. Если не считать метода `paint()`, все остальные методы достаточно просты. Это прямое следствие того факта, что делегат ориентирован на данные конкретного типа – в нашем случае форматированный текст – и ассоциируется с одним столбцом (или строкой), поэтому ему не приходится анализировать строку или столбец, в которой находится переданный элемент, чтобы понять, с каким типом данных ему предстоит иметь дело; все элементы обрабатываются единообразно.

Делегаты для конкретной модели

Если количество создаваемых моделей невелико, то может оказаться удобнее писать ориентированные на конкретную модель делегаты по мере необходимости, а не разрабатывать набор более общих делегатов, ассоциируемых со строкой или столбцом определенного типа. В этом разделе мы рассмотрим типичный пример делегата для конкретной модели; он использовался в приложениях из главы 3, а его вид в процессе редактирования показан на рис. 3.3 (стр. 109).

Мы назвали класс этого пользовательского делегата `ItemDelegate` и сделали его производным от `QStyledItemDelegate`. Конструктор (не показан) просто передает аргумент `parent` конструктору базового класса, его тело пусто. Из методов класса `QStyledItemDelegate` (стр. 217) мы переопределили `paint()` (только для вывода почтовых индексов, для остальных столбцов достаточно версии из базового класса) и три метода редактирования данных: `createEditor()`, `setEditorData()` и `setModelData()`. Переопределять остальные методы нет нужды, и эта ситуация достаточно типична. Как обычно, рассмотрим все методы поочередно.

```
void ItemDelegate::paint(QPainter *painter,
                        const QStyleOptionViewItem &option,
                        const QModelIndex &index) const
{
    if (index.column() == Zipcode) {
        QStyleOptionViewItemV4 opt(option);
        initStyleOption(&opt, index);
        QString text = QString("%1").arg(opt.text.toInt(),
                                         5, 10, QChar('0'));

        painter->save();
        painter->setRenderHints(QPainter::Antialiasing|
                               QPainter::TextAntialiasing);
        if (opt.state & QStyle::State_Selected) {
            painter->fillRect(opt.rect, opt.palette.highlight());
            painter->setPen(opt.palette.highlightedText().color());
        }
        else
            painter->setPen(opt.palette.windowText().color());
        painter->drawText(opt.rect.adjusted(0, 0, -3, 0), text,
                         QTextOption(Qt::AlignVCenter|Qt::AlignRight));
    }
```

```
        painter->restore();
    }
    else
        QStyledItemDelegate::paint(painter, option, index);
}
```

Мы решили рисовать почтовые индексы самостоятельно, но отображение прочих данных оставили базовому классу. Обратите внимание, что в этом и в последующих методах используется метод `index.column()` для определения столбца (и, следовательно, типа), в котором находятся обрабатываемые данные, — для делегатов, рассмотренных в предыдущем разделе, это было не нужно, потому что заранее было известно, с каким столбцом или строкой они ассоциированы.

Класс `QStyleOptionViewItem`, впервые введенный в версии Qt 4.0, получил развитие в последующих версиях серии Qt 4.x. Так появились классы `QStyleOptionViewItemV2`, `QStyleOptionViewItemV3` и `QStyleOptionViewItemV4`, в каждый из которых добавлены новые открытые члены. Вообще говоря, вполне можно пользоваться и первоначальной версией `QStyleOptionViewItem`, но в некоторых ситуациях удобнее работать с более поздними, так как в них имеются полезные дополнительные члены.

Правильный способ получить объект класса `QStyleOptionViewItem` более поздней версии показан в приведенном выше фрагменте: нужно создать объект версии `QStyleOptionViewItemV4` (или той, что нам необходима), передав его конструктору полученный от вызывающей программы параметр `option`, а затем вызвать метод `QStyledItemDelegate::initStyleOption()`, передав ему ссылку на созданный объект и модельный индекс того элемента, с которым мы работаем.

Класс `QStyleOptionViewItemV4` позволяет нам получить доступ к своему члену `text` (он содержит текст элемента с заданным модельным индексом) непосредственно, вместо того чтобы писать `index.model()->data(index).toString()` (хотя в данном примере надо было бы воспользоваться методом `toInt()`).

Получив почтовый индекс, мы преобразуем его в целое число и затем создаем строку с его представлением в желаемом формате — в данном случае ровно пять цифр, возможно, дополненных слева нулями.

Перед началом рисования мы сохраняем состояние объекта-рисовальщика `painter`. Это следует делать обязательно, если делегат изменяет состояние рисовальщика, поскольку в целях эффективности один и тот же рисовальщик используется для рисования всех элементов представления. Мы включили сглаживание, поскольку, как отмечалось в сноске выше, неизвестно, какие флаги установлены по умолчанию, поэтому безопаснее всегда явно указывать нужные режимы. Мы закрашиваем фон цветом выделения, если данный элемент выделен, и выбираем соответствующий цвет пера (для рисования текста). Затем текст рисуется в переданном прямоугольнике и выравнивается по правому краю; мы оставляем поле величиной 3 пиксела (это достигается небольшим

уменьшением ширины прямоугольника), чтобы текст не наползал на контур ячейки. В конце восстанавливается исходное состояние объекта painter, после чего его можно использовать для рисования следующего элемента.

```
QWidget *ItemDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    static QStringList usStates;
    if (usStates.isEmpty())
        usStates << "(Unknown)" << "Alabama" << "Alaska"
        ...
        << "West Virginia" << "Wisconsin" << "Wyoming";
    if (index.column() == Zipcode)
        return new ZipcodeSpinBox(parent);
    if (index.column() == State) {
        QComboBox *editor = new QComboBox(parent);
        editor->addItem(usStates);
        return editor;
    }
    return QStyledItemDelegate::createEditor(parent, option, index);
}
```

Для редактирования большей части данных мы пользуемся стандартными виджетами Qt, а для почтовых индексов применяем специализированный счетчик. Метод createEditor() должен сначала определить, какой нужен редактор (в данной модели это зависит от столбца), затем создать его, сконфигурировать и вернуть в состоянии, готовом для приема данных из редактируемого элемента и показа пользователю. Отметим, что, создавая редактор самостоятельно, мы должны передать ему того родителя, которого получили от вызывающей программы, — это гарантирует, что Qt примет на себя бремя владения редактором и в нужный момент удалит его.

Для редактирования колонки «Штат» мы используем комбинированный список, помещая в него строку «(Unknown)» (Неизвестно) и названия всех штатов США. Названия штатов хранятся в статическом списке QStringList, который, следовательно, создается только один раз.

Редактирование почтовых отделений и округов (представленных обычным текстом) мы поручаем базовому классу, который возвращает виджет QLineEdit, — если только мы не назначили другой редактор с помощью фабрики QItemEditorFactory, как обсуждалось выше, на стр. 209.

```
void ItemDelegate::setEditorData(QWidget *editor,
    const QModelIndex &index) const
{
    if (index.column() == Zipcode) {
        int value = index.model()->data(index).toInt();
        ZipcodeSpinBox *spinBox = qobject_cast<ZipcodeSpinBox*>(editor);
        Q_ASSERT(spinBox);
    }
```



```

        spinBox->setValue(value);
    }
    else if (index.column() == State) {
        QString state = index.model()->data(index).toString();
        QComboBox *comboBox = qobject_cast<QComboBox*>(editor);
        Q_ASSERT(comboBox);
        comboBox->setCurrentIndex(comboBox->findText(state));
    }
    else
        QStyledItemDelegate::setEditorData(editor, index);
}

```

После того как редактор создан, Qt вызывает метод `setEditorData()`, давая нам возможность заполнить его данными перед тем, как показывать пользователю. Мы всегда должны делать это для виджетов, которые создали сами, препоручая базовому классу заботу о тех виджетах, за которые отвечает он.

Логика в этой ситуации почти всегда одинакова: найти элемент с заданным модельным индексом, привести указатель на `QWidget` к типу конкретного редактора и заполнить этот редактор данными. В случае комбинированного списка мы поступили несколько иначе, потому что список уже содержит все данные и нам остается только сделать в нем текущей ту строку, которая соответствует тексту выделенного элемента модели.

```

void ItemDelegate::setModelData(QWidget *editor,
    QAbstractItemModel *model, const QModelIndex &index) const
{
    if (index.column() == Zipcode) {
        ZipcodeSpinBox *spinBox = qobject_cast<ZipcodeSpinBox*>(editor);
        Q_ASSERT(spinBox);
        spinBox->interpretText();
        model->setData(index, spinBox->value());
    }
    else if (index.column() == State) {
        QComboBox *comboBox = qobject_cast<QComboBox*>(editor);
        Q_ASSERT(comboBox);
        model->setData(index, comboBox->currentText());
    }
    else
        QStyledItemDelegate::setModelData(editor, model, index);
}

```

Если пользователь подтверждает вновь введенное значение, то в случае виджета, за который отвечаем мы сами, необходимо получить новое значение от редактора и записать его в элемент с заданным модельным индексом. Для прочих виджетов мы просто перепоручаем работу базовому классу.

Для счетчика мы решили перестраховаться, поэтому вызываем метод `interpretText()` на случай, если пользователь вводил или удалял цифры, а не пользовался стрелками счетчика. Для комбинированного списка достаточно просто извлечь текущее значение.

Для ясности приведен полный код класса `ZipcodeSpinBox`:

```
class ZipcodeSpinBox : public QSpinBox
{
    Q_OBJECT

public:
    explicit ZipcodeSpinBox(QWidget *parent) : QSpinBox(parent)
    {
        setRange(MinZipcode, MaxZipcode);
        setAlignment(Qt::AlignVCenter|Qt::AlignRight);
    }

protected:
    QString textFromValue(int value) const
    { return QString("%1").arg(value, 5, 10, QChar('0')); }
};
```

Если бы мы хотели только задать диапазон значений счетчика и выравнивание, то могли бы воспользоваться стандартным классом `QSpinBox` и сконфигурировать объект в методе `createEditor()`. Но мы решили сделать это в конструкторе, потому что от создания подкласса `QSpinBox` все равно никуда не деться, так как необходимо переопределить метод `textFromValue()`. Это нужно для того, чтобы в текстовом представлении почтового индекса было ровно пять цифр, дополненных слева нулями, – так же, как при рисовании почтовых кодов в интерфейсе.

На этом мы завершаем рассмотрение типичного пользовательского делегата, ориентированного на конкретную модель. Такие делегаты менее гибки по сравнению с универсальными делегатами, ассоциируемыми со строкой или столбцом и рассчитанными на конкретный тип, зато весь код делегатов находится в одном месте и позволяет точно контролировать внешний вид и редактирование элементов модели. Кроме того, было показано, что значительную часть работы можно перепоручать базовому классу.

Пользовательские делегаты – самый распространенный и удобный способ контроля над представлением и редактированием элементов модели. Но если мы хотим представить данные способом, который сильно отличается от всего, что есть во встроенных в Qt представлениях, или если требуется видоизменить представление нескольких связанных друг с другом элементов (например, каким-то образом объединить представление двух и более элементов), то придется создавать пользовательское представление. Этим вопросом мы и займемся в следующей главе.

6

Архитектура модель/представление: представления

- Подклассы `QAbstractItemView`
- Визуализация моделей с помощью специальных представлений

В этой главе рассматриваются представления в архитектуре Qt модель/представление. Как и в двух предыдущих главах, предполагается знакомство с основными принципами архитектуры модель/представление в Qt, изложенными в начале главы 3 (стр. 106).

Для большинства задач достаточно стандартных, встроенных в Qt представлений моделей: `QListView`, `QTableView`, `QColumnView` и `QTreeView`. Чтобы изменить способ представления элементов модели, можно создать подклассы этих классов (как всегда в Qt) или прибегнуть к пользовательским делегатам. Однако есть две ситуации, когда без создания пользовательского представления не обойтись. Первая – когда требуется представление, радикально отличающееся от стандартных представлений, вторая – когда визуализация подразумевает объединение нескольких элементов данных в одном поле.

Вообще говоря, имеется два подхода к созданию пользовательских представлений. Один применяется, когда мы хотим создать представление-компонент, который потенциально можно использовать с разными моделями и который должен быть согласован с архитектурой модель/представление. В таких случаях мы обычно создаем подкласс класса `QAbstractItemView` и реализуем стандартный API представления, так, чтобы его можно было использовать в сочетании с любой моделью. Другой подход полезен, когда требуется визуализировать данные в конкретной модели некоторым уникальным способом, имеющим мало

шансов на повторное использование. Тогда мы можем просто создать специализированное средство просмотра модели, обладающее в точности той функциональностью, которая необходима, и ни на йоту больше. Обычно для этого создается подкласс `QWidget` с уникальным API, который, однако, должен включать метод `setModel()`.

В этой главе мы рассмотрим примеры как того, так и другого. Сначала будет представлен универсальный подкласс `QAbstractItemView`, предоставляющий такой же API, как и все встроенные в Qt представления; его можно использовать для любой модели, хотя проектировался он, имея в виду прежде всего отображение и редактирование списковых моделей. Затем мы рассмотрим визуализатор, ориентированный на конкретную модель и предоставляющий собственный API.

Подклассы `QAbstractItemView`

В этом разделе мы покажем, как создать подкласс класса `QAbstractItemView`, который можно будет подставить вместо стандартного представления Qt. На практике, конечно, для списковых, табличных и древовидных моделей имеются соответствующие представления, поэтому мы займемся разработкой пользовательского спискового представления, хотя те же принципы применимы к любым подклассам `QAbstractItemView`.

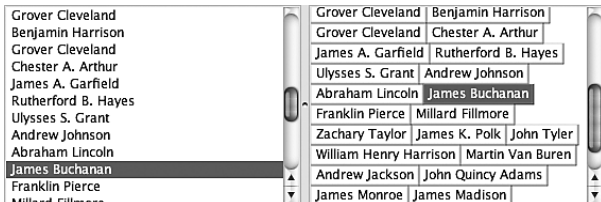


Рис. 6.1. `QListView` и `TiledListView`

На рис. 6.1 показана центральная область окна приложения `Tiled List View` (`tiledlistview`). В ней находятся два представления для одной и той же модели: слева – стандартный виджет `QListView`, справа – `TiledListView`. Обратите внимание, что при одном и том же размере области и одинаковом шрифте в `TiledListView` помещается гораздо больше данных. Кроме того, как показано на рисунке, в `TiledListView` нет понятия столбца – он показывает в каждой строке столько данных, сколько в нее умещается. Если увеличить ширину виджета, то в том же количестве строк уместилось бы больше данных.

С точки зрения удобства пользования `TiledListView` обладает одним достоинством, которое делает навигацию по нему быстрее и проще, – клавиши со стрелками не просто перемещают вперед и назад по списку. Когда пользователь нажимает клавишу со стрелкой вверх (или вниз), то выделенным становится элемент, визуально расположенный над

(или под) текущим. Стрелки влево и вправо работают, как обычно, то есть переходят к элементу, расположенному слева (справа) от текущего, пока текущим не станет крайний левый (правый) элемент в строке. В этом случае нажатие клавиши со стрелкой влечет за собой переход к логически предшествующему (последующему) элементу списка.

API класса QAbstractItemView обширен, и во время работы над книгой в документации по Qt ничего не говорилось о том, какие части API обязательно следует переопределять в подклассах, а когда достаточно реализации в базовом классе. Однако некоторые методы являются чисто виртуальными, и потому должны быть переопределены. Кроме того, в дистрибутив Qt входит пример *examples/itemviews/chart*, который может служить полезным руководством по реализации пользовательских представлений.

В табл. 6.1 перечислены методы, которые мы реализовали в классе TiledListView и которые, по нашему мнению, составляют необходимый минимум во всех подклассах QAbstractItemView. В примере *chart* из дистрибутива Qt переопределены все методы, указанные в этой таблице, а также обработчики событий `mouseReleaseEvent()` и `mouseMoveEvent()` (для поддержки выделения «эластичным контуром» – в классе TiledListView это не нужно). В том же примере реализован еще метод `edit()` для инициирования редактирования, но и он нам ни к чему – для редактирования представления TiledListView достаточно поведения, унаследованного от базового класса.

Таблица 6.1. API класс QAbstractItemView

Метод	Описание
<code>dataChanged</code> (<i>topLeft</i> , <i>bottomRight</i>)	Этот слот вызывается при изменении любого элемента, модельный индекс которого принадлежит прямоугольнику с левым верхним углом <i>topLeft</i> и правым нижним <i>bottomRight</i>
<code>horizontalOffset()*</code>	Возвращает горизонтальный сдвиг представления
<code>indexAt(point)*</code>	Возвращает модельный индекс элемента в позиции <i>point</i> порта просмотра представления
<code>isIndexHidden(index)*</code>	Возвращает <code>true</code> , если элемент с индексом <i>index</i> скрыт (и, значит, не должен показываться)
<code>mousePressEvent(event)</code>	Обычно используется для того, чтобы сделать текущим модельный индекс элемента, по которому щелкнули мышью
<code>moveCursor(how, modifiers)*</code>	Возвращает модельный индекс элемента, ставшего текущим после навигации в направлении <i>how</i> (например, вверх, вниз, влево или вправо) с учетом клавиатурных <i>модификаторов</i>
<code>paintEvent(event)</code>	Рисует содержимое представления в порту просмотра

Таблица 6.1 (продолжение)

Метод	Описание
<code>resizeEvent(event)</code>	Обычно используется для обновления полос прокрутки
<code>rowsAboutToBeRemoved(parent, start, end)</code>	Этот слот вызывается перед тем, как будут удалены строки с номерами от <i>start</i> до <i>end</i> , находящиеся под родителем с модельным индексом <i>parent</i>
<code>rowsInserted(parent, start, end)</code>	Этот слот вызывается после того, как были вставлены строки с номерами от <i>start</i> до <i>end</i> под родителем с модельным индексом <i>parent</i>
<code>scrollContentsBy(dx, dy)</code>	Прокручивает порт просмотра на <i>dx</i> пикселей по горизонтали и <i>dy</i> пикселей по вертикали
<code>scrollTo(index, hint)*</code>	Прокручивает представление так, чтобы элемент с модельным индексом <i>index</i> стал виден; при прокрутке учитывается указание <i>hint</i>
<code>setModel(model)</code>	Ассоциирует модель <i>model</i> с представлением
<code>setSelection(rect, flags)*</code>	Применяет флаги выделения <i>flags</i> ко всем элементам, которые лежат внутри или пересекают границы прямоугольника <i>rect</i>
<code>updateGeometries()</code>	Обычно используется для обновления геометрических характеристик виджетов, являющихся потомками представления, например полос прокрутки
<code>verticalOffset()</code>	Возвращает вертикальный сдвиг представления
<code>visualRect(index)*</code>	Возвращает прямоугольник, занятый элементом с модельным индексом <i>index</i>
<code>visualRegionForSelection(selection)*</code>	Возвращает область порта просмотра для элементов, принадлежащих <i>selection</i>

* Это чисто виртуальный метод, который *должен быть* реализован в подклассах.

Прежде чем приступать к рассмотрению кода класса `TiledListView`, покажем, как создается и инициализируется его экземпляр:

```
TiledListView *tiledListView = new TiledListView;
tiledListView->setModel(model);
```

Как видим, класс `TiledListView` используется точно так же, как любой другой класс представления.

Поскольку подлежащий реализации API показан в табл. 6.1, мы не станем приводить объявление класса в заголовочном файле. Исключение сделаем лишь для закрытых данных, специфичных для `TiledListView`.

```
private:
    mutable int idealWidth;
    mutable int idealHeight;
    mutable QHash<int, QRectF> rectForRow;
    mutable bool hashIsDirty;
```

Члены `idealWidth` и `idealHeight` — это ширина и высота, при которых будут показаны все элементы модели без полос прокрутки. В хеше `rectForRow` хранятся прямоугольники `QRectF` для каждой строки представления — с установленными размерами и положением. (Отметим, что, поскольку класс `TiledListView` предназначен для показа списка, строка соответствует одному элементу.) Все эти переменные нужны для внутренних манипуляций, а так как используются они в константных методах, то объявлены с модификатором `mutable`.

Обновление хеша `rectForRow` производится не сразу после изменения, а откладывается — то есть в момент изменения мы только устанавливаем флаг `hashIsDirty`. А пересчет выполняется, когда программа реально обращается к хешу, и только при условии, что `hashIsDirty` равно `true`.

Вот теперь мы почти готовы к рассмотрению реализации `TiledListView` и начнем с конструктора, объясняя закрытые вспомогательные методы по мере необходимости. Но сначала следует упомянуть о важной концептуальной особенности подклассов `QAbstractItemView`.

Базовый класс `QAbstractItemView` предоставляет для отображаемых данных область прокрутки. Та часть виджета, представленного подклассом `QAbstractItemView`, которая видна на экране, называется портом просмотра (`viewport`); именно она показана в области прокрутки. Доступ к этой видимой области дает метод `viewport()`. Фактический размер виджета не важен, важно лишь, каким должен быть размер, чтобы можно было показать все данные модели (даже если он намного превышает размер экрана). Как это влияет на код, мы увидим при рассмотрении методов `calculateRectsIfNecessary()` и `updateGeometries()`.

```
TiledListView::TiledListView(QWidget *parent)
    : QAbstractItemView(parent), idealWidth(0), idealHeight(0),
      hashIsDirty(false)
{
    setFocusPolicy(Qt::WheelFocus);
    setFont(QApplication::font("QListView"));
    horizontalScrollBar()->setRange(0, 0);
    verticalScrollBar()->setRange(0, 0);
}
```

Наш конструктор вызывает конструктор базового класса и инициализирует свои закрытые данные. В начале «идеальный» размер представления равен `0×0`, потому что еще нечего отображать.

Здесь мы производим необычное действие — вызываем `setFont()` для установки шрифта виджета, хотя обычно в пользовательских виджетах

используется шрифт, унаследованный от родителя. Метод `QApplication::font()` при передаче имени класса возвращает платформенно-зависимый шрифт, применяемый для этого класса. В результате в `TiledListView` используется правильный шрифт даже на тех платформах (например, Mac OS X), где размер шрифта в `QListView` слегка отличается от применяемого по умолчанию в `QWidget`¹.

Поскольку данных пока нет, мы задаем для обеих полос прокрутки диапазон (0, 0); это гарантирует, что полосы прокрутки не будут видны, пока не понадобятся, при этом ответственность за их сокрытие и показ возлагается на базовый класс.

```
void TiledListView::setModel(QAbstractItemModel *model)
{
    QAbstractItemView::setModel(model);
    hashIsDirty = true;
}
```

При задании модели мы сначала вызываем реализацию из базового класса, а затем присваиваем флагу `hashIsDirty` значение `true`, чтобы при первом же вызове метода `calculateRectsIfNecessary()` хеш `rectForRow` был обновлен.

Методы `indexAt()`, `setSelection()` и `viewportRectForRow()` необходимы для того, чтобы знать размер и положение всех элементов модели. Косвенно это относится и к методам `mousePressEvent()`, `moveCursor()`, `paintEvent()` и `visualRect()`, потому что все они вызывают методы, которым необходим размер и положение. Вместо того чтобы динамически вычислять прямоугольники всякий раз, как в них возникает нужда, мы решили пожертвовать памятью ради скорости и кэшировать их в объекте `rectForRow`. А чтобы не актуализировать хеш путем вызова `calculateRectsIfNecessary()` в момент изменения, мы просто запоминаем, что хеш нуждается в обновлении, и пересчитываем прямоугольник в тот момент, когда к нему кто-то обращается.

```
const int ExtraHeight = 3;

void TiledListView::calculateRectsIfNecessary() const
{
    if (!hashIsDirty)
        return;
    const int ExtraWidth = 10;
    QFontMetrics fm(font());
    const int RowHeight = fm.height() + ExtraHeight;
    const int MaxWidth = viewport()->width();
    int minimumWidth = 0;
    int x = 0;
    int y = 0;
```

¹ Подробнее о механизме наследования шрифтов и палитр в Qt можно прочитать на странице по адресу labs.qt.nokia.com/blogs/2008/11/16.

```

for (int row = 0; row < model()->rowCount(rootIndex()); ++row)
{
    QModelIndex index = model()->index(row, 0, rootIndex());
    QString text = model()->data(index).toString();
    int textWidth = fm.width(text);
    if (!(x == 0 || x + textWidth + ExtraWidth < MaxWidth)) {
        y += RowHeight;
        x = 0;
    }
    else if (x != 0)
        x += ExtraWidth;
    rectForRow[row] = QRectF(x, y, textWidth + ExtraWidth, RowHeight);
    if (textWidth > minimumWidth)
        minimumWidth = textWidth;
    x += textWidth;
}
idealWidth = minimumWidth + ExtraWidth;
idealHeight = y + RowHeight;
hashIsDirty = false;
viewport()->update();
}

```

Этот метод — сердце виджета `TiledListView`, по крайней мере в части его внешнего вида, поскольку — как мы скоро увидим — в основу всего рисования положены вычисляемые в этом методе прямоугольники.

Сначала мы смотрим, нужно ли вообще пересчитывать прямоугольники. Если да, то вычисляем высоту, необходимую для отображения строки, и максимальную ширину, доступную порту просмотра, то есть *видимую* ширину.

В главном цикле внутри этого метода мы обходим все строки (то есть все элементы) модели и извлекаем текст каждого элемента. Затем вычисляем ширину этого элемента и координаты точки, начиная с которой этот элемент надо отображать, — они зависят от того, помещается элемент в той же *строке* (визуальной), что и предыдущий, или нужно начинать новую строку. Зная размер и положение элемента, мы вычисляем соответствующий ему прямоугольник и помещаем его в хеш `rectForRow` с ключом, равным порядковому номеру строки модели.

Отметим, что во всех расчетах фигурирует фактическая ширина видимой области, но предполагается, что мы располагаем высотой, необходимой для показа всех элементов при заданной ширине. Кроме того, чтобы получить требуемый модельный индекс, мы передаем индекс родителя `QAbstractItemView::rootIndex()`, а не недопустимый модельный индекс (`QModelIndex()`). Для списковых моделей, в принципе, годится и то и другое, но с точки зрения хорошего стиля в подклассах `QAbstractItemView` правильнее использовать более общий `rootIndex()`.

В заключение мы пересчитываем идеальную ширину (она равна ширине самого широкого элемента плюс небольшое поле) и идеальную высоту (необходимую для показа всех элементов при текущей ширине порта

просмотра вне зависимости от его фактической высоты). В этот момент в переменной `y` находится суммарная высота всех строк. Идеальная ширина может оказаться больше доступной, например если порт просмотра уже имеет ширину самого широкого элемента, – в таком случае автоматически появится горизонтальная полоса прокрутки. По завершении вычислений мы вызываем метод `update()` *порта просмотра* (поскольку все рисование происходит именно в порту просмотра, а не в самом пользовательском виджете `QAbstractItemView`), чтобы перерисовать данные.

Мы нигде не используем фактическую ширину пользовательского виджета `QAbstractItemView`, да она нам и не интересна – все вычисления производятся в терминах порта просмотра и идеальных ширины и высоты.

```
QRect TiledListView::visualRect(const QModelIndex &index) const
{
    QRect rect;
    if (index.isValid())
        rect = viewportRectForRow(index.row()).toRect();
    return rect;
}
```

Этот чисто виртуальный метод должен вернуть прямоугольник, занятый элементом с данным модельным индексом. К счастью, его реализация очень проста, потому что вся работа поручается нашему закрытому методу `viewportRectForRow()`, который пользуется хешем `rectForRow`.

```
QRectF TiledListView::viewportRectForRow(int row) const
{
    calculateRectsIfNecessary();
    QRectF rect = rectForRow.value(row).toRect();
    if (!rect.isValid())
        return rect;
    return QRectF(rect.x() - horizontalScrollBar()->value(),
                  rect.y() - verticalScrollBar()->value(),
                  rect.width(), rect.height());
}
```

Этот метод вызывается из методов `visualRect()`, `moveCursor()` и `paintEvent()`. Он возвращает `QRectF` для обеспечения максимальной точности (необходимой, например, в методе `paintEvent()`); те методы, которым такая точность не нужна, преобразуют возвращенное значение в обычный `QRect` с целочисленными сторонами, обращаясь к методу `QRectF::toRect()`.

Метод `calculateRectsIfNecessary()` *должен* вызываться любым методом, который обращается к хешу `rectForRow`, еще до самого обращения. Если хеш `rectForRow` актуален, то `calculateRectsIfNecessary()` ничего не делает; в противном случае он пересчитывает все прямоугольники в хеше.

Координаты прямоугольников в хеше `rectForRow` рассчитываются, исходя из идеальной ширины (обычно видимой) и идеальной высоты (не

обходимой для отображения всех элементов при текущей ширине). Это означает, что прямоугольники, по существу, представлены в системе координат виджета, основанной на его идеальном размере (*фактический* размер виджета не имеет значения). Метод `viewportRectForRow()` должен вернуть прямоугольник в системе координат порта просмотра, поэтому мы корректируем координаты с учетом возможной прокрутки. На рис. 6.2 иллюстрируется различие между координатами виджета и порта просмотра.

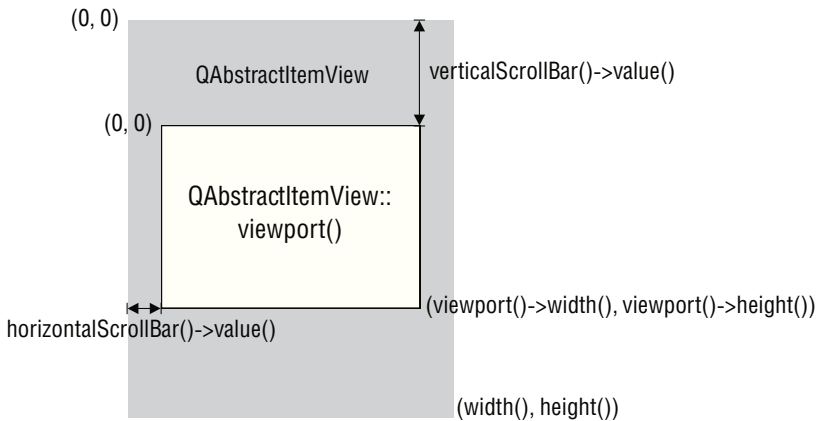


Рис. 6.2. Соотношение между координатами виджета и порта просмотра

```
bool isIndexHidden(const QModelIndex&) const { return false; }
```

Мы должны реализовать этот чисто виртуальный метод, а поскольку реализация тривиальна, то делаем это прямо в заголовочном файле. Метод рассчитан на данные, в которых могут быть скрытые элементы, например на таблицу со скрытыми строками или столбцами. Но в данном представлении скрытых элементов нет, поскольку мы не предоставляли поддержку для операции сокрытия, так что этот метод всегда возвращает `false`.

```
void TiledListView::scrollTo(const QModelIndex &index,
                             QAbstractItemView::ScrollHint)
{
    QRect viewRect = viewport()->rect();
    QRect itemRect = visualRect(index);
    if (itemRect.left() < viewRect.left())
        horizontalScrollBar()->setValue(horizontalScrollBar()->value()
                                         + itemRect.left() - viewRect.left());
    else if (itemRect.right() > viewRect.right())
        horizontalScrollBar()->setValue(horizontalScrollBar()->value()
                                         + qMin(itemRect.right() - viewRect.right(),
                                                  itemRect.left() - viewRect.left()));
}
```

```

if (itemRect.top() < viewRect.top())
    verticalScrollBar()->setValue(verticalScrollBar()->value() +
        itemRect.top() - viewRect.top());
else if (itemRect.bottom() > viewRect.bottom())
    verticalScrollBar()->setValue(verticalScrollBar()->value() +
        qMin(itemRect.bottom() - viewRect.bottom(),
            itemRect.top() - viewRect.top()));
viewport()->update();
}

```

Это еще один чисто виртуальный метод, который мы обязаны реализовать. К счастью, реализация очевидна (и почти не отличается от приведенной в примере *chart* из дистрибутива Qt).

Если прямоугольник элемента, который мы хотим сделать видимым путем прокрутки, находится слева от левого края порта просмотра, то порт следует прокрутить. Для этого нужно изменить сдвиг горизонтальной полосы прокрутки, добавив к нему разность между левым краем прямоугольника элемента и левым краем порта. Все остальные случаи рассматриваются аналогично.

Отметим, что этот метод вызывает метод `visualRect()`, который, в свою очередь, вызывает `viewportRectForRow()`, а тот уже обращается к `calculateRectsIfNecessary()`. Как уже отмечалось, этот последний метод пересчитывает прямоугольники в хеше `rectForRow`, если установлен флаг `hashIsDirty`.

```

QModelIndex TiledListView::indexAt(const QPoint &point_) const
{
    QPoint point(point_);
    point.rx() += horizontalScrollBar()->value();
    point.ry() += verticalScrollBar()->value();
    calculateRectsIfNecessary();
    QHashIterator<int, QRectF> i(rectForRow);
    while (i.hasNext()) {
        i.next();
        if (i.value().contains(point))
            return model()->index(i.key(), 0, rootIndex());
    }
    return QModelIndex();
}

```

Этот чисто виртуальный метод должен возвращать модельный индекс элемента, находящегося в данной точке. Координаты точки задаются относительно порта просмотра, но прямоугольники в хеше `rectForRow` вычислены в координатах виджета. Вместо того чтобы преобразовывать каждый прямоугольник, пытаюсь понять, содержит ли он данную точку, мы один раз преобразуем саму точку в систему координат виджета.

Методы `QPoint::rx()` и `QPoint::ry()` возвращают неконстантные ссылки на координаты точки, чтобы их можно было изменять. Не будь этих

методов, мы должны были бы писать, к примеру, `point.setX(horizontalScrollBar()->value() + point.x())`.

Мы проверяем, что хеш `rectForRow` актуален, а затем обходим все строки (элементы) в хеше в произвольном порядке, поскольку хеш – неупорядоченная структура данных. Если мы находим прямоугольник, содержащий данную точку, то сразу же возвращаем соответствующий ему модельный индекс.

Для моделей с большим количеством элементов (больше нескольких тысяч) этот метод может работать медленно, потому что в худшем случае надо будет проверить каждый прямоугольник и даже в среднем проверять приходится половину прямоугольников. Для класса `TiledListView` это вряд ли составит проблему, потому что помещать в списковую модель, какой бы она ни была, тысячи элементов – это неуважение к пользователю. В этом случае почти наверняка удобнее будет работать с древовидной моделью, каким-то образом распределив элементы по уровням.

```
void TiledListView::dataChanged(const QModelIndex &topLeft,
                               const QModelIndex &bottomRight)
{
    hashIsDirty = true;
    QAbstractItemView::dataChanged(topLeft, bottomRight);
}
```

Этот метод вызывается при любом изменении данных. Мы присваиваем флагу `hashIsDirty` значение `true`, чтобы при вызове `calculateRectsIfNecessary()` хеш `rectForRow` был обновлен, а затем вызываем реализацию из базового класса. Обратите внимание, что мы не вызываем `viewport->update()`, чтобы запланировать перерисовку. Измененные данные могут и отсутствовать на экране, поэтому перерисовка не всегда необходима, а если все же необходима, то ее запланирует реализация `dataChanged()` из базового класса.

```
void TiledListView::rowsInserted(const QModelIndex &parent,
                                int start, int end)
{
    hashIsDirty = true;
    QAbstractItemView::rowsInserted(parent, start, end);
}

void TiledListView::rowsAboutToBeRemoved(const QModelIndex &parent,
                                          int start, int end)
{
    hashIsDirty = true;
    QAbstractItemView::rowsAboutToBeRemoved(parent, start, end);
}
```

Если в модель уже вставлены новые строки или будут удалены старые, то необходимо обеспечить правильную реакцию представления. Эти случаи легко обработать, перепоручив все базовому классу; мы должны

лишь не забыть пометить, что хеш `rectForRow` не актуален, чтобы в нужный момент — например, когда методы базового класса планируют перерисовку, — он был пересчитан.

```
QModelIndex TiledListView::moveCursor(
    QAbstractItemView::CursorAction cursorAction,
    Qt::KeyboardModifiers)
{
    QModelIndex index = currentIndex();
    if (index.isValid()) {
        if ((cursorAction == MoveLeft && index.row() > 0) ||
            (cursorAction == MoveRight &&
             index.row() + 1 < model()->rowCount())) {
            const int offset = (cursorAction == MoveLeft ? -1 : 1);
            index = model()->index(index.row() + offset,
                                   index.column(), index.parent());
        }
        else if ((cursorAction == MoveUp && index.row() > 0) ||
                 (cursorAction == MoveDown &&
                  index.row() + 1 < model()->rowCount())) {
            QFontMetrics fm(font());
            const int RowHeight = (fm.height() + ExtraHeight) *
                                   (cursorAction == MoveUp ? -1 : 1);
            QRect rect = viewportRectForRow(index.row()).toRect();
            QPoint point(rect.center().x(),
                          rect.center().y() + RowHeight);
            while (point.x() >= 0) {
                index = indexAt(point);
                if (index.isValid())
                    break;
                point.rx() -= fm.width("n");
            }
        }
    }
    return index;
}
```

Если метод `calculateRectsIfNecessary()` определяет внешний вид виджета `TiledListView`, то этот метод лежит в основе его поведения. Он должен возвращать модельный индекс элемента, который нужно сделать текущим после выполнения запрошенного навигационного действия, или недопустимый индекс, если это действие выполнить невозможно.

Если пользователь нажимает клавишу со стрелкой влево (вправо), то мы должны вернуть модельный индекс предыдущего (следующего) элемента списка — или текущего элемента, если мы уже находимся в начале (в конце) списка. Это легко сделать, создав новый модельный индекс на основе индекса текущего элемента, но заменив в нем строку на предыдущую (следующую).

Обработка клавиш со стрелками вверх и вниз чуть сложнее. В обоих случаях необходимо вычислить координаты точки, лежащей выше или ниже текущего элемента. Вычисленная точка может находиться и вне порта просмотра, это неважно; главное, чтобы она принадлежала прямоугольнику какого-то элемента.

Когда пользователь нажимает клавишу со стрелкой вверх (вниз), мы должны вернуть модельный индекс элемента, находящегося над (под) текущим. Сначала мы получаем прямоугольник текущего элемента в порту просмотра. Затем определяем точку, расположенную ровно на одну строку выше (ниже) текущего элемента по вертикали и точно в центре найденного прямоугольника по горизонтали. Далее с помощью метода `indexAt()` находим модельный индекс элемента, расположенного в этой точке. Если получен допустимый индекс, значит, над (под) текущим действительно имеется какой-то элемент, и мы нашли его модельный индекс, так что можем его вернуть.

Но полученный модельный индекс может быть недопустим; так произойдет, когда над (под) текущим элементом ничего нет. Взгляните еще раз на снимок экрана на стр. 230: элементы не выровнены по правому краю, потому что все строки разной длины. Если ситуация именно такова, то мы сдвигаем точку влево на ширину символа `n` и пробуем еще раз, повторяя эту операцию, пока либо не найдем какой-то элемент (и тогда будет получен допустимый модельный индекс), либо не выйдем за левый край. Последнее означает, что над (под) текущим элементом ничего нет. Элемент выше (ниже) текущего отсутствует, когда текущий элемент находится в первой (последней) строке и пользователь нажимает клавишу со стрелкой вверх (вниз).

Если метод `moveCursor()` возвращает недопустимый объект `QModelIndex`, то базовый класс `QAbstractItemView` ничего не делает.

В нашем коде нет обработки выделения, да это и не нужно, потому что мы пользуемся API класса `QAbstractItemView`. Если пользователь осуществляет навигацию с нажатой клавишей `Shift`, то выделение распространяется на соседние элементы. Аналогично, при нажатой клавише `Ctrl` (или `⌘` на платформе `Mac OS X`) каждый элемент, по которому пользователь щелкнул мышью, становится выделенным; таким способом можно выделять несмежные элементы.

Поддержку клавиш `Home`, `End`, `Page Up` и `Page Down` мы оставили в качестве упражнения для читателя. Для реализации достаточно включить в метод `moveCursor()` код для обработки дополнительных действий `CursorAction` (например, `QAbstractItemView::MoveHome` и `QAbstractItemView::MovePageUp`).

```
int TiledListView::horizontalOffset() const
{
    return horizontalScrollBar()->value();
}
```



```
int TiledListView::verticalOffset() const
{
    return verticalScrollBar()->value();
}
```

Эти чисто виртуальные методы необходимо реализовать. Они должны возвращать смещение порта просмотра относительно начала виджета идеального размера соответственно по горизонтали и по вертикали. Реализация тривиальная, потому что нужные величины уже дают смещения полос прокрутки.

```
void TiledListView::scrollContentsBy(int dx, int dy)
{
    scrollDirtyRegion(dx, dy);
    viewport()->scroll(dx, dy);
}
```

Этот метод вызывается при перемещении полос прокрутки; он сдвигает порт просмотра на указанную величину и планирует перерисовку. За перерисовку здесь отвечает метод `QAbstractItemView::scrollDirtyRegion()`, вызываемый *до* прокрутки. Можно было бы вместо `scrollDirtyRegion()` вызвать метод `viewport->update()` *после* выполнения прокрутки.

Реализация в базовом классе просто вызывает `viewport->update()`, не выполняя никакой прокрутки. Кстати, если мы хотим выполнять прокрутку программно, то нужно вызывать не этот метод, а `QScrollBar::setValue()`.

```
void TiledListView::setSelection(const QRect &rect,
                                QFlags<QItemSelectionModel::SelectionFlag> flags)
{
    QRect rectangle = rect.translated(horizontalScrollBar()->value(),
                                       verticalScrollBar()->value()).normalized();
    calculateRectsIfNecessary();
    QHashIterator<int, QRectF> i(rectForRow);
    int firstRow = model()->rowCount();
    int lastRow = -1;
    while (i.hasNext()) {
        i.next();
        if (i.value().intersects(rectangle)) {
            firstRow = firstRow < i.key() ? firstRow : i.key();
            lastRow = lastRow > i.key() ? lastRow : i.key();
        }
    }
    if (firstRow != model()->rowCount() && lastRow != -1) {
        QItemSelection selection(
            model()->index(firstRow, 0, rootIndex()),
            model()->index(lastRow, 0, rootIndex()));
        selectionModel()->select(selection, flags);
    }
    else {
```

```

        QModelIndex invalid;
        QItemSelection selection(invalid, invalid);
        selectionModel()->select(selection, flags);
    }
}

```

Этот чисто виртуальный метод применяет заданные флаги выделения ко всем элементам, которые находятся внутри или пересекают границы указанного прямоугольника. Чтобы фактически произвести выделение, необходимо вызвать метод `QAbstractItemView::selectionModel()->select()`. Приведенная здесь реализация очень похожа на ту, что используется в примере *chart* из дистрибутива Qt.

Координаты переданного прямоугольника вычислены относительно порта просмотра, поэтому сначала мы преобразуем их в систему координат виджета, так как именно в ней вычислены координаты прямоугольников в хеше `rectForRow`. Мы обходим все строки (элементы) в хеше в произвольном порядке и, обнаружив элемент, чей прямоугольник пересекает заданный, включаем этот элемент в состав выделения (если его там еще нет), обновляя номера первой и последней строк.

Если номера первой и последней строк выделения установлены, то мы создаем объект `QItemSelection`, охватывающий эти строки и все промежуточные, и обновляем модель выделения для данного представления. Если же номер хотя бы одной строки не установлен, то мы создаем недопустимый индекс `QModelIndex` и обновляем модель выделения с его помощью.

```

QRegion TiledListView::visualRegionForSelection(
    const QItemSelection &selection) const
{
    QRegion region;
    foreach (const QItemSelectionRange &range, selection) {
        for (int row = range.top(); row <= range.bottom(); ++row) {
            for (int column = range.left(); column < range.right();
                ++column) {
                QModelIndex index = model()->index(row, column, rootIndex());
                region += visualRect(index);
            }
        }
    }
    return region;
}

```

Этот чисто виртуальный метод необходимо реализовать, так чтобы он возвращал регион `QRegion`, охватывающий все выделенные в представлении элементы, показанные в порту просмотра; при этом используется система координат порта. Приведенная реализация очень похожа на ту, что используется в примере *chart* из дистрибутива Qt.

Сначала мы создаем пустой регион. Затем обходим все выделенные области, если таковые имеются. Для каждой выделенной области нахо-

дим модельный индекс каждого входящего в нее элемента и добавляем его визуальный прямоугольник в регион.

Наша реализация `visualRect()` вызывает метод `viewportRectForRow()`, который получает прямоугольник из хеша `rectForRow` и возвращает его, предварительно преобразовав в систему координат порта просмотра (поскольку координаты прямоугольников в хеше вычислены относительно виджета). В данном случае мы могли бы обойтись без метода `visualRect()` и работать с хешем напрямую, но предпочли более общую реализацию, которая легко адаптируется к другим пользовательским представлениям.

```
void TiledListView::paintEvent(QPaintEvent*)
{
    QPainter painter(viewport());
    painter.setRenderHints(QPainter::Antialiasing|
                           QPainter::TextAntialiasing);
    for (int row = 0; row < model()->rowCount(rootIndex()); ++row){
        QModelIndex index = model()->index(row, 0, rootIndex());
        QRectF rect = viewportRectForRow(row);
        if (!rect.isValid() || rect.bottom() < 0 ||
            rect.y() > viewport()->height())
            continue;
        QStyleOptionViewItem option = viewOptions();
        option.rect = rect.toRect();
        if (selectionModel()->isSelected(index))
            option.state |= QStyle::State_Selected;
        if (currentIndex() == index)
            option.state |= QStyle::State_HasFocus;
        itemDelegate()->paint(&painter, option, index);
        paintOutline(&painter, rect);
    }
}
```

Нарисовать представление просто до изумления — ведь прямоугольники всех элементов уже вычислены и хранятся в хеше `rectForRow`. Но обратите внимание, что мы рисуем *порт просмотра* виджета, а не сам виджет. И, как обычно, явно включаем сглаживание, поскольку не делаем предположение о том, как выставлены значения по умолчанию.

Мы обходим все элементы, получая для каждого модельный индекс и прямоугольник в координатах порта просмотра. Если прямоугольник недопустим (такого быть не должно) или не виден в порту просмотра (то есть нижний край оказался выше порта или верхний край — ниже), то мы его и не рисуем.

Если же элемент рисуется, то сначала мы получаем объект `QStyleOptionViewItem`, предоставляемый базовым классом, записываем в него прямоугольник элемента, преобразованный из `QRectF` в `QRect` методом `QRectF::toRect()`, и изменяем состояние, если элемент является выделенным или текущим.

Но самое главное – мы *не* рисуем элемент сами! Вместо этого мы обращаемся к делегату представления – это может быть как встроенный делегат `QStyledItemDelegate` из базового класса, так и пользовательский делегат, установленный клиентом класса, – и поручаем ему нарисовать элемент. Так что наше представление поддерживает пользовательские делегаты.

Элементы рисуются строка за строкой, максимально используя доступное пространство. Но поскольку текст элемента может состоять из нескольких слов, мы должны обеспечить пользователю возможность понять, где кончается один элемент и начинается следующий. Для этого мы окружаем каждый элемент рамкой.

```
void TiledListView::paintOutline(QPainter *painter, const QRectF &rectangle)
{
    const QRectF rect = rectangle.adjusted(0, 0, -1, -1);
    painter->save();
    painter->setPen(QPen(palette().dark().color(), 0.5));
    painter->drawRect(rect);
    painter->setPen(QPen(Qt::black, 0.5));
    painter->drawLine(rect.bottomLeft(), rect.bottomRight());
    painter->drawLine(rect.bottomRight(), rect.topRight());
    painter->restore();
}
```

Чтобы нарисовать рамку, мы сначала рисуем прямоугольник, а потом проводим два отрезка – под нижней стороной прямоугольника и справа от него, – чтобы создать эффект тени.

```
void TiledListView::resizeEvent(QResizeEvent*)
{
    hashIsDirty = true;
    calculateRectsIfNecessary();
    updateGeometries();
}
```

Если изменяется размер представления, то мы должны пересчитать прямоугольники всех элементов и обновить параметры полос прокрутки. С методом `calculateRectsIfNecessary()` мы уже ознакомились (стр. 234), теперь рассмотрим метод `updateGeometries()`.

```
void TiledListView::updateGeometries()
{
    QFontMetrics fm(font());
    const int RowHeight = fm.height() + ExtraHeight;
    horizontalScrollBar()->setSingleStep(fm.width("n"));
    horizontalScrollBar()->setPageStep(viewport()->width());
    horizontalScrollBar()->setRange(0,
        qMax(0, idealWidth - viewport()->width()));
    verticalScrollBar()->setSingleStep(RowHeight);
    verticalScrollBar()->setPageStep(viewport()->height());
    verticalScrollBar()->setRange(0,
```

```

        qMax(0, idealHeight - viewport()->height()));
    }

```

Этот защищенный слот появился в версии Qt 4.4 и служит для обновления дочерних виджетов представления, например полос прокрутки.

Идеальная ширина и высота виджета вычислены в методе `calculateRectsIfNecessary()`. Высота всегда достаточна для показа *всех* данных модели, и ширина тоже – если порт просмотра достаточно широк для показа самого широкого элемента. Выше уже отмечалось, что фактический размер виджета представления не имеет значения, так как пользователь видит только порт просмотра.

В качестве величины шага горизонтальной полосы прокрутки (то есть расстояния, на которое сдвигается ее ползунок, когда пользователь щелкает по одной из стрелок) мы задаем ширину буквы *n*, то есть один символ. А в качестве величины шага прокрутки страницы (то есть расстояния, на которое она сдвигается, когда пользователь щелкает слева или справа от ползунка) – ширину порта. Кроме того, диапазон горизонтальной полосы прокрутки простирается от 0 до идеальной ширины виджета за вычетом ширины порта (потому что порт уже виден). Вертикальная полоса прокрутки настраивается аналогично.

```

void TiledListView::mousePressEvent(QMouseEvent *event)
{
    QAbstractItemView::mousePressEvent(event);
    setCurrentIndex(indexAt(event->pos()));
}

```

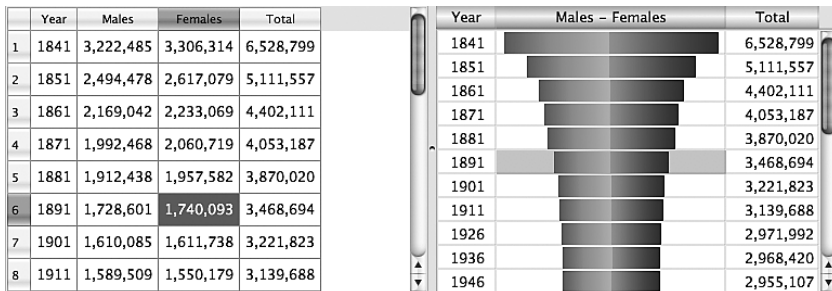
Это последний обработчик события, который нам необходимо реализовать. Он нужен для того, чтобы элемент, по которому щелкнул пользователь, стал выделенным и текущим. Поскольку наше представление является подклассом `QAbstractItemView`, который сам является подклассом `QAbstractScrollArea`, то позиция мыши в объекте `QMouseEvent` представлена в координатах порта просмотра. Но это не проблема, потому что метод `indexAt()` ожидает получения параметра типа `QPoint` тоже в координатах порта.

Последнее, что стоит отметить по поводу класса `TiledListView`, – это предположение о том, что пользователь работает с языком, читаемым слева направо, как, например, английский. Пользователи, для которых родным является арабский язык или иврит, будут ощущать неудобство при работе с этим классом, поскольку они привыкли читать справа налево. Доработка этого класса так, чтобы он умел выводить списки элементов как слева направо, так и справа налево, оставлена читателю в качестве упражнения. (Флаг направления вывода в виджете можно получить от метода `QWidget::layoutDirection()`, обычно он совпадает с флагом `QApplication::layoutDirection()`, но для полной корректности лучше все же пользоваться вариантом из класса `QWidget`.)

Как и во всех стандартных классах представлений, включенных в Qt, в классе `TiledListView` имеется взаимно однозначное соответствие между элементами данных и отображаемыми элементами. Но бывают случаи, когда несколько элементов данных желательно объединить при выводе. В API класса `QAbstractItemView` отсутствует соответствующая поддержка, пользовательские делегаты тут тоже не помогут. Однако мы все же можем написать представление, которое визуализирует данные точно так, как нам нужно (этому посвящен следующий раздел), но для этого должны будем отказаться от API класса `QAbstractItemView` и предоставить свой собственный.

Визуализация моделей с помощью специальных представлений

В этом разделе мы создадим класс представления с нуля, унаследовав его непосредственно от класса `QWidget`, и предоставим свой API, полностью отличающийся от API класса `QAbstractItemView`. Можно было бы создать также подкласс `QAbstractItemView`, но поскольку мы собираемся написать представление, ориентированное на одну конкретную модель, и объединить при показе несколько элементов, то вряд ли имеет смысл следовать принципам API, который не будет использоваться.



	Year	Males	Females	Total
1	1841	3,222,485	3,306,314	6,528,799
2	1851	2,494,478	2,617,079	5,111,557
3	1861	2,169,042	2,233,069	4,402,111
4	1871	1,992,468	2,060,719	4,053,187
5	1881	1,912,438	1,957,582	3,870,020
6	1891	1,728,601	1,740,093	3,468,694
7	1901	1,610,085	1,611,738	3,221,823
8	1911	1,589,509	1,550,179	3,139,688

	Year	Males – Females	Total
	1841		6,528,799
	1851		5,111,557
	1861		4,402,111
	1871		4,053,187
	1881		3,870,020
	1891		3,468,694
	1901		3,221,823
	1911		3,139,688
	1926		2,971,992
	1936		2,968,420
	1946		2,955,107

Рис. 6.3. Представления `QTableView` и `CensusVisualizer`

Визуализатор, который мы создадим, предназначен для представления данных переписи населения. Данные хранятся в табличной модели; в каждой строке отображаются год, количество мужчин, количество женщин и общее количество мужчин и женщин. На рис. 6.3 показана центральная часть приложения `Census Visualizer` (`censusvisualizer`). В ней есть два представления данных: слева стандартный `QTableView`, справа `CensusVisualizer`, в котором данные о мужчинах и женщинах представлены полосками с градиентной заливкой, длина которых пропорциональна числовому значению.

Мы не можем воспользоваться для вывода заголовков классом `QHeaderView`, потому что два столбца объединены в один. Из-за этого мы соз-

дали представление `CensusVisualizer` в виде подкласса `QWidget`, агрегирующего три виджета: `CensusVisualizerHeader` для представления горизонтального заголовка, `CensusVisualizerView` для визуализации данных и `QScrollArea`, который содержит `CensusVisualizerView` и предоставляет поддержку для прокрутки и изменения размера. Соотношение между этими классами показано на рис. 6.4.

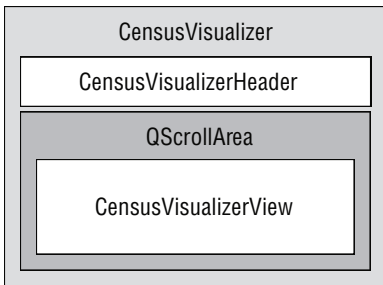


Рис. 6.4. Соотношение между классами в приложении *CensusVisualizer*

Начинаем с создания визуализатора в функции `main()` приложения.

```
CensusVisualizer *censusVisualizer = new CensusVisualizer;
censusVisualizer->setModel(model);
```

Тут нет никаких неожиданностей — создается объект визуализатора и с помощью метода `CensusVisualizer::setModel()` ему передается модель. Далее в функции `main()` создается объект `QTableView`, оба представления размещаются в окне и устанавливаются соединения между сигналами и слотами, обеспечивающие желаемое поведение программы. Все это мы опустим и сосредоточимся на проектировании и кодировании класса визуализатора и агрегируемых им классов заголовка и представления.

Виджет визуализатора

Именно с виджетом визуализатора имеют дело пользователи приложения, поэтому мы начнем с рассмотрения класса `CensusVisualizer`. А затем перейдем к двум пользовательским классам, которые визуализатор агрегирует для придания требуемого внешнего вида. Ниже приведено объявление `CensusVisualizer` в заголовочном файле, но без закрытых данных-членов:

```
class CensusVisualizer : public QWidget
{
    Q_OBJECT

public:
    explicit CensusVisualizer(QWidget *parent=0);
    QAbstractItemModel *model() const { return m_model; }
    void setModel(QAbstractItemModel *model);
    QScrollArea *scrollArea() const { return m_scrollArea; }
```

```

    int maximumPopulation() const { return m_maximumPopulation; }
    int widthOfYearColumn() const { return m_widthOfYearColumn; }
    int widthOfMaleFemaleColumn() const;
    int widthOfTotalColumn() const { return m_widthOfTotalColumn; }
    int selectedRow() const { return m_selectedRow; }
    void setSelectedRow(int row);
    int selectedColumn() const { return m_selectedColumn; }
    void setSelectedColumn(int column);
    void paintItemBorder(QPainter *painter,
                        const QPalette &palette, const QRect &rect);
    QString maleFemaleHeaderText() const;
    int maleFemaleHeaderTextWidth() const;
    int xOffsetForMiddleOfColumn(int column) const;
    int yOffsetForRow(int row) const;

public slots:
    void setCurrentIndex(const QModelIndex &index);

signals:
    void clicked(const QModelIndex&);

private:
    ...
};

```

Хотя данные-члены не показаны, стоит отметить, что агрегированный экземпляр `CensusVisualizerHeader` хранится в закрытой переменной `header`, а экземпляр `CensusVisualizerView` — в закрытой переменной `view`. И та и другая, разумеется, являются указателями. Кроме того, в классе хранится указатель на модель и на экземпляр `QScrollArea`, содержащий `CensusVisualizerView`. Остальные закрытые данные-члены — целые числа. Методы чтения для них по большей части встраиваемые и находятся в заголовочном файле, а методы установки (для тех переменных, которые допускают установку) мы рассмотрим чуть ниже.

Метод `maximumPopulation()` служит для вычисления максимальной ширины полос «Мужчины — Женщины» (`Males — Females`) и вызывается из метода `setModel()`.

Методы чтения ширины используются как заголовком, так и представлением для рисования. Номера выделенной строки и выделенного столбца запоминаются и используются заголовком для подсветки выделенного столбца, а представлением — для подсветки выделенного элемента (или пары элементов «Мужчины — Женщины»).

Если выделенный элемент изменяется в результате щелчка пользователя по представлению, то мы генерируем сигнал `clicked()`, чтобы уведомить все заинтересованные объекты.

К невстраиваемым частям класса `CensusVisualizer` относятся конструктор и десять методов. Методы `paintItemBorder()`, `maleFemaleHeaderText()` и `maleFemaleHeaderTextWidth()` используются как агрегированным заго-

ловком, так и представлением, поэтому мы пока отложим их рассмотрение и займемся остальными.

```
const int Invalid = -1;

CensusVisualizer::CensusVisualizer(QWidget *parent)
    : QWidget(parent), m_model(0), m_selectedRow(Invalid),
      m_selectedColumn(Invalid), m_maximumPopulation(Invalid)
{
    QFontMetrics fm(font());
    m_widthOfYearColumn = fm.width("W9999W");
    m_widthOfTotalColumn = fm.width("W9,999,999W");
    view = new CensusVisualizerView(this);
    header = new CensusVisualizerHeader(this);
    m_scrollArea = new QScrollArea;
    m_scrollArea->setBackgroundRole(QPalette::Light);
    m_scrollArea->setWidget(view);
    m_scrollArea->installEventFilter(view);
    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(header);
    layout->addWidget(m_scrollArea);
    layout->setContentsMargins(0, 0, 0, 0);
    layout->setSpacing(0);
    setLayout(layout);
    connect(view, SIGNAL(clicked(const QModelIndex&)),
           this, SIGNAL(clicked(const QModelIndex&)));
}
```

Сначала мы фиксируем ширину столбцов года и общего количества исходя из максимальных ожидаемых числовых значений плюс небольшое поле¹. Ширина столбца «Всего» (Total) – это лишь начальное значение по умолчанию; фактическая величина пересчитывается в методе `setModel()` и зависит от максимального значения численности населения в модели. Затем мы создаем агрегированные виджеты представления и заголовка. В качестве родителя передается `this`, но поскольку представление содержится в объекте `QScrollArea`, то `QScrollArea` автоматически назначит родителем себя.

Класс `QScrollArea` необычен для Qt в том смысле, что *не* рассчитан на наследование. Предполагается, что объект `QScrollArea` будет агрегирован каким-то другим виджетом, как мы здесь и поступили. Это самый простой подход, но если наследование все-таки необходимо, то можно создать подкласс `QAbstractScrollArea` – некоторые встроенные в Qt классы так и делают.

¹ В этой книге мы используем букву W для вычисления величины горизонтального отступа, а букву n – когда хотим получить ширину одного символа, например, для горизонтальной прокрутки.

Мы устанавливаем представление в качестве фильтра событий для области прокрутки. Это означает, что любое событие, адресованное области прокрутки, сначала будет послано методу `eventFilter()` представления. Зачем это нужно, мы увидим, когда ознакомимся с классом `CensusVisualizerView` поближе.

Компоновка виджетов вполне традиционна, единственная особенность в том, что поля и промежутки равны 0; в результате `CensusVisualizer` выглядит так же, как другие виджеты, не занимая дополнительную площадь под границы и без промежутка между `CensusVisualizerHeader` и `CensusVisualizerView` (который располагается внутри `QScrollArea`).

Соединение немного необычно, так как здесь сигнал соединяется с сигналом. Это означает, что генерация первого сигнала вызывает генерацию второго. Таким образом, когда пользователь щелкает по представлению (то есть выбирает элемент), сигнал `clicked()` от представления поступает в `CensusVisualizer`, а тот тоже генерирует сигнал `clicked()` с тем же самым параметром `QModelIndex`. Поэтому клиенты виджета `CensusVisualizer` могут подключаться к его сигналу `clicked()`, не заботясь о том, как все устроено внутри. В результате компонент `CensusVisualizer` оказывается гораздо более автономным, чем это было бы возможно, если бы он раскрывал агрегированные объекты.

```
enum {Year, Males, Females, Total};

void CensusVisualizer::setModel(QAbstractItemModel *model)
{
    if (model) {
        QLocale locale;
        for (int row = 0; row < model->rowCount(); ++row) {
            int total = locale.toInt(model->data(
                model->index(row, Total)).toString());
            if (total > m_maximumPopulation)
                m_maximumPopulation = total;
        }
        QString population = QString::number(m_maximumPopulation);
        population = QString("%1%2")
            .arg(population.left(1).toInt() + 1)
            .arg(QString(population.length() - 1, QChar('0')));
        m_maximumPopulation = population.toInt();
        QFontMetrics fm(font());
        m_widthOfTotalColumn = fm.width(QString("W%1%2W")
            .arg(population)
            .arg(QString(population.length() / 3, ',')));
    }
    m_model = model;
    header->update();
    view->update();
}
```

Когда устанавливается новая модель, мы должны потребовать, чтобы заголовок и само представление обновили себя. Но сначала необходимо подсчитать максимальную численность населения в столбце. Для этого мы находим в данных наибольшее значение этого поля, а затем округляем его до ближайшего большего целого числа, у которого цифра в старшем разряде на единицу больше. Например, если максимальная численность населения равна 8 392 174, то после округления получится 9 000 000.

Алгоритм очень грубый, но эффективный: мы создаем строку, в которой первый символ равен цифре в старшем разряде плюс единица, а затем следует столько нулей, сколько имеется цифр в числе (без единицы). Потом эта строка преобразуется в `int`. Для создания строки нулей мы вызываем конструктор `QString`, который принимает два аргумента, счетчик и символ, и возвращает строку, содержащую символ, повторенный указанное число раз.

Обратите внимание, что мы не можем получить значения общей численности, воспользовавшись методом `model->data(model->index(row, Total).toInt()`, потому что в модели данные хранятся в виде локализованных строк (например, «8,392,174» в США и Великобритании и «8.392.174» в Германии), а не целых чисел. Решение состоит в том, чтобы извлечь данные с помощью метода `toString()`, а затем обратиться к методу `QLocale::toInt()`, который принимает целое число в виде локализованной строки и возвращает соответствующее ей целое значение.

В классе `QLocale` имеются также методы `toFloat()` и `toDouble()`, методы для других целых типов, например `toInt()`, и методы для преобразования локализованных строк в дату и время. Конструктор `QLocale` без аргументов создает объект, соответствующий текущей локализации приложения, но можно воспользоваться конструктором с одним аргументом, которому передается имя локализации, состоящее из кода языка в стандарте ISO 639 и кода страны в стандарте ISO 3166, либо конструктором с двумя аргументами: перечислениями для кодов языков и стран, определенными в Qt.

В конструкторе мы задали начальную ширину столбца «Всего» по умолчанию, но теперь можем указать правильную ширину, достаточную для отображения фактических данных. Она равна числу пикселей, которые занимает максимальное значение, плюс ширина двух букв W для отступа, плюс место для запятых (или иного разделителя групп) между каждыми тремя цифрами.

```
const int ExtraWidth = 5;

int CensusVisualizer::widthOfMaleFemaleColumn() const
{
    return width() - (m_widthOfYearColumn +
                     m_widthOfTotalColumn + ExtraWidth +
                     m_scrollArea->verticalScrollBar()->sizeHint().width());
}
```

Этот метод возвращает подходящую ширину для столбца «Мужчины – Женщины». Он занимает все место в представлении `CensusVisualizer`, кроме ширины, отведенной под два других столбца и вертикальную полосу прокрутки, и небольшого поля. В результате при изменении размера `CensusVisualizer` все не занятое ничем другим место отдается столбцу «Мужчины – Женщины».

```
void CensusVisualizer::setSelectedRow(int row)
{
    m_selectedRow = row;
    view->update();
}

void CensusVisualizer::setSelectedColumn(int column)
{
    m_selectedColumn = column;
    header->update();
}
```

Если выделенная строка была изменена программно, то представление должно обновить себя, правильно показывая подсвеченный элемент. Аналогично, если изменен выделенный столбец, то заголовок должен подсветить его название.

```
void CensusVisualizer::setCurrentIndex(const QModelIndex &index)
{
    setSelectedRow(index.row());
    setSelectedColumn(index.column());
    int x = xOffsetForMiddleOfColumn(index.column());
    int y = yOffsetForRow(index.row());
    m_scrollArea->ensureVisible(x, y, 10, 20);
}
```

Этот слот служит удобным подспорьем для клиентов, позволяя изменять выделенный в `CensusVisualizer` элемент с помощью соединения сигнала со слотом.

Установив строку и столбец, мы делаем их видимыми в области прокрутки. Метод `QScrollArea::ensureVisible()` принимает две координаты и необязательные поля по горизонтали и по вертикали (по умолчанию оба равны 50 пикселям). Мы уменьшили поля, чтобы избежать нежелательной прокрутки, когда пользователь щелкает по верхней или нижней видимой строке.

Тут необходимо выбрать компромиссное решение. Если вертикальное поле слишком велико, то щелчок по верхнему или нижнему элементу приведет к ненужной прокрутке. А если оно слишком мало и пользователь попадает в виджет в результате нажатия клавиши `Tab`, а затем, нажимая клавишу со стрелкой вниз, доходит до нижнего элемента, то этот элемент будет показан не полностью.

```
int CensusVisualizer::xOffsetForMiddleOfColumn(int column) const
```

```

{
    switch (column) {
        case Year: return widthOfYearColumn() / 2;
        case Males: return widthOfYearColumn() +
            (widthOfMaleFemaleColumn() / 4);
        case Females: return widthOfYearColumn() +
            ((widthOfMaleFemaleColumn() * 4) / 3);
        default: return widthOfYearColumn() +
            widthOfMaleFemaleColumn() +
            (widthOfTotalColumn() / 2);
    }
}

```

Этот метод возвращает смещение по горизонтали для текущего столбца. Для этого он вычисляет среднюю линию переданного столбца, учитывая ширины всех столбцов.

```

const int ExtraHeight = 5;

int CensusVisualizer::yOffsetForRow(int row) const
{
    return static_cast<int>((QFontMetricsF(font()).height()
        + ExtraHeight) * row);
}

```

Этот метод возвращает смещение по вертикали для заданной строки, для чего умножает порядковый номер строки на высоту одной строки.

Смещения, возвращаемые методами `xOffsetForMiddleOfColumn()` и `yOffsetForRow()`, вычислены в предположении, что размер `CensusVisualizerView` в точности такой, какой нужен для показа всех данных. Это допущение верно, потому что класс `CensusVisualizerView` следит за его выполнением; мы убедимся в этом при рассмотрении метода `CensusVisualizerView::eventFilter()`. Таким образом, даже если возможно отобразить только часть представления, заниматься вычислениями, связанными с прокруткой, нам не придется, потому что об этом позаботится виджет `QScrollArea`, в котором располагается `CensusVisualizerView`.

На этом мы заканчиваем рассмотрение класса `CensusVisualizer`. Помимо конструктора и метода `setModel()`, в нем почти нет кода. Объясняется это тем, что все аспекты внешнего вида виджета и большая часть его поведения реализуются экземплярами классов `CensusVisualizerHeader` и `CensusVisualizerView`, которые создаются и компонуются в конструкторе класса `CensusVisualizer`. Далее мы рассмотрим оба этих класса, начав с заголовка.

Агрегированный заголовок визуализатора

Виджет `CensusVisualizerHeader` предоставляет заголовки столбцов для класса `CensusVisualizer` (см. рис. 6.3 на стр. 247). Поскольку мы рисуем его самостоятельно, то не упустим возможность придать ему большую

трехмерность, чем у стандартного `QHeaderView`, воспользовавшись разными градиентными заливками. (Если бы мы хотели добиться точного сходства с `QHeaderView`, то могли бы выполнять рисование с помощью методов класса `QStyle`.)

Объявление класса в заголовочном файле очень простое, поэтому приведем его открытую часть целиком:

```
class CensusVisualizerHeader : public QWidget
{
    Q_OBJECT

public:
    explicit CensusVisualizerHeader(QWidget *parent) : QWidget(parent) {}
    QSize minimumSizeHint() const;
    QSize sizeHint() const { return minimumSizeHint(); }

protected:
    void paintEvent(QPaintEvent *event);
    ...
};
```

Тело конструктора пусто. Реализуются только методы `minimumSizeHint()`, `sizeHint()`, `paintEvent()` и два закрытых вспомогательных метода, вызываемых из `paintEvent()`.

```
QSize CensusVisualizerHeader::minimumSizeHint() const
{
    CensusVisualizer *visualizer = qobject_cast<CensusVisualizer*>(
        parent());

    Q_ASSERT(visualizer);
    return QSize(visualizer->widthOfYearColumn() +
        visualizer->maleFemaleHeaderTextWidth() +
        visualizer->widthOfTotalColumn(),
        QFontMetrics(font()).height() + ExtraHeight);
}
```

Значения ширины столбцов возвращает родительский объект `CensusVisualizer`, поэтому мы вынуждены применить приведение типов – с помощью `qobject_cast<>()`, как здесь, или `dynamic_cast<>()`, – чтобы получить указатель на родителя, а затем уже у него затребовать нужные данные. (Для использования `dynamic_cast<>()` в компиляторе должна быть включена поддержка RTTI – динамической идентификации типа данных, – которая в современных компиляторах обычно включена по умолчанию.) Минимальная необходимая нам ширина равна сумме ширин всех столбцов, а минимальная высота равна высоте символа из шрифта виджета плюс небольшое поле.

Метод `maleFemaleHeaderTextWidth()` и метод, который он вызывает, предоставляются классом `CensusVisualizer`, поскольку они используются в обоих агрегированных виджетах. Приведем их для полноты картины.

```

int CensusVisualizer::maleFemaleHeaderTextWidth() const
{
    return QFontMetrics(font()).width(maleFemaleHeaderText());
}

QString CensusVisualizer::maleFemaleHeaderText() const
{
    if (!m_model)
        return " - ";
    return QString("%1 - %2")
        .arg(m_model->headerData(Males, Qt::Horizontal).toString())
        .arg(m_model->headerData(Females, Qt::Horizontal)
            .toString());
}

```

Метод `maleFemaleHeaderTextWidth()` возвращает ширину, необходимую для показа названия столбца «Мужчины – Женщины», а метод `maleFemaleHeaderText()` возвращает само название.

```

void CensusVisualizerHeader::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHints(QPainter::Antialiasing|
                           QPainter::TextAntialiasing);
    paintHeader(&painter, height());
    painter.setPen(QPen(palette().button().color().darker(), 0.5));
    painter.drawRect(0, 0, width(), height());
}

```

Метод `paintEvent()` конфигурирует объект `painter`, поручая основную работу методу `paintHeader()`, а в конце рисует прямоугольник вокруг всего заголовка.

```

void CensusVisualizerHeader::paintHeader(QPainter *painter,
const int RowHeight)
{
    const int Padding = 2;

    CensusVisualizer *visualizer = qobject_cast<CensusVisualizer*>(
        parent());

    Q_ASSERT(visualizer);
    paintHeaderItem(painter,
        QRect(0, 0, visualizer->widthOfYearColumn() + Padding,
            RowHeight),
        visualizer->model()->headerData(Year, Qt::Horizontal)
            .toString(),
        visualizer->selectedColumn() == Year);
    paintHeaderItem(painter,
        QRect(visualizer->widthOfYearColumn() + Padding, 0,
            visualizer->widthOfMaleFemaleColumn(), RowHeight),
        visualizer->maleFemaleHeaderText(),
        visualizer->selectedColumn() == Males ||

```

```

        visualizer->selectedColumn() == Females);
    ...
}

```

Этот метод по очереди рисует заголовки всех столбцов. Для каждого столбца он вызывает метод `paintHeaderItem()`, которому передается объект `painter`, прямоугольник, где выполняется рисование, текст названия и признак выделенного элемента (столбца). Мы опустили код для столбца «Всего», потому что он в точности аналогичен коду для столбца «Год» (Year).

```

void CensusVisualizerHeader::paintHeaderItem(QPainter *painter,
        const QRect &rect, const QString &text, bool selected)
{
    CensusVisualizer *visualizer = qobject_cast<CensusVisualizer*>(
        parent());

    Q_ASSERT(visualizer);
    int x = rect.center().x();
    QLinearGradient gradient(x, rect.top(), x, rect.bottom());
    QColor color = selected ? palette().highlight().color()
        : palette().button().color();
    gradient.setColorAt(0, color.darker(125));
    gradient.setColorAt(0.5, color.lighter(125));
    gradient.setColorAt(1, color.darker(125));
    painter->fillRect(rect, gradient);
    visualizer->paintItemBorder(painter, palette(), rect);
    painter->setPen(selected ? palette().highlightedText().color()
        : palette().buttonText().color());
    painter->drawText(rect, text, QTextOption(Qt::AlignCenter));
}

```

Это тот метод, который, собственно, и рисует заголовки столбцов. Сначала мы получаем указатель на объект `CensusVisualizer`, потому что собираемся воспользоваться одним из его методов. Затем создаем линейный градиент, цвет которого зависит от того, выделен элемент или нет. Градиент меняется от светлого тона в середине к более темному сверху и снизу, причем цвета выбираются соответственно светлее и темнее, чем в классе `QHeaderView`, чтобы создать более выпуклый трехмерный эффект. Настроив градиент, мы используем его для рисования фона. Затем рисуем рамку вокруг элемента, состоящую из двух линий: вдоль нижнего и правого края. И напоследок выводим текст по центру прямоугольника.

Для полноты приведем код метода `paintItemBorder()`:

```

void CensusVisualizer::paintItemBorder(QPainter *painter,
        const QPalette &palette, const QRect &rect)
{
    painter->setPen(QPen(palette.button().color().darker(), 0.33));
    painter->drawLine(rect.bottomLeft(), rect.bottomRight());
    painter->drawLine(rect.bottomRight(), rect.topRight());
}

```


Мы решили нарисовать «рамку» всего из двух отрезков, потому что в данном случае визуальный эффект получается лучше, чем при рисовании полного прямоугольника.

На этом мы прощаемся с классом `CensusVisualizerHeader`. Он на удивление прост, основная работа сводится к настройке объекта `painter` и градиента, ну и еще немного несложного рисования. Налицо разительный контраст с классом `CensusVisualizerView`, где нам придется реализовывать как внешний вид, так и поведение. Ему посвящен следующий раздел.

Агрегированное представление визуализатора

Для отображения данных модели служит пользовательский класс `CensusVisualizerView`. Фактический размер виджета не имеет значения, потому что он помещен в объект `QScrollArea`, который при необходимости показывает полосы прокрутки и занимается всеми относящимися к прокрутке вопросами. А мы можем сконцентрироваться на внешнем виде и поведении виджета. Ниже приведена открытая часть объявления класса из заголовочного файла.

```
class CensusVisualizerView : public QWidget
{
    Q_OBJECT

public:
    explicit CensusVisualizerView(QWidget *parent);
    QSize minimumSizeHint() const;
    QSize sizeHint() const;

signals:
    void clicked(const QModelIndex&);

protected:
    bool eventFilter(QObject *target, QEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void keyPressEvent(QKeyEvent *event);
    void paintEvent(QPaintEvent *event);
    ...
};
```

В этом классе есть еще несколько закрытых методов – все они, как мы увидим ниже, нужны для обеспечения рисования данных – и одна закрытая переменная, указатель на родительский объект `CensusVisualizer`. Мы вкратце рассмотрим открытые методы, а затем займемся трудной частью: защищенными обработчиками событий, их назначением и принципами работы. Но сначала взгляните на конструктор.

```
CensusVisualizerView::CensusVisualizerView(QWidget *parent)
    : QWidget(parent)
{
    visualizer = qobject_cast<CensusVisualizer*>(parent);
```

```

    Q_ASSERT(visualizer);
    setFocusPolicy(Qt::WheelFocus);
    setMinimumSize(minimumSizeHint());
}

```

Объект `CensusVisualizerView` создается в конструкторе `CensusVisualizer`, и в качестве родителя ему передается сам `CensusVisualizer` (стр. 250). Тем не менее мы решили сохранить указатель на `CensusVisualizer` в закрытом члене `visualizer`, чтобы иметь доступ к `CensusVisualizer`, поскольку сконструированное представление передается объекту `QScrollArea`, а тот принимает владение им на себя, то есть становится его родителем. (Можно было отказаться от этой переменной-члена и получать доступ к визуализатору с помощью конструкции `qobject_cast<CensusVisualizer*>(parent()->parent())`.)

Qt предоставляет несколько политик управления фокусом: `Qt::NoFocus` (для меток и других виджетов, допускающих только чтение), `Qt::TabFocus` (виджет может принимать фокус в результате нажатия клавиши Tab), `Qt::ClickFocus` (виджет принимает фокус при щелчке по нему мышью), `Qt::StrongFocus` (сочетание `Qt::TabFocus` и `Qt::ClickFocus`) и `Qt::WheelFocus` (`Qt::StrongFocus` плюс принятие фокуса в результате использования колесика мыши). В данном случае мы задали политику `Qt::WheelFocus` — типичный выбор для виджетов, допускающих редактирование.

Мы опустили реализацию метода `minimumSizeHint()`, потому что она почти идентична `CensusVisualizerHeader::minimumSizeHint()` (стр. 255), единственное отличие в том, что член `visualizer` встроен в класс. (Родителем `CensusVisualizerHeader` является `CensusVisualizer`, и смены родителя не происходит, поэтому и отдельная переменная-член `visualizer` не нужна.)

```

QSize CensusVisualizerView::sizeHint() const
{
    int rows = visualizer->model()
        ? visualizer->model()->rowCount() : 1;
    return QSize(visualizer->widthOfYearColumn() +
        qMax(100, visualizer->maleFemaleHeaderTextWidth()) +
        visualizer->widthOfTotalColumn(),
        visualizer->yOffsetForRow(rows));
}

```

Если модель уже задана, то мы отводим достаточное место для всех ее строк, в противном случае только для одной строки. Вертикальное смещение, возвращаемое методом `CensusVisualizer::yOffsetForRow()`, — это и есть нужная нам высота, поскольку в качестве аргумента мы передали ему количество строк в модели. Для столбцов используются фиксированные значения ширины, вычисленные на этапе конструирования `CensusVisualizer`, плюс вычисленная позднее ширина столбца «Мужчины — Женщины» (или 100 пикселей в зависимости от того, что больше).

```

bool CensusVisualizerView::eventFilter(QObject *target, QEvent *event)
{

```

```

    if (QScrollArea *scrollArea = visualizer->scrollArea()) {
        if (target == scrollArea &&
            event->type() == QEvent::Resize) {
            if (QResizeEvent *resizeEvent =
                static_cast<QResizeEvent*>(event)) {
                QSize size = resizeEvent->size();
                size.setHeight(sizeHint().height());
                int width = size.width() - (ExtraWidth +
                    scrollArea->verticalScrollBar()->sizeHint()
                        .width());
                size.setWidth(width);
                resize(size);
            }
        }
    }
    return QWidget::eventFilter(target, event);
}

```

Объект `CensusVisualizerView` был установлен в качестве фильтра событий для содержащего его объекта `QScrollArea` (стр. 250). Это означает, что любое событие, отправленное `QScrollArea`, сначала попадает к этому методу.

Нас интересует только событие `QEvent::Resize`. Оно означает, что был изменен размер области прокрутки, и значит, мы должны изменить размер `CensusVisualizerView`. Высота представления всегда поддерживается такой, чтобы хватило места для показа всех данных, а под ширину отводится все доступное пространство за вычетом места для вертикальной полосы прокрутки. Это означает, что если пользователь прокрутил представление, а затем, например, щелкнул по строке, то мы можем обрабатывать все так, как будто виден весь виджет, не забывая себе голову учетом прокрутки при определении того, по какой строке щелкнули.

Внутри реализации `eventFilter()` мы вольны, по крайней мере в принципе, делать с событием все, что угодно: изменить его, заменить другим, удалить или игнорировать. Чтобы воспрепятствовать продолжению события (неважно, сделали мы с ним что-то или нет), а также после удаления события мы должны вернуть `true`, сообщив тем самым, что оно уже обработано. Иначе возвращаем `false`. В данном случае мы воспользовались событием, но не хотим воздействовать на поведение, поэтому не изменяем хранящиеся в нем данные и в конце вызываем реализацию из базового класса.

```

void CensusVisualizerView::mousePressEvent(QMouseEvent *event)
{
    int row = static_cast<int>(event->y() /
        (QFontMetricsF(font()).height() + ExtraHeight));
    int column;
    if (event->x() < visualizer->widthOfYearColumn())
        column = Year;
    else if (event->x() < (visualizer->widthOfYearColumn() +

```

```

        visualizer->widthOfMaleFemaleColumn() / 2))
        column = Males;
    else if (event->x() < (visualizer->widthOfYearColumn() +
        visualizer->widthOfMaleFemaleColumn()))
        column = Females;
    else
        column = Total;
    visualizer->setSelectedRow(row);
    visualizer->setSelectedColumn(column);
    emit clicked(visualizer->model()->index(row, column));
}

```

Метод `QMouseEvent::y()` возвращает вертикальное смещение точки щелчка мышью от верхнего края виджета. Поскольку `CensusVisualizerView` помещен в `QScrollArea` и при этом его высота как раз достаточна для размещения всех данных – об этом мы позаботились в `eventFilter()`, – то можно работать непосредственно с вертикальным смещением, не заботясь о том, был виджет прокручен или нет. Поэтому номер строки здесь мы определяем просто путем деления смещения на высоту одной строки.

Чтобы определить номер столбца, мы анализируем горизонтальное смещение: если оно меньше ширины столбца «Год», то пользователь щелкнул по столбцу «Год»; если меньше суммы ширины столбца «Год» и половины ширины столбца «Мужчины – Женщины», то по столбцу «Мужчины», и т. д.

Определив строку и столбец, мы просим объект `CensusVisualizer` выделить их, точно зная, что это приведет к вызову метода `update()` как для представления, так и для заголовка, поэтому строка и столбец будут правильно подсвечены. И наконец, мы генерируем сигнал `clicked()`, передавая в нем модельный индекс (вычисленный моделью) выделенного элемента. Это, в свою очередь, заставляет `CensusVisualizer` сгенерировать свой собственный сигнал `clicked()` с тем же модельным индексом – к вящей пользе подключившихся объектов.

```

void CensusVisualizerView::keyPressEvent(QKeyEvent *event)
{
    if (visualizer->model()) {
        int row = Invalid;
        int column = Invalid;
        if (event->key() == Qt::Key_Left) {
            column = visualizer->selectedColumn();
            if (column == Males || column == Total)
                --column;
            else if (column == Females)
                column = Year;
        }
        ...
        else if (event->key() == Qt::Key_Up)
            row = qMax(0, visualizer->selectedRow() - 1);
        else if (event->key() == Qt::Key_Down)

```

```

        row = qMin(visualizer->selectedRow() + 1,
                   visualizer->model()->rowCount() - 1);
        row = row == Invalid ? visualizer->selectedRow() : row;
        column = column == Invalid ? visualizer->selectedColumn() : column;
        if (row != visualizer->selectedRow() ||
            column != visualizer->selectedColumn()) {
            QModelIndex index = visualizer->model()->index(row, column);
            visualizer->setCurrentIndex(index);
            emit clicked(index);
            return;
        }
    }
    QWidget::keyPressEvent(event);
}

```

Этот обработчик событий обеспечивает навигацию внутри представления с помощью клавиш со стрелками.

Внутри `CensusVisualizer` мы отслеживаем выделенную строку и столбец, но столбцы с данными о количестве мужчин и женщин визуально, а значит, и с точки зрения пользователя, представляются как единое целое. Поэтому, когда пользователь нажимает клавишу со стрелкой влево при активном столбце «Мужчины» или «Женщины», мы переходим в столбец «Год». Если текущим уже является столбец «Год», то мы не делаем ничего, а если столбец «Всего», то переходим в столбец «Женщины». Обработка нажатия клавиши со стрелкой вправо производится аналогично (поэтому код опущен): если текущим является столбец «Мужчины» или «Женщины», то мы переходим в столбец «Всего». Если же текущим является столбец «Год», то мы переходим в столбец «Мужчины», а если столбец «Всего», то ничего не делаем.

Если пользователь нажимает клавишу со стрелкой вверх, то мы уменьшаем номер текущей строки на единицу – или ничего не делаем, если текущей является первая строка. И аналогично, если пользователь нажимает клавишу со стрелкой вниз, то мы увеличиваем номер текущей строки на единицу – или ничего не делаем, если текущей является последняя строка.

Если новая строка или столбец отличны от текущих, то мы переустанавливаем номера выделенных строки и столбца. В результате и для представления, и для заголовка будет вызван метод `update()`, который заодно сделает выделенный элемент видимым на экране. Кроме того, мы генерируем сигнал `clicked()`, передавая в нем модельный индекс выделенного элемента.

Если выделен какой-то новый элемент, то в конце мы *не* должны вызывать реализацию из базового класса, потому что уже обработали нажатие клавиши сами и не хотим, чтобы событие было передано области прокрутки. Дело в том, что область прокрутки тоже обрабатывает клавиши со стрелками, интерпретируя их как запрос на прокрутку, но нам это не нужно, так как мы уже все сделали. И наоборот, если мы не обра-

ботали нажатие какой-то клавиши, то должны вызвать метод базового класса, чтобы он восполнил упущение.

Сравните этот метод с обработчиком события мыши, где мы установили номера текущих строки и столбца, но не позаботились о том, чтобы сделать ячейку видимой, – поскольку раз пользователь сумел по ней щелкнуть, значит, она и так видна. Однако в данном случае пользователь мог, к примеру, нажать клавишу со стрелкой вниз, находясь на последней видимой строке, поэтому необходимо вызвать `QScrollArea::ensureVisible()` (это делает метод `CensusVisualizer::setCurrentIndex()`; стр. 253), чтобы прокрутить представление до нужного положения.

Поддержка клавиш `Home`, `End`, `Page Up` и `Page Down` устроена по тому же принципу, поэтому ее реализация оставлена в качестве упражнения. (При обработке нажатия клавиш `Page Up` и `Page Down` принято смещаться вверх или вниз на количество видимых строк минус единица, так, чтобы одна строка оставалась в качестве контекста, помогающего пользователю сориентироваться.)

Только что рассмотренные методы `eventFilter()`, `mousePressEvent()` и `keyPressEvent()` обеспечивают требуемое поведение представления. Теперь перейдем к методу `paintEvent()` и вызываемому из него закрытому вспомогательному методу, чтобы понять, как формируется внешний вид представления.

```
void CensusVisualizerView::paintEvent(QPaintEvent *event)
{
    if (!visualizer->model())
        return;
    QFontMetricsF fm(font());
    const int RowHeight = fm.height() + ExtraHeight;
    const int MinY = qMax(0, event->rect().y() - RowHeight);
    const int MaxY = MinY + event->rect().height() + RowHeight;
    QPainter painter(this);
    painter.setRenderHints(QPainter::Antialiasing|
                           QPainter::TextAntialiasing);
    int row = MinY / RowHeight;
    int y = row * RowHeight;
    for (; row < visualizer->model()->rowCount(); ++row) {
        paintRow(&painter, row, y, RowHeight);
        y += RowHeight;
        if (y > MaxY)
            break;
    }
}
```

В начале этого метода мы вычисляем некоторые константы, в том числе высоту, отводимую под каждую строку, и минимальную и максимальную ординаты области рисования (минус или плюс высота одной строки соответственно, чтобы частично видимые строки все равно прорисовывались).

Поскольку виджет располагается внутри `QScrollArea` и его высота всегда в точности такова, чтобы были показаны все элементы, то нам *необязательно* вычислять смещения и самим разбираться в том, что видно на экране, а что нет. Однако эффективности ради мы рисуем только видимые элементы.

В переданном событии рисования хранится объект `QRect`, описывающий прямоугольник, который следует перерисовать. Если виджет невелик, то мы обычно игнорируем эту информацию и перерисовываем прямоугольник целиком. Однако виджет для визуализации модели может содержать очень много данных, поэтому, чтобы он работал эффективно, желательно перерисовывать только ту область, которая в этом нуждается. Поэтому, вычислив константы, мы конфигурируем объект `painter` и находим номер и ординату первой перерисовываемой строки. (Может возникнуть искушение инициализировать `y` значением `MinY`, но `MinY` обычно не то же самое, что `row * RowHeight`, из-за — желательного — отбрасывания дробной части при вычислении выражения `MinY / RowHeight`.)

Все подготовив, мы запускаем цикл обхода строк модели, начиная с первой видимой, в котором рисуем строки до тех пор, пока ордината не окажется за пределами перерисовываемого прямоугольника; в этот момент мы выходим из цикла. Таким образом, мы перерисовали только видимые строки плюс две дополнительных; это может дать существенный выигрыш во времени, если количество строк исчисляется тысячами и десятками тысяч.

```
void CensusVisualizerView::paintRow(QPainter *painter, int row,
                                     int y, const int RowHeight)
{
    paintYear(painter, row,
              QRect(0, y, visualizer->widthOfYearColumn(), RowHeight));
    paintMaleFemale(painter, row,
                   QRect(visualizer->widthOfYearColumn(), y,
                        visualizer->widthOfMaleFemaleColumn(), RowHeight));
    paintTotal(painter, row,
              QRect(visualizer->widthOfYearColumn() +
                   visualizer->widthOfMaleFemaleColumn(), y,
                   visualizer->widthOfTotalColumn(), RowHeight));
}
```

Этот метод просто создает подходящий прямоугольник и вызывает нужный метод рисования — свой для каждого столбца.

```
void CensusVisualizerView::paintYear(QPainter *painter, int row,
                                       const QRect &rect)
{
    paintItemBackground(painter, rect,
                       row == visualizer->selectedRow() &&
                       visualizer->selectedColumn() == Year);
    painter->drawText(rect,
```

```

        visualizer->model()->data(
            visualizer->model()->index(row, Year)).toString(),
        QTextOption(Qt::AlignCenter));
    }

```

После того как фон закрашен, остается только вывести текст элемента. Текст мы получаем от модели и располагаем его в центре столбца.

Метод `CensusVisualizerView::paintTotal()` очень похож на предыдущий (поэтому мы его не приводим), отличие лишь в том, что итоговая цифра выравнивается по правому краю.

```

void CensusVisualizerView::paintItemBackground(QPainter *painter,
        const QRect &rect, bool selected)
{
    painter->fillRect(rect, selected ? palette().highlight()
                                    : palette().base());
    visualizer->paintItemBorder(painter, palette(), rect);
    painter->setPen(selected ? palette().highlightedText().color()
                          : palette().windowText().color());
}

```

Цвета фона и текста зависят от того, является элемент выделенным или нет. Этот метод рисует фон и рамку и настраивает перо, чтобы вызывающая программа могла воспользоваться им для вывода текста.

Метод `paintMaleFemale()` несколько длиннее прочих, поэтому мы разобьем его на три части.

```

void CensusVisualizerView::paintMaleFemale(QPainter *painter,
        int row, const QRect &rect)
{
    QRect rectangle(rect);
    QLocale locale;
    int males = locale.toInt(visualizer->model()->data(
        visualizer->model()->index(row, Males)).toString());
    int females = locale.toInt(visualizer->model()->data(
        visualizer->model()->index(row, Females)).toString());
    qreal total = males + females;
    int offset = qRound(
        ((1 - (total / visualizer->maximumPopulation())) / 2) *
        rectangle.width());
}

```

Сначала мы выясняем, каковы в данной строке значения для мужчин и женщин по отдельности и в сумме. (Использование класса `QLocale` для преобразования локализованных строк в числа обсуждалось на стр. 252.) Затем вычисляем, сколько места следует отвести под всю закрашенную полосу, и, зная эту величину, определяем, сколько места оставить слева и справа, чтобы полоска оказалась правильно расположена (отцентрирована) внутри имеющегося прямоугольника.

```

painter->fillRect(rectangle,
    (row == visualizer->selectedRow() &&

```



```
(visualizer->selectedColumn() == Females ||
 visualizer->selectedColumn() == Males))
? palette().highlight() : palette().base());
```

Первым делом мы закрашиваем фон цветом, зависящим от того, выделен ли столбец «Мужчины – Женщины».

```
visualizer->paintItemBorder(painter, palette(), rectangle);
rectangle.setLeft(rectangle.left() + offset);
rectangle.setRight(rectangle.right() - offset);
int rectY = rectangle.center().y();
painter->fillRect(rectangle.adjusted(0, 1, 0, -1),
 maleFemaleGradient(rectangle.left(), rectY,
 rectangle.right(), rectY, males / total));
}
```

Далее мы рисуем рамку вокруг элемента и изменяем размер доступного прямоугольника – возможно, уменьшаем его, – чтобы размер и положение отвечали цели – рисованию закрашенной полоски. Наконец, мы рисуем саму полоску – немного уменьшив высоту, – применяя градиентную заливку с переходом цвета от темно-зеленого к светло-зеленому (слева направо) для мужской половины и от светло-красного к темно-красному (слева направо) – для женской.

```
QLinearGradient CensusVisualizerView::maleFemaleGradient(
    qreal x1, qreal y1, qreal x2, qreal y2, qreal crossOver)
{
    QLinearGradient gradient(x1, y1, x2, y2);
    QColor maleColor = Qt::green;
    QColor femaleColor = Qt::red;
    gradient.setColorAt(0, maleColor.darker());
    gradient.setColorAt(crossOver - 0.001, maleColor.lighter());
    gradient.setColorAt(crossOver + 0.001, femaleColor.lighter());
    gradient.setColorAt(1, femaleColor.darker());
    return gradient;
}
```

Этот метод приведен только для полноты картины. Он создает линейный градиент с переходом от темного к светлому оттенку одного цвета, а затем от светлого к темному оттенку другого цвета, причем смена цветов происходит в указанной точке. Точка смены вычисляется вызывающей программой как `males / total`; тем самым гарантируется, что размеры ширины мужской и женской частей будут пропорциональны их числовым значениям.

В Qt имеются также классы `QConicalGradient` и `QRadialGradient` с аналогичным API.

На этом мы закончили рассмотрение класса `CensusVisualizer` и агрегируемых им классов `CensusVisualizerHeader` и `CensusVisualizerView`, которые и выполняют основную работу. Создание подобных классов – идеальное решение в случае, когда некоторую модель требуется визуализировать

особым способом, а элементы следует объединить, то есть ни пользовательского делегата, ни пользовательского представления, основанного на API класса `QAbstractItemView`, будет недостаточно.

Итак, мы ознакомились с классами `TiledListView` и `CensusVisualizer`. Класс `TiledListView` гораздо короче, потому что ему не надо показывать названия столбцов, а часть своей функциональности он может позаимствовать у базового класса. Если требуется представить данные модели каким-то уникальным способом, например графически, или хочется визуально объединить несколько элементов модели, то пользовательского делегата недостаточно, и приходится прибегать к пользовательскому представлению. Подход, принятый в классе `CensusVisualizer`, позволяет получить полный контроль, а реализовывать нужно будет только те возможности, который действительно необходимы. Решив же создать подкласс `QAbstractItemView`, мы тоже получаем полный контроль, а в придачу кое-какую готовую функциональность и больше шансов на повторное использование, однако обязаны как минимум реализовать все чисто виртуальные методы, а обычно еще и методы, перечисленные в табл. 6.1 (стр. 232).

Это была последняя из четырех глав, посвященных архитектуре модель/представление в Qt. Вообще говоря, проще всего начать с создания подкласса `QStandardItemModel` (или `QStandardItem`), чтобы получить возможность сериализации и десериализации данных. Впоследствии, если возникнет необходимость, всегда можно будет подменить эту модель пользовательской. Аналогично, начинать визуализацию данных модели лучше всего с применения одного из стандартных представлений Qt. Если же требуется какое-то специальное внешнее представление или редактирование элементов, то самое простое – написать пользовательский делегат. Но если сочетание стандартного представления с пользовательским делегатом не позволяет визуализировать данные требуемым образом, то приходится писать пользовательское представление; два подхода к решению этой задачи описаны в настоящей главе.

7

Многопоточное программирование с модулем QtConcurrent

- Выполнение функции в потоке
- Фильтрация и отображение в потоках

Многопоточное программирование весьма популярно, а иногда и очень полезно. Однако многие программисты применяют его без достаточных оснований, а в результате приложения получаются избыточно сложными (см. врезку «Такая противоречивая многопоточность» чуть ниже). В этой главе предполагается знакомство с основами многопоточного программирования в Qt; наша задача здесь – показать, как следует использовать поддержку многопоточности, а не предложить учебное руководство по этой теме¹.

Прежде чем нырнуть в бурные многопоточные воды, стоит сделать паузу и поразмышлять о некоторых связанных с этим вопросах. Как правило, мы применяем многопоточность, чтобы повысить производительность, но при этом ко многим вещам приходится подходить иначе, чем при написании однопоточных приложений.

Вообще-то, нет никакой гарантии, что наличие нескольких потоков действительно повысит производительность. Например, если количество потоков превышает количество процессорных ядер, то производительность может даже уменьшиться, поскольку весь выигрыш съедается возросшей конкуренцией за ресурсы. А иногда алгоритм, оптималь-

¹ Читатели, знакомые с многопоточным программированием вообще, но не с его реализацией в Qt, могут обратиться к документации на странице qt.nokia.com/doc/threads.html или к главе «Многопоточная обработка» в книге «Qt 4: программирование GUI на C++».

ный для однопоточного приложения, оказывается не самым эффективным для многопоточного. Поэтому, желая убедиться в том, что мы действительно обеспечили наилучшую производительность, нужно написать разные реализации и профилировать их на предмет сравнения производительности – на той же аппаратуре и в той же конфигурации ПО, какие будут у потенциальных пользователей.

Такая противоречивая многопоточность

В основном благодаря встроенной поддержке многопоточности в Java, а с недавних пор также появлению на рынке многоядерных процессоров интерес к многопоточным программам в последние годы значительно вырос.

Но, несмотря на популярность, многопоточная обработка остается внутренне противоречивой. Она может существенно увеличить сложность программ и сильно затруднить отладку и сопровождение по сравнению с однопоточными приложениями. И не всегда возможно разбить алгоритм на части, так, чтобы оправдать использование многопоточности. Кроме того, достижению выигрыша в производительности иногда мешают сами издержки многопоточной обработки, да и вообще допустить ошибку при таком подходе гораздо проще.

Ведущий разработчик из компании Sun Тим Брей (Tim Bray) говорит: «После того как лучшие и ярчайшие умы потратили десять лет на создание и отладку инфраструктуры многопоточности в Java и .NET, начинает складываться впечатление, что эта идея оказалась порочной» (параграф «Processors» в блоге Тима Брея по адресу www.tbray.org/ongoing/When/200x/2008/04/24/Inflection). И его голос не одинок. Один из отцов-основателей науки о компьютерах, Дональд Кнут, пишет: «Я совершенно не удивлюсь, если вся идея многопоточной обработки окажется пшиком» (интервью с Дональдом Кнутом на странице www.informit.com/articles/article.aspx?p=1193856).

На поверхности лежат две основные проблемы. Одна заключается в том, что использование многопоточности требует написания значительного объема кода для поддержки самой технологии, не имеющего отношения к решению поставленной задачи. Что еще хуже, этот код может оказаться очень нетривиальным, правильно написать его трудно, а отлаживать и того труднее. Другая проблема состоит в том, что на аппаратном уровне существует много разных подходов к реализации параллелизма, в каждом случае авторы компиляторов должны применять различные приемы, а по мере развития все это может оказаться устаревшим из-за появления новых аппаратных решений.

Но есть и хорошие новости. Существует технология программной транзакционной памяти, которая позволяет осуществлять многопоточную обработку на высоком уровне, не обременяя программиста низкоуровневыми деталями (блокировкой и разблокировкой). Такие библиотеки уже разрабатываются для C++. И в Qt тоже есть модуль *QtConcurrent* (с ним мы ознакомимся в этой главе), предназначенный для организации высокоуровневого доступа к многопоточности и берущий на себя заботу обо всех низкоуровневых деталях. В числе других решений можно назвать языки Erlang и Go, а также технологию Grand Central Dispatch компании Apple.

И разумеется, существует принципиально иной подход, который позволяет воспользоваться наличием нескольких ядер без большинства недостатков, присущих многопоточности, но и без некоторых его достоинств. Это многопроцессорная обработка, которая, правда, страдает и от ряда собственных недостатков. Смысл ее заключается в передаче части работы отдельным процессам, например, с помощью класса *QProcess*. Хотя этот подход избавляет программиста от части неочевидных подводных камней и от написания дополнительного кода для поддержки многопоточности, ответственность за все межпроцессные коммуникации по-прежнему лежит на нас.

Но оставим подводные камни. Если вы решили написать многопоточное приложение, то библиотека Qt готова предоставить вам всяческую поддержку. В частности, в версии Qt 4.4 появился класс *QRunnable* и пространство имен *QtConcurrent*, предлагающие высокоуровневый API для поддержки многопоточности, так что программисту не нужно вникать в детали низкоуровневого API класса *QThread* и связанных с ним классов. Этот высокоуровневый API освобождает нас от многих забот, обычно ассоциируемых с многопоточным программированием (хотя кое-какие меры предосторожности принимать все же надо).

Класс *QRunnable* и функция *QtConcurrent::run()* прекрасно подходят в случае, когда нужно выполнить фоновую обработку в одном или нескольких вспомогательных потоках, не прибегая ко всему арсеналу гибких средств, предоставляемому классом *QThread*. Они будут рассмотрены в первом разделе этой главы.

В пространстве имен *QtConcurrent* имеются также функции для фильтрации (*filter*), отображения (*map*) и редукции (*reduce*, свертка) – что это означает, мы объясним при рассмотрении этих функций во втором разделе. Они идеальны для ситуаций, когда необходимо обработать очень много элементов. К сожалению, мы не можем воспользоваться ими для обработки элементов модели *QAbstractItemModel* или сцены

QGraphicsScene, потому что Qt не поддерживает блокировку моделей, сцен и содержащихся в них элементов. Однако мы покажем, как обойти эту проблему ценой дополнительной памяти и процессорного времени.

Иногда бывает разумно обратиться к низкоуровневому API. В следующей главе мы будем рассматривать работу с классом QThread. Потенциально это самый трудный способ организации нескольких потоков в Qt, но в награду вы получаете возможность предельно точного контроля.

В этой и следующих главах мы попытаемся свести к минимуму опасности и сложности многопоточной обработки. Основной способ достижения этой цели – попытаться вообще избежать блокировок – например, поручая каждому потоку независимую часть работы. Если же блокировка все-таки необходима, то мы стараемся минимизировать ее или сделать прозрачной – например, за счет создания классов, которые сами управляют инкапсулированными в них блокировками, не обременяя этим клиентов.

В Qt есть также несколько классов еще более низкого уровня, например QAtomicInt и QAtomicPointer. Они идеально подходят для создания потокобезопасных структур данных и прочих низкоуровневых компонентов, но выходят за рамки этой книги. (Некоторые мысли о том, как ими пользоваться, можно почерпнуть в указанной в списке литературы книге «The Art of Multiprocessor Programming», см. стр. 532.)

API многопоточной обработки в Qt включает также классы QSemaphore, QThreadStorage и QWaitCondition. Чаще всего они используются в сочетании с подклассами QThread, хотя в этой и в следующей главах мы с ними не встретимся, потому что отдадим предпочтение другим классам и приемам, например типу `volatile bool`. (О применении классов QSemaphore, QThreadStorage и QWaitCondition см. главу «Многопоточная обработка» в книге «Qt 4: программирование GUI на C++», указанной в списке литературы.)

Ключевым словом `volatile` помечается переменная, которая может измениться незаметно для программы, и значит, компилятор ни в коем случае не должен устранять ее в результате оптимизации или кэшировать. Это полезно, когда переменная размещается по адресу памяти, который может модифицироваться вне программы, как, например, аппаратный порт. Но, кроме того, переменные типа `volatile bool` бывают весьма удобны в многопоточных программах в случае, когда значение переменной может быть изменено в одном потоке, а прочитано в другом. Важно отметить, что слово `volatile` не годится для многопоточной работы с другими типами данных (даже `int`), поскольку нельзя исключить возможность обновления двух разных байтов разными потоками. Но с типом `bool` его можно применять безопасно¹. Помимо конструкции

¹ См., например, статью Андрея Александреску «`volatile`—Multithreaded Programmer's Best Friend» по адресу www.ddj.com/cpp/184403766.

`volatile bool`, в этой книге будут приведены примеры использования классов `QMutex`, `QMutexLocker`, `QReadWriteLock`, `QReadLocker` и `QWriteLocker`, а из пространства имен `QtConcurrent` – функции `QFuture` и `QFutureWatcher`.

Вообще говоря, многопоточность дает максимальную отдачу, когда расходы на подготовку и запуск отдельных потоков перекрываются выигрышем от распределения работы между несколькими ядрами или процессорами. Если оставить в стороне очевидные применения в параллельных алгоритмах, то эффект от использования многопоточности особенно нагляден, когда имеется по крайней мере один (а потенциально и несколько) ресурсоемкий участок обработки, который может быть полностью или по большей части независимым.

Один важный случай применения многопоточности встречается в контексте графических интерфейсов. Если требуется выполнить какую-то длительную операцию, не «подвешивая» при этом пользовательский интерфейс, то можно создать дополнительный поток и поручить ему обработку. Для сетевых взаимодействий это необязательно, потому что `Qt` и так реализует доступ к сети асинхронно, но когда речь идет о реализации какой-то обработки собственными силами, наличие одного или нескольких вспомогательных потоков может оказаться весьма кстати. В некоторых случаях можно воспользоваться и менее накладным подходом – организацией локального цикла обработки событий (мы встречались с этим решением в главе 2, стр. 91).

Выполнение функции в потоке

В случаях, когда количество подлежащих обработке элементов невелико, но обработка каждого занимает много времени, часто удобно делать это в отдельном потоке выполнения, чтобы сохранить работоспособное состояние пользовательского интерфейса. И наоборот, если требуется обработать очень много элементов, но их можно как-то сгруппировать (или поместить в очередь), то идея распределить работу между несколькими потоками выглядит весьма привлекательно.

Существует четыре основных способа распределить обработку между несколькими потоками или процессами: запустить отдельные процессы с помощью класса `QProcess` (например, несколько экземпляров автономной «рабочей» программы); использовать `QtConcurrent::run()` для запуска функций или методов во вспомогательных потоках, взятых из глобального пула потоков `Qt`; создать объекты `QRunnable` и запустить их во вспомогательных потоках, взятых из глобального пула потоков, или создать объекты `QThread` и выполнять их как вспомогательные потоки. В этом разделе мы рассмотрим использование `QtConcurrent::run()` и `QRunnable`, а в следующей главе – применение `QThread`.

Работать с функцией `QtConcurrent::run()` несложно: создаем функцию или метод, который будет заниматься обработкой, и передаем его в `QtConcurrent::run()` для выполнения. Если желательно задействовать не-

сколько вспомогательных потоков, то можно передавать одну и ту же функцию (обычно с разными аргументами) несколько раз. Использование класса `QRunnable` аналогично. Создаем его подкласс и помещаем всю свою обработку в реализацию чисто виртуального метода `run()`. Затем передаем методу `QThreadPool::start()` столько экземпляров этого подкласса, сколько нужно вспомогательных потоков.

У обработки во вспомогательных потоках с помощью `QtConcurrent::run()` или `QRunnable` есть два потенциальных недостатка по сравнению с использованием класса `QThread`. Во-первых, отсутствует поддержка сигналов и слотов, так что нет никакого встроенного механизма коммуникации (например, для индикации хода выполнения). Во-вторых, мы не получаем уведомления о завершении обработки; если это необходимо, приходится самим искать выход. Впрочем, решить обе проблемы – а заодно поддержать останов потока – совсем просто, как мы убедимся в конце этого раздела.

В этом разделе мы напишем приложение `Image2Image` (`image2image`), которое будет искать в указанном каталоге графические файлы и для каждого создавать копию изображения в указанном формате (например, *bmp*, *tiff* и т. д.). Внешний вид окна программы показан на рис. 7.1¹.

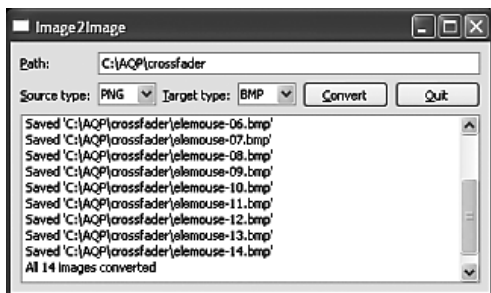


Рис. 7.1. Приложение `Image2Image`

Для того чтобы конвертировать изображение, его нужно считать в память в исходном формате, а затем сохранить в другом – это сочетание дискового ввода/вывода и процессорной обработки. Мы решили использовать в приложении несколько вспомогательных потоков, каждому из которых будет передан свой список файлов для конвертирования. Поскольку у каждого вспомогательного потока имеется независимая часть работы, то коммуникация между ними не нужна, так что можно обойтись без блокировок. Разумеется, мы хотели бы уведомлять глав-

¹ Кстати, со строкой для ввода пути ассоциирован объект `QCompleter`, который выводит раскрывающийся список подходящих каталогов, чтобы пользователю пришлось меньше печатать. Эта тема рассматривается в главе 9, стр. 349.

ный поток (интерфейс пользователя) программы о ходе работы, но для этого применим способ, который возлагает заботу о возможных блокировках на Qt.

Мы написали приложение так, чтобы в зависимости от некоторой константы препроцессора, определенной в директиве `#define`, оно могло использовать либо функцию `QtConcurrent::run()`, либо класс `QRunnable`. Использование `QtConcurrent::run()` мы рассмотрим в следующем подразделе, а сейчас посмотрим, как работать с `QRunnable`. В обоих подразделах объяснение кода, относящегося к пользовательскому интерфейсу, сведено к минимуму, все внимание уделяется вопросам многопоточной обработки.

Хотя в обоих случаях применяются одни и те же инфраструктура главного окна и средства управления отменой, техника извещения о ходе выполнения различна. В примере `QtConcurrent::run()` для этого отправляется пользовательское событие, а в примере `QRunnable` вызывается слот. Однако, поскольку `QRunnable` не является подклассом `QObject`, вызывать слот приходится без использования `emit` (то есть не посылая сигнал естественным для Qt способом). Мы просто хотели продемонстрировать два разных приема; в принципе, никто не мешает в обоих случаях использовать события или вызывать слоты.

Помимо методов и виджетов, в классе главного окна определены три закрытые переменные:

```
int total;
int done;
volatile bool stopped;
```

В переменной `total` хранится общее число подлежащих обработке изображений, в переменной `done` – количество уже конвертированных изображений, а в булевой переменной `stopped` – признак, используемый для того, чтобы уведомить вспомогательные потоки о том, что пользователь пожелал прервать выполнение. Отметим, что и для инициирования процесса конвертирования, и для его прерывания служит одна и та же кнопка (`convertOrCancelButton`).

Выбрав каталог, а также исходный и конечный формат, пользователь может начать конвертирование, нажав кнопку `Convert`. После нажатия эта кнопка заменяется на `Cancel`, и теперь ее нажатие прекращает конвертирование. Кнопка соединена со слотом `convertOrCancel()`.

```
void MainWindow::convertOrCancel()
{
    stopped = true;
    if (QThreadPool::globalInstance()->activeThreadCount())
        QThreadPool::globalInstance()->waitForDone();
    if (convertOrCancelButton->text() == tr("&Cancel")) {
        updateUi();
        return;
    }
}
```

```

    }
    QString sourceType = sourceTypeComboBox->currentText();
    QStringList sourceFiles;
    QDirIterator i(directoryEdit->text(), QDir::Files|QDir::Readable);
    while (i.hasNext()) {
        const QString &filenameAndPath = i.next();
        if (i.fileInfo().suffix().toUpper() == sourceType)
            sourceFiles << filenameAndPath;
    }
    if (sourceFiles.isEmpty())
        AQP::warning(this, tr("No Images Error"),
                    tr("No matching files found"));
    else {
        logEdit->clear();
        convertFiles(sourceFiles);
    }
}

```

В начале слота переменной `stopped` присваивается значение `true`, чтобы уведомить все работающие вспомогательные потоки о необходимости остановиться. Затем проверяется, остались ли еще вспомогательные потоки в глобальной очереди потоков Qt, и если да, то слот ожидает (блокируя программу), пока все они не завершатся.

Метод `QThreadPool::globalInstance()` возвращает указатель на глобальный объект `QThreadPool`, а `QThreadPool::activeThreadCount()` — число активных потоков в пуле — оно, разумеется, может быть равно 0. Метод `QThreadPool::waitForDone()` ожидает завершения *всех* работающих потоков, взятых из пула, поэтому он может блокировать пользовательский интерфейс на длительное время. Чтобы избежать такого развития событий, мы должны сообщить всем потокам о необходимости прекратить работу, прежде чем приступить к ожиданию. В данной программе мы для этого просто устанавливаем для переменной `stopped` значение `true`.

Если пользователь отменяет выполнение, то мы вызываем метод `updateUi()` (не показан), чтобы изменить текст кнопки `Cancel` на `Convert`, и возвращаем управление, поскольку в этот момент все вспомогательные потоки уже остановлены.

Когда пользователь нажимает кнопку `Convert`, мы составляем список файлов в выбранном каталоге, суффикс которых соответствует выбранному исходному формату. Если список пуст, то мы сообщаем об этом пользователю и возвращаем управление. Если же список не пуст, то мы очищаем журнал (виджет `QPlainTextEdit`, сконфигурированный только для чтения) и вызываем метод `convertFiles()`, который и отвечает за конвертирование.

Мы включили две реализации метода `convertFiles()`: с использованием функции `QtConcurrent::run()` (показан в следующем подразделе) и с применением класса `QRunnable` (показан далее). В обоих случаях, помимо

самого метода `convertFiles()`, приведены соответствующие инфраструктурные методы.

Функция `QtConcurrent::run()`

Функция `QtConcurrent::run()` принимает указатель на функцию и один или несколько необязательных аргументов, которые будут переданы ей при вызове, и выполняет эту функцию в отдельном потоке, взятом из глобального пула потоков Qt. Она имеет следующую сигнатуру:

```
QFuture<T> run(Function, ...)
```

Аргумент `Function` должен быть указателем на функцию (или функтором), которая возвращает объект типа `T`. Многоточием (...) обозначается список аргументов переменной длины (содержащий 0 или более аргументов). Если аргументы заданы, то они передаются функции `Function` при вызове, поэтому должны соответствовать сигнатуре `Function`.

А теперь вернемся к приложению `image2image` и посмотрим, как используется `QtConcurrent::run()` на практике.

```
void MainWindow::convertFiles(const QStringList &sourceFiles)
{
    stopped = false;
    updateUi();
    total = sourceFiles.count();
    done = 0;
    const QVector<int> sizes = AQP::chunkSizes(sourceFiles.count(),
        QThread::idealThreadCount());
    int offset = 0;
    foreach (const int chunkSize, sizes) {
        QtConcurrent::run(convertImages, this, &stopped,
            sourceFiles.mid(offset, chunkSize),
            targetTypeComboBox->currentText());
        offset += chunkSize;
    }
    checkIfDone();
}
```

Сначала мы устанавливаем для переменной `stopped` значение `false`, а затем вызываем метод `updateUi()` (не показан), который превращает кнопку `Convert` в `Cancel`, изменяя надпись на ней. Далее мы записываем в переменную `total` общее количество файлов в списке, а переменной `done` присваиваем значение 0, поскольку еще ни один файл не конвертирован.

Можно было бы написать функцию для конвертирования одного графического файла и вызывать `QtConcurrent::run()`, передавая имя функции и имя очередного обрабатываемого файла. При этом потенциально можно создать столько рабочих потоков, сколько есть файлов в списке. Для очень больших файлов такой подход, возможно, и имеет смысл, но когда файлов много, то затраты на создание и обслуживание такого

большого количества потоков (особенно в Windows) вполне могут свести на нет весь выигрыш от распределения работы между несколькими потоками.

К счастью, метод `QThread::idealThreadCount()` возвращает оптимальное количество рабочих потоков для данного компьютера – принимая во внимание операционную систему и число процессоров и ядер. При наличии всего одного процессора и ядра он вернет 1, а если процессоров и ядер несколько, то большее значение. Маловероятно, что эта величина будет в точности равна количеству подлежащих обработке файлов, поэтому мы должны разделить всю работу на части так, чтобы каждому потоку (в предположении, что их несколько) досталось примерно равное число файлов. (Разумеется, разбиение работы исходя из числа файлов – не всегда оптимальное решение, например, если имеется список из двадцати файлов, причем первые десять очень велики, а последние десять совсем маленькие.)

Чтобы распределить работу между оптимальным количеством вспомогательных потоков, мы сначала вызываем функцию `AQP::chunkSizes()` (не показана, но включена в модуль `aqp.{hpp,cpp}` в составе исходного кода, прилагаемого к книге) – она получает число элементов в контейнере (в данном случае число файлов) и количество нужных нам порций (в данном случае идеальное число вспомогательных потоков), а возвращает вектор, содержащий размеры порций, которые в сумме дают общее число элементов и распределены максимально равномерно. Например, если в списке 97 элементов, а число порций равно 1 (один поток), то функция вернет вектор [97]; если число порций равно 2, то получим вектор [49, 48]; если 3 – то вектор [33, 32, 32], и т. д.

Получив вектор размеров, мы обходим его (количество итераций будет равно 1, если идеальное количество потоков равно 1, поскольку в этом случае есть всего один размер порции) и для каждого размера вызываем `QtConcurrent::run()`. При этом мы передаем пять аргументов – функцию `convertImages()` (она будет выполняться в потоке) и четыре аргумента, передаваемых ей при вызове, а именно: указатель на главное окно (`this`), необходимый для взаимодействия с ним, указатель на булеву `volatile`-переменную `stopped`, чтобы мы могли узнать о том, что пользователь отменил операцию, часть полного списка файлов и суффикс конечных файлов. Вызовы функции `QtConcurrent::run()` неблокирующие, и сразу после вызова функция `convertImages()` начинает работать в отдельном потоке, получив предназначенные ей аргументы¹.

Метод `QStringList::mid()` (а точнее, `QList<T>::mid()`) принимает смещение `offset` и необязательный счетчик `count` и возвращает часть списка, со-

¹ Выше (на стр. 272) мы отмечали, что использование переменных типа `volatile bool` безопасно относительно потоков, но к другим типам аналогичная техника неприменима.

держущую *count* элементов, начиная с элемента с индексом *offset*, или все элементы, начиная с *offset*, если величина *offset + count* больше общего числа элементов в списке или аргумент *count* не задан.

Запустив все рабочие потоки, мы вызываем слот `checkIfDone()` (будет рассмотрен ниже), который проверяет, завершилась ли обработка. Функция `QtConcurrent::run()` возвращает объект `QFuture<T>`, который, если передать его объекту `QFutureWatcher<T>`, позволяет обнаружить факт завершения рабочего потока. О том, как работать с объектами `QFuture<T>`, мы узнаем в следующем разделе, а пока просто будем игнорировать значение, возвращенное `QtConcurrent::run()`, и опрашивать состояние – просто чтобы показать, что такая техника тоже возможна.

```
void convertImages(QObject *receiver, volatile bool *stopped,
                  const QStringList &sourceFiles, const QString &targetType)
{
    foreach (const QString &source, sourceFiles) {
        if (*stopped)
            return;
        QImage image(source);
        QString target(source);
        target.chop(QFileInfo(source).suffix().length());
        target += targetType.toLowerCase();
        if (*stopped)
            return;
        bool saved = image.save(target);
        QString message = saved
            ? QObject::tr("Saved '%1'")
              .arg(QDir::toNativeSeparators(target))
            : QObject::tr("Failed to convert '%1'")
              .arg(QDir::toNativeSeparators(source));
        QApplication::postEvent(receiver,
                                new ProgressEvent(saved, message));
    }
}
```

Эта функция выполняется в одном или нескольких рабочих потоках, каждый раз со своим собственным списком обрабатываемых файлов. Перед тем как начать дорогостоящую операцию (загрузку или сохранение графического файла), она проверяет, не потребовал ли пользователь прекратить работу, и если это так, то функция возвращает управление, и поток, в котором она выполнялась, становится неактивным.

Сама обработка несложна: для каждого файла в списке мы создаем объект типа `QImage` (конструктор считывает указанный графический файл), затем формируем имя конечного файла и сохраняем в нем изображение. Метод `QImage::save()` возвращает булевый флаг, индицирующий успех или неудачу.

Статический метод `QDir::toNativeSeparators()` принимает путь или путь и имя файла и возвращает строку, в которой каталоги разделены сим-

волом, возвращающим метод `QDir::separator()` (то есть символом `\` в Windows и `/` в UNIX). В исходном коде обычно удобнее использовать символ `/`, потому что его не нужно экранировать в строках, а Qt понимает его на любой платформе. Но если мы хотим показывать пути пользователям, то лучше представлять их в виде, соответствующем той платформе, на которой работает приложение.

Мы хотим уведомлять пользователя о завершении обработки каждого файла. Проще всего это сделать, обратившись к слоту в вызывающем виджете, – этот прием мы рассмотрим в следующем подразделе. А здесь воспользуемся чуть менее удобным способом – отправим пользовательское событие, просто чтобы показать, как это делается.

Сначала мы создаем строку сообщения, а потом отправляем ее вместе с флагом успеха объекту-получателю (в данном случае главному окну), обернув в пользовательское событие. Метод `QApplication::postEvent()` принимает на себя владение событием, поэтому нам удалять событие не нужно.

На самом деле, для отправки событий есть два метода: `QApplication::sendEvent()` и `QApplication::postEvent()`.

Метод `sendEvent()` доставляет событие немедленно, но использовать его следует с осторожностью и только когда иначе никак нельзя. Например, в многопоточных программах событие, отправленное методом `sendEvent()`, обрабатывается в потоке отправителя, а не получателя. Кроме того, не выполняется сжатие и переупорядочение событий, например несколько событий рисования не упаковываются в одно. Наконец, метод `sendEvent()` не удаляет событие, поэтому на практике отправляемые с его помощью события создаются в стеке.

Метод `postEvent()` помещает событие – которое должно быть создано не в стеке, а в куче с помощью оператора `new` – в очередь событий получателя, так что оно обрабатывается в стандартном цикле обработки сообщений. Именно этот способ следует предпочесть при обычных обстоятельствах, потому что он хорошо согласован с принципами обработки событий в Qt.

В любом случае нас не должно беспокоить, что событие передается из одного потока в другой, – Qt знает, какие действия нужно выполнить.

```
struct ProgressEvent : public QEvent
{
    enum {EventId = QEvent::User};

    explicit ProgressEvent(bool saved_, const QString &message_)
        : QEvent(static_cast<Type>(EventId)),
          saved(saved_), message(message_) {}
    const bool saved;
    const QString message;
};
```

Это полное определение класса `ProgressEvent`. Каждому пользовательскому событию необходимо назначать уникальный идентификатор (`QEvent::User`, `QEvent::User + 1` и т. д.) типа `QEvent::Type`, чтобы по ошибке система не приняла одно сообщение за другое. Мы объявили событие в виде `struct` и оставили булевый флаг `saved` и текст сообщения `message` открытыми членами.

```
bool MainWindow::event(QEvent *event)
{
    if (!stopped && event->type() ==
        static_cast<QEvent::Type>(ProgressEvent::EventId)) {
        ProgressEvent *progressEvent =
            static_cast<ProgressEvent*>(event);
        Q_ASSERT(progressEvent);
        logEdit->appendPlainText(progressEvent->message);
        if (progressEvent->saved)
            ++done;
        return true;
    }
    return QMainWindow::event(event);
}
```

Если мы хотим распознавать и обрабатывать пользовательские события в некотором виджете, то должны переопределить в нем метод `QWidget::event()`. В данном случае если обработка продолжается (то есть не была отменена) и мы получаем событие `ProgressEvent`, то добавляем текст хранящегося в нем сообщения в журнал `QPlainTextEdit` и, если сохранение прошло успешно, увеличиваем счетчик обработанных файлов. Кроме того, мы возвращаем `true`, давая понять, что обработали событие и Qt может его удалить, не утруждаясь поисками других обработчиков для него. Обработку же всех остальных событий, а также и этого в случае, если оно получено уже после остановки конвертирования, мы поручаем обработчику событий из базового класса.

```
const int PollTimeout = 100;

void MainWindow::checkIfDone()
{
    if (QThreadPool::globalInstance()->activeThreadCount())
        QTimer::singleShot(PollTimeout, this, SLOT(checkIfDone()));
    else {
        QString message;
        if (done == total)
            message = tr("All %n image(s) converted", "", done);
        else
            message = tr("Converted %n/%1 image(s)", "", done)
                .arg(total);
        logEdit->appendPlainText(message);
        stopped = true;
        updateUi();
    }
}
```



```
    }
}
```

Этот слот вызывается в конце метода `convertFiles()` для опроса состояния. (Использование метода `QObject::tr()` обсуждается ниже во врезке «Использование `tr()` с тремя аргументами» на стр. 302.) Можно было бы снова применить пользовательское событие или сгенерировать сигнал, извещающий о завершении метода `convertImages()`, но нам все равно пришлось бы проверять, все ли активные потоки завершились, поэтому никакого выигрыша по сравнению с опросом мы не получили бы. Альтернативный подход – воспользоваться объектами `QFuture<T>`, которые возвращают вызовы `QtConcurrent::run()`, и объектами `QFutureWatcher<T>`, извещающими о завершении рабочих потоков, – как это делается, мы увидим во втором разделе данной главы.

Сначала мы смотрим, есть ли хотя бы один активный вспомогательный поток. Если да, то создаем однократный таймер, который вызовет этот же слот через 100 мс. Если же все потоки уже завершились, то мы дописываем в журнал финальное сообщение, восстанавливаем исходное значение переменной `stopped` (которая и так уже равна `true`, если конвертирование было прервано пользователем) и обновляем пользовательский интерфейс (то есть изменяем кнопку с `Cancel` на `Convert`).

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    stopped = true;
    if (QThreadPool::globalInstance()->activeThreadCount())
        QThreadPool::globalInstance()->waitForDone();
    event->accept();
}
```

Чтобы корректно выйти из многопоточной программы, рекомендуется сначала остановить все потоки, а только потом завершить саму программу. Мы реализовали эту идею, переопределив метод `closeEvent()`, в котором проверяем, что все активные потоки завершились.

Класс `QRunnable`

Альтернативой функции `QtConcurrent::run()` является создание подкласса `QRunnable` и выполнение его в потоке, взятом из глобального пула. Ниже мы рассмотрим этот подход, продемонстрировав другую реализацию метода `convertFiles()`, а также всех вспомогательных методов.

```
void MainWindow::convertFiles(const QStringList &sourceFiles)
{
    stopped = false;
    updateUi();
    total = sourceFiles.count();
    done = 0;
    const QVector<int> sizes = AQP::chunkSizes(sourceFiles.count(),
        QThread::idealThreadCount());
```

```

        int offset = 0;
        foreach (const int chunkSize, sizes) {
            ConvertImageTask *convertImageTask = new ConvertImageTask(
                this, &stopped, sourceFiles.mid(offset, chunkSize),
                targetTypeComboBox->currentText());
            QThreadPool::globalInstance()->start(convertImageTask);
            offset += chunkSize;
        }
        checkIfDone();
    }
}

```

Этот вариант метода `convertFiles()` по структуре мало чем отличается от рассмотренного выше (стр. 277). Основное различие состоит в том, что для обработки каждой порции файлов мы создаем объект класса `ConvertImageTask` (подкласса `QRunnable`) и передаем конструктору объекта точно те же самые аргументы, что и методу `QtConcurrent::run()`, предназначенная для последующей передачи функции `convertImages()`. Создав объект, мы вызываем его метод `QThreadPool::start()` – в результате владение объектом передается глобальному пулу потоков Qt и начинается его выполнение.

Пул потоков удалит объект `QRunnable`, когда его выполнение закончится, а именно это нам сейчас и нужно. Но можно было бы отменить такое поведение, вызвав метод `QRunnable::setAutoDelete(false)`, и в таком случае ответственность за удаление легла бы на нас.

```

class ConvertImageTask : public QRunnable
{
public:
    explicit ConvertImageTask(QObject *receiver,
        volatile bool *stopped, const QStringList &sourceFiles,
        const QString &targetType)
        : m_receiver(receiver), m_stopped(stopped),
          m_sourceFiles(sourceFiles),
          m_targetType(targetType.toLower()) {}

private:
    void run();
    ...
}

```

Это определение класса `ConvertImageTask`, в котором мы опустили лишь закрытые данные-члены. Сделав метод `run()` закрытым, мы запретили не только наследовать этому классу, но также и вызывать этот метод у созданных экземпляров (он должен вызываться только из `QThreadPool::start()`).

```

void ConvertImageTask::run()
{
    foreach (const QString &source, m_sourceFiles) {

```

```

        if (*m_stopped)
            return;
        QImage image(source);
        QString target(source);
        target.chop(QFileInfo(source).suffix().length());
        target += m_targetType;
        if (*m_stopped)
            return;
        bool saved = image.save(target);

        QString message = saved
            ? QObject::tr("Saved '%1'")
              .arg(QDir::toNativeSeparators(target))
            : QObject::tr("Failed to convert '%1'")
              .arg(QDir::toNativeSeparators(source));
        QMetaObject::invokeMethod(m_receiver, "announceProgress",
            Qt::QueuedConnection, Q_ARG(bool, saved),
            Q_ARG(QString, message));
    }
}

```

Структурно этот метод идентичен методу `convertImages()`, который мы видели раньше (стр. 277). Единственное различие в том, что взаимодействие с главным окном осуществляется не с помощью пользовательского события, а путем вызова слота `MainWindow::announceProgress()`.

Поскольку класс `QRunnable` не является подклассом `QObject`, в него не встроена поддержка сигналов и слотов. Можно было бы, конечно, унаследовать наш класс одновременно от `QObject` и `QRunnable`, но тогда уж лучше просто создать подкласс `QThread`, так как он наследует `QObject` и уже имеет несколько полезных встроенных сигналов и слотов. Другое решение – обратиться к пользовательским событиям, как мы поступили в предыдущем подразделе.

Но здесь мы решили просто вызвать слот напрямую, воспользовавшись методом `QMetaObject::invokeMethod()`. Ему передается объект-получатель, имя вызываемого слота, тип соединения (`Qt::QueuedConnection` подходит лучше всего, потому что соединения такого типа реализуются с помощью очереди сообщений, как и метод `QApplication::postEvent()` и передаваемые слоту аргументы. Каждый аргумент следует задавать с помощью макроса `Q_ARG`, который принимает тип и значение. В методе `QMetaObject::invokeMethod()` можно передать до девяти аргументов и, кроме того, разрешается указать возвращаемое значение¹, однако последнее имеет смысл, только если задан тип соединения `Qt::DirectConnection`.

Поскольку запрос на вызов слота помещается в главную очередь сообщений (пользовательского интерфейса), то выполняется он в потоке GUI, а не в рабочем потоке, который обратился к `QMetaObject::invokeMethod()`.

¹ При помощи макроса `Q_RETURN_ARG`. – *Прим. науч. ред.*

Это справедливо также для сигналов, генерируемых из рабочего потока, поскольку «за кулисами» Qt преобразует сигналы из потоков в события.

Некоторые программисты считают, что использование метода `QMetaObject::invokeMethod()` лучше отправки пользовательского события, потому что он бесшовно интегрируется с механизмом сигналов и слотов Qt и не требует ни создания подкласса `QEvent`, ни переопределения метода `QWidget::event()` в виджетах-получателях события. На внутреннем уровне сигналы и слоты, пересекающие границы потоков, все равно реализуются с помощью событий, но при использовании `QMetaObject::invokeMethod()` нам это совершенно не интересно, мы можем радоваться удобству вызова методов, не создавая какие-то там пользовательские события.

```
void MainWindow::announceProgress(bool saved, const QString &message)
{
    if (stopped)
        return;
    logEdit->appendPlainText(message);
    if (saved)
        ++done;
}
```

Этот метод добавляет переданное сообщение в конец журнала и увеличивает количество обработанных файлов, если сохранение завершилось успешно. Или ничего не делает, если обработка остановлена.

Вся прочая инфраструктура, необходимая для поддержки решения на основе подкласса `ConvertImageTask` класса `QRunnable`, точно такая же, как и для функции `QtConcurrent::run()`: опрос для выяснения того, закончилась ли обработка, выполняется в слоте `checkIfDone()` (стр. 279), а проверка завершения потоков при выходе из приложения – в переопределенном методе `closeEvent()` (стр. 282).

Хотя для извещения главной программы о ходе выполнения мы использовали события в версии с `QtConcurrent::run()` и вызов слота в версии с подклассом `QRunnable`, вполне можно было бы в обоих случаях использовать один и тот же механизм: как события, так и вызов слота. Вообще говоря, вызов слота лучше, потому что более удобен и требует меньше кода. В решении на основе `QRunnable` отслеживать ход выполнения можно с помощью опроса, как мы и поступили, а в решении на основе `QtConcurrent::run()` можно использовать как опрос, так и объект класса `QFutureWatcher<T>` (классы `QFuture<T>` и `QFutureWatcher<T>` рассматриваются в следующем разделе).

Основное различие между `QtConcurrent::run()` и `QRunnable` заключается в том, что функция `QtConcurrent::run()` возвращает объект `QFuture<T>`, а это позволяет отслеживать ход обработки (и управлять им). Эту тему мы будем рассматривать в следующем разделе. Напротив, при использовании `QRunnable` мы должны сами реализовывать механизм отслеживания и управления выполнением.

Использование `QtConcurrent::run()` или `QRunnable` особенно полезно при большом объеме обработки во вспомогательных потоках, например для обработки крупных объектов или большого числа элементов порциями, как в данном случае. Иногда, впрочем, приходится обрабатывать много элементов, но не существует удобного способа разбить их на порции. Для таких ситуаций в модуле *QtConcurrent* есть и другие функции, с которыми мы ознакомимся в следующем разделе.

Фильтрация и отображение в потоках

Функции из пространства имен `QtConcurrent` идеально подходят для ситуаций, когда имеется много элементов данных, требующих единообразной обработки. Если количество элементов невелико (скажем, не более пяти на одно ядро), то можно написать функцию для обработки каждого и запускать ее через `QtConcurrent::run()`, `QRunnable` или `QThread`. Однако если элементов действительно много – порядка сотен или даже тысяч, – то создавать поток для каждого элемента было бы чересчур накладно; последовательная обработка окажется гораздо быстрее. В предыдущем разделе мы видели, что один из способов решить эту проблему – создать небольшое количество потоков и поручить каждому обработку группы элементов. Но иногда нам действительно нужно обрабатывать каждый элемент индивидуально, и в модуле *QtConcurrent* имеются для этого функции.

Модуль *QtConcurrent* содержит функции четырех видов: фильтры, отображатели, редукторы (о них мы будем говорить в этом разделе) и функцию `run`, обсуждавшуюся в предыдущем разделе. Принцип работы с фильтрами, отображателями и редукторами прост: вы подаете им на вход набор элементов для обработки и поручаете `Qt` заботу о распределении работы между потоками из своего глобального пула.

Концепции фильтрации и отображения заимствованы из функционального программирования. В этом контексте фильтром называется функция высшего порядка (то есть функция, принимающая другую функцию в качестве одного из аргументов), на вход которой подается некая последовательность, а на выходе получается новая последовательность, состоящая из элементов, для которых функция фильтрации вернула `true`. А отображатель – это функция, принимающая последовательность и функцию отображения и возвращающая новую последовательность, элементы которой получены применением функции отображения к соответствующему элементу входной последовательности.

Фильтры и отображатели в модуле *QtConcurrent* устроены очень похоже. Так, фильтр принимает последовательность элементов и функцию фильтрации и порождает новую последовательность, содержащую только те элементы, для которых функция фильтрации возвращает `true`. Это означает, что результирующая последовательность может во-

обще не содержать элементов, содержать только часть исходных элементов или все исходные элементы. Концептуально фильтр работает следующим образом:

```
QList<Item> filter(QList<Item> items, FilterFunction isOK)
{
    QList<Item> results;
    foreach (Item item, items)
        if (isOK(item))
            results << item;
    return results;
}
```

Здесь мы использовали `QList`, но для хранения подлежащих фильтрации данных и конечного результата подойдет любой последовательный контейнер или его часть, заданная с помощью итераторов, указывающих на начало и конец.

Отображатели (не путать с контейнерным классом `QMap!`) принимают последовательность элементов и функцию отображения и порождают новую последовательность, содержащую столько же элементов, сколько исходная (хотя, возможно, других типов), причем каждый новый элемент получается из соответствующего исходного путем применения функции отображения. Концептуально отображатель работает следующим образом:

```
QList<Type2> mapper(QList<Type1> items, MapFunction map)
{
    QList<Type2> results;
    foreach (Type1 item, items)
        results << map(item);
    return results;
}
```

Здесь типы `Type1` и `Type2` могут совпадать или различаться — важно лишь, что функция отображения принимает элемент типа `Type1` и возвращает элемент типа `Type2`.

Редуктор принимает последовательность элементов и редуцирует его, сводя к одному элементу. Например, можно вычислить сумму или среднее значение последовательности элементов. В пространстве имен `QtConcurrent` имеются функции, которые объединяют фильтрацию с редукцией и отображение с редукцией. Концептуально они работают следующим образом:

<pre>// Фильтрация с редукцией // (Filter-Reduce) ResultType result; foreach (Item item, items) if (isOK(item)) result.merge(item);</pre>	<pre>// Отображение с редукцией // (Map-Reduce) ResultType result; foreach (Item item, items) result.merge(map(item));</pre>
---	--

Здесь в качестве функции фильтрации взята `isOk()`, а в качестве функции отображения — `map`. Метод `merge()` результирующего объекта принимает на входе прошедший фильтрацию (или, в случае отображения с редукцией, обработку) элемент и каким-то образом инкорпорирует его. Например:

```
struct ResultType
{
    ResultType() : sum(0) {}
    void merge(int item) { sum += item; }
    int sum;
};
```

Если этот тип `ResultType` применить к последовательности чисел, то после отображения с редукцией (в предположении, что отображение осуществляется тождественной функцией `int identity(int x) { return x; }`) в `result.sum` окажется сумма чисел.

Разумеется, у функций из модуля *QtConcurrent* есть существенное отличие от показанных выше простых циклов, и заключается оно в том, что обработка осуществляется не последовательно, а в одном или нескольких потоках, так что все множество элементов обрабатывается параллельно. Надо отчетливо понимать, что для несложной обработки дополнительные издержки, скорее всего, окажутся неоправданными, однако, если обработка потребляет много ресурсов, то применение функций из модуля *QtConcurrent* может повысить производительность.

В пространстве имен *QtConcurrent* есть как блокирующие, так и неблокирующие (поточковые) варианты функций фильтрации, отображения и редукции. Блокирующие варианты предназначены для использования в уже запущенных вспомогательных потоках (например, в подклассах *QRunnable* или *QThread*) или там, где нам нужно всего лишь поведение данной функции, а блокирует она выполнение программы или нет, неважно. В этой главе мы будем рассматривать только неблокирующие варианты.

Во время работы над этой книгой в документации по Qt было сказано, что при использовании функций из пространства имен *QtConcurrent* никакой блокировки не требуется. Это верно, когда речь идет об обработке независимых данных, которая больше ни на что не влияет. Однако обрабатывать таким образом элементы модели или графической сцены невозможно, потому что Qt не предоставляет средств для блокировки моделей, сцен или содержащихся в них элементов. Проблему нетрудно решить, создав собственные суррогатные элементы, которые можно обработать параллельно, а затем последовательно обновить модель или сцену — при условии, конечно, что издержки не перевешивают выигрыш от распараллеливания обработки элементов.

В этом разделе мы рассмотрим приложение *Number Grid* (*numbergrid*). Оно рисует матрицу чисел, как показано на рис. 7.2. Мы можем сцене-

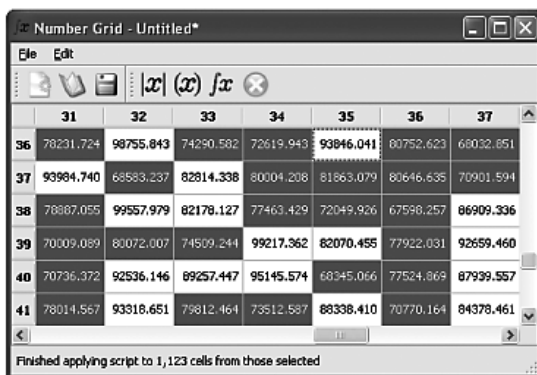


Рис. 7.2. Приложение Number Grid

рировать начальный набор предопределенных или случайных чисел и динамически заполнить или изменить матрицу, выполнив написанный нами JavaScript-сценарий. (Наш конкретный сценарий будет вызываться один раз для каждой ячейки и помещать в нее предопределенные значения.)

Мы решили производить всю обработку во вспомогательных рабочих потоках. Для этого, прежде чем приступить к обработке, мы должны последовательно создать суррогатный элемент для каждого обрабатываемого элемента модели, передать суррогатные элементы функции из модуля *QtConcurrent* для параллельной обработки и по ее завершении последовательно перебрать результирующие суррогатные элементы, обновляя по ходу дела элементы модели. Использование независимых суррогатных элементов позволяет обойтись без блокирования модели, представления или элементов модели (что в любом случае невозможно).

Что касается применения пользовательского JavaScript-сценария к элементам, то, хотя основные шаги те же самые, нам еще необходимо отслеживать число возникших ошибок и (уникальные) сообщения об ошибках. Следовательно, все рабочие потоки должны будут обновлять счетчик ошибок и набор сообщений об ошибках, поэтому придется прибегнуть к блокировке. Однако мы вынесем все детали блокировки в небольшой класс, так что клиентский код сможет обращаться к счетчику и сообщениям, не захватывая блокировку явно.

В приложении Number Grid используются функции из пространства имен *QtConcurrent* для фильтрации, фильтрации с редукцией и отображения. Точнее, фильтрация нужна для выделения всех ячеек матрицы, удовлетворяющих определенному критерию, фильтрация с редукцией – для подсчета числа ячеек, удовлетворяющих критерию, а отображение – для применения JavaScript-сценария к каждой ячейке (или каждой выделенной ячейке) матрицы.

Применение функций из пространства имен `QtConcurrent` для подсчета и выделения ячеек в приложении `Number Grid`, пожалуй, медленнее, чем подсчет и выделение напрямую, поскольку это очень дешевые операции. Поэтому данный пример следует рассматривать лишь как иллюстрацию работы с функциями фильтрации и фильтрации с редукцией. Однако отображение, сводящееся в данном случае к параллельному применению сценария ко всем или только к выделенным ячейкам, возможно, будет работать быстрее, чем если бы мы применяли сценарий последовательно к каждой ячейке. Потенциальное ускорение тем более вероятно, чем выше затраты на обработку каждой ячейки и чем больше ячеек предстоит обработать. Объясняется это тем, что стоимость начального конфигурирования и запуска потоков может быть покрыта за счет распараллеливания обработки двух и более ячеек.

Приложение `Number Grid` представляет собой традиционную для C++/Qt программу с главным окном. В нем есть меню `File` (Файл) с пунктами `New` (Создать), `Open...` (Открыть...), `Save` (Сохранить), `Save as...` (Сохранить как...) и `Quit` (Выход). Они вполне стандартны, поэтому рассматривать их мы не будем.

В меню `Edit` (Правка) есть пункты `Count...` (Подсчитать...), `Select...` (Выделить...), `Apply Script` (Применить сценарий) и `Stop` (Остановить), в каждом из которых используются функции из пространства имен `QtConcurrent`. При выборе операции подсчета или выделения открывается диалоговое окно, в котором можно задать критерий отбора. Для подсчета следует задать оператор числового сравнения (`<`, `<=`, `=`, `>` и `~=` – приблизительно равно), числовую величину и признак – подсчитывать все ячейки, удовлетворяющие критерию, или только выделенные. Для выделения пользователь задает оператор числового сравнения и числовую величину, заданный критерий применяется ко всем ячейкам. Когда пользователь выбирает пункт `Apply Script`, открывается диалоговое окно, в котором можно ввести произвольный JavaScript-сценарий с уже предопределенными глобальными переменными `cellValue`, `cellRow` и `cellColumn`. Далее можно указать, применяется ли сценарий ко всем ячейкам или только к выделенным.

Для выполнения описанных выше действий вызываются слоты `editCount()`, `editSelect()` и `editApplyScript()`. Мы рассмотрим их поочередно наряду с необходимыми им классами, методами и функциями, чтобы составить полное представление об использовании функций из модуля `QtConcurrent`. Хотя данные, обрабатываемые этими функциями, по идее, должны быть независимы, в одном случае мы все же используем объект `QMutex`, защищающий разделяемые данные. Прежде чем заняться самими слотами, приведем объявления некоторых закрытых переменных из заголовочного файла класса главного окна:

```
QStandardItemModel *model;  
QFutureWatcher<SurrogateItem> selectWatcher;  
QFutureWatcher<Results> countWatcher;
```

```
QFutureWatcher<SurrogateItem> applyScriptWatcher;  
MatchCriteria countCriteria;  
bool applyToAll;  
mutable bool cacheIsDirty;  
QString script;  
ThreadSafeErrorInfo errorInfo;
```

Мы предполагаем, что читатель знаком с классом `QStandardItemModel`, который используется как табличная модель для хранения чисел, представленных подклассом `QStandardItem`, где они хранятся в виде элементов типа `double` в роли `Qt::EditRole`. (Табличные модели Qt рассматривались в главе 3.)

Ниже мы увидим, что неблокирующая функция при вызове из модуля *QtConcurrent* немедленно возвращает объект типа `QFuture<T>`, который представляет будущий результат вычисления. Но сразу после возврата из функции этот объект обычно бесполезен, потому что вычисление только началось. Если обратиться к объекту `QFuture<T>`, то доступ будет заблокирован до завершения вычисления, а это нам, как правило, не нужно. Как только вычисление завершится, в объекте `QFuture<T>` окажется полученный результат – один в случае редукции или несколько при применении фильтрации и отображения.

Обычно при использовании неблокирующих функций из пространства имен `QtConcurrent` возвращенный объект `QFuture<T>` передается объекту-наблюдателю `QFutureWatcher<T>`. Наблюдатель следит за ходом выполнения функции и генерирует сигналы, извещающие о том, какой результат (или результаты – в случае операций фильтрации или отображения, порождающих последовательность результатов) получен, а также о завершении (либо приостановке, возобновлении, отмене) вычисления. «Наблюдатели будущего» предоставляют также API, позволяющий управлять поведением функции, например методы `pause()`, `resume()` и `cancel()`.

Отметим, что хотя фильтрация и отображение порождают последовательности результатов, тип `T`, параметризующий шаблонный класс `QFuture<T>`, всегда представляет собой тип одного результата, а тип `T` в классе `QFutureWatcher<T>` должен совпадать с типом наблюдаемого будущего¹.

В данном случае мы имеем по одному наблюдателю будущего для каждой поддерживаемой приложением операции из модуля *QtConcurrent*. Для выделения ячеек и применения к ним сценария предназначен класс `QFutureWatcher<SurrogateItem>` – напомним, что мы не можем параллельно обрабатывать элементы модели, поскольку нет средств для бло-

¹ То есть класса `QFuture<T>`. Это не распространяется на `QFutureWatcher<void>`, который может наблюдать за `QFuture<T>`, параметризованным любым типом, однако предоставляет для него только информацию о состоянии. – *Прим. науч. ред.*

кировки модели или ее элементов, поэтому приходится использовать суррогаты. Для подсчета используется класс `QFutureWatcher<Results>`; `Results`, как мы увидим позже, – это простая структура `struct`, в которой хранится счетчик и общее количество.

И для подсчета, и для выделения пользователь может задать критерий отбора. Ниже приведено определение структуры `MatchCriteria` и используемого в ней перечисления:

```
enum ComparisonType {LessThan, LessThanOrEqual, GreaterThan,
                    GreaterThanOrEqual, ApproximatelyEqual};

struct MatchCriteria
{
    ComparisonType comparisonType;
    double value;
    bool applyToAll;
};
```

В закрытой секции класса главного окна объявлена переменная `countCriteria` типа `MatchCriteria`, в которой хранится критерий, применяемый при подсчете. Критерии для выделения и применения сценария хранятся внутри объектов `SurrogateItem`, как мы увидим ниже. При выделении булевый флаг `applyToAll` игнорируется.

Имеется также закрытая булевая переменная `MainWindow::applyToAll`; если она равна `true`, то подсчет или применение сценария распространяется на все ячейки матрицы, иначе только на выделенные.

В булевой переменной `cacheIsDirty` хранится признак, показывающий, были ли изменены данные в матрице. Поскольку для нее задан модификатор `mutable`, то эту переменную можно изменять в константных методах.

В строке `script` хранится текст JavaScript-сценария, который будет выполняться для каждой ячейки (или для каждой выделенной ячейки), когда пользователь выберет пункт `Apply Script`. Во время выполнения сценария может произойти ошибка. Но мы не хотим всякий раз при возникновении ошибки открывать окно предупреждения, поскольку обрабатываться могут сотни, а то и тысячи ячеек, и ошибка может произойти в каждой. Поэтому мы просто подсчитываем общее число ошибок и ведем список уникальных сообщений об ошибках. Поскольку счетчик и сообщения могут обновляться в любом потоке, выполняющем функции из модуля *QtConcurrent*, то необходим какой-то механизм блокировки для сериализации доступа, который гарантирует, что обновлять счетчик и список сообщений в каждый момент разрешено только одному потоку.

Один из способов сериализовать доступ к информации об ошибках – создать три переменных, скажем, `int errorCount`, `QStringList errorMessages` и `QMutex errorMutex`. Когда понадобится получить доступ к информации об ошибках, мы должны захватить мьютекс. Но при таком подходе за-

бота обо всех служебных деталях лежит на нас. Мы решили поступить по-другому – создать класс `ThreadSafeErrorInfo`, который содержит методы для чтения и обновления информации об ошибках и сам осуществляет синхронизацию доступа. Это означает, что пользователи объектов данного класса могут не беспокоиться о блокировке – все будет сделано автоматически. Ниже приведен фрагмент объявления класса `ThreadSafeErrorInfo`:

```
class ThreadSafeErrorInfo
{
public:
    explicit ThreadSafeErrorInfo() : m_count(0) {}
    ...

private:
    mutable QMutex mutex;
    int m_count;
    QSet<QString> m_errors;
};
```

По существу, это довольно простая структура, содержащая два интересующих нас элемента данных плюс мьютекс. Мы рассмотрим несколько ее методов, чтобы понять, как используется мьютекс.

```
QStringList errors() const
{
    QMutexLocker locker(&mutex);
    return QStringList::fromSet(m_errors);
}
```

Конструктор класса `QMutexLocker` принимает указатель на мьютекс и блокирует программу, пока не сможет захватить этот мьютекс. При выходе объекта из области видимости мьютекс освобождается. Статический метод `QStringList::fromSet()` порождает список из множества.

В классе `ThreadSafeErrorInfo` есть также метод `count()`, который возвращает количество имевших место сообщений об ошибках (включая дубликаты), и метод `isEmpty()`, возвращающий `true`, если счетчик ошибок равен 0. Оба эти метода структурированы точно так же, как метод `errors()` (и потому не показаны).

```
void add(const QString &error)
{
    QMutexLocker locker(&mutex);
    ++m_count;
    m_errors << error;
}
```

Этот метод увеличивает счетчик сообщений и добавляет новое сообщение в множество. Поскольку сообщения хранятся в объекте `QSet`, то при попытке добавить дубликат сообщение просто игнорируется.

Альтернативное решение – хранить сообщения в списке `QStringList`, а внутри метода `errors()` копировать список, вызывать для копии метод `QStringList::removeDuplicates()` и возвращать результат. Но это означает, что вместо единственного экземпляра в множестве `m_errors` в памяти могут оказаться тысячи, а то и десятки тысяч повторяющихся сообщений.

```
void clear()
{
    QMutexLocker locker(&mutex);
    m_count = 0;
    m_errors.clear();
}
```

Этот метод очищает данные, например, перед вызовом новой функции из модуля *QtConcurrent*.

`QMutex` прекрасно работает в классе `ThreadSafeErrorInfo`, но оптимален ли он в данной ситуации? Мьютекс запрещает *любой* доступ к защищаемому объекту, в том числе для чтения. В классе `ThreadSafeErrorInfo` есть ряд методов, которые только читают внутренние данные, а также методы, обновляющие данные. В принципе, мы могли бы разрешить нескольким потокам одновременно читать данные, так как это не создаст никаких проблем.

Чтобы разграничить доступ для чтения и записи, необходимо заменить `QMutex` классом `QReadWriteLock`. Тогда в методах чтения (`count()`, `errors()` и `isEmpty()`) мы могли бы использовать объект `QReadLocker`, а в методах записи (`add()` и `clear()`) – объект `QWriteLocker`. Пример работы с классом `QReadWriteLock` мы увидим позже (стр. 331). В данном приложении при наличии только одного рабочего потока (например, на машине с одним процессорным ядром) между применением `QMutex` и `QReadWriteLock` не будет особых различий, но в остальных случаях использование `QReadWriteLock` должно повысить производительность – по крайней мере, теоретически.

Класс, обладающий собственным внутренним механизмом синхронизации (типа `ThreadSafeErrorInfo` или `ThreadSafeHash`, с которым мы ознакомимся позже), называется *монитором*. Во избежание взаимоблокировок метод монитора, чей код находится в области видимости его внутренней блокировки, не должен вызывать другие методы, пытающиеся захватить эту блокировку. И разумеется, любая блокировка, захваченная в каком-то методе монитора, должна освобождаться перед возвратом из этого метода – этому условию легко удовлетворить с помощью применения объектов `QMutexLocker` (или `QReadLocker`, или `QWriteLocker`)¹.

Теперь, ознакомившись с закрытыми данными-членами, обратимся к общему порядку работы с функциями из модуля *QtConcurrent*. Вначале мы вызываем какую-либо функцию (`filtered()`, `mapped()`, `filteredReduced()` или

¹ Более подробно с созданием мониторов можно ознакомиться в состоящей из двух частей статье в журнале *Qt Quarterly* «Monitors and Wait Conditions in Qt» по адресу qt.nokia.com/doc/qq/qq21-monitors.html.

`mappedReduced()` и сохраняем возвращенный ей объект `QFuture<T>` в переменной. Затем вызывается метод `QFutureWatcher<T>::setFuture()`, которому передается `QFuture<T>`. Мы можем следить за ходом выполнения, подключившись к сигналам наблюдателя будущего, но в любом случае необходимо установить соединение с сигналом `finished()`, чтобы знать, когда обработка завершилась и можно обращаться к результатам. Этот паттерн показан на рис. 7.3.

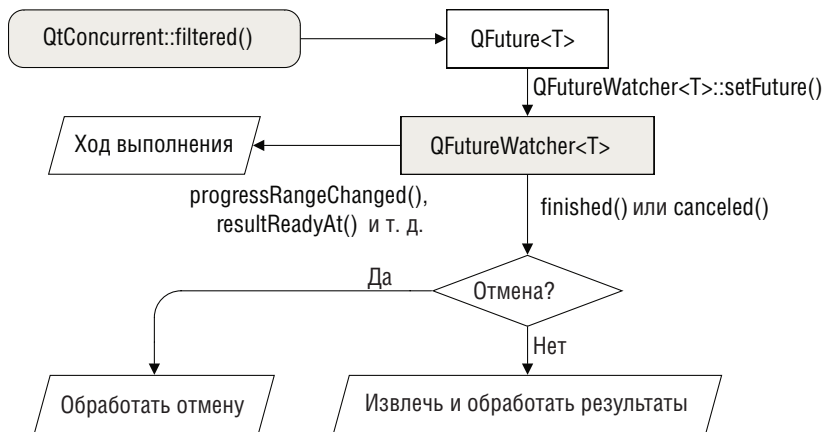


Рис. 7.3. Использование объекта `QFutureWatcher`

Все три наблюдателя будущего, которые используются в приложении `Number Grid`, хранятся закрытых переменных-членах, и для каждого создается два соединения сигнала со слотом.

```

connect(editStopAction, SIGNAL(triggered()),
        &selectWatcher, SLOT(cancel()));
connect(&selectWatcher, SIGNAL(finished()),
        this, SLOT(finishedSelecting()));
    
```

Здесь приведены исходные соединения для объекта `selectWatcher`; для объектов `countWatcher` и `applyScriptWatcher` все то же самое, только сигнал `finished()` соединен соответственно со слотами `finishedCounting()` и `finishedApplyingScript()`.

Чтобы при желании можно было прервать длительную операцию, в пользовательском интерфейсе имеется пункт меню `Stop` (Стоп) и соответствующая кнопка на панели инструментов. Для реализации этой возможности мы соединяем действие `stop` со слотом `cancel()` наблюдателя будущего. При любом – нормальном или в результате отмены – завершении вычисления генерируется сигнал `finished()`.

Ниже мы увидим, что в момент, когда наблюдателю будущего передается наблюдаемый объект, устанавливается дополнительное соединение, позволяющее пользователю следить за ходом выполнения с помощью индикатора, который появляется в строке состояния.

В следующем подразделе мы покажем, как с помощью функций из модуля *QtConcurrent* можно осуществлять фильтрацию, отображение и редукцию.

Использование QtConcurrent для фильтрации

Один из способов обработать все элементы модели, удовлетворяющие некоторому критерию, который задан программно или самим пользователем, – воспользоваться функцией из модуля *QtConcurrent*. В этом модуле имеется несколько функций фильтрации, но нас будет интересовать только `QtConcurrent::filtered()`, имеющая такие сигнатуры:

```
QFuture<T> filtered(Sequence, FilterFunction)
QFuture<T> filtered(ConstIterator, ConstIterator, FilterFunction)
```

Здесь `Sequence` – итерируемый контейнер элементов, например `QList<T>` или `QVector<T>`. Объекты типа `ConstIterator` – это итераторы, указывающие на начало и конец участка последовательности. Объект-функция `FilterFunction` служит для отбора элементов, включаемых в результат. Он должен иметь следующую сигнатуру:

```
bool filterFunction(const T&)
```

Тип `T` должен совпадать с типом элементов, хранящихся в контейнере `Sequence` (и с типом, используемым в `QFuture<T>`). Эта функция должна возвращать `true` для элементов, которые следует поместить в результирующий набор, и `false` – для тех, что нужно отфильтровать.

Поскольку мы не можем применять функции из модуля *QtConcurrent* непосредственно к элементам модели, то должны прибегнуть к описанной выше трехшаговой процедуре: сначала создать последовательность суррогатных элементов, соответствующих элементам модели, затем применить к ней функцию фильтрации, передав ей критерий отбора, – при этом порождается новая последовательность суррогатных элементов, удовлетворяющих критерию, – и наконец обойти эту последовательность и для каждого находящегося в ней элемента обновить соответствующий ему элемент модели.

В приложении *Number Grid* мы применяем фильтрацию для выделения элементов модели, удовлетворяющих заданному пользователем критерию, – например, всех ячеек матрицы, значение в которых меньше 3250. Для этого нам понадобятся суррогатные элементы; ниже приведено полное определение класса `SurrogateItem`:

```
struct SurrogateItem
{
    explicit SurrogateItem(int row_=0, int column_=0,
                           double value_=0.0)
        : row(row_), column(column_), value(value_) {}

    int row;
```

```
        int column;  
        double value;  
    };
```

Несмотря на свою тривиальность, этот класс очень важен. Посмотрим, как описанные выше три шага реализуются на практике, и начнем со слота `editSelect()`, который вызывается, когда пользователь выбирает пункт **Select... (Выделить...)**.

```
void MainWindow::editSelect()  
{  
    MatchForm matchForm(MatchForm::Select, this);  
    if (matchForm.exec()) {  
        MatchCriteria matchCriteria = matchForm.result();  
        stop();  
        view->setEditTriggers(QAbstractItemView::NoEditTriggers);  
        QList<SurrogateItem> items = allSurrogateItems();  
        QFuture<SurrogateItem> future = QtConcurrent::filtered(items,  
            SurrogateItemMatcher(matchCriteria));  
        selectWatcher.setFuture(future);  
        setUpProgressBar(selectWatcher);  
        editStopAction->setEnabled(true);  
    }  
}
```

Сначала мы выводим диалоговое окно типа `MatchForm` (не показано), в котором запрашиваем ввод критерия, то есть оператора сравнения и числового значения, с которым нужно сравнивать. После того как пользователь нажал в диалоговом окне кнопку **OK**, мы получаем критерий и запускаем процесс выделения (класс `MatchCriteria` представлен на стр. 291). Первым делом мы останавливаем уже запущенный процесс выделения, подсчета или применения сценария. Затем мы делаем представление доступным только для чтения, поскольку не хотим, чтобы пользователь изменял значения во время обработки, так как это может сделать результаты вычисления недействительными. Далее формируем список суррогатных элементов – по одному для каждой ячейки.

Получив суррогатные элементы, мы вызываем функцию из модуля *QtConcurrent*, в данном случае неблокирующую функцию `QtConcurrent::filtered()`, передавая ей последовательность подлежащих обработке элементов и функцию фильтрации. Последняя на самом деле представляет собой функтор (экземпляр класса, в котором реализован метод `operator()()`, рассматриваемый ниже). Функция фильтрации (или метод `operator()()` функтора) вызывается по одному разу для каждого элемента и обязана возвращать `true` для элементов, которые должны войти в состав результата, и `false` – для всех остальных.

Функция `QtConcurrent::filtered()` немедленно возвращает объект `QFuture<T>` и приступает к обработке в одном или нескольких вспомогательных потоках, предоставляя методу, из которого была вызвана,

возможность продолжить свою работу. Мы назначаем объект `QFuture<SurrogateItem>` наблюдаемым будущим для объекта `selectWatcher` (типа `QFutureWatcher<SurrogateItem>`); наблюдатель извещает о ходе выполнения и позволяет управлять обработкой, например приостанавливать, возобновлять или прерывать ее.

Напоследок мы настраиваем индикатор хода выполнения так, чтобы он показывал, насколько продвинулась обработка, и делаем доступным действие `Stop` (которому соответствует пункт меню и кнопка на панели инструментов), чтобы пользователь в любой момент мог прервать обработку. В данном случае мы не предоставили средств для приостановки и возобновления, но сделать это ничуть не сложнее, чем прервать обработку.

```
void MainWindow::stop()
{
    editStopAction->setEnabled(false);
    if (selectWatcher.isRunning())
        selectWatcher.cancel();
    ...
    if (selectWatcher.isRunning())
        selectWatcher.waitForFinished();
    ...
    editStopAction->setEnabled(false);
}
```

Этот метод вызывается, когда пользователь тем или иным способом активирует действие `Stop`. Мы показали код, работающий во всех случаях (первая и последняя строка), а также строки, относящиеся к объекту `selectWatcher`; для `countWatcher` и `applyScriptWatcher` все аналогично. Сначала мы делаем недоступным действие `Stop`, чтобы пользователь видел, что идет завершение обработки. Процедура завершения состоит из двух шагов: сначала посылается запрос на прекращение, а затем вызывается блокирующий метод `QFutureWatcher<T>::waitForFinished()` — это дает уверенность, что обработка действительно завершилась.

Примененный здесь подход — уведомление всех рабочих потоков о необходимости завершиться с последующим ожиданием каждого по очереди — в данном случае работает хорошо, так как мы точно знаем, что в каждый момент времени может работать не более одного вспомогательного потока (выделения, подсчета или применения сценария). Если же может одновременно выполняться два или более потоков, то общее время ожидания будет суммироваться (так как мы начинаем ждать следующий поток, только когда завершится предыдущий). Ниже мы покажем, как можно остановить несколько потоков, ожидая немногим дольше, чем время завершения самого медленного потока.

```
const QList<SurrogateItem> MainWindow::allSurrogateItems() const
{
    static QList<SurrogateItem> items;
```

```
    if (cacheIsDirty) {
        items.clear();
        for (int row = 0; row < model->rowCount(); ++row) {
            for (int column = 0; column < model->columnCount();
                ++column) {
                double value = model->item(row, column)->
                    data(Qt::EditRole).toDouble();
                items << SurrogateItem(row, column, value);
            }
        }
        cacheIsDirty = false;
    }
    return items;
}
```

Этот метод создает список суррогатных элементов, по одному для каждой ячейки матрицы. Хотя возвращаемый список может содержать десятки тысяч элементов, встроенный в Qt механизм копирования при записи гарантирует, что объем возвращаемых методом данных будет приблизительно равен размеру указателя.

Мы повышаем производительность этого метода еще больше, сделав список статическим, — чтобы он кэшировался между вызовами (ценой дополнительного расхода памяти), а обновлялся только при обновлении матрицы. (Для булевого флага `cacheIsDirty` устанавливается значение `true` в ответ на сигнал `dataChanged()` от модели благодаря соединению, которое здесь не показано.)

```
void MainWindow::setUpProgressBar(QFutureWatcher<T> &futureWatcher)
{
    progressBar->setRange(futureWatcher.progressMinimum(),
                          futureWatcher.progressMaximum());
    connect(&futureWatcher, SIGNAL(progressRangeChanged(int,int)),
            progressBar, SLOT(setRange(int,int)));
    connect(&futureWatcher, SIGNAL(progressValueChanged(int)),
            progressBar, SLOT(setValue(int)));
    progressBar->show();
}
```

Эта функция вызывается после того, как началась обработка. Ей передается соответствующий наблюдатель будущего. Скрытый индикатор хода выполнения создается и помещается в строку состояния в конструкторе. А сейчас мы устанавливаем диапазон изменения индикатора в соответствии с параметрами наблюдателя и создаем два соединения, благодаря которым текущее значение индикатора будет действительно отражать ход выполнения. Конечно, виджет необходимо сделать видимым — при этом он появится в строке состояния. При рассмотрении метода `finishedSelecting()` мы узнаем, когда индикатор снова становится невидимым.

Приведем полное определение класса `SurrogateItemMatcher`:

```
class SurrogateItemMatcher
{
public:
    explicit SurrogateItemMatcher(MatchCriteria matchCriteria_)
        : matchCriteria(matchCriteria_) {}

    typedef bool result_type;

    bool operator()(const SurrogateItem &item)
    {
        switch (matchCriteria.comparisonType) {
            case LessThan:
                return item.value < matchCriteria.value;
            case LessThanOrEqual:
                return item.value <= matchCriteria.value;
            case GreaterThanOrEqual:
                return item.value >= matchCriteria.value;
            case GreaterThan:
                return item.value > matchCriteria.value;
            case ApproximatelyEqual:
                return qFuzzyCompare(item.value, matchCriteria.value);
        }
        Q_ASSERT(false);
        return false;
    }

private:
    MatchCriteria matchCriteria;
};
```

Экземплярами класса `SurrogateItemMatcher` являются функторы. Для этого в классе должен быть реализован метод `operator()()`. А чтобы функтор можно было использовать в качестве «функции», передаваемой фильтру, в нем *обязательно* должен быть определен открытый псевдоним типа `typedef result_type`, который определяет тип результата, возвращаемого методом `operator()()`.

Конструктору функтора передается критерий отбора (он содержит оператор сравнения и числовое значение).

При обработке элемента функция `QtConcurrent::filtered()` вызывает метод `operator()()` и передает ему рассматриваемый элемент. Оператор возвращает булево значение, зависящее от того, удовлетворяет элемент критерию или нет.

В случае сравнения вида `ApproximatelyEqual (~=)` мы используем функцию Qt `qFuzzyCompare()`. Она сравнивает два значения типа `float` или `double` на приблизительное равенство (это лучшее, что можно сделать при стандартном представлении чисел с плавающей точкой).

Разумеется, в качестве функции фильтрации можно передать и обычную функцию – при условии, что она принимает аргумент типа `const T&` (то есть ссылку на элемент), а возвращает `bool`.

```
void MainWindow::finishedSelecting()
{
    editStopAction->setEnabled(false);
    progressBar->hide();
    if (!selectWatcher.isCanceled()) {
        view->clearSelection();
        QItemSelectionModel *selectionModel =
            view->selectionModel();
        const QList<SurrogateItem> items = selectWatcher.future().results();
        QListIterator<SurrogateItem> i(items);
        while (i.hasNext()) {
            const SurrogateItem &item = i.next();
            selectionModel->select(
                model->index(item.row, item.column),
                QItemSelectionModel::Select);
        }
        statusBar()->showMessage(
            tr("Selected %Ln cell(s)", "", items.count()),
            StatusTimeout);
    }
    view->setEditTriggers(editTriggers);
}
```

После создания наблюдателей будущего мы соединяем их сигнал `finished()` с соответствующим слотом, в случае `selectWatcher` это метод `finishedSelecting()`. Поэтому, когда процесс фильтрации останавливается по любой причине – из-за естественного завершения или в результате прерывания пользователем, активировавшим действие `Stop`, – вызывается этот слот.

Сначала мы деактивируем действие `Stop`, поскольку оно уже не имеет смысла, затем скрываем индикатор хода выполнения, так как обработка завершена. Если обработка не была прервана, то мы сбрасываем текущее выделение. Далее обходим в цикле все суррогатные элементы, которые не были отфильтрованы (то есть удовлетворяют заданному пользователем критерию отбора), и для каждого выделяем в представлении соответствующий ему элемент модели, получая от модели его модельный индекс.

Вызов `QObject::tr()` для форматирования сообщения в строке состояния выглядит необычно. Как правило, применяется форма этого метода с одним аргументом. Второй аргумент – строка-дискриминатор контекста, здесь она не используется. А третий аргумент – это числовой счетчик, который обсуждается во врезке «Использование `tr()` с тремя аргументами».

Использование `tr()` с тремя аргументами

Наиболее распространенной является форма метода `QObject::tr()` с одним аргументом – подлежащим переводу текстом, или с двумя аргументами, где второй – это строка, позволяющая различить контексты, в которых исходный текст один и тот же, но переводиться должен по-разному.

Форма с тремя аргументами появилась в версии Qt 4.2 и используется, когда текст содержит числа; в этом случае третий аргумент – целочисленный счетчик. В этой форме подлежащий переводу текст должен содержать спецификатор `%n`, который будет заменен счетчиком (или `%Ln` для локализованного варианта счетчика, то есть с группировкой по три цифры). Для английского языка полезно также добавлять `(s)` в конце слова, которое должно быть представлено в единственном или множественном числе. Без перевода текст будет представлен, например, так: «Selected 1 cell(s)», если счетчик равен 1. Он вполне читабелен, пусть даже выглядит немного по-дилетантски. Но в переводе мы можем выбрать разные тексты для единственного или множественного числа.

Входящие в состав Qt инструменты переводчика достаточно «разумны» и предлагают простые формы единственного или множественного числа для языков, в которых форма множественного числа образуется, как в английском (голландский, греческий, иврит и зулусский), и более широкий набор вариантов для остальных языков (например, арабского, чешского, французского, ирландского, маори, польского и русского)¹.

Кажется странным предоставлять файл перевода на английский для приложения, которое изначально разрабатывалось на английском языке. Однако часто это бывает полезно. Сразу же отметим, что почти никакие фразы переводить не нужно, потому что Qt по умолчанию возьмет исходный английский текст, если перевод отсутствует. Поэтому нам предстоит перевести только строки, встречающиеся в вызовах `QObject::tr()` с тремя аргументами, а все остальное оставить без перевода.

Кроме того, если уже после развертывания приложения обнаружатся опечатки или иные проблемы в строках, видимых пользователям, то при наличии автономного *qm*-файла ошибочные сообщения можно будет «перевести», то есть исправить ошибки, просто отправив пользователям новый *qm*-файл.

¹ Полному рассмотрению этого вопроса посвящена статья «Plural Form(s) in Translation(s)» в журнале *Qt Quarterly* по адресу qt.nokia.com/doc/qq/q19-plurals.html.

В приложении `numbergrid` всего имеется больше шестидесяти видимых строк, а перевести из них нужно только четыре. Например, текст `Selected %Ln cell(s)` был переведен как `Selected one cell` (в единственном числе) и `Selected %Ln cells` (во множественном числе).

В заключение мы активируем триггеры редактирования, чтобы пользователь снова мог взаимодействовать с ячейками матрицы. (Триггеры имеют тип `QAbstractItemView::EditTriggers`; они были получены от представления в момент его создания и сохранены в переменной-члене.)

Только что мы видели, как функция `QtConcurrent::filtered()`, получив последовательность элементов и функцию фильтрации (в нашем случае функтор), порождает профильтрованную последовательность в одном или нескольких вспомогательных потоках. В следующем подразделе мы продемонстрируем фильтрацию с редукцией, а затем отображение.

Применение модуля `QtConcurrent` для фильтрации с редукцией

В модуле `QtConcurrent` есть несколько функций редукции, в том числе `QtConcurrent::mappedReduced()` и `QtConcurrent::filteredReduced()`; последнюю мы и рассмотрим в этом подразделе. У нее имеются варианты со следующими сигнатурами:

```
QFuture<T> filteredReduced(Sequence, FilterFunction,
                          ReduceFunction, QtConcurrent::ReduceOptions)
QFuture<T> filteredReduced(ConstIterator, ConstIterator,
                          FilterFunction, ReduceFunction, QtConcurrent::ReduceOptions)
```

Как и в случае функции `QtConcurrent::filtered()`, `Sequence` — это итерируемый контейнер элементов, например `QList<T>` или `QVector<T>`, а объекты типа `ConstIterator` — итераторы, указывающие на начало и конец участка последовательности. Функция `FilterFunction` определяет, какие элементы поместить в результирующую последовательность, а какие отбросить (ее сигнатура была приведена на стр. 296). Функция `ReduceFunction` должна иметь такую сигнатуру:

```
void reduceFunction(R&, const T&)
```

Строго говоря, тип возвращаемого значения не обязан быть `void`, потому что он игнорируется. Неконстантный тип `R` служит для аккумуляции результата; тип `T` должен быть таким же, как в `Sequence` и в `QFuture<T>`.

`QtConcurrent::ReduceOptions` — это перечисление, и в обоих перегруженных вариантах этот аргумент имеет значение по умолчанию `UnorderedReduce`, так что он необязателен. Флаг `UnorderedReduce` означает, что элементы можно обрабатывать в любом порядке, какой

QtConcurrent сочтет нужным. Если вместо него задать флаг `OrderedReduce`, обработка будет производиться в естественном порядке контейнера `Sequence`. Флаг `SequentialReduce` информирует о том, что в каждый момент времени внутри функции `ReduceFunction` может находиться только один поток. Это означает, что сама функция редукции не обязана быть реентерабельной или потокобезопасной. (Во время написания данной книги у этого режима не было альтернативы, но в будущем Qt, возможно, станет поддерживать параллельную редукцию, и тогда к функции редукции будет предъявляться требование реентерабельности или потокобезопасности.)

Фильтрация с редукцией аналогична простой фильтрации в том смысле, что мы должны передать функции из модуля *QtConcurrent* последовательность элементов и функцию или функтор для решения вопроса о том, следует ли учитывать элемент. Но помимо этого необходимо передать функцию, которая будет играть роль аккумулятора, то есть каким-то образом объединять все прошедшие фильтрацию элементы или просто подсчитывать их; мы реализуем и то и другое.

По структуре код фильтрации с редукцией практически не отличается от простой фильтрации: слот для запуска процесса (в данном случае `editCount()`) и слот, вызываемый в результате завершения или прерывания (здесь `finishedCounting()`). Ниже мы рассмотрим и эти слоты, и вспомогательный код.

```
void MainWindow::editCount()
{
    MatchForm matchForm(MatchForm::Count, this);
    if (matchForm.exec()) {
        countCriteria = matchForm.result();
        stop();
        view->setEditTriggers(QAbstractItemView::NoEditTriggers);
        applyToAll = countCriteria.applyToAll;
        QList<SurrogateItem> items = applyToAll ? allSurrogateItems()
                                                : selectedSurrogateItems();
        QFuture<Results> future = QtConcurrent::filteredReduced(
            items, SurrogateItemMatcher(countCriteria),
            itemAccumulator);
        countWatcher.setFuture(future);
        setUpProgressBar(countWatcher);
        editStopAction->setEnabled(true);
    }
}
```

Если конструктору диалогового окна `MatchForm` передан аргумент `MatchForm::Count`, то пользователю предоставляется возможность выбрать оператор сравнения, числовое значение и режим — подсчитывать ли все элементы или только выделенные. При нажатии кнопки ОК мы получаем от диалогового окна критерий отбора (типа `MatchCriteria`; стр. 291), а затем,

как и в рассмотренном выше слоте `editSelect()`, останавливаем текущую операцию и делаем представление доступным только для чтения.

В переменной-члене `applyToAll` мы запоминаем, надо ли обрабатывать все элементы или только выделенные, потому что эта информация понадобится позже, когда обработка закончится и нужно будет вывести подходящее сообщение. Если пользователь заказал подсчет всех элементов, то мы получаем последовательность суррогатных элементов от метода `allSurrogateItems()` (стр. 297), в противном случае вызываем очень похожий метод `selectedSurrogateItems()`, который рассмотрим чуть ниже. Суррогатные элементы представлены тем же классом, что и выше (стр. 296).

Функция `QtConcurrent::filteredReduced()` принимает последовательность элементов, функцию или функтор фильтрации (здесь мы снова пользуемся функтором `SurrogateItemMatcher`; см. стр. 300) и функцию аккумулятора, `itemAccumulator()` (см. ниже). Функция немедленно возвращает объект типа `QFuture<Results>`; здесь `Results` – структура, созданная специально для работы с аккумулятором; мы приведем ее код, когда будем рассматривать аккумулятор.

Последние несколько строк почти такие же, как в слоте `editSelect()`: мы передаем объект, представляющий будущее, наблюдателю, настраиваем индикатор хода выполнения (стр. 298) и делаем доступным действие `Stop`, чтобы пользователь мог прервать операцию.

```
QList<SurrogateItem> MainWindow::selectedSurrogateItems() const
{
    QList<SurrogateItem> items;
    QItemSelectionModel *selectionModel = view->selectionModel();
    for (int row = 0; row < model->rowCount(); ++row) {
        for (int column = 0; column < model->columnCount();
            ++column) {
            QStandardItem *item = model->item(row, column);
            if (selectionModel->isSelected(item->index())) {
                double value = item->data(Qt::EditRole).toDouble();
                items << SurrogateItem(row, column, value);
            }
        }
    }
    return items;
}
```

Этот метод структурно аналогичен методу `allSurrogateItems()` (стр. 297). Единственное отличие в том, что мы добавляем суррогат не для каждого элемента, а лишь для выделенных, и не кэшируем элементы.

Здесь кэширование только потребляло бы лишнюю память, не давая никакого выигрыша в скорости, потому что результирующий список элементов зависит от того, что именно выделено, а модель выделения при следующем вызове, скорее всего, изменится.


```
struct Results
{
    explicit Results() : count(0), sum(0.0) {}

    int count;
    long double sum;
};
```

Эта структура объявлена в заголовочном файле класса главного окна – она должна быть там (возможно, за счет включения другого файла), потому что используется в объявлении наблюдателя будущего `QFutureWatcher<Results> countWatcher`. В переменной-члене `count` хранится счетчик отобранных элементов, а в переменной `sum` – их сумма.

```
void itemAccumulator(Results &results, const SurrogateItem &item)
{
    ++results.count;
    results.sum += item.value;
}
```

Эта функция передается в качестве предпоследнего¹ аргумента функции `QtConcurrent::filteredReduced()`; ее сигнатура была приведена на стр. 303. Она вызывается только для тех элементов, которые прошли фильтрацию. Мы просто увеличиваем счетчик и сумму отобранных элементов. Отметим, что начальный объект результатов (в данном примере имеющий тип `Results`) создается Qt, поэтому важно предоставить конструктор, правильно инициализирующий переменные-члены структуры.

Вычисление суммы многих чисел с плавающей точкой путем их сложения по одному – простой и очень наивный алгоритм. Потенциальная проблема заключается в том, что сложение двух таких чисел, сильно различающихся по порядку величины, может привести к потере точности; иначе говоря, прибавление достаточно малого числа к достаточно большому оставит большое число без изменения, то есть сложение «потеряется». Это может произойти даже в том случае, когда все складываемые числа малы (если их достаточно много), потому что итоговая сумма в какой-то момент может оказаться настолько большой, что прибавление к ней малых чисел не даст никакого эффекта. Известны и решения этой проблемы, например алгоритм суммирования Кахана (en.wikipedia.org/wiki/Kahan_summation_algorithm).

```
void MainWindow::finishedCounting()
{
    editStopAction->setEnabled(false);
    progressBar->hide();
    if (!countWatcher.isCanceled()) {
        Results results = countWatcher.result();
        QString selected(applyToAll ? QString()
```

¹ Напомним, что последний аргумент функции – набор опций, имеющий значение по умолчанию. – *Прим. науч. ред.*

```

        : tr(" from those selected"));
    AQP::information(this, tr("Count"),
        tr("A total of %Ln cell(s)%2 are %3 %4.\n"
            "Their total value is %L5.", "", results.count)
        .arg(selected)
        .arg(comparisonName(countCriteria.comparisonType))
        .arg(countCriteria.value)
        .arg(stringForLongDouble(results.sum)));
    }
    view->setEditTriggers(editTriggers);
}

```

Этот метод структурно аналогичен методу `finishedSelecting()`, рассмотренному выше, на стр. 301. Основное отличие в том, что вместо последовательности результатов мы извлекаем единственный результирующий объект — он был создан Qt и обновлялся функцией аккумулятора, вызываемой по одному разу для каждого пропущенного фильтром элемента.

Функция `comparisonName()` (не показана) просто возвращает строку `QString`, соответствующую указанному типу сравнения, например для `LessThan` возвращается строка `"<"`.

К сожалению, во время работы над этой книгой не существовало перегруженного метода `QString::arg()`, принимающего аргумент типа `long double`, поэтому нам пришлось написать свою функцию, преобразующую `long double` в `QString`.

```

QString stringForLongDouble(const long double &x)
{
    const int BUFFER_SIZE = 20;

    char longDouble[BUFFER_SIZE + 1];
    int i = snprintf(longDouble, BUFFER_SIZE, "%.3Lf", x);
    if (i < 0 || i >= BUFFER_SIZE) // Переполнение
        return QString("#####");
    return QString(longDouble);
}

```

Здесь мы используем функцию `snprintf()` из модуля `<stdio>`. В случае ошибки преобразования или при переполнении мы решили не возбуждать исключение и не возвращать код ошибки, а вернуть специальную строку, как это принято в электронных таблицах.

Итак, мы продемонстрировали простую фильтрацию и фильтрацию с редукцией. В следующем подразделе мы рассмотрим отображение. Посвящать отдельный раздел отображению с редукцией мы не стали, потому что принцип работы здесь такой же, как для фильтрации с редукцией, а единственное различие заключается в том, что аккумулятору передаются все элементы (после обработки), а не только пропущенные функцией фильтрации.

Применение модуля QtConcurrent для отображения

Под отображением понимается подача каждого элемента последовательности на вход функции отображения, которая в ответ возвращает обработанный элемент (возможно, другого типа). В модуле *QtConcurrent* имеется несколько отображателей, нас будут интересовать только функции со следующими сигнатурами:

```
QFuture<T> mapped(Sequence, MapFunction)
QFuture<T> mapped(ConstIterator, ConstIterator, MapFunction)
```

Как и в случае фильтрации и редукции, *Sequence* — это итерируемый контейнер, например *QList<T>* или *QVector<T>*, а объекты типа *ConstIterator* — итераторы, указывающие на начало и конец участка последовательности. Функция *MapFunction* должна иметь такую сигнатуру:

```
U mapFunction(const T&)
```

T — это тот же тип, что у элементов последовательности *Sequence* и в шаблонном классе *QFuture<T>*. *U* — тип результата, возвращаемого после обработки каждого элемента типа *T*; он может совпадать с типом *T*, если выходная последовательность состоит из элементов того же типа, что и входная.

В этом разделе мы будем рассматривать обработку всех элементов табличной модели. Как и в случае фильтрации, мы не можем работать непосредственно с элементами модели, потому что не существует способа синхронизировать доступ к ним. Поэтому мы создадим последовательность суррогатных элементов, передадим их по одному функции отображения, а затем обойдем результирующую последовательность и обновим модель.

В данном примере обработка будет заключаться в применении пользовательского JavaScript-сценария к каждому значению матрицы.

```
void MainWindow::editApplyScript()
{
    ScriptForm scriptForm(script, this);
    if (scriptForm.exec()) {
        script = scriptForm.script();
        stop();
        view->setEditTriggers(QAbstractItemView::NoEditTriggers);
        errorInfo.clear();
        applyToAll = scriptForm.applyToAll();
        QList<SurrogateItem> items = applyToAll
            ? allSurrogateItems()
            : selectedSurrogateItems();
        QFuture<SurrogateItem> future = QtConcurrent::mapped(items,
            SurrogateItemApplier(script, &errorInfo));
        applyScriptWatcher.setFuture(future);
        setUpProgressBar(applyScriptWatcher);
        editStopAction->setEnabled(true);
    }
}
```

```
    }  
}
```

Класс `ScriptForm` (не показан) описывает простое диалоговое окно, в котором пользователь может ввести код сценария. Помимо стандартных функций и переменных JavaScript, мы завели три глобальные переменные, `cellValue`, `cellRow` и `cellColumn`, которые можно употреблять в сценарии. Мы не стали предоставлять доступ к значениям в произвольной ячейке матрицы – это было бы слишком сложно и далеко увело бы нас от темы многопоточной обработки. (Это ограничение как раз и обуславливает тот факт, что наша программа – простая матрица чисел, а не электронная таблица.)

При нажатии кнопки OK мы получаем текст введенного пользователем сценария. Метод `ScriptForm::accept()` вызывает метод `QScriptEngine::checkSyntax()`, появившийся в версии Qt 4.5, и разрешает пользователю закрыть диалоговое окно, только если в сценарии нет синтаксических ошибок или при нажатии кнопки Cancel.

Как обычно, мы прерываем уже начатую обработку и делаем представление доступным только для чтения. Кроме того, мы обнуляем счетчик ошибок и очищаем список сообщений об ошибках, вызывая метод `clear()` переменной-члена `errorInfo` рассмотренного выше типа `ThreadSafeErrorInfo` (стр. 293). Мы отбираем либо все, либо только выделенные элементы – в зависимости от заданного пользователем режима. Функция `QtConcurrent::mapped()` принимает последовательность элементов и функцию (или функтор) отображения – в данном случае мы воспользовались функтором `SurrogateItemApplier`.

Функция отображения сразу же возвращает управление, и мы передаем полученный объект-будущее наблюдателю. Затем обновляем строку состояния и делаем доступным действие Stop.

Вот полное определение класса `SurrogateItemApplier`:

```
class SurrogateItemApplier  
{  
public:  
    explicit SurrogateItemApplier(const QString &script_,  
                                   ThreadSafeErrorInfo *errorInfo_)  
        : script(script_), errorInfo(errorInfo_) {}  
  
    typedef SurrogateItem result_type;  
  
    SurrogateItem operator()(const SurrogateItem &item)  
    {  
        QScriptEngine javaScriptParser;  
        javaScriptParser.globalObject().setProperty("cellRow", item.row);  
        javaScriptParser.globalObject().setProperty("cellColumn", item.column);  
        javaScriptParser.globalObject().setProperty("cellValue", item.value);  
        QScriptValue result = javaScriptParser.evaluate(script);
```

```

        if (javaScriptParser.hasUncaughtException()) {
            QString error = javaScriptParser.uncaughtException().toString();
            errorInfo->add(error);
            return item;
        }
        return SurrogateItem(item.row, item.column, result.toNumber());
    }

private:
    QString script;
    ThreadSafeErrorInfo *errorInfo;
};

```

В классе *должен* быть определен псевдоним типа `typedef result_type`, иначе его экземпляры нельзя будет передать функциям из модуля *QtConcurrent*. Кроме того, в закрытых переменных хранится текст сценария и переданный конструктору объект `ThreadSafeErrorInfo`.

Для каждого элемента последовательности, переданной функции `QtConcurrent::mapped()`, вызывается метод `operator()()`. В данном случае мы возвращаем объект того же типа `SurrogateItem`, но это необязательно – можно возвращать любой тип, лишь бы он был определен в `typedef result_type`.

Сначала мы создаем анализатор JavaScript и внедряем в него три глобальных переменных: номера строки и столбца элемента, а также его значение. Затем выполняем сценарий и получаем возвращенное им значение (то есть значение последнего вычисленного выражения). Ниже приведен пример простого JavaScript-сценария:

```

Java | var result = cellValue;
Script | if (cellRow < 10 && cellColumn < 10)
      |     result *= 2;
      | result;

```

Понятно, что этот сценарий всего лишь удваивает значения в ячейках, расположенных в левом верхнем углу матрицы, размером 10×10 .

Если сценарий возбудит необработанное исключение, то мы добавим сообщение об ошибке в объект `errorInfo`. Нам ни к чему беспокоиться о синхронизации доступа, хотя элементы могут обрабатываться в нескольких потоках, – этими деталями занимается класс `ThreadSafeErrorInfo` (стр. 293).

В заключение мы возвращаем либо исходный неизменный элемент (если имело место необработанное исключение), либо новый элемент с такими же номерами строки и столбца, как у исходного, но с другим числовым значением – тем, которое было вычислено сценарием.

Хотя слот `finishedApplyingScript()` структурно аналогичен слотам `finishedSelecting()` (стр. 301) и `finishedCounting()` (стр. 304), мы все же рассмотрим его, чтобы продемонстрировать обработку ошибок сценария.

```

void MainWindow::finishedApplyingScript()
{
    editStopAction->setEnabled(false);
    progressBar->hide();
    if (!applyScriptWatcher.isCanceled() &&
        (errorInfo.isEmpty() || applyDespiteErrors())) {
        const QList<SurrogateItem> items =
            applyScriptWatcher.future().results();
        QListIterator<SurrogateItem> i(items);
        while (i.hasNext()) {
            const SurrogateItem &item = i.next();
            model->item(item.row, item.column)->setData(item.value,
                                                         Qt::EditRole);
        }
        QString selected(applyToAll ? QString()
                                     : tr(" from those selected"));
        statusBar()->showMessage(tr("Finished applying script "
                                     "to %Ln cell(s)%1", "", items.count())
                                 .arg(selected), StatusTimeout);
    }
    view->setEditTriggers(editTriggers);
}

```

Если обработка закончилась нормально (то есть не была прервана) и не возникло ошибок или пользователь хочет записать результаты работы сценария, несмотря на ошибки, то мы запрашиваем результирующую последовательность. (В этот момент данные модели еще не изменены, а результаты находятся в последовательности суррогатных элементов.) Затем мы обходим результаты и заносим в соответствующий элемент модели вычисленные сценарием новые значения. После этого выводим в строку состояния сообщение о том, сколько ячеек было изменено¹. А в самом конце восстанавливаем триггеры редактирования, чтобы пользователь снова мог взаимодействовать с ячейками матрицы.

```

bool MainWindow::applyDespiteErrors()
{
    const int MaxErrorStrings = 15;
    QStringList errors = errorInfo.errors();
    if (errors.count() > MaxErrorStrings) {
        errors = errors.mid(0, MaxErrorStrings);
        errors.append(tr("(and %L1 others...)")
                     .arg(errorInfo.count() - MaxErrorStrings));
    }
}

```

¹ Отметим, что избранный автором прием с добавлением строки «from those selected» (из выбранного) является не самым универсальным: в переводе может быть другой порядок слов в предложении. Более гибким решением было бы предусмотреть два разных (хотя и похожих) сообщения. — *Прим. науч. ред.*

```
return AQP::question(this, tr("Apply Script Error"),
    tr("%Ln error(s) occurred:\n%1", "", errorInfo.count())
    .arg(errors.join("\n")),
    "", tr("&Apply Anyway"), tr("&Don't Apply"));
}
```

Этот метод вызывается, если произошла хотя бы одна ошибка. Он выводит диалоговое окно, содержащее не более 15 сообщений об ошибках (а если их больше, то также информацию о том, сколько не показано). Такая осторожность объясняется тем, что в матрице может быть много тысяч ячеек и, в принципе, мы можем получить столько же сообщений об ошибках, то есть гораздо больше, чем принято показывать в диалоговом окне.

Метод `QStringList::mid()` мы рассматривали выше (стр. 277). Функция `AQP::question()` похожа на аналогичные уже встречавшиеся нам функции (например, `AQP::okToDelete()`; стр. 121).

На этом мы завершаем обзор функции из модуля *QtConcurrent*. Мы продемонстрировали фильтрацию, фильтрацию с редукцией, отображение, а в предыдущем разделе также выполнение функций в потоках. Мы не привели пример отображения с редукцией, но оно отличается от фильтрации с редукцией только тем, что вместо функции (или функтора) фильтрации применяется функция (или функтор) отображения.

Применение неблокирующих функций из модуля *QtConcurrent* сопряжено с непроизводительными издержками на запуск вспомогательных потоков, а в случае модели или графической сцены – еще и с созданием суррогатных элементов. Но эти расходы компенсируются, если обработка каждого элемента обходится дорого, особенно когда таких элементов много.

Хотя модуль *QtConcurrent* идеален для выполнения дорогостоящих вычислений при большом количестве элементов, бывают случаи, когда требуется обработать всего один или совсем немного элементов, но обработка отнимает много времени, в течение которого пользовательский интерфейс все равно должен отвечать на действия пользователя. Одно из решений этой проблемы – воспользоваться функцией `QtConcurrent::run()` или объектом `QRunnable`, как было показано в предыдущем разделе. Но если нам нужен точный контроль и все удобства механизма сигналов и слотов, то, быть может, стоит обратиться к классу `QThread`.

Мы завершили рассмотрение высокоуровневых классов Qt для многопоточной обработки. Это самый короткий и безопасный путь к благам многопоточного программирования. Но в некоторых ситуациях хотелось бы контролировать программу более точно, и ради этого мы готовы взять на себя ответственность за синхронизацию доступа и предотвращение взаимоблокировок. Вот тогда-то и приходит на помощь низкоуровневый класс `QThread`, которым мы займемся в следующей главе.

8

Многопоточное программирование с классом QThread

- Обработка независимых элементов
- Обработка разделяемых данных

В этой главе мы рассмотрим класс `QThread`, который обеспечивает точный контроль над многопоточной обработкой в сочетании с поддержкой механизма сигналов и слотов. Как и в предыдущей главе, предполагается знакомство с основами многопоточного программирования в целом и в Qt в частности. Предполагается также, что вы прочитали хотя бы начало предыдущей главы.

Если имеется сравнительно немного объектов (или небольших групп объектов), которые надо обработать в фоновом режиме, получая информацию о ходе выполнения и о завершении обработки, то часто оптимальным решением оказывается создание подкласса класса `QThread`. Класс `QThread` Qt (как и `QRunnable`) был спроектирован по образцу класса `Thread` в языке Java и имеет сходные с ним черты – например, в любом его подклассе должен быть реализован метод `run()`, а запуск потока осуществляется методом `start()`.

Основное различие между классами `QThread` и `QRunnable` заключается в том, что `QThread` – подкласс `QObject`, поэтому для слежения за ходом выполнения можно использовать сигналы и слоты. И сам класс `QThread` предоставляет ряд полезных сигналов и слотов, к которым можно подключаться. На рис. 8.1 схематически изображен порядок создания и использования нескольких экземпляров подкласса `QThread`.

В этой главе мы рассмотрим два разных приложения с использованием класса `QThread`: в одном вспомогательные потоки используются для

обработки независимых элементов, поэтому синхронизация доступа не нужна; в другом заполняется разделяемая структура данных и, значит, для обеспечения безопасного доступа необходима синхронизация.

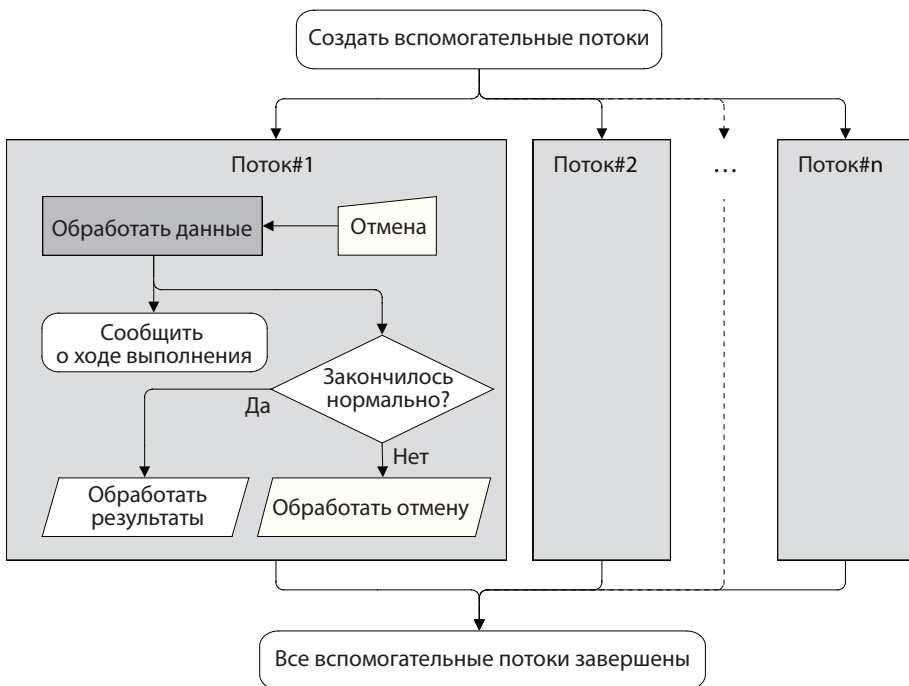


Рис. 8.1. Использование класса *QThread* для обработки данных

Обработка независимых элементов

В этом разделе мы рассмотрим приложение *Cross Fader* (*crossfader*), показанное на рис. 8.2. Пользователь может выбрать два изображения и задать количество промежуточных изображений, а программа построит указанное количество изображений для осуществления постепенного перехода между крайними. Например, если пользователь закажет три промежуточных изображения, то в результате получит последовательность из пяти изображений, в которых пропорциональные доли первого и второго изображения меняются следующим образом: начальное изображение (100:0), первое микшированное изображение (75:25), второе микшированное изображение (50:50), третье микшированное изображение (25:75), конечное изображение (0:100).

Эффект плавного микширования цвета с четырьмя промежуточными изображениями показан на снимке экрана на рис. 8.2, а получившиеся в результате изображения – на рис. 8.3; доли начального и конечного изображений меняются следующим образом: (100:0), (80:20), (60:40), (40:60), (20:80), (0:100).

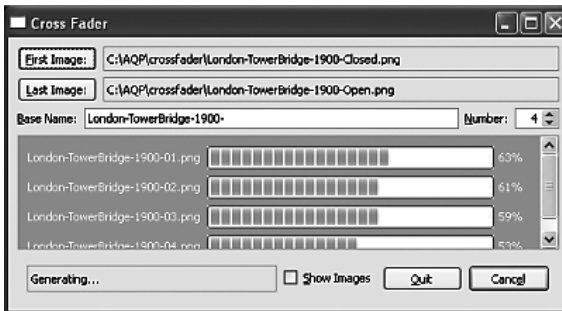


Рис. 8.2. Приложение Cross Fader



Рис. 8.3. Шесть изображений, из которых четыре микшированных

В приложении Cross Fader для создания каждого микшированного изображения используется свой экземпляр подкласса `QThread`. Это разумно, потому что программа ограничивает количество промежуточных изображений. Кроме того, поскольку обработка производится независимо, то синхронизация не нужна. Если бы нужно было параллельно создавать много изображений, то, наверное, было бы лучше использовать несколько рабочих потоков и разделяемую между ними очередь; такой подход мы применим в следующем разделе.

Приложение создает также виджеты `QLabel` и `QProgressBar` для каждого микшированного изображения, чтобы пользователь мог следить за ходом работы. Но число промежуточных изображений заранее не известно (оно задается с помощью счетчика с максимальным значением 14), и мы создаем столько потоков, меток и индикаторов, сколько изображений закажет пользователь. Поэтому все метки и индикаторы размещены внутри области `QScrollArea`, чтобы при необходимости появлялась вертикальная полоса прокрутки.

Мы решили ограничиться одной кнопкой для запуска и отмены генерации изображений и в зависимости от ситуации выводить на ней надпись Generate (Генерировать) или Cancel (Отмена). Кроме того, если отмечен флажок Show Images (Показать изображения), то после создания всех изображений мы показываем их в системной программе просмотра изображений (если таковая имеется).

Как и в предыдущей главе, мы сначала опишем базовую инфраструктуру, чтобы было понятно, о чем речь, а потом перейдем к коду, относящемуся к многопоточной обработке, – в данном случае подклассу CrossFader класса QThread, который применяется для создания микшированных изображений.

Сначала покажем перечисление и несколько закрытых членов класса MainWindow.

```
enum StopState {Stopping, Terminating};

QWidget *progressWidget;

QMap<QString, QPointer<QProgressBar> > progressBarForFilename;
QList<QPointer<QLabel> > progressLabels;
QList<QPointer<CrossFader> > crossFaders;
bool canceled;
```

Перечисление StopState различает причины завершения приложения: из-за прерывания пользователем или естественное. Почему это так важно, мы увидим при рассмотрении метода cleanUp().

Виджет progressWidget помещен внутрь QScrollArea и содержит компоновщик QGridLayout. Когда пользователь запускает генерацию изображений, мы создаем новый набор меток и индикаторов и размещаем их внутри компоновщика. Указатели на индикаторы хранятся в объекте QMap; ключом является имя графического файла, построение которого отображает индикатор.

Вместо обычных указателей на изображения мы используем защищенные указатели (guarded pointer) QPointer, которые автоматически принимают значение null, когда объект, на который они указывают, удаляется. Как и все прочие интеллектуальные указатели, QPointer обходится чуть дороже обычного указателя – он потребляет больше памяти, а доступ через него несколько медленнее. Но все затраты окупаются удобством проверки в ситуации, когда мы хотим обращаться к объекту *только* в том случае, если он не удален. (Другие интеллектуальные указатели, имеющиеся в Qt, были описаны во врезке «Интеллектуальные указатели в Qt» на стр. 78.)

Класс QWeakPointer, появившийся в версии Qt 4.6, можно использовать как слабый указатель общего вида или, для подклассов QObject, как более эффективную замену QPointer. Однако интерфейс класса QWeakPointer отличается от API QPointer и не так удобен, поэтому здесь мы предпочита-

таем использовать `QPointer`, поскольку его относительная неэффективность теряется на фоне затрат на обработку изображений (и, стало быть, несущественна). Кроме того, применение `QPointer` означает, что код будет компилироваться без изменений и в Qt 4.5, и в Qt 4.6.

Список микшеров `crossFaders` нужен для того, чтобы было проще производить различные действия сразу над всеми работающими вспомогательными потоками, – например, останавливать их, когда пользователь прерывает генерацию. Зачем для хранения указателей на микшеры используется класс `QPointer`, мы увидим, когда будем обсуждать метод `cleanUp()`. Булевый флаг `canceled` введен только для нужд пользовательского интерфейса, вспомогательные потоки к нему не обращаются, поэтому добавлять модификатор `volatile` нет необходимости.

Кнопка `Generate` становится активна сразу после того, как пользователь выберет оба изображения (благодаря соединениям сигналов со слотами, которые мы опустили). При нажатии на эту кнопку вызывается слот `generateOrCancelImages()` и начинается генерация.

```
void MainWindow::generateOrCancelImages()
{
    if (generateOrCancelButton->text() == tr("G&enerate")) {
        generateOrCancelButton->setEnabled(false);
        statusBar->showMessage(tr("Generating..."));
        canceled = false;
        cleanUp();
        QImage firstImage(firstLabel->text());
        QImage lastImage(lastLabel->text());
        for (int i = 0; i < numberSpinBox->value(); ++i)
            createAndRunACrossFader(i, firstImage, lastImage);
        generateOrCancelButton->setText(tr("Canc&el"));
    }
    else {
        canceled = true;
        cleanUp();
        generateOrCancelButton->setText(tr("G&enerate"));
    }
    updateUi();
}
```

Этот слот вызывается при нажатии кнопки `generateOrCancelButton`, но его поведение зависит от того, используется эта кнопка для запуска или прерывания генерации.

Если нажата кнопка `Generate`, то производятся различные действия, относящиеся к интерфейсу, например изменение строки состояния, и флаг `canceled` сбрасывается в `false`. Кроме того, мы вызываем метод `cleanUp()`, чтобы гарантированно остановить все вспомогательные потоки и удалить ранее созданные метки и индикаторы хода выполнения, подготовившись тем самым к новому запуску.

Затем создается по одному объекту-микшеру для каждого промежуточного изображения, и надпись кнопки `generateOrCancelButton` меняется на `Cancel`.

Если же нажата кнопка `Cancel`, то флагу `canceled` присваивается значение `true`, чтобы другие методы главного окна узнали о том, что генерация была прервана. Затем мы производим очистку и превращаем кнопку обратно в `Generate`.

В любом случае в заключение вызывается метод `updateUi()` (не показан); он всего лишь активирует или деактивирует кнопку `Generate` в зависимости от того, выбраны уже оба изображения или еще нет.

Как обычно, клавиши-акселераторы устанавливаются автоматически; в данном случае функция `AQP::accelerateWidget(this)` вызывается в конструкторе после создания виджетов. Однако метод `generateOrCancelImages()` должен знать, какая надпись в данный момент отображается на кнопке `Generate/Cancel`, а мы не можем заранее сказать, куда `AQP::accelerateWidget()` поставит амперсанд. Можно было бы перед сравнением с надписью на кнопке удалить амперсанд, например, так: `generateOrCancelButton->text().replace("&", "") == tr("Generate")`. Но мы предпочли поставить амперсанды в обоих надписях, отображаемых на этой кнопке, вручную, чтобы с ней была связана одна и та же клавиша-акселератор `Alt+E` вне зависимости от того, что отображается: `Generate` или `Cancel`. Функции `accelerate*()` не изменяют заданные вручную клавиши-акселераторы. (Подробнее о модуле *alt_key.{hpp,cpp}*, в котором находятся функции `accelerate*()`, см. во врезке «Клавиши-акселераторы», стр. 30.)

Теперь пора рассмотреть метод `cleanUp()`. Он вызывается перед генерацией нового набора изображений, в случае прерывания генерации и при завершении программы. Для большей понятности мы разобьем этот метод на две части. Сначала приведем наивную реализацию первой части, потом обсудим, какая в ней скрыта проблема, а затем покажем корректную реализацию. А уж после этого перейдем ко второй части, одинаковой в обоих случаях.

Первая часть метода `cleanUp()` связана с остановом работающих потоков микширования. Сначала покажем наивную версию:

```
void MainWindow::cleanUp(StopState stopState)
{
    foreach (CrossFader *crossFader, crossFaders) { // Наивно!
        crossFader->stop();
        crossFader->wait();
        crossFader->deleteLater();
    }
    crossFaders.clear();
}
```

Это простейший подход. Мы обходим все объекты-микшеры и для каждого вызываем метод `CrossFader::stop()`. Время на это практически не тратится, так как единственное, что делает метод, — установка для

некоторой булевой переменной значения `true`. Затем вызывается метод `QThread::wait()` — микшер знает, что должен остановиться, поэтому завершится, как только дойдет до выражения `if (m_stopped)`. Когда метод `QThread::wait()` вернет управление, мы понимаем, что поток остановлен, и просим его удалить себя. А в самом конце очищаем список `crossFaders`, поскольку в данный момент в нем находятся висячие указатели.

Это решение настолько просто, что даже не нужны интеллектуальные указатели `QPointer` на объекты `CrossFader`, достаточно и обычных. Но у него есть один потенциальный недостаток: мы ждем завершения потоков по очереди, и значит, общее время ожидания составит сумму времен, необходимых для завершения каждого потока. Если мы готовы применить более сложный алгоритм, то эту проблему можно устранить и сократить время ожидания, сделав его почти равным времени завершения самого медленного потока.

Конечно, ничего дурного в использовании простого подхода нет и переходить к более сложному следует лишь в том случае, когда на реальных тестах будет показано, что он существенно повышает производительность. Но даже в этом простом решении лучше ввести два цикла: первый для останова, второй для ожидания и удаления. Мы убедимся в этом, когда будем обсуждать альтернативу.

В исходном коде программы `Cross Fader` присутствуют обе реализации; какая именно компилируется, зависит от того, определен ли (`#define`) специальный макрос препроцессора¹. Вот вариант реализации, выбираемый по умолчанию:

```
const int StopWait = 100;

void MainWindow::cleanUp(StopState stopState)
{
    foreach (CrossFader *crossFader, crossFaders)
        crossFader->stop();
    while (crossFaders.count()) {
        QMutableListIterator<QPointer<CrossFader> > i(crossFaders);
        while (i.hasNext()) {
            CrossFader *crossFader = i.next();
            if (crossFader) {
                if (crossFader->wait(StopWait)) {
                    delete crossFader;
                    i.remove();
                }
            }
            else
                i.remove();
        }
    }
}
```

¹ Он называется `slow_stop`; см. `defines` в *pro*-файле программы. — *Прим. науч. ред.*

Первым делом метод предлагает каждому микшеру остановиться. Как мы уже отмечали, вызов метода `CrossFader::stop()` занимает очень мало времени, потому что сводится к присваиванию значения `true` некоей булевой переменной. Итак, в конце цикла каждый микшер знает, что должен остановиться.

В следующем цикле мы ждем, пока микшеры действительно остановятся, чтобы их можно было удалить. Но зачем нам два цикла? Затем, что если вызвать `stop()`, а потом сразу `wait()` (блокирующий вызов), то микшеры будут останавливаться последовательно, а не параллельно, так что общее время останова будет равно сумме времен останова каждого микшера. Мы же хотим, чтобы общее время по возможности приближалось ко времени останова самого медленного потока. Поэтому в первом цикле мы просим все потоки остановиться и ждем их завершения во втором цикле.

В нашем алгоритме останова потоков мы раз за разом обходим все потоки, выбирая их из списка. Если микшер уже был удален, то благодаря свойствам класса `QPointer` указатель на него окажется нулевым, поэтому мы можем просто исключить его из списка (что и делается в ветви `else`). Если же микшер все еще существует, то мы вызываем его метод `QThread::wait()`. Обычно этот метод ожидает «вечно», но в данном случае мы ждем всего 100 мс. Если за это время поток завершится, то `wait()` вернет `true`, и тогда мы удаляем поток и исключаем его из списка. В противном случае не делаем ничего, но на следующей итерации цикла попытаемся еще раз.

По существу, мы даем каждому потоку столько 100-миллисекундных квантов времени, сколько ему потребуется для останова. Но даже если потоку нужно много времени, чтобы остановиться, он не мешает останавливаться прочим потокам, так как сразу по истечении тайм-аута мы переходим к следующему потоку. В результате общее время останова оказывается гораздо ближе к времени останова самого медленного потока, чем при суммировании времен.

В приложении `Cross Fader` такое усложнение алгоритма не дает значимого выигрыша по сравнению с простым решением. Дело в том, что метод `QImage::save()` блокирует выполнение потока и является медленным (в той мере, в какой доступ к диску медленнее процессорных операций), так что любой поток, начавший запись на диск, должен будет дожидаться ее завершения. Поэтому квантование времени не дает ожидаемого преимущества.

Вторая часть метода `cleanUp()` не зависит от алгоритма останова потоков; она приведена ниже.

```
if (stopState == Terminating)
    return;
foreach (QProgressBar *progressBar, progressBarForFilename)
    if (progressBar)
```



```

        progressBar->deleteLater();
    progressBarForFilename.clear();
    foreach (QLabel *progressLabel, progressLabels)
        if (progressLabel)
            progressLabel->deleteLater();
    progressLabels.clear();
}

```

Если после останова всех потоков выясняется, что мы выходим из приложения, то все, что нужно, уже сделано, и можно возвращать управление. Если же метод вызван вследствие подготовки к новой или прерывания текущей генерации, то мы должны удалить все метки и индикаторы, оставшиеся от предыдущей генерации (если таковая имела место).

Удаление не вызывает никаких сложностей: мы просто обходим все индикаторы и метки и планируем удаление тех из них, которые реально существуют, на более позднее время, когда у цикла обработки сообщений появится такая возможность. А затем очищаем контейнер, в котором они хранятся. Нам нет нужды заботиться об этом в случае, когда приложение завершается, потому что метки и индикаторы помещены внутрь компоновщика и, значит, у них есть родитель, который удалит их естественным для Qt образом, когда будет удален сам.

Удаление меток и индикаторов с последующим созданием новых, конечно, менее эффективно, чем повторное использование. Но для повторного использования понадобилось бы написать больше кода – например, чтобы скрыть ранее созданные, но более не нужные виджеты (из-за того, что для новой генерации пользователь заказал меньше промежуточных изображений) или создать дополнительные (потому что заказано больше промежуточных изображений). Кроме того, если пользователь передумал и вместо первоначальных 14 промежуточных изображений заказал всего 5, мы будем держать в памяти 9 ненужных меток и индикаторов. Удаление и повторное создание позволяет сэкономить память за счет увеличения времени. Но это разумный компромисс, потому что время создания всего одного микшированного изображения заведомо превышает время удаления и создания меток и индикаторов хода выполнения.

Все потоки микширования создаются и запускаются в методе `generateOrCancelImages()` путем обращения к методу `createAndRunACrossFader()`, который мы рассмотрим в два приема.

```

void MainWindow::createAndRunACrossFader(int number,
    const QImage &firstImage, const QImage &lastImage)
{
    QString filename = QString("%1%2.png")
        .arg(baseNameEdit->text())
        .arg(number + 1, 2, 10, QChar('0'));
    QLabel *progressLabel = new QLabel(filename);

```



```

progressLabels << progressLabel;
QProgressBar *progressBar = new QProgressBar;
progressBar->setRange(0, 100);
progressBarForFilename[filename] = progressBar;
QGridLayout *layout = qobject_cast<QGridLayout*>(
    progressWidget->layout());
Q_ASSERT(layout);
layout->addWidget(progressLabel, number, 0);
layout->addWidget(progressBar, number, 1);

```

Аргумент `number` — это номер создаваемого микшированного изображения (отсчитываемый от 0). Сначала мы формируем имя соответствующего графического файла (`Image-01.png`, `Image-02.png` и т. д.), используя двузначные десятичные числа, начиная с 1. Затем создаем новую метку и добавляем ее в список меток, а потом новый индикатор, который добавляем в словарь индикаторов с ключом, равным имени файла. После этого получаем указатель на табличный компоновщик, ассоциированный с виджетом `progressWidget`, — который благодаря предшествующему вызову `cleanUp()` сейчас пуст — и добавляем метку и индикатор в очередную свободную строку, чей номер совпадает с номером изображения.

```

double firstWeight = (number + 1) /
    static_cast<double>(numberSpinBox->value() + 1);
double secondWeight = 1.0 - firstWeight;
CrossFader *crossFader = new CrossFader(filename, firstImage,
    firstWeight, lastImage, secondWeight, this);
crossFaders << crossFader;
connect(crossFader, SIGNAL(progress(int)),
    progressBar, SLOT(setValue(int)));
connect(crossFader, SIGNAL(saving(const QString&)),
    this, SLOT(saving(const QString&)));
connect(crossFader, SIGNAL(saved(bool, const QString&)),
    this, SLOT(saved(bool, const QString&)));
connect(crossFader, SIGNAL(finished()),
    this, SLOT(finished()));
crossFader->start();
}

```

Подготовив пользовательский интерфейс, мы далее создаем и запускаем поток, в котором будет создаваться микшированное изображение. Сначала вычисляются веса; например, если пропорциональные доли изображений составляют (60:40), то веса будут равны 0.6 и 0.4. Конструктору объекта `CrossFader` мы передаем имя файла, начальное и конечное изображения и веса. Созданный объект помещается в список микшеров, чтобы впоследствии его было легко удалить.

Так как `CrossFader` — подкласс `QThread` (а тот является подклассом `QObject`), то мы можем воспользоваться механизмом сигналов и слотов, а не создавать пользовательские события и не вызывать слоты вручную, как делали в примерах из предыдущей главы.

Мы соединяем сигнал `progress()` микшера прямо с индикатором хода выполнения, а остальные пользовательские сигналы, равно как и сигнал `QThread::finished()`, — с соответствующими слотами главного окна. В самом конце вызывает метод `QThread::start()`, который запускает поток.

```
const int StatusTimeout = AQP::MSecPerSecond * 10;

void MainWindow::saving(const QString &filename)
{
    statusBar->showMessage(tr("Saving '%1'").arg(filename),
                          StatusTimeout);
    if (QProgressBar *progressBar = progressBarForFilename[filename])
        progressBar->setRange(0, 0);
}
```

Завершив создание изображения, но еще до начала сохранения микшер генерирует сигнал `saving()`, что приводит к вызову показанного выше слота. Этот слот выводит сообщение для пользователя в строку состояния. А кроме того, устанавливает диапазон индикатора хода выполнения (0, 0); эта специальная конфигурация означает, что нужно показывать индикатор занятости, а не процент выполнения, — разумное решение, так как мы заранее не знаем, сколько времени займет сохранение.

```
void MainWindow::saved(bool saved, const QString &filename)
{
    const QString message = saved ? tr("Saved '%1'")
                                  : tr("Failed to save '%1'");
    statusBar->showMessage(message.arg(filename), StatusTimeout);
    if (QProgressBar *progressBar =
        progressBarForFilename[filename]) {
        progressBar->setRange(0, 1);
        progressBar->setValue(saved ? 1 : 0);
        progressBar->setEnabled(false);
    }
}
```

Закончив сохранение изображения, микшер генерирует сигнал `saved()`, который приводит к вызову этого слота. Булевый аргумент `saved` — это значение, возвращенное методом `QImage::save()`, оно показывает, успешно или нет завершилось сохранение. Как и `saving()`, этот метод выводит для пользователя сообщение в строке состояния. Затем он обновляет соответствующий индикатор, устанавливая произвольный диапазон (в котором, однако, максимум должен быть больше минимума) и присваивая текущему значению максимальную величину, если изображение успешно сохранено (тогда полоска индикатора будет заполнена на 100%), и минимальную в противном случае (тогда полоска индикатора окажется незаполненной). Кроме того, индикатор делается неактивным, чтобы показать, что работа над данным изображением завершена.

```
void MainWindow::finished()
{
}
```

```

foreach (CrossFader *crossFader, crossFaders)
    if (crossFader && !crossFader->isFinished())
        return;
generateOrCancelButton->setText(tr("Generate"));
if (canceled)
    statusBar->showMessage(tr("Canceled"), StatusTimeout);
else {
    statusBar->showMessage(tr("Finished"));
    if (statusBar->checkBox()->isChecked())
        QDesktopServices::openUrl(QUrl::fromLocalFile(
            firstLabel->text()));
}
}

```

Помимо сигналов `saving()` и `saved()`, микшер по завершении работы генерирует сигнал `finished()`, в результате чего вызывается этот слот. Он перебирает все объекты-микшеры и, обнаружив хотя бы один незавершенный, возвращает управление, ничего не сделав, так как генерация еще продолжается.

Если же все микшеры закончили работу, то надпись кнопки меняется с `Generate` на `Cancel` и в строку состояния выводится информационное сообщение. Если причина завершения – не прерывание, и флажок `Show Images` отмечен, то вызывается метод `QDesktopServices::openUrl()`, которому в качестве URL передается имя файла первого изображения со схемой протокола `file://`. Для получения такого URL применяется статический метод `QUrl::fromLocalFile()`¹. Если методу `openUrl()` передан URL-адрес со схемой `http://`, то он пытается запустить системный веб-браузер (или открыть новую вкладку, если браузер уже запущен²) и перейти на страницу с указанным адресом. В случае же, когда в переданном URL указана схема `file://`, запускается системное приложение, ассоциированное с суффиксом файла, если такая ассоциация существует. Таким образом, если на данном компьютере установлено приложение для просмотра изображений, то оно будет запущено и получит в качестве параметра имя файла первого изображения.

Некоторые программы просмотра показывают заданное изображение, а также миниатюры всех прочих изображений в том же каталоге, позволяя легко переходить от одного к другому. Разумеется, было бы не сложно добавить средство просмотра изображений прямо в приложение `Cross Fader`; оставляем это читателю в качестве упражнения.

Обращение к методу `statusBar->checkBox()` выглядит не вполне обычно. Вместо того чтобы воспользоваться классом `QStatusBar`, мы создали пользовательский виджет `StatusButtonBar` (не показан), включающий

¹ В версии Qt 4.6 появился еще один статический метод, `QUrl::fromUserInput()`, который принимает строку и возвращает объект `QUrl` со схемой `file://`, `ftp://` или `http://` в зависимости от входной строки.

² И настроен соответствующим образом. – *Прим. науч. ред.*

QLabel, QCheckBox и QDialogButtonBox, размещенные внутри компоновщика QHBoxLayout. Именно поэтому они все расположены на одном уровне, а не над строкой состояния (см. рис. 8.2 на стр. 315).

На этом мы завершили рассмотрение структуры пользовательского интерфейса и можем перейти к подклассу CrossFader класса QThread, где и происходит основная работа. Сначала приведем объявление класса, взятое из заголовочного файла.

```
class CrossFader : public QThread
{
    Q_OBJECT

public:
    explicit CrossFader(const QString &filename,
                        const QImage &first,
                        const double &firstWeight, const QImage &last,
                        const double &lastWeight, QObject *parent=0);

public slots:
    void stop() { m_stopped = true; }

signals:
    void progress(int);
    void saving(const QString&);
    void saved(bool, const QString&);

private:
    void run();
    ...
    volatile bool m_stopped;
```

Переменная `m_stopped` с модификатором `volatile` нужна для того, чтобы известить поток о необходимости прекратить работу. (О булевых переменных с таким модификатором мы говорили в предыдущей главе на стр. 272.) Выше мы видели слоты, соединенные с тремя сигналами, определенными в этом классе; сигнал `finished()` унаследован от `QThread`. Мы опустили закрытые переменные, в которых хранятся аргументы, переданные конструктору (`m_filename`, `m_first` и т. д.).

```
CrossFader::CrossFader(const QString &filename,
                       const QImage &first,
                       const double &firstWeight, const QImage &last,
                       const double &lastWeight, QObject *parent)
: QThread(parent),
  m_filename(filename), m_firstWeight(firstWeight),
  m_lastWeight(lastWeight), m_stopped(false)
{
    QSize size = first.size().boundedTo(last.size());
    m_first = first.scaled(size, Qt::IgnoreAspectRatio,
                          Qt::SmoothTransformation);
```

```

        m_last = last.scaled(size, Qt::IgnoreAspectRatio,
                             Qt::SmoothTransformation);
    }

```

При создании микшера мы первым делом определяем размер меньшего из двух изображений – или обоих, если размеры одинаковы. (Кстати, метод `QSize::expandedTo()` возвращает больший из двух размеров: объекта `QSize`, от имени которого он вызван, и объекта `QSize`, переданного в качестве аргумента.) Затем оба изображения масштабируются до размера наименьшего из них, чтобы все изображения, с которыми мы работаем, в том числе и микшированные, имели один и тот же размер.

При вызове метода `QImage::scaled()` Qt сравнивает размеры исходного и переданного изображений, и если они одинаковы, то просто возвращает первое. Возврат изображения обходится дешево, так как Qt применяет механизм копирования при записи, поэтому реально передается только указатель или чуть больше данных.

Для простоты мы разобьем рассмотрение метода `run()` на две части. При этом покажем две версии первой части: одна простая, но работает медленно, другая – чуть сложнее, зато очень быстрая.

```

void CrossFader::run()
{
    QImage image(m_first.width(), m_first.height(), QImage::Format_RGB32);
    emit progress(0);

    const float onePercent = image.width() / 100.0;
    // Просто, но медленно!
    for (int x = 0; x < image.width(); ++x) {
        for (int y = 0; y < image.height(); ++y) {
            QRgb firstPixel = m_first.pixel(x, y);
            QRgb lastPixel = m_last.pixel(x, y);
            int red = qRound((qRed(firstPixel) * m_firstWeight) +
                             (qRed(lastPixel) * m_lastWeight));
            int green = qRound((qGreen(firstPixel) * m_firstWeight) +
                               (qGreen(lastPixel) * m_lastWeight));
            int blue = qRound((qBlue(firstPixel) * m_firstWeight) +
                              (qBlue(lastPixel) * m_lastWeight));
            image.setPixel(x, y, qRgb(red, green, blue));
            if ((y % 64) == 0 && m_stopped)
                return;
        }
        if (m_stopped)
            return;
        emit progress(qRound(x / onePercent));
    }
}

```

Именно в этом методе и производится вся работа. Сначала мы создаем новый объект `QImage` нужного размера с 32-разрядным цветом, заданным в формате RGB. (Нетрудно было бы добавить и альфа-канал – про-

зрачность, – но это лишь удлинило бы код, не изменив принципиально его структуры, так что оставляем это читателю в качестве упражнения.) Создав изображение, мы генерируем первый сигнал `progress()`. Слот, с которым соединен этот сигнал, находится в другом потоке – основном потоке пользовательского интерфейса. Сигналы, передаваемые между потоками, Qt преобразует в события, которые добавляются в очередь сообщений потока-приемника. Когда основной поток доходит до такого события, он вызывает слот, соединенный с сигналом (передавая ему аргументы, посланные вместе с сигналом), так что этот слот выполняется в основном потоке, а не в том, где был сгенерирован сигнал.

Для выполнения микширования мы читаем RGB-значения соответствующих пикселей первого и последнего изображений и вычисляем их взвешенную сумму с округлением. Например, если в пикселе первого изображения красный компонент равен 240 и вес первого изображения составляет 0.6, а в соответствующем пикселе второго изображения красный компонент равен 120 и вес второго изображения составляет 0.4, то доля красного в пикселе микшированного изображения будет равна $192: (240 \times 0,6) + (120 \times 0,4) = 144 + 48 = 192$.

Метод `QImage::pixel()` возвращает беззнаковое целое; `QRgb` – это просто `typedef`, проясняющий семантику. Функции `qRed()`, `qGreen()` и `qBlue()` получают на входе значение типа `QRgb` и возвращают целое – соответственно красный, зеленый и синий компоненты цвета. Имеется еще `typedef QRgba`, включающий альфа-канал (прозрачность), и функция `qAlpha()`, а также функция `qGray()`, порождающая оттенок серого цвета по данному значению типа `QRgb` (или по трем целым числам, задающим величины красного, зеленого и синего компонентов).

Во внешнем цикле обходятся столбцы матрицы изображения, а во внутреннем – строки. Чтобы вовремя распознать прерывание от пользователя, на каждом 64-м пикселе мы проверяем, не надо ли остановить поток. В конце каждого столбца мы еще раз проверяем условие останова и генерируем сигнал `progress()`, передавая номер только что обработанного столбца.

К сожалению, методы `QImage::pixel()` и `QImage::setPixel()` работают слишком медленно. Есть две альтернативы, и обе основаны на знании того, что в классе `QImage` данные пикселей хранятся в одном непрерывном массиве. Первое решение состоит в том, чтобы работать на уровне горизонтальных строк изображения. Метод `QImage::scanLine()` возвращает указатель на одну «строку» пикселей. Если работать с ней напрямую вместо чтения и установки отдельных пикселей, то можно добиться существенного ускорения. (В исходном коде этот вариант присутствует, но здесь мы его опускаем.) Другое решение – работать на уровне самого массива пикселей. Доступ к нему дает метод `QImage::bits()`, и такой подход оказывается самым быстрым. Ниже показано, как достичь того же результата, что и выше, с применением метода `QImage::bits()`:

```

void CrossFader::run()
{
    QImage image(m_first.width(), m_first.height(),
                 QImage::Format_RGB32);
    emit progress(0);

    const int onePercent = qRound(image.width() * image.height() /
                                   100.0);
    QRgb *firstPixels = reinterpret_cast<QRgb*>(m_first.bits());
    QRgb *lastPixels = reinterpret_cast<QRgb*>(m_last.bits());
    // Самый быстрый способ
    QRgb *pixels = reinterpret_cast<QRgb*>(image.bits());
    for (int i = 0; i < image.width() * image.height(); ++i) {
        QRgb firstPixel = firstPixels[i];
        QRgb lastPixel = lastPixels[i];
        int red = qRound((qRed(firstPixel) * m_firstWeight) +
                        (qRed(lastPixel) * m_lastWeight));
        int green = qRound((qGreen(firstPixel) * m_firstWeight) +
                          (qGreen(lastPixel) * m_lastWeight));
        int blue = qRound((qBlue(firstPixel) * m_firstWeight) +
                          (qBlue(lastPixel) * m_lastWeight));
        pixels[i] = qRgb(red, green, blue);
        if ((i % onePercent) == 0) {
            if (m_stopped)
                return;
            emit progress(i / onePercent);
        }
    }
}

```

Мы вызываем `QImage::bits()` трижды — для получения доступа на чтение-запись ко всем трем изображениям. Затем вместо двух циклов — по столбцам и по строкам — мы обходим все биты в одном цикле. (В исходном коде вариант решения выбирается с помощью директивы `#if`; рекомендуем поэкспериментировать, чтобы увидеть, насколько подход на основе `QImage::bits()` быстрее.) Единственное, что замедляет работу, — это оператор `if`; без него код работал бы еще быстрее, но пользователю пришлось бы дольше ждать прерывания генерации, а индикация хода выполнения оказалась бы гораздо грубее.

```

    emit progress(image.width());
    if (m_stopped)
        return;
    emit saving(m_filename);
    if (m_stopped)
        return;
    emit saved(image.save(m_filename), m_filename);
}

```

Закончив создание нового изображения, мы генерируем сигнал `progress()` и, если пользователь не прервал операцию, еще и сигнал `saving()`. Далее мы пытаемся сохранить изображение и генерируем сигнал `saved()`,

извещая о результате работы метода `QImage::save()`. Сигнал `finished()` явно не генерируется, это сделает базовый класс по выходе из метода `run()`.

Использовать класс `QThread` в этом приложении оказалось нетрудно, потому что не потребовалось синхронизировать доступ и мы смогли в полной мере воспользоваться механизмом сигналов и слотов. Но если бы количество параллельно создаваемых изображений было велико, то применение разделяемой очереди работ и небольшого числа потоков – подход, рассматриваемый в следующем разделе, – позволило бы получить более масштабируемое решение.

Обработка разделяемых данных

Если количество подлежащих обработке элементов велико и их размеры сильно различаются, так что мы заранее не знаем, как равномерно распределить их между потоками для достижения максимальной пропускной способности, то будет лучше воспользоваться разделяемой очередью работ. В ряде случаев мы можем сначала поместить в очередь несколько работ и запустить фиксированное число потоков, которые приступят к обработке, а затем добавлять работы по мере необходимости. Если сделать доступ к очереди работ потокобезопасным (то есть включить какой-то механизм синхронизации), то каждый поток сможет обращаться с очередью как с обычной структурой данных.

Но в этом разделе мы будем иметь дело с несколько иным сценарием: сначала используем вспомогательные потоки для заполнения разделяемой структуры данных (хеша), а затем приложение заполнит модель, которая отобразит находящиеся в хеше данные с помощью представления.

Мы разработаем приложение `Find Duplicates (findduplicates)`, показанное на рис. 8.4. Оно будет искать в указанном пользователем каталоге и его подкаталогах файлы-дубликаты¹. Дубликаты определяются по совпадению размеров файлов и их MD5-сумм. MD5 (алгоритм Message-Digest версии 5) – это криптографическая функция, которая получает на входе блок данных (например, файл) и возвращает 128-разрядную (16 байт) свертку. Если размеры и MD5-суммы двух файлов одинаковы, то с высокой вероятностью одинаково и их содержимое.

Приложение обходит все каталоги и заполняет хеш `QHash<QPair<QByteArray, qint64>, QStringList>`. В качестве ключа хеша выступает пара, состоящая из массива байтов – MD5-суммы – и числа типа `qint64` – разме-

¹ Кстати, как и в приложении `Image2Image`, рассмотренном в предыдущей главе, со строкой редактирования, где пользователь вводит имя начального каталога, ассоциирован объект `QCompleter`, который выводит список подходящих каталогов, – чтобы пользователю не надо было вводить имя целиком. Этот объект обсуждается в главе 9, на стр. 349.

ра файла, а значением является список строк `QStringList`, содержащий полные пути к файлам с данной MD5-сверткой и размером. Далее мы можем обойти заполненный хеш и посмотреть, в каких записях список содержит более одной строки, – это и будут файлы-дубликаты. Одно из достоинств такого подхода заключается в том, что мы находим файлы-дубликаты, даже если их имена различаются. (См., например, строку с именем `w39MLRes.DLL` на рис. 8.4.)

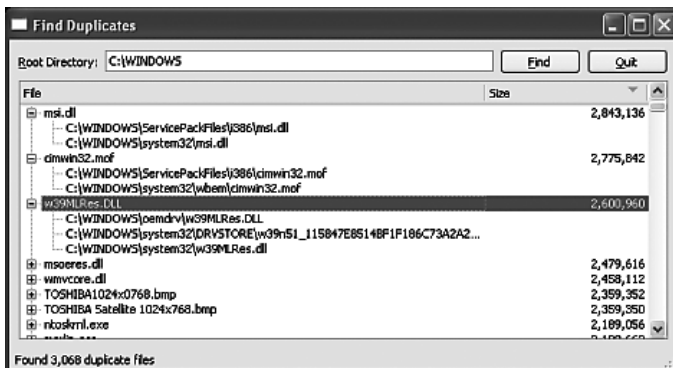


Рис. 8.4. Приложение *Find Duplicates*

Вычисление MD5-суммы файла – потенциально дорогостоящая операция (время ее выполнения пропорционально размеру файла), поэтому мы хотели бы распределить работу между одним или несколькими вспомогательными потоками (их число равно идеальному для данной машины количеству потоков, полученному от Qt). Но зачем вообще вычислять сумму, если можно в массиве `QByteArray`, являющемся частью ключа, хранить все содержимое файла, тем более что для вычисления свертки файл все равно придется прочитать? Но если так поступить, то ключи хеша займут столько же памяти, сколько все содержимое каталога (быть может, десятки, сотни мегабайтов, а то и больше), тогда как MD5-сумма занимает всего 16 байт вне зависимости от размера файла.

Приложение должно знать, какие файлы ему предстоит обработать (то есть вычислить сумму и затем поместить в хеш). Мы решили создать список подкаталогов указанного пользователем каталога и по возможности равномерно распределить его между вспомогательными потоками. Далее каждый поток должен будет помещать в хеш записи для всех обработанных им файлов. Это не гарантирует справедливого распределения работы – например, одному из потоков может достаться каталог, содержащий только крохотные пиктограммы, а другому – каталог с кучей музыкальных файлов или образов DVD.

Чтобы гарантировать справедливое распределение работы, мы могли бы поступить следующим образом: создать структуру данных, содержащую все файлы, например отображение `QMap<qint64, QString>` с размером

файла в качестве ключа и именем в качестве значения; тогда записи словаря оказались бы внутренне упорядочены по размеру файла. После этого нужно было бы распределить файлы по потокам в соответствии с размерами. Например, если есть три потока и двадцать файлов, то работу нужно было бы распределить так: поток 1 – [0, 3, 6, 9, 12, 15, 18]; поток 2 – [1, 4, 7, 10, 13, 16, 19]; поток 3 – [2, 5, 8, 11, 14, 17] (мы указываем индексы записей в отображении, хотя на практике воспользовались бы итератором). Сложность программы увеличилась бы при этом не сильно, но недостаток такого решения по сравнению с теоретически несправедливым заключается в том, что общее время работы может увеличиться, а уж памяти наверняка потребуется больше. И дело тут в дополнительных затратах на предварительное построение потенциально очень длинного списка файлов (даже если распределить эту работу по нескольким потокам) по сравнению с простым назначением каждому потоку своего каталога. Было бы любопытно сравнить оба решения на практике, но это мы оставим в качестве упражнения читателю.

Понятно, что в основе приложения лежит разделяемый хеш, который могут обновлять все вспомогательные потоки. Сделать полноценный потокобезопасный хеш – нетривиальная задача, поэтому мы решили создать урезанный хеш, предоставляющий необходимую программе функциональность, и опустить массу полезных, но не нужных нам возможностей.

Начнем с рассмотрения упрощенного до предела класса `ThreadSafeHash`. Затем ознакомимся со структурой главного окна и с подклассом `QThread`, в котором и происходит вся обработка.

Ниже приведено объявление класса `ThreadSafeHash`, его методы мы рассмотрим чуть позже.

```
template<typename Key, typename Value>
class ThreadSafeHash
{
public:
    explicit ThreadSafeHash() {}
    ...

private:
    mutable QReadWriteLock lock;
    QMultiHash<Key, Value> hash;
};
```

Разница между классами `QMultiHash` и `QHash` (а также между `QMultiMap` и `QMap`) заключается в том, что в первом случае с каждым ключом может быть связано несколько значений, а во втором – только одно. Иными словами, с ключом ассоциирован список значений. Это влечет за собой тонкие семантические отличия, которые нужно иметь в виду при работе с классами `QMultiHash` и `QMultiMap`. (На самом деле в классах `QHash` и `QMap` имеется метод `insertMulti()`, который позволяет сохранить

несколько значений для одного ключа, но использование QMultiHash и QMultiMap в случаях типа нашего удобнее.)

Мы воспользовались классом QReadWriteLock, чтобы минимизировать время ожидания блокировки, — например, если нет активных блокировок записи, то блокировку чтения может получить любое количество потоков. Переменная-член lock должна быть объявлена как mutable, потому что она используется в константных методах.

```
QList<Value> values(const Key &key) const
{
    QReadLocker locker(&lock);
    return hash.values(key);
}
```

Этот метод возвращает значения, соответствующие указанному ключу, или пустой список, если этот ключ отсутствует в хеше.

Конструктор QReadLocker блокирует выполнение, пока не сможет получить блокировку чтения для переданного ему объекта QReadWriteLock. При уничтожении объекта QReadLocker его деструктор снимает блокировку. Тем самым гарантируется, что блокировка обязательно будет снята, — неважно, произойдет ли выход из функции, в которой объявлен объект QReadLocker, нормально или в результате необработанного исключения.

В программе Find Duplicates ключи имеют тип QPair<QByteArray, qint64>, а значения — тип QString, поэтому данный метод возвращает список типа QList<QString> (быть может, пустой).

Тип QList<QString> совместим с типом QStringList — производным от QList, — но не содержит некоторых вспомогательных методов, имеющих в QStringList. При необходимости можно легко преобразовать QList<QString> в QStringList и получить недостающие методы, потому что имеется конструктор QStringList, принимающий аргумент типа QList<QString>.

В классе ThreadSafeHash есть еще несколько методов с аналогичной структурой, в которых создается объект QReadLocker, а затем возвращается результат обращения к методу агрегированного хеша. К их числу относятся методы contains(), count() и isEmpty() (не показаны).

```
void insert(const Key &key, const Value &value)
{
    QWriteLocker locker(&lock);
    hash.insert(key, value);
}
```

Этот метод вставляет в хеш одно значение с указанным ключом. Если в хеше оказывается несколько значений с одинаковым ключом, то они хранятся в порядке вставки. Если в хеше еще нет значения с таким ключом, то создается новый элемент с указанными ключом и значением.

Конструктор `QWriteLocker` блокирует выполнение, пока не получит блокировку записи для переданного ему объекта `QReadWriteLock`. При уничтожении объекта `QWriteLocker` его деструктор снимает блокировку – точно так же, как в случае `QReadLocker`.

```
const QList<Value> takeOne(bool *more)
{
    Q_ASSERT(more);
    QWriteLocker locker(&lock);
    typename QMultiHash<Key, Value>::const_iterator i =
        hash.constBegin();
    if (i == hash.constEnd()) {
        *more = false;
        return QList<Value>();
    }
    *more = true;
    const QList<Value> values = hash.values(i.key());
    hash.remove(i.key());
    return values;
}
```

Реализация потокобезопасного механизма обхода – непростое занятие, поэтому мы решили вместо этого написать метод, который удаляет элементы из хеша, поскольку для нужд приложения `Find Duplicates` этого достаточно.

Сначала мы захватываем блокировку записи, так как планируем изменять хеш. Нас не интересует какой-то определенный элемент (устроит любой), но нужно придумать способ получить этот элемент. Мы решили просто взять константный итератор, указывающий на «первый» элемент хеша. (Мы пишем «первый» в кавычках, потому что, в отличие от `QMap`, контейнеры `QHash` и `QMultiHash` не упорядочены.) К сожалению, простое объявление итератора не нравится некоторым компиляторам, поэтому мы были вынуждены добавить ключевое слово `typename`, чтобы ясно выразить свои намерения.

Если этот итератор указывает на позицию за концом хеша, значит, хеш пуст. В таком случае мы присваиваем булевой переменной, на которую указывает аргумент `more`, значение `false` и возвращаем пустой список значений.

Если же хеш не пуст, то мы записываем в `more` значение `true` и получаем список значений для элемента, на который указывает ключ, хранящийся в итераторе. (Мы не можем написать `i.value()`, так как этот метод возвращает единственное значение, а именно, первое в списке.) Получив копию списка значений, мы удаляем найденный элемент из хеша и возвращаем список. Далее мы увидим, что приложение `Find Duplicates` вызовет этот метод для заполнения модели после того, как все вспомогательные потоки закончат запись в потокобезопасный хеш.

Итак, все относящиеся к делу части класса `ThreadSafeHash` рассмотрены. Наличие такой структуры данных означает, что вспомогательные по-

токи могут трактовать объект `ThreadSafeHash` как обычную структуру данных, не заботясь о синхронизации доступа.

Теперь мы готовы заняться главным окном приложения и начнем с данных-членов (опуская указатели на модель и виджеты, которые не таят никаких сюрпризов).

```
volatile bool stopped;
QList<QPointer<GetMD5sThread> > threads;
FilesForMD5 filesForMD5;
```

Переменная `stopped` нужна для того, чтобы извещать потоки о необходимости прекратить работу. Хранить список вспомогательных потоков удобно, это станет ясно при рассмотрении методов `processDirectories()`, `finished()` и `stopThreads()`. Подкласс `GetMD5sThread` класса `QThread` мы рассмотрим ближе к концу этого раздела. Переменная `filesForMD5` — это разделяемый потокобезопасный хеш, который заполняют вспомогательные потоки.

```
typedef ThreadSafeHash<QPair<QByteArray, quint64>, QString> FilesForMD5;
```

Этот `typedef` мы определили для удобства, чтобы не писать каждый раз громоздкий тип хеша¹. Типом ключа хеша является `QPair`, а типом значений в списке, ассоциированным с каждым ключом, — `QString`.

```
void MainWindow::find()
{
    stopThreads();
    rootDirectoryEdit->setEnabled(false);
    view->setSortingEnabled(false);
    model->clear();
    model->setColumnCount(2);
    model->setHorizontalHeaderLabels(QStringList() << tr("File")
                                     << tr("Size"));

    findButton->hide();
    cancelButton->show();
    cancelButton->setEnabled(true);
    cancelButton->setFocus();
    stopped = false;
    prepareToProcess();
}
```

Этот метод вызывается, когда пользователь нажимает кнопку Find (Найти). Сначала мы останавливаем все работающие вспомогательные потоки, затем обновляем пользовательский интерфейс и очищаем модель, чтобы убрать с экрана ранее найденные дубликаты. Чтобы повысить производительность, лучше отключить в представлении сортировку

¹ Такое использование `typedef` встречается в этой книге редко. Мы вообще стараемся по возможности избегать использования `typedef`, чтобы не заставлять читателя запоминать их или возвращаться к месту, где они определены. Однако всячески поощряем их использование в реальных программах.

перед тем, как производить крупные изменения. А поскольку операция очистки модели заодно обнуляет счетчики строк и столбцов и стирает заголовки, то их следует восстановить, прежде чем приступить к заполнению модели. В самом конце мы сбрасываем переменную `stopped` в `false`, поскольку пользователь еще ничего не прервал, и вызываем метод `prepareToProcess()`, который сформирует список подлежащих обработке каталогов.

Кстати говоря, мы создали не одну кнопку, меняющую надпись с `Find` на `Cancel` и обратно, а две разных, расположив их рядом в компоновщике `QHBoxLayout`. В любой момент видна только одна кнопка. Это удобно, потому что не приходится менять надпись и можно написать два слота, `find()` и `cancel()`, вместо единственного `findOrCancel()`.

```
const int StopWait = 100;

void MainWindow::stopThreads()
{
    stopped = true;
    while (threads.count()) {
        QMutableListIterator<QPointer<GetMD5sThread> > i(threads);
        while (i.hasNext()) {
            QPointer<GetMD5sThread> thread = i.next();
            if (thread) {
                if (thread->wait(StopWait)) {
                    delete thread;
                    i.remove();
                }
            }
            else
                i.remove();
        }
    }
    Q_ASSERT(threads.isEmpty());
}
```

Алгоритм этого метода такой же, как метода `cleanUp()` из примера `Cross Fader` (стр. 321); его задача – приблизить общее время останова ко времени останова самого медленного потока.

```
void MainWindow::prepareToProcess()
{
    statusBar()->showMessage(tr("Reading files..."));
    QStringList directories;
    directories << rootDirectoryEdit->text();
    QDirIterator i(directories.first());
    while (!stopped && i.hasNext()) {
        const QString &pathAndFilename = i.next();
        const QFileInfo &info = i.fileInfo();
        if (info.isDir() && !info.isSymLink() &&
            i.fileName() != "." && i.fileName() != "..")
            directories << pathAndFilename;
    }
```

```

    }
    if (stopped)
        return;
    processDirectories(directories);
}

```

Этот метод строит список подлежащих обработке каталогов. Первым в нем идет каталог, введенный пользователем (корневой), все прочие (если таковые имеются) – подкаталоги корневого каталога. Даже если каталог велик, создание списка не займет много времени и памяти, потому что мы углубляемся только на один уровень. Построив список, мы вызываем метод `processDirectories()`, который и выполняет основную работу.

Если могут встретиться очень большие каталоги, то, возможно, имеет смысл поместить проверку `if (stopped)` внутрь цикла¹, чтобы пользователь мог прервать обработку, если передумает. Или даже завести для построения списка несколько потоков.

```

void MainWindow::processDirectories(const QStringList &directories)
{
    const QVector<int> sizes = AQP::chunkSizes(directories.count(),
        QThread::idealThreadCount());
    int offset = 0;
    foreach (const int chunkSize, sizes) {
        QPointer<GetMD5sThread> thread = QPointer<GetMD5sThread>(
            new GetMD5sThread(&stopped, directories.first(),
                directories.mid(offset, chunkSize),
                &filesForMD5));

        threads << thread;
        connect(thread, SIGNAL(readOneFile()), this, SLOT(readOneFile()));
        connect(thread, SIGNAL(finished()), this, SLOT(finished()));
        thread->start();
        offset += chunkSize;
    }
}

```

Здесь используется тот же алгоритм распределения работы, что и в методах `convertFiles()` из предыдущей главы (стр. 277). Конструктору каждого объекта `GetMD5sThread` передается указатель на переменную `stopped` (чтобы можно было обнаружить прерывание обработки пользователем), корневой каталог (поскольку его следует пропустить при рекурсии), каталоги, подлежащие обработке, и хеш, который этот поток будет обновлять.

Создав поток `GetMD5sThread`, мы добавляем его в список потоков. Затем соединяем определенный в этом классе сигнал `readOneFile()` и унасле-

¹ Собственно, проверка и так производится на каждой итерации цикла благодаря условию `!stopped` в его заголовке. Видно, автор запутался в собственном коде. – *Прим. перев.*

дованный сигнал `finished()` со слотами главного окна и запускаем выполнение потока.

```
void MainWindow::readOneFile()
{
    statusBar()->showMessage(tr("Read %Ln file(s)", "",
                                filesForMD5.count()));
}
```

Добавив очередное имя файла в хеш, объект `GetMD5sThread` генерирует сигнал `readOneFile()`, с которым соединен этот слот. Поскольку хеш `FilesForMD5` потокобезопасен, то мы можем не задумываться о синхронизации доступа к его счетчику; и понятно, что счетчик, возвращаемый методом `count()`, учитывает вклад всех потоков, поскольку они работают с одним и тем же хешем.

Строго говоря, значение счетчика некорректно, потому что относится только к количеству уникальных пар (MD5, размер), а не к количеству обработанных файлов; последнее может оказаться больше при наличии дубликатов. Тем не менее для нужд индикации хода выполнения этого счетчика достаточно. Это пример компромисса между точностью и эффективностью; иногда на них приходится идти. Здесь нам особая точность не нужна – показанное количество все равно продолжает изменяться, но этого вполне хватает, чтобы убедить пользователя в том, что обработка продолжается. Поэтому не будем тратить ни память, ни время процессора на вычисление абсолютно точного результата.

```
void MainWindow::finished()
{
    foreach (QPointer<GetMD5sThread> thread, threads)
        if (thread && thread->isRunning())
            return;
    processResults();
}
```

По завершении работы объект `GetMD5sThread` генерирует сигнал `finished()` (это поведение он наследует от базового класса `QThread`). Поскольку потоков может быть несколько, то мы проверяем, есть ли еще работающие. Если есть, то мы ничего не делаем и просто возвращаем управление. А если работающих вспомогательных потоков не осталось, значит, этот слот был вызван последним потоком и мы можем вызывать метод `processResults()`.

```
void MainWindow::processResults()
{
    stopThreads();

    qint64 maxSize;
    forever {
        bool more;
        QStringList files = filesForMD5.takeOne(&more);
        if (!more)
```



```

        break;
    if (files.count() < 2)
        continue;
    addOneResult(files, &maximumSize);
}
updateView(maximumSize);
statusBar()->showMessage(tr("Found %Ln duplicate file(s)", "",
                           model->rowCount()));
completed();
}

```

Этот метод представляет результаты пользователю. Первым делом он вызывает метод `stopThreads()`, удаляющий все вспомогательные потоки, которые больше не нужны.

Мы сохраняем размер самого большого файла, потому что он понадобится для вычисления необходимой ширины столбца «Size». Упрощенный класс `ThreadSafeHash` не предоставляет ни итератора, ни какого-либо другого способа доступа к элементам, иначе как по ключам (а способа получить все ключи тоже нет). Все, чем мы располагаем, — это деструктивный метод `takeOne()`, но его вполне достаточно. Мы входим в бесконечный цикл (макрос `forever` — то же самое, что `while(1)`) и пытаемся получить список имен файлов. (Метод `ThreadSafeHash::takeOne()` возвращает объект типа `QList<QString>`, но существует объявленный без ключевого слова `explicit` конструктор `QStringList(QList<QString>&)`, который выполняет нужное преобразование.) Если флаг `more` равен `false`, то хеш пуст и мы выходим из цикла. В противном случае при условии, что в списке имен файлов есть по меньшей мере два элемента (то есть хотя бы один дубликат), мы вызываем метод `addOneResult()`, который помещает в модель эту информацию.

В заключение мы обновляем представление — сортируем его и задаем значения ширины столбцов, сообщаем пользователю, сколько всего обнаружено дубликатов, и вызываем метод `completed()`, который подготавливает пользовательский интерфейс к очередному поиску.

```

void MainWindow::addOneResult(const QStringList &files,
                             quint64 *maximumSize)
{
    QFileInfo info(files.first());
    if (info.size() > *maximumSize)
        *maximumSize = info.size();
    QStandardItem *parentItem = model->invisibleRootItem();
    QStandardItem *treeItem = new QStandardItem(info.fileName());
    QStandardItem *sizeItem = new QStandardItem(
        QString("%L1").arg(info.size(), 20, 10, QChar(' ')));
    sizeItem->setTextAlignment(Qt::AlignVCenter|Qt::AlignRight);
    parentItem->appendRow(QList<QStandardItem*>() << treeItem
                        << sizeItem);

    foreach (const QString &filename, files)
        treeItem->appendRow(new QStandardItem(

```

```

        QDir::toNativeSeparators(filename)));
    }

```

Этот метод создает новую строку верхнего уровня, содержащую имя первого файла в списке. В первом элементе строки (то есть в первом столбце) находится имя файла без пути, а во втором элементе (втором столбце) – размер файла (в виде строки). Затем метод добавляет для каждого файла по одной строке, в которой находится только имя файла (с путем). Таким образом, у каждой строки верхнего уровня оказывается по меньшей мере две дочерних строки. Отметим также, что имена файлов в дочерних строках могут не совпадать с именем файла в строке верхнего уровня, поскольку приложение Find Duplicates находит дубликаты, не обращая внимания на имена файлов.

Размер файла выводится в виде локализованной строки, но дополняется слева пробелами, чтобы можно было сортировать дерево по размерам файлов (щелкнув по заголовку столбца «Size»); в этом случае подразумеваемая по умолчанию алфавитная сортировка будет работать корректно. И разумеется, мы используем статический метод `QDir::toNativeSeparators()`, чтобы выводить разделители компонентов пути в соответствии с платформенными соглашениями.

```

void MainWindow::updateView(qint64 maximumSize)
{
    if (model->rowCount()) {
        model->sort(0, Qt::AscendingOrder);
        view->expand(model->invisibleRootItem()->child(0)->index());
        QFontMetrics fm(font());
        int sizeWidth = fm.width(QString("W%L1W").arg(maximumSize));
        view->setColumnWidth(1, sizeWidth);
        sizeWidth += fm.width("W");
        view->setColumnWidth(0, view->width() - (sizeWidth +
            view->verticalScrollBar()->sizeHint().width()));
    }
}

```

Этот метод вызывается после заполнения модели именами файлов-дубликатов. Он сортирует модель по имени файла в строке верхнего уровня и раскрывает первый элемент, чтобы показать находящиеся в нем дубликаты. Затем он вычисляет ширину столбца размера, ориентируясь на локализованную строку, содержащую максимальный размер и две буквы *W* для формирования отступов. Столбцу с именами файлов отводится все оставшееся место за вычетом ширины вертикальной полосы прокрутки и небольшого дополнительного поля.

```

void MainWindow::completed()
{
    view->setSortingEnabled(true);
    cancelButton->setEnabled(false);
    cancelButton->hide();
    findButton->show();
}

```

```

        findButton->setEnabled(true);
        findButton->setFocus();
        rootDirectoryEdit->setEnabled(true);
    }

```

После того как поиск завершен (или прерван), вызывается этот метод, который подготавливает пользовательский интерфейс к следующему поиску. Он также заново активирует возможность сортировки (поскольку все изменения модели уже произведены), чтобы пользователь мог сортировать дерево щелчком по заголовку столбца.

```

void MainWindow::cancel()
{
    stopThreads();
    completed();
    statusBar()->showMessage(tr("Canceled"), StatusTimeout);
}

```

Этот слот вызывается, когда пользователь прерывает обработку. Он останавливает все потоки, подготавливает пользовательский интерфейс к следующему поиску и подтверждает прерывание. Но он *не* очищает модель, так что все найденные к этому моменту дубликаты показываются. Конечно, это поведение легко изменить.

```

void MainWindow::closeEvent(QCloseEvent *event)
{
    stopThreads();
    event->accept();
}

```

Когда пользователь хочет выйти из приложения, мы прежде всего останавливаем все работающие потоки, а затем разрешаем продолжить процедуру завершения.

Итак, мы в достаточной мере ознакомились с инфраструктурой приложения, чтобы понять, как создаются потоки, соединяются их сигналы и производится запуск. Теперь обратимся к классу GetMD5sThread — на удивление простому. Сначала покажем часть его объявления, а затем рассмотрим метод run().

```

class GetMD5sThread : public QThread
{
    Q_OBJECT

public:
    explicit GetMD5sThread(volatile bool *stopped,
                           const QString &root, const QStringList &directories,
                           FilesForMD5 *filesForMD5)
        : m_stopped(stopped), m_root(root),
          m_directories(directories), m_filesForMD5(filesForMD5) {}

signals:
    void readOneFile();

```

```
private:
    void run();
    ...
};
```

Конструктор принимает аргументы, определяющие работу, которую предстоит выполнить потоку, и сохраняет их в переменных-членах. Мы объявляем только один дополнительный сигнал, потому что сигнал `finished()` унаследован от класса `QThread`. Сделав метод `run()` закрытым, мы предотвращаем наследование классу, а также запрещаем вызывать этот метод для экземпляров (поскольку его разрешено вызывать только из `QThread::start()`).

```
void GetMD5sThread::run()
{
    foreach (const QString &directory, m_directories) {
        QDirIterator::IteratorFlag flag = directory == m_root
            ? QDirIterator::NoIteratorFlags
            : QDirIterator::Subdirectories;
        QDirIterator i(directory, flag);
        while (i.hasNext()) {
            const QString &filename = i.next();
            const QFileInfo &info = i.fileInfo();
            if (!info.isFile() || info.isSymLink() ||
                info.size() == 0)
                continue;
            if (*m_stopped)
                return;
            QFile file(filename);
            if (!file.open(QIODevice::ReadOnly))
                continue;
            QByteArray md5 = QCryptographicHash::hash(
                file.readAll(),
                QCryptographicHash::Md5);
            if (*m_stopped)
                return;
            m_filesForMD5->insert(qMakePair(md5, info.size()), filename);
            emit readOneFile();
        }
    }
}
```

Этот метод – самое сердце приложения, именно в нем происходит вся работа. Поток обходит список переданных ему каталогов и для каждого создает объект `QDirIterator`, с помощью которого перебирает подлежащие обработке файлы. В начальный список был помещен каталог, указанный пользователем (корневой), и его подкаталоги первого уровня. Следовательно, первым каталогом в первом потоке будет корневой, а его подкаталоги уже находятся в списке, поэтому, если обрабатываемый каталог совпадает с корневым, то рекурсивно спускаться в него не нужно – мы учли это при создании флага `QDirIterator`.

Мы игнорируем все объекты, не являющиеся файлами, а также файлы нулевой длины (хотя логически все они являются дубликатами друг друга). Файлы, которые мы собираемся обрабатывать, открываются в двоичном режиме, и если метод `QFile::open()` завершается успешно, то мы читаем все содержимое файла методом `QFile::readAll()` и передаем его функции `QCryptographicHash::hash()`, чтобы она вычислила MD5-сумму. Затем мы обновляем хеш, для чего создаем (или находим) запись с ключом, который состоит из MD5-суммы и размера текущего файла, и добавляем имя файла в список значений для этого ключа. Глобальная функция `qMakePair()` используется для создания объектов `QPair` – в данном случае содержащих пару (`QByteArray`, `qint64`). И конечно, беспокоиться о синхронизации доступа не нужно, так как за нас это делает класс `ThreadSafeHash`.

В заключение мы генерируем сигнал, извещающий о том, что файл прочитан. Поскольку `QThread` – подкласс `QObject`, то можно спокойно пользоваться механизмом сигналов и слотов, не прибегая к созданию пользовательских событий или вызову слотов напрямую. (Для обработки соединения сигналов со слотами в другом потоке Qt «за кулисами» применяет механизм событий, но все это делается прозрачно, и нам об этом думать необязательно.)

Отметим, что в двух местах мы проверяем, не прервал ли пользователь обработку, и если это так, немедленно завершаем ее выполнение.

Класс `QCryptographicHash` появился в версии Qt 4.3. Он может вычислять криптографические свертки с помощью алгоритмов MD5, MD4 и SHA1. Используемый нами статический метод `QCryptographicHash::hash()` принимает массив `QByteArray` (возвращенный методом `QFile::readAll()`) и название алгоритма, а возвращает криптографический хеш в виде массива `QByteArray`. Можно создать объект класса `QCryptographicHash`, наполнить его данными с помощью одного из методов `addData()` (они принимают указатель `char*` и длину массива или объект `QByteArray`) и в заключение вызвать метод `QCryptographicHash::result()` для получения хеша.

На этом мы завершаем рассмотрение приложения Find Duplicates. На его примере мы продемонстрировали, как использовать класс `QThread` в сочетании с потокобезопасной структурой данных. Чуть более полезной оказалась бы программа, показывающая третий столбец – количество дубликатов для каждого файла, – чтобы, отсортировав по этому столбцу, пользователь мог узнать, у какого файла больше всего дубликатов. Еще одно возможное усовершенствование – возможность удалить файл или – для UNIX-подобных операционных систем – сделать ссылку на файл (то есть удалить дубликат и создать вместо него символическую ссылку на оригинал). Все это оставляем читателю в качестве упражнения.

Вот и подошло к концу обсуждение высокоуровневых классов многопоточной обработки в Qt (в предыдущей главе) и класса `QThread`. В Qt имеется первоклассная поддержка многопоточности, но никуда не уйти от

того факта, что написание – и особенно сопровождение – многопоточных программ требует гораздо больше умения и внимания, чем однопоточных. Поэтому применять многопоточную обработку следует лишь в тех случаях, когда это действительно необходимо, и с большой осторожностью. Уменьшить риски и получить максимальный выигрыш от многопоточности можно, постаравшись вообще избежать необходимости синхронизации (обрабатывая элементы независимо) или, если без нее не обойтись, инкапсулировав ее в класс, так, чтобы клиенты этого класса не занимались синхронизацией самостоятельно. Применение функций из модуля *QtConcurrent*, а также класса *QRunnable* существенно упрощает обработку независимых элементов, не запрещая при этом использовать синхронизацию там, где это необходимо (см., например, класс *ThreadSafeErrorInfo*, созданный во втором разделе предыдущей главы). Но даже сложность работы с подклассами *QThread* можно свести к минимуму, если пользоваться потокобезопасными структурами данных, хотя их написание может оказаться непростой задачей, особенно если требуется добиться эффективности и развитой функциональности.

9

Создание редакторов форматированного текста

- Введение в класс `QTextDocument`
- Создание специализированных текстовых редакторов
- Однострочный редактор форматированного текста
- Многострочный редактор форматированного текста

В библиотеке Qt имеется развитый механизм для форматирования и отображения текстов, списков, таблиц и изображений¹. В основе этого механизма лежит класс `QTextDocument`, который может содержать как всего одну строку текста, так и целый многостраничный документ и поддерживает все стандартные виды форматирования (например, выделение полужирным шрифтом, курсивом или цветом, надстрочные и подстрочные индексы) вплоть до уровня отдельных символов.

Одно из важных достоинств класса `QTextDocument` заключается в том, что ему можно передать HTML-код, это упрощает включение форматированного текста в приложения. И подобно веб-браузеру, `QTextDocument` умеет применять каскадные таблицы стилей (CSS), обеспечивающие

¹ Форматированный текст (rich text) в Qt – это формат данных в памяти; не следует путать его ни с форматом RTF, разработанным корпорацией Microsoft для обмена документами, ни с форматом *enriched text* (RFC 1896), ни с MIME-типом `text/richtext` (RFC 1341 и 1521). Все они не имеют между собой ничего общего.

полное форматирование всего текста с помощью стилей. Еще одна особенность `QTextDocument`, которая роднит его с браузерами, – игнорирование разметки, которую он не понимает. HTML-теги и свойства CSS для форматирования текста, поддерживаемые Qt, перечислены на странице qt.nokia.com/doc/richtext-html-subset.html.

В этой главе мы будем заниматься созданием редакторов форматированного текста, в том числе с возможностью автозавершения и подсветки синтаксиса. А в следующей поговорим о выводе форматированного текста – как в файл (например, в формате ODT (Open Document Text) или PDF (Portable Document Format)), так и на печать.

В первом разделе мы дадим обзор класса `QTextDocument`, реализующего хранение форматированных текстовых документов в памяти. Изложенный материал важен для понимания как этой, так и следующей главы. Во втором разделе мы покажем, как добавить в строчный редактор функцию автозавершения и как создать специализированный многострочный XML-редактор с автозавершением и подсветкой синтаксиса. В третьем разделе мы рассмотрим создание однострочного редактора форматированного текста, а в четвертом – многострочного редактора, который предоставляет больше возможностей, например выравнивание текста и показ атрибутов форматирования символа и абзаца, находящихся в текущей позиции курсора.

Введение в класс `QTextDocument`

В этом разделе мы кратко опишем структуру класса `QTextDocument`. В настоящей главе этот класс будет использоваться для хранения форматированного текста в специализированных редакторах, а для инициализации, программного редактирования и наполнения объектов `QTextDocument` мы будем применять класс `QTextCursor`.

Поначалу может показаться, что хранение форматированного текста не вызывает сложностей, однако даже беглое знакомство с 700-страничной спецификацией формата Open Document Format (или со спецификацией формата OOXML корпорации Microsoft на 6000 страницах) наводит на мысль, что не все так просто. К счастью, Qt поддерживает лишь тщательно отобранное подмножество функций, изучить которое не так уж трудно, и для большинства повседневных нужд его хватает.

На внутреннем уровне `QTextDocument` хранит документ в виде рекурсивной структуры, состоящей из корневого фрейма, который содержит один (возможно, пустой) текстовый блок и последовательность из нуля или более дополнительных фреймов, текстовых блоков либо таблиц. Каждый фрейм внутри корневого также содержит один (возможно, пустой) текстовый блок, за которым опять-таки идет последовательность из нуля или более дополнительных фреймов, текстовых блоков или таблиц. Эта структура рекурсивно повторяется в каждом фрейме. Qt всег-

да помещает текстовый блок (даже если он пуст) в качестве разделителя между соседними фреймами или таблицами.

На рис. 9.1 схематически представлен пример первой страницы документа в том виде, как она хранится в `QTextDocument`. Вторая страница также находится внутри корневого фрейма и располагается вслед за последним показанным на рисунке текстовым блоком. Обе страницы включают текстовый блок (с заголовком), текстовую таблицу (состоящую из ячеек, в каждой из которых хранится текст) и еще один текстовый блок (содержащий заключительный абзац и для всех страниц, кроме последней, еще и индикатор разрыва страницы).

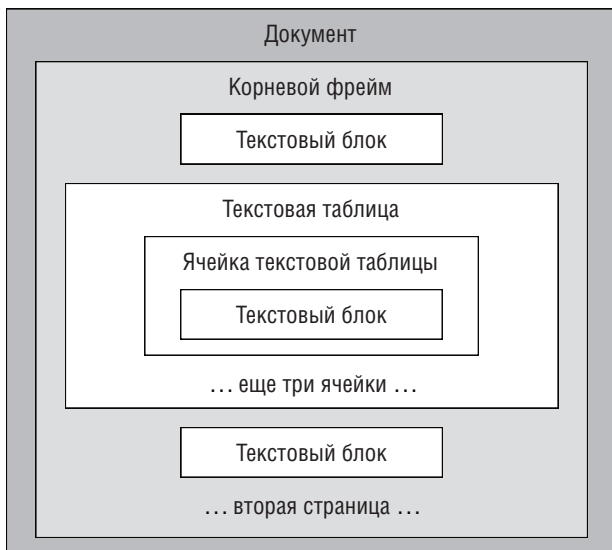


Рис. 9.1. Схематическое представление первой страницы документа

Класс `QTextBlock` может представлять абзац или список, причем поддерживаются вложенные списки произвольной глубины. Если текстовый блок представляет список, то метод `QTextBlock::textList()` возвращает указатель на объект `QTextList`, в противном случае (если это просто абзац текста) – нуль. Объект `QTextList` содержит один или несколько объектов `QTextBlock`, атрибуты которых (признак нумерованного или маркированного списка и отступ) хранятся в одном объекте `QTextListFormat`.

Атрибуты форматирования абзаца `QTextBlock` – выравнивание, поля, отступы и т. д. – хранятся в объекте `QTextBlockFormat`. Текстовый блок состоит из одного или нескольких фрагментов `QTextFragment`, в каждом из которых хранится текст с одинаковыми атрибутами форматирования (шрифт, подчеркивание и т. д.), которые, в свою очередь, содержатся в объекте `QTextCharFormat`. Фрагмент может содержать как всего один символ (например, подчеркнутую букву), так и весь текст абзаца. Не-

пустой текстовый блок, все символы которого отформатированы одинаково, обычно состоит из единственного фрагмента.

Класс `QTextFragment` не следует путать с классом `QTextDocumentFragment`. Объект `QTextBlock` включает один или несколько объектов `QTextFragment`, тогда как объект `QTextDocumentFragment` применяется для хранения произвольного фрагмента текстового документа `QTextDocument` и может содержать абзацы, таблицы и даже объект `QTextDocument` целиком. Выделенный пользователем фрагмент текста можно получить посредством метода `QTextCursor::selection()`, который возвращает объект `QTextDocumentFragment`.

Изображения представляются с помощью символа подстановки (`U+FFFC` в кодировке Unicode) внутри `QTextFragment`. С этим символом ассоциирован объект класса `QTextImageFormat` (подкласса `QTextCharFormat`), в котором хранятся размер и название изображения. Под названием понимается имя изображения в ресурсах приложения.

Таблицы представлены объектом `QTextTable`, атрибуты которого (выравнивание, отступ внутри ячейки, промежуток между ячейками, количество столбцов и т. д.) хранятся в объекте `QTextTableFormat`. Ячейки таблиц представлены классом `QTextTableCell` и могут содержать текстовые блоки или фреймы; таким образом, можно организовать сложную иерархическую структуру. Каждая ячейка имеет свою позицию в таблице (номер строки и столбца), с ней также можно ассоциировать атрибуты «количество занимаемых столбцов» и «количество занимаемых строк» и объект `QTextCharFormat`.

Таблицы и фреймы обрабатываются в документе единообразно, поскольку `QTextTable` – подкласс класса `QTextFrame`, представляющего фреймы. Атрибуты фрейма (рамка, поля, отступы, ширина и т. д.) хранятся в объекте `QTextFrameFormat`.

В Qt есть много классов, относящихся к форматированному тексту, некоторые из них касаются не столько хранения элементов документа, сколько верстки текста. Один из наиболее важных классов, связанных с `QTextDocument`, – `QTextCursor`, который предоставляет средства редактирования объектов `QTextDocument` из программы. В этой главе мы будем использовать `QTextCursor` для редактирования существующих документов, а в следующей – для создания документов с нуля.

Создание специализированных текстовых редакторов

В этом разделе мы покажем, как создать однострочный и простой многострочный редакторы, удобные для пользователей.

В качестве первого улучшения мы добавим функцию *автозавершения*. Когда пользователь вводит текст (в некоторых реализациях – нажимает

определенную последовательность клавиш), программа предлагает ему возможные варианты завершения – обычно в виде списка. Пользователь может подвести курсор к нужному варианту с помощью клавиш со стрелками и нажать Enter или просто щелкнуть мышью по нужному варианту. А может игнорировать список, просто продолжая вводить текст с клавиатуры, нажав клавишу Esc или щелкнув мышью вне списка¹.

Но этим мы не ограничимся. Многострочный редактор мы обогатим функциями выделения текущей строки и подсветки синтаксиса. Когда текущая строка выделена, пользователю легче ориентироваться в документе, а подсветка синтаксиса позволяет выявить структуру текста (в нашем случае подсвечиваться будет синтаксис XML) и помогает находить синтаксические ошибки.

Автозавершение в однострочных редакторах и комбинированных списках

В программах Image2Image (глава 7, стр. 274) и Find Duplicates (глава 8, стр. 329) мы применяли класс `QCompleter` в сочетании с `QLineEdit`, чтобы упростить ввод путей к файлам. Мы использовали раскрывающийся список вариантов завершения, хотя класс `QCompleter` умеет также выполнять внутристрочное автозавершение; оба варианта показаны на рис. 9.2.

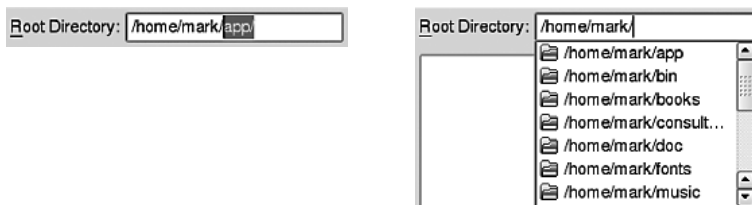


Рис. 9.2. Два вида автозавершения – внутристрочное и с помощью раскрывающегося списка

Объекту `QCompleter` передается модель `QAbstractItemModel` (списковая или древовидная), элементы которой предлагаются в качестве кандидатов. Модель, переданная `QCompleter`, может быть отсортирована (с учетом или без учета регистра) или не отсортирована.

¹ К автозавершению очень близка идея *методов ввода* – средств ввода текста. Широко известный пример – использование буквенно-цифровых кнопок на сотовом телефоне для ввода текстовых сообщений. А если говорить о настольных компьютерах, то, например, японец вводит последовательности символов латиницы, которые воспринимаются программой как японские символы; подробности см. в документации по классу `QInputContext`.

Вот как начинается метод `MainWindow::createWidgets()` в программе **Find Duplicates**, где мы создаем строчный редактор и функцию автозавершения для него.

```
void MainWindow::createWidgets()
{
    rootDirectoryLabel = new QLabel(tr("Root Directory:"));
    rootDirectoryEdit = new QLineEdit(QDir::toNativeSeparators(
        QDir::homePath()));
    rootDirectoryLabel->setBuddy(rootDirectoryEdit);
    QCompleter *directoryCompleter = new QCompleter(this);
    directoryCompleter->setCaseSensitivity(Qt::CaseInsensitive);
    directoryCompleter->setModel(new DirModel(directoryCompleter));
    rootDirectoryEdit->setCompleter(directoryCompleter);
}
```

Это все, что необходимо сделать для подключения данной функции: создать объект для механизма автозавершения, назначить ему модель и ассоциировать его с виджетом-редактором. После этого Qt автоматически будет управлять взаимодействием с клавиатурой и раскрываемым списком.

В данном случае мы не стали использовать класс `QDirModel` как таковой, а создали его подкласс¹. Нужен он только для того, чтобы на каждой платформе показывались соответствующие ей разделители компонентов пути. Для полноты приведем код этой пользовательской модели (он почти идентичен коду из примера *examples/tools/completer*):

```
class DirModel : public QDirModel
{
public:
    explicit DirModel(QObject *parent=0) : QDirModel(parent) {}

    QVariant data(const QModelIndex &index, int role=Qt::DisplayRole) const
    {
        if (role == Qt::DisplayRole && index.column() == 0) {
            QString path = QDir::toNativeSeparators(filePath(index));
            if (path.endsWith(QDir::separator()))
                path.chop(1);
            return path;
        }
        return QDirModel::data(index, role);
    }
};
```

В этом подклассе просто переопределен метод `QAbstractItemModel::data()` в той части, которая относится к роли `Qt::DisplayRole`, — он расставляет правильные разделители и при необходимости убирает завершающий разделитель.

¹ В версии Qt 4.7 класс `QDirModel` будет заменен новым классом `QFileSystemModel`, работающим асинхронно.

Понятно, что очень легко сконфигурировать механизм автозавершения для объекта `QLineEdit` – или `QComboBox`, у которого также имеется аналогичный метод `setCompleter()`. Причем, если имеется статический список строк, то задача еще больше упрощается, потому что список можно передать конструктору `QCompleter` и не связываться с моделью вообще.

Создание функции автозавершения для многострочного редактора сложнее, потому что нам необходимо предоставить какой-то механизм ее вызова (например, специальную комбинацию клавиш), заполнить список вариантов автозавершения и правильно его позиционировать. Как все это делается, мы увидим в следующем подразделе.

Автозавершение и подсветка синтаксиса в текстовых редакторах

В этом подразделе мы разработаем простой виджет текстового редактора для XML. Помимо функциональности, получаемой от классов `QPlainTextEdit` и `QTextEdit` (копирование и вставка, отмена и повтор операций, изменение масштаба и т. п.), мы добавим еще две функции: автозавершение и подсветку синтаксиса.

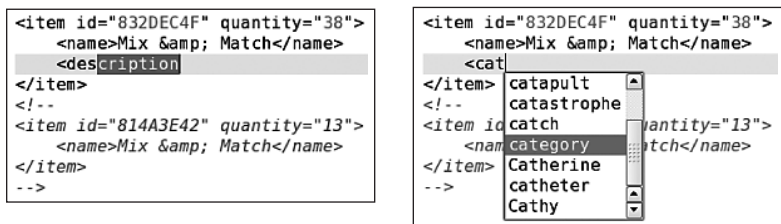


Рис. 9.3. Виджет `XmlEdit` – автозавершение при наличии одного и нескольких вариантов

На рис. 9.3 изображен виджет `XmlEdit` (`xmledit`), который мы рассмотрим в этом подразделе. Слева показано, что происходит, когда пользователь вводит строку `des` и нажимает `Ctrl+M` (комбинация клавиш для вызова функции автозавершения). В этом случае в модели имеется только одно слово, начинающееся с «des», поэтому редактор сразу же вставляет недостающую часть слова и выделяет вставленный текст. Пользователь может либо принять предлагаемую вставку, нажав клавишу `Enter` (тогда вставленный текст перестает быть выделенным и курсор перемещается в конец слова), либо отвергнуть ее, нажав `Esc` (или `Delete`, или просто продолжив вводить символы). Справа показано, что происходит, когда пользователь вводит строку «cat» и нажимает комбинацию клавиш `Ctrl+M`. Поскольку имеется несколько слов, начинающихся с «cat», редактор открывает список слов на выбор. Чтобы выбрать слово из списка, пользователь может щелкнуть по нему мышью или подвести к нему курсор с помощью клавиш со стрелками и нажать клавишу `Enter`. Чтобы отказаться от предложения, достаточно нажать клавишу

Esc, щелкнуть мышью вне списка или просто продолжить ввод символов с клавиатуры¹.

Во фрагменте XML-документа на рисунке показано также, как выглядит подсветка синтаксиса. Мы решили раскрасить теги темно-синим цветом, имена атрибутов – голубым, значения атрибутов – темно-желтым, XML-компоненты – темно-красным, а комментарии – зеленым и при этом выделить курсивом. Отметим еще, что на обоих рисунках текущая строка выделена цветом фона. (На снимках экрана выделение более темное, чем в исходном коде; чтобы увидеть, как цвета выглядят на самом деле, соберите и запустите программу.)

Захоти мы пойти немного дальше, было бы нетрудно добавить еще нумерацию строк, но оставим это в качестве упражнения для читателя – посмотреть, как это делается, можно в примере *examples/widgets/code-editor*. Однако если вам нужен полнофункциональный текстовый редактор, написанный на Qt и поддерживающий подсветку синтаксиса для многих языков, а также целый ряд других возможностей, то лучше обратиться к проекту *QScintilla* (www.riverbankcomputing.co.uk/software/qscintilla), представляющему собой перенос на Qt написанного на C++ компонента Scintilla (www.scintilla.org).

Автозавершение для многострочных редакторов

Для виджета `XmlEdit` мы реализуем два вида автозавершения. Если при вызове функции автозавершения имеется всего одно слово-кандидат, то мы сразу же вставляем оставшуюся часть слова и выделяем вставленный текст. Если в этот момент пользователь нажмет клавишу Esc, Delete или любой символ, кроме Enter (эта клавиша подтверждает вставку), то выбранный текст удаляется, и если нажатая клавиша соответствует символу, имеющему графическое представление, то этот символ будет вставлен в текст, как обычно. Если же слов-кандидатов несколько, то появится их список и пользователь сможет выбрать из него слово либо отказаться от предложения.

Создавая свой класс `XmlEdit`, мы решили унаследовать классу `QPlainTextEdit`, а не `QTextEdit`. Имя класса `QPlainTextEdit` может сбить с толку – следовало бы назвать его *QBasicTextEdit*, поскольку он в полной мере поддерживает форматирование на уровне символов, в частности выделение полужирным шрифтом, курсивом и цветом, и даже предоставляет метод `QPlainTextEdit::appendHtml(!)`. Класс `QPlainTextEdit` очень похож на `QTextEdit`

¹ На большинстве платформ не определены стандартные клавиши для автозавершения, поэтому в разных приложениях для Linux и Windows могут применяться различные соглашения. Но в Mac OS X для вызова автозавершения предназначена клавиша Esc, поэтому, если мы хотим придерживаться стандарта, то должны включить директиву `#ifdef Q_WS_MAC` и при компиляции программы для Mac OS X указать, что автозавершение инициируется клавишей Esc.

и предоставляет в основном ту же функциональность, включая возможность подключить объект `QSyntaxHighlighter`, в чем мы убедимся в следующем подразделе. Основное отличие заключается в том, что `QPlainTextEdit` не поддерживает ни фреймов, ни таблиц и применяет гораздо более простой алгоритм верстки текста. Поэтому `QPlainTextEdit` значительно быстрее `QTextEdit` обрабатывает большие документы, что делает его идеальным выбором для просмотра разного рода журналов и в качестве основы для специализированных редакторов простого текста.

Для поддержки автозавершения в классе `XmlEdit` есть три закрытых переменных-члена: `completedAndSelected` типа `bool`, `completer` типа `QCompleter*` и `model` типа `QStringListModel*`. Помимо конструктора, методов для создания дочерних виджетов и установки соединений, мы должны переопределить два обработчика событий и написать три слота и шесть (небольших) вспомогательных методов. Начнем с конструктора.

```
XmlEdit::XmlEdit(QWidget *parent)
    : QPlainTextEdit(parent), completedAndSelected(false)
{
    createWidgets();
    createConnections();
    highlightCurrentLine();
}
```

В конструкторе нет ничего необычного. Метод `highlightCurrentLine()` мы рассмотрим в следующем подразделе, когда речь пойдет о выделении.

```
void XmlEdit::createWidgets()
{
    (void) new XmlHighlighter(document());
    model = new QStringListModel(this);
    completer = new QCompleter(this);
    completer->setWidget(this);
    completer->setCompletionMode(QCompleter::PopupCompletion);
    completer->setModel(model);
    completer->setModelSorting(
        QCompleter::CaseInsensitivelySortedModel);
    completer->setCaseSensitivity(Qt::CaseInsensitive);
    completer->setWrapAround(true);
}
```

В первой строке создается пользовательский объект для подсветки синтаксиса и ассоциируется с объектом `QTextDocument` (из базового класса), в котором хранятся текст и данные о форматировании `XmlEdit`.

Оставшаяся часть метода относится к автозавершению. Сначала мы создаем модель для механизма автозавершения, а потом сам механизм. При его создании мы задали режим автозавершения `QCompleter::PopupCompletion` (раскрывающийся список), но могли бы с тем же успехом использовать `QCompleter::InlineCompletion` или `QCompleter::UnfilteredPopupCompletion`.

С помощью метода `QCompleter::setModelSorting()` механизм автозавершения получает информацию о том, как отсортирована модель, – никакая сортировка при этом *не* производится! В данном случае мы сообщили, что модель отсортирована без учета регистра, альтернативные варианты: `QCompleter::CaseSensitivelySortedModel` и `QCompleter::UnsortedModel`. Если данные модели, хранящиеся в соответствующей роли в используемом для автозавершения столбце, отсортированы, то об этом следует уведомить механизм автозавершения (что мы и сделали). Если механизм автозавершения узнает о том, что модель отсортирована, то сможет при поиске подходящих вариантов воспользоваться быстрым двоячным поиском вместо медленного линейного – для больших моделей это даст существенный выигрыш в скорости.

С помощью метода `QCompleter::setCaseSensitivity()` механизм автозавершения получает информацию о том, должен ли учитываться при поиске регистр; если нет, то он будет показывать и вставлять варианты без учета регистра, в противном случае – только те, что соответствуют регистру завершаемого текста. Последний вызов, `QCompleter::setWrapAround()`, определяет, что происходит при достижении конца или начала списка: если аргумент равен `true`, то при попытке пользователя подняться «выше» верхней (первой) строки текущей становится нижняя (последняя) строка.

```
void XmlEdit::createConnections()
{
    connect(this, SIGNAL(cursorPositionChanged()),
           this, SLOT(highlightCurrentLine()));
    connect(completer, SIGNAL(activated(const QString&)),
           this, SLOT(insertCompletion(const QString&)));
    (void) new QShortcut(QKeySequence(tr("Ctrl+M", "Complete")),
                       this, SLOT(performCompletion()));
}
```

Нам необходимы три соединения сигналов со слотами. Последние два нужны для автозавершения, а первое – чтобы при перемещении курсора (вследствие ввода текста или навигации – например, с помощью клавиш со стрелками или щелчка мышью) выделялась текущая строка. Этот вопрос мы обсудим в следующем подразделе.

Когда пользователь выбирает элемент из списка, механизм автозавершения генерирует сигнал `activated()`; мы подключаемся к этому сигналу, чтобы вставить в текст выбранный вариант.

Третье соединение устанавливается в процессе создания нового объекта `QShortcut` и используется для того, чтобы при нажатии комбинации клавиш `Ctrl+M` вызывался слот `performCompletion()`.

```
void XmlEdit::performCompletion()
{
    QTextCursor cursor = textCursor();
    cursor.select(QTextCursor::WordUnderCursor);
```

```

const QString completionPrefix = cursor.selectedText();
if (!completionPrefix.isEmpty() &&
    completionPrefix.at(completionPrefix.length() - 1).isLetter())
    performCompletion(completionPrefix);
}

```

При нажатии Ctrl+M мы запускаем процедуру автозавершения, только если имеется текст, который можно завершить. В начале этого слота мы получаем положение курсора в документе `QTextDocument`, а затем находим слово (это может быть единственный символ), на котором или за которым (без промежуточных пробелов) находится курсор. Если курсор не находится внутри слова или сразу после него, то мы получим пустую строку.

Если найденное слово (префикс автозавершения) не пусто и заканчивается буквой, то мы вызываем закрытый перегруженный метод `performCompletion()`, передавая ему префикс.

Обратите внимание, что мы не вызываем метод `QPlainTextEdit::setTextCursor()` с модифицированным курсором. Это означает, что произведенное изменение (то есть выделение слова) не попадает в документ, а в данном случае именно это нам и нужно.

Метод `QTextCursor::select()` можно применять для выделения не только отдельных слов, но и всей текущей строки (параметр `QTextCursor::LineUnderCursor`), текущего абзаца (`QTextCursor::BlockUnderCursor`) и даже всего документа (`QTextCursor::Document`). API класса `QTextCursor` показан ниже в табл. 9.1–9.3.

```

void XmlEdit::performCompletion(const QString &completionPrefix)
{
    populateModel(completionPrefix);
    if (completionPrefix != completer->completionPrefix()) {
        completer->setCompletionPrefix(completionPrefix);
        completer->popup()->setCurrentIndex(
            completer->completionModel()->index(0, 0));
    }
    if (completer->completionCount() == 1)
        insertCompletion(completer->currentCompletion(), true);
    else {
        QRect rect = cursorRect();
        rect.setWidth(completer->popup()->sizeHintForColumn(0) +
            completer->popup()->verticalScrollBar()->
            sizeHint().width());
        completer->complete(rect);
    }
}

```

Этот метод сначала заполняет модель, ассоциированную с механизмом автозавершения. Затем он передает механизму автозавершения новый префикс, если тот не совпадает с хранящимся там в данный момент, и выделяет первый элемент в списке вариантов завершения.

Таблица 9.1. API класса *QTextCursor* (часть 1)

Метод	Описание
<code>anchor()</code>	Возвращает позицию якоря; см. также <code>position()</code>
<code>atBlockEnd()</code>	Возвращает <code>true</code> , если курсор находится в конце блока
<code>atBlockStart()</code>	Возвращает <code>true</code> , если курсор находится в начале блока
<code>atEnd()</code>	Возвращает <code>true</code> , если курсор находится в конце документа
<code>atStart()</code>	Возвращает <code>true</code> , если курсор находится в начале документа
<code>beginEditBlock()</code>	Уведомляет курсор о том, что последующие операции редактирования следует рассматривать как одну операцию с точки зрения отмены/повтора; см. <code>endEditBlock()</code>
<code>block()</code>	Возвращает объект <code>QTextBlock</code> , содержащий курсор
<code>blockCharFormat()</code>	Возвращает объект <code>QTextCharFormat</code> для блока, содержащего курсор; см. также <code>charFormat()</code>
<code>blockFormat()</code>	Возвращает объект <code>QTextBlockFormat</code> для блока, содержащего курсор
<code>blockNumber()</code>	Возвращает номер строки позиции курсора в документах, не содержащих ни таблиц, ни фреймов (кроме корневого), например <code>QTextDocument</code> из объекта типа <code>QPlainTextEdit</code>
<code>charFormat()</code>	Возвращает объект <code>QTextCharFormat</code> для символа, непосредственно предшествующего курсору
<code>clearSelection()</code>	Перемещает якорь в позицию курсора, так, чтобы ничего не было выделено; см. также <code>removeSelectedText()</code>
<code>columnNumber()</code>	Возвращает позицию курсора в строке
<code>createList (QTextListFormat)</code>	Вставляет и возвращает объект <code>QTextList</code> , применяя заданный формат (или заданный стиль <code>QTextListFormat::Style</code>) и считая текущий абзац первым элементом списка; см. также <code>insertList()</code>
<code>currentFrame()</code>	Возвращает текущий фрейм в виде указателя <code>QTextFrame*</code>

Метод	Описание
<code>currentList()</code>	Возвращает текущий список в виде указателя <code>QTextList*</code> или <code>0</code> , если курсор не находится внутри списка
<code>currentTable()</code>	Возвращает текущую таблицу в виде указателя <code>QTextTable*</code> или <code>0</code> , если курсор не находится внутри таблицы
<code>deleteChar()</code>	Если имеется выделенный текст, удаляет его; в противном случае удаляет символ в позиции курсора
<code>deletePreviousChar()</code>	Если имеется выделенный текст, удаляет его; в противном случае удаляет символ, предшествующий курсору
<code>document()</code>	Возвращает документ, с которым ассоциирован курсор, в виде указателя <code>QTextDocument*</code>

Таблица 9.2. API класса *QTextCursor* (часть 2)

Метод	Описание
<code>endEditBlock()</code>	Уведомляет курсор о том, что последовательность операций редактирования, начатая методом <code>beginEditBlock()</code> , завершилась
<code>hasComplexSelection()</code>	Возвращает <code>true</code> , если выделен не просто непрерывный участок текста, а, например, две или более ячеек таблицы
<code>hasSelection()</code>	Возвращает <code>true</code> , если что-то выделено
<code>insertBlock(QTextBlockFormat, QTextCharFormat)</code>	Вставляет новый пустой блок в позиции курсора; есть также два перегруженных варианта: один принимает только формат блока, второй вообще не имеет аргументов
<code>insertFragment(QTextDocumentFragment)</code>	Вставляет переданный фрагмент документа в позиции курсора
<code>insertFrame(QTextFrameFormat)</code>	Вставляет фрейм <code>QTextFrame</code> с заданным форматом в позиции курсора и перемещает позицию внутрь фрейма
<code>insertHtml(QString)</code>	Вставляет строку HTML в позиции курсора
<code>insertImage(...)</code>	Вставляет изображение в позиции курсора. Есть также перегруженные варианты: с аргументом <code>QTextImageFormat</code> (и необязательным аргументом <code>QTextFrameFormat::Position</code>), или <code>QString (filename)</code> , или <code>Qimage</code>

Таблица 9.2. (продолжение)

Метод	Описание
<code>insertList(QTextListFormat)</code>	Вставляет новый блок в позиции курсора и делает его первым элементом списка типа <code>QTextList</code> с заданным форматом (или заданным стилем <code>QTextListFormat::Style</code>). Затем возвращает этот блок. См. также <code>createList()</code>
<code>InsertTable(int, int, QTextTableFormat)</code>	Вставляет и возвращает новый объект <code>QTextTable</code> с заданным количеством строк и столбцов и (необязательным) форматом; см. текст
<code>insertText(QString, QTextCharFormat)</code>	Вставляет текст в позиции курсора, применяя переданный (необязательный) формат
<code>isCopyOf(QTextCursor)</code>	Возвращает <code>true</code> , если данный курсор является копией переданного в качестве аргумента
<code>isNull()</code>	Возвращает <code>true</code> , если данный курсор пустой (то есть сконструирован без указания <code>QTextDocument</code>)
<code>joinPreviousEditBlock()</code>	Отменяет последний вызов <code>endEditBlock()</code> , тем самым продолжая действие предыдущего вызова <code>beginEditBlock()</code>
<code>mergeBlockCharFormat(QTextCharFormat)</code>	Объединяет переданный формат с форматом символов текущего блока (или выделения)
<code>mergeBlockFormat(QTextBlockFormat)</code>	Объединяет переданный формат с форматом текущего блока (или выделения)

Таблица 9.3. API класса `QTextCursor` (часть 3)

Метод	Описание
<code>mergeCharFormat(QTextCharFormat)</code>	Объединяет переданный формат с форматом символа (или выделения) в позиции курсора
<code>movePosition(MoveOperation, MoveMode, int)</code>	Перемещает позицию курсора, применяя данную операцию указанное количество раз (необязательный аргумент). Если задан (необязательный) режим <code>KeepAnchor</code> , то якорь остается на месте, в результате чего создается выделение; по умолчанию подразумевается режим <code>MoveAnchor</code> (операции перемещения перечислены в табл. 9.4 на стр. 362)
<code>position()</code>	Возвращает позицию курсора; см. <code>setPosition()</code>

Метод	Описание
<code>removeSelectedText()</code>	Удаляет все, что выделено
<code>select(SelectionType)</code>	Выделяет текст в соответствии с указанным типом (<code>Document</code> , <code>BlockUnderCursor</code> , <code>LineUnderCursor</code> , <code>WordUnderCursor</code>)
<code>selectedTableCells(int*, int*, int*, int*)</code>	Записывает в переданные по указателям целые числа номер первой строки, количество строк, номер первого столбца и количество столбцов, описывающие диапазон выделенных ячеек
<code>selectedText()</code>	Возвращает текст выделения в виде простого текста
<code>selection()</code>	Возвращает выделение в виде объекта <code>QTextDocumentFragment</code>
<code>selectionEnd()</code>	Возвращает позицию конца выделения
<code>selectionStart()</code>	Возвращает позицию начала выделения
<code>setBlockCharFormat(QTextCharFormat)</code>	Задает формат символов для текущего блока (или текущего выделения); см. также <code>mergeBlockCharFormat()</code>
<code>setBlockFormat(QTextBlockFormat)</code>	Задает формат текущего блока (или текущего выделения); см. также <code>mergeBlockFormat()</code>
<code>setCharFormat(QTextCharFormat)</code>	Задает формат текущего символа; см. также <code>mergeCharFormat()</code>
<code>setPosition(int, MoveMode)</code>	Перемещает курсор в указанную позицию; если задан режим <code>KeepAnchor</code> , якорь остается на месте и создается выделение; см. <code>movePosition()</code>
<code>setVisualNavigation(bool)</code>	Если аргумент равен <code>true</code> , то при перемещении скрытые абзацы пропускаются; по умолчанию подразумевается <code>false</code>
<code>visualNavigation()</code>	Возвращает режим визуальной навигации в виде <code>bool</code>

Если имеется только один вариант, то мы сразу же вставляем его, передавая `true` в качестве второго аргумента метода `insertCompletion()`. В противном случае вызываем `QCompleter::complete()`, чтобы открыть список вариантов; ширина и позиция списка передаются в аргументе типа `QRect`. Метод `QPlainTextEdit::cursorRect()` возвращает прямоугольник текстового курсора, но очевидно, что он слишком узок (всего несколько пикселей) для показа списка. Поэтому мы задаем ширину прямоугольника равной ширине первого (и обычно единственного) столбца плюс ширина вертикальной полосы прокрутки.

Благодаря установленному соединению сигнала со слотом (стр. 353) выбор пользователем варианта из списка приведет к вызову слота `insertCompletion()` (причем его второй аргумент будет иметь значение по умолчанию, `false`, означающее, что вариант выбран из списка слов, а не путем вставки единственного слова).

```
void XmlEdit::populateModel(const QString &completionPrefix)
{
    QStringList strings = toPlainText().split(QRegExp("\\W+"));
    strings.removeAll(completionPrefix);
    strings.removeDuplicates();
    qSort(strings.begin(), strings.end(), caseInsensitiveLessThan);
    model->setStringList(strings);
}
```

Этот метод вызывается при каждой активации функции автозавершения. Он динамически заполняет модель механизма автозавершения словами из текущего документа. Для этого весь текст документа разбивается на слова, из которых формируется список. Затем мы удаляем из списка префикс (если он там имеется) и слова-дубликаты, сортируем то, что останется, и заменяем хранящиеся в модели строки новыми.

Подобный алгоритм создания нового списка слов для автозавершения подходит для небольших документов, но если документ велик, то может потребовать много времени. Кроме того, для пустого документа вообще не будет вариантов завершения. Возможная альтернатива – прочитать список слов из словаря на этапе запуска приложения и заполнить модель только один раз. Другая возможность – использовать начальный словарь, как описано выше, и добавлять в него новые слова по мере ввода текста в документ. И в том и в другом случае модель инициализируется только один раз (правда, во втором случае обновляется).

Сортировка обязательна, потому что ранее при вызове метода `QCompleter::setModelSorting(QCompleter::CaseInsensitivelySortedModel)` мы сообщили механизму автозавершения, как именно отсортирована модель. Поэтому мы должны сдерживать слово и гарантировать, что модель отсортирована именно так, как было обещано.

Глобальную функцию `qSort()` можно вызывать для последовательно-сти, например `QList<T>`, `QStringList` или `QVector<T>`, либо для ее участка, заданного итераторами, указывающими на начало и конец, если мы хотим отсортировать лишь часть последовательности или (как в данном случае) дополнительно передать функцию либо функтор сравнения. В типе последовательности, сортируемой функцией `qSort()`, должен быть определен метод `operator<()`, но в классе `QString` этот метод по умолчанию выполняет сравнение с учетом регистра, поэтому мы написали собственную крохотную функцию сравнения.

В Qt имеется также функция `qStableSort()` с таким же API, как у `qSort()`. С точки зрения вычислительной сложности устойчивая сортировка не

отличается от стандартной, но сохраняет относительное взаиморасположение элементов, считающихся равными.

```
bool caseInsensitiveLessThan(const QString &a, const QString &b)
{
    return a.compare(b, Qt::CaseInsensitive) < 0;
}
```

Методы `QString::compare()` используются при локализации исключительно для перевода символов из одного регистра в другой в случае, когда задан аргумент `Qt::CaseInsensitive`.

Мы предпочли бы воспользоваться методом `QString::localeAwareCompare()`, который специально предназначен для сортировки списков строк, видимых пользователям. Однако, поскольку на момент написания этой книги в документации ничего не говорилось о том, какую сортировку понимает класс `QCompleter`, мы решили остановиться на более простом и быстром методе `QString::compare()`.

```
void XmlEdit::insertCompletion(const QString &completion, bool singleWord)
{
    QTextCursor cursor = textCursor();
    int numberOfCharsToComplete = completion.length() -
        completer->completionPrefix().length();
    int insertionPosition = cursor.position();
    cursor.insertText(completion.right(numberOfCharsToComplete));
    if (singleWord) {
        cursor.setPosition(insertionPosition);
        cursor.movePosition(QTextCursor::EndOfWord, QTextCursor::KeepAnchor);
        completedAndSelected = true;
    }
    setTextCursor(cursor);
}
```

Этот метод вставляет символы, дополняющие слово до выбранного варианта. Сначала мы получаем курсор документа. Затем вычисляем, сколько символов нужно вставить, – в списке автозавершения показаны целые слова, но нам требуется только закончить слово, которое пользователь уже начал вводить. Далее мы запоминаем текущую позицию курсора (ту, в которую нужно вставить символы) и вставляем часть выбранной строки, необходимую для завершения слова.

Если аргумент `singleWord` равен `true`, значит, подошло только одно слово, и мы должны выделить вставленные символы, чтобы пользователь их увидел и смог легко от них избавиться, если это не то, что нужно. Для этого мы устанавливаем курсор в то место, где он находился перед вставкой, а затем вызываем метод `QTextCursor::movePosition()`, который переместит курсор в конец слова. При этом вставленные символы выделяются (как мы скоро увидим). Кроме того, мы присваиваем переменной `completedAndSelected` значение `true`, поскольку хотим обрабатывать следующее нажатие клавиши (или щелчок мышью) специальным образом,

чтобы пользователь мог легко принять или отклонить предложенное продолжение слова.

В заключение мы устанавливаем новый курсор, чтобы изменения (вставка и, возможно, выделение) возымели эффект.

Метод `QTextCursor::movePosition()` имитирует навигацию пользователя по документу; в качестве аргумента он принимает перечисление, в котором определены все стандартные операции перемещения, – они перечислены в табл. 9.4.

Таблица 9.4. Перечисление `QTextCursor::MoveOperation`

Метод	Описание
Down	Перемещает курсор вниз на одну строку
End	Перемещает курсор в конец документа
EndOfBlock	Перемещает курсор в конец текущего блока
EndOfLine	Перемещает курсор в конец текущей строки
EndOfWord	Перемещает курсор в конец текущего слова
Left	Перемещает курсор влево на один символ
NextBlock	Перемещает курсор в начало следующего блока
NextCell	Перемещает курсор в начало следующей ячейки
NextCharacter	Перемещает курсор на следующий символ
NextRow	Перемещает курсор в первую ячейку следующей строки таблицы
NextWord	Перемещает курсор в начало следующего слова
NoMove	Не изменяет позицию курсора
PreviousBlock	Перемещает курсор в начало предыдущего блока
PreviousCell	Перемещает курсор в начало предыдущей ячейки
PreviousCharacter	Перемещает курсор на предыдущий символ
PreviousRow	Перемещает курсор в последнюю ячейку предыдущей строки таблицы
PreviousWord	Перемещает курсор в начало предыдущего слова
Right	Перемещает курсор вправо на один символ
Start	Перемещает курсор в начало документа
StartOfBlock	Перемещает курсор в начало текущего блока
StartOfLine	Перемещает курсор в начало текущей строки
StartOfWord	Перемещает курсор в начало текущего слова

Метод	Описание
Up	Перемещает курсор вверх на одну строку
WordLeft	Перемещает курсор влево на одно слово
WordRight	Перемещает курсор вправо на одно слово

В объекте `QTextCursor` хранятся две позиции: `QTextCursor::position()` (текущая позиция курсора) и `QTextCursor::anchor()` (некая другая позиция курсора). Обычно позиция и якорь совпадают, но если они различаются, то все находящееся между ними считается выделением. Таким образом, `QTextCursor` определяет выделение в терминах якоря и позиции – если они совпадают, то ничего не выделено.

Позицию можно установить напрямую с помощью метода `QTextCursor::setPosition()`, но якорь устанавливается только косвенно. Метод `QTextCursor::movePosition()` может работать в двух режимах (режим задается вторым аргументом): `QTextCursor::MoveAnchor` (по умолчанию) и `QTextCursor::KeepAnchor`. В режиме `KeepAnchor` якорь остается там, где был до перемещения, а позиция смещается.

Отметим слабое место нашей реализации (проявляющееся в некоторых случаях): она не изменяет регистр уже введенного текста в соответствии с выбранным вариантом завершения. Например, если пользователь ввел строку «ali» и в качестве продолжения выбрал «AlignLeft», то получится «alignLeft». Оставляем исправление этого недочета читателю в качестве упражнения.

```
void XmlEdit::keyPressEvent(QKeyEvent *event)
{
    if (completedAndSelected && handledCompletedAndSelected(event))
        return;
    completedAndSelected = false;
    if (completer->popup()->isVisible()) {
        switch (event->key()) {
            case Qt::Key_Up:      // Проваливаемся
            case Qt::Key_Down:    // Проваливаемся
            case Qt::Key_Enter:    // Проваливаемся
            case Qt::Key_Return:  // Проваливаемся
            case Qt::Key_Escape:  event->ignore(); return;
            default: completer->popup()->hide(); break;
        }
    }
    QPlainTextEdit::keyPressEvent(event);
}
```

Этот обработчик события вызывается при нажатии любой клавиши внутри виджета `XmlEdit`. Мы переопределили реакцию для двух случаев. Во-первых, если был найден всего один вариант завершения, то уже следующее нажатие клавиши (или щелчок мышью) определяет, при-

нято предложение или отклонено. А во-вторых, мы игнорируем все нажатия клавиш, обработанные раскрывающимся списком с вариантами автозавершения.

Если переменная `completedAndSelected` равна `true`, значит, единственный вариант завершения уже вставлен и сейчас обрабатывается нажатие следующей клавиши. В этом случае мы вызываем метод `handledCompletedAndSelected()`, который возвращает булево значение: если оно равно `true`, то нажатие уже им обработано, и тогда мы возвращаем управление, в противном случае продолжаем анализ.

Если раскрывающийся список виден, то мы игнорируем клавиши, которые обрабатывает механизм автозавершения (то есть клавиши со стрелками вверх и вниз для навигации, Esc для отмены и Enter для выбора). При нажатии любой другой клавиши мы скрываем список, ничего не выбрав из него, и передаем событие методу `keyPressEvent()` из базового класса, чтобы он обработал его обычным образом.

Отметим, что в любом случае переменной `completedAndSelected` присваивается значение `false` — она может быть равна `true` только *сразу* после вставки единственного найденного продолжения, а при нажатии следующей клавиши или щелчке мышью ее значение должно быть сброшено в `false`.

```
bool XmlEdit::handledCompletedAndSelected(QKeyEvent *event)
{
    completedAndSelected = false;
    QTextCursor cursor = textCursor();
    switch (event->key()) {
        case Qt::Key_Enter:           // Проваливаемся
        case Qt::Key_Return: cursor.clearSelection(); break;
        case Qt::Key_Escape: cursor.removeSelectedText(); break;
        default: return false;
    }
    setTextCursor(cursor);
    event->accept();
    return true;
}
```

Класс `QTextCursor` позволяет программно создавать и редактировать документы `QTextDocument`. Нужный курсор можно получить путем обращения к методу `QPlainTextEdit::textCursor()`, результат будет такой же, как при вызове `QTextCursor(document())`, поскольку метод `QPlainTextEdit::document()` возвращает указатель на объект `QTextDocument`, в котором хранятся текст и атрибуты форматирования документа. (Все сказанное выше относится также к классу `QTextEdit`.)

Класс `QTextCursor` позволяет имитировать действия пользователя — перемещаться по документу, вставлять, удалять и выделять текст и т. д. — с помощью предоставляемого им API. В следующей главе, где речь пойдет о программном создании объектов `QTextDocument`, мы поговорим об

этом классе гораздо подробнее. Порядок использования `QTextCursor` совместно с `QPlainTextEdit` и `QTextEdit` чрезвычайно прост: получить курсор для документа, выполнить те или иные действия с помощью API класса `QTextCursor` и сделать модифицированный объект курсором документа, чтобы изменения возымели действие.

Метод `handledCompletedAndSelected()` вызывается при нажатии любой клавиши сразу после вставки единственного найденного завершения. Если нажата клавиша `Enter`, то есть предложение подтверждено, то мы снимаем выделение (напомним, что при наличии единственного варианта завершения вставленный текст выделяется, см. стр. 351). Если нажата клавиша `Esc`, то есть предложение отклонено, то мы удаляем выделенный текст. В обоих случаях мы вызываем метод события `accept()`, уведомляющий `Qt` о том, что оно обработано, и возвращаем `true`, чтобы сообщить переопределенному методу `keyPressEvent()` о ненужности дальнейших действий.

При нажатии любой другой клавиши мы возвращаем `false`, давая возможность обработать нажатую клавишу в другом месте программы. Например, если нажата клавиша с буквой, то обработчик событий нажатия клавиши в базовом классе удалит выделенный текст и вставит букву в текст. Такое поведение при вводе поверх выделенного текста принимается по умолчанию практически в любом виджете редактирования (не только в предоставляемых `Qt`). Следовательно, пользователь может отклонить предложенное завершение, просто продолжая вводить текст.

```
void XmlEdit::mousePressEvent(QMouseEvent *event)
{
    if (completedAndSelected) {
        completedAndSelected = false;
        QTextCursor cursor = textCursor();
        cursor.removeSelectedText();
        setTextCursor(cursor);
    }
    QPlainTextEdit::mousePressEvent(event);
}
```

В момент сразу после вставки единственно возможного продолжения переменная `completedAndSelected` равна `true`, поэтому пользователь может принять или отклонить его с помощью клавиатуры, как мы только что видели. А чтобы сохранить согласованность с поведением в случае раскрывающегося списка вариантов, мы также разрешаем отклонить предложение щелчком мышью в любом месте документа. Код удаления вставленного продолжения точно такой же, как в методе `handledCompletedAndSelected()`.

Вне зависимости от того, принято продолжение или отклонено, сразу после нажатия любой клавиши или щелчка мышью для переменной

`completedAndSelected` необходимо установить значение `false`, иначе последующие нажатия клавиш будут обрабатываться неправильно.

На этом мы завершаем рассмотрение автозавершения. Как мы видели, для включения этой возможности в объекты `QLineEdit` и `QComboBox` достаточно вызвать их метод `setCompleter()`. Но для многострочных виджетов типа `QPlainTextEdit` или `QTextEdit` пришлось поработать. А вот выделение текущей строки и подсветка синтаксиса – вещи куда более простые, хотя последняя задача требует хорошего владения аппаратом регулярных выражений.

Подсветка синтаксиса

В этом подразделе мы сосредоточимся в основном на подсветке синтаксиса, но сначала сделаем небольшое отступление и покажем, как выделить текущую строку. В предыдущем подразделе мы видели, как устанавливается соединение между сигналом `cursorPositionChanged()` объекта `QPlainTextEdit` и слотом `highlightCurrentLine()` (стр. 353). Благодаря ему слот будет вызываться при любом перемещении курсора (все равно, с помощью клавиатуры или мыши).

```
void XmlEdit::highlightCurrentLine()
{
    QList<QTextEdit::ExtraSelection> extraSelections;
    QTextEdit::ExtraSelection selection;
    QBrush highlightColor = palette().alternateBase();
    selection.format.setBackground(highlightColor);
    selection.format.setProperty(QTextFormat::FullWidthSelection,
                                true);
    selection.cursor = textCursor();
    selection.cursor.clearSelection();
    extraSelections.append(selection);
    setExtraSelections(extraSelections);
}
```

Начиная с версии Qt 4.2 классы `QPlainTextEdit` и `QTextEdit` поддерживают возможность добавлять дополнительные области выделения из программы. Основное ее применение – реализовать дополнительную подсветку, например выделение текущей строки или строк программы с расставленными точками прерывания и т. п.

Сначала мы создаем список выделений (в который собираемся добавить новое). Затем устанавливаем цвет фона в ассоциированном с выделением объекте `QTextCharFormat`, а также еще одно из его свойств. Класс `QTextFormat` (базовый класс `QTextCharFormat`) предоставляет механизм свойств, чтобы упростить добавление новых свойств, не нарушая двоичной совместимости с будущими версиями Qt. Одни свойства относятся к абзацам (например, свойства объекта `QTextBlockFormat`), другие – к символам, третьи – к тому и другому. Здесь мы решили, что форматирование выделения должно распространяться на всю его ширину.

Настроив формат выделения, мы получаем курсор документа и снимаем существующее выделение – в результате якорь оказывается в текущей позиции курсора. Затем мы добавляем дополнительное выделение, а именно то, которое только что создали. При обычных обстоятельствах мы после этого не увидели бы никакого выделения – якорь и текущая позиция совпадают, то есть между ними ничего нет, поэтому и выделять нечего. Однако, включив свойство `QTextFormat::FullWidthSelection`, мы распространили выделение на всю строку, содержащую курсор, вне зависимости от положения якоря.

Соединение между сигналом и слотом и метод `highlightCurrentLine()` – вот и все, что нужно для выделения текущей строки. Напротив, для подсветки синтаксиса нам придется создать экземпляр подкласса `QSyntaxHighlighter`, передав ему указатель на наш объект `QTextDocument`, и разумеется, написать сам подкласс (как создается экземпляр, мы видели на стр. 352).

В примерах и в исходном коде Qt имеются готовые подклассы `QSyntaxHighlighter`. В примере *examples/richtext/syntaxhighlighter* есть класс для подсветки синтаксиса C++/Qt. А в примерах из каталога *examples/xmlpatterns* – класс для подсветки синтаксиса XML, который несколько короче и проще показанного ниже. Кроме того, в исходном коде программы *Qt Designer* в каталоге *tools/designer/src/lib/shared* имеется три таких класса: *csshighlighter.cpp* для CSS (каскадные таблицы стилей), *htmlhighlighter.cpp* для HTML и *qscripthighlighter.cpp* для JavaScript. Ни один из них не является частью открытого API Qt, но в качестве отправных точек они вполне годятся. Стоит также обратить внимание на библиотеку GNU Source-Highlight (www.gnu.org/software/src-highlight). Она поддерживает подсветку синтаксиса для многочисленных языков и форматов, а библиотека Source-Highlight Qt (*srchiliteqt.sourceforge.net*) содержит обертку для нее в виде подкласса `QSyntaxHighlighter`.

Один из самых удобных способов реализовать подсветку синтаксиса – воспользоваться регулярными выражениями. Именно так мы здесь и поступим в предположении, что читатель знаком с этим аппаратом¹. При работе с большими документами применение регулярных выражений может привести к снижению производительности (наверное, именно поэтому они не используются в *htmlhighlighter.cpp*).

В любом подклассе `QSyntaxHighlighter` необходимо реализовать метод `highlightBlock()`. Он автоматически вызывается, когда нужно подсветить слова в одной строке текста. Для подсветки синтаксических кон-

¹ Очень краткое введение в регулярные выражения приведено в документации по классу `QRegExp` на странице qt.nokia.com/doc/qregexp.html. Имеется также прекрасная книга Дж. Фридла «Регулярные выражения» (3-е издание, Символ-Плюс, 2008). В ней не упоминается класс `QRegExp`, но рассматриваются регулярные выражения в языке Perl, которые очень похожи (хотя обладают и массой дополнительных возможностей).

струкций, занимающих несколько строк (например, многострочных комментариев), часто приходится хранить в классе переменные состояния, позволяющие выяснить, принадлежит ли заданная строка многострочной конструкции. Для этой цели мы можем ассоциировать с каждой строкой целое число (идентификатор состояния), воспользовавшись методом `setCurrentBlockState()`, и найти состояние предыдущей строки с помощью метода `previousBlockState()`.

В классе `XmlHighlighter` мы поддерживаем два состояния, четыре типа синтаксических элементов и подсвечиваем текст с использованием пяти разных форматов символов. Все переменные определены в закрытой секции класса в файле `xmlhighlighter.hpp`:

```
enum State {Normal=0x01, InComment=0x02};
enum Type {Tag, Attribute, Entity, Comment};

QTextCharFormat tagFormat;
QTextCharFormat attributeNameFormat;
QTextCharFormat attributeValueFormat;
QTextCharFormat entityFormat;
QTextCharFormat commentFormat;
QMultiHash<Type, QRegExp> regexForType;
```

Для ассоциирования одного или нескольких регулярных выражений с каждым типом элемента мы создали объект `QMultiHash`. Обнаружив сопоставление с одним из регулярных выражений, мы применяем форматирование, соответствующее его типу. (Класс `QMultiHash` был описан в предыдущей главе, стр. 301.)

Далее мы рассмотрим конструктор, метод `highlightBlock()` и все вспомогательные методы.

```
XmlHighlighter::XmlHighlighter(QTextDocument *parent)
    : QSyntaxHighlighter(parent)
{
    tagFormat.setForeground(Qt::darkBlue);
    attributeNameFormat.setForeground(Qt::blue);
    attributeValueFormat.setForeground(Qt::darkYellow);
    entityFormat.setForeground(Qt::darkRed);
    commentFormat.setForeground(Qt::darkGreen);
    commentFormat.setFontItalic(true);
    addRegex(Tag, "<[!]?\\w+(?:/>)?", false);
    addRegex(Tag, "(?:</\\w+)?[?]?>");
    addRegex(Attribute, "(\\w+(?:\\.\\w+)?)=(\\\"[^\"]+\\\"|'[^']*'+)");
    addRegex(Entity, "&(?:#\\d+|\\w+);");
    addRegex(Comment, "<!--.*-->");
}
```

В конструкторе мы передаем базовому классу объект `QTextDocument`, к которому применяется подсветка. Кроме того, мы конфигурируем объекты `QTextCharFormat` — в большинстве случаев задается только цвет текста, но для комментариев еще и курсивный шрифт. Естественно, можно

было бы применить и другие виды форматирования, поддерживаемые классом `QTextCharFormat`.

Для каждого XML-элемента, синтаксис которого требуется подсветить, мы добавляем в `QMultiHash` регулярное выражение с помощью закрытого метода `addRegex()`. Его третий аргумент по умолчанию равен `true` (это означает, что нужно использовать нежадный алгоритм поиска самой короткой сопоставляемой подстроки).

Первое регулярное выражение сопоставляется с открывающей частью тега и с samozакрывающимися тегами, например: `<tag`, `<!tag`, `<?tag` или `<tag/>`. Второе выражение сопоставляется с закрывающим тегом или с закрывающей частью тега, например `</tag>`, `?>`, `>`. Третье выражение сопоставляется с атрибутом *ключ=значение*, например `key="value"`. Строго говоря, атрибуты следует подсвечивать, только если они встречаются внутри тега, но мы упростили себе задачу и подсвечиваем любые строки вида *ключ=значение*. Четвертое выражение сопоставляется с XML-компонентами, например `´` или `é`. Последнее выражение сопоставляется с комментариями в пределах одной строки — о том, как распознавать многострочные комментарии, мы поговорим позже. Во всех регулярных выражениях применяется сопоставление с самой короткой возможной строкой.

```
void XmlHighlighter::addRegex(Type type, const QString &pattern,
                               bool minimal)
{
    QRegExp regex(pattern);
    regex.setPatternSyntax(QRegExp::RegExp2);
    regex.setMinimal(minimal);
    regexForType.insert(type, regex);
}
```

Этот вспомогательный метод создает объект `QRegExp` и вставляет его в `QMultiHash`, используя в качестве ключа тип.

При использовании в регулярном выражении «запоминающих» скобок следует всегда вызывать метод `QRegExp::setPatternSyntax(QRegExp::RegExp2)`. Это гарантирует, что при поиске сопоставляемой подстроки максимальной длины (жадный алгоритм) Qt будет вести себя так же, как большинство других движков (в частности, Perl), а не проявлять собственное, довольно странное, поведение. (Ожидается, что такое Perl-подобное поведение будет вариантом по умолчанию в Qt 5.)

```
void XmlHighlighter::highlightBlock(const QString &text)
{
    setCurrentBlockState(Normal);
    highlightPatterns(text);
    highlightComments(text);
}
```

По умолчанию предполагается, что состояние текущей строки `Normal` (хотя это можно изменить). Мы реализовали два подхода к подсветке

синтаксиса и поместили их в два разных метода. В методе `highlightPatterns()` регулярные выражения применяются в контексте одной строки, а в методе `highlightComments()` производится поиск в строке и обрабатываются многострочные комментарии. Метод `highlightComments()` следует вызывать вторым, потому что мы хотим, чтобы в закоментированном XML-коде никакой дополнительной подсветки не было.

```
void XmlHighlighter::highlightPatterns(const QString &text)
{
    QHashIterator<Type, QRegExp> i(regexForType);
    while (i.hasNext()) {
        i.next();
        Type type = i.key();
        const QRegExp &regex = i.value();
        int index = regex.indexIn(text);
        while (index > -1) {
            int length = regex.matchedLength();
            if (type == Tag)
                setFormat(index, length, tagFormat);
            else if (type == Attribute) {
                setFormat(index, regex.pos(2) - index - 1,
                           attributeNameFormat);
                setFormat(regex.pos(2)+1, regex.cap(2).length()-2,
                           attributeValueFormat);
            }
            else if (type == Entity)
                setFormat(index, length, entityFormat);
            else if (type == Comment)
                setFormat(index, length, commentFormat);
            index = regex.indexIn(text, index + length);
        }
    }
}
```

Концептуально в этом методе нет ничего сложного: мы перебираем все регулярные выражения в `QMultiHash`, сопоставляем их с текущей строкой (в аргументе `text`), и если обнаружено соответствие, то применяем подсветку, соответствующую типу `Type`, который ассоциирован с данным регулярным выражением.

То, что для некоторых ключей имеется несколько значений, несущественно, — итератор вернет столько пар ключ–значение, сколько имеется значений в хеше. Обнаружив соответствие, мы, как правило, применяем формат `QTextCharFormat` ко всей сопоставленной подстроке. Исключения составляют атрибуты, в этом случае ключ и значение форматировются по отдельности. Обратите внимание, что при первом обращении к методу `QRegExp::indexIn()` поиск начинается с начала текста (так как смещение не задано), а при последующих обращениях внутри цикла `while` — с позиции, следующей за последней сопоставленной подстрокой.

Одно из рассмотренных выше регулярных выражений сопоставляется с комментариями, которые начинаются и заканчиваются в одной и той же строке. Метод же `highlightComments()` позволяет подсветить комментарии, занимающие несколько строк.

```
void XmlHighlighter::highlightComments(const QString &text)
{
    const QString StartOfComment("<!--");
    const QString EndOfComment("-->");
    if (previousBlockState() > -1 &&
        (previousBlockState() & InComment) == InComment) {
        int end = text.indexOf(EndOfComment);
        if (end == -1) {
            setFormat(0, text.length(), commentFormat);
            setCurrentBlockState(currentBlockState() | InComment);
            return;
        }
        else
            setFormat(0, end + EndOfComment.length(),
                      commentFormat);
    }
    int start = text.lastIndexOf(StartOfComment);
    if (start != -1) {
        int end = text.lastIndexOf(EndOfComment);
        if (end < start) {
            setFormat(start, text.length(), commentFormat);
            setCurrentBlockState(currentBlockState() | InComment);
        }
    }
}
```

По умолчанию «состояние» блока равно -1, поэтому все определенные нами состояния – положительные числа, в данном случае 0x01 (Normal) и 0x02 (InComment). Мы записываем значения в шестнадцатеричном виде, давая понять, что собираемся производить над ними поразрядные операции. В данном примере это необязательно, но в более сложных механизмах подсвечивания состояний могло бы быть больше (0x04, 0x08, 0x10 и т. д.) и имело бы смысл комбинировать их (например, 0x01 | 0x04 дает 0x05, то есть сочетание состояний 1 и 4).

В этом методе нужно рассмотреть три возможных случая:

1. Мы уже находимся внутри многострочного комментария, который либо заканчивается, либо не заканчивается в данной строке.
2. Многострочный комментарий начинается в этой строке.
3. Ничто из вышеперечисленного.

Если мы уже находимся внутри многострочного комментария, то метод `QSyntaxHighlighter::previousBlockState()` вернет состояние `InComment`. Поскольку мы хотим предусмотреть возможность комбинирования

состояний, то вместо того, чтобы сравнивать состояние предыдущего блока непосредственно с `InComment`, мы вычисляем поразрядное И, то есть выделяем бит, соответствующий `InComment`, и сравниваем именно его. Так мы сможем распознать нужное нам состояние, даже если оно комбинируется с каким-то другими. Находясь внутри многострочного комментария, мы проверяем, заканчивается ли он в данной строке. Если нет, то форматируем всю строку как комментарий и комбинируем состояние `InComment` с уже имеющимся состоянием текущего блока (которому мы раньше присвоили значение `Normal`, стр. 368), а затем возвращаем управление, потому что больше делать нечего. Если же комментарий заканчивается в данной строке, то мы форматируем как комментарий ее часть до конца комментария, а потом продолжаем выполнение метода, потому что дальше в строке может начинаться новый многострочный комментарий.

Если мы не находимся внутри многострочного комментария или комментарий закончился в текущей строке, то смотрим, не начинается ли в этой строке новый многострочный комментарий. Обнаружив начало комментария, мы смотрим, нет ли в той же строке его конца. Комментарии, которые начинаются и заканчиваются в одной строке, следует игнорировать, потому что их мы форматируем в другом месте. Кроме того, надо следить за тем, не найден ли конец комментария раньше начала (тогда любой найденный конец комментария будет относиться к предыдущему комментарию и не может являться концом многострочного комментария, начинающегося в этой строке). Если в этой строке действительно начинается многострочный комментарий, то мы применяем соответствующее форматирование к части строки от его начала и до конца строки; при этом состояние `InComment` комбинируется с текущим состоянием.

В любом другом случае мы не делаем ничего, потому что либо нечего форматировать, либо форматирование уже было произведено в методе `highlightPatterns()`.

На этом мы завершаем рассмотрение класса `XmlHighlighter` и закрываем вопрос о подсветке синтаксиса. Для реализации подсветки мы в одних случаях использовали регулярные выражения, а в других – простой поиск в строке. Можно было бы вместо этого написать лексический анализатор и читать текст посимвольно. А если бы переменных состояния и битовых флагов оказалось недостаточно, то можно ассоциировать с каждой строкой пользовательские данные (с помощью метода `QSyntaxHighlighter::setCurrentBlockUserData()`).

Разумеется, иногда нас интересует не столько подсветка, сколько предоставление пользователю возможности вводить текст шрифтами разных цветов и начертаний по его выбору. В следующих разделах мы покажем, как это можно сделать в однострочном и многострочном редакторе.

Однострочный редактор форматированного текста

В Qt уже имеется виджет, предназначенный для редактирования форматированного текста, — `QTextEdit`, но он ориентирован на ввод нескольких строк, а нам иногда желательно ограничиться одной строкой. Ничего страшного: написав подкласс `QTextEdit`, мы сможем воспользоваться всеми средствами отображения и редактирования, которые он предоставляет, но при этом разрешим отображать и редактировать только одну строку. Можно было бы, конечно, унаследовать `QPlainTextEdit`, но класс `QTextEdit` удобнее, потому что содержит много полезных слотов, например `QTextEdit::setFontItalic()`, которых в `QPlainTextEdit` нет.

В этом разделе мы создадим виджет `RichTextLineEdit` для редактирования одной строки текста, который позволит применять форматирование к отдельным символам и словам, в частности выделять их полужирным или курсивным шрифтом и задавать цвет. Виджет `RichTextLineEdit` использовался делегатом `RichTextDelegate` в приложениях `Timelog` из главы 5 (стр. 216). На рис. 9.4 показаны всплывающая подсказка и контекстное меню, ассоциированные с этим виджетом.

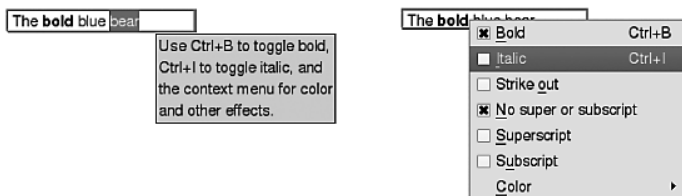


Рис. 9.4. Всплывающая подсказка и контекстное меню в виджете `RichTextLineEdit`

Ниже приведен фрагмент из объявления класса, в котором, однако, опущена большая часть закрытых слотов, методов и данных (хотя мы рассмотрим их по ходу описания открытых и защищенных методов и слотов).

```
class RichTextLineEdit : public QTextEdit
{
    Q_OBJECT

public:
    explicit RichTextLineEdit(QWidget *parent=0);
    QString toSimpleHtml() const;
    QSize sizeHint() const;
    QSize minimumSizeHint() const;

public slots:
    void toggleItalic() { setFontItalic(!fontItalic()); }
```

```

void toggleBold() { setFontWeight(fontWeight() > QFont::Normal
                             ? QFont::Normal : QFont::Bold); }

signals:
    void returnPressed();

protected:
    void keyPressEvent(QKeyEvent *event);
    ...

private:
    enum Style {Bold, Italic, StrikeOut, NoSuperOrSubscript,
               Subscript, Superscript};
    ...
};

```

Методы `setFontItalic()` и `setFontWeight()` унаследованы от класса `QTextEdit`, и хотя Qt умеет обрабатывать различные вариации шрифтовых начертаний, мы решили ограничиться только включением и выключением полужирного.

Сигнал `returnPressed()` нужен для того, чтобы сделать `RichTextLineEdit` больше похожим на `QLineEdit` и дать возможность делегату (или любому объекту, подключившемуся к этому сигналу) узнать о том, что пользователь закончил редактирование. В переопределенном обработчике события `keyPressEvent()` мы обнаруживаем нажатие клавиши Enter и там же генерируем сигнал.

Поскольку мы унаследовали классу `QTextEdit`, то нет необходимости самостоятельно заниматься рисованием и реализовывать значительную часть стандартного поведения (например, копирование/вставку и отмену/повтор операции). А значит, единственное, что необходимо сделать, — это гарантировать наличие в `RichTextLineEdit` всего одной строки и добавить нужное нам специализированное поведение. Как обычно, начнем рассмотрение с конструктора.

```

RichTextLineEdit::RichTextLineEdit(QWidget *parent)
    : QTextEdit(parent)
{
    setLineWrapMode(QTextEdit::NoWrap);
    setWordWrapMode(QTextOption::NoWrap);
    setAcceptRichText(true);
    setTabChangesFocus(true);
    setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    createShortcuts();
    createActions();
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, SIGNAL(customContextMenuRequested(
        const QPoint&)), this,
        SLOT(customContextMenuRequested(const QPoint&)));
}

```

В конструкторе производятся все очевидные настройки: отключение разбиения на строки и переноса слов, включение режима, при котором вставляемая из буфера обмена HTML-разметка воспринимается как форматированный текст, настройка клавиши Tab для передачи фокуса, а не вставки символа табуляции, и сокрытие полос прокрутки.

Реализовать контекстное меню проще всего, добавив в виджет действия QAction и задав в качестве политики контекстного меню Qt::Actions-ContextMenu, после чего всю заботу о нем можно переложить на Qt. Однако в этом конкретном виджете такое решение не годится, потому что мы хотим обновлять флажки действий в момент вызова контекстного меню. Поэтому мы установим политики Qt::CustomContextMenu и соединим сигнал customContextMenuRequested(), генерируемый перед показом контекстного меню, если установлена эта политика, с одноименным пользовательским слотом, который будет настраивать и показывать меню самостоятельно.

```
void RichTextLineEdit::createShortcuts()
{
    QShortcut *boldShortcut = new QShortcut(QKeySequence::Bold,
        this, SLOT(toggleBold()));
    QShortcut *italicShortcut = new QShortcut(QKeySequence::Italic,
        this, SLOT(toggleItalic()));
    setToolTip(tr("<p>Use %1 to toggle bold, %2 to toggle italic, "
        "and the context menu for color and other effects.")
        .arg(boldShortcut->key().toString(
            QKeySequence::NativeText))
        .arg(italicShortcut->key().toString(
            QKeySequence::NativeText)));
}
```

Мы назначаем комбинации клавиш для выбора полужирного и курсивного начертаний, создав подходящие объекты QShortcut и используя стандартные элементы перечисления QKeySequence. На платформах Linux и Windows комбинацией клавиш для выбора полужирного начертания будет Ctrl+B, а в Mac OS X – ⌘+B¹.

Назначив комбинации клавиш, мы создаем сообщающую о них всплывающую подсказку, где комбинации клавиш преобразуются в стро-

¹ Клавише Ctrl, входящей во многие стандартные комбинации в Windows и Linux (например Ctrl+C для копирования), в Mac OS X обычно соответствует клавиша ⌘ (например, ⌘+C). Поэтому при первоначальном переносе Qt на платформу Mac OS X для упрощения кросс-платформенной разработки Ctrl просто заменяли на ⌘. И наоборот, нажатие, скажем, ⌘+C в Mac OS X считалось эквивалентным нажатию Ctrl+C на других платформах. Но по прошествии времени это решение перестало казаться таким уж удачным, и начиная с версии Qt 4.6 стало возможно отменить подмену, обратившись к методу QApplication::setAttribute(Qt::AA_MacDontSwapCtrlAndMeta).

ковую форму, соответствующую платформе, на которой работает программа.

```
void RichTextLineEdit::createActions()
{
    boldAction = createAction(tr("Bold"), Bold);
    ...
    colorAction = new QAction(tr("Color"), this);
    colorAction->setMenu(createColorMenu());

    addActions(QList<QAction*>() << boldAction << italicAction
        << strikeOutAction << noSubOrSuperScriptAction
        << superScriptAction << subScriptAction << colorAction);
    AQP::accelerateActions(actions());
}
```

Этот метод создает действия – большую их часть мы опустили, так как все, кроме последнего, создаются одинаково. Метод `createAction()`, который мы скоро рассмотрим, принимает два аргумента: текст, отображаемый в меню, и данные – в нашем случае один из элементов перечисления.

Действие `colorAction` необычно тем, что с ним не соединяется никакой слот, а ассоциируется другое меню.

В конце метода мы добавляем в `RichTextLineEdit` все действия; с тем же успехом их можно было бы сохранить в закрытом списке `QList<QAction*>`. А напоследок назначаем действиям клавиши-акселераторы.

```
QAction *RichTextLineEdit::createAction(const QString &text,
                                         const QVariant &data)
{
    QAction *action = new QAction(text, this);
    action->setData(data);
    action->setCheckable(true);
    action->setChecked(false);
    connect(action, SIGNAL(triggered()), SLOT(applyTextEffect()));
    return action;
}
```

Этот вспомогательный метод создает нужные нам действия, устанавливает для них режим вывода флажков и первоначально делает последние неотмеченными. Все действия соединяются со слотом `applyTextEffect()`.

```
QMenu *RichTextLineEdit::createColorMenu()
{
    QMenu *colorMenu = new QMenu(this);
    QPixmap pixmap(22, 22);

    typedef QPair<QColor, QString> ColorPair;

    foreach (const ColorPair &pair, QList<ColorPair>()
        << qMakePair(QColor(Qt::black), tr("Black"))
```

```

...
        << qMakePair(QColor(Qt::darkRed), tr("Dark Red"))) {
        pixmap.fill(pair.first);
        QAction *action = colorMenu->addAction(pixmap,
                                                pair.second);

        action->setData(pair.first);
    }
    connect(colorMenu, SIGNAL(triggered(QAction*)),
            this, SLOT(applyColor(QAction*)));
    AQP::accelerateMenu(colorMenu);
    return colorMenu;
}

```

Этот метод создает меню цветов, ассоциированное с действием `colorAction`; оно показано на рис. 9.5. Сначала мы создаем в куче новый объект `QMenu`, а затем список пар — цвет `QColor` и его название, — который тут же обходим. Для каждой пары создается действие, представленное растровым изображением соответствующего цвета и его именем, а в качестве связанных с действием данных задаем значение `QColor`. (В Qt объекты `QPixmap` ведут себя как значения, потому что копируются при записи, и значит, каждое действие получает свое собственное уникальное изображение. В главе 12 мы покажем, как создать образцы цветов, более симпатичные, чем невзрачный квадратик.)

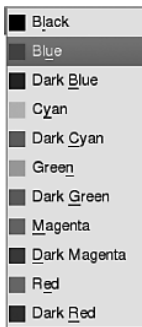


Рис. 9.5. Подменю цветов в виджете *RichTextLineEdit*

Мы соединяем действие меню `triggered()` со слотом `applyColor()`; передаваемый вместе с сигналом указатель `QAction*` соответствует пункту меню, выбранному пользователем. В самом конце мы назначаем пунктам меню клавиши-акселераторы и возвращаем это меню вызывающей программе.

Использовать `typedef` здесь необходимо, потому что макрос `foreach` имеет синтаксис `foreach (item, sequence)`. Запятая является неотъемлемой частью конструкции и служит для разделения элемента и последовательности. (То, что последовательность может содержать запятые, неважно; значение имеет лишь первая запятая.) Поэтому для пар элемен-


```

        break;
    }
    mergeCurrentCharFormat(format);
}
}

```

Этот слот вызывается при выборе любого пункта контекстного меню, кроме подменю цветов. Сначала мы приводим вызвавший объект к типу указателя на QAction (это приведение всегда должно завершаться успешно), а затем получаем ассоциированные с действием данные – в нашем случае значение закрытого перечисления Style, определенного в классе RichTextLineEdit. Далее на основе этого значения мы решаем, какое форматирование применить. Если речь идет о полужирном и курсивном начертании, то достаточно переключить соответствующий режим и вернуться. А для перечеркивания или выравнивания по вертикали необходимо сначала модифицировать копию текущего объекта QTextCharFormat, а потом объединить ее с текущим форматом, чтобы изменение вступило в силу.

Метод QObject::sender() возвращает объект QObject, вызвавший слот (или 0, если слот был вызван напрямую, как метод). В данном случае слот вызывается только объектами QAction. Можно было бы вместо qobject_cast<>() использовать dynamic_cast<>(), но мы предпочитаем для классов, производных от QObject, всегда вызывать метод qobject_cast<>(), потому что он работает даже при отключенной поддержке RTTI и через границы динамических библиотек. Альтернативой методу sender() может служить использование QSignalMapper; это обеспечило бы лучшую инкапсуляцию, правда, ценой небольшого увеличения размера кода. И конечно, можно было бы создать отдельные слоты для каждого действия.

```

void RichTextLineEdit::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_Enter ||
        event->key() == Qt::Key_Return) {
        emit returnPressed();
        event->accept();
    }
    else
        QTextEdit::keyPressEvent(event);
}

```

Этот обработчик переопределен только ради сигнала returnPressed(); при нажатии клавиши Enter или Return кроме генерации этого сигнала больше ничего не происходит. Остальные же клавиши передаются базовому классу.

```

void RichTextLineEdit::customContextMenuRequested(const QPoint &pos)
{
    updateContextMenuActions();
    QMenu menu(this);
    menu.addAction(actions());
}

```

```

        menu.exec(mapToGlobal(pos));
    }

```

Этот слот получает управление, когда пользователь открывает контекстное меню (щелкнув правой кнопкой мыши или воспользовавшись платформенно-зависимой комбинацией клавиш). Соответствующее соединение было установлено в конструкторе. Сначала мы вызываем метод `updateContextMenuActions()`, который правильно устанавливает состояние отмеченности всех действий. Затем создаем само меню и добавляем в него действия виджета (они были созданы ранее). В заключение мы открываем меню, предварительно преобразовав координаты виджета в координаты экрана (глобальные), которые ожидает метод `QMenu::exec()`.

Ранее мы уже видели, что при выборе цвета вызывается метод `applyColor()`. Если же выбран любой другой пункт, то будет вызван метод `applyTextEffect()`.

```

void RichTextLineEdit::updateContextMenuActions()
{
    boldAction->setChecked(fontWeight() > QFont::Normal);
    italicAction->setChecked(fontItalic());
    const QTextCharFormat &format = currentCharFormat();
    strikeOutAction->setChecked(format.fontStrikeOut());
    noSubOrSuperScriptAction->setChecked(format.verticalAlignment() ==
                                         QTextCharFormat::AlignNormal);
    superScriptAction->setChecked(format.verticalAlignment() ==
                                  QTextCharFormat::AlignSuperScript);
    subScriptAction->setChecked(format.verticalAlignment() ==
                                QTextCharFormat::AlignSubScript);
}

```

Этот метод обновляет действия контекстного меню, чтобы они отражали состояние текста в текущей позиции курсора. Несмотря на длинные строки, код очень простой: мы просто устанавливаем или сбрасываем флаги отмеченности каждого действия в зависимости от формата текста.

```

QSize RichTextLineEdit::sizeHint() const
{
    QFontMetrics fm(font());
    return QSize(document()->idealWidth() + fm.width("W"), fm.height() + 5);
}

```

Предпочтительный размер (size hint) должен быть равен идеальному размеру виджета. При вычислении размера мы учитываем только текстовое содержимое (поскольку неотображаемая HTML-разметка искажала бы результат, о чем уже шла речь при обсуждении метода `RichTextDelegate::sizeHint()` на стр. 221) и добавляем ширину одной буквы **W** для полей. Аналогично при вычислении высоты мы добавили 5 пикселей для вертикального поля.

```

QSize RichTextLineEdit::minimumSizeHint() const
{
    QFontMetrics fm(font());
    return QSize(fm.width("WWW"), fm.height() + 5);
}

```

Минимальный предпочтительный размер — это тот наименьший размер, до которого Qt может уменьшить виджет. Здесь мы установили ширину равной ширине четырех букв W, а высоту оставили той же, что в `sizeHint()`.

Вот мы и рассмотрели все методы, реально необходимые классу `RichTextLineEdit`. Однако мы включили еще один дополнительный метод, который сейчас и обсудим.

В базовом классе `QTextEdit` имеется метод `toHtml()`, который возвращает текст в формате HTML. Мы решили игнорировать его и предоставить собственный метод `toSimpleHtml()`, потому что нам нужна лишь малая часть подмножества HTML, поддерживаемого классом `QTextEdit`, и, введя подобное ограничение, мы сумеем получить максимально компактный HTML-код. Чтобы было понятно, о чем идет речь, рассмотрим такой HTML-текст: `bold blue bear`. Для него метод `QTextEdit::toHtml()` вернет следующий код (некоторые пробелы заменены знаками новой строки, чтобы текст уместился на странице):

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html40/strict.dtd">
<html><head><meta name="qrichtext" content="1" />
<style type="text/css">p, li { white-space: pre-wrap; }</style>
</head><body style=" font-family:'Nimbus Sans L';
font-size:12pt; font-weight:400; font-style:normal;">
<p style=" margin-top:0px; margin-bottom:0px; margin-left:0px;
margin-right:0px; -qt-block-indent:0; text-indent:0px;">The
<span style=" font-weight:600;">bold</span>
<span style=" color:#0000ff;">blue</span> bear</p>
</body></html>

```

Длина этого кода составляет 545 знаков, но на практике возможны небольшие вариации, потому что шрифт по умолчанию (в данном случае `Nimbus Sans L`) на разных компьютерах различается. Влияние может оказывать также версия Qt. А теперь сравните это с кодом, который порождает метод `toSimpleHtml()`:

```

The <b>bold</b> <font color="#0000ff">blue</font> bear

```

Всего 54 знака. (Строго говоря, это некорректная HTML-разметка; необходимы еще по крайней мере объявление `DOCTYPE` и теги `<html>`, `<head>` и `<body>`.) Понятно, что возможности метода `toSimpleHtml()` крайне ограничены и метод `toHtml()` гораздо более развитый, но для однострочных простых фрагментов HTML весьма желательно использовать максимально компактный формат.

```

QString RichTextLineEdit::toSimpleHtml() const
{
    QString html;
    for (QTextBlock block = document()->begin(); block.isValid();
         block = block.next()) {
        for (QTextBlock::iterator i = block.begin(); !i.atEnd();
             ++i) {
            QTextFragment fragment = i.fragment();
            if (fragment.isValid()) {
                QTextCharFormat format = fragment.charFormat();
                QColor color = format.foreground().color();
                QString text = Qt::escape(fragment.text());
                QStringList tags;
                if (format.verticalAlignment() ==
                    QTextCharFormat::AlignSubScript)
                    tags << "sub";
                else if (format.verticalAlignment() ==
                        QTextCharFormat::AlignSuperScript)
                    tags << "sup";
                if (format.fontItalic())
                    tags << "i";
                if (format.fontWeight() > QFont::Normal)
                    tags << "b";
                if (format.fontStrikeOut())
                    tags << "s";
                while (!tags.isEmpty())
                    text = QString("<%1>%2</%1>")
                        .arg(tags.takeFirst()).arg(text);
                if (color != QColor(Qt::black))
                    text = QString("<font color=\"%1\">%2</font>")
                        .arg(color.name()).arg(text);
                html += text;
            }
        }
    }
    return html;
}

```

Нам нужно пробежаться по тексту, который хранится во внутреннем для базового класса объекте `QTextDocument`, и вывести соответствующую HTML-разметку, учитывая все встретившиеся атрибуты форматирования текста. Объект `QTextDocument` имеет иерархическую структуру: «корневой фрейм», содержащий последовательность объектов `QTextBlock` и `QTextFrame`. Фреймы могут содержать блоки (и другие элементы, например списки и таблицы). Каждый блок состоит из одного или нескольких фрагментов текста, каждый из которых отформатирован единообразно. (Структура `QTextDocument` обсуждалась в первом разделе этой главы; см. также рис. 9.1 на стр. 347.)

Так как нас интересует только текст и, более того, только один блок, поскольку виджет `RichTextLineEdit` содержит всего одну строку, то мы

можем ограничиться обходом одних лишь текстовых блоков, игнорируя фреймы, – да и в любом случае никаких фреймов, кроме корневого, быть не может. (В данном конкретном случае можно было бы даже опустить внешний цикл и обрабатывать только первый текстовый блок, но мы предпочитаем более общие решения.)

Получив текстовый блок (в данном случае строку), мы должны обойти все ее текстовые фрагменты. Если весь текст в строке отформатирован одинаково, то будет всего один фрагмент. Но в нашем примере присутствует несколько разных форматов (например, полужирный и цветной текст), поэтому блок будет разбит на такие пять фрагментов: «The » (включая следующий за словом пробел), «bold» (полужирный шрифт), « » (пробел), «blue» (синий цвет) « bear» (включая предшествующий пробел).

Для каждого фрагмента мы извлекаем его формат (хранится в объекте `QTextCharFormat`), цвет и текст. К тексту применяется HTML-кодирование, то есть символы `&`, `<` и `>` заменяются своими HTML-эквивалентами (`&`, `<` и `>`). Затем мы создаем список строк, содержащих HTML-теги, соответствующие формату фрагмента. Список необходим потому, что к одному фрагменту может применяться несколько атрибутов форматирования, например полужирный и курсив. Получив нужные теги, мы обертываем ими текст, по очереди добавляя в начало открывающий, а в конец – закрывающий тег. А в заключение, если цвет текста отличен от черного, добавляем еще тег ``, задающий цвет. После этого текст вместе с добавленной HTML-разметкой дописывается в конец строки `html`, которую и возвращает этот метод.

На этом мы завершаем рассмотрение класса `RichTextLineEdit`. С помощью этого или написанного на его основе класса мы можем предложить пользователям средства для ввода форматированного текста в одной строке и сохранения этого текста в максимально компактном формате. Разумеется, в дистрибутиве Qt уже есть многострочный редактор форматированного текста – класс `QTextEdit`. Но, чтобы сделать его по-настоящему полезным пользователям, мы должны предоставить какие-то средства для форматирования, подобные тем, что реализованы в нашем виджете `RichTextLineEdit`. Этим мы и займемся в следующем разделе.

Многострочный редактор форматированного текста

Входящий в состав Qt класс `QTextEdit` предоставляет столь обширную функциональность, что нам не составит большого труда превратить его в полезный редактор форматированного текста. В этом разделе мы создадим виджет `TextEdit`, показанный на рис. 9.6. Он объединяет `QTextEdit` с двумя панелями инструментов, которые позволяют применять и отменять выделение полужирным шрифтом и курсивом, задавать гар-

нитуру шрифта и его кегль и выравнивать текст. Мы могли бы поддерживать гораздо больше функций форматирования символов и абзацев, но и этого достаточно для демонстрации принципов и приемов. Приложение Text Edit (`textedit`) предоставляет средства для тестирования и экспериментирования с классом `TextEdit`, а в приложении Page Designer в главе 12 (стр. 470) этот класс реально используется, правда, без возможности выравнивания.

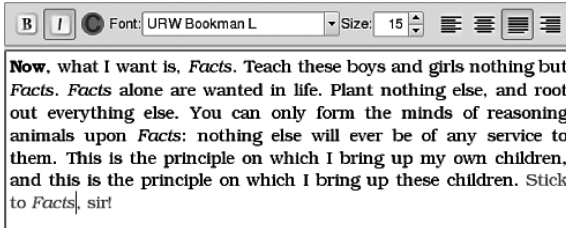


Рис. 9.6. Компонент `TextEdit`

Класс `TextEdit` должен реализовывать две основных функции. Во-первых, предоставить средства для нужного пользователю форматирования. Во-вторых, как-то индизировать форматирование, действующее в позиции курсора.

Большая часть конструирования экземпляра `TextEdit` вполне традиционная: мы создаем и конфигурируем виджеты, создаем действия, komponуем виджеты и устанавливаем соединения. Приведем для начала полный перечень закрытых переменных-членов виджета (из файла `textedit.hpp`).

```
QToolBar *fontToolBar;
QAction *boldAction;
QAction *italicAction;
QAction *colorAction;
QColorDialog *colorDialog;
QFontComboBox *fontComboBox;
QDoubleSpinBox *fontSizeSpinBox;
QToolBar *alignmentToolBar;
QAction *alignLeftAction;
QAction *alignCenterAction;
QAction *alignJustifyAction;
QAction *alignRightAction;
QTextEdit *textEdit;
```

Действия, связанные с выравниванием, помещаются в группу действий в методе `createWidgets()` (не показан). При этом гарантируется, что в каждый момент времени может быть выбран только один вид выравнивания, поскольку по умолчанию группа считает, что входящие в нее действия взаимно исключают друг друга.

Значительную часть поведения виджета можно реализовать, обращаясь к агрегированному объекту `textEdit` с помощью простых методов-адаптеров. Вот несколько примеров, взятых из заголовочного файла:

```
QString toHtml() const { return textEdit->toHtml(); }
void setHtml(const QString &html) { textEdit->setHtml(html); }
void setBold(bool on)
    { textEdit->setFontWeight(on ? QFont::Bold : QFont::Normal); }
void setFontPointSize(double points)
    { textEdit->setFontPointSize(static_cast<qreal>(points)); }
void setFontFamily(const QFont &font)
    { textEdit->setFontFamily(font.family()); }
void alignLeft() { textEdit->setAlignment(Qt::AlignLeft); }
```

Имеются и другие методы-адаптеры, которые также перепоручают работу агрегированному объекту `QTextEdit`, например `toPlainText()`, `alignCenter()` и т. д. Мы не стали их показывать, потому что все они аналогичны приведенным выше.

Одни действия, например переключение курсива, можно реализовать просто путем соединения сигнала со слотом, другие требуют написания адаптера или специального метода. Мы рассмотрим некоторые соединения и специальные методы, но опустим конструктор, а также методы `createWidgets()` и `createLayout()` — все они стандартны.

```
void QTextEdit::createConnections()
{
    connect(fontComboBox, SIGNAL(currentFontChanged(const QFont&)),
            this, SLOT(setFontFamily(const QFont&)));
    connect(fontSizeSpinBox, SIGNAL(valueChanged(double)),
            this, SLOT(setFontPointSize(double)));
    connect(boldAction, SIGNAL(toggled(bool)), this, SLOT(setBold(bool)));
    connect(italicAction, SIGNAL(toggled(bool)),
            textEdit, SLOT(setFontItalic(bool)));
    connect(colorAction, SIGNAL(triggered()), this, SLOT(setColor()));
    connect(alignLeftAction, SIGNAL(triggered()), this, SLOT(alignLeft()));
    ...
    connect(textEdit, SIGNAL(currentCharFormatChanged(
            const QTextCharFormat&)),
            this, SLOT(currentCharFormatChanged(const QTextCharFormat&)));
    connect(textEdit, SIGNAL(cursorPositionChanged()),
            this, SLOT(cursorPositionChanged()));
    connect(textEdit, SIGNAL(textChanged()), this, SIGNAL(textChanged()));
}
```

Первые пять соединений касаются смены атрибутов символов, шестое — смены атрибутов абзаца. Сигнал `currentFontChanged()` от комбинированного списка `QFontComboBox` нужен для изменения шрифтовой гарнитуры. Кегль шрифта изменяется с помощью соединения со счетчиком `fontSizeSpinBox`. Для переключения полужирного и курсивного начертания достаточно соединений с двухпозиционными действиями, которые изменяют соответствующие атрибуты шрифта. Выбор действия `color-`

Action приводит к появлению диалогового окна выбора цвета. Помимо соединения с действием для выравнивания по левому краю, имеются аналогичные соединения для других выравниваний (по центру, по ширине и по правому краю), но они не показаны.

Обратите внимание, что действие переключения курсива удалось напрямую соединить с агрегированным объектом `QTextEdit`, тогда как для всех остальных действий (за исключением выбора цвета) пришлось написать небольшие методы-адаптеры, реализованные в заголовочном файле (см. выше), которые уже вызывают нужные методы `QTextEdit`.

Два предпоследних соединения нужны для того, чтобы виджеты на панели инструментов правильно отражали форматирование символа и абзаца в позиции курсора.

Это состояние может изменяться как вследствие перемещения по документу (пользователь щелкает где-то мышью или нажимает клавиши со стрелками и т. п.), так и из-за смены состояния в текущей позиции (например, пользователь нажимает кнопку полужирного шрифта на панели инструментов). Последнее соединение (сигнала с сигналом) позволяет виджету `TextEdit` сгенерировать свой сигнал `textChanged()` при получении одноименного сигнала от агрегированного объекта `QTextEdit`.

Чуть позже мы рассмотрим оба метода, позволяющих актуализировать состояние кнопок на панели инструментов, но сначала приведем метод `setColor()` и сопутствующий ему слот, потому что это единственный метод, реализованный не с помощью адаптера в заголовочном файле.

```
void QTextEdit::setColor()
{
    if (!colorDialog) {
        colorDialog = new QColorDialog(this);
        connect(colorDialog, SIGNAL(colorSelected(const QColor&)),
                this, SLOT(updateColor(const QColor&)));
    }
    colorDialog->setCurrentColor(textEdit->textColor());
    colorDialog->open();
}
```

Этот метод выводит платформенно-зависимое диалоговое окно для выбора цвета, в котором текущий цвет уже установлен в качестве начального¹. При нажатии кнопки ОК генерируется сигнал `colorSelected()`. Этот сигнал мы соединили со слотом `updateColor()`.

¹ К сожалению, в Qt 4.6 на платформе Mac OS X это решение работает не вполне корректно. На ноутбуке MacBook автора с ОС Leopard (10.5.8) программа, собранная с версией Qt 4.6.0, завершается аварийно, а в случае версий Qt 4.6.1 и 4.6.2 первое задание цвета работает нормально, но попытка задать его во второй раз приводит к зависанию. Поэтому в исходном коде имеются директивы `#if`, благодаря которым для комбинации Mac OS X с версией Qt 4.6 вызывается статический метод `QColorDialog::getColor()`.

```
void TextEdit::updateColor(const QColor &color)
{
    textEdit->setTextColor(color);
    updateColorSwatch();
}
```

Этот слот изменяет цвет текста на текущий и обновляет образец цвета на панели инструментов. Действие смены цвета на панели представлено пиктограммой с жирной буквой С на фоне того цвета, которым выводится текст в позиции курсора. При смене цвета пиктограмму необходимо обновить, что и делает метод `updateColorSwatch()`, показанный ниже.

```
void TextEdit::updateColorSwatch()
{
    colorAction->setIcon(colorSwatch(textEdit->textColor(), QSize(48, 48)));
}
```

Этот метод обращается к глобальной функции `colorSwatch()`, которая принимает значения цвета и размера и возвращает соответствующую пиктограмму. Функция `colorSwatch()` и некоторые ей подобные (`brushSwatch()`, `penStyleSwatch()`, `penCapSwatch()` и `penJoinSwatch()`) используются в приложении Page Designer в главе 12 (`colorSwatch()`).

Хотя код занимает всего одну строку, мы сочли разумным поместить его в отдельный метод, поскольку он используется в нескольких местах.

```
void TextEdit::currentCharFormatChanged(const QTextCharFormat &format)
{
    fontComboBox->setCurrentFont(format.font());
    fontSizeSpinBox->setValue(format.fontPointSize());
    boldAction->setChecked(format.fontWeight() == QFont::Bold);
    italicAction->setChecked(format.fontItalic());
    updateColorSwatch();
}
```

Этот слот вызывается при смене текущего формата *символов*, но не абзацев (например, смены выравнивания). Он просто обновляет состояние виджетов на панели инструментов в соответствии с переданным форматом. Формат может измениться как вследствие смены состояния (например, пользователь выбирает другой цвет), так и в результате перемещения по документу (курсор переместился к символу, формат которого отличается от формата предыдущего символа).

```
void TextEdit::cursorPositionChanged()
{
    QTextCursor cursor = textEdit->textCursor();
    QTextBlockFormat format = cursor.blockFormat();
    switch (format.alignment()) {
        case Qt::AlignLeft:
            alignLeftAction->setChecked(true); break;
        case Qt::AlignCenter:
            alignCenterAction->setChecked(true); break;
```

```
case Qt::AlignJustify:
    alignJustifyAction->setChecked(true); break;
case Qt::AlignRight:
    alignRightAction->setChecked(true); break;
}
}
```

Этот слот нужен, чтобы отслеживать смену формата *абзаца*. Любое изменение позиции курсора может привести к тому, что курсор окажется в другом абзаце. Мы разрешаем пользователю менять только выравнивание абзацев, поэтому лишь за этим аспектом форматирования и нужно следить. Здесь мы просто делаем отмеченным действие, соответствующее выравниванию текущего абзаца, полагаясь на то, что группа, в которую оно включено, сделает все остальные действия неотмеченными.

Для виджета мы ограничились наиболее употребительными способами форматирования символов и одним лишь выравниванием для абзацев. Мы ставили целью продемонстрировать, как можно предоставить пользователю средства изменения формата и как отразить изменения в пользовательском интерфейсе. Но, конечно, можно было бы реализовать и массу других атрибутов. Например, в классе `QTextCharFormat` есть методы для управления надчеркиванием, зачеркиванием и подчеркиванием (с указанием стиля и цвета линии), а также для задания URL-адреса и всплывающей подсказки. Выше, при обсуждении однострочного редактора, мы видели, что поддерживаются также надстрочные и подстрочные индексы (стр. 373). Для абзацев тоже есть немало других атрибутов. Так, в классе `QTextBlockFormat` имеются методы для задания отступа первой строки абзаца, общего отступа, полей и положения позиций табуляции.

Будь мы более амбициозны, мы могли бы предоставить пользователю средства для создания списков и таблиц, вставки гиперссылок и изображений. На самом деле реализовать базовую поддержку маркированных списков очень легко: стоит лишь вызвать метод `QTextEdit::setAutoFormatting(QTextEdit::AutoBulletList)` — и пользователь сможет начать маркированный список, просто введя символ звездочки (*) в начале строки. Однако для увеличения и уменьшения отступа, а также для вставки и редактирования таблиц придется поработать. Все это мы оставляем в качестве упражнения тем читателям, которые захотят глубже разобраться в механизмах Qt для работы с форматированным текстом.

На этом мы завершаем рассмотрение редакторов форматированного текста, в том числе автозавершения и подсветки синтаксиса. Но как быть, если нам понадобится создавать отформатированные документы программно, например расчетные листки или счета, выставляемые клиентам в конце месяца? А как распечатать такие документы или экспортировать их в одном из стандартных форматов? Этими вопросами мы займемся в следующей главе. Там же мы увидим, как можно просто «нарисовать» желаемое, вообще не создавая объект `QTextDocument`.

10

Создание форматированных текстовых документов

- Качество файла, в который экспортирован QTextDocument
- Создание объектов QTextDocument
- Экспорт и печать документов
- Рисование страниц

В этой главе мы рассмотрим три разных подхода к созданию форматированных текстовых документов и покажем, как экспортировать такие документы в различных стандартных форматах и как их распечатывать¹. Предполагается знакомство читателя с основами класса QTextDocument в объеме первого раздела предыдущей главы (стр. 347). Сначала мы рассмотрим заполнение документа QTextDocument текстом в формате HTML, затем – заполнение документа с помощью класса QTextCursor и, наконец, рисование документа с помощью QPainter, в данном случае просто для экспорта и печати, без какого-либо представления в памяти.

К сожалению, в этой области программирования Qt может разочаровать, особенно тех, кто ищет кросс-платформенное решение. Дело в том, что поведение Qt в части экспорта и печати форматированных документов существенно зависит от платформы и номера версии. Поэтому мы проведем сравнение особенностей версий Qt 4.5.2 и Qt 4.6.1 на платфор-

¹ В предыдущей главе мы отмечали, что под форматированным текстом в Qt понимается способ представления данных в памяти; его не следует путать с другими определениями форматированного текста.

мах Mac OS X, Linux и Windows, чтобы вам было проще выбрать подход, наиболее отвечающий конкретным обстоятельствам. Отметим, однако, что выбранный в качестве примера документ довольно сложен – он включает таблицу, внедренные SVG-изображения и форматированный текст, поэтому не исключено, что для других документов качество результата будет несколько, а иногда даже радикально, отличаться.

Пример двухстраничного документа, который мы собираемся создать, показан на рис. 10.1. Это снимок *pdf*-файла, который был экспортирован из объекта `QTextDocument`, созданного в Linux на основе *HTML*-документа с помощью версии `Qt 4.5.2`.

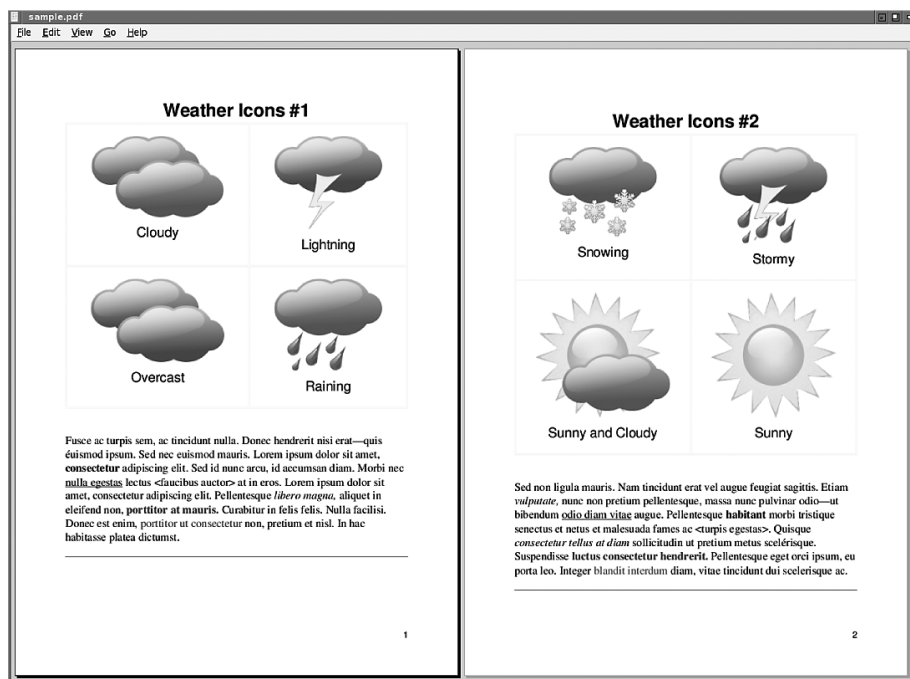


Рис. 10.1. Пример отображения двухстраничного документа, содержащего форматированный текст, в программе *evince*

Мы начнем эту главу со сравнения качества и размеров экспортированных файлов, представляющих наш пример в различных форматах. Затем, во втором разделе, мы покажем, как заполняется сам объект `QTextDocument`, – сначала путем импорта *HTML*, а потом с помощью `QTextCursor`. В третьем разделе мы экспортируем `QTextDocument` в различных форматах, а также научимся печатать такие документы. И в последнем разделе продемонстрируем создание точно такого же документа путем рисования и объясним, как экспортировать и распечатать результат.

Прежде чем переходить к содержательному обсуждению, дадим краткий обзор структуры данных, на основе которой получаются страницы документа, поскольку она является общей для всех рассматриваемых подходов.

```
struct OnePage
{
    QString title;
    QStringList filenames;
    QStringList captions;
    QString descriptionHtml;
};
```

Поле `title` содержит заголовок страницы в виде простого текста. В двух параллельных списках строк хранятся имена файлов изображений и подписи к ним. В строке `descriptionHtml` находится абзац, который расположен после таблицы и содержит HTML-разметку с различными выделениями (цветом, полужирным шрифтом и т. д.).

Данные документа хранятся в объекте класса `PageData`, в котором имеется закрытая переменная `pages` типа `QList<OnePage>`. Мы не будем обсуждать классы драйверов и даже опустим большую часть самого класса `PageData`; нас будут интересовать только методы, необходимые для заполнения `QTextDocument`, а также экспорта и печати документов.

Качество файла, в который экспортирован QTextDocument

В этом разделе мы сравним качество и размеры файлов, получаемых в результате экспорта (и печати) документа в различных стандартных форматах, поддерживаемых Qt.

Цифры на рис. 10.3 призваны дать некоторое представление о том, чего можно ожидать от каждого подхода¹. Но ввиду различий между форматами, платформами, версиями Qt и самими документами мы призываем относиться к представленным результатам скептически и проводить тестирование самостоятельно. Все результаты были получены с помощью программы `Output Sampler` (`outputsampler`), показанной на рис. 10.2. Вы без труда сможете адаптировать ее к тестированию собственных документов.

На рис. 10.3 сравниваются *pdf*-файлы, полученные в результате экспорта с помощью `QPrinter` и путем вывода в файл. *pdf*-файл можно создать,

¹ Количество звездочек, обозначающих качество, отражает субъективное мнение автора и различно для разных документов. Размеры в байтах также меняются – они включены, только чтобы дать представление об *относительных* размерах.

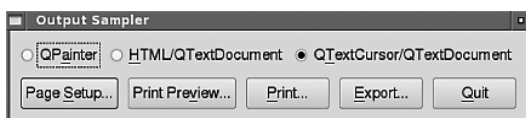


Рис. 10.2. Программа Output Sampler

распечатав документ с помощью диалогового окна печати и выбрав режим print to file (печать в файл), – обычно при этом выводится файл, который может быть чуть лучше или чуть хуже получающегося в результате экспорта¹. Еще стоит отметить, что QTextDocument предоставляет номера страниц в правом нижнем углу, хотим мы того или нет, и никакого способа управлять форматированием номеров страниц не существует. Если же документ выводится с помощью QPainter, тогда мы рисуем все сами.

Приведенные на рис. 10.3 размеры файлов показывают, что при использовании QPainter pdf-файлы, содержащие SVG-изображения, получаются очень большими. Если бы мы вместо этого выводили растровые изображения (этого можно добиться, убрав из pro-файла определение макроса EMBED_SVG), то выведенные QPainter pdf-файлы уменьшились бы примерно на 100 Кбайт.

Платформа/ Версия Qt	Метод вывода PDF (Качество, Байты)		
	QTextDocument/ HTML	QTextDocument/ QTextCursor	QPainter
Linux/4.5.2	★★★★★ 136 251	★★★★★ 136 216	★★★★☆ 5 956 141
Linux/4.6.1	★★★★★ 143 985	★★★★★ 136 302	★★★☆☆ 6 669 547
Mac/4.5.2	★★★☆☆ 135 407	★★★★☆ 135 316	★★★★☆ 5 991 845
Mac/4.6.1	★★★★☆ 135 890	★★★★★ 135 481	★★★☆☆ 6 687 444
Windows/4.5.2	★★★★★ 147 838	★★★★★ 150 680	★★★★★ 5 998 190
Windows/4.6.1	★★★★★ 148 045	★★★★★ 150 881	★★★★☆ 6 689 564

Рис. 10.3. Сравнение качества и размера результирующего pdf-файла

Программа outputsampler умеет также экспортировать документ в формате PostScript (.ps). Наши тесты – с использованием Qt 4.6.1 в Linux – показали, что размер получаемого файла существенно зависит от того,

¹ В Windows в режиме печати в файл создается pdf-файл только при условии, что установлен драйвер принтера для вывода в PDF. Если его нет, то единственный способ создания документа в формате PDF в Windows – воспользоваться экспортом.

применялся ли для этого QTextDocument (~7 Мбайт с внедренными SVG-изображениями и ~170 Кбайт с внедренными PNG-изображениями) или QPainter (~16 Мбайт вне зависимости от формата внедренных изображений). Кроме того, мы обнаружили, что полученные из исходного документа PostScript-файлы невозможно просмотреть в программе *evince* (projects.gnome.org/evince), зато с ними отлично работает не столь развитая программа просмотра *gv* (www.gnu.org/software/gv). Впрочем, таблицу сравнения мы приводить не стали, потому что в большинстве ситуаций формат PDF уже вытеснил PostScript.

Платформа/ Версия Qt	Метод вывода ODT (Качество, Байты)	
	QTextDocument/ HTML	QTextDocument/ QTextCursor
Linux/4.5.2	★★☆☆☆ 135 432	☆☆☆☆☆ <i>invalid</i>
Linux/4.6.1	★★☆☆☆ 135 437	★★☆☆☆ 135 438
Mac/4.5.2	★★☆☆☆ 135 441	☆☆☆☆☆ <i>invalid</i>
Mac/4.6.1	★★☆☆☆ 135 444	★★☆☆☆ 135 443
Windows/4.5.2	★★☆☆☆ 135 407	☆☆☆☆☆ <i>invalid</i>
Windows/4.6.1	★★☆☆☆ 135 413	★★☆☆☆ 135 409

Рис. 10.4. Сравнение качества и размера результирующего файла в формате Open Document Format

На рис. 10.4 сравниваются результаты экспорта того же документа в формате Open Document Format (*.odt*). Их можно получить только с помощью класса QTextDocument; нарисовать посредством QPainter не получится. Мы протестировали экспортированные *odt*-файлы в OpenOffice.org 3 (www.openoffice.org)¹. На рисунке ясно видно, что поддержка этого формата в Qt пока весьма несовершенна, но постепенно улучшается.

Несмотря на слабые показатели в наших тестах, не следует считать, что библиотека Qt не способна создавать качественные *odt*-файлы. Документ, с которым мы работаем, хотя и короткий, но довольно сложный, и вполне может оказаться, что для других документов качество будет гораздо выше. И разумеется, в будущих версиях Qt поддержка формата ODT обязательно улучшится.

¹ Термин «Open Document Format» является обобщенным. Он подразумевает спецификации форматированного текста (*.odt*), электронных таблиц (*.ods*) и других документов. Использование суффикса *.odt* (Open Document Text) обязательно, иначе OpenOffice.org не сможет корректно интерпретировать содержимое файла.

На рис. 10.5 сравниваются результаты экспорта в формате HTML. Этот формат настолько распространен, что возможность вывода в нем результатов считается обязательной для многих приложений. Входящие в состав Qt классы для работы с XML, например `QXmlStreamWriter`, упрощают эту задачу, но часто желательно иметь единое представление документа в памяти (например, в виде `QTextDocument`) и выводить его в различных форматах, в том числе HTML. Как и в случае Open Document Format, результат в формате HTML невозможно создать с помощью `QPainter`.

Платформа/ Версия Qt	Метод вывода HTML (Качество, Байты)	
	QTextDocument/ HTML	QTextDocument/ QTextCursor
Linux/4.5.2	★★★★☆ 8 355	★★★★☆ 6 795
Linux/4.6.1	★★★★☆ 8 346	★★★★☆ 6 795
Mac/4.5.2	★★★★★ 8 363	★★★★★ 6 803
Mac/4.6.1	★★★★★ 8 363	★★★★★ 6 803
Windows/4.5.2	★★★★☆ 8 106	★★★★☆ 6 539
Windows/4.6.1	★★★★☆ 8 106	★★★★☆ 6 539

Рис. 10.5. Сравнение качества и размера результирующего *html*-файла

На платформе Mac OS X результат тестировался в браузере Safari – он прекрасно справился с задачей отображения HTML-страниц, в том числе содержащих внедренные SVG-изображения. В Linux и Windows использовался браузер Firefox, который вообще не показал SVG-изображения¹.

На рис. 10.6 сравниваются результаты экспорта в формат Scalable Vector Graphics (*.svg*) – мы выводили только первую страницу документа. Формат SVG обеспечивает то, что обещает его название, – масштабируемость без потери качества.

Документы можно вводить также в растровых форматах, например PNG (Portable Network Graphics) или BMP (Windows Bitmap). Но они плохо масштабируются – это внутреннее ограничение формата, не имеющее ничего общего с Qt, – поэтому такие форматы пригодны только

¹ Вероятно, проблема во встраивании изображений в HTML-код: Firefox прекрасно отображает *svg*-файлы, в том числе используемые в тестовом документе, по отдельности. Однако имейте в виду, что приложение Output Sampler не добавляет суффикс *.html* к имени файла автоматически – вы должны сделать это вручную в диалоговом окне экспорта. – *Прим. науч. ред.*

Платформа/ Версия Qt	Метод вывода SVG (Качество, Байты)		
	QTextDocument/ HTML	QTextDocument/ QTextCursor	QPainter
Linux/4.5.2	★★★★☆ 94 535	★★★★☆ 94 560	★★★★★ 94 156
Linux/4.6.1	★★★★☆ 91 756	★★★★☆ 91 436	★★★★☆ 102 061
Mac/4.5.2	★★★★☆ 91 203	★★★★☆ 91 195	★★★★☆ 94 124
Mac/4.6.1	★★★★☆ 92 905	★★★★☆ 91 804	★★★★☆ 99 360
Windows/4.5.2	★★★★☆ 92 066	★★★★☆ 92 087	★★★★★ 95 338
Windows/4.6.1	★★★★☆ 92 597	★★★★☆ 92 663	★★★★☆ 100 544

Рис. 10.6. Сравнение качества и размера результирующего svg-файла

для изображений, которые не предполагается масштабировать, зато в этом случае они идеальны. Мы не стали приводить таблицу с результатами сравнения, потому что качество получается отличным для всех платформ, версий Qt и методов вывода.

Создание объектов QTextDocument

Заполнить объект QTextDocument можно двумя способами: передать ему строку QString, содержащую HTML-разметку, или воспользоваться классом QTextCursor. В этом разделе мы продемонстрируем оба подхода, начав с HTML.

Конечно, иногда нам нужен не документ, а просто способ вывести данные в файл в каком-нибудь стандартном формате или распечатать их. В таких случаях – и при условии, что речь не идет о формате HTML или ODF, – мы можем вообще обойтись без QTextDocument и воспользоваться классом QPainter; эта тема рассматривается в последнем разделе данной главы (стр. 406).

Создание QTextDocument с помощью HTML

Заполнение документа QTextDocument с помощью HTML – очень удобный способ, если вы хорошо разбираетесь в языке HTML; из всех рассмотренных в этой главе он самый лаконичный.

В программе outputsampler мы сначала создаем пустой объект QTextDocument, а затем передаем указатель на него методу PageData::populateDocumentUsingHtml(), который вызывает два вспомогательных метода.

```
void PageData::populateDocumentUsingHtml(QTextDocument *document)
```

```

{
    QString html("<html>\n<body>\n");
    for (int page = 0; page < pages.count(); ++page) {
        html += pageAsHtml(page, false);
        if (page + 1 < pages.count())
            html += "<br style='page-break-after:always;' />\n";
    }
    html += "</body>\n</html>\n";
    document->setHtml(html);
}

```

Этот метод создает строку, содержащую HTML-разметку, а затем передает ее объекту `QTextDocument`, полученному от вызывающей программы. Сначала в строку записывается стандартный тег `<html>`, а затем тег `<body>` (тег `<head>` опускается). Далее для каждой страницы в списке объектов `OnePage` в строку добавляется ее HTML-разметка и для всех, кроме последней, еще и тег `
` со специфичным для Qt атрибутом `page-break-after`, который вызывает переход на другую страницу. В конце открытые теги закрываются.

Обратите внимание, что мы включили в HTML-разметку символы новой строки (`\n`); это необязательно, но иногда удобно для отладки, к тому же такой код проще читать человеку.

```

QString PageData::pageAsHtml(int page, bool selfContained)
{
    const OnePage &thePage = pages.at(page);
    QString html;
    if (selfContained)
        html += "<html>\n<body>\n";
    html += QString("<h1 align='center'>%1</h1>\n")
        .arg(Qt::escape(thePage.title));
    html += "<p>";
    html += itemsAsHtmlTable(thePage);
    html += "</p>\n";
    html += QString("<p style='font-size:15pt;font-family:times'>"
        "%1</p><hr>\n").arg(thePage.descriptionHtml);
    if (selfContained)
        html += "</body>\n</html>\n";
    return html;
}

```

Этот метод создает простую законченную HTML-страницу, если второй аргумент равен `true`; метод `populateDocumentUsingHtml()` всегда вызывает его со вторым аргументом `false`.

Сначала мы получаем ссылку на данные, которые предстоит конвертировать в HTML. Мы создаем тег `<h1>`, содержащий заголовок, не забыв воспользоваться функцией `Qt::escape()`, которая преобразует встречаю-

щиеся в заголовке символы `&`, `<` и `>` в HTML-кодированные эквиваленты `&`, `<` и `>`¹.

HTML-таблицы, применяемые для размещения изображений и подписей к ним, обрабатываются в отдельном методе, который будет рассмотрен ниже. Текст `descriptionHtml` добавляется в конец без экранирования – потому что он уже представлен в формате HTML, – но ему предшествует тег абзаца с атрибутом `style`, описывающим гарнитуру и кегль шрифта. Кроме того, после текста добавляется горизонтальная черта.

Номер страницы, который виден на снимке экрана на рис. 10.1 (стр. 390), добавляет Qt, мы им управлять не можем.

```
QString PageData::itemsAsHtmlTable(const OnePage &thePage)
{
    QString html("<table border='1' cellpadding='20' width='100%'>");
    for (int i = 0; i < thePage.files.count(); ++i) {
        if (i % 2 == 0)
            html += "<tr>\n";
        html += QString("<td align='center'><img src='%1' />"
            "<p style='font-size:18pt'>%2</p></td>\n")
            .arg(thePage.files.at(i))
            .arg(Qt::escape(thePage.captions.at(i)));
        if (i % 2 != 0)
            html += "</tr>\n";
    }
    if (!html.endsWith("</tr>\n"))
        html += "</tr>\n";
    html += "</table>\n";
    return html;
}
```

Этот метод создает HTML-таблицу с четырьмя ячейками (2×2). В каждой ячейке находятся тег ``, в атрибуте `src` которого указывается имя файла изображения, и тег `<p>` с увеличенным кеглем шрифта для показа HTML-кодированной подписи. (По умолчанию в программе `outputsampler` подразумеваются изображения в формате SVG, но можно заставить ее использовать формат PNG, если закомментировать определение макроса `EMBED_SVG` в *pro*-файле.)

Этих трех методов достаточно для создания длинной строки `QString`, содержащей документ в формате HTML. Для заполнения объекта `QTextDocument` этой HTML-разметкой достаточно вызвать всего один метод: `QTextDocument::setHtml()`. И хотя в этом примере мы не использовали ка-

¹ В версии Qt 4.5 функция `Qt::escape()` не годится для экранирования текста в значениях HTML-атрибутов, поскольку не экранирует кавычки. Начиная с версии Qt 4.6, `Qt::escape()` экранирует двойные кавычки.

скадные таблицы стилей (CSS), Qt их поддерживает; см. qt.nokia.com/doc/richtext-html-subset.html.

Создание QTextDocument с помощью QTextCursor

Класс `QTextCursor` позволяет перемещаться по документу `QTextDocument` и редактировать его, ничего не зная об HTML. Мы ознакомились с этим классом в предыдущей главе, где с его помощью редактировали документы, а в этом подразделе применим его для создания документа `QTextDocument` с нуля. API класса `QTextCursor` был описан в предыдущей главе (стр. 356–358). Метод `QTextCursor::movePosition()` также обсуждался в предыдущей главе вместе с перечислением `QTextCursor::MoveOperation` (стр. 362).

В классе `QTextCursor` есть одна необычная особенность – два метода для вставки списков: `createList()` и `insertList()`. Первый создает и вставляет в документ список, делая текущий абзац его первым элементом. Второй тоже создает и вставляет в документ список, но первым элементом делается новый абзац.

Код заполнения `QTextDocument` с помощью `QTextCursor` структурно очень напоминает код создания документа из HTML-разметки, но вспомогательных методов требуется больше. Как и в предыдущей версии, мы начинаем с создания пустого объекта `QTextDocument` и передаем указатель на него методу, который заполняет документ.

```
void PageData::populateDocumentUsingQTextCursor(
    QTextDocument *document)
{
    document->setDefaultFont(QFont("Times", 15));
    QTextCursor cursor(document);
    for (int page = 0; page < pages.count(); ++page) {
        addPageToDocument(&cursor, page);
        if (page + 1 < pages.count()) {
            QTextBlockFormat blockFormat;
            blockFormat.setPageBreakPolicy(
                QTextFormat::PageBreak_AlwaysAfter);
            cursor.mergeBlockFormat(blockFormat);
        }
    }
}
```

Сначала мы задаем для документа шрифт по умолчанию – чуть больше, чем обычно, для того чтобы документ хоть как-то читался на снимке экрана на рис. 10.1 (стр. 390).

Затем мы создаем курсор `QTextCursor` для `QTextDocument` и будем передавать вспомогательным методам именно курсор, а не сам документ. Далее, как и раньше, по очереди добавляем в документ все страницы.

После каждой страницы, кроме последней, мы вставляем разрыв страницы. Делается это путем создания нового объекта `QTextBlockFormat` (формат абзаца) и задания для него политики разрыва страниц `PageBreak_AlwaysAfter`; есть еще две политики: `PageBreak_Auto`, которая поручает Qt самостоятельно выбирать места разрыва страниц, и `PageBreak_AlwaysBefore`, вставляющая разрыв страницы перед абзацем, для которого задан этот формат. Создав объект форматирования, мы *объединяем* его с текущим форматом блока, на который указывает курсор (то есть последнего вставленного с помощью курсора абзаца). Метод `QTextCursor::mergeBlockFormat()` замещает параметры в текущем формате блока значениями одноименных параметров из переданного ему формата при условии, что эти значения отличаются от подразумеваемых по умолчанию. Существует аналогичный метод `QTextCursor::mergeCharFormat()` для объединения форматов символов.

```
void PageData::addPageToDocument(QTextCursor *cursor, int page)
{
    const OnePage &thePage = pages.at(page);
    addTitleToDocument(cursor, thePage.title);
    addItemsToDocument(cursor, thePage);
    cursor->insertHtml(thePage.descriptionHtml);
    addRuleToDocument(cursor);
}
```

Этот метод препоручает большую часть своей работы вспомогательным методам. Исключение составляет вызов метода `QTextCursor::insertHtml()`, который вставляет произвольную HTML-разметку в позицию курсора в документе `QTextDocument`.

```
void PageData::addTitleToDocument(QTextCursor *cursor, const QString &title)
{
    QTextBlockFormat blockFormat;
    blockFormat.setAlignment(Qt::AlignHCenter);
    blockFormat.setTopMargin(0);
    cursor->insertBlock(blockFormat);
    QTextCharFormat charFormat;
    charFormat.setFont(QFont("Helvetica", 24, QFont::Bold));
    cursor->insertText(title, charFormat);
}
```

Мы хотим, чтобы заголовок документа был выровнен по центру и отображался крупным шрифтом. Кроме того, заголовок должен располагаться в начале страницы (сразу под верхним полем). Для этого мы создаем новый формат блока, содержащий параметры форматирования абзаца – выравнивание и верхнее поле, – и вставляем в документ новый блок (то есть пустой абзац с заданным форматированием). Затем мы создаем формат символов, в котором задан нужный нам крупный шрифт, и вставляем в текущий абзац текст заголовка, указывая для него только что созданный формат.

Имеется еще метод `QTextCursor::insertText()`, который принимает только аргумент типа `QString` и применяет к этой строке уже действующее форматирование, а также перегруженный `QTextCursor::insertBlock()`, который принимает одновременно формат блока и формат символов. Мы почти всегда предпочитаем использовать вариант метода `insertBlock()`, принимающий только формат блока, и вставлять текст с помощью метода `insertText()` с двумя аргументами, как показано выше.

```
void PageData::addItemToDocument(QTextCursor *cursor,
                                const OnePage &thePage)
{
    QTextDocument tableDocument;
    QTextCursor tableCursor(&tableDocument);
    QTextTable *table = tableCursor.insertTable(2, 2, tableFormat());
    for (int i = 0; i < thePage.fileNames.count(); ++i)
        populateTableCell(table->cellAt(i / 2, i % 2), thePage, i);
    cursor->insertFragment(QTextDocumentFragment(&tableDocument));
}
```

Программное добавление таблицы в `QTextDocument` с помощью `QTextCursor` требует чуть больше кода, хотя все шаги несложны.

Если попытаться вставить `QTextTable` в `QTextDocument` непосредственно, применяя метод `QTextCursor::insertTable()`, то очень легко запутаться! Дело в том, что, заполнив таблицу, мы должны не забыть переместить курсор в позицию сразу за таблицей, а только потом продолжить заполнение документа. Объясняется это тем, что после вставки последнего элемента в последнюю ячейку таблицы курсор позиционируется за этим элементом – но *внутри* последней ячейки.

К счастью, у этой проблемы есть элегантное общее решение: мы можем создать отдельный объект `QTextDocument`, который содержит только таблицу, а затем вставить его как фрагмент документа в заполняемый документ. Тогда курсор окажется позиционирован за фрагментом (то есть сразу после таблицы), и мы сможем продолжить заполнение документа, не думая о том, как вывести курсор за пределы таблицы.

Таким образом, в этом методе мы создаем новый объект `QTextDocument`, `tableDocument`, и новый курсор, с помощью которого этот объект заполняется. Сначала мы вызываем метод `QTextCursor::insertTable()`, передавая ему число строк и столбцов и объект форматирования `QTextTableFormat`. (Метод `tableFormat()` будет рассмотрен ниже.)

Получив таблицу `QTextTable`, мы заполняем все ее ячейки `QTextTableCell` – извлекая их методом `QTextTable::cellAt()` – с помощью вспомогательного метода.

В заключение мы вставляем содержащий таблицу документ в тот документ, который заполняем, в виде объекта `QTextDocumentFragment`. В этот момент курсор позиционируется (как всегда после вставки) сразу вслед

за вставленным элементом, то есть именно там, где нам нужно, – после таблицы.

```
QTextTableFormat PageData::tableFormat()
{
    QTextTableFormat tableFormat;
    tableFormat.setAlignment(Qt::AlignCenter);
    tableFormat.setCellPadding(10);
    tableFormat.setTopMargin(10);
    tableFormat.setBottomMargin(10);
    QVector<QTextLength> widths;
    widths << QTextLength(QTextLength::PercentageLength, 50)
           << QTextLength(QTextLength::PercentageLength, 50);
    tableFormat.setColumnWidthConstraints(widths);
    return tableFormat;
}
```

В этом методе мы создаем объект форматирования `QTextTableFormat` и задаем в нем выравнивание, отступ и два поля. (Левое и правое поля оставляем заданными по умолчанию.) Ширину столбцов мы задаем не явно, а отводим каждому по 50% общей ширины таблицы. При этом Qt самостоятельно произведет все вычисления и сделает столбцы одинаковыми по ширине.

Первый параметр конструктора класса `QTextLength` может принимать еще два значения: `VariableLength` и `FixedLength`; и, конечно, разрешается использовать любую комбинацию ширины столбцов. Значением фиксированной длины `FixedLength` является число с плавающей точкой (типа `qreal`), равное количеству пикселей. Чтобы создать объект `QTextLength`, представляющий переменную длину, следует воспользоваться конструктором по умолчанию (без аргументов).

```
void PageData::populateTableCell(QTextTableCell tableCell,
                                const OnePage &thePage, int index)
{
    QTextBlockFormat blockFormat;
    blockFormat.setAlignment(Qt::AlignHCenter);
    QTextCursor cursor = tableCell.firstCursorPosition();
    cursor.insertBlock(blockFormat);
    cursor.insertImage(thePage.fileNames.at(index));
    blockFormat.setTopMargin(30);
    cursor.insertBlock(blockFormat);
    QTextCharFormat charFormat;
    charFormat.setFont(QFont("Helvetica", 18));
    cursor.insertText(thePage.captions.at(index), charFormat);
}
```

Этот метод вставляет изображение и подпись в одну ячейку `QTextTableCell`. Поскольку и то и другое должно быть центрировано по горизонта-

ли, мы сначала создаем формат блока и задаем в нем нужное выравнивание. Затем получаем для ячейки курсор от метода `QTextTableCell::firstCursorPosition()` и с его помощью вставляем пустой абзац с только что созданным форматом, после чего вставляем в него изображение, обращаясь к методу `QTextCursor::insertImage()`.

У метода `QTextCursor::insertImage()` есть несколько перегруженных вариантов; мы воспользовались тем, что принимает имя файла, но есть и другой – принимающий `QImage`.

Чтобы оставить место между нижним краем изображения и верхним краем подписи, мы задаем в формате верхнее поле – эта установка будет действовать только при последующем использовании формата. Затем мы вставляем пустой абзац с тем же самым форматом и далее создаем формат символов, устанавливаем в нем крупный шрифт и вставляем подпись с этим форматом.

```
void PageData::addRuleToDocument(QTextCursor *cursor)
{
    QTextBlockFormat blockFormat;
    blockFormat.setProperty(
        QTextFormat::BlockTrailingHorizontalRulerWidth, 1);
    cursor->insertBlock(blockFormat);
}
```

В конце каждой страницы мы хотим нарисовать горизонтальную черту. Для этого нужно просто создать формат блока, установить в нем свойство `BlockTrailingHorizontalRulerWidth` равным 1 и вставить блок. (На самом деле, как показали тесты, для появления черты можно установить для этого свойства любое значение.)

Класс `QTextFormat` (базовый для класса `QTextBlockFormat`) поддерживает свыше семидесяти свойств. Многие из них доступны с помощью специализированных для каждого свойства методов, однако наличие метода `QTextFormat::setProperty()` и различных вариаций методов чтения (например, `QTextFormat::intProperty()`) означает, что количество свойств можно в любой момент увеличить (даже в дополнительных (minor) и исправляющих ошибки выпусках), не нарушая двоичную совместимость.

Экспорт и печать документов

Документ `QTextDocument` можно вывести в различных стандартных форматах, в том числе Open Document Format и HTML, а также в векторных форматах (PDF, PostScript, или SVG) и в любом растровом формате, поддерживаемом Qt (например, PNG или BMP). Документы можно также распечатывать. Все эти возможности будут рассмотрены в данном разделе.

Экспорт QTextDocument

В этом подразделе мы покажем, как экспортировать содержимое объекта `QTextDocument` с сохранением форматирования, в том числе атрибутов текста и внедренных изображений. Для графических форматов (SVG и растровых) мы будем экспортировать образ одной страницы, а для всех остальных – весь многостраничный документ.

В любом случае мы начинаем с создания пустого объекта `QTextDocument`. Для экспорта одной страницы в формате SVG или растровом мы явно задаем размер страницы и поля. В документации по Qt единицы измерения обычно не оговариваются, но в общем случае для печати подразумеваются пиксели, хотя в типографской сфере принято измерять линейные размеры в пунктах (1/72 дюйма).

```
QTextDocument document;  
document.setPageSize(printer.pageRect().size());  
document.setDocumentMargin(25);
```

Для одностраничных документов мы задаем шрифт по умолчанию и создаем курсор `QTextCursor`, передавая конструктору последнего объект `QTextDocument` в качестве аргумента. Затем вызываем уже рассмотренный выше метод `PageData::addPageToDocument()` (стр. 399), который добавляет в документ одну страницу.

Для многостраничных документов мы вызываем метод `populateDocument()`, который в качестве аргумента принимает указатель на пустой объект `QTextDocument`. Мы уже встречались с двумя вариантами этого метода: `PageData::populateDocumentUsingHtml()` и `PageData::populateDocumentUsingQTextCursor()` (стр. 398). Теоретически между ними не должно быть различия, но на практике мы обнаружили, что результаты обычно отличаются, поэтому рекомендуем самостоятельно провести тесты для интересующих вас документов, платформ и форматов вывода.

Экспорт в формате PDF и PostScript

Экспортировать документ в формате PDF или PostScript очень просто, достаточно передать имя файла и объект `QTextDocument`. В показанном ниже методе предполагается, что имя файла `filename` заканчивается суффиксом `.pdf` или `.ps`.

```
bool MainWindow::exportPdfOrPs(const QString &filename,  
                               QTextDocument *document)  
{  
    Q_ASSERT(filename.endsWith(".ps") || filename.endsWith(".pdf"));  
    QPrinter printer(QPrinter::HighResolution);  
    printer.setOutputFileName(filename);  
    printer.setOutputFormat(filename.endsWith(".pdf")  
        ? QPrinter::PdfFormat : QPrinter::PostScriptFormat);
```

```
document->print(&printer);  
return true;  
}
```

Чтобы экспортировать файл в формате PDF или PostScript, мы применяем объект `QPrinter`, но вместо распечатки страниц на бумаге настраиваем его для печати в файл¹. (Можно также нарисовать *pdf*- и PostScript-документ с помощью `QPainter`, как будет показано ниже.)

В большинстве приложений объект `QPrinter` хранится в закрытой переменной-члене в классе главного окна. Достоинство такого решения в том, что пользователь приложения может один раз сконфигурировать объект-принтер, и при каждом последующем применении будут действовать настройки, заданные в последний раз. Но в программе `output-sampler` мы предпочли создавать новый объект `QPrinter` для документов, которые выводятся без вмешательства пользователя (в частности, для экспорта тестового документа).

Если не считать несложного конфигурирования `QPrinter`, то собственно экспорт сводится к вызову метода `QTextDocument::print()` — ему передается указатель на объект `QPrinter`, то есть устройство, на котором мы хотим печатать. По умолчанию метод `print()` разбивает документ на страницы, используя в качестве области печати прямоугольник, возвращаемый методом `QPrinter::paperRect()`, с дополнительными полями по 20 мм.

Экспорт в формате Open Document Format

Экспортировать `QTextDocument` в формате Open Document Format еще проще, чем в PDF или PostScript, — благодаря классу `QTextDocumentWriter`.

```
bool MainWindow::exportOdf(const QString &filename, QTextDocument *document)  
{  
    Q_ASSERT(filename.endsWith(".odt"));  
    QTextDocumentWriter writer(filename);  
    return writer.write(document);  
}
```

Мы снова передаем имя файла (на этот раз с суффиксом *.odt*) и указатель на `QTextDocument`. Метод `QTextDocumentWriter::write()` возвращает булевый признак успеха или неудачи, который мы передаем вызывающей программе. Класс `QTextDocumentWriter` можно использовать и для экспорта в других форматах, нужно лишь предварительно вызвать метод `QText`

¹ На платформе Mac OS X при использовании формата `QPrinter::NativeFormat` получится *pdf*-файл, сгенерированный движком двумерного рисования Quartz компании Apple. В нашем примере размер файла оказался вдвое больше при очень незначительном улучшении качества.

`DocumentWriter::setFormat()`, передав ему строку `plaintext` или `html` в виде массива `QByteArray`. По умолчанию предполагается формат “`odf`”, но, как мы уже отмечали выше, `OpenOffice.org` правильно распознает тип файла, только если его имя заканчивается суффиксом `.odt`. Полный список поддерживаемых форматов (который всегда включает три вышеуказанных) можно получить от метода `QTextDocumentWriter::supportedFormats()`.

К сожалению, как следует из рис. 10.4 (стр. 393), поддержка формата `Open Document Format` в `Qt` пока оставляет желать лучшего. Наши эксперименты показали, что результат вывода в этом формате очень сильно зависит от содержимого документа, а иногда даже от того, в каком порядке документ заполнялся. Поэтому мы рекомендуем вам тестировать качество вывода своих документов на целевых платформах и с предполагаемыми версиями `Qt`, чтобы гарантировать приемлемость результатов.

Экспорт в формате HTML

Для экспорта в формате `HTML` мы должны создать строку, содержащую `HTML`-разметку, а затем вывести ее в файл. Это стандартный для `C++/Qt` код, хотя в нем есть одно тонкое место, на которое мы укажем.

```
bool MainWindow::exportHtml(const QString &filename,
                             QTextDocument *document)
{
    Q_ASSERT(filename.endsWith(".htm") || filename.endsWith(".html"));
    QFile file(filename);
    if (!file.open(QIODevice::WriteOnly|QIODevice::Text)) {
        AQP::warning(this, tr("Error"),
                     tr("Failed to export %1: %2").arg(filename)
                     .arg(file.errorString()));
        return false;
    }
    QTextStream out(&file);
    out.setCodec("utf-8");
    out << document->toHtml("utf-8");
    file.close();
    return true;
}
```

Метод `QTextStream::setCodec()` весьма либерально относится к именам кодировок. Здесь, например, можно было бы указать и имя “`utf8`”. Однако метод `QTextDocument::toHtml()` требует указывать имя кодировки (если оно вообще указывается) в соответствии с требованиями консорциума `W3C (World Wide Web Consortium)`¹. Если кодировка не задана,

¹ О кодировках XML см. www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl и www.iana.org/assignments/charactersets.

то в составе метаданных HTML не будет атрибута `charset`; мы рекомендуем для *html*-файлов всегда использовать кодировку UTF-8.

Если требуется просто создавать *html*-файлы, то можно делать это напрямую в коде с использованием `QString` (так мы и поступали ранее при заполнении объекта `QTextDocument` из HTML-разметки) или воспользоваться классом `QXmlStreamWriter`, который гарантирует правильное экранирование атрибутов и текста (делая это автоматически).

Экспорт в формате SVG

Для демонстрации экспорта в формате SVG мы решили немного упростить задачу и заполнить объект `QTextDocument` всего одной страницей. Код экспортирования не сложен, но в нем есть несколько важных деталей. Отметим, что для поддержки SVG в *pro*-файл проекта необходимо добавить строку `QT += svg`.

```
bool MainWindow::exportSvg(const QString &filename, QTextDocument *document)
{
    Q_ASSERT(filename.endsWith(".svg"));
    QSvgGenerator svg;
    svg.setFileName(filename);
    QRect rect = printer.pageRect().adjusted(25, 25, -25, 25);
    svg.setSize(rect.size());
    QPainter painter(&svg);
    painter.setViewport(rect);
    document->drawContents(&painter);
    return true;
}
```

SVG-изображение создается путем рисования в объекте `QSvgGenerator`. В данном случае мы использовали переменную-член типа `QPrinter` и подстроили размер SVG-изображения под размер страницы принтера, оставив небольшие поля. Сконфигурировав SVG-генератор, мы создаем объект `QPainter`, который будет на нем рисовать, и в качестве порта просмотра рисовальщика (то есть той области, в пределах которой будет происходить рисование) задаем прямоугольник SVG-изображения. Метод `QTextDocument::drawContents()` аналогичен методу `QTextDocument::print()` с тем отличием, что выводит данные в объект-рисовальщик, а не на принтер, и принимает в качестве дополнительного аргумента отсекающий прямоугольник.

Здесь мы рисуем SVG-изображение с помощью вспомогательного метода из класса `QTextDocument`. Однако с тем же успехом мы могли бы создавать SVG-изображения, пользуясь API класса `QPainter` для рисования фигур, изображений и текста, поскольку SVG-генератор – такое же устройство рисования, как и любое другое. Мы убедимся в этом, когда будем рассматривать метод `paintSvg()` (стр. 419).

Экспорт в растровых форматах

Qt изначально поддерживает множество растровых графических форматов. Как минимум, это BMP (Windows Bitmap), JPG и JPEG (Joint Photographic Experts Group), PNG (Portable Network Graphics), PPM (Portable Pixmap), TIFF (Tagged Image File Format) и XPM (X11 Pixmap), но могут быть и другие. Точный перечень возвращает метод `QImageWriter::supportedImageFormats()`. (Отметим, что обычно Qt может выводить данные не во всех форматах, которые умеет читать; список поддерживаемых форматов ввода возвращает метод `QImageReader::supportedImageFormats()`. Кроме того, перечень форматов может быть расширен за счет подключаемых модулей.)

```
bool MainWindow::exportImage(const QString &filename,
                             QTextDocument *document)
{
    QImage image(printer.paperRect().size(), QImage::Format_ARGB32);
    QPainter painter(&image);
    painter.setRenderHints(QPainter::Antialiasing|
                           QPainter::TextAntialiasing);
    painter.fillRect(painter.viewport(), Qt::white);
    painter.setViewport(printer.paperRect());
    document->drawContents(&painter);
    return image.save(filename);
}
```

Этот метод экспортирует указанный `QTextDocument` в указанный файл в формате, определяемом по суффиксу имени файла, при условии, что этот суффикс присутствует в перечне поддерживаемых форматов вывода графики.

Структурно код очень похож на код экспорта SVG-изображения, только вначале создается не `QSvgGenerator`, а `QImage`. Кроме того, код практически идентичен коду для вывода растрового изображения в файл, в чем мы убедимся, когда будем рассматривать метод `paintImage()` ниже (стр. 419).

Мы создаем объект-рисовальщик `painter`, который будет рисовать изображение, задаем режим сглаживания и белый цвет фона. (Можно было бы вместо этого задать прозрачный фон `Qt::transparent`.) Обычно сглаживание задается для генерации растровых и предназначенных для отображения на экране изображений (например, BMP или PNG), но не для изображений в векторном формате или предназначенных для печати (например, SVG). Порт просмотра рисовальщика устанавливается в соответствии с форматом листа бумаги в принтере (на этот раз без полей), после чего рисуется содержимое документа. А в заключение вызывается метод `QImage::save()` (который возвращает булевый признак успеха), чтобы сохранить изображение в файле в запрошенном формате.

Печать и предварительный просмотр документа QTextDocument

Печать документов QTextDocument очень похожа на экспорт, но обычно мы предоставляем пользователю возможность настроить принтер и, возможно, предварительно просмотреть страницу перед печатью.

В программе `outputsampler` мы решили сохранить указатель на диалоговое окно `QPageSetupDialog` в переменной-члене класса главного окна и создавать упомянутое окно только при необходимости. Тем самым мы предоставляем пользователю простые средства для настройки параметров страницы, но практически не потребляем память, если диалоговое окно остается невостребованным.

```
void MainWindow::pageSetup()
{
    if (!pageSetupDialog)
        pageSetupDialog = new QPageSetupDialog(&printer, this);
    pageSetupDialog->open();
}
```

Если бы мы не стали реализовывать предварительный просмотр, то могли бы передать параметры при вызове метода `QPageSetupDialog::open()`, например `pageSetupDialog->open(this, SLOT(print()))`; в результате после подтверждения пользователем заданных в диалоговом окне параметров был бы вызван слот `print()`. Мы используем метод `open()`, а не `exec()`, чтобы на платформе `Mac OS X` диалоговое окно выглядело как отрывной листок, а на других платформах открывалось бы в виде обычного модального окна. (Мы уже затрагивали эту тему ранее в главе 2, стр. 77, и главе 3, стр. 119.)

Для предварительного просмотра мы решили создавать объект `QPrintPreviewDialog` по запросу — каждый раз заново. Это необходимо, так как диалоговое окно предварительного просмотра, похоже, кэширует результаты. В программе `outputsampler` это неудобно, так как способ генерации страницы может измениться в зависимости от выбранного переключателя, поэтому мы должны каждый раз генерировать предварительный вид страницы заново. В большинстве приложений этого не требуется.

```
void MainWindow::printPreview()
{
    if (printPreviewDialog)
        delete printPreviewDialog;
    printPreviewDialog = new QPrintPreviewDialog(&printer, this);
    QSize size = qApp->desktop()->availableGeometry().size();
    size.rwidth() /= 2;
    printPreviewDialog->resize(size);
    if (painterRadioButton->isChecked())
```

```

        connect(printPreviewDialog,
                SIGNAL(paintRequested(QPrinter*)),
                &pageData, SLOT(paintPages(QPrinter*)));
    else
        connect(printPreviewDialog,
                SIGNAL(paintRequested(QPrinter*)),
                this, SLOT(printDocument(QPrinter*)));
    printPreviewDialog->open();
}

```

Мы отводим под диалоговое окно всю высоту экрана, чтобы пользователь видел как можно больше. Объект `QDesktopWidget`, возвращаемый методом `QApplication::desktop()`, может сообщить истинные геометрические размеры экрана (`screenGeometry()`) или, как здесь, истинные размеры за вычетом панели задач и т. п. На платформе Mac OS X исключаются также строка меню и док. Оба метода принимают в качестве необязательного аргумента номер экрана для систем с несколькими мониторами, а метод `QDesktopWidget::screenCount()` возвращает количество имеющихся экранов. Поскольку у многих современных мониторов ширина гораздо больше высоты, мы устанавливаем ширину диалогового окна равной половине ширины экрана.

Чтобы предварительный просмотр заработал, необходимо установить соединение с сигналом `paintRequested()` от диалогового окна предварительного просмотра. В программе `outputsampler` мы предложили две возможности, хотя обычно достаточно одной. Второе соединение установлено со слотом `printDocument()`, который мы рассмотрим прямо сейчас, тогда как слот `PageData::paintPages()` будет обсуждаться в следующем разделе.

Как и в случае `QPageSetupDialog::open()`, методу `QPrintPreviewDialog::open()` следует передать объект и его слот, который будет вызван, если пользователь нажмет кнопку ОК.

На рис. 10.7 показаны результаты тестирования предварительного просмотра. К сожалению, на платформе Mac OS X были проблемы: предварительный просмотр объектов `QTextDocument` вообще не работал, а при предварительном просмотре нарисованных документов область печати на странице оказывалась слишком узкой.

```

void MainWindow::printDocument(QPrinter *printer)
{
    QTextDocument document;
    populateDocument(&document);
    document.print(printer);
}

```

Этот метод создает пустой объект `QTextDocument`, заполняет его посредством одного из методов `PageData::populateDocumentUsingHtml()` или `Page-`

Платформа/ Версия Qt	Метод генерации предварительного просмотра страницы		
	QTextDocument/ HTML	QTextDocument/ QTextCursor	QPainter
Linux/4.5.2	✓	✓	✓
Linux/4.6.1	✓	✓	✓
Mac/4.5.2	✗	✗	✓
Mac/4.6.1	✗	✗	✓
Windows/4.5.2	✓	✓	✓
Windows/4.6.1	✓	✓	✓

Рис. 10.7. Сравнение результатов предварительного просмотра

`Data::populateDocumentUsingQTextCursor()` (стр. 399), а затем иницирует печать документа на указанном принтере.

```
void MainWindow::populateDocument(QTextDocument *document)
{
    Q_ASSERT(!painterRadioButton->isChecked());
    if (htmlRadioButton->isChecked())
        pageData.populateDocumentUsingHtml(document);
    else if (cursorRadioButton->isChecked())
        pageData.populateDocumentUsingQTextCursor(document);
}
```

Этот метод мы привели просто для полноты. Он выбирает метод заполнения документа в зависимости от решения пользователя.

Рисование страниц

Чтобы нарисовать одностраничный документ (например, файл в формате SVG или в растровом формате), мы создаем объект `QPainter` и обращаемся к методу `PageData::paintPage()`. Для многостраничных документов мы делаем почти то же самое, только на этот раз используем метод `PageData::paintPages()`. В этом разделе мы рассмотрим методы класса `PageData` для рисования страниц, а в его подразделах покажем, как методы `paintPage()` и `paintPages()` применяются для рисования в *pdf*, *PostScript*, *svg*- и растровых файлах. Мы уже видели выше, что из диалогового окна предварительного просмотра вызывается метод `PageData::paintPages()`. Когда речь идет о выводе с помощью `QPainter`, различие между экспортом и рисованием стирается.

При использовании `QPainter` ответственность за все вычисления ложится на нас, а не на, скажем, объект `QTextDocument`, но взамен мы получаем полный контроль над тем, что и где рисуется.

```
void PageData::paintPages(QPrinter *printer, bool noUserInteraction)
{
    if (noUserInteraction)
        printer->setPageMargins(25, 25, 25, 25, QPrinter::Millimeter);
    QPainter painter(printer);
    for (int page = 0; page < pages.count(); ++page) {
        paintPage(&painter, page);
        if (page + 1 < pages.count())
            printer->newPage();
    }
}
```

Структура этого метода в основном такая же, как у методов `PageData::populateDocumentUsingHtml()` и `PageData::populateDocumentUsingQTextCursor()` (стр. 398), рассмотренных выше.

Если страницы выводятся без взаимодействия с пользователем, то мы задаем поля сами, иначе берем параметры, заданные пользователем. Затем создаем объект `QPainter` и рисуем страницы по одной. Мы не стали включать режим сглаживания, потому что считаем вывод на принтер аналогом рисования в векторный графический файл (например, SVG). По завершении каждой страницы, кроме последней, вызывается метод `QPrinter::newPage()`, который выбрасывает отпечатанную страницу и загружает новую. Метод `newPage()` возвращает булевый признак успеха, но мы его игнорируем.

```
void PageData::paintPage(QPainter *painter, int page)
{
    const OnePage &thePage = pages.at(page);
    int y = paintTitle(painter, thePage.title);
    y = paintItems(painter, y, thePage);
    paintHtmlParagraph(painter, y, thePage.descriptionHtml);
    paintFooter(painter, tr("- %1 -").arg(page + 1));
}
```

И этот метод по структуре напоминает встречавшиеся ранее (например, `PageData::addPageToDocument()`, стр. 399), то есть поручает обработку каждой части страницы тому или иному вспомогательному методу. Но у этих методов есть важная особенность — каждый, кроме последнего, возвращает ординату того места на странице, до которого дошла печать. Это позволяет следующему методу продолжить печать, не затирая напечатанное ранее. (Метод `paintHtmlParagraph()` тоже возвращает ординату строки после абзаца, но мы ее игнорируем, поскольку метод `paintFooter()` сам вычисляет для себя ординату, исходя из высот порта просмотра и заголовка и не обращая внимания на то, что уже было напечатано.)

Если этот метод применяется для вывода страницы в растровое изображение, то предполагается, что вызывающая программа, создавшая объект-рисовальщик, включит режим сглаживания.

```
int PageData::paintTitle(QPainter *painter, const QString &title)
{
    painter->setFont(QFont("Helvetica", 24, QFont::Bold));
    QRect rect(0, 0, painter->viewport().width(),
               painter->fontMetrics().height());
    painter->drawText(rect, title, QTextOption(Qt::AlignCenter));
    return qRound(painter->fontMetrics().lineSpacing() * 1.5);
}
```

Этот метод печатает заголовок очень крупным шрифтом, центрируя его по горизонтали. Ордината верхней стороны прямоугольника равна 0, то есть эта сторона начинается там, где заканчивается верхнее поле. В этом месте рисовальщик располагает верхнюю границу шрифта. Иными словами, ордината нижнего края текста равна `y + painter->fontMetrics().height()`.

Нарисовав текст, мы возвращаем ординату, начиная с которой можно рисовать следующий элемент, оставив между ними небольшой промежуток. Новая ордината вычисляется как полторы высоты строки, то есть промежуток составит половину высоты строки.

```
int PageData::paintItems(QPainter *painter, int y, const OnePage &thePage)
{
    const int ItemHeight = painter->viewport().height() / 3;
    const int ItemWidth = painter->viewport().width() / 2;
    paintItem(painter, QRect(0, y, ItemWidth, ItemHeight),
               thePage.fileNames.at(0), thePage.captions.at(0));
    paintItem(painter, QRect(ItemWidth, y, ItemWidth, ItemHeight),
               thePage.fileNames.at(1), thePage.captions.at(1));
    y += ItemHeight;
    paintItem(painter, QRect(0, y, ItemWidth, ItemHeight),
               thePage.fileNames.at(2), thePage.captions.at(2));
    paintItem(painter, QRect(ItemWidth, y, ItemWidth, ItemHeight),
               thePage.fileNames.at(3), thePage.captions.at(3));
    return y + ItemHeight + painter->fontMetrics().height();
}
```

В программе `outputsampler` нам нужно нарисовать таблицу 2×2, в ячейках которой находятся ее элементы – изображения с подписями. Этот метод вычисляет прямоугольник, занимаемый каждым элементом, а собственно рисование поручает вспомогательному методу `paintItem()`.

Порт просмотра рисовальщика уже учитывает поля страницы, поэтому мы можем производить вычисления относительно порта просмотра. Здесь мы решили, что высота каждого элемента будет составлять одну треть высоты страницы, а ширина – половину ширины страницы.

В заключение мы возвращаем ординату, равную высоте двух элементов плюс `QPainter::fontMetrics().height()`, то есть высота символа в текущем шрифте. Можно было бы вместо последнего слагаемого взять высоту строки в текущем шрифте (`QFontMetrics::lineSpacing()`).

```
void PageData::paintItem(QPainter *painter, const QRect &rect,
                        const QString &filename, const QString &caption)
{
    painter->drawRect(rect);

    const int Margin = 20;
    painter->setFont(QFont("Helvetica", 18));
    const int LineHeight = painter->fontMetrics().lineSpacing();

    QRect imageRect(rect);
    imageRect.adjust(Margin, Margin, -Margin, -(Margin + LineHeight));
    QSvgRenderer svg(filename);
    QSize size(svg.defaultSize());
    size.scale(imageRect.size(), Qt::KeepAspectRatio);
    imageRect.setSize(size);
    const int Xoffset = (imageRect.width() - size.width()) / 2;
    imageRect.moveTo(imageRect.x() + Xoffset, imageRect.y());
    svg.render(painter, imageRect);
    int y = rect.y() + rect.height() - LineHeight;
    QRect captionRect(rect.x(), y, rect.width(), LineHeight);
    painter->drawText(captionRect, caption, QTextOption(Qt::AlignCenter));
}
```

Рисование каждого элемента включает три этапа: контурный прямоугольник, изображение и подпись. Нарисовать прямоугольник легко, потому что он передается в качестве аргумента. Мы хотим аккуратно вписать изображение в прямоугольник, чтобы оно не напоздало на подпись, поэтому сначала устанавливаем величину поля и шрифт подписи. Установив шрифт, мы можем воспользоваться его метриками, полученными от рисовальщика, для вычисления высоты строки текста.

Мы предпочитаем SVG-изображения, потому что они масштабируются без потери качества. Сначала создаем прямоугольник изображения исходя из прямоугольника элемента, а затем уменьшаем на величину поля все размеры этого прямоугольника, кроме высоты (то есть ординаты нижней стороны), которую уменьшаем на высоту поля плюс высота строки подписи.

Чтобы загрузить SVG-изображение и нарисовать его, мы используем объект `QSvgRenderer`. Загрузив изображение, мы определяем его оригинальный размер и изменяем последний так, чтобы вписать изображение в вычисленный ранее прямоугольник с сохранением пропорций. Это делает метод `scale()` с параметром `Qt::KeepAspectRatio`. Другие возможные параметры — `Qt::IgnoreAspectRatio` и `Qt::KeepAspectRatioByExpand-`

ing; последнее означает, что изображение будет расширено так, чтобы одно из его измерений (ширина или высота) совпало с заданной шириной или высотой, а другое может при этом превысить заданное измерение, если это нужно для сохранения пропорций.

Поскольку мы хотим отцентрировать изображение по горизонтали, то вычисляем смещение по оси *X*. Затем масштабируем прямоугольник изображения и перемещаем его по горизонтали. И напоследок вызываем метод `QSvgRenderer::render()`, который рисует SVG-изображение на поверхности рисовальщика в области прямоугольника изображения.

Код рисования растрового изображения почти не отличается от предыдущего:

```
QImage image(filename);
QSize size(image.size());
size.scale(imageRect.size(), Qt::KeepAspectRatio);
imageRect.setSize(size);
const int Xoffset = (imageRect.width() - size.width()) / 2;
imageRect.moveTo(imageRect.x() + Xoffset, imageRect.y());
painter->drawImage(imageRect, image);
```

Единственное различие заключается в том, что для загрузки изображения и получения его размера мы используем объект `QImage` вместо `QSvgRenderer`, а для рисования — метод `QPainter::drawImage()`, а не `QSvgRenderer::render()`. Можно было бы внести одно усовершенствование: масштабировать прямоугольник растрового изображения (а, значит, и само изображение) только в том случае, когда размер изображения превышает размер прямоугольника. (Отметим, что в исходном коде имеется директива `#define`, выбирающая SVG-изображения или растры, причем по умолчанию подразумевается SVG.)

Закончив с изображением, мы должны нарисовать подпись. Ординату подписи мы вычисляем следующим образом: прибавляем к ординате левого верхнего угла переданного прямоугольника его высоту, чтобы получить ординату нижней стороны, а потом уменьшаем результат (то есть поднимаемся вверх) на высоту одной строки. Затем мы создаем прямоугольник и рисуем в нем подпись, центрируя ее по горизонтали и по вертикали. В результате подпись оказывается вровень с нижней стороной прямоугольника, а между верхним краем подписи и нижним краем изображения оставлен промежуток (величина поля). При этом нижние выносные элементы подписи касаются нижней стороны прямоугольника (и потому подходят близко к контуру объемлющего прямоугольника, нарисованного в самом начале). Оставляем добавление нижнего поля читателю в качестве упражнения.

Чтобы нарисовать абзацы HTML-разметки, мы можем передать ее объекту `QTextDocument`. Это открывает два подхода к рисованию содержимого документа. Более простой состоит в том, чтобы заставить `QTextDocument` выводить HTML прямо на поверхность рисовальщика; более

трудный – рассматривать `QTextDocument` как контейнер фрагментов форматированного текста и обходить эти фрагменты самостоятельно, рисуя их по очереди. Мы продемонстрируем оба подхода и начнем с кода создания объекта `QTextDocument`, общего для того и другого. (В исходном коде представлены оба варианта, а для выбора одного из них на этапе компиляции предусмотрена директива `#define`.)

```
int PageData::paintHtmlParagraph(QPainter *painter, int y,
                                const QString &html)
{
    const QFont ParagraphFont("Times", 15);
    painter->setFont(ParagraphFont);
    QTextDocument document;
    document.setHtml(html);
```

Сначала создается шрифт абзаца по умолчанию; как мы уже отмечали, он больше, чем обычно, – для того, чтобы сделать разборчивым снимок экрана (стр. 390). Этот шрифт передается рисовальщику, чтобы мы могли затем получить метрики шрифта. Затем мы создаем объект `QTextDocument` и заполняем его абзацем HTML, который нам был передан.

```
    document.setDefaultFont(ParagraphFont);
    document.setUseDesignMetrics(true);
    document.setTextWidth(painter->viewport().width());
    QRect rect(0, y, painter->viewport().width(),
               painter->viewport().height());
    painter->save();
    painter->setViewport(rect);
    document.drawContents(painter);
    painter->restore();
    return y + document.documentLayout()->documentSize().height() +
           painter->fontMetrics().lineSpacing();
}
```

Сначала мы передаем `QTextDocument` тот же самый шрифт по умолчанию, что и рисовальщику. Мы просим документ использовать проектные метрики, потому что при этом должен получиться результат более высокого качества. Кроме того, мы ограничиваем ширину документа шириной порта просмотра – это важно для правильной работы компоновщика (хотя, не будь мы ограничены физическим размером страницы, могли бы использовать ширину `QTextDocument::idealWidth()`). Метод `QTextDocument::drawContents()` может принимать в качестве второго аргумента отсекающий прямоугольник, но здесь он нам не нужен. Мы хотим, чтобы текст начинался на уровне примерно две трети от высоты страницы, но сообщить об этом `QTextDocument` невозможно. Поэтому мы поступаем иначе: сохраняем состояние рисовальщика и изменяем порт просмотра так, чтобы он занимал прямоугольник, верхняя сторона которого имеет нужную нам ординату, а нижняя совпадает с нижним краем страницы (его ширина совпадает с шириной страницы). Затем

мы иницилируем рисование содержимого документа на поверхности рисовальщика (при этом рисование будет производиться в области порта просмотра, что нам и нужно), после чего восстанавливаем исходный порт.

В заключение мы вычисляем ординату следующего за данным абзацем элемента (например, другого абзаца), прибавляя высоту документа (которая теперь известна, потому что для рисования содержимого документа должен был вычислить свое расположение) плюс промежуток высотой в одну строку.

Второй способ нарисовать абзац HTML с помощью `QTextDocument` состоит в том, чтобы рисовать текст самостоятельно. В этом случае мы используем `QTextDocument` как анализатор, преобразующий HTML-разметку во внутреннее представление документа, которое можно обойти, рисуя последовательно каждое слово. Мы продемонстрируем этот подход на примере другой версии второй части метода `paintHtmlParagraph()`.

```
QTextBlock block = document.begin();

Q_ASSERT(block.isValid());
int x = 0;
for (QTextBlock::iterator i = block.begin(); !i.atEnd(); ++i) {
    QTextFragment fragment = i.fragment();
    if (fragment.isValid()) {
        QTextCharFormat format = fragment.charFormat();
        foreach (QString word, fragment.text().split(QRegExp("\\s+"))) {
            int width = painter->fontMetrics().width(word);
            if (x + width > painter->viewport().width()) {
                x = 0;
                y += painter->fontMetrics().lineSpacing();
            }
            else if (x != 0)
                word.prepend(" ");
            x += paintWord(painter, x, y, word, ParagraphFont, format);
        }
    }
    y += painter->fontMetrics().lineSpacing();
}
```

Абзац хранится в одном блоке `QTextBlock`, а блок содержит один или несколько фрагментов `QTextFragment`, у каждого из которых свой формат `QTextCharFormat`. Мы обходим все фрагменты (или *один* фрагмент, если весь текст абзаца отформатирован одинаково), и для каждого получаем формат и перебираем все содержащиеся в нем слова. Текст разбивается на слова с помощью регулярного выражения, выделяющего «один или несколько пробелов», то есть любая последовательность, состоящая из одного или нескольких пробелов, считается одним пробелом. Это правильно, потому что источником текста была HTML-разметка, а в язы-

ке HTML логика обработки пробелов именно такая. (Кстати, создавать регулярное выражение внутри цикла не так дорого, как могло бы показаться: класс `QRegExp` запоминает последние встретившиеся выражения, так что выражение компилируется во внутренний формат при первом конструировании, а на последующих итерациях используется уже откомпилированная форма.)

Выделив слово, мы запрашиваем у рисовальщика объект метрик шрифта и просим его вычислить ширину слова. Если слово не помещается на текущей строке, то мы обнуляем абсциссу и увеличиваем ординату на высоту одной строки. В противном случае добавляем в начало слова пробел, чтобы отделить его от предыдущего слова в той же строке, а координаты не изменяем. Затем мы вызываем вспомогательный метод `paintWord()` и увеличиваем абсциссу на возвращенную им величину.

В заключение мы возвращаем сумму ординаты и высоты строки, чтобы продолжение абзаца рисовалось на одну строку ниже.

```
int PageData::paintWord(QPainter *painter, int x, int y,
    const QString &word, const QFont &paragraphFont,
    const QTextCharFormat &format)
{
    QFont font(format.font());
    font.setFamily(paragraphFont.family());
    font.setPointSize(paragraphFont.pointSize());
    painter->setFont(font);
    painter->setPen(format.foreground().color());
    painter->drawText(x, y, word);
    return painter->fontMetrics().width(word);
}
```

Для каждого рисуемого слова (то есть либо собственно слова, либо слова с предшествующим пробелом) мы первым делом создаем такой же шрифт, как в формате символов, но затем заменяем в нем гарнитуру и кегль значениями из шрифта абзаца. Далее этот шрифт передается рисовальщику, а в качестве цвета пера устанавливается цвет текста из формата (чтобы цветной текст выводился правильно). После этого мы рисуем текст, начиная с точки с переданными координатами. И возвращаем ширину только что нарисованного слова, чтобы вызывающая программа могла увеличить абсциссу.

```
void PageData::paintFooter(QPainter *painter, const QString &footer)
{
    painter->setFont(QFont("Helvetica", 11));
    painter->setPen(Qt::black);
    const int LineHeight = painter->fontMetrics().lineSpacing();
    int y = painter->viewport().height() - LineHeight;
    painter->drawLine(0, y, painter->viewport().width(), y);
    y += LineHeight / 10;
    painter->drawText(
        QRect(0, y, painter->viewport().width(), LineHeight),
```



```
        footer, QTextOption(Qt::AlignCenter));
    }
```

Нижний колонтитул рисуется другим шрифтом с цветом по умолчанию. Сначала мы рисуем горизонтальную черту поперек страницы, отступив на одну строку от нижнего поля. А затем – неформатированный текст нижнего колонтитула прямо под чертой с центрированием по горизонтали.

На этом мы завершаем рассмотрение кода для прямого рисования документа без помощи `QTextDocument`. Если посчитать строки кода, то выяснится, что короче всего оказалось заполнение `QTextDocument` из HTML-разметки, а длиннее всего – рисование. Но это справедливо для нашего тестового документа; для других документов ситуация может сложиться иначе, поэтому мы рекомендуем выбирать подход, лучше отвечающий конкретным условиям, а не тот, что выглядит самым коротким.

Все рассмотренные нами методы вызываются из `PageData::paintPages()` (принимающего в качестве аргумента `QPrinter`), который, в свою очередь, вызывает метод `PageData::paintPage()` (принимающий аргумент типа `QPainter`). Это означает, что те же методы можно использовать для рисования тестового документа на любом принтере или ином устройстве рисования, в чем мы убедимся в следующих подразделах.

Рисование PDF- и PostScript-документов

Нарисовать PDF- или PostScript-документ можно, не прибегая к дополнительным средствам; все, что нам нужно, – это имя файла с суффиксом *.pdf* или *.ps*.

```
bool MainWindow::paintPdfOrPs(const QString &filename)
{
    Q_ASSERT(filename.endsWith(".ps") || filename.endsWith(".pdf"));
    QPrinter printer(QPrinter::HighResolution);
    printer.setOutputFileName(filename);
    printer.setOutputFormat(filename.endsWith(".pdf")
        ? QPrinter::PdfFormat : QPrinter::PostScriptFormat);
    pageData.paintPages(&printer);
    return true;
}
```

Сначала создаем объект `QPrinter`, но вместо печати на листах бумаги задаем режим печати в файл, устанавливая имя выходного файла и формат. (Структурно этот код ничем не отличается от метода `exportPdfOrPs()`, рассмотренного выше, на стр. 403.)

Рисование SVG-документов

SVG-документ можно нарисовать с помощью объекта `QSvgGenerator`, передав ему имя выходного файла и размер страницы. (Хотя SVG-изображения масштабируются, для них принято задавать оригинальный раз-

мер.) Для *svg*-файлов мы решили создавать SVG-изображение одной страницы, а не всего многостраничного документа.

```
bool MainWindow::paintSvg(const QString &filename)
{
    Q_ASSERT(filename.endsWith(".svg"));
    QSvgGenerator svg;
    svg.setFileName(filename);
    QRect rect = printer.pageRect().adjusted(25, 25, -25, 25);
    svg.setSize(rect.size());
    QPainter painter(&svg);
    painter.setViewport(rect);
    pageData.paintPage(&painter, 0);
    return true;
}
```

Сама процедура рисования очень проста: создаем объект `QPainter` и рисуем страницу (с номером 0). В данном случае мы сдвинули изображение, оставив небольшое поле. (Структурно этот код ничем не отличается от метода `exportSvg()`, рассмотренного выше, на стр. 406.)

Рисование растровых изображений

Объект `QPainter` рисует изображения, которые можно затем вывести в виде растровых или векторных файлов. То есть `QPainter` рисует в `QImage`, а `QImage` можно сохранить в любом из поддерживаемых Qt растровых форматов. (Список таких форматов возвращает метод `QImageWriter::supportedImageFormats()`.)

```
bool MainWindow::paintImage(const QString &filename)
{
    QImage image(printer.paperRect().size(), QImage::Format_ARGB32);
    QPainter painter(&image);
    painter.setRenderHints(QPainter::Antialiasing|
                          QPainter::TextAntialiasing);
    painter.fillRect(painter.viewport(), Qt::white);
    painter.setViewport(printer.pageRect());
    pageData.paintPage(&painter, 0);
    return image.save(filename);
}
```

Здесь мы создаем объект `QImage` нужного нам размера, включаем режим сглаживания, закрашиваем фон белым цветом (можно было бы установить и прозрачный фон `Qt::transparent`) и рисуем одну страницу (с номером 0). Как и в случае SVG, мы решили создавать по одному изображению для каждой страницы. (Структурно этот код ничем не отличается от метода `exportImage()`, рассмотренного выше, на стр. 407.)

Итак, мы узнали, как печатать и экспортировать документы, содержащие форматированный текст, изображения и сложную разметку (например, таблицы). Мы обсудили, как напечатать и экспортировать

одну страницу документа `QTextDocument` в виде SVG-изображения, а также в большинстве стандартных растровых форматов, поддерживаемых Qt, и как достичь того же результата с помощью объекта `QPainter`. Кроме того, мы научились печатать и экспортировать целые многостраничные документы `QTextDocument` в форматах PDF, PostScript, Open Document Format и HTML, а также рисовать эти документы непосредственно с помощью `QPainter` для вывода файлов в формате PDF и PostScript.

В этой главе мы активно использовали класс `QPainter` для рисования документов. В предыдущих главах мы видели, что этот класс применяется также для рисования пользовательских виджетов и графики. Но для написания сложных графических программ библиотека Qt предлагает впечатляющую альтернативу `QPainter`: архитектуру графических представлений, которой мы и займемся в следующих двух главах.

11

Подсистема Graphics View: создание окон

- Архитектура Graphics View
- Графические представления: виджеты и компоновщики
- Введение в графические элементы

Переопределив метод `paintEvent()` в подклассе `QWidget` и воспользовавшись классом `QPainter`, мы можем нарисовать все, что угодно. Это идеально для пользовательских виджетов, но совсем не так удобно, когда требуется нарисовать много отдельных объектов, особенно если нужно предоставить пользователю возможность взаимодействовать с ними. Например, в прошлом некоторые программисты создавали графические приложения, содержавшие буквально тысячи пользовательских виджетов, которые играли роль графических объектов. И хотя виджеты рисуются очень быстро, для обработки единственного щелчка мышью в такой ситуации вполне может потребоваться вся вычислительная мощность процессора. К счастью, в версию Qt 4.2 была включена архитектура графических представлений, которая прекрасно подходит для реализации высокопроизводительного рисования многих элементов и взаимодействия с ними.

Первоначально графические представления задумывались как более совершенная замена классу `QCanvas` из версии Qt 3, но впоследствии возможности этой архитектуры вышли далеко за пределы функциональности холста. Фактически в некоторых современных приложениях виджет `QGraphicsView` используется в качестве центрального виджета главного окна, а все виджеты, предназначенные для организации поль-

зовательского интерфейса, располагаются внутри него как полноценные графические элементы.

В первом разделе этой главы мы дадим краткий обзор архитектуры графических представлений и, в частности, во врезках расскажем о некоторых существенных новшествах версии Qt 4.6. Во втором разделе мы рассмотрим приложение, в котором центральным виджетом главного окна является `QGraphicsView`, где есть как виджеты, так и обычные графические элементы. И наконец, в третьем разделе мы ознакомимся с подклассом класса `QGraphicsItem` и API этого класса.

В следующей главе мы обратимся к более традиционному применению архитектуры графических представлений – простой программе рисования – и получим возможность внимательнее рассмотреть большинство классов, составляющих эту архитектуру. Там же будут приведены дополнительные примеры создания пользовательских графических элементов. Кстати говоря, в главе 13 мы снова вернемся к примерам из этой и следующей глав и модифицируем их с использованием некоторых средств, появившихся в Qt 4.6.

Архитектура Graphics View

Архитектура графических представлений похожа на архитектуру модель/представление в том смысле, что в ней имеется невидуальный класс для хранения данных (`QGraphicsScene`) и класс для визуализации этих данных (`QGraphicsView`). Одну и ту же сцену можно визуализировать разными представлениями, если это необходимо. Графическая сцена содержит элементы – объекты классов, производных от абстрактного класса `QGraphicsItem`.

С самого начала в разработку архитектуры графических представлений было вложено много усилий, направленных на повышение быстродействия и расширение возможностей. Сцены можно масштабировать, поворачивать и распечатывать, а для их отображения использовать как встроенный в Qt движок, так и библиотеку OpenGL. Архитектура поддерживает также анимацию и перетаскивание мышью. Графические сцены пригодны для представления от единиц до десятков тысяч и более элементов.

В Qt включено много уже готовых типов графических элементов; они показаны на рис. 11.1. Большинство имен классов не нуждаются в пояснениях, однако остановимся на менее очевидных. Класс `QGraphicsPathItem` представляет объект типа `QPainterPath` – произвольную фигуру, составленную из фундаментальных примитивов, которые Qt умеет рисовать: дуг, кривых Безье, сегментов, эллипсов, прямых, прямоугольников и текстовых строк. Класс `QGraphicsSimpleTextItem` представляет простой текст, а `QGraphicsTextItem` – форматированный текст (который можно задать с помощью HTML-разметки; форматированный текст

рассматривался в двух предыдущих главах). Класс `QGraphicsWidget` является базовым для создания пользовательских виджетов, предназначенных для размещения на графической сцене. На сцену можно помещать и виджеты, производные от стандартного класса `QWidget`, для этого нужно передать виджет конструктору класса `QGraphicsProxyWidget` и поместить прокси-виджет на сцену. Прокси-виджеты (или сами объекты `QWidget`) работают медленно, но будет ли это замедление заметно, зависит от приложения¹. Класс `QGraphicsWebView` появился в версии Qt 4.6, он представляет вариант обсуждавшегося в главе 1 класса `QWebView` и ориентирован на сцены.

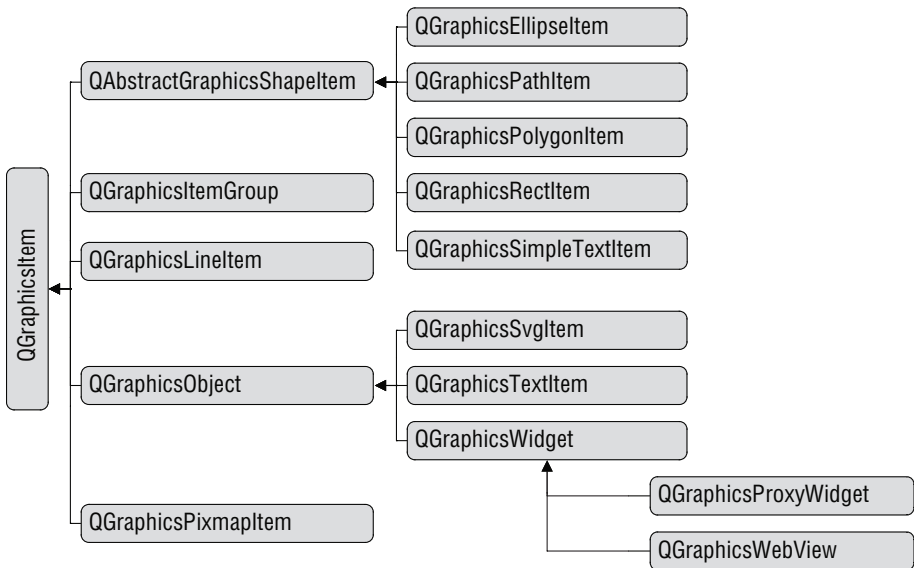


Рис. 11.1. Иерархия классов, производных от `QGraphicsItem`

Для сцен с небольшим числом элементов можно использовать класс `QGraphicsObject`, включенный в Qt 4.6, а в Qt 4.5 и более ранних версиях можно произвести собственный класс одновременно от `QObject` и `QGraphicsItem` и сделать его основой для своих графических элементов. При этом возрастают издержки (каждый элемент потребляет больше памяти), зато мы получаем все преимущества сигналов и слотов и системы свойств Qt. Для сцен с очень большим числом элементов обычно лучше применять в качестве основы последних облегченный класс `QGraphics-`

¹ Производительность `QWidget` и прокси-виджетов в контексте сцен рассматривается в статье по адресу labs.qt.nokia.com/blogs/2010/01/11/qt-graphics-and-performance-the-cost-of-convenience.

Item, а QGraphicsObject оставить для элементов, которых сравнительно не много.

Классы графических представлений двумерны, хотя у каждого элемента есть координата z ; элементы с большим значением z рисуются поверх элементов с меньшим значением. Столкновения определяются путем анализа координат x и y . Помимо информации о столкновениях, сцена может сообщить, какие элементы содержат данную точку или находятся в указанной области и какие из них выделены. У сцены также имеется передний план; это бывает полезно, например, когда нужно нарисовать сетку поверх всех присутствующих элементов. Есть также задний план, рисуемый под всеми элементами; это полезно, если нужно поместить на сцену фоновое изображение или залить ее каким-то цветом.

Каждый элемент является потомком либо сцены, либо другого элемента – аналог обычного отношения родитель–потомок для виджетов Qt. Когда к элементу применяется некоторое преобразование, оно рекурсивно распространяется на всех его потомков произвольной глубины. Это означает, что если некоторый элемент перемещается, – например, перетаскивается пользователем, – то вместе с ним перетаскиваются и все его потомки. Можно также заставить потомка игнорировать преобразования, применяемые к его родителю, вызвав метод `QGraphicsItem::setFlag(QGraphicsItem::ItemIgnoresTransformations)`. Чаще, однако, употребляются флаги, позволяющие сделать элемент перемещаемым, выделяемым и способным принять фокус. (Все возможные флаги перечислены в табл. 11.6 и 11.7; стр. 441.) Элементы можно также сгруппировать, сделав их потомками объекта `QGraphicsItemGroup`; это бывает полезно для создания временных наборов элементов.

Для работы с классами графических представлений существует три разных системы координат, хотя на практике мы обычно имеем дело только с двумя. Для представлений используется физическая система координат, а для сцен – логическая, которая обычно определяется путем передачи объекта `QRectF` конструктору сцены. Qt автоматически отображает координаты сцены на координаты представления. По существу, для сцен применяются координаты окна (логические), а для представлений – координаты порта просмотра (физические). При позиционировании элементов указываются координаты сцены. Третья система координат используется самими элементами. Это удобно, поскольку начало координат $(0, 0)$ находится в логическом центре элемента (за исключением текстовых элементов, для которых начало находится в верхнем левом углу). На практике это означает, что элемент всегда можно рисовать относительно его центра, не думая о том, какие преобразования были применены к нему родителем, поскольку об этом позаботится сцена. Отметим, что в Qt ось ординат направлена вниз, то есть точка $(5, 8)$ расположена на 6 пикселей *выше* точки $(5, 14)$. На рис. 11.2 показано соотношение между координатами сцены и элементов.

При переходе от версии Qt 4.5 к 4.6 некоторые аспекты архитектуры Graphics View изменились; все отличия перечислены во врезке «Изменения в поведении графических представлений в Qt 4.6» (стр. 426).

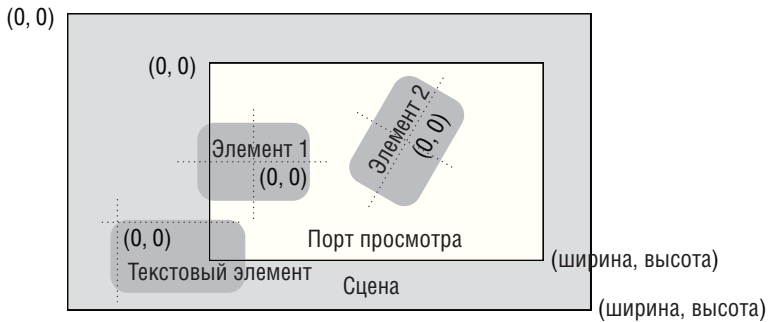


Рис. 11.2. Для графических элементов применяется локальная система координат

Графические представления: виджеты и компоновщики

В этом разделе мы рассмотрим приложение Petri Dish (Чашка Петри) (petridish1), изображенное рис. 11.3. Класс его главного окна `MainWindow` наследует классу `QMainWindow`, а центральным виджетом мы сдела-

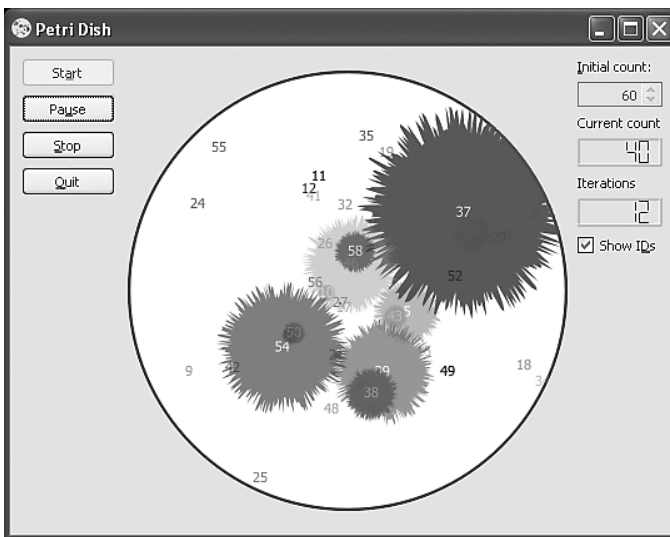


Рис. 11.3. Приложение Petri Dish

ли QGraphicsView. Приложение Petri Dish моделирует поведение клеток, которые растут, если им не тесно, и уменьшаются, если они оказались сильно удаленными от других клеток, или популяция чересчур разрослась, или клетка стала слишком большой. Маленькие клетки в случайные моменты времени «погибают». Мы не будем долго распространяться на тему самой модели или логики приложения, а сосредоточим внимание на предмете этой главы – архитектуре графических представлений.

Изменения в поведении графических представлений в Qt 4.6

При переходе от версии Qt 4.5 к 4.6 классы графических представлений претерпели существенную переработку, что привело к резкому повышению производительности. Вследствие внутренних изменений оказалось необходимо внести некоторые модификации и в видимое пользователям¹ поведение, чтобы добиться максимально возможной оптимизации. Основные изменения таковы:

- В открытой переменной `exposedRect` типа `QRectF`, принадлежащей классу `QStyleOptionGraphicsItem`, хранится видимый (`exposed`) прямоугольник элемента в координатах самого элемента. Однако эта переменная устанавливается только для графических элементов с флагом `ItemUsesExtendedStyleOption`.
- Переменные `levelOfDetail` и `matrix` из класса `QStyleOptionGraphicsItem` более не используются. В Qt 4.6 правильным способом получения уровня детализации считается обращение к статическому методу `QStyleOptionGraphicsItem::levelOfDetailFromTransform()`.
- `QGraphicsView` больше не вызывает метод `QGraphicsView::drawItems()` или `QGraphicsView::drawItem()`, если не установлен флаг «оптимизации» `QGraphicsView::IndirectPainting` (что не рекомендуется).
- `QGraphicsItem` больше не вызывает метод `itemChange()` в случае изменения позиции и преобразования. Чтобы получать уведомление о таких изменениях, нужно установить флаг `QGraphicsItem::ItemSendsGeometryChanges`. (Этот флаг по умолчанию установлен для объектов классов `QGraphicsWidget` и `QGraphicsProxyWidget`.)

Даже если флаг `ItemSendsGeometryChanges` установлен, при выполнении преобразований метод `itemChange()` вызывается, только когда используется метод `setTransform()`. В версии Qt 4.7 при

¹ Автор имеет в виду пользователей библиотеки, то есть разработчиков приложений. – *Прим. науч. ред.*

установленном флаге метод `itemChange()` будет также вызываться, если используются методы `setRotation()`, `setScale()` и `setTransformOriginPoint()` (все они уже включены в Qt 4.6).

В какой мере эти изменения затрагивают конкретное приложение – и затрагивают ли вообще, – зависит от того, какие именно возможности графических представлений использовались. Из примеров, представленных в этой книге, можно отметить приложение *Page Designer*, рассматриваемое в следующей главе, – для него характерно последнее из перечисленных выше изменений.

Ниже мы рассмотрим относящиеся к делу методы главного окна (или их фрагменты), чтобы показать, как создается главное окно с графической сценой в центре. А в следующем разделе обратимся к объектам типа *Cell* (класс, производный от *QGraphicsItem*), заострив внимание на основных принципах создания графических элементов. Там же мы ознакомимся с API класса *QGraphicsItem*, не останавливаясь на логике моделирования, поскольку она не имеет отношения к обсуждаемой теме. (Исходный код находится в подкаталоге *petridish1*.)

В приложении есть кнопки для запуска процесса, его приостановки/возобновления, останова и выхода, позволяющие управлять моделированием. Пользователь может задать начальное число клеток и режим показа идентификаторов – последнее удобно для клеток, которые настолько малы, что иначе их не разглядишь. (Во время моделирования поле для задания числа клеток недоступно.)

В пользовательском интерфейсе есть два виджета *QLCDNumber*, показывающих, сколько клеток осталось и сколько шагов моделирования уже выполнено.

Начнем с рассмотрения конструктора главного окна, а затем перейдем к вспомогательным методам, относящимся к программированию графических представлений.

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), simulationState(Stopped), iterations(0)
{
    scene = new QGraphicsScene(this);
    scene->setItemIndexMethod(QGraphicsScene::NoIndex);
    createWidgets();
    createProxyWidgets();
    createLayout();
    createCentralWidget();
    createConnections();
    startButton->setFocus();
}
```

```

        setWindowTitle(QApplication::applicationName());
    }

```

Создание графической сцены `QGraphicsScene` выглядит не совсем обычно, потому что мы не задали ее размеров. Хотя высоту мы знаем заранее (достаточную для размещения чашки Петри и небольших полей), ширина зависит от ширины дочерних виджетов, поэтому размеры мы установим после того, как создадим все виджеты и скомбинируем их.

При добавлении, перемещении и удалении элементов со сцены необходимо пересчитывать их расположение. Например, если элемент добавлен в видимую часть сцены, то его следует нарисовать, а если видимый элемент перемещен или удален, то следует нарисовать то, что раньше было им закрыто, а теперь появилось. Для сцен с большим числом статических элементов вычисления можно существенно ускорить за счет применения алгоритма двоичного разбиения пространства (`Binary Space Partitioning`) и соответствующего метода `QGraphicsScene::BspTreeIndex`, но для очень динамичных сцен, когда элементы постоянно добавляются, перемещаются и удаляются, индексирование лучше отключить (мы так и поступили), поскольку связанные с ним непроизводительные издержки могут перевесить получаемый выигрыш.

Следуя принятому в этой книге стилю кодирования, мы поручаем большую часть инициализации виджета вспомогательным методам, вызываемым из конструктора. Поскольку центральным виджетом главного окна является графическая сцена, то интерес представляют все эти методы, так что мы приведем и обсудим их (по возможности опуская повторяющийся код).

```

void MainWindow::createWidgets()
{
    startButton = new QPushButton(tr("St&art"));
    pauseOrResumeButton = new QPushButton(tr("Pa&use"));
    pauseOrResumeButton->setEnabled(false);
    stopButton = new QPushButton(tr("Stop"));
    quitButton = new QPushButton(tr("Quit"));
    QString styleSheet("background-color: bisque;");
    initialCountLabel = new QLabel(tr("Initial count:"));
    initialCountLabel->setStyleSheet(styleSheet);
    ...
    AQP::accelerateWidgets(QList<QWidget*>() << startButton
        << stopButton << quitButton << initialCountLabel
        << showIdsCheckBox);
}

```

В этом приложении используются стандартные виджеты, производные от `QWidget`, их создание не таит никаких сюрпризов. Единственная необычная вещь — это задание для всех виджетов (кроме кнопок) таблицы стилей; в результате у них оказывается одинаковый цвет фона. С кнопками стиль не ассоциируется, поскольку мы предпочитаем, чтобы они

выглядели так, как это уместно для конкретной платформы и специфики приложения.

```
void MainWindow::createProxyWidgets()
{
    proxyForName["startButton"] = scene->addWidget(startButton);
    proxyForName["pauseOrResumeButton"] = scene->addWidget(
        pauseOrResumeButton);
    ...
}
```

Все виджеты необходимо поместить на сцену, потому что представление сцены – центральный виджет главного окна. Мы могли бы поступить и по-другому, например сделать центральным виджетом `QWidget`, связать с ним компоновщик `QHBoxLayout`, внутри него поместить `QVBoxLayout` с кнопками, потом `QGraphicsView` и еще один `QVBoxLayout` со всеми остальными виджетами. Но просто ради демонстрации мы решили сделать центральным виджетом именно `QGraphicsView`, а остальные поместить внутри него как графические элементы.

Чтобы добавить стандартный `QWidget` на сцену, необходимо сначала создать прокси-объект `QGraphicsProxyWidget` и поместить на сцену именно его. В этом методе мы вызываем метод `QGraphicsScene::addWidget()`, который сам создает объект `QGraphicsProxyWidget`, представляющий виджет, переданный в качестве аргумента, и возвращает указатель на него. Для удобства мы создали хеш, в котором ключами служат имена виджетов, а значениями – указатели на прокси-виджеты. Каждый созданный прокси-виджет помещается в этот хеш. (Хеш объявлен в заголовочном файле в виде `QHash<QString, QGraphicsProxyWidget*> proxyForName.`)

После того как виджеты и их прокси созданы, можно приступить к компоновке. Принцип тот же, что для стандартных компоновщиков `Qt`, только необходимо использовать классы компоновки, специфичные для графических сцен. Метод `createLayout()` мы рассмотрим в два приема: сначала покажем, как создаются компоновщики, а потом – как задаются размеры сцены.

```
const int DishSize = 350;
const int Margin = 20;

void MainWindow::createLayout()
{
    QGraphicsLinearLayout *leftLayout = new QGraphicsLinearLayout(
        Qt::Vertical);
    leftLayout->addItem(proxyForName["startButton"]);
    leftLayout->addItem(proxyForName["pauseOrResumeButton"]);
    leftLayout->addItem(proxyForName["stopButton"]);
    leftLayout->addItem(proxyForName["quitButton"]);
    QGraphicsLinearLayout *rightLayout = new QGraphicsLinearLayout(
        Qt::Vertical);
```

```

foreach (const QString &name, QStringList())
    << "initialCountLabel" << "initialCountSpinBox"
    << "currentCountLabel" << "currentCountLCD"
    << "iterationsLabel" << "iterationsLCD"
    << "showIdsCheckBox")
rightLayout->addItem(proxyForName[name]);

QGraphicsLinearLayout *layout = new QGraphicsLinearLayout;
layout->addItem(leftLayout);
layout->setItemSpacing(0, DishSize + Margin);
layout->addItem(rightLayout);
QGraphicsWidget *widget = new QGraphicsWidget;
widget->setLayout(layout);
scene->addItem(widget);

```

Класс QGraphicsLinearLayout — это компоновщик для графических представлений, соответствующий классу QVBoxLayout, базовому для QHBoxLayout и QVBoxLayout. АРІ очень похожи, только добавляются не виджеты методом QVBoxLayout::addWidget(), а элементы методом QGraphicsLinearLayout:: addItem(). Этот метод включает в компоновку объект типа QGraphicsLayoutItem (это один из базовых классов QGraphicsWidget, а значит, и QGraphicsProxyWidget). Имеется также класс QGraphicsGridLayout, соответствующий классу QGridLayout. А в версии Qt 4.6 появился класс QGraphicsAnchorLayout, который реализует совершенно новый подход к компоновке, не встречавшийся в Qt ранее. Он основан на принципе позиционирования виджетов относительно друг друга, а также сторон и углов прямоугольника, занятого компоновщиком.

В этом методе мы создаем три компоновщика QGraphicsLinearLayout. Первый отвечает за вертикальный столбец виджетов слева, второй — за вертикальный столбец виджетов справа, третий — за общее размещение по горизонтали: левый компоновщик, промежуток (для самой чашки Петри), правый компоновщик. Схематически компоновка виджетов показана на рис. 11.4.

Закончив с компоновкой, мы создаем новый «пустой» объект QGraphicsWidget. У этого класса нет визуального представления, его назначение — служить базовым классом для пользовательских виджетов графических представлений, а также применяться для той цели, для которой он использован здесь, — как контейнер для одного или нескольких дочерних уже скомпонованных виджетов. Создав этот виджет, мы ассоциируем с ним компоновщик и помещаем его на сцену. В результате у всех компоновщиков и прокси-виджетов изменяется родитель — например, прокси-виджеты становятся дочерними элементами сцены. (Сами виджеты становятся потоками своих прокси-виджетов после вызова метода QGraphicsScene::addWidget().)

```

int width = qRound(layout->preferredWidth());
int height = DishSize + (2 * Margin);

```

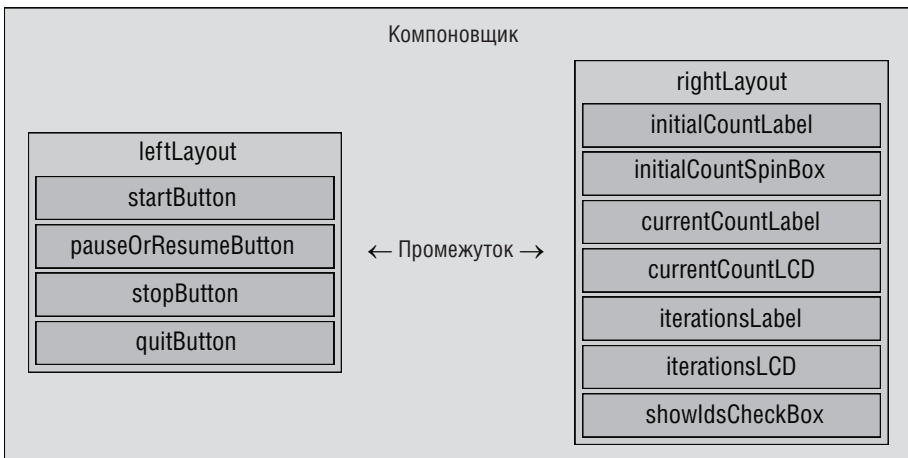


Рис. 11.4. Компоновка главного окна приложения *Petri Dish*

```

setMinimumSize(width, height);
scene->setSceneRect(0, 0, width, height);
}

```

Мы задаем ширину сцены настолько большой, чтобы компоновщик поместился на ней целиком, а высоту – достаточной для отображения чашки Петри и вертикального поля. Кроме того, мы задаем минимальный размер главного окна, чтобы его нельзя было уменьшить до такой степени, что чашка Петри и виджеты станут не видны.

```

void MainWindow::createCentralWidget()
{
    dishItem = new QGraphicsEllipseItem;
    dishItem->setFlags(QGraphicsItem::ItemClipsChildrenToShape);
    dishItem->setPen(QPen(QColor("brown"), 2.5));
    dishItem->setBrush(Qt::white);
    dishItem->setRect(pauseOrResumeButton->width() + Margin,
                    Margin, DishSize, DishSize);
    scene->addItem(dishItem);
    view = new QGraphicsView(scene);
    view->setRenderHints(QPainter::Antialiasing|
                        QPainter::TextAntialiasing);
    view->setBackgroundBrush(QColor("bisque"));
    setCentralWidget(view);
}

```

После создания сцены и помещения на нее виджетов (точнее, прокси-виджетов) мы вызываем этот метод, который создает чашку Петри и представление, в результате чего приложение принимает законченную форму.

Сначала создается элемент-эллипс, хотя в этом случае он будет просто окружностью, так как ширина и высота одинаковы. Мы задаем для этого элемента режим отсечения потомков. Все модельные клетки создаются как потомки чашки Петри, и такой режим означает, что клетки, оказавшиеся вне чашки, не показываются, а для клеток, пересекающих границу чашки, показываются только части, расположенные внутри. Смещение прямоугольника чашки Петри по оси *X* равно ширине одной из кнопок в левом столбце плюс небольшое поле, а смещение по оси *Y* такое, чтобы над чашкой оставалось небольшое свободное пространство. Создав чашку, мы помещаем ее на сцену.

Мы создаем стандартный объект `QGraphicsView`, включаем для него режим сглаживания и заливаем фон тем же цветом, что указан в таблице стилей, примененной к некоторым виджетам. Затем мы делаем это представление центральным виджетом главного окна и на этом завершаем оформление приложения.

С точки зрения структуры кода использование архитектуры Graphics View для размещения виджетов в главном окне не сильно отличается от более привычного подхода. Существенных различий всего два: для каждого реального виджета необходимо создать прокси-виджет, а вместо обычных классов компоновки использовать классы, специфичные для графических представлений. Конечно, если бы мы захотели использовать подклассы `QGraphicsWidget`, нужды в прокси-виджетах не возникло бы, поскольку их можно помещать на сцену непосредственно. (Во время написания этой книги единственным подклассом `QGraphicsWidget`, помимо `QGraphicsProxyWidget`, был класс `QGraphicsWebView`, хотя никто не мешает создавать такие подклассы самостоятельно.)

```
void MainWindow::createConnections()
{
    connect(startButton, SIGNAL(clicked()), this, SLOT(start()));
    connect(pauseOrResumeButton, SIGNAL(clicked()),
           this, SLOT(pauseOrResume()));
    connect(stopButton, SIGNAL(clicked()), this, SLOT(stop()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(showIdsCheckBox, SIGNAL(toggled(bool)),
           this, SLOT(showIds(bool)));
}
```

Этот метод полностью аналогичен уже встречавшимся в предыдущих главах; устанавливаются соединения между сигналами `clicked()` реальных виджетов и соответствующими слотами. К программированию графических представлений как таковому это не имеет отношения, но мы все же привели этот код, чтобы впоследствии сравнить его с вариантом для версии Qt 4.6 в главе 13, где для управления поведением приложения используется класс `QStateMachine`, что позволяет уменьшить количество слотов и упростить логику.

Введение в графические элементы

Класс `QGraphicsItem` является базовым для всех графических элементов. Хотя количество методов в нем внушает уважение – в версии Qt 4.6 их больше двух сотен, – создавать его экземпляры нельзя, так как в нем есть два чисто виртуальных метода: `boundingRect()` и `paint()`. Метод `paint()` соответствует методу `QWidget::paintEvent()`; его реализация должна рисовать элемент. Метод `boundingRect()` сообщает инфраструктуре об ограничивающем прямоугольнике элемента – он применяется для обнаружения столкновений и для того, чтобы не перерисовывать элемент, находящийся вне порта просмотра.

Если мы собираемся создавать непрямоугольные элементы, то имеет смысл переопределить также метод `shape()`. Он должен возвращать объект `QPainterPath`, который точно описывает контур элемента. Это полезно для точного распознавания столкновений и определения элемента, в котором имел место щелчок мышь.

Можно переопределить и многие другие виртуальные методы, в частности `advance()`, `collidesWithItem()`, `collidesWithPath()`, `contains()`, `isObscuredBy()`, `opaqueArea()` и `type()`. Все защищенные методы (за исключением `prepareGeometryChange()`) также виртуальны, поэтому можно переопределить все обработчики событий графического элемента (в частности, `contextMenuEvent()`, `keyPressEvent()` и события мыши). Список методов приведен в табл. 11.1–11.4 на стр. 436–440.

Чтобы создать свою фигуру, проще всего воспользоваться одним из стандартных подклассов `QGraphicsItem`, например `QGraphicsPathItem` или `QGraphicsPolygonItem`. А чтобы создать фигуру со специальным поведением, можно унаследовать какому-нибудь подклассу `QGraphicsItem` и переопределить некоторые защищенные обработчики событий, например `keyPressEvent()` или `mousePressEvent()`. Если же нужно нарисовать фигуру самостоятельно, то можно унаследовать самому классу `QGraphicsItem` и переопределить методы `boundingRect()`, `paint()` и `shape()`, а также обработчики событий, определяющие желаемое поведение. Во всех подклассах `QGraphicsItem` рекомендуется определить перечисление `Type` и переопределить метод `type()` (пример будет приведен ниже).

В этом разделе мы кратко рассмотрим те аспекты класса `Cell` – прямого подкласса `QGraphicsItem`, – которые имеют отношение к архитектуре графических представлений. Начнем с объявления этого класса в заголовочном файле, опустив закрытую секцию.

```
class Cell : public QGraphicsItem
{
public:
    enum {Type = UserType + 1};

    explicit Cell(int id, QGraphicsItem *parent=0);
```



```

    QRectF boundingRect() const { return m_path.boundingRect(); }
    QPainterPath shape() const { return m_path; }
    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option, QWidget *widget);
    int type() const { return Type; }
    ...
};

```

Хотя определять перечисление `Type` и переопределять метод `type()` не обязательно, мы рекомендуем делать это во всех подклассах графических элементов. Это позволяет уникально идентифицировать все типы пользовательских графических элементов и означает, что они будут совместимы с функцией `qgraphicsitem_cast<>()`, которая приводит указатели на `QGraphicsItem` к правильному типу подкласса `QGraphicsItem`. (Функция `qgraphicsitem_cast<>()` поддерживает только приведение указателя на `QGraphicsItem` к его подклассу, но не обратное приведение указателя на подкласс к указателю на `QGraphicsItem`. Для приведения обратно к типу указателя на `QGraphicsItem` необходима другая техника. Мы обсудим эту тему в следующей главе, на стр. 457.)

В этом конкретном примере мы создали закрытую переменную `m_path` типа `QPainterPath` (путь, который динамически изменяет форму по ходу моделирования). Этот путь мы можем использовать для вычисления ограничивающего прямоугольника и формы элемента. Отметим, однако, что вычисление ограничивающего прямоугольника по пути — не особенно быстрый процесс, хотя для чашки Петри его скорость вполне достаточна. В других приложениях, где класс `QPainterPath` используется подобным образом, будет разумно кэшировать ограничивающий прямоугольник пути.

Метод `shape()` реализуется тривиально, потому что, как мы скоро увидим, путь рисуется без пера, одной лишь кистью. Если бы мы хотели нарисовать путь пером — скажем, «бублик», получающийся в случае, когда путь является эллипсом, а перо толстое, — то возвращенная форма была бы неточной, поскольку не учитывала бы толщину контура. Это означает, что щелчок мышью по контуру мог бы не дать никакого эффекта, потому что мышь оказалась «вне» эллипса. В таких случаях мы могли бы создать объект `QPainterPathStroker`, установить для него относящиеся к перу параметры с помощью таких методов, как `setWidth()`, `setJoinStyle()` и т. п., а затем вызвать метод `QPainterPathStroker::createStroke()`, передав путь в качестве аргумента. Этот метод возвращает новый путь, являющийся контуром исходного пути, но с установленными нами параметрами пера.

В классе `Cell` конструктор (не показан) всего лишь задает перо и начальный размер, после чего вызывает закрытый метод (тоже не показан) для создания исходной фигуры. В результате метод `paint()` оказывается гораздо проще, чем мог бы быть, поскольку ему нужно только нарисовать путь и, возможно, идентификатор элемента.

```
void Cell::paint(QPainter *painter,
                const QStyleOptionGraphicsItem *option, QWidget*)
{
    painter->setPen(Qt::NoPen);
    painter->setBrush(m_brush);
    painter->drawPath(m_path);
    if (s_showIds) {
        QPointF center = m_path.boundingRect().center();
        QString id = QString::number(m_id);
        center.setX(center.x() - (option->fontMetrics.width(id) / 2));
        center.setY(center.y() + (option->fontMetrics.height() / 4));
        painter->setPen(QPen());
        painter->drawText(center, id);
    }
}
```

Сначала мы определяем перо рисовальщика (параметр `Qt::NoPen` означает, что контур рисовать не нужно) и его кисть, а затем рисуем путь, описывающий клетку. (В классе `Cell` имеется также статическая булевая переменная `s_showIds` с соответствующими статическими методами доступа и переменная-член `m_id` типа `int` – все это не показано.) Если требуется выводить идентификаторы элементов, то мы находим центр пути и рисуем идентификатор пером `QPen()`, располагая его по горизонтали в центре, а по вертикали – отступив на четверть высоты от верхней стороны. Конструктор класса `QPen` без аргументов создает *косметическое* перо в виде сплошной черной линии толщиной 1 пиксел. Перо называется косметическим, если оно игнорирует преобразования.

В параметре типа `QStyleOptionGraphicsItem*` хранятся среди прочего размер видимого прямоугольника элемента, метрики шрифта (именно их мы здесь и используем) и палитра. Параметр `QWidget*` применяется редко¹.

Реализация метода `paint()` работает достаточно быстро для приложения `Petri Dish`, но далека от оптимальной. Пожалуй, в данном случае не имеет смысла кэшировать ограничивающий прямоугольник пути, потому что на каждом шаге моделирования клетки могут как увеличиваться, так и уменьшаться в размере. Однако идентификатор клетки никогда не изменяется, поэтому можно было бы пожертвовать толикой памяти в обмен на скорость и создать закрытую переменную-член `m_id-String` типа `QString`, инициализировав ее в конструкторе, – это позволит обойтись без вызова из `paint()` метода `QString::number()`, который каждый раз выделяет память. Вычисление ширины и высоты по метрикам шрифта тоже работает медленно; можно было бы один раз вычислить их в конструкторе и закэшировать. Но, вообще говоря, лучше сначала получить работающий код рисования, а потом, если его скорость ока-

¹ В нем передается указатель на виджет, в котором происходит рисование. В ряде случаев этот параметр может быть равен 0. – *Прим. науч. ред.*

жется недостаточной, начать думать о том, что бы такое закэшировать. Естественно, эффект изменений следует измерять на тестах, проверяя, что предполагаемое улучшение действительно достигнуто.

В классе `Cell` не переопределяются обработчики событий и не устанавливаются никакие флаги (например, `ItemIsMovable` или `ItemIsSelectable`), поэтому пользователь не сможет взаимодействовать с клетками. Примеры установки этих флагов и реализации обработчиков событий мы увидим в следующей главе. Конец этой главы отдан под таблицы с методами API класса `QGraphicsItem`. В табл. 11.1–11.4 (стр. 436–440) приведены методы этого класса, а в табл. 11.5–11.7 (стр. 441) – наиболее важные перечисления.

На этом мы заканчиваем обзор приложения `Petri Dish` и API класса `QGraphicsItem`. В исходном коде есть малозначительные детали, которые мы опустили. Например, после каждой итерации используется однократный таймер, который запускает следующую итерацию, – мы не можем использовать объект `QTimer` с фиксированным промежутком времени, потому что время вычислений на каждой итерации разнится. Кроме того, на время, пока приложение приостановлено, мы делаем все его окно немного прозрачным – лучше всего этот эффект работает в Windows.

В следующей главе мы рассмотрим более традиционное применение архитектуры графических представлений и приведем дополнительные примеры создания пользовательских графических элементов, а также обсудим загрузку и сохранение сцен в файлах и манипулирование элементами на сцене: геометрические преобразования, копирование, вырезание и вставку.

Таблица 11.1. API класса `QGraphicsItem`, избранные методы (часть 1)

Метод	Описание
<code>advance()</code>	Переопределяется для выполнения анимации; возможны и другие подходы (см. примеры в главах 12 и 13)
<code>boundingRect()</code>	Переопределяется так, чтобы возвращать ограничивающий прямоугольник в координатах элемента; см. также <code>sceneBoundingRect()</code> и <code>shape()</code>
<code>childItems()</code>	Возвращает список непосредственных потомков элемента (Qt 4.4)
<code>collidesWithItem(QGraphicsItem*, Qt::ItemSelectionMode)</code>	Возвращает <code>true</code> , если этот элемент перекрывает переданный элемент в соответствии с указанным режимом; см. перечисление <code>Qt::ItemSelectionMode</code> (стр. 441)

Метод	Описание
<code>collidesWithPath(QPainterPath, Qt::ItemSelectionMode)</code>	Возвращает <code>true</code> , если этот элемент перекрывает переданный путь в соответствии с указанным режимом
<code>collidingItems (Qt::ItemSelectionMode)</code>	Возвращает список всех элементов, перекрывающихся с данным в соответствии с указанным режимом
<code>contains(QPointF)</code>	Возвращает <code>true</code> , если точка находится внутри элемента
<code>ensureVisible()</code>	Если необходимо, производит прокрутку всех представлений <code>QGraphicsView</code> , которые ассоциированы со сценой, содержащей этот элемент, так, чтобы он стал видимым
<code>group()</code>	Возвращает группу <code>QGraphicsItemGroup</code> , которой принадлежит этот элемент, или <code>0</code> , если элемент не принадлежит никакой группе
<code>hide()</code>	Скрывает элемент; см. также <code>show()</code> и <code>setVisible()</code>
<code>isObscuredBy(QGraphicsItem*)</code>	Возвращает <code>true</code> , если ограничивающий прямоугольник элемента полностью перекрыт фигурой переданного непрозрачного элемента
<code>isSelected()</code>	Возвращает <code>true</code> , если элемент выделен; см. также <code>setSelected()</code>
<code>isVisible()</code>	Возвращает <code>true</code> , если элемент логически видимый (даже если он полностью перекрыт или находится вне порта просмотра представления)
<code>keyPressEvent(QKeyEvent*)</code>	Переопределяется для обработки нажатий клавиш при активном данном элементе – вызывается только в случае, когда флаг <code>ItemIsFocusable</code> установлен
<code>mouseDoubleClickEvent (QGraphicsSceneMouseEvent*)</code>	Переопределяется для обработки двойных щелчков мышью

Таблица 11.2. API класса *QGraphicsItem*, избранные методы (часть 2)

Метод	Описание
<code>mouseMoveEvent (QGraphicsSceneMouseEvent*)</code>	Переопределяется для обработки перемещения указателя мыши
<code>mousePressEvent (QGraphicsSceneMouseEvent*)</code>	Переопределяется для обработки нажатия кнопки мыши

Таблица 11.2. (продолжение)

Метод	Описание
<code>moveBy(qreal, qreal)</code>	Перемещает элемент на заданное расстояние по горизонтали и вертикали
<code>opaqueArea()</code>	Переопределяется для возврата графического пути, который охватывает области, где элемент непрозрачен (используется в методе <code>isObscuredBy()</code>)
<code>paint(QPainter*, QStyleOptionGraphicsItem*, QWidget*)</code>	Переопределяется для рисования элемента; см. также <code>boundingRect()</code> и <code>shape()</code>
<code>parentItem()</code>	Возвращает родительский элемент данного элемента или 0
<code>pos()</code>	Возвращает позицию элемента в координатах его родителя или сцены, если у элемента нет родителя; см. <code>scenePos()</code>
<code>prepareGeometryChange()</code>	Этот метод <i>обязательно</i> следует вызывать перед изменением ограничивающего прямоугольника элемента; он автоматически вызывает <code>update()</code>
<code>resetTransform()</code>	Делает матрицу преобразования тождественной, отменяя тем самым поворот, масштабирование или наклон
<code>rotation()</code>	Возвращает угол поворота элемента в градусах ((-360.0°, 360.0°); по умолчанию 0.0° (Qt 4.6))
<code>scale()</code>	Возвращает коэффициент масштабирования элемента; по умолчанию 1.0, то есть без масштабирования (Qt 4.6)
<code>scene()</code>	Возвращает сцену, которой принадлежит элемент, или 0, если элемент еще не помещен на сцену
<code>scenePos()</code>	Возвращает позицию элемента в координатах сцены – для элементов без родителя то же самое, что <code>pos()</code>
<code>setFlag(GraphicsItemFlag, bool)</code>	Устанавливает (по умолчанию) или сбрасывает указанный флаг в соответствии со значением булевого параметра
<code>setFlags(GraphicsItemFlags)</code>	Устанавливает битовую комбинацию флагов; см. перечисление <code>QGraphicsItem::GraphicsItemFlag</code> (стр. 441)

Таблица 11.3. API класса *QGraphicsItem*, избранные методы (часть 3)

Метод	Описание
<code>setGraphicsEffect(QGraphicsEffect*)</code>	Применяет указанный графический эффект к элементу (удаляя предыдущий эффект); доступны следующие эффекты: <code>QGraphicsBlurEffect</code> , <code>QGraphicsDropShadowEffect</code> , <code>QGraphicsOpacityEffect</code> ¹ (Qt 4.6)
<code>setGroup(QGraphicsItemGroup*)</code>	Включает элемент в указанную группу
<code>setParentItem(QGraphicsItem*)</code>	Делает указанный элемент родителем данного
<code>setPos(QPointF)</code>	Устанавливает позицию элемента в координатах родителя; имеется также перегруженный вариант, принимающий два аргумента типа <code>qreal</code>
<code>setRotation(qreal)</code>	Устанавливает угол поворота элемента в градусах (-360.0° , 360.0°) (Qt 4.6)
<code>setScale(qreal)</code>	Масштабирует элемент; 1.0 означает отсутствие масштабирования (Qt 4.6)
<code>setSelected(bool)</code>	Делает элемент выделенным или невыделенным в зависимости от значения булевого аргумента
<code>setToolTip(QString)</code>	Задаёт всплывающую подсказку для элемента
<code>setTransform(QTransform, bool)</code>	Устанавливает новую матрицу преобразования элемента или комбинирует текущую матрицу с переданной, если булевый аргумент равен <code>true</code> (Qt 4.3); существует также совершенно другой метод <code>setTransformations()</code>
<code>setVisible()</code>	Скрывает или показывает элемент в зависимости от значения булевого аргумента
<code>setX(qreal)</code>	Устанавливает абсциссу элемента в координатах его родителя (Qt 4.6)
<code>setY(qreal)</code>	Устанавливает ординату элемента в координатах его родителя (Qt 4.6)
<code>setZValue(qreal)</code>	Устанавливает координату <i>z</i> элемента

¹ А также `QGraphicsColorizeEffect`. Вдобавок вы можете создавать свои собственные эффекты, унаследовав `QGraphicsEffect`. – Прим. науч. ред.

Таблица 11.3. (продолжение)

Метод	Описание
shape()	Переопределяется так, чтобы возвращался графический путь, описывающий точную форму элемента; см. также boundingRect() и paint()
show()	Показывает элемент; см. также hide() и setVisible()
toolTip()	Возвращает всплывающую подсказку элемента
transform()	Возвращает матрицу преобразования элемента; существует также метод transformations()
type()	Возвращает значение QGraphicsItem::Type в виде int; пользовательские подклассы QGraphicsItem обычно переопределяют этот метод и включают определение перечисления Type

Таблица 11.4. API класса QGraphicsItem, избранные методы (часть 4)

Метод	Описание
update()	Помещает запрос на рисование элемента в очередь событий
x()	Возвращает абсциссу элемента в координатах его родителя
y()	Возвращает ординату элемента в координатах его родителя
z()	Возвращает значение координаты z

Таблица 11.5. Перечисление Qt::ItemSelectionMode

Элемент перечисления	Описание
Qt::ContainsItemShape	Выделяет элементы, которые целиком находятся внутри заданной области
Qt::IntersectsItemShape	Выделяет элементы, которые находятся внутри заданной области или пересекают ее границу
Qt::ContainsItemBoundingRect	Выделяет элементы, ограничивающий прямоугольник которых целиком находится внутри заданной области

Элемент перечисления	Описание
Qt::IntersectsItemBoundingRect	Выделяет элементы, ограничивающий прямоугольник которых находится внутри заданной области или пересекает ее границу

Таблица 11.6. Перечисление *QGraphicsItem::GraphicsItemFlag* (часть 1)

Элемент перечисления	Описание
QGraphicsItem::ItemAcceptsInputMethod	Элемент поддерживает методы ввода (Qt 4.6)
QGraphicsItem::ItemClipsChildrenToShape	Элемент отсекает всех своих потомков (рекурсивно), показывая только те их части, которые лежат внутри его контура (Qt 4.3)
QGraphicsItem::ItemClipsToShape	Элемент отсекается собственным контуром вне зависимости от того, как он рисуется, и не может принимать событий (например, щелчков мыши), произошедших вне его контура (Qt 4.3)
QGraphicsItem::ItemDoesntPropagateOpacityToChildren	Элемент не распространяет собственный режим прозрачности на своих потомков (Qt 4.5)
QGraphicsItem::ItemHasNoContents	Элемент ничего не рисует (Qt 4.6)
QGraphicsItem::ItemIgnoresParentOpacity	Прозрачность элемента не комбинируется с прозрачностью его родителя (Qt 4.5)

Таблица 11.7. Перечисление *QGraphicsItem::GraphicsItemFlag* (часть 2)

Элемент перечисления	Описание
QGraphicsItem::ItemIgnoresTransformations	Элемент игнорирует преобразования, примененные к родителю (хотя его позиция по-прежнему привязана к родителю); полезно для элементов, используемых как текстовые метки (Qt 4.3)
QGraphicsItem::ItemIsFocusable	Элемент воспринимает нажатия клавиш
QGraphicsItem::ItemIsMovable	Элемент (и рекурсивно его потомки) можно перетаскивать мышью
QGraphicsItem::ItemIsPanel	Элемент представляет собой панель (Qt 4.6); дополнительные сведения о панелях см. в онлайн-официальной документации

Таблица 11.7. (продолжение)

Элемент перечисления	Описание
<code>QGraphicsItem::ItemIsSelectable</code>	Элемент можно выделить щелчком мыши, охватом эластичным контуром или с помощью вызова <code>QGraphicsScene::setSelectionArea()</code> , если его регион подпадает под действие этого вызова
<code>QGraphicsItem::ItemNegativeZStacksBehindParent</code>	Элемент автоматически оказывается в стеке позади своего родителя, если значение координаты <i>z</i> отрицательно
<code>QGraphicsItem::ItemSendsGeometryChanges</code>	Элемент вызывает метод <code>itemChange()</code> при изменении позиции или преобразовании (Qt 4.6); см. также врезку «Изменения в поведении графических представлений в Qt 4.6» на стр. 426
<code>QGraphicsItem::ItemSendsScenePositionChanges</code>	Элемент вызывает метод <code>itemChange()</code> при изменении позиции (Qt 4.6)
<code>QGraphicsItem::ItemStacksBehindParent</code>	Элемент помещается в стек позади своего родителя, а не впереди (по умолчанию); полезно для создания теней
<code>QGraphicsItem::ItemUsesExtendedStyleOption</code>	Предоставляет элементу доступ к дополнительным атрибутам <code>QStyleOptionGraphicsItem</code>

12

Подсистема Graphics View: создание сцен

- Сцены, элементы и действия
- Расширение класса QGraphicsView
- Создание стыкуемого виджета с набором инструментов
- Создание пользовательских графических элементов

В этой главе мы рассмотрим традиционное приложение, построенное на базе архитектуры графических представлений. Это простая программа рисования, демонстрирующая создание различных специализированных графических элементов и сохранение их в файле определенного нами формата. Мы покажем также, как предоставить пользователю средства для добавления элементов на сцену, изменения их свойств (в том числе свойств групп выделенных элементов) и удаления. В совокупности примеры из этой и предыдущей глав охватывают заметную часть функциональности графических представлений, хотя ни в коем случае не всю. Но, так или иначе, эти две главы закладывают фундамент для дальнейшего самостоятельного изучения и разработки собственных приложений. И как уже было обещано, в главе 13 мы переделаем эти примеры с использованием возможностей, появившихся в версии Qt 4.6.

В этой главе мы создадим конструктор страниц Page Designer (page-designer1). Это самый большой и «полный» из представленных в книге примеров, он насчитывает примерно 3300 строк кода. Но тем не менее

Page Designer – всего лишь упрощенная программа, которой недостает многих полезных функций. Впрочем, для демонстрации возможностей графических представлений ее более чем достаточно. Окно приложения показано на рис. 12.1.

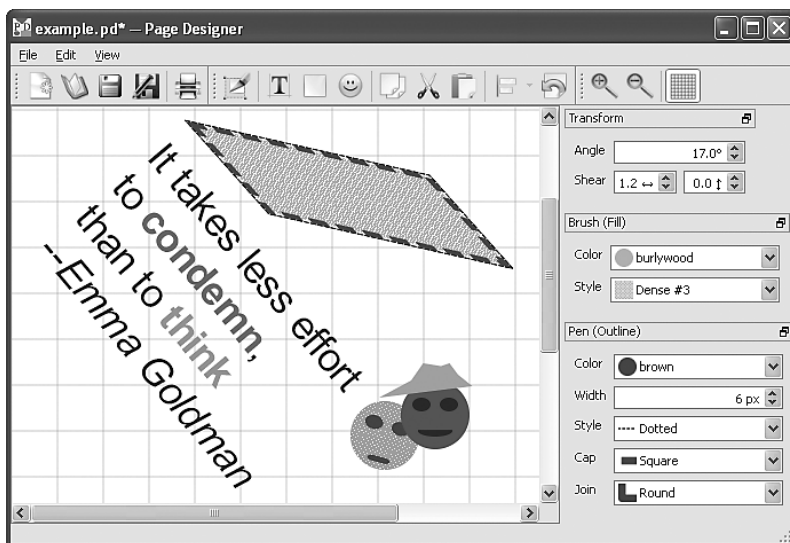


Рис. 12.1. Приложение Page Designer

С точки зрения компоновки Page Designer – стандартное приложение с главным окном и тремя стыкуемыми виджетами, содержащими «наборы инструментов»: для задания параметров преобразования, кисти (заливки) и пера (контура). В меню File (Файл) имеются все стандартные функции: New (Создать), Open... (Открыть...), Save (Сохранить), Save As... (Сохранить как...), Export... (Экспорт...) в растровом формате или в формате SVG, Print... (Печать...) и Quit (Выход). И что особенно важно для программирования графических представлений, имеется меню Edit (Правка) с пунктами Edit Selected Item... (Править выделенный элемент...) (который вызывает слот `edit()` элемента, если таковой имеется, например, для того, чтобы вывести специфическое для элемента контекстное меню или диалоговое окно), Add Text... (Добавить текст...) для добавления элемента с форматированным текстом, Add Box (Добавить блок) для добавления прямоугольника с изменяемыми размерами и Add Smiley (Добавить смайлик) для добавления специального пользовательского элемента непрямоугольной формы. Кроме того, в меню Edit имеются обычные пункты Cut (Вырезать), Copy (Копировать) и Paste (Вставить), а также подменю Alignment (Выравнивание) для выравнивания двух и более выделенных элементов относительно друг друга и удобная команда Clear Transformations (Сбросить преобразования) для обнуления угла поворота и коэффициентов сдвига выделенных элементов. Имеет-

ся также меню View (Вид), с помощью которого пользователь может масштабировать сцену и скрывать либо показывать сетку. Большинство действий доступны не только через меню, но и с помощью кнопок на панели инструментов.

Хотя в этом приложении лишь три вида графических элементов (текст, прямоугольник и смайлики), в совокупности они представляют все, что необходимо программе рисования. Например, код, относящийся к прямоугольникам, может служить образцом для всех прочих стандартных фигур, а код добавления смайликов – образцом для других нестандартных форм.

Если выделен один элемент (например, щелчком мыши), то на панели инструментов отображается его угол поворота, коэффициенты сдвига, параметры пера и кисти. Page Designer поддерживает также некоторые операции над группами выделенных элементов. (Класс `QGraphicsView` поддерживает выделение нескольких элементов щелчком мыши при нажатой клавише `Ctrl` или `⌘` на платформе `Mac OS X`. Разрешается также включать режим охвата эластичным контуром – тогда выделенными оказываются все элементы, которые находятся внутри контура или пересекают его.) Например, все выделенные элементы можно вырезать, скопировать, удалить или выравнивать относительно друг друга. Точно так же, к выделенным элементам применяются преобразования или их отмена и установка параметров кисти и пера.

В этой главе мы рассмотрим сначала общую структуру приложения (включая сохранение и загрузку сцен), чтобы стало понятно, как оно работает, а затем перейдем к детальному обсуждению различных аспектов, относящихся к программированию графических представлений. Начнем с главного окна приложения.

Сцены, элементы и действия

В классе главного окна имеются слоты для каждого действия меню `File`, `Edit` и `View`. В нем также много закрытых вспомогательных методов. Сначала рассмотрим закрытые данные (опуская переменные типа `QAction`).

```
private:
    ...
    TransformWidget *transformWidget;
    BrushWidget *brushWidget;
    PenWidget *penWidget;
    QPainter *printer;
    QGraphicsScene *scene;
    QGraphicsView *view;
    QGraphicsItemGroup *gridGroup;
    QPoint previousPoint;
    int addOffset;
    int pasteOffset;
};
```

Все три пользовательских виджета – это наборы инструментов, входящие в стыкуемых виджетах. Мы сохраняем указатели на них, потому что каждый добавленный на сцену элемент соединяется с соответствующими наборами, чтобы при его выборе на панели инструментов отражались свойства (к примеру, параметры кисти и пера). Один из таких виджетов мы рассмотрим позже.

Мы сохраняем также указатель на `QPrinter`, чтобы при очередном выводе на печать показывались параметры печати, установленные в прошлый раз. На этапе конструирования объекта `QPrinter` для него задаются разумные умолчания – например, формат листа бумаги U.S. Letter в США и A4 в Европе.

Сетку мы решили оформить не в виде фонового изображения, а составить из графических элементов. Сделано это только для того, чтобы продемонстрировать использование класса `QGraphicsItemGroup` и показать, как можно избирательно формировать множество сохраняемых, печатаемых и экспортируемых элементов.

Переменные `previousPoint`, `addOffset` и `pasteOffset` мы обсудим, когда до них дойдет дело.

Строку, содержащую имя файла, мы создавать не стали, вместо этого имя файла сохраняется с помощью метода `setWindowFilePath()`. Этот метод помещает в строку заголовка окна имя файла (при условии, что не использовался метод `setWindowTitle()`) в формате, зависящем от платформы (например, может быть показано только имя без пути), и название приложения. Чтобы было удобно пользоваться этим методом, мы должны предварительно, обычно в функции `main()`, вызвать метод `QApplication::setApplicationName()`. Чтобы получить имя файла, мы вызываем метод `windowFilePath()`, – для новых файлов будет возвращена строка “Unnamed” в предположении, что именно она была задана в методе `fileNew()`.

Далее мы рассмотрим интересные для нас части кода главного окна, начав с конструктора и вызываемых из него вспомогательных методов.

Создание главного окна

Как и в большинстве приложений Qt небольшого и среднего объема, главное окно – сердце всего. И как обычно, именно в конструкторе главного окна описывается внешний вид и поведение пользовательского интерфейса.

```
const int OffsetIncrement = 5;
const QString ShowGrid("ShowGrid");
const QString MostRecentFile("MostRecentFile");

MainWindow::MainWindow(QWidget *parent)
: QMainWindow(parent), gridGroup(0),
  addOffset(OffsetIncrement), pasteOffset(OffsetIncrement)
{
```

```
printer = new QPrinter(QPrinter::HighResolution);
createSceneAndView();
createActions();
createMenusAndToolBars();
createDockWidgets();
createConnections();

QSettings settings;
viewShowGridAction->setChecked(
    settings.value(ShowGrid, true).toBool());
QString filename = settings.value(MostRecentFile).toString();
if (filename.isEmpty() || filename == tr("Unnamed"))
    QTimer::singleShot(0, this, SLOT(fileNew()));
else {
    setWindowFilePath(filename);
    QTimer::singleShot(0, this, SLOT(loadFile()));
}
}
```

Прежде всего конструктор присваивает значения некоторым переменным-членам, которые мы обсудим несколько позже. Затем создается объект-принтер – заданный в нем размер листа бумаги используется для вычисления начального размера окна в переопределенном методе `sizeHint()` (не показан). Дальнейшая настройка пользовательского интерфейса распределена по различным вспомогательным методам, как мы это всегда делаем. Методы `createActions()`, `createMenusAndToolBars()` и `createConnections()` устроены по тому же образцу, что уже много раз встречался, поэтому не будем на них останавливаться. (Разумеется, весь исходный код можно найти в примерах к этой книге, в данном случае в подкаталоге *pagedesigner1*.)

В начальный момент сетка видна или скрыта в зависимости от настроек, действовавших при последнем запуске приложения; при самом первом запуске она по умолчанию показывается. Закрытый слот `viewShowGrid()`, соединенный с сигналом `triggered()` от объекта `viewShowGridAction`, мы обсудим ниже.

В конце конструктора мы получаем имя файла, который приложение редактировало при последнем запуске. Если оно пусто (так как это самый первый запуск) или равно «Unnamed» (потому что страница, редактировавшаяся в последний раз, не была сохранена), то мы вызываем слот `fileNew()` (не показан), чтобы пользователь мог сразу же приступить к рисованию. В противном случае загружаем файл. Как обычно, методы `fileNew()` и `loadFile()` вызываются через однократный таймер, чтобы они начали работать не раньше, чем будет завершено конструирование.

Параметры `ShowGrid` и `MostRecentFile` сохраняются в переопределенном методе `closeEvent()` (не показан).

```
void MainWindow::createSceneAndView()
{
```

```

view = new GraphicsView;
scene = new QGraphicsScene(this);
QSize pageSize = printer->paperSize(QPrinter::Point).toSize();
scene->setSceneRect(0, 0, pageSize.width(), pageSize.height());
view->setScene(scene);
setCentralWidget(view);
}

```

Создание и конфигурирование представления, сцены и центрального виджета приложения не вызывают вопросов. Размер сцены устанавливается пропорциональным размеру листа бумаги путем перевода пунктов (1/72 дюйма) в пиксели. (Метод `paperSize()` возвращает объект типа `QSizeF`, который мы преобразуем в тип `QSize` методом `QSizeF::toSize()`.) Стоит отметить лишь пользовательский класс `GraphicsView` (подкласс `QGraphicsView`), который обеспечивает панорамирование и поддержку колесика мыши. Это небольшой класс мы рассмотрим ниже (стр. 474).

```

void MainWindow::createDockWidgets()
{
    setDockOptions(QMainWindow::AnimatedDocks);
    QDockWidget::DockWidgetFeatures features =
        QDockWidget::DockWidgetMovable|
        QDockWidget::DockWidgetFloatable;
    transformWidget = new TransformWidget;
    QDockWidget *transformDockWidget = new QDockWidget(
        tr("Transform"), this);
    transformDockWidget->setFeatures(features);
    transformDockWidget->setWidget(transformWidget);
    addDockWidget(Qt::RightDockWidgetArea, transformDockWidget);
    ...
}

```

В этом методе создаются и помещаются в стыкуемые виджеты все три набора инструментов. Мы показали только первый, потому остальные создаются точно так же.

Сначала задаются параметры стыковки. Значение `QMainWindow::AnimatedDocks` обеспечивает то, что когда пользователь перетаскивает стыкуемый виджет по области дока, док создает пустое пространство, показывая, где окажется виджет, если его бросить, отодвигая при необходимости остальные пристыкованные виджеты. Это помогает пользователю лучше ориентироваться в происходящем.

По умолчанию устанавливаются параметры стыковки `QMainWindow::AnimatedDocks|QMainWindow::AllowTabbedDocks`, поэтому, задавая только первый флаг, мы отключаем второй. Если включен режим доков с вкладками, то пользователь может бросить стыкуемый виджет *между* другими (обычное поведение) или *на* другой стыкуемый виджет, и тогда Qt поместит брошенный виджет и тот, на который он брошен, во вкладки, так что виден будет только брошенный виджет. Затем пользователь может переключаться между виджетами щелчком по вкладке. По умолчанию

вкладки располагаются в нижней части дока, но это можно изменить с помощью метода `QMainWindow::setTabPosition()`. Понятно, что эта возможность полезна, когда число стыкуемых виджетов велико или отдельные виджеты занимают много места.

Есть еще два флага стыковки (взаимно исключающие). Флаг `QMainWindow::ForceTabbedDocks` принудительно располагает все стыкуемые виджеты во вкладках; недостаток этого режима в том, что одновременно в доке виден только один виджет. Флаг `QMainWindow::AllowNestedDocks` позволяет разбить док на части, организовав в нем больше одной строки (или столбца). Недостаток в том, что пользователю труднее работать с таким интерфейсом. По возможности обоих этих флагов следует избегать.

Для каждого стыкуемого виджета задаются два свойства: `QDockWidget::DockWidgetMovable` означает, что пользователь может перетаскивать виджет из одного дока в другой, а `QDockWidget::DockWidgetFloatable` – что можно выдвигать виджеты из дока и делать их плавающими. Если мы хотим разрешить пользователю закрывать (то есть скрывать) стыкуемые виджеты, то нужно еще задать свойство `QDockWidget::DockWidgetClosable`. Поддерживается также свойство `QDockWidget::DockWidgetVerticalTitleBar`. Оно позволяет уменьшить занимаемое стыкуемыми виджетами место по вертикали, что особенно полезно для виджетов, пристыкованных к левому или правому краю. На рис. 12.2 показан виджет с вертикальной строкой заголовка и плавающий виджет, выдвинутый из дока.

Мы опускаем стандартный код создания действий, меню и панели инструментов главного окна, но в последующих подразделах ознакомимся с методами, которые реализуют высокоуровневое поведение, относящееся к графическим представлениям. И начнем с сохранения и загрузки сцен.

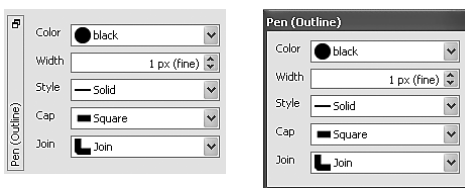


Рис. 12.2. Программа Page Designer: стыкуемый виджет с вертикальной строкой заголовка и плавающий

Сохранение, загрузка, печать и экспорт сцен

В приложении Page Designer имеются стандартные действия `Save`, `Open`, `Print...` и `Export...` и соответствующие им слоты `fileSave()`, `fileOpen()`, `filePrint()` и `fileExport()`. Описание инфраструктуры сохранения и загрузки мы опустим (поскольку она очень похожа на то, что мы видели раньше) и сосредоточимся исключительно на методах, касающихся программирования графических представлений.

Сохранение сцены

В слоте `fileSave()` приложение открывает в двоичном режиме объект `QFile` и поток `QDataStream` для записи в файл. В поток данных записываются магическое число и номер версии формата (целые числа), а затем номер версии сериализатора – в данном случае Qt 4.5. Это означает, что приложение нельзя будет собрать с более ранней версией Qt, но в любой версии Qt 4.x (где $x \geq 5$) данные можно будет прочитать. Затем вызывается метод `MainWindow::writeItems()` с двумя аргументами: поток данных и список всех элементов на сцене.

Метод `QGraphicsScene::items()` возвращает список всех графических элементов на сцене, который передается в качестве второго аргумента. У него есть несколько перегруженных вариантов: принимающий порядок сортировки, возвращающий только элементы, которые содержат указанную точку, и возвращающий только элементы, которые находятся внутри либо пересекают границу указанного прямоугольника, многоугольника или графического пути. В нашем случае в состав возвращенных элементов входят и те, что составляют сетку, хотя сохранять их мы не хотим. Но, как мы скоро увидим, это не проблема.

```
void MainWindow::writeItems(QDataStream &out,
                           const QList<QGraphicsItem*> &items)
{
    foreach (QGraphicsItem *item, items) {
        if (item == gridGroup || item->group() == gridGroup)
            continue;
        qint32 type = static_cast<qint32>(item->type());
        out << type;
        switch (type) {
            case BoxItemType:
                out << *static_cast<BoxItem*>(item); break;
            case SmileyItemType:
                out << *static_cast<SmileyItem*>(item); break;
            case TextItemType:
                out << *static_cast<TextItem*>(item); break;
            default: Q_ASSERT(false);
        }
    }
}
```

Этот метод обходит переданный ему список графических элементов. Для каждого элемента проверяется, принадлежит ли он группе `gridGroup` (типа `QGraphicsItemGroup*`), в которую входят элементы, образующие сетку, и если да, то элемент пропускается, так как мы не хотим сохранять сетку. Для всех остальных элементов сначала выводится значение `QGraphicsItem::Type` (уникальная константа, определяющая тип графического элемента и возвращаемая переопределенным методом `QGraphicsItem::type()`), а затем сам элемент. Метод главного окна `writeItems()` не предназначен для сохранения графических элементов, он просто по-

лагается на предоставленный ими метод `operator<<()`, который выводит данные в объект `QDataStream` (разумеется, такой метод определен в классе каждого графического элемента, как мы скоро увидим).

Дополнительный плюс вынесения метода `writeItems()` за пределы `fileSave()` состоит в том, что его можно будет использовать повторно для вырезания и вставки, но это тема следующего подраздела.

С помощью метода `writeItems()` мы сохранили сцену максимально простым способом – в виде последовательности графических элементов. В более сложном приложении можно было бы сохранить не только сами элементы, но и метаданные. Теперь, разобравшись с сохранением сцены, посмотрим, как она загружается, а затем перейдем к печати и экспорту сцены в стандартных растровых и векторных форматах.

Загрузка сцены

В методе `fileOpen()` программа просит пользователя указать имя файла, а затем создает для него объект `QFile`. Далее для чтения содержимого файла создается объект `QDataStream` и считываются магическое число и номер версии формата. При обнаружении ошибки выводится предупреждение. В противном случае устанавливается версия десериализатора Qt 4.5, соответствующая указанной при сохранении; все элементы, присутствующие в данный момент на сцене, стираются, и сцена заполняется элементами, прочитанными из файла.

К сожалению, метод `readItems()` (как и рассмотренный выше метод `writeItems()`, и `editAddItem()`, который нам еще предстоит обсудить) оперирует только известными типами. Это означает, что при добавлении новых типов все три метода придется модифицировать. Увы, элегантного решения этой проблемы не найдено. С другой стороны, все остальные методы пользуются системой свойств Qt (так как все наши классы пользовательских элементов наследуют `QObject`) и, значит, могут работать без изменения – правильно работать! – даже после добавления новых типов элементов, при условии, конечно, что новые типы тоже наследуют `QObject`.

```
void MainWindow::clear()
{
    scene->clear();
    gridGroup = 0;
    viewShowGrid(viewShowGridAction->isChecked());
}
```

Метод `QGraphicsScene::clear()` удаляет все графические элементы – включая и образующие сетку, – поэтому нужно не забыть обнулить `gridGroup`, так как этого объекта больше не существует. Разумеется, необходимости в этой группе не возникло бы, если бы мы рисовали сетку фоновой кистью или переопределили метод `QGraphicsView::drawBackground()`, – именно так мы и поступили бы в реальной программе. (В приложении Page Designer мы применили для формирования фона класс `QGraphics-`

ItemGroup только с целью его демонстрации и чтобы показать, как можно различать элементы.) Затем мы вызываем метод `viewShowGrid()`, который заново создает сетку, делая ее видимой или скрытой в зависимости от состояния `viewShowGridAction`.

```
void MainWindow::readItems(QDataStream &in, int offset, bool select)
{
    QSet<QGraphicsItem*> items;
    qint32 itemType;
    QGraphicsItem *item = 0;
    while (!in.atEnd()) {
        in >> itemType;
        switch (itemType) {
            case BoxItemType: {
                BoxItem *boxItem = new BoxItem(QRect(), scene);
                in >> *boxItem;
                connectItem(boxItem);
                item = boxItem;
                break;
            }
            ...
        }
        if (item) {
            item->moveBy(offset, offset);
            if (select)
                items << item;
            item = 0;
        }
    }
    if (select)
        selectItems(items);
    else
        selectionChanged();
}
```

Этот метод читает элементы сцены из потока `QDataStream`, открытого для файла. Он же применяется для вставки скопированных или вырезанных элементов на сцену, но в этом случае поток будет открыт не для файла, а для массива `QByteArray`, полученного из буфера обмена. Аргументы `offset` (по умолчанию равен 0) и `select` (по умолчанию равен false) используются только при вставке.

Метод читает поток данных, пока не достигнет конца. Каждый элемент читается в два приема: сначала его тип `QGraphicsItem::Type`, затем сам элемент. Код чтения смайлика и текста опущен, поскольку он отличается от чтения прямоугольников только тем, что первым аргументом конструктору передается объект `QPoint()`, а не `QRect()`. Все элементы необходимо соединить с соответствующими виджетами из набора инструментов; это делает рассматриваемый далее метод `connectItem()`. Как и в случае `writeItems()`, метод `readItems()` не предназначен для чтения пользовательских

типов элементов, а полагается на переопределенный в соответствующем классе метод `operator>>()`, который оперирует потоком `QDataStream`.

При вставке элементов аргумент `offset` будет отличен от нуля, а `select` равен `true`. Это означает, что элемент сдвигается относительно своей текущей позиции и добавляется к набору элементов `items`. В конце метода, если `select` равно `true`, все элементы набора `items` выделяются, в противном случае мы вызываем метод `selectionChanged()`, который позаботится о том, чтобы в виджетах наборов инструментов были отражены свойства выделенного элемента, поскольку последний прочитанный элемент делает себя выделенным в конструкторе.

```
void MainWindow::connectItem(QObject *item)
{
    connect(item, SIGNAL(dirty()), this, SLOT(setDirty()));
    const QMetaObject *metaObject = item->metaObject();
    if (metaObject->indexOfProperty("brush") > -1)
        connect(brushWidget, SIGNAL(brushChanged(const QBrush&)),
                item, SLOT(setBrush(const QBrush&)));
    if (metaObject->indexOfProperty("pen") > -1)
        connect(penWidget, SIGNAL(penChanged(const QPen&)),
                item, SLOT(setPen(const QPen&)));
    if (metaObject->indexOfProperty("angle") > -1) {
        connect(transformWidget, SIGNAL(angleChanged(double)),
                item, SLOT(setAngle(double)));
        connect(transformWidget, SIGNAL(shearChanged(double, double)),
                item, SLOT(setShear(double, double)));
    }
}
```

Этот метод вызывается из метода `readItems()` (после того как элемент прочитан из файла сцены или вставлен на сцену из буфера обмена) при добавлении на сцену нового элемента. Предполагается, что классы элементов наследуют `QObject`, то есть мы можем пользоваться системой свойств `Qt` для выяснения того, какие именно соединения следует устанавливать. Недостаток такого подхода в том, что пользовательские типы элементов должны наследовать `QObject`, но зато нам нет необходимости заранее знать, объект какого конкретно типа соединяется, — и в результате при добавлении произвольных новых типов элементов этот метод не изменится вовсе.

Мы приняли соглашение, что в приложении `Page Designer` все элементы — за исключением образующих сетку — являются объектами подклассов `QObject` и генерируют сигнал `dirty()`, когда происходит значимое изменение состояния.

Для каждого интересующего нас свойства мы спрашиваем у метаобъекта элемента, обладает ли этот элемент (точнее, элемент того подкласса `QObject`, с которым ассоциирован метаобъект) данным свойством. (Если индекс свойства равен `-1`, значит, такого свойства нет.)

У элементов-прямоугольников имеются свойства для кисти, пера, угла поворота и коэффициентов сдвига по горизонтали и вертикали, поэтому для них устанавливаются все соединения. В программе Page Designer мы приняли соглашение, что если у элемента есть свойство `angle`, то его можно сдвигать (скашивать). При желании эти свойства можно было бы легко развести и, наоборот, ввести соглашение о том, что всякий объект, имеющий свойство `pen`, имеет также свойство `brush`, поэтому их можно не проверять по отдельности.

Благодаря установленным соединениям при любом изменении свойства с помощью набора инструментов (например, смене кисти) данный прямоугольник (а на самом деле все элементы, соединенные с сигналом `brushChanged()` виджета `BrushWidget`) будет уведомлен об изменении. А как мы увидим при рассмотрении класса элемента-прямоугольника, на это уведомление реагируют только выделенные элементы.

Соединения можно было бы установить и в методе `readItems()`, но мы вынесли эту операцию в отдельный метод, поскольку ее придется выполнять еще и при добавлении нового элемента на сцену.

Мы видели, что в конце метода `readItems()` проверяется, нужно ли выделять прочитанные элементы (это так, когда элементы читаются из буфера обмена, а не из файла). Если да, то вызывается метод `selectItems()`.

```
void MainWindow::selectItems(const QSet<QGraphicsItem*> &items)
{
    scene->clearSelection();
    foreach (QGraphicsItem *item, items)
        item->setSelected(true);
    selectionChanged();
}
```

Этот метод сначала снимает выделение со всех элементов на сцене, а затем проверяет, какие элементы нужно выделить, и выделяет их. Далее вызывается метод `selectionChanged()`, чтобы обновить содержимое виджетов из набора инструментов в соответствии со свойствами одного из выделенных элементов.

Печать и экспорт сцены

Помимо сохранения и загрузки, мы еще можем печатать и экспортировать сцены – в векторном формате (SVG) и в любом из поддерживаемых Qt растровых форматов. Чтобы напечатать или экспортировать сцену, необходимо создать подходящий объект `QPainter`, временно убрать сетку и вывести сцену на поверхность рисовальщика.

```
const int StatusTimeout = AQP::MSecPerSecond * 30;

void MainWindow::filePrint()
{
    QPrintDialog dialog(printer);
    if (dialog.exec()) {
```

```

    {
        QPainter painter(printer);
        paintScene(&painter);
    }
    statusBar()->showMessage(tr("Printed %1")
        .arg(windowFilePath()), StatusTimeout);
}

```

Диалоговое окно позволяет пользователю изменить различные параметры печати, а поскольку мы передаем ему указатель на объект `QPrinter`, то по умолчанию устанавливаются параметры, хранящиеся в этом объекте. Если пользователь нажимает кнопку ОК, то мы создаем рисовальщик, который будет рисовать на принтере (то есть на печатаемой странице), и рисуем сцену методом `paintScene()`.

Не вполне обычной особенностью этого и других методов, вызывающих `paintScene()`, является то, что мы создаем рисовальщик и рисуем сцену внутри локальной области видимости (то есть блока, ограниченного фигурными скобками). Это просто удобный прием, гарантирующий, что рисовальщик будет уничтожен сразу после выхода из блока, поскольку захваченные им ресурсы освобождаются в деструкторе. Альтернатива, не требующая использования локальной области видимости, – вызвать метод `QPainter::end()` по завершении рисования.

Строка, возвращаемая методом `windowFilePath()`, – это имя файла (вместе с путем), которое мы сохранили при обращении к методу `setWindowFilePath()`; если файл еще не сохранялся, то мы получим строку “Unnamed” в предположении, что именно эта строка была задана в качестве пути к файлу в методе `fileNew()` (а так оно, конечно, и есть).

```

void MainWindow::paintScene(QPainter *painter)
{
    bool showGrid = viewShowGridAction->isChecked();
    if (showGrid)
        viewShowGrid(false);
    QList<QGraphicsItem*> items = scene->selectedItems();
    scene->clearSelection();
    scene->render(painter);
    if (showGrid)
        viewShowGrid(true);
    foreach (QGraphicsItem *item, items)
        item->setSelected(true);
    selectionChanged();
}

```

В нарисованной сцене мы не хотим видеть ни сетку, ни прямоугольники выделения. Для этого мы скрываем сетку (с помощью метода `viewShowGrid()`), затем запоминаем, какие элементы выделены, и снимаем выделение.

Закончив подготовительную работу, мы выводим сцену на поверхность рисовальщика. Методу `QGraphicsScene::render()` дополнительно можно

передать целевой прямоугольник (чтобы рисовать только на части существующего изображения) и исходный прямоугольник (чтобы рисовать только часть сцены). И если заданы и конечный, и начальный прямоугольники, то можно передать еще один необязательный аргумент типа `Qt::AspectRatioMode`, что бывает полезно, если размеры этих прямоугольников различаются.

Завершив рисование, мы включаем отображение сетки, если она была видна ранее, и заново выделяем те элементы, которые были выделены в самом начале, так что сцена восстанавливается в состоянии, предшествующем печати или экспорту. Кроме того, вызывается метод `selectionChanged()`, чтобы в виджетах из набора инструментов были показаны свойства одного из выделенных элементов.

Процедура настройки рисовальщика при экспорте очень похожа на ту, что мы видели в главе 10, когда говорили об экспорте документов `QTextDocument` (стр. 411). Для полноты картины приведем оба метода экспорта, и тот и другой основаны на только что рассмотренном методе `paintScene()`. В обоих случаях используется один и тот же слот `fileExport()`, который открывает диалоговое окно и выбирает метод экспорта в зависимости от суффикса файла, указанного пользователем.

```
void MainWindow::exportSvg(const QString &filename)
{
    QSvgGenerator svg;
    svg.setFileName(filename);
    svg.setSize(printer->paperSize(QPrinter::Point).toSize());
    {
        QPainter painter(&svg);
        paintScene(&painter);
    }
    statusBar()->showMessage(tr("Exported %1").arg(filename),
        StatusTimeout);
}
```

Чтобы воспользоваться встроенной в Qt функциональностью, касающейся формата SVG, в *pro*-файл программы нужно поместить директиву `QT += svg`. Сначала мы создаем объект генератора SVG и передаем ему имя файла, в который собираемся записывать сцену. Размер страницы в нем мы устанавливаем равным размеру страницы *Page Designer*, затем создаем рисовальщик, который будет выводить данные в генератор SVG, после чего вызываем вспомогательный метод `paintScene()`, который и делает всю работу. Можно было бы применить аналогичный подход для вывода сцены в *pdf*-файл; нужно лишь в качестве устройства рисования взять объект `QPrinter`, как мы делали в главе 10. Оставляем это читателю в качестве упражнения.

```
void MainWindow::exportImage(const QString &filename)
{
    QImage image(printer->paperSize(QPrinter::Point).toSize(),
        QImage::Format_ARGB32);
```

```

    {
        QPainter painter(&image);
        painter.setRenderHints(QPainter::Antialiasing|
                               QPainter::TextAntialiasing);
        paintScene(&painter);
    }
    if (image.save(filename))
        statusBar()->showMessage(tr("Exported %1").arg(filename),
                                StatusTimeout);
    else
        AQP::warning(this, tr("Error"), tr("Failed to export: %1")
                     .arg(filename));
}

```

Экспорт сцены в виде растрового изображения работает почти так же, как экспорт в формате SVG, только нужно включить режим сглаживания (потому что мы, как правило, включаем сглаживание, когда выводим рисунок на экран или готовим растровые изображения, и выключаем в случае вывода на принтер и для векторных изображений). Кроме того, поскольку метод `QImage::save()` возвращает булевый признак успеха, то мы не преминули им воспользоваться.

Теперь, научившись сохранять, загружать, печатать и экспортировать сцены, мы готовы разобраться с тем, как на сцену добавляются элементы и как этими элементами манипулировать, например копировать, вырезать, вставлять и выравнивать.

Манипулирование графическими элементами

Программа Page Designer поддерживает добавление элементов, манипулирование индивидуальными элементами (то есть одним выделенным элементом), а для некоторых операций – также манипулирование всеми выделенными элементами. Например, если пользователь активирует действие `Edit Selected Item...` (Править выделенный элемент...), то будет вызван соответствующий слот.

```

void MainWindow::editSelectedItem()
{
    QList<QGraphicsItem*> items = scene->selectedItems();
    if (items.count() != 1)
        return;
    if (QObject *item = dynamic_cast<QObject*>(items.at(0))) {
        const QMetaObject *metaObject = item->metaObject();
        metaObject->invokeMethod(item, "edit", Qt::DirectConnection);
    }
}

```

Мы разрешаем редактировать только один элемент, поэтому сначала проверяем, что ровно один элемент и выделен. Если это так, то получаем ассоциированный с элементом метаобъект и пытаемся вызвать слот элемента `edit()`. Метод `QMetaObject::invokeMethod()` возвращает `true`,

если слот с указанным именем существует и его можно вызвать; в противном случае метод ничего не делает и возвращает `false`. Аргумент `Qt::DirectConnection` сообщает Qt, что слот нужно вызвать немедленно, не дожидаясь, пока до него дойдет очередь в цикле обработки сообщений. Мы обсуждали метод `invokeMethod()` ранее, на стр. 154. Таким образом, методу `editSelectedItem()` не нужно ничего знать о типе выделенного элемента, он будет правильно работать даже после добавления новых типов элементов и вне зависимости от того, определен в них слот `edit()` или нет. Главное, чтобы они наследовали классу `QObject`.

В приложении Page Designer в классе `BoxItem` слот `edit()` отсутствует, поэтому для таких элементов этот метод ничего не делает. Однако в классах `SmileyItem` и `TextItem` этот слот есть, причем для смайликов он открывает контекстное меню, а для текстов – диалоговое окно. Таким образом, если бы мы разрешили одновременно редактировать более одного элемента, то на экране могли бы оказаться десятки и даже сотни открытых меню и диалоговых окон.

Применение оператора `dynamic_cast<>()` здесь (и в других местах) означает, что программа Page Designer зависит от механизма RTTI (динамической идентификации типа данных). Большинство современных компиляторов для настольных операционных систем поддерживают этот механизм и включают его по умолчанию. Но в компиляторах, ориентированных на встраиваемые системы, поддержка RTTI может быть отключена и даже вообще отсутствовать. В качестве альтернативы можно было бы рассмотреть функцию `qobject_cast<>()`, которая не зависит от RTTI, но, к сожалению, в этом конкретном случае она работать не будет, потому что не принимает указатели на `QGraphicsItem`. Тем не менее есть возможность заставить `qobject_cast<>()` работать. Надо лишь, чтобы все наши графические элементы были произведены от класса `QGraphicsObject`, и тогда приведение можно было бы записать в виде `QObject *object = qobject_cast<QObject*>(item->toGraphicsObject())`, по крайней мере в версии Qt 4.6, где был введен метод `QGraphicsItem::toGraphicsObject()`. В случае Page Designer это будет работать для смайликов и текстовых элементов, поскольку и те и другие наследуют `QGraphicsObject`, но не для прямоугольников, хотя и в этом случае имеется обходной путь, который мы опишем чуть ниже.

Есть также решение, позволяющее обойтись как без `dynamic_cast<>()`, так и без системы метаобъектов Qt (например, если мы не хотим, чтобы все наши типы элементов были подклассами `QObject`), – воспользоваться функцией `qgraphicsitem_cast<>()`. Минус здесь в том, что мы должны зашивать в код конкретные типы элементов, правда, это все равно приходится делать при записи, чтении и добавлении элементов. Чтобы воспользоваться функцией `qgraphicsitem_cast<>()`, нужно заменить второй оператор `if` и связанный с ним блок в методе `editSelectedItem()` такими строками:

```

QGraphicsItem *item = items.at(0);
if (TextItem *textItem = qgraphicsitem_cast<TextItem*>(item))
    textItem->edit();
else if (SmileyItem *smileyItem =
    qgraphicsitem_cast<SmileyItem*>(item))
    smileyItem->edit();

```

Здесь мы получаем выделенный элемент в виде указателя на `QGraphicsItem`, а не на `QObject`. Отметим, что мы не упоминаем элементы `BoxItem`, так как у них нет слота `edit()`, и стало быть, их можно спокойно игнорировать. Если включить слот `edit()` в класс `BoxItem`, то нужно было бы добавить еще одну ветвь `else if` в показанный выше код, — тогда как в версию, где используется система метаобъектов Qt, никаких изменений вносить не потребуется.

В версии `pagedesigner2` (которая требует Qt 4.6 и обсуждается в главе 13) мы вообще не используем оператор `dynamic_cast<>()`. Вместо этого мы обращаемся к функции `QObjectFrom()`, которая возвращает указатель на `QObject`, получив указатель на `QGraphicsItem`.

```

QObject *QObjectFrom(QGraphicsItem *item)
{
    if (!item)
        return 0;
    // Типы, не наследующие QGraphicsObject, следует обрабатывать
    // явно
    if (item->type() == BoxItemType)
        return qobject_cast<QObject*>(static_cast<BoxItem*>(item));
    // Типы, наследующие QGraphicsObject, можно обработать
    // обобщенно
    return item->toGraphicsObject();
}

```

Те элементы, которые не наследуют `QGraphicsObject`, необходимо явно привести к их фактическому типу, производному от `QGraphicsItem` (он должен наследовать также `QObject`), который затем привести к `QObject`. А для типов, производных от `QGraphicsObject`, мы можем просто воспользоваться методом `QGraphicsItem::toGraphicsObject()`, который даст нужный нам указатель на `QObject`.

Имея эту функцию, мы можем в версии Page Designer для Qt 4.6 (с помощью директивы `#ifdef`) заменить все вхождения `dynamic_cast<QObject*>(item)` на `QObjectFrom(item)`. Недостаток этого подхода в том, что при добавлении новых (производных от `QObject`) типов элементов нужно будет явно модифицировать функцию `QObjectFrom()`, добавив типы, не являющиеся подклассами `QGraphicsObject`, тогда как при использовании `dynamic_cast<>()` никаких изменений вносить не потребовалось бы.

Добавление элементов

В пользовательском интерфейсе приложения предусмотрены отдельные действия для добавления поддерживаемых типов элементов. У каждого такого действия в качестве пользовательских данных хранится тип `Type` элемента, и все они соединены с одним и тем же слотом `editAddItem()`.

```
void MainWindow::editAddItem()
{
    QAction *action = qobject_cast<QAction*>(sender());
    if (!action)
        return;
    QObject *item = 0;
    int type = action->data().toInt();
    if (type == BoxItemType)
        item = new BoxItem(QRect(position(), QSize(90, 30)), scene);
    else if (type == SmileyItemType)
        item = new SmileyItem(position(), scene);
    else if (type == TextItemType) {
        TextItemDialog dialog(0, position(), scene, this);
        if (dialog.exec())
            item = dialog.textItem();
    }
    if (item) {
        connectItem(item);
        setDirty(true);
    }
}
```

Этот слот вызывается, когда пользователь хочет добавить какой-нибудь элемент. (Метод `QObject::sender()` и альтернативы ему обсуждались выше, на стр. 379.) Если был запрошен прямоугольник, то ему назначается размер по умолчанию и позиция, возвращаемая методом `position()` (см. ниже). В случае текстового элемента мы решили не ограничиваться умолчаниями, как для прямоугольников и смайликов, а открыть диалоговое окно, в котором пользователь может сразу же ввести текст. Класс `TextItemDialog` (его код опущен, но результат работы представлен на рис. 12.3, стр. 486) – это типичное диалоговое окно для создания/редактирования текстового элемента. В качестве первого аргумента конструктору передается редактируемый элемент (в случае, когда конструктор вызван из слота `edit()` элемента) или 0 (как здесь), если создается новый элемент. Когда пользователь нажимает кнопку ОК, мы получаем вновь созданный объект `TextItem`, обращаясь к методу `TextItemDialog::textItem()`.

Создав запрошенный элемент, мы соединяем его с соответствующими виджетами из наборов инструментов, чтобы пользователь мог изменять свойства элемента, – для прямоугольника это параметры кисти, пера и преобразований. Метод `connectItem()` был рассмотрен выше (стр. 453). Ну и, конечно, нужно вызвать метод `setDirty()`, поскольку добавление нового элемента – это существенное изменение сцены.

```
const int OffsetIncrement = 5;

QPoint MainWindow::position()
{
    QPoint point = mapFromGlobal(QCursor::pos());
    if (!view->geometry().contains(point)) {
        point = previousPoint.isNull()
            ? view->pos() + QPoint(10, 10) : previousPoint;
    }
    if (!previousPoint.isNull() && point == previousPoint) {
        point += QPoint(addOffset, addOffset);
        addOffset += OffsetIncrement;
    }
    else {
        addOffset = OffsetIncrement;
        previousPoint = point;
    }
    return view->mapToScene(point - view->pos()).toPoint();
}
```

Этот метод вычисляет подходящую позицию для нового элемента. Сначала создается объект `QPoint`, соответствующий положению указателя мыши. Если его позиция оказывается вне порта просмотра, то мы выбираем точку внутри порта или предыдущую точку, в которую был помещен новый элемент, если таковая имеется. Но если выбранная точка совпадает с предыдущей, в которую был помещен новый элемент, то мы немного сдвигаем ее вправо и вниз, а также увеличиваем величину сдвига для следующей вставки. В противном случае – если новая точка не совпадает с предыдущей – мы восстанавливаем исходную величину сдвига и делаем «предыдущей» новую точку. По завершении вычислений метод возвращает позицию новой точки в координатах сцены. Следовательно, если добавить несколько элементов подряд, то каждый последующий будет находиться чуть правее и ниже предыдущего, а не прямо поверх него. Так пользователю будет проще понять, где что находится.

Разобравшись с тем, как вызывается слот `edit()` для редактирования элемента (если таковой существует) и как добавляются новые элементы, посмотрим, какую поддержку приложение `Page Designer` предлагает для копирования, вырезания и вставки.

Копирование, вырезание и вставка элементов

Для поддержки копирования, вырезания и вставки используется системный буфер обмена. Преимущество такого подхода в том, что другие приложения могут получить доступ к нашим скопированным или вырезанным элементам (если они, конечно, распознают наш формат данных), и наоборот – `Page Designer` может помещать на сцену элементы, созданные в других приложениях (при условии распознавания их

формата). Но есть и недостаток – если вырезанные или скопированные нами элементы находятся в буфере обмена (а не, скажем, в закрытой переменной-члене типа `QByteArray`), а пользователь перейдет в другое приложение и там выполнит операцию копирования или вырезания, то наши элементы будут удалены из буфера. Решение простое – поддерживать отмену и повтор операций. Здесь мы этого делать не будем, но вернемся к этому вопросу в конце главы.

В дополнение к буферу обмена мы еще создали закрытую переменную-член `pasteOffset` (типа `int`), предназначенную для того, чтобы элементы вставлялись с небольшим смещением от исходного положения, иначе пользователь может не понять, что вставка произошла.

```
void MainWindow::editCopy()
{
    QList<QGraphicsItem*> items = scene->selectedItems();
    if (items.isEmpty())
        return;
    pasteOffset = OffsetIncrement;
    copyItems(items);
    updateUi();
}
```

Когда пользователь активирует действие `Copy` (Копировать), этот слот получает список выделенных элементов, восстанавливает первоначальное значение смещения (5 пикселей) и вызывает вспомогательный метод `copyItems()`, который копирует выделенные элементы в буфер обмена. Впоследствии эти элементы можно будет снова вставить, но если в этом же или в другом приложении будет произведена новая операция копирования или вырезания либо пользователь выйдет из программы, не выполнив вставку, то находящиеся в буфере обмена элементы будут потеряны (затерты или удалены). В заключение вызывается метод `updateUi()`, чтобы активировать действие `Paste`.

```
const QString MimeType = "application/vnd.qtrac.pagedesigner";

void MainWindow::copyItems(const QList<QGraphicsItem*> &items)
{
    QByteArray copiedItems;
    QDataStream out(&copiedItems, QIODevice::WriteOnly);
    writeItems(out, items);
    QMimeData *mimeTypeData = new QMimeData;
    mimeTypeData->setData(MimeType, copiedItems);
    QClipboard *clipboard = QApplication::clipboard();
    clipboard->setMimeData(mimeTypeData);
}
```

Этот метод создает пустой массив `QByteArray` и с помощью метода `writeItems()` помещает в него подлежащие копированию элементы. Затем он создает в куче объект `QMimeData` и записывает в него этот массив, поме-

тив его нестандартным MIME-типом, описывающим формат данных. Затем мы получаем указатель на системный буфер обмена и помещаем в него данные. После этого владение данными принимает на себя буфер обмена, так что мы можем не думать об их удалении.

```
void MainWindow::editCut()
{
    QList<QGraphicsItem*> items = scene->selectedItems();
    if (items.isEmpty())
        return;
    copyItems(items);
    QListIterator<QGraphicsItem*> i(items);
    while (i.hasNext()) {
        QScopedPointer<QGraphicsItem> item(i.next());
        scene->removeItem(item.data());
    }
    setDirty(true);
}
```

Вырезание элементов – чуть более сложная операция, чем копирование, потому что данные нужно еще и удалить. Метод начинается так же, как `editCopy()`, но после копирования поочередно убирает и уничтожает все скопированные в буфер обмена элементы.

Метод `QGraphicsScene::removeItem()` убирает со сцены переданный ему элемент (и рекурсивно всех его потомков) и передает владение ими вызывающей программе. В данном случае, чтобы избежать утечек памяти, мы удаляем убранные элементы (и их потомков), как только ограниченный указатель выходит из области видимости, то есть в конце каждой итерации цикла¹. В заключение вызывается метод `setDirty()`, потому что вырезание элементов (в отличие от копирования) означает изменение сцены. (Метод `setDirty()` вызывает слот `updateUi()`, поэтому после вырезания становится доступно действие **Paste (Вставить)**.)

```
void MainWindow::editPaste()
{
    QClipboard *clipboard = QApplication::clipboard();
    const QMimeData *mimeData = clipboard->mimeData();
    if (!mimeData)
        return;
    if (mimeData->hasFormat(MimeType)) {
        QByteArray copiedItems = mimeData->data(MimeType);
        QDataStream in(&copiedItems, QIODevice::ReadOnly);
        readItems(in, pasteOffset, true);
        pasteOffset += OffsetIncrement;
    }
}
```

¹ В исходном коде имеется директива `#if QT_VERSION`, благодаря которой программа будет компилироваться и в версии Qt 4.5, где вместо `QScopedPointer` используется обычный указатель `QGraphicsItem*` и для удаления убранного со сцены элемента явно вызывается оператор `delete`.

```

    }
    else if (mimeType->hasHtml() || mimeType->hasText()) {
        TextItem *textItem = new TextItem(position(), scene);
        connectItem(textItem);
        if (mimeType->hasHtml())
            textItem->setHtml(mimeType->html());
        else
            textItem->setPlainText(mimeType->text());
    }
    else
        return;
    setDirty(true);
}

```

Если пользователь скопировал или вырезал какие-то элементы, то с помощью этого метода он может вставить их обратно на сцену. Кроме того, мы добавили возможность вставлять HTML-разметку и простой текст, помещенный в буфер обмена другими приложениями.

Сначала мы получаем указатель на системный буфер обмена и извлекаем оттуда объект типа `QMimeType`, в котором хранятся данные, – если в буфере обмена что-то есть. Затем мы проверяем формат данных. Если это внутренний формат `Page Designer`, мы извлекаем данные в виде `QByteArray` и с помощью рассмотренного выше метода `readItems()` (стр. 451) помещаем элементы на сцену. Но, в отличие от чтения из файла, здесь мы указываем смещение, чтобы вставленные элементы не оказались в точности на тех же местах, что исходные (поскольку тогда пользователь может не понять, что мы что-то вставили), а кроме того, вставленные элементы выделяются, чтобы к ним можно было применять различные групповые операции, например перемещения или удаления. Затем мы увеличиваем смещение; тогда, если пользователь сразу же вставит те же самые элементы еще раз, они окажутся немного сдвинуты относительно предыдущих и будет видно, что вставка действительно произошла.

Если формат данных отличается от формата `Page Designer`, то мы смотрим, представлены ли они в формате HTML или простого текста. И в том и в другом случае мы создаем новый объект `TextItem` в подходящем месте на сцене, которое нам возвращает метод `position()`, и соединяем его с виджетами из наборов инструментов. Затем мы записываем полученный HTML- или простой текст в текстовый элемент. (Отметим, что проверять MIME-типы данных всегда следует в порядке убывания предпочтения, то есть наиболее желательный формат – первым. Например, если в буфере обмена находится HTML-разметка, то оба метода, `QMimeType::hasText()` и `QMimeType::hasHtml()`, скорее всего, вернут `true`, поскольку многие приложения копируют текст в буфер обмена в разных форматах и вполне возможно, что среди них окажется как `text/html`, так и `text/plain`.)

Если на сцену были добавлены элементы – все равно, в результате считывания или создания нового элемента, – то перед выходом мы вызываем метод `setDirty()`, поскольку сцена существенно изменилась.

Манипулирование выделенными элементами

В этом подразделе мы рассмотрим слот `editAlign()` и связанный с ним вспомогательный метод, чтобы продемонстрировать типичную реализацию операции над двумя или более выделенными элементами. Кроме того, мы посмотрим, что происходит, когда изменяется множество выделенных элементов.

Пользователь может запросить выравнивание нескольких элементов двумя способами: вызвать нужное действие (например, `Align Top` (Выровнять по верхнему краю)) с помощью пункта главного меню или меню кнопки `Align` (Выровнять) на панели инструментов либо просто нажать кнопку `Align`; в последнем случае будет применено то же выравнивание, что в последний раз, или подразумеваемое по умолчанию, если это первая операция выравнивания с момента запуска приложения.

Чтобы было понятнее, о чем речь, покажем, как конфигурируются действия выравнивания на этапе инициализации (фрагмент метода `createMenusAndToolBars()`):

```
QMenu *alignmentMenu = new QMenu(tr("Align"), this);
foreach (QAction *action, QList<QAction*>())
    << editAlignLeftAction << editAlignRightAction
    << editAlignTopAction << editAlignBottomAction)
    alignmentMenu->addAction(action);
editAlignmentAction->setMenu(alignmentMenu);
```

Затем действие `editAlignmentAction` добавляется в меню `Edit` и на панель инструментов. При работе с меню пользователь выбирает конкретный вид выравнивания, а при использовании панели инструментов можно как нажать саму кнопку `Align` (и тем самым выбрать действие `editAlignmentAction`), так и выбрать пункт из меню этой кнопки (и, стало быть, конкретный вид выравнивания). Поэтому наш код должен обрабатывать оба случая: само действие `editAlignmentAction` и конкретную операцию выравнивания. Отметим, что слот `updateUi()` (стр. 472) активирует действие выравнивания только в ситуации, когда выделено хотя бы два элемента, поскольку в противном случае выравнивание не имеет смысла.

Для простоты мы разобьем этот слот на три части.

```
void MainWindow::editAlign()
{
    QAction *action = qobject_cast<QAction*>(sender());
    if (!action)
        return;
```



```

Qt::Alignment alignment = static_cast<Qt::Alignment>(
    action->data().toInt());
if (action != editAlignmentAction) {
    editAlignmentAction->setData(action->data());
    editAlignmentAction->setIcon(action->icon());
}

```

Сначала мы определяем, какое действие вызвало слот. Требуемое выравнивание хранится в данных, ассоциированных с действием, поэтому мы получаем его и сохраняем в локальной переменной. Если это не само действие `editAlignmentAction`, то мы записываем в данные `editAlignmentAction` идентификатор конкретного выбранного действия и соответственно изменяем его пиктограмму. В результате при последующем выборе `editAlignmentAction` будет выполнена та конкретная операция выравнивания, которая выбиралась в последний раз.

```

QList<QGraphicsItem*> items = scene->selectedItems();
QVector<double> coordinates;
populateCoordinates(alignment, &coordinates, items);
double offset;
if (alignment == Qt::AlignLeft || alignment == Qt::AlignTop)
    offset = *std::min_element(coordinates.constBegin(),
                                coordinates.constEnd());
else
    offset = *std::max_element(coordinates.constBegin(),
                                coordinates.constEnd());

```

Алгоритм выравнивания очень простой. Создаем вектор, содержащий координаты тех краев элементов, которые соответствуют выбранному выравниванию. Например, если требуется выровнять по левому краю, то берется абсцисса каждого элемента, а если по нижнему, то ордината плюс высота. Затем в этом векторе ищется минимальная (в случае выравнивания по левому или по верхнему краю) или максимальная (в случае выравнивания по правому или по нижнему краю) координата, и каждый элемент сдвигается на разность между этой величиной и его фактической координатой.

Функции `std::min_element()` и `std::max_element()` определены в заголовочном файле `<algorithm>` из стандартной библиотеки шаблонов (STL). Они получают на входе итераторы, указывающие на начало и конец диапазона, и возвращают итератор, указывающий соответственно на минимальный или максимальный элемент последовательности. Поэтому мы сразу извлекаем значение, на которое указывает итератор, с помощью метода `operator*()`. (Если бы мы захотели обойтись без STL, то могли бы написать собственные шаблонные функции `min()` и `max()`, принимающие в качестве аргумента последовательность, или поступить так: `qSort(coordinates); offset = coordinates.first();` — чтобы найти минимум, и `coordinates.last()` — чтобы найти максимум.)

```

        if (alignment == Qt::AlignLeft || alignment == Qt::AlignRight){
            for (int i = 0; i < items.count(); ++i)
                items.at(i)->moveBy(offset - coordinates.at(i), 0);
        }
        else {
            for (int i = 0; i < items.count(); ++i)
                items.at(i)->moveBy(0, offset - coordinates.at(i));
        }
        setDirty(true);
    }
}

```

Именно здесь мы проверяем каждый элемент и сдвигаем его по горизонтали или по вертикали на расстояние, необходимое для выравнивания с самым левым (или самым правым и т. д.) элементом. Поставив все элементы на место, мы вызываем метод `setDirty()`, поскольку сцена существенно изменилась.

Пользователь видит, как сразу после запроса выравнивания все элементы оказываются в новых позициях. В главе 13 мы вернемся к этому методу и покажем, как можно анимировать выравнивание, чтобы, с одной стороны, сделать перемещение более наглядным, а с другой — более элегантным.

```

void MainWindow::populateCoordinates(const Qt::Alignment &alignment,
    QVector<double> *coordinates, const QList<QGraphicsItem*> &items)
{
    QListIterator<QGraphicsItem*> i(items);
    while (i.hasNext()) {
        QRectF rect = i.next()->sceneBoundingRect();
        switch (alignment) {
            case Qt::AlignLeft:
                coordinates->append(rect.x()); break;
            case Qt::AlignRight:
                coordinates->append(rect.x() + rect.width());
                break;
            case Qt::AlignTop:
                coordinates->append(rect.y()); break;
            case Qt::AlignBottom:
                coordinates->append(rect.y() + rect.height());
                break;
        }
    }
}

```

Этот метод обходит переданный ему список элементов и заполняет переданный вектор чисел типа `double` абсциссами или ординатами. Метод `QGraphicsItem::sceneBoundingRect()` возвращает ограничивающий прямоугольник элемента в координатах сцены. Гораздо чаще используется метод `QGraphicsItem::boundingRect()`, который возвращает ограничивающий прямоугольник элемента в координатах элемента, но здесь нам нужны

именно координаты сцены, потому что мы собираемся перемещать элементы на сцене (например, выровнять их). Избранные методы, входящие в API класса `QGraphicsItem`, а также основные перечисления приведены в табл. 11.1–11.7 (стр. 436–441).

Еще одно требование, обычно предъявляемое к программам рисования, – возможность равномерно распределить элементы по горизонтали или по вертикали. Это означает следующее: при наличии трех и более элементов оставить два элемента на месте, а остальные поместить между ними так, чтобы промежутки между соседними элементами были одинаковы. Поскольку реализация этой операции не добавит ничего нового к нашим знаниям об архитектуре графических представлений, то мы оставляем ее в качестве упражнения читателю.

Но прежде, чем распрощаться с темой манипулирования графическими элементами, рассмотрим еще один слот.

```
void MainWindow::selectionChanged()
{
    QList<QGraphicsItem*> items = scene->selectedItems();
    if (items.count() == 1) {
        if (QObject *item = dynamic_cast<QObject*>(items.at(0))) {
            if (item->property("brush").isValid())
                brushWidget->setBrush(
                    item->property("brush").value<QBrush>());
            if (item->property("pen").isValid())
                penWidget->setPen(
                    item->property("pen").value<QPen>());
            if (item->property("angle").isValid()) {
                transformWidget->setAngle(
                    item->property("angle").toDouble());
                transformWidget->setShear(
                    item->property("shearHorizontal").toDouble(),
                    item->property("shearVertical").toDouble());
            }
        }
    }
    updateUi();
}
```

Этот слот соединен с сигналом `QGraphicsScene::selectionChanged()`, он нужен для того, чтобы в виджетах из набора инструментов отображались свойства выделенного элемента. (Выше мы отмечали, что использование оператора `dynamic_cast<>()` делает приложение зависимым от поддержки RTTI, хотя существуют способы устранить эту зависимость; стр. 457.)

Здесь нас интересует случай, когда выделение изменилось таким образом, что остался выделенным только один элемент. Если выделено два и более элементов, то мы не знаем, чьи кисть, перо и преобразование показывать, поэтому оставляем виджеты без изменения. Сделать их

недоступными нельзя, потому что у пользователя может возникнуть желание изменить параметры сразу нескольких элементов – например, повернуть все выделенные элементы или изменить их кисти.

И снова мы воспользовались системой метаобъектов, чтобы сделать метод максимально общим и не модифицировать его при каждом добавлении новых типов элементов. (Как уже отмечалось, можно было бы вместо этого применить функцию `qgraphicsitem_cast<>()`, если мы не хотим, чтобы типы наших элементов наследовали классу `QObject`.) Раньше, в методе `connectItem()` (стр. 453), мы использовали метаобъект класса элемента и метод `QMetaObject::indexOfProperty()` для того, чтобы узнать, обладает ли некий элемент определенным свойством. Здесь же мы выбрали более прямой – хотя и не такой однозначный – подход, воспользовавшись методом `QObject::property()`. Этот метод принимает имя свойств и возвращает объект `QVariant` – недопустимый, если у объекта нет свойства с указанным именем¹.

Пользовательский интерфейс существенно зависит от того, сколько объектов выделено (ни одного, один, два или более). Если не выделено ни одного элемента, то все виджеты в наборах инструментов должны быть недоступны, поскольку их изменение не даст никакого эффекта. Если выделен ровно один элемент, то должно быть доступно действие `Edit Selected Item...`, а если два и более – то действия выравнивания. Все это и многое другое учтено в методе `updateUi()`, который мы скоро рассмотрим (стр. 472).

В программе `Page Designer` виджеты `BrushWidget` и `PenWidget` показывают параметры соответственно кисти и пера, если выделен прямоугольник или смайлик, а `TransformWidget` – параметры преобразований (поворота и сдвига). (Класс `BrushWidget` мы рассмотрим ниже, на стр. 476.) Напомним, что если пользователь изменяет свойства в каком-то виджете из набора инструментов – например `BrushWidget`, – то уведомление получают *все* виджеты, обладающие свойством «кисть», но лишь те, что в данный момент выделены, обновляют в ответ свои кисти; мы убедимся в этом, когда перейдем к рассмотрению пользовательских графических элементов (стр. 470).

Соккрытие и показ сетки

По-видимому, проще и лучше всего отображать сетку, задав для сцены подходящую кисть с помощью метода `QGraphicsScene::setBackgroundBrush()` или переопределив метод `QGraphicsScene::drawBackground()`. Мы вы-

¹ Метод `QObject::property()` возвращает недопустимый `QVariant` в случае, когда свойства с указанным именем не существует вовсе или значение этого свойства – недопустимый объект. Если же мы хотим точно знать, существует свойство или нет, то нужно вызывать метод `QMetaObject::indexOfProperty()`, как раньше.

брали другой подход только для того, чтобы продемонстрировать работу с классом `QGraphicsItemGroup` и показать, как можно избирательно сохранять или обрабатывать элементы.

В приложении **Page Designer** имеется действие-переключатель `view>ShowGridAction`, соединенное со слотом `viewShowGrid()`, который создает и показывает либо скрывает сетку.

```
void MainWindow::viewShowGrid(bool on)
{
    if (!gridGroup) {
        const int GridSize = 40;
        QPen pen(QColor(175, 175, 175, 127));
        gridGroup = new QGraphicsItemGroup;
        const int MaxX = static_cast<int>(std::ceil(scene->width())
            / GridSize) * GridSize;
        const int MaxY = static_cast<int>(std::ceil(scene->height())
            / GridSize) * GridSize;
        for (int x = 0; x <= MaxX; x += GridSize) {
            QGraphicsLineItem *item = new QGraphicsLineItem(x, 0, x, MaxY);
            item->setPen(pen);
            item->setZValue(std::numeric_limits<int>::min());
            gridGroup->addToGroup(item);
        }
        ...
        scene->addItem(gridGroup);
    }
    gridGroup->setVisible(on);
}
```

В этом методе мы создаем группу, содержащую линии сетки, если она еще не существует, — например, потому что приложение только что запущено или сцена была очищена. Затем настраивается перо, которым мы будем рисовать полупрозрачные светло-серые линии. А поскольку толщина пера явно не задана, то по умолчанию она будет равна 0, то есть мы получим косметическое перо шириной 1 пиксел. Косметические перья (любой ширины) отличаются тем, что рисуют линии точно заданной толщины вне зависимости от действующих геометрических преобразований (за исключением толщины 0, которая трактуется как 1). Толщина любого другого пера изменяется в соответствии с действующим коэффициентом масштабирования.

Мы создаем объект `QGraphicsItemGroup` и вычисляем абсциссу правого и ординату нижнего края сцены. Затем выполняем цикл по x от 0 до MaxX с шагом `GridSize` и на каждой итерации создаем объект стандартного класса `QGraphicsLineItem`, передавая ему четверку координат (x_1, y_1, x_2, y_2) . Поскольку абсциссы в этой четверке совпадают, а ординаты различны, то получается вертикальная линия. Затем мы устанавливаем для этого объекта созданное ранее перо и задаем большое отрицательное значение z , чтобы линия всегда оставалась под другими элементами. После этого

объект добавляется в группу сетки. Горизонтальные линии рисуются аналогично, поэтому мы опустили соответствующий код.

При создании объекта `QGraphicsItem` все флаги графического элемента сброшены, то есть по умолчанию пользователь не может ни переместить, ни выделить его. Это именно то поведение, которое требуется для линий сетки.

После того как все линии созданы и добавлены в группу, сама группа помещается на сцену. И в конце метода мы показываем либо скрываем группу (а значит, и все принадлежащие ей элементы) в зависимости от булевого аргумента, переданного слоту в соответствующем ему сигнале `toggled(bool)`.

Попутно отметим, что функция `std::ceil()` (из заголовочного файла `<cmath>`) возвращает наименьшее целое число, большее или равное своему аргументу, а функция `std::numeric_limits<int>::min()` (из файла `<limits>`) – наименьшее (то есть «самое отрицательное») представимое целое.

Нерассмотренным остался метод `editClearTransforms()`, к которому мы вернемся в конце раздела «Преобразования графических элементов» (стр. 490).

Актуализация интерфейса

Для особо ленивых есть прекрасный способ отображать состояние приложения – не отображать его вовсе. При таком подходе пользователь в любой момент может вызвать любое действие, а уж само действие проверяет, имеет оно смысл в данном состоянии или нет. Если нет, то ничего и не делается. Например, можно было бы оставить действие `Paste` постоянно доступным, даже если вставлять нечего, а действие `Copy` активным даже тогда, когда нет выделенных элементов (и, стало быть, нечего копировать).

К сожалению, такое положение вещей может привести пользователя в замешательство. Например, пользователь вызывает действие `Paste` и недоумевает, почему ничего не происходит. Поэтому всюду, где возможно, мы предпочитаем активировать и деактивировать действия и виджеты в наборах инструментов, так, чтобы они отражали текущее состояние приложения. Это не всегда просто. Например, если пользователь выделил три элемента, прямоугольник, смайлик и текст, то какие виджеты в наборе инструментов следует активировать? Можно было бы активировать тот набор или те наборы инструментов, которые применимы ко всем выделенным элементам; тогда в данном случае нужно было бы активировать только набор инструментов для преобразований. Но мы решили, что это будет неудобно пользователям, поэтому активируем те наборы инструментов, которые применимы хотя бы к одному выделенному элементу. Следовательно, в данном случае активируются наборы инструментов для задания параметров преобразований, пера

и кисти, несмотря на то, что у текстовых элементов ни пера, ни кисти нет. Это решение вполне применимо, потому что если мы, к примеру, изменим кисть, то текстовые элементы это изменение благополучно проигнорируют, а к прямоугольникам и смайликам оно будет применено.

В программе Page Designer при любом существенном изменении вызывается слот `setDirty()`. В случае действия Save он вызывается с аргументом `false`, так как после сохранения несохраненных объектов на сцене, естественно, не осталось. Для большинства же прочих изменений метод вызывается с аргументом `true` (это значение подразумевается по умолчанию).

```
void MainWindow::setDirty(bool on)
{
    setWindowModified(on);
    updateUi();
}
```

Вместо того чтобы хранить собственную булеву переменную `dirty`, мы пользуемся свойством `windowModified` главного окна. А при каждом изменении состояния вызываем метод `updateUi()` (который вызывается и из других мест, например из `editCopy()` и `selectionChanged()`), чтобы активировать или деактивировать действия и виджеты в соответствии с новым состоянием.

```
void MainWindow::updateUi()
{
    fileSaveAction->setEnabled(isWindowModified());
    bool hasItems = sceneHasItems();
    fileSaveAsAction->setEnabled(hasItems);
    fileExportAction->setEnabled(hasItems);
    filePrintAction->setEnabled(hasItems);
    int selected = scene->selectedItems().count();
    editSelectedItemAction->setEnabled(selected == 1);
    editCopyAction->setEnabled(selected >= 1);
    editCutAction->setEnabled(selected >= 1);
    QClipboard *clipboard = QApplication::clipboard();
    const QMimeData *mimeType = clipboard->mimeType();
    editPasteAction->setEnabled(mimeType &&
        (mimeType->hasFormat(MimeType) || mimeType->hasHtml() ||
         mimeType->hasText()));
    editAlignmentAction->setEnabled(selected >= 2);
    editClearTransformsAction->setEnabled(selected >= 1);
    transformWidget->setEnabled(selected >= 1);
    bool hasBrushProperty;
    bool hasPenProperty;
    getSelectionProperties(&hasBrushProperty, &hasPenProperty);
    brushWidget->setEnabled(hasBrushProperty);
    penWidget->setEnabled(hasPenProperty);
}
```

Действие `Save` доступно только при наличии несохраненных изменений. Действия `Save As`, `Export` и `Print` доступны, если на сцене есть хотя бы один элемент. Действие `Edit Selected Item...` доступно, если выделен ровно один элемент, а действия `Copy` и `Cut` – если выделен хотя бы один элемент. Действие `Paste` доступно, если в системном буфере обмена имеются данные во внутреннем формате `Page Designer` или в формате `HTML` либо простого текста (работу с буфером обмена мы обсуждали выше, на стр. 461). Действие `Align` (а следовательно, также `Align Left` (Выровнять по левому краю), `Align Right` (Выровнять по правому краю) и пр.) доступно, если выделены хотя бы два элемента, а действие `Clear Transformations` – если выделен хотя бы один элемент. Строго говоря, действие `Clear Transformations` следовало бы активировать лишь в случае, когда для выбранного (или, по крайней мере, одного из нескольких выбранных) элемента определен ненулевой угол поворота либо коэффициент сдвига; несложную проверку этого условия оставляем читателю в качестве упражнения.

Набор инструментов для задания преобразования доступен, если выделен хотя бы один элемент, потому что все пользовательские элементы в программе `Page Designer` поддерживают преобразования. А вот виджеты для задания параметров кисти и пера доступны лишь в том случае, если у единственного выделенного элемента (или хотя бы у одного из нескольких выделенных) имеется соответствующее свойство.

Для полноты рассмотрим еще два вспомогательных метода, вызываемых из `updateUi()`.

```
bool MainWindow::sceneHasItems() const
{
    foreach (QGraphicsItem *item, scene->items())
        if (item != gridGroup && item->group() != gridGroup)
            return true;
    return false;
}
```

Этот метод возвращает `true`, если на сцене есть хотя бы один элемент (помимо сетки).

```
void MainWindow::getSelectionProperties(bool *hasBrushProperty,
                                       bool *hasPenProperty) const
{
    Q_ASSERT(hasBrushProperty && hasPenProperty);
    *hasBrushProperty = false;
    *hasPenProperty = false;
    foreach (QGraphicsItem *item, scene->selectedItems()) {
        if (QObject *object = dynamic_cast<QObject*>(item)) {
            const QMetaObject *metaObject = object->metaObject();
            if (metaObject->indexOfProperty("brush") > -1)
                *hasBrushProperty = true;
            if (metaObject->indexOfProperty("pen") > -1)
                *hasPenProperty = true;
        }
    }
}
```



```

        if (*hasBrushProperty && *hasPenProperty)
            break;
    }
}
}

```

Этот метод обходит все выделенные элементы, проверяя, какие из них обладают свойством «перо» или «кисть». В качестве небольшой оптимизации мы прекращаем просмотр, как только обнаружен элемент, обладающий обоими свойствами. (Выше уже отмечалось, что использование оператора `dynamic_cast<>()` делает приложение зависимым от поддержки RTTI, хотя это ограничение можно обойти; стр. 457.)

Расширение класса QGraphicsView

Класс `QGraphicsView` изначально не поддерживает панорамирование, поэтому мы написали небольшой подкласс (реализованный целиком в заголовочном файле), который добавляет необходимую функциональность, а заодно уж воспользовались возможностью, чтобы включить режим сглаживания и поддержать выделение с помощью эластичного контура. Вот полное определение класса `GraphicsView`:

```

class GraphicsView : public QGraphicsView
{
    Q_OBJECT

public:
    explicit GraphicsView(QWidget *parent=0) : QGraphicsView(parent)
    {
        setDragMode(RubberBandDrag);
        setRenderHints(QPainter::Antialiasing|
                      QPainter::TextAntialiasing);
    }

public slots:
    void zoomIn() { scaleBy(1.1); }
    void zoomOut() { scaleBy(1.0 / 1.1); }

protected:
    void wheelEvent(QWheelEvent *event)
    { scaleBy(std::pow(4.0 / 3.0, (-event->delta() / 240.0))); }

private:
    void scaleBy(double factor) { scale(factor, factor); }
};

```

В конструкторе мы первым делом включаем режим перетаскивания с помощью эластичного контура. Это означает, что когда пользователь нажимает кнопку мыши и начинает перетаскивание, рисуется эластичный контур (то есть прямоугольник, который на некоторых плат-

формах закрашен полупрозрачным цветом) и каждый элемент, оказавшийся внутри него или пересекающий его границу, становится выделенным. По умолчанию предполагается режим перетаскивания `QGraphicsView::NoDrag`, в котором ничего не делается. Поддерживается также режим `QGraphicsView::ScrollHandDrag`; если он установлен, то во время перетаскивания представление прокручивается.

Для поддержки панорамирования мы включили два слота, оба масштабируют изображение. И еще переопределили обработчик событий от колесика мыши, чтобы при масштабировании учитывался угол поворота колесика, причем вращение вперед означает уменьшение масштаба, а вращение назад – его увеличение. Значение `QWheelEvent::delta()` сообщает угол поворота в «шагах» (один шаг обычно соответствует вращению на угол 15°); положительное значение означает вращение вперед, отрицательное – назад. Функция `std::pow()` (из заголовочного файла `<cmath>`) возводит свой первый аргумент в степень, заданную вторым аргументом. По сути дела, мы здесь увеличиваем или уменьшаем масштаб с коэффициентом $1\frac{1}{2}$ для каждого шага колесика мыши.

Собственно масштабирование производится в закрытом методе `scaleBy()`, который просто вызывает метод `QGraphicsView::scale()`, передавая ему один и тот же коэффициент масштабирования по горизонтали и вертикали, чтобы не исказить пропорции представления.

По умолчанию в классе `QGraphicsView` события колесика мыши используются для прокрутки, но, переопределив обработчик `wheelEvent()`, мы подавили это поведение. Это означает, что осуществлять прокрутку колесиком теперь можно, только когда указатель мыши находится внутри полосы прокрутки.

Создание стыкуемого виджета с набором инструментов

В главном окне программы `Page Designer` находятся три стыкуемых виджета: для преобразования (поворота и сдвига), кисти и пера. Структурно все три виджета похожи, а логически используются для двух целей: показать свойства единственного выделенного элемента (к примеру, стиль и цвет кисти) и изменить свойства всех выделенных элементов при изменении значений в виджетах редактирования.

Все стыкуемые виджеты в программе `Page Designer` схожи по структуре и назначению, поэтому мы рассмотрим только один – `BrushWidget`. Код остальных, естественно, имеется в примерах к этой книге. Все три стыкуемых виджета изображены на рис. 12.1 (стр. 444).

Как обычно, начнем с объявления класса в заголовочном файле, но опустим закрытые слоты и методы; мы рассмотрим их по ходу обсуждения открытых методов и слотов.

```

class BrushWidget : public QWidget
{
    Q_OBJECT

public:
    explicit BrushWidget(QWidget *parent=0);
    QBrush brush() const { return m_brush; }

public slots:
    void setBrush(const QBrush &brush);

signals:
    void brushChanged(const QBrush &brush);
    ...
};

```

В классе этого виджета имеется одна закрытая переменная-член `m_brush` типа `QBrush`. Предоставляется соответствующий метод чтения, а метод установки сделан слотом, чтобы параметры кисти можно было назначить в результате активации соединения между сигналом и слотом. Если кисть изменена (например, в результате изменения цвета или стиля пользователем), то генерируется сигнал `brushChanged()`. Ранее мы видели, что каждый элемент, обладающий свойством «кисть», то есть прямоугольники и смайлики, подключен к этому сигналу. (Как они реагируют на сигнал, мы покажем ниже, при обсуждении реализации пользовательских элементов; стр. 470.)

```

BrushWidget::BrushWidget(QWidget *parent)
    : QWidget(parent)
{
    createWidgets();
    setBrush(QBrush());
    createLayout();
    createConnections();
    setFixedSize(minimumSizeHint());
}

```

Конструктор поручает большую часть работы закрытым вспомогательным методам. Первоначально для кисти задается черный цвет и стиль `Qt::NoBrush` (то есть фактически эта кисть ничего не делает). После того как дочерние виджеты созданы и скомпонованы, мы фиксируем размер виджета, делая его равным минимальному предпочтительному размеру, потому что изменять размер у пользователя нет причин. (Любопытным читателям предлагаем закомментировать обращения к методу `setFixedSize()` во всех трех виджетах наборов инструментов и посмотреть, что получится.)

```

void BrushWidget::createWidgets()
{
    colorComboBox = new QComboBox;

```

```
foreach (const QString &name, QColor::colorNames()) {
    QColor color(name);
    colorComboBox->addItem(colorSwatch(color), name, color);
}
styleComboBox = new QComboBox;
typedef QPair<QString, Qt::BrushStyle> BrushPair;
foreach (const BrushPair &pair, QList<BrushPair>())
    << qMakePair(tr("No Brush"), Qt::NoBrush)
    << qMakePair(tr("Solid"), Qt::SolidPattern)
    ...
    << qMakePair(tr("Diagonal Cross"), Qt::DiagCrossPattern))
styleComboBox->addItem(brushSwatch(pair.second), pair.first,
pair.second);
}
```

У этого виджета есть четыре дочерних: две метки и два комбинированных списка. Чтобы пользователь видел не просто название цвета и стиль кисти, а и визуальное представление, мы включили образцы (в виде небольших значков). Каждому цвету сопоставлен соответственно закрашенный кружочек, а каждому стилю кисти – квадратик, залитый ее текущим цветом (первоначально черным).

Для каждого цвета в комбинированный список помещается элемент, содержащий образец, название цвета (в виде строки `QString` с HTML-представлением цвета, например `"#FF0000"` для красного¹) и данные – объект `QVariant`, в котором хранится сам цвет `QColor`. Статический метод `QColor::colorNames()` возвращает отсортированный список `QStringList` понятных человеку названий цветов (например, «бледно-зеленый», «красный» и т. д.)². Точно так же мы поступаем со стилями кисти – помещаем в комбинированный список элементы, содержащие образец стиля, его название и данные в виде объекта `QVariant`, в котором хранится соответствующий элемент перечисления. Разумеется, большая часть кода заполнения списка стилей опущена, потому что ничего нового в нем нет.

В главе 9 (стр. 377) мы уже отмечали, что внутри конструкции `foreach` первая запятая отделяет элемент от последовательности, поэтому сами элементы не должны содержать запятых. Эту проблему мы решили стандартным способом – ввели `typedef`.

¹ На практике, как правило, добавляются человеко-читаемые названия цветов, возвращаемые статическим методом `QColor::colorNames()`. – *Прим. науч. ред.*

² На момент написания этой книги в документации явно не говорилось, что названия цветов отсортированы, поэтому можно было подстраховаться – перед тем как добавлять названия в список, поместить их в переменную и вызвать метод `QStringList::sort()`. В качестве альтернативы можно просто вызвать `QComboBox::model()->sort(0)` после заполнения списка, чтобы отсортировать его по первому (и единственному) столбцу в порядке возрастания.

Также стоит обратить внимание на то, что мы создаем только комбинированные списки, без меток. Как мы скоро увидим, метки создает класс компоновки.

В приложении имеется пять функций, генерирующих образцы: `colorSwatch()`, `brushSwatch()`, `penStyleSwatch()`, `penCapSwatch()` и `penJoinSwatch()`. Мы рассмотрим только самую простую – `brushSwatch()` – в качестве типичного представителя. (Остальные чуть сложнее, но не принципиально; например, функция, генерирующая образец цвета, может добавить символ `C` контрастным цветом, а функция, воссоздающая образец соединения линий, проведенных пером, рисует ломаную, состоящую из двух отрезков, которые соединяются в соответствии с данным стилем.)

```
QPixmap brushSwatch(const Qt::BrushStyle style,
                    const QColor &color, const QSize &size)
{
    QString key = QString("BRUSHSTYLESWATCH:%1:%2:%3x%4")
        .arg(static_cast<int>(style)).arg(color.name())
        .arg(size.width()).arg(size.height());
    QPixmap pixmap(size);
    if (!QPixmapCache::find(key, &pixmap)) {
        pixmap.fill(Qt::transparent);
        QPainter painter(&pixmap);
        painter.setRenderHint(QPainter::Antialiasing);
        painter.setPen(Qt::NoPen);
        painter.setBrush(QBrush(color, style));
        painter.drawRect(0, 0, size.width(), size.height());
        painter.end();
        QPixmapCache::insert(key, pixmap);
    }
    return pixmap;
}
```

Эта функция принимает стиль кисти, необязательный цвет (по умолчанию черный) и необязательный размер (по умолчанию 24×24 пиксела), а возвращает квадратик, нарисованный кистью указанного стиля, цвета и размера.

Вместо того чтобы при каждом вызове функции создавать новый растр, мы кэшируем их – по умолчанию не более 10 Мбайт, но эту величину можно изменить, обратившись к методу `QPixmapCache::setCacheLimit()`. В кэше каждому растру должен быть сопоставлен уникальный ключ. В качестве такого ключа мы используем строку, идентифицирующую данный растр как образец кисти, а не чего-то другого (например, цвета, стиля пера и т. д.), за которой следует идентификатор самого образца – в данном случае стиль кисти (в виде `int`), ее цвет (в виде HTML-кода цвета) и ширина и высота растра. Например, сплошной коричневой кисти размером 24×24 соответствует ключ `"BRUSHSTYLESWATCH:1:#a52a2a:24x24"`.

Сформировав ключ, мы создаем растр нужного размера. Метод `QPixmapCache::find()` ищет растр с указанным ключом в кэше. Если ключ най-

ден, то метод возвращает `true` и заполняет данными объект `QPixmap`, переданный по указателю (или, в случае версии Qt 4.5 и более ранних, по неконстантной ссылке, то есть без символа `&`). В противном случае возвращается `false`. Таким образом, при первом запросе растра мы не найдем его в кэше и должны будем создать сами. Сначала мы устанавливаем для растра прозрачный фон, потом создаем объект `QPainter`, который нарисует указанной кистью прямоугольник, занимающий весь растр. Но хотя растр закрашивается целиком, мы все равно должны предварительно сделать его прозрачным или залить каким-то цветом, потому что стили кисти по большей части не сплошные и сквозь них просвечивает фон. Созданный растр помещается в кэш с ранее сформированным ключом. В заключение метод возвращает растр – полученный из кэша или только что созданный.

Структурно все остальные методы создания образцов аналогичны приведенному выше, единственное различие в способе формирования ключей и в том, что именно рисуется. Отметим, что ключ не должен начинаться строкой `“$qt”`, потому что Qt использует кэш `QPixmapCache` для своих целей тоже и этот префикс зарезервирован для ключей библиотеки.

```
void BrushWidget::createLayout()
{
    QFormLayout *layout = new QFormLayout;
    layout->addRow(tr("Color"), colorComboBox);
    layout->addRow(tr("Style"), styleComboBox);
    setLayout(layout);
}
```

Мы решили показать метод `createLayout()`, потому что в нем используется класс `QFormLayout`, появившийся в версии Qt 4.4, и с его помощью создаются две метки. Удобство создания меток подобным образом состоит в том, что Qt автоматически делает их партнером (buddy) добавляемого виджета. Это означает, что всякий раз, как метка получает фокус, она тут же передает его своему партнеру.

```
void BrushWidget::createConnections()
{
    connect(colorComboBox, SIGNAL(currentIndexChanged(int)),
            this, SLOT(updateColor(int)));
    connect(styleComboBox, SIGNAL(currentIndexChanged(int)),
            this, SLOT(updateStyle(int)));
}
```

Этот метод приведен только для полноты картины. Если пользователь изменяет цвет или стиль кисти, то вызываются соответствующие слоты всех подключенных к сигналу виджетов, и если речь идет об изменении цвета, то мы получаем возможность сменить цвет образца кисти.

На этом мы закончили рассмотрение конструктора и вызываемых им закрытых вспомогательных методов. Теперь можно перейти к откры-

тому слоту `setBrush()`, а затем к закрытым слотам и вспомогательным методам.

```
void BrushWidget::setBrush(const QBrush &brush)
{
    if (m_brush != brush) {
        m_brush = brush;
        colorComboBox->setCurrentIndex(
            colorComboBox->findData(m_brush.color()));
        styleComboBox->setCurrentIndex(styleComboBox->findData(
            static_cast<int>(m_brush.style())));
    }
}
```

Этот слот вызывается, когда выделен ровно один элемент, при условии, что у этого элемента есть свойство «кисть» (то есть это прямоугольник или смайлик). Слот подменяет закрытую переменную `m_brush` переданной кистью и в обоих комбинированных списках делает текущими элементы, соответствующие цвету и стилю новой кисти.

Метод `QComboBox::findData()` получает на входе `QVariant` и возвращает индекс первого совпадающего элемента или `-1`. Объекты `QColor` можно искать непосредственно, но элементы перечисления необходимо сначала привести к типу `int`, потому что именно так они хранятся в `QVariant`.

```
void BrushWidget::updateColor(int index)
{
    m_brush.setColor(colorComboBox->itemData(index).value<QColor>());
    updateSwatches();
    emit brushChanged(m_brush);
}
```

Этот метод вызывается при смене цвета кисти — при выборе его пользователем из комбинированного списка или потому, что в результате выделения элемента был вызван метод `setBrush()`.

Метод `QComboBox::itemData()` возвращает объект `QVariant`, содержащий данные, которые ассоциированы с элементом с указанным индексом. (По умолчанию это недопустимый `QVariant`.) Но в классе `QVariant` определены только методы преобразования для встроенных в *QtCore* типов (`QVariant::toInt()`, `QVariant::toSize()` и т. д.), для остальных нужно использовать шаблонный метод `QVariant::value<T>()`, указав тип значения, которое мы хотим вернуть.

После того как закрытая переменная `m_brush` изменена в соответствии с кистью нужного цвета, мы вызываем закрытый метод `updateSwatches()`, чтобы образцы стилей кисти тоже отображались новым цветом. А в заключение генерируем сигнал `brushChanged()`, чтобы уведомить подключенные объекты о смене кисти.

```
void BrushWidget::updateStyle(int index)
{

```

```
m_brush.setStyle(static_cast<Qt::BrushStyle>(
    styleComboBox->itemData(index).toInt()));
emit brushChanged(m_brush);
}
```

Это аналог метода `updateColor()` для стиля кисти. Стиль задается с помощью перечисления, поэтому мы сначала извлекаем из объекта `QVariant` с данными значение типа `int`, затем приводим его к типу перечисления и только потом обновляем закрытую переменную `m_brush`. В конце метод генерирует сигнал `brushChanged()`, передавая в нем новую кисть.

Вызывать метод `updateSwatches()` нет необходимости, потому что все образцы в комбинированном списке обновляются при смене цвета, так что если текущий элемент в списке стилей изменен пользователем или программой, то ассоциированный с ним образец уже имеет нужный цвет.

```
void BrushWidget::updateSwatches()
{
    QColor color = colorComboBox->itemData(
        colorComboBox->currentIndex()).value<QColor>();
    for (int i = 0; i < styleComboBox->count(); ++i)
        styleComboBox->setItemIcon(i, brushSwatch(
            static_cast<Qt::BrushStyle>(
                styleComboBox->itemData(i).toInt()), color));
}
```

Чтобы максимально наглядно представить пользователю эффект смены цвета кисти, мы соответственно обновляем образцы стилей в списке `styleComboBox`. Для того чтобы увидеть, как это происходит, запустите приложение `Page Designer`, выберите из списка стилей кисти любой элемент, кроме `NoBrush`, а потом перейдите в список цветов и перемещайтесь по нему с помощью клавиш со стрелками – вы заметите, что образец в списке стилей меняет цвет. (На некоторых платформах необходимо после щелчка по списку цветов нажать клавишу `Esc`, чтобы при использовании клавиш со стрелками был виден список стилей.) Тот же эффект можно наблюдать в наборе инструментов для пера – при перемещении по списку цветов с помощью клавиш со стрелками стили пера, его кончика и соединения линий будут менять цвет.

В начале метода мы получаем текущий цвет. Затем перебираем все элементы списка стилей кисти и для каждого изменяем значок (полагаясь на не-`explicit` конструктор `QIcon` с аргументом типа `QPixmap`) на тот, что возвращает функция `brushSwatch()`. В качестве первого аргумента `brushSwatch()` мы передаем стиль кисти, который получаем путем извлечения значения типа `int` из объекта `QVariant` и приведения его к типу перечисления. А вторым аргументом передается текущий цвет, установленный в комбинированном списке цветов.

Классы `TransformWidget` и `PenWidget` (не показаны) структурно и логически очень похожи на `BrushWidget`.

На этом мы заканчиваем рассмотрение инфраструктуры приложения Page Designer: сохранение, загрузку и экспорт сцен, добавление элементов и работу с ним, манипулирование группами элементов. Осталось только рассмотреть реализацию пользовательских графических элементов, чем мы и займемся в следующем разделе.

Создание пользовательских графических элементов

Все элементы графических представлений являются прямыми или непрямыми наследниками класса `QGraphicsItem`. Большинство созданных для удобства подклассов `QGraphicsItem` было включено вместе с архитектурой графических представлений в версию Qt 4.2, но классы `QGraphicsProxyWidget` и `QGraphicsWidget` были добавлены лишь в Qt 4.4, а классы `QGraphicsObject` и `QGraphicsWebView` – в Qt 4.6. Иерархия классов была показана выше, на стр. 423, а избранные методы класса `QGraphicsItem` и основные перечисления – в табл. 11.1–11.7; стр. 436–441.

Наличие столь большого числа классов графических элементов означает, что в большинстве случаев нам не придется создавать подклассы исключительно для рисования нестандартных фигур. В конце концов, практически все можно нарисовать с помощью классов `QGraphicsEllipseItem`, `QGraphicsLineItem`, `QGraphicsPathItem`, `QGraphicsPolygonItem` и `QGraphicsRectItem`. Таким образом, собственные подклассы пишутся преимущественно ради специального поведения.

В этом разделе мы рассмотрим три типа пользовательских графических элементов, чтобы продемонстрировать разнообразие подходов. Первый подкласс, `QGraphicsTextItem`, лишь добавляет простое поведение, оставляя все рисование Qt. Далее мы рассмотрим класс, наследующий одновременно `QObject` и `QGraphicsRectItem`, в котором реализовано более сложное поведение, в частности обработка нажатий клавиш и событий мыши. В третьем примере мы напишем подкласс `QGraphicsObject` (или `QObject` и `QGraphicsItem` в версии Qt 4.5 и более ранних), который реализует специальное поведение и сам рисует себя, а также переопределяет методы `boundingRect()` и `shape()`. Вообще-то в этом последнем классе можно было бы обойтись и без самостоятельного рисования, если бы он был подклассом `QGraphicsPathItem` (что верно для большинства фигур), но мы хотели показать, как реализуются одновременно поведение и внешний вид.

Для удобства мы создали отдельный заголовочный файл, в котором определили номера типов пользовательских графических элементов:

```
const int BoxItemType = QGraphicsItem::UserType + 1;
const int SmileyItemType = QGraphicsItem::UserType + 2;
const int TextItemType = QGraphicsItem::UserType + 3;
```

Это поможет нам избежать ситуации, когда по ошибке двум разным пользовательским элементам присваиваются одинаковые номера типов.

Расширение класса QGraphicsTextItem

Класс `QGraphicsTextItem`, множественно наследующий `QObject` и `QGraphicsItem`, служит для отображения на сцене форматированного текста. (Если достаточно простого текста, отображаемого одним шрифтом и цветом, то можно ограничиться классом `QGraphicsSimpleTextItem`.)

Классу `QGraphicsTextItem` необходимо унаследовать, потому что мы хотим наделить его специальным поведением. В частности, мы хотим, чтобы пользователь мог поворачивать и сдвигать (скашивать) элемент, а также редактировать содержащийся в нем текст. Кроме того, желательно, чтобы класс уведомлял об изменениях своего состояния с помощью нестандартного сигнала `dirty()` и мог записывать себя в поток `QDataStream` и читать оттуда. Рисование, равно как вычисление ограничивающего прямоугольника и формы, оставляем базовому классу.

Начнем с объявления класса `TextItem` в заголовочном файле, опуская закрытую секцию.

```
class TextItem : public QGraphicsTextItem
{
    Q_OBJECT
    Q_PROPERTY(double angle READ angle WRITE setAngle)
    Q_PROPERTY(double shearHorizontal READ shearHorizontal
                WRITE setShearHorizontal)
    Q_PROPERTY(double shearVertical READ shearVertical
                WRITE setShearVertical)

public:
    enum {Type = TextItemType};
    explicit TextItem(const QPoint &position,
                     QGraphicsScene *scene);
    int type() const { return Type; }
    double angle() const { return m_angle; }
    double shearHorizontal() const { return m_shearHorizontal; }
    double shearVertical() const { return m_shearVertical; }

public slots:
    void setAngle(double angle);
    void setShearHorizontal(double shearHorizontal)
        { setShear(shearHorizontal, m_shearVertical); }
    void setShearVertical(double shearVertical)
        { setShear(m_shearHorizontal, shearVertical); }
    void setShear(double shearHorizontal, double shearVertical);
    void edit();

signals:
    void dirty();
```

```
protected:
    QVariant itemChange(GraphicsItemChange change, const QVariant &value);
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent*)
    { edit(); }
    ...
};
```

В предыдущей главе мы отмечали, что всегда полезно объявить перечисление `Type` и переопределить метод `type()`, поскольку это необходимо для функции `qgraphicsitem_cast<>()`, которая приводит указатели на `QGraphicsItem` к правильному подклассу `QGraphicsItem`.

Мы сделали углы поворота и коэффициенты сдвига свойствами, чтобы их можно было опрашивать и устанавливать с помощью встроенной в Qt системы свойств, например опросить в методе `selectionChanged()` и установить в методе `editClearTransforms()`. А в качестве типа свойств мы выбрали `double`, а не `qreal`, чтобы они правильно сохранялись и загружались. (В главе 3 на стр. 122 мы говорили, что тип `qreal` ни в коем случае не следует использовать совместно с `QDataStream`.)

Первые два слота¹ нужны для того, чтобы элемент мог реагировать на изменения в виджете `TransformWidget`. Метод `itemChange()` генерирует сигнал `dirty()`, если произошло существенное изменение, например сменилась позиция или преобразование, но в случае, когда элемент становится или перестает быть выделенным, сигнал не генерируется.

Слот `edit()` введен в качестве удобства специально для приложения `Page Designer` – чтобы действие `Edit Selected Item...` могло вызвать его для редактирования выделенного элемента. (Слот `editSelectedItem()` был рассмотрен выше, на стр. 457.) Кроме того, мы переопределили обработчик двойного щелчка мышью, чтобы предоставить еще один способ вызова слота `edit()`.

Помимо объявления класса, заголовочный файл включает еще два объявления:

```
QDataStream &operator<<(QDataStream &out, const TextItem &textItem);
QDataStream &operator>>(QDataStream &in, TextItem &textItem);
```

Эти операторы используются в методах `readItems()` и `writeItems()`, обсуждавшихся выше (стр. 451).

Теперь рассмотрим методы, реализованные вне заголовочного файла, а также глобальные операторы потокового ввода/вывода. Начнем с конструктора.

```
TextItem::TextItem(const QPoint &position, QGraphicsScene *scene)
    : QGraphicsTextItem(), m_angle(0.0), m_shearHorizontal(0.0),
      m_shearVertical(0.0)
{
```

¹ Точнее, первый и предпоследний слоты — см. метод `MainWindow::connectItem()` на стр. 453 — *Прим. науч. ред.*

```

        setFont(QFont("Helvetica", 11));
        setFlags(QGraphicsItem::ItemIsSelectable|
                QGraphicsItem::ItemSendsGeometryChanges|
                QGraphicsItem::ItemIsMovable);
        setPos(position);
        scene->clearSelection();
        scene->addItem(this);
        setSelected(true);
    }

```

При конструировании объекта `TextItem` мы обнуляем углы поворота и коэффициенты сдвига, а затем устанавливаем начальную позицию (в координатах сцены) и сцену, которой принадлежит объект.

Мы хотим, чтобы пользователь мог выделять и перемещать элемент, поэтому устанавливаем соответствующие флаги. В версии Qt 4.5 и более ранних при изменении геометрии (то есть размера) вызывается метод `itemChange()`, но начиная с версии Qt 4.6 этого из соображений производительности не происходит, если явно не задать флаг `ItemSendsGeometryChanges`, что мы и делаем. (Этот вопрос обсуждался в предыдущей главе во врезке «Изменения в поведении графических представлений в Qt 4.6», стр. 426.) Далее мы задаем позицию элемента, снимаем существующее выделение и помещаем элемент на сцену. В заключение метод выделяет элемент, чтобы пользователь мог сразу же удалить или отредактировать его.

```

void TextItem::edit()
{
    QWidget *window = 0;
    QList<QGraphicsView*> views = scene()->views();
    if (!views.isEmpty())
        window = views.at(0)->window();
    TextItemDialog dialog(this, QPoint(), scene(), window);
    if (dialog.exec())
        emit dirty();
}

```

Этот слот вызывается, если выделен единственный элемент `TextItem` и пользователь вызывает действие `Edit Selected Item...` (с помощью меню или панели инструментов) либо дважды щелкает по элементу.

Класс `TextItemDialog` (код опущен, но само окно показано на рис. 12.3) – это диалоговое окно для добавления и редактирования элемента, которое позволяет изменить его текст, в частности задать различные шрифты и цвета. Мы хотим, чтобы окно открывалось в нужном месте экрана и при этом не отображалось на панели задач. И то и другое легко сделать, назначив в качестве родителя окно верхнего уровня. Однако у нас нет под рукой указателя на окно верхнего уровня. Одно из решений – обратиться к методу `QApplication::topLevelWidgets()` и выбрать из полученного от него списка первое не скрытое окно. Так мы, конечно, получим какое-то окно верхнего уровня, но не обязательно то, что нам нужно. По-

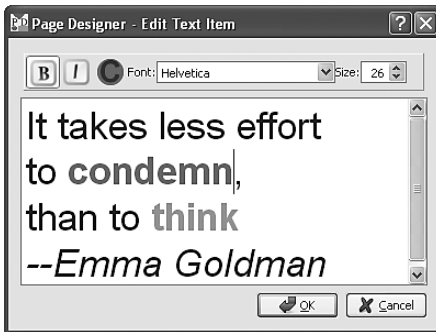


Рис. 12.3. Приложение Page Designer: диалоговое окно `TextItemDialog` в режиме редактирования

этому мы поступаем иначе: получаем список представлений, ассоциированных со сценой элементом – в данном приложении такое представление всего одно, – а затем вызываем метод `QWidget::window()`, который возвращает окно верхнего уровня для этого представления (или само представление, если оно уже является окном верхнего уровня).

На внутреннем уровне класс `TextItemDialog` пользуется классом `TextEdit`, который рассматривался в главе 9 (стр. 383), но без средств выравнивания текста. Диалоговое окно *интеллектуальное* (то есть обладает знаниями о приложении и может работать почти независимо), поэтому, когда пользователь нажимает кнопку `OK`, диалоговое окно обновляет элемент `TextItem` и планирует его перерисовку, так что нам остается только сгенерировать сигнал `dirty()`.

```
QVariant TextItem::itemChange(GraphicsItemChange change,
                              const QVariant &value)
{
    if (isDirtyChange(change))
        emit dirty();
    return QGraphicsTextItem::itemChange(change, value);
}
```

Этот метод `Qt` вызывает, когда состояние элемента изменяется. Однако, как отмечалось выше, начиная с версии `Qt 4.6` для некоторых изменений (например, касающихся только позиции и преобразования) метод вызывается автоматически лишь в случае, когда явно установлен один из флагов, определенных в перечислении `QGraphicsItem::GraphicsItemChange`, а именно `QGraphicsItem::ItemSendsGeometryChanges`. (Напомним, что, даже когда этот флаг установлен, метод `itemChange()` вызывается только в результате изменения преобразования, произведенного методом `setTransform()`. В `Qt 4.7` это поведение распространено на методы `setRotation()`, `setScale()` и `setTransformOriginPoint()`, которые появились в `Qt 4.6`.)

Мы хотим генерировать сигнал `dirty()` только для изменений, которые должны быть отражены при сохранении сцены. Например, мы не сохраняем в файл информацию о выделенных элементах, поэтому изменения, сводящиеся только к выделению или его снятию, не делают сцену «измененной».

Поскольку во всех наших пользовательских графических элементах метод `itemChange()` переопределяется, мы решили выделить в отдельную глобальную функцию `isDirtyChange()` ту его часть, в которой выясняется, приводит ли изменение к необходимости сохранения.

```
bool isDirtyChange(QGraphicsItem::GraphicsItemChange change)
{
    return (change == QGraphicsItem::ItemPositionChange ||
            change == QGraphicsItem::ItemPositionHasChanged ||
            change == QGraphicsItem::ItemTransformChange ||
            change == QGraphicsItem::ItemTransformHasChanged);
}
```

Эта функция возвращает `true`, если изменилась позиция или преобразование элемента, для всех остальных изменений возвращается `false`.

Методы `setAngle()` и `setShear()` мы рассмотрим в следующем подразделе при обсуждении преобразований графических элементов. Но сначала обсудим глобальные операторы ввода/вывода в `QDataStream`, с помощью которых объекты `TextItem` записываются в поток и читаются из него.

```
QDataStream &operator<<(QDataStream &out, const TextItem &textItem)
{
    out << textItem.pos() << textItem.angle()
        << textItem.shearHorizontal() << textItem.shearVertical()
        << textItem.zValue() << textItem.toHtml();
    return out;
}
```

Во всех операторах вывода для пользовательских элементов мы записываем позицию элемента (в координатах сцены), угол поворота, коэффициенты сдвига и значение координаты z (в программе *Page Designer* не предусмотрены средства для изменения z , добавление такой функциональности оставлено читателю в качестве упражнения). Затем записываются данные элемента, в нашем случае просто текст в формате HTML (для сохранения шрифтов, цветов и другого форматирования).

Ранее мы видели, что этот оператор применяется для записи объектов `TextItem` в файл и в массив `QByteArray` (для копирования и вырезания при работе с буфером обмена, стр. 451).

```
QDataStream &operator>>(QDataStream &in, TextItem &textItem)
{
    QPointF position;
    double angle;
```

```

double shearHorizontal;
double shearVertical;
double z;
QString html;
in >> position >> angle >> shearHorizontal >> shearVertical >> z >> html;
textItem.setPos(position);
textItem.setAngle(angle);
textItem.setShear(shearHorizontal, shearVertical);
textItem.setZValue(z);
textItem.setHtml(html);
return in;
}

```

Этот оператор ввода применяется для считывания элементов, записанных оператором вывода. Как всегда при работе с `QDataStream`, важно, чтобы данные читались в том же порядке, в котором записывались, с сохранением типов. И напомним, что оператор ввода также используется для двух целей: чтения из файла и вставки на сцену элементов из буфера обмена, где они хранятся в виде массива `QByteArray` (стр. 451).

Преобразования графических элементов

Если элемент только повернут (без сдвига или масштабирования), то для восстановления его исходного положения необходимо присвоить углу поворота первоначальное значение (например, 0°). Аналогично, если элемент скошен по горизонтали (без поворота, масштабирования и скоса по вертикали), то восстановить его положение можно, присвоив коэффициенту сдвига исходное значение (например, 0.0). То же самое относится к сдвигу по вертикали. Но если эти преобразования комбинируются (поворот со сдвигом, или сдвиг по вертикали и по горизонтали, или поворот с масштабированием), то присваивание углу поворота значения 0° , коэффициентам сдвига – значений 0.0, а коэффициенту масштабирования – значения 1.0 не восстановит исходное изображение. В общем случае для отмены преобразования недостаточно инвертировать произведенные действия (то есть применить «противоположное» преобразование). Объясняется это тем, что преобразования обычно (и в Qt тоже) представляются матрицами, а не всякая матрица обратима¹.

В программе Page Designer мы решили эту проблему, запретив комбинацию преобразований. Каждый раз мы создаем и применяем новое преобразование, так что описанная проблема не может возникнуть в принципе. Достигается это созданием в каждом элементе закрытых

¹ Матрица любого преобразования унитарна и, значит, обратима. Однако, чтобы вернуть объект в исходное состояние, необходимо применить обратные преобразования в обратном порядке. Установка для соответствующих коэффициентов значения 0 не поможет — надо повернуть объект на тот же угол, но в другую сторону и т. п. – *Прим. науч. ред.*

переменных, в которых хранятся угол поворота и коэффициенты сдвига. Это означает, что всякий раз, как пользователь изменяет угол поворота и коэффициент сдвига некоторого элемента, мы создаем новый объект `QTransform` и ассоциируем его с этим элементом.

Мы не стали реализовывать контроль над масштабированием по горизонтали и вертикали, потому что код и логика почти такие же, как для сдвига, так что ничему новому мы не научились бы. Оставляем реализацию масштабирования читателю в качестве упражнения.

Для поддержки преобразований нам необходимо включить по одному методу для установки параметра каждого вида преобразований (угла поворота, коэффициентов сдвига и т. д.), а также закрытый метод `updateTransform()`, который должен создавать и устанавливать соответствующий объект `QTransform`. Начнем с рассмотрения слотов `setAngle()` и `setShear()`, а затем перейдем к методу `updateTransform()`.

```
void TextItem::setAngle(double angle)
{
    if (isSelected() && !qFuzzyCompare(m_angle, angle)) {
        m_angle = angle;
        updateTransform();
    }
}
```

Этот метод вызывается, когда пользователь поворачивает элемент, то есть изменяет величину угла в счетчике, который расположен в виджете `TransformWidget`. Сигнал `valueChanged()` получает *каждый* подключенный к нему элемент, но мы должны применять изменение только к выделенным элементам. Кроме того, обратите внимание на употребление функции `qFuzzyCompare()` — это оптимизация, нужная для того, чтобы преобразование применялось лишь в том случае, когда новый угол в достаточной мере отличается от старого.

```
void TextItem::setShear(double shearHorizontal, double shearVertical)
{
    if (isSelected() &&
        (!qFuzzyCompare(m_shearHorizontal, shearHorizontal) ||
         !qFuzzyCompare(m_shearVertical, shearVertical))) {
        m_shearHorizontal = shearHorizontal;
        m_shearVertical = shearVertical;
        updateTransform();
    }
}
```

Как и в методе `setAngle()`, мы производим изменение только в том случае, когда элемент выделен и по крайней мере один из новых коэффициентов сдвига в достаточной мере отличается от старого.

```
void TextItem::updateTransform()
{
}
```



```

    QTransform transform;
    transform.shear(m_shearHorizontal, m_shearVertical);
    transform.rotate(m_angle);
    setTransform(transform);
}

```

Этот закрытый метод применяет преобразование к элементу. Сначала мы создаем новый объект `QTransform` (инкапсулирующий тождественную матрицу, в которой нет ни поворота, ни сдвига, ни масштабирования). Затем применяем к ней преобразования сдвига и поворота, после чего ассоциируем новое преобразование с элементом.

Обратите внимание, что мы не генерируем сигнал `dirty()` явно; поскольку установлен флаг `ItemSendsGeometryChanges`, то будет вызван метод `itemChange()`, который и сгенерирует `dirty()`.

Ранее мы отмечали, что в классе главного окна определено действие `Clear Transformations` (Сбросить преобразования). И чтобы подвести черту под этим разделом о преобразованиях, рассмотрим слот главного окна, который вызывается при активации этого действия.

```

void MainWindow::editClearTransforms()
{
    QList<QGraphicsItem*> items = scene->selectedItems();
    Q_ASSERT(!items.isEmpty());
    QListIterator<QGraphicsItem*> i(items);
    while (i.hasNext()) {
        if (QObject *item = dynamic_cast<QObject*>(i.next())) {
            if (item->property("angle").isValid()) {
                item->setProperty("angle", 0.0);
                item->setProperty("shearHorizontal", 0.0);
                item->setProperty("shearVertical", 0.0);
            }
        }
    }
    transformWidget->setAngle(0.0);
    transformWidget->setShear(0.0, 0.0);
    setDirty(true);
}

```

Этот метод сбрасывает преобразование для всех выделенных элементов, то есть присваивает углу поворота значение 0.0° , а коэффициентам сдвига — 0.0 . Затем мы обновляем виджет `TransformWidget`, чтобы в нем правильно отражались параметры преобразования выделенных элементов, — обнуляем углы поворота и коэффициенты сдвига. И в заключение вызываем метод `setDirty()`, так как произведены существенные изменения. (Применение оператора `dynamic_cast<>()` подразумевает поддержку RTTI, хотя эту трудность можно обойти, см. стр. 457.)

В утверждении `Q_ASSERT` проверяется, что список не пуст, — ведь действие `Clear Transformations` должно быть доступно, только если выделен хотя бы

один элемент, поэтому пустой список здесь означал бы, что в программу вкралась ошибка.

Строго говоря, этот метод необязателен, так как того же эффекта можно достичь, выделив нужные элементы и обнулив углы поворота и коэффициенты сдвига в `TransformWidget`. Но он удобен, потому что дает пользователю возможность нажать всего одну кнопку на панели инструментов, а не задавать значения в трех разных полях. А если в приложение будет добавлена поддержка масштабирования, этот метод станет еще полезнее, потому что без него пользователю пришлось бы задавать значения в пяти полях.

Расширение существующего класса графических элементов

В этом подразделе мы рассмотрим класс `BoxItem`, который множественно наследует классам `QObject` и `QGraphicsRectItem`. В нем реализовано больше возможностей, чем в классе `TextItem` из предыдущего подраздела. В частности, пользователь может перемещать элемент путем перетаскивания мышью или клавишами со стрелками, нажимаемыми одновременно с клавишей `Ctrl` (⌘ на платформе `Mac OS X`). Можно также изменять размер прямоугольника, перетаскивая его угол мышью с нажатой клавишей `Shift` или с помощью клавиш со стрелками, нажимаемых одновременно с клавишей `Shift`.

```
class BoxItem : public QObject, public QGraphicsRectItem
{
    Q_OBJECT
    Q_PROPERTY(QBrush brush READ brush WRITE setBrush)
    Q_PROPERTY(QPen pen READ pen WRITE setPen)
    ...

public:
    enum {Type = BoxItemType};
    explicit BoxItem(const QRect &rect, QGraphicsScene *scene);
    int type() const { return Type; }
    ...

signals:
    void dirty();

public slots:
    void setPen(const QPen &pen);
    void setBrush(const QBrush &brush);
    ...

protected:
    QVariant itemChange(GraphicsItemChange change, const QVariant &value);
    void keyPressEvent(QKeyEvent *event);
```

```

void mousePressEvent(QGraphicsSceneMouseEvent *event);
void mouseMoveEvent(QGraphicsSceneMouseEvent *event);
void mouseReleaseEvent(QGraphicsSceneMouseEvent *event);
...
};

```

Мы опустили свойства, методы доступа и слоты, относящиеся к углу поворота и коэффициентам сдвига, потому что они ничем не отличаются от имеющихся в классе `TextItem`.

Реализация конструктора (не показана) очень похожа на то, что мы видели в классе `TextItem`, только для объектов `BoxItem` устанавливается дополнительный флаг (`QGraphicsItem::ItemIsFocusable`), позволяющий им принимать события клавиатуры. И разумеется, мы передаем конструктору базового класса `QGraphicsRectItem` полученный от вызывающей программы прямоугольник. В классе `BoxItem` есть также закрытая переменная `m_resizing` типа `bool`, помимо переменных для хранения углов поворота и коэффициентов сдвига типа `double` (как в `TextItem`).

В классе `BoxItem` имеются открытые слоты для задания параметров пера и кисти, а также угла поворота и коэффициентов сдвига. Переопределены защищенные обработчики событий `itemChange()`, `keyPressEvent()`, `mousePressEvent()`, `mouseMoveEvent()` и `mouseReleaseEvent()`. Мы рассмотрим все обработчики событий, кроме `itemChange()`, который ничем не отличается от определенного в классе `TextItem`. Из слотов мы рассмотрим только `setBrush()`, потому что `setPen()` имеет точно такую же структуру, а `setAngle()` и `setShear()` идентичны одноименным слотам в классе `TextItem`. Начнем со слота `setBrush()`.

```

void BoxItem::setBrush(const QBrush &brush_)
{
    if (isSelected() && brush_ != brush()) {
        QGraphicsRectItem::setBrush(brush_);
        emit dirty();
    }
}

```

С виджетом `BrushWidget` соединены все элементы, обладающие свойством «кисть», но изменение кисти применяется только к выделенным и лишь в том случае, когда новая кисть отличается от старой. Если новая кисть установлена, мы генерируем сигнал `dirty()`, уведомляя об этом все заинтересованные объекты.

```

void BoxItem::keyPressEvent(QKeyEvent *event)
{
    if (event->modifiers() & Qt::ShiftModifier ||
        event->modifiers() & Qt::ControlModifier) {
        bool move = event->modifiers() & Qt::ControlModifier;
        bool changed = true;
        double dx1 = 0.0;
        double dy1 = 0.0;

```

```

double dx2 = 0.0;
double dy2 = 0.0;
switch (event->key()) {
    case Qt::Key_Left:
        if (move)
            dx1 = -1.0;
            dx2 = -1.0;
            break;
    ...
    default:
        changed = false;
}
if (changed) {
    setRect(rect().adjusted(dx1, dy1, dx2, dy2));
    event->accept();
    emit dirty();
    return;
}
}
QGraphicsRectItem::keyPressEvent(event);
}

```

Мы поддержали перемещение и изменение размеров элементов с помощью клавиатуры (соответственно Ctrl+клавиша со стрелкой, или &+клавиша со стрелкой в Mac OS X, и Shift+клавиша со стрелкой). (В стандартном графическом представлении QGraphicsView нажатие клавиши со стрелкой без модификатора означает прокрутку представления.) В обоих случаях мы вычисляем сдвиг по оси *X* или *Y* в зависимости от того, какая клавиша со стрелкой нажата, и соответственно изменяем прямоугольник элемента. Если нажатие клавиши обработано, то мы подтверждаем событие (чтобы Qt не обработала его второй раз), генерируем сигнал `dirty()`, поскольку перемещение и изменение размера – существенная модификация, и выходим. Иначе обработка клавиши поручается базовому классу (что в данном случае означает ее игнорирование).

```

void BoxItem::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    if (event->modifiers() & Qt::ShiftModifier) {
        m_resizing = true;
        setCursor(Qt::SizeAllCursor);
    }
    else
        QGraphicsRectItem::mousePressEvent(event);
}

```

Поскольку флаг `QGraphicsItem::ItemIsMovable` установлен, то пользователь может перемещать прямоугольник путем перетаскивания мышью. Мы добавили новое поведение, поддержав также изменение размера с помощью клавиатуры (это было показано выше) и перетаскивание с нажатой клавишей Shift.

Если пользователь щелкает мышью при нажатой клавише Shift, то мы переходим в режим изменения размера – устанавливаем для переменной `m_resizing` значение `true` и меняем форму указателя мыши.

```
void BoxItem::mouseMoveEvent(QGraphicsSceneMouseEvent *event)
{
    if (m_resizing) {
        QRectF rectangle = rect();
        if (event->pos().x() < rectangle.x())
            rectangle.setBottomLeft(event->pos());
        else
            rectangle.setBottomRight(event->pos());
        setRect(rectangle);
    }
    else
        QGraphicsRectItem::mouseMoveEvent(event);
}
```

Если мы находимся в режиме изменения размера, то смещаем левый нижний или правый нижний угол в точку, где находится указатель мыши. Более изощренный алгоритм обрабатывал бы координаты по осям *X* и *Y* отдельно, но и того, что мы сделали, достаточно для иллюстрации идеи.

```
void BoxItem::mouseReleaseEvent(QGraphicsSceneMouseEvent *event)
{
    if (m_resizing) {
        m_resizing = false;
        setCursor(Qt::ArrowCursor);
        emit dirty();
    }
    else
        QGraphicsRectItem::mouseReleaseEvent(event);
}
```

Как только пользователь отпустит кнопку мыши, мы выходим из режима изменения размера и восстанавливаем форму указателя мыши. Кроме того, генерируется сигнал `dirty()`, поскольку изменение размера элемента – существенная модификация.

Мы рассмотрели класс `BoxItem` почти целиком. Операторы ввода/вывода не показаны, потому что они очень похожи на те, что мы уже видели в классе `TextItem`. Отличие лишь в том, что вместо записи и считывания HTML-разметки мы записываем и читаем координаты, размеры, перо и кисть прямоугольника.

Создание пользовательского графического элемента с нуля

В этом подразделе мы рассмотрим производный от `QGraphicsItem` класс `SmileyItem`, в котором реализовано как поведение, так и внешний вид

элемента. В версии Qt 4.5 и более ранних для этого необходимо множественно унаследовать классам `QObject` и `QGraphicsItem`, а начиная с Qt 4.6 достаточно унаследовать только классу `QGraphicsObject`. Поскольку заголовочный файл довольно длинный, мы опустили свойства, методы доступа к ним и слоты (все это не отличается от уже рассмотренного в классе `BoxItem`), а также закрытые члены.

```
class SmileyItem : public QGraphicsObject
{
    ...

public:
    enum Face {Happy, Sad, Neutral};
    enum {Type = SmileyItemType};

    explicit SmileyItem(const QPoint &position, QGraphicsScene *scene);
    int type() const { return Type; }
    ...
    Face face() const { return m_face; }
    bool isShowingHat() const { return m_showHat; }
    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option, QWidget *widget);
    QRectF boundingRect() const;
    QPainterPath shape() const;

signals:
    void dirty();

public slots:
    ...
    void setFace(Face face);
    void setShowHat(bool on);
    void edit();

protected:
    QVariant itemChange(GraphicsItemChange change, const QVariant &value);
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent*) { edit(); }
    void contextMenuEvent(QGraphicsSceneContextMenuEvent*) { edit(); }
    ...
};
```

Помимо стандартного перечисления `Type`, мы включили еще одно перечисление, описывающее вид смайлика. А поскольку рисуем элемент мы сами, то должны переопределить методы `paint()`, `boundingRect()` и `shape()`. За поведение элемента отвечают различные слоты (показана только часть из них), которые устанавливают свойства, а также сигнал `dirty()` и слот `edit()`, соответствующие общим соглашениям, принятым в программе `Page Designer`.

Как обычно, мы переопределили метод `itemChange()`, чтобы при необходимости генерировать сигнал `dirty()`. Мы также предоставили два способа вызвать слот `edit()` элемента – двойной щелчок мышью, как принято в Page Designer, и выбор из контекстного меню, поскольку для этого элемента действие редактирования сводится к открытию контекстного меню.

Для запоминания пера, кисти, вида личика, признака показа шляпы, коэффициентов сдвига и углов поворота заведены закрытые переменные (не показаны). Кроме того, мы храним два графических пути; путь личика – это просто эллипс, а путь шляпы – более сложный многоугольник с восемью вершинами.

Конструктор (не показан) аналогичен конструктору `TextItem` с тем отличием, что мы инициализируем закрытые данные-члены (счастливое личико, нет пера, желтая кисть, шляпа не показывается, угол поворота 0.0° , коэффициенты сдвига 0.0) и вызываем вспомогательный метод `createPaths()`, который создает пути личика и шляпы.

```
const int SmileySize = 60;
...

void SmileyItem::createPaths()
{
    m_facePath.addEllipse(-SmileyHalfSize, -SmileyHalfSize,
                          SmileySize, SmileySize);

    const int LeftX = -(SmileyHalfSize + (SmileyMargin / 2));
    const int RightX = SmileyHalfSize - (SmileyMargin / 2);
    const int Y = -SmileyHalfSize + (SmileyMargin / 2);
    QPolygonF polygon;
    polygon << QPointF(LeftX * 1.4, Y + SmileyMargin)
    ...
    << QPointF(LeftX * 1.4, Y + SmileyMargin);
    m_hatPath.addPolygon(polygon);
}
```

Мы опустили большую часть вершин многоугольника шляпы и большинство констант. Как выглядит шляпа, показано на рис. 12.4.

```
void SmileyItem::setFace(Face face)
{
    if (m_face != face) {
        m_face = face;
        update();
        emit dirty();
    }
}
```

Этот метод изменяет личико. Если новое личико отличается от старого, то мы обновляем закрытую переменную, вызываем метод `update()`,

который планирует перерисовку, и генерируем сигнал `dirty()`, так как изменение личика – существенная модификация.

Мы опустили методы `setPen()`, `setBrush()`, `setAngle()`, `setShear()` и `itemChange()`, потому что они ничем не отличаются от приведенных в двух предыдущих подразделах.

```
void SmileyItem::setShowHat(bool on)
{
    if (m_showHat != on) {
        prepareGeometryChange();
        m_showHat = on;
        emit dirty();
    }
}
```

Этот слот показывает или скрывает шляпу. Понятно, что ограничивающий прямоугольник и форма элемента зависят от того, показывается шляпа или нет, поэтому мы *обязаны* уведомить графическое представление о том, что геометрия элемента изменилась. Это легко сделать, обратившись к методу `QGraphicsItem::prepareGeometryChange()`, который сам вызовет `update()`, так что нам это делать необязательно.

```
void SmileyItem::edit()
{
    QMenu menu;
    QAction *showHatAction = createMenuAction(&menu, QIcon(),
        tr("Show Hat"), m_showHat);
    connect(showHatAction, SIGNAL(triggered(bool)),
        this, SLOT(setShowHat(bool)));
    menu.addSeparator();
    QActionGroup *group = new QActionGroup(this);
    createMenuAction(&menu, QIcon(":/smileysmile.png"),
        tr("Happy"), m_face == Happy, group, Happy);
    ...
    AQP::accelerateMenu(&menu);
    QAction *chosen = menu.exec(QCursor::pos());
    if (chosen && chosen != showHatAction)
        setFace(static_cast<Face>(chosen->data().toInt()));
}
```

Для этого элемента действие редактирования открывает контекстное меню, в котором есть флажок для показа или сокрытия шляпы и пункты для изменения личика; меню показано на рис. 12.4.

Мы опустили создание нейтрального и грустного личика, поскольку структура кода такая же, как для улыбающегося личика, – во всех трех случаях большая часть работы поручается вспомогательному методу. Для действия `showHatAction` мы устанавливаем соединение сигнала со слотом `setShowHat()`, чтобы действие само позаботилось о себе. Однако для действий выбора личика не устанавливается вообще никаких соединений. Вместо этого, когда пользователь выбирает действие (а он не



Рис. 12.4. Приложение Page Designer: контекстное меню смайлика

обязан это делать и может просто закрыть меню, нажав клавишу Esc), мы смотрим, какое действие он выбрал, и если это не `showHatAction`, то оно должно быть одним из действий смены личика, поэтому мы вызываем слот `setFace()`. В методе `createMenuAction()` мы ассоциировали с каждым действием одно из значений перечисления `Face`, поэтому теперь можем его получить и узнать, какое личико выбрано.

```
QAction *SmileyItem::createMenuAction(QMenu *menu,
                                       const QIcon &icon, const QString &text, bool checked,
                                       QActionGroup *group, const QVariant &data)
{
    QAction *action = menu->addAction(icon, text);
    action->setCheckable(true);
    action->setChecked(checked);
    if (group)
        group->addAction(action);
    action->setData(data);
    return action;
}
```

Этот небольшой вспомогательный метод позволяет сэкономить несколько строк кода в слоте `edit()`. Объединение действий в группу гарантирует, что в каждый момент времени будет отмечено только одно действие.

Итак, мы рассмотрели все методы и слоты, отвечающие за поведение элемента (за исключением тех, что были рассмотрены в предыдущих подразделах). Операторы ввода/вывода почти такие же, как в классе `TextItem`, только вместо записи и считывания HTML-разметки мы пишем и читаем личико элемента (выполняя преобразование из перечисления в `qint16` и обратно), перо, кисть и признак показа шляпы. Осталось рассмотреть лишь рисование и реализацию методов `shape()` и `boundingRect()`.

```
void SmileyItem::paint(QPainter *painter,
                      const QStyleOptionGraphicsItem*, QWidget*)
{
    paintFace(painter);
    if (m_showHat)
        paintHat(painter);
    if (isSelected())
        paintSelectionOutline(painter);
}
```

Внеэкранное рисование

Производительность приложений, основанных на архитектуре графических представлений, зависит от того, что и как рисуется. Если для изображения пользовательского элемента необходимо рисовать сложные фигуры, текст, градиентную заливку или применять сложное отсечение, то приложение будет работать медленно.

Один из способов избежать медленной и сложной прорисовки состоит в том, чтобы создать изображение элемента в каком-нибудь графическом редакторе, а в переопределенном методе `QGraphicsItem::paint()` просто загрузить это изображение в `QPixmap` и нарисовать растр. В большинстве случаев рисование одного растра оказывается существенно быстрее классической поэлементной прорисовки. К сожалению, у этого подхода есть целый ряд ограничений, и самое очевидное из них то, что растр не может реагировать на изменение состояния (скажем, изменение цвета элемента), а после любого геометрического преобразования (например, масштабирования) изображение, скорее всего, станет нечетким или проявится пикселизация.

Более гибкое решение – воспользоваться внеэкранной кэшью, предоставляемым классом `QGraphicsItem`. По умолчанию режим `CacheMode` равен `QGraphicsItem::NoCache`, но если включить кэширование, обратившись к методу `QGraphicsItem::setCacheMode()`, то кэшированный элемент будет прорисовывать себя в растре. Размер растра можно установить, передав методу `setCacheMode()` второй аргумент типа `QSize`; если этого не сделать, то размер будет вычисляться, исходя из значения, возвращенного методом `boundingRect()` элемента. Затем этот растр будет использоваться при каждом последующем показе данного элемента, исключив тем самым обращения к `QGraphicsItem::paint()`. Кэш работает совершенно прозрачно; в частности, при вызове метода `update()` сначала обновится растр, а только потом элемент будет прорисован.

Можно, конечно, включить кэширование и для ранее рассмотренных элементов, например `QGraphicsTextItem`, если есть потребность ускорить их прорисовку. Однако не забывайте, что чрезмерное использование кэширования может привести к исчерпанию графической памяти компьютера.

У кэша есть два режима работы, определяющих, какая система координат используется. Если установить режим `ItemCoordinateCache`, то элемент будет прорисовываться в локальных координатах. В этом случае логические единицы измерения элемента преобразуются в пиксели, то есть качество видимого результата

ухудшается при масштабировании либо панорамировании элемента или представления – например, когда мы приближаем элемент, изображение распадается на пиксели и становится нечетким. В этом режиме кэшированное изображение не изменяется при преобразованиях. Он идеален для элементов, подвергаемых преобразованиям анимации, например поворотам. Его можно с успехом использовать, скажем, в приложениях OpenGL с большим количеством преобразований.

Во втором режиме кэширования, `DeviceCoordinateCache`, элемент прорисовывается в координатах устройства. В этом случае результат получается идеальным на уровне пикселей, поскольку преобразование приводит к регенерации растра в кэше. Фактически растр, хранящийся в кэше, остается неизменным до тех пор, пока мы только перемещаем элемент или прокручиваем представление, и генерируется заново, если элемент, к примеру, поворачивается. Следовательно, этот режим обеспечивает очень высокую скорость – намного большую, чем перерисовка элементов при перемещении, – и одновременно идеальную прорисовку.

Мы выделили в этом методе отдельные части, чтобы проявить логику верхнего уровня, не отвлекаясь на детали. Во всех случаях мы рисуем личико, затем, если необходимо, шляпу и наконец, если элемент выделен, также контур выделения.

```
void SmileyItem::paintFace(QPainter *painter)
{
    painter->setPen(m_pen);
    painter->setBrush(m_brush);
    painter->drawPath(m_facePath);
    int leftX = -SmileyHalfSize + SmileyMargin;
    int rightX = SmileyHalfSize - SmileyEyeWidth - SmileyMargin;
    paintEyes(painter, leftX, rightX);
    paintMouth(painter, leftX, rightX);
}
```

Для изображения личика мы рисуем графический путь (эллипс), а поверх него глазки и ротик.

```
void SmileyItem::paintEyes(QPainter *painter, int leftX, int rightX)
{
    int y = -SmileyHalfSize + qRound(SmileyMargin * 1.5);
    painter->setBrush(m_brush.color().darker());
    painter->drawEllipse(leftX, y, SmileyEyeWidth, SmileyEyeHeight);
    painter->drawEllipse(rightX, y, SmileyEyeWidth, SmileyEyeHeight);
}
```

Глазки – это просто два эллипса, нарисованных чуть более темной кистью, чем само личико.

```
void SmileyItem::paintMouth(QPainter *painter, int leftX, int rightX)
{
    int y = SmileyHalfSize - qRound(SmileyMargin * 1.1);
    int offset = 0;
    if (m_face == Neutral)
        offset = SmileyMargin;
    else {
        offset = SmileyMargin / 2;
        if (m_face == Happy)
            y -= SmileyMargin;
        else if (m_face == Sad)
            y += SmileyMargin / 2;
    }
    QPointF leftPoint(leftX + offset, y);
    QPointF rightPoint(rightX + SmileyEyeWidth - offset, y);
    QRectF mouthRect(leftPoint, rightPoint);
    mouthRect.setHeight(m_face == Neutral ? SmileyMargin / 2
                                           : SmileyMargin);
    if (m_face == Neutral)
        painter->drawRoundedRect(mouthRect, 5, 5);
    else if (m_face == Happy)
        painter->drawChord(mouthRect, 170 * 16, 200 * 16);
    else
        painter->drawChord(mouthRect, 30 * 16, 120 * 16);
}
```

Это самая сложная часть рисунка. Для нейтрального личика ротик рисуется в виде скругленного прямоугольника, а для улыбающегося и грустного – в виде сегмента, повернутого вверх или вниз. Метод `QPainter::drawRoundedRect()` принимает прямоугольник и радиусы эллипсов, определяющие скругление углов, а метод `QPainter::drawChord()` – прямоугольник, начальный угол и угол раствора, причем оба угла задаются в шестнадцатых долях градуса.

```
void SmileyItem::paintHat(QPainter *painter)
{
    QPen pen(m_pen);
    if (pen.style() != Qt::NoPen)
        pen.setColor(pen.color().lighter());
    painter->setPen(pen);
    QBrush brush(m_brush);
    if (brush.style() != Qt::NoBrush)
        brush.setColor(brush.color().lighter());
}
```

```

    painter->setBrush(brush);
    painter->drawPath(m_hatPath);
}

```

Если нужно нарисовать шляпу, то мы устанавливаем чуть более светлое перо (или обходимся без пера) и чуть более светлую кисть (или обходимся без кисти), а затем рисуем графический путь шляпы.

```

void SmileyItem::paintSelectionOutline(QPainter *painter)
{
    QPen pen(Qt::DashLine);
    pen.setColor(Qt::black);
    painter->setPen(pen);
    painter->setBrush(Qt::NoBrush);
    painter->drawPath(m_showHat ? m_facePath.united(m_hatPath)
                      : m_facePath);
}

```

Для визуализации выделения мы выбрали пунктирное черное косметическое (то есть немасштабируемое) перо шириной 1 пиксел, без кисти. Чтобы показать, что элемент выделен, мы рисуем этим пером путь личика, объединенный с путем шляпы (если шляпа отображается).

```

QRectF SmileyItem::boundingRect() const
{
    QRectF rect(-SmileyHalfSize, -SmileyHalfSize, SmileySize,
                SmileySize);
    if (m_showHat)
        rect = rect.united(m_hatPath.boundingRect());
    return rect;
}

```

Ограничивающий прямоугольник можно было бы вернуть, написав выражение `QRectF rect(m_facePath.boundingRect())`, однако мы решили остановиться на более быстром и простом вычислении, основанном на размере элемента. Но если показывается шляпа, то вычисленный прямоугольник нужно расширить, объединив его с прямоугольником шляпы.

```

QPainterPath SmileyItem::shape() const
{
    QPainterPath path;
    path.addPath(m_facePath);
    if (m_showHat)
        path.addPath(m_hatPath);
    return path;
}

```

Форма элемента вычисляется просто, потому что это не что иное, как путь личика, — сам по себе или объединенный с путем шляпы.

На этом мы завершаем рассмотрение приложения Page Designer вместе с использованными в нем графическими элементами. Программу можно развивать в разных направлениях, самое очевидное – добавление новых фигур. Это не так уж сложно сделать, потому что новые фигуры можно обрабатывать по образцу уже имеющихся: прямоугольника, смайлика и текста. Кроме того, неплохо было бы избавиться от элементов, образующих сетку, заменив их фоновой кистью или переопределив метод `QGraphicsView::drawBackground()`. И конечно, не стоит забывать об уже упоминавшихся функциях, например равномерном распределении элементов в указанном направлении. Хорошо бы также поддерживать перетаскивание значка элемента с панели инструментов на сцену. Разумеется, просто щелкнуть по значку проще, но перетаскивание позволило бы сразу поместить элемент в нужное место. Гораздо более масштабная и сложная задача – воспользоваться встроенной в Qt инфраструктурой отмены/повтора операций, чтобы пользователь мог отменять и повторять все свои действия¹. Все это мы оставляем читателю в качестве упражнения.

Здесь мы расстаемся с архитектурой графических представлений в Qt. В рассмотренных примерах мы применяли классический подход, при котором поведение и внешний вид виджетов (или графических элементов) реализуется самими виджетами (или элементами). Но если на сцене находятся тысячи или десятки тысяч элементов, то стоит задуматься о другом подходе. Вместо того чтобы делать каждый элемент подклассом `QObject`, мы можем воспользоваться средствами, предлагаемыми классами `QGraphicsScene` и `QGraphicsView` для определения поведения элементов. Например, можно создать подкласс сцены или представления, переопределить в нем обработчики событий клавиатуры и мыши, а затем с помощью различных вариантов метода `items()` узнавать о том, с какими элементами взаимодействовал пользователь, и выполнять соответствующие изменения. Архитектура графических представлений очень развита, и с каждой новой версией Qt ее быстродействие и качество повышаются. Разобраться и поэкспериментировать с ней весьма полезно, а демонстрационные программы и примеры, входящие в дистрибутив Qt, очень интересны.

¹ В дистрибутиве Qt есть два примера графических представлений с поддержкой отмены и повтора операций – *examples/tools/undoframework* (основы) и *demos/undo* (более развитые возможности). В документации по Qt приведен обзор инфраструктуры отмены/повтора операций. Кроме того, автор настоящей книги написал также работу, посвященную этой инфраструктуре с использованием PyQt4, ее можно найти по адресу www.informit.com/articles/article.aspx?p=1187104.

13

Анимация и конечные автоматы

- Введение в подсистему анимации
- Введение в подсистему конечных автоматов
- Сочетание анимации с конечными автоматами

В версии Qt 4.6 появилось множество нововведений, в том числе две крупные подсистемы: анимация (часть проекта «Kinetic») и конечные автоматы. Эта глава – краткое введение в обе подсистемы¹. В Qt всегда была возможна анимация с помощью таймеров, а после появления в версии Qt 4.2 класса `QTimeLine` эта задача значительно упростилась. (Мы мимоходом упоминали этот класс ранее, на стр. 153.) В новой подсистеме анимации в Qt 4.6 реализован более гибкий и развитый высокоуровневый подход к анимации.

В первом разделе мы рассмотрим подсистему анимации и немного модифицируем приложение `Page Designer` из главы 12 (`pagedesigner2`), чтобы анимировать выравнивание графических элементов, а не просто мгновенно помещать их в новые позиции. Во втором разделе мы ознакомимся с подсистемой конечных автоматов и приведем модифицированную версию приложения `Petri Dish` из главы 11 (`petridish2`), показав, как реализовать его логику с помощью конечного автомата, не делая все самостоятельно. И в последнем разделе на примере небольшого диалогового окна мы продемонстрируем совместное использование анимации и конечных автоматов; на этот раз мы будем анимировать стандартные виджеты, а не графические элементы, как в первом разделе.

¹ В отличие от предыдущих глав, представленные здесь примеры ориентированы на Qt 4.6 и не будут собираться с версией Qt 4.5.

Введение в подсистему анимации

Подсистема анимации в Qt – весьма развитый и сложный продукт, но понять заложенные в нее фундаментальные идеи нетрудно. Теоретическую основу составляют конечные диаграммы состояний Дэвида Харела (David Harel), а семантика выполнения конечного автомата основана на языке SCXML (State Chart XML). Практическая реализация базируется на классе `QObject` и встроенной в Qt системе свойств. Самый простой вариант применения состоит в том, чтобы создать объект `QPropertyAnimation` для каждого свойства объекта `QObject`, которое мы хотим анимировать, и задать для него продолжительность, начальное и конечное значения.

Все анимации свойств (объекты `QPropertyAnimation`) оперируют объектами `QVariant`, а сами свойства должны допускать запись (то есть должен быть определен метод установки свойства). В версии Qt 4.6 анимацию допускают следующие типы `QVariant`, для которых возможна интерполяция значений: `int`, `float`, `double`, `QColor`, `QLine`, `QLineF`, `QPoint`, `QPointF`, `QRect`, `QRectF`, `QSize` и `QSizeF`.

Например, можно определить продолжительность анимации 5000 мс, а начальное и конечное значения задать в виде объектов `QRect`. В начале анимации объект получает начальную геометрию, которая в течение 5 секунд переходит в конечную, причем промежуточные формы Qt вычисляет с помощью линейной интерполяции. Так, если начальная ширина составляет 100 пикселей, а конечная – 400 пикселей, то в конце первой секунды ширина составит 160 пикселей, в конце второй – 220, в конце третьей – 280, в конце четвертой – 340 и в конце пятой – 400 пикселей. (В данном случае приращение, равное 60 пикселям, вычисляется как частное от деления разности между конечным и начальным значениями на продолжительность, то есть $(400 - 100) / 5 = 60$. Разумеется, Qt оперирует гораздо меньшими единицами времени, чем секунда, так что в реальности переходы могут быть такими: 100 – 103 – 106 пикселей и т. д.)

Как видим, в простых случаях использовать анимацию совсем нетрудно, но мы можем добиться и более интересных эффектов. Во-первых, линейной интерполяцией дело не ограничивается, Qt предоставляет класс `QEasingCurve`, с помощью которого можно реализовать свыше 40 различных переходных кривых. Кроме того, анимации можно объединять в группы и выполнять последовательно или параллельно. Подсистему анимации можно использовать в сочетании как с архитектурой графических представлений (но только для элементов, производных от `QObject`), так и с обычными виджетами.

В этом разделе мы приведем очень короткий и простой пример, в котором добавим анимацию к средствам манипулирования несколькими графическими элементами, а в последнем разделе главы покажем более сложную анимацию виджетов.

В приложении Page Designer имеются действия выравнивания, с помощью которых пользователь может выровнять выделенные графические элементы по левому, правому, верхнему или нижнему краю. Выравнивание происходит мгновенно, то есть элементы перемещаются в новые позиции неумовимо для глаза. Хорошо бы предложить какую-то визуальную обратную связь, чтобы пользователь видел, какие элементы выравниваются и как происходит этот процесс. Конечно, не следует это дело затягивать, чтобы не раздражать пользователя.

Код выравнивания находится в слоте `editAlign()`, который обсуждался в разделе «Манипулирование выделенными элементами» в главе 12, стр. 465). Там мы разбили рассмотрение слота на три части и выравнивание поместили в последнюю. Теперь приведем новую версию последней части, но на этот раз вместо обращения к методу `QGraphicsItem::moveBy()` в цикле обхода элементов мы просто запомним новую позицию элемента.

```
QList<QPointF> positions;
if (alignment == Qt::AlignLeft || alignment == Qt::AlignRight) {
    for (int i = 0; i < items.count(); ++i)
        positions << items.at(i)->pos() +
            QPointF(offset - coordinates.at(i), 0);
}
else {
    for (int i = 0; i < items.count(); ++i)
        positions << items.at(i)->pos() +
            QPointF(0, offset - coordinates.at(i));
}
animateAlignment(items, positions);
setDirty(true);
```

В конце обоих циклов нам становится известна позиция, которую должен занять элемент. Теперь мы вызываем метод `animateAlignment()`, передавая ему два параллельных списка – элементов и новых позиций.

```
void MainWindow::animateAlignment(const QList<QGraphicsItem*>
                                &items, const QList<QPointF> &positions)
{
    int duration = ((qApp->keyboardModifiers() & Qt::ShiftModifier)
                    != Qt::ShiftModifier) ? 1000 : 5000;
    for (int i = 0; i < items.count(); ++i) {
        QObject *object = dynamic_cast<QObject*>(items.at(i));
        if (!object)
            continue;
        QPropertyAnimation *animation = new QPropertyAnimation(
            object, "pos", this);
        animation->setDuration(duration);
        animation->setEasingCurve(QEasingCurve::InOutBack);
        animation->setKeyValueAt(0.0, items.at(i)->pos());
        animation->setKeyValueAt(1.0, positions.at(i));
        animation->start(QAbstractAnimation::DeleteWhenStopped);
    }
}
```

Продолжительность анимации задается равной либо 1 с, либо 5 с, если нажата клавиша Shift. Одной секунды вполне достаточно при нормальной работе, но если во время разработки мы хотим лучше рассмотреть или продемонстрировать кому-то работу анимации, то можем нажать Shift и увеличить продолжительность до 5 с. (Отметим, что метод `QApplication::keyboardModifiers()` возвращает состояние модификаторов в момент последнего нажатия клавиши. Существует аналогичный метод `QApplication::mouseButtons()`.)

Мы перебираем все подлежащие выравниванию элементы и для каждого создаем объект `QPropertyAnimation`, передавая его конструктору объект `QObject`, который нужно анимировать (напомним, что все графические элементы в приложении Page Designer, за исключением линий сетки, – объекты, производные от `QObject`), имя анимируемого свойства и родителя. (В главе 12 отмечалось, что использование оператора `dynamic_cast<>()` делает приложение зависимым от поддержки RTTI, но существуют способы обойти это ограничение; стр. 457.)

Вероятно, вы помните, что при обсуждении пользовательских графических элементов в приложении Page Designer ни разу не упоминалось свойство `pos`. На самом деле мы его просто не показывали, потому что в тот момент оно было ни к чему, однако в объявлениях всех классов есть такая строка:

```
Q_PROPERTY(QPointF pos READ pos WRITE setPos)
```

Методы чтения и установки предоставляются базовым классом `QGraphicsItem`, поэтому для получения свойства `pos` нам больше ничего делать не нужно.

Создав анимацию свойства, мы задаем ее продолжительность, переходную кривую (это необязательно, по умолчанию подразумевается линейная интерполяция `QEasingCurve::Linear`), а также начальное и конечное значения свойства. Метод `QPropertyAnimation::setKeyValueAt()` задает значение свойства в конкретной точке анимации, причем 0.0 означает начальный, а 1.0 – конечный момент. В данном случае начальным значением свойства `pos` является текущая позиция элемента, а конечным – позиция после выравнивания. Можно при желании задать и промежуточные значения, например 0.5 или 0.3 и 0.6 и т. д. Если задаются только начальное и конечное значения, то можно было бы воспользоваться методами `QPropertyAnimation::setStartValue()` и `QPropertyAnimation::setEndValue()`, которые принимают один лишь аргумент типа `QVariant`, но мы предпочитаем метод `setKeyValueAt()` как более гибкий.

Настроив анимацию, мы вызываем метод `QPropertyAnimation::start()`, который запускает ее выполнение. По умолчанию объекты `QPropertyAnimation` остаются в памяти, пока не будет удален родитель, но после выравнивания нам эта анимация уже не нужна, поэтому мы передаем методу `start()` политику удаления, которая заставляет анимацию удалить себя по завершении.

В качестве переходной кривой мы задали `QEasingCurve::InOutBack`, поскольку при этом создается забавный эффект. Например, если пользователь выделит какие-то элементы и выберет действие `Align Left` (Выровнять по левому краю), то сначала все элементы сдвинутся *вправо* (в направлении, противоположном желаемому!), но только чуть-чуть. Затем они отправятся влево, но не до конечной позиции, а чуть дальше. И только на третьем шаге выровняются по левому краю с тем элементом, который изначально находился левее всех. А благодаря применяемой Qt интерполяции все эти перемещения происходят плавно. Выравнивание по левому краю иллюстрируется на рис. 13.1.

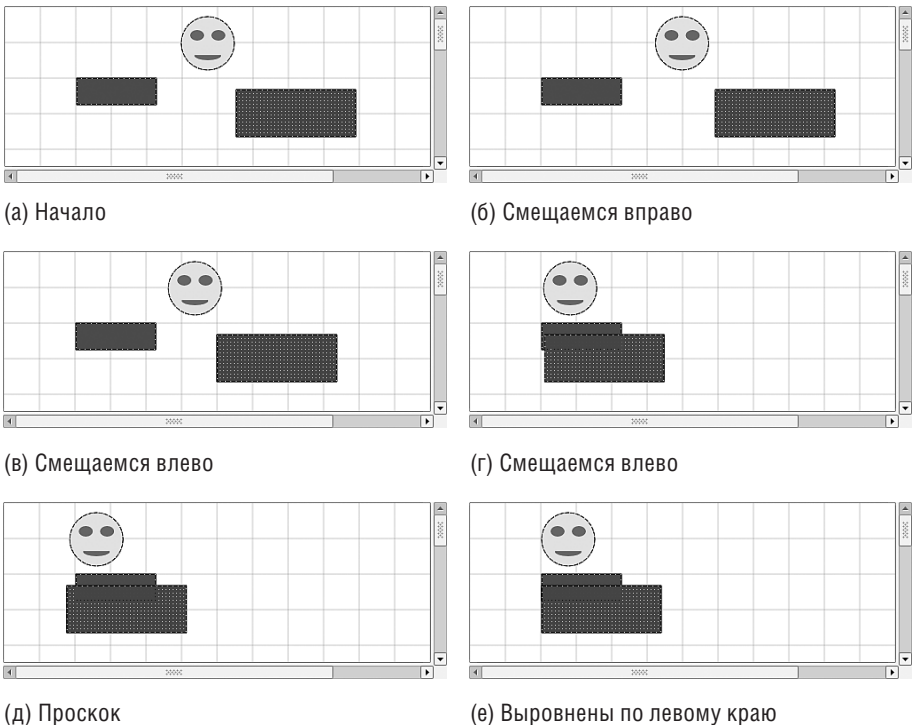


Рис. 13.1. Выравнивание по левому краю в соответствии с переходной кривой *InOutBack*

Еще стоит отметить тот факт, что анимации запускаются последовательно, друг за другом. На практике это не составляет проблемы, потому что метод `start()` возвращается почти мгновенно. Но можно было бы запустить анимации и параллельно, воспользовавшись классом `QParallelAnimationGroup`, как показано в следующей реализации метода `animateAlignment()`.

```
void MainWindow::animateAlignment(const QList<QGraphicsItem*>
                                &items, const QList<QPointF> &positions)
```

```

{
    int duration = ((qApp->keyboardModifiers() & Qt::ShiftModifier)
        != Qt::ShiftModifier) ? 1000 : 5000;
    QParallelAnimationGroup *group = new QParallelAnimationGroup;
    for (int i = 0; i < items.count(); ++i) {
        QObject *object = dynamic_cast<QObject*>(items.at(i));
        if (!object)
            continue;
        QPropertyAnimation *animation = new QPropertyAnimation(
            object, "pos", this);
        animation->setDuration(duration);
        animation->setEasingCurve(QEasingCurve::InOutBack);
        animation->setStartValue(items.at(i)->pos());
        animation->setEndValue(positions.at(i));
        group->addAnimation(animation);
    }
    group->start(QAbstractAnimation::DeleteWhenStopped);
}

```

Здесь вместо того, чтобы запускать каждую анимацию сразу после создания, мы добавляем ее в группу `QParallelAnimationGroup`, которую запускаем после выхода из цикла. Метод `start()` запускает все находящиеся в группе анимации одновременно, а благодаря политике удаления по завершении будет удалена как сама группа, так и все принадлежащие ей анимации. Отметим также, что мы воспользовались методами `setStartValue()` и `setEndValue()`; это удобнее в случаях, где никакие промежуточные значения не задаются.

Подсистема анимации в `Qt` проста в применении и очень эффективна для создания специальных эффектов. В последнем разделе этой главы мы снова прибегнем к ней, объединив ее с конечным автоматом для анимации стандартных виджетов `QWidget`.

Введение в подсистему конечных автоматов

Подсистема конечных автоматов предлагает средства для сохранения состояния в сложных приложениях. В простых диалоговых и главных окнах эта подсистема не дает ощутимых преимуществ и к тому же требует написания гораздо большего объема кода. Но по мере возрастания сложности конечные автоматы становятся все более привлекательной возможностью, поскольку прекрасно масштабируются и значительно упрощают управление сложным набором состояний по сравнению с ручным подходом.

В некоторых приложениях, например калькуляторах и медиаплеерах, наличие определенных состояний интерфейса является неотъемлемым свойством последнего. Но отслеживать состояние (градусы/радианы, воспроизводится/приостановлено и т. д.) с помощью многочисленных переменных достаточно трудно, особенно если состояния могут быть

вложенными. В таких случаях конечные автоматы – просто спасение, поскольку диаграмма переходов автомата может служить документацией, а сопровождение зачастую сводится к добавлению или удалению некоторых переходов.

Работать с подсистемой конечных автоматов в Qt просто, нужно лишь понимать несколько основных идей. Как и подсистема анимации, она в значительной мере опирается на класс `QObject` и систему свойств. Чтобы настроить конечный автомат, мы сначала создаем объект `QStateMachine`. Затем создаются нужные состояния (экземпляры класса `QState` или `QFinalState`) и для каждого состояния задаются тройки (`QObject`, свойство, значение), чтобы автомат знал, что в данном состоянии свойству объекта следует присвоить указанное значение. Настроив состояния, мы начинаем создавать переходы – они определяют, как автомат переходит из одного состояния в другое. Например, щелчок той или иной кнопкой мыши в некотором состоянии может перевести автомат в другое состояние.

При смене состояний прежнее состояние генерирует сигнал `exited()`, а новое – `entered()`. Когда (и если) автомат заканчивает работу, он генерирует сигнал `finished()`.

После того как все настроено, мы сообщаем автомату, в каком состоянии он должен начать работу, и вызываем метод `QStateMachine::start()`.

Даже в таком упрощенном описании конечный автомат представляет собой весьма мощный и гибкий механизм, но его функциональность этим далеко не исчерпывается. Например, состояния можно объединять в группы, историю перехода состояний можно запоминать для последующего сохранения и восстановления, есть возможность задания параллельных состояний и многое другое. К тому же подсистема не ограничивается только графикой, ее можно, к примеру, применять для моделирования сетевых коммуникационных протоколов.

В этом разделе мы продемонстрируем практическое применение конечного автомата на примере новой версии приложения *Petri Dish* (оно находится в каталоге *petridish2* и показано на рис. 11.3, стр. 425). Напомним, что в приложении есть четыре кнопки: `Start` (Пуск), `Pause/Resume` (Пауза/Продолжить), `Stop` (Стоп) и `Quit` (Выход), счетчик `Initial count` (Счетчик времени) и флажок `Show Ids` (Показать идентификаторы). Модель может работать, быть приостановлена или остановлена, причем в каждом случае должны быть активированы или деактивированы определенные виджеты, а на кнопке `Pause/Resume` еще и показана правильная надпись. В первоначальной версии (*petridish1*) эта задача решалась с помощью перечисления `SimulationState` и слотов `start()`, `pauseOrResume()` и `stop()`, которые не только определяли поведение модели, но также активировали или деактивировали виджеты и меняли надпись на кнопке `Pause/Resume`.

В конечном автомате *Petri Dish* перечисление нам не понадобится, слот `start()` будет нужен только для настройки модели, слот `pauseOrResume()`,

насчитывавший тринадцать строк, будет заменен слотом `pause()` из двух строк, а слот `stop()` из шести строк вообще исчезнет. И тем не менее файл *mainwindow.cpp* в новой версии оказывается примерно на сорок строк длиннее первоначального. И все из-за кода настройки конечного автомата. Однако по мере увеличения сложности или количества состояний мы рано или поздно достигнем точки, в которой версия на базе конечного автомата будет состоять из меньшего количества строк, потому что она лучше масштабируется.

Прежде чем приступить к написанию кода, составим план, в котором идентифицируем необходимые состояния и порядок переходов между ними. Имеется три очевидных состояния модели: остановлена, работает и приостановлена. Но помимо них мы создадим еще начальное состояние, в котором выполним всю настройку автомата на этапе запуска приложения, и конечное состояние, где проведем очистку и завершим приложение. А так как мы хотим, чтобы пользователь мог выйти из программы в любой момент, то создадим еще «нормальное» состояние, которое будет родителем состояний «остановлено», «работает» и «приостановлено». В результате вне зависимости от того, в каком состоянии находится автомат, нормальное состояние будет всегда активно и его можно использовать для перехода в конечное состояние.

Сразу после того как свойства начального состояния настроены, мы можем перейти в состояние «остановлено». Если пользователь нажмет кнопку *Start*, то мы перейдем в состояние «работает». В состоянии «работает» пользователь может нажать кнопку *Stop*, и в этом случае мы должны перейти в состояние «остановлено». А может нажать кнопку *Pause/Resume*, и тогда мы перейдем в состояние «приостановлено». В состоянии «приостановлено» пользователь может нажать кнопку *Stop*, что приведет к переходу в состояние «остановлено», или кнопку *Pause/Resume*, и тогда мы перейдем в состояние «работает». Поскольку состояния «остановлено», «работает» и «приостановлено» являются потомками нормального состояния, то после выхода из начального состояния автомат попадает в нормальное состояние (и некоторое его дочернее состояние, например, «работает»). Если пользователь нажмет кнопку *Quit*, когда автомат находится в нормальном состоянии (неважно, в каком из его дочерних), то мы должны перейти в конечное состояние. Все переходы изображены на рис. 13.2.

Уяснив, что требуется сделать, посмотрим на сам код. Начнем с нескольких фрагментов заголовочного файла.

```
Q_PROPERTY(bool running READ running WRITE setRunning)
...

private:
    ...
    bool running() const { return m_running; }
    void setRunning(bool running) { m_running = running; }
    ...
```

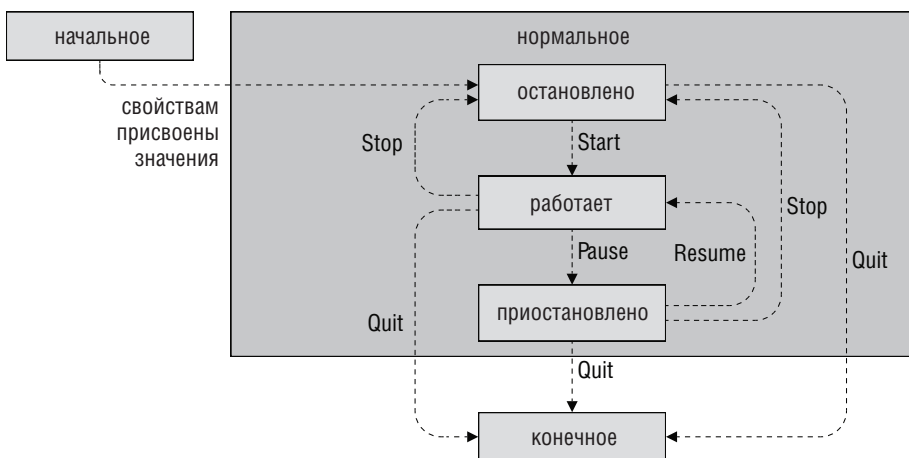


Рис. 13.2. Переходы состояний в приложении *Petri Dish*

```

QStateMachine stateMachine;
QState *initialState;
QState *normalState;
QState *stoppedState;
QState *runningState;
QState *pausedState;
QFinalState *finalState;
bool m_running;

```

Свойство `running` показывает, работает ли сейчас модель. Мы храним в закрытых переменных сам конечный автомат и по одному объекту `QState` для каждого состояния плюс объект `QFinalState` для конечного состояния¹. Кроме того, мы создали булеву переменную `m_running`, в которой храним значение свойства `running`.

```

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), iterations(0), m_running(false)
{
    scene = new QGraphicsScene(this);
    scene->setItemIndexMethod(QGraphicsScene::NoIndex);
    createWidgets();
    createProxyWidgets();
    createLayout();
    createCentralWidget();
}

```

¹ Из-за одной мелкой ошибки, проявляющейся в этом примере в версиях Qt 4.6.0–4.6.2 на платформах Windows и Mac OS X, мы были вынуждены сделать переменные-члены `QState` и `QFinalState` переменными модуля в *cpp*-файле с помощью директив `#if`, иначе приложение аварийно завершалось во время запуска.


```

createStates();
createTransitions();
createConnections();
setWindowTitle(tr("%1 (State Machine)")
               .arg(QApplication::applicationName()));
stateMachine.setInitialState(initialState);
QTimer::singleShot(0, &stateMachine, SLOT(start()));
}

```

Этот конструктор очень похож на конструктор из первоначальной версии, только на этот раз мы создаем состояния и переходы, устанавливаем некоторые соединения сигналов со слотами (не те, что были раньше), назначаем начальное состояние автомата и запускаем его. Как обычно, мы пользуемся однократным таймером, этот прием гарантирует, что к моменту начала обработки окно уже будет полностью сконструировано. Теперь рассмотрим методы `createStates()`, `createTransitions()` и `createConnections()`. Для простоты разобьем метод `createStates()` на четыре части.

```

void MainWindow::createStates()
{
    initialState = new QState(&stateMachine);
    initialState->assignProperty(showIdsCheckBox, "checked", true);
    initialState->assignProperty(initialCountSpinBox, "minimum", 1);
    initialState->assignProperty(initialCountSpinBox, "maximum", 100);
    initialState->assignProperty(initialCountSpinBox, "value", 60);
}

```

Прежде всего мы создаем первое состояние, делая его потомком конечного автомата, и с помощью метода `QState::assignProperty()` задаем тройки параметров для нескольких свойств. Отметим, что обращения к этому методу *не* присваивают свойствам указанного объекта заданные значения! Они лишь фиксируют тот факт, что при входе в соответствующее состояние свойствам указанных объектов должны быть присвоены заданные значения.

```

normalState = new QState(&stateMachine);
runningState = new QState(normalState);
runningState->assignProperty(startButton, "enabled", false);
runningState->assignProperty(initialCountSpinBox, "enabled", false);
runningState->assignProperty(stopButton, "enabled", true);
runningState->assignProperty(pauseOrResumeButton, "enabled", true);
runningState->assignProperty(pauseOrResumeButton, "text", tr("Pa&use"));
runningState->assignProperty(this, "running", true);

```

Нормальное состояние, как и начальное, создается как потомок самого конечного автомата. Единственное назначение нормального состояния – сгруппировать другие состояния (работает, приостановлено, остановлено), так чтобы при определении переходов можно было не думать о том, какое именно дочернее состояние нормального сейчас активно.

Для осуществления этой идеи мы должны указать, что родителем дочерних состояний является нормальное состояние, а не автомат.

В состоянии «работает» кнопка Start и счетчик Initial count должны быть неактивны, кнопки Stop и Pause/Resume доступны, причем на последней должна отображаться правильная надпись. Кроме того, свойству `running` главного окна присваивается значение `true`. Выше уже отмечалось, что все эти параметры применяются не в момент вызова `assignProperty()`, а лишь при входе в состояние.

```
pausedState = new QState(normalState);
pausedState->assignProperty(pauseOrResumeButton, "text", tr("Res&ume"));
pausedState->assignProperty(this, "running", false);
```

В состоянии «приостановлено» мы изменяем надпись на кнопке Pause/Resume и присваиваем свойству `running` главного окна значение `false`. В этом состоянии кнопка Start должна быть неактивна, однако явно мы для этого ничего не делаем (хотя могли бы без всякой опаски), поскольку эта кнопка неактивна уже при входе в состояние «работает», а в состояние «приостановлено» можно перейти только из состояния «работает», поэтому есть уверенность, что кнопка Start и так неактивна.

```
stoppedState = new QState(normalState);
stoppedState->assignProperty(startButton, "enabled", true);
stoppedState->assignProperty(initialCountSpinBox, "enabled", true);
stoppedState->assignProperty(pauseOrResumeButton, "enabled", false);
stoppedState->assignProperty(pauseOrResumeButton, "text", tr("Pa&use"));
stoppedState->assignProperty(stopButton, "enabled", false);
stoppedState->assignProperty(this, "running", false);
finalState = new QFinalState(&stateMachine);
}
```

В состоянии «остановлено» мы активируем кнопку Start и счетчик Initial count, деактивируем кнопки Pause/Resume (и меняем надпись на ней) и Stop. Кроме того, значение свойства `running` меняется на `false`.

Конечное состояние создано только для того, чтобы было куда перейти (из нормального состояния), когда пользователь завершает приложение, поскольку это избавляет нас от необходимости создавать отдельные переходы из состояний «работает», «приостановлено» и «остановлено».

```
void MainWindow::createTransitions()
{
    initialState->addTransition(initialState,
        SIGNAL(propertiesAssigned()), stoppedState);
    stoppedState->addTransition(startButton, SIGNAL(clicked()),
        runningState);
    runningState->addTransition(pauseOrResumeButton,
        SIGNAL(clicked()), pausedState);
}
```

```

runningState->addTransition(stopButton, SIGNAL(clicked()),
                           stoppedState);
pausedState->addTransition(pauseOrResumeButton,
                           SIGNAL(clicked()), runningState);
pausedState->addTransition(stopButton,
                           SIGNAL(clicked()), stoppedState);
normalState->addTransition(quitButton, SIGNAL(clicked()), finalState);
}

```

Первый переход создается из начального состояния в состояние «остановлено». Он производится сразу после задания свойств начального состояния. С точки зрения пользователя, этот переход происходит, когда приложение только запускается (на самом деле – при вызове слота `QStateMachine::start()` после выхода из конструктора окна), так что в начальное состояние мы попадаем только один раз, а потом приложение оказывается в нормальном состоянии, точнее, в одном из его дочерних (работает, приостановлено, остановлено).

Далее мы создаем переход из состояния «остановлено» в состояние «работает». А из состояния «работает» есть два перехода: в состояния «приостановлено» и «остановлено». Аналогично из состояния «приостановлено» имеются переходы в состояния «работает» и «остановлено». Кроме того, мы создаем переход из нормального состояния (которое включает всех своих потомков) в конечное.

Отметим, что при нажатии кнопки `Pause/Resume` может произойти один из двух переходов. Если мы находимся в состоянии «работает», то нажатие этой кнопки приведет к переходу в состояние «приостановлено», а если мы уже находимся в состоянии «приостановлено», – то к переходу в состояние «работает». Возможность интерпретировать нажатие одной и той же кнопки разными способами – весьма полезная особенность подсистемы конечных автоматов. (В первой версии приложения `Petri Dish` мы были вынуждены создать слот `pauseOrResume()`, соединить его с сигналом `clicked()` и в этом слоте решать, что делать – приостанавливать или возобновлять моделирование, – в зависимости от надписи на кнопке. Конечный автомат дает гораздо более простое и элегантное решение.)

```

void MainWindow::createConnections()
{
    connect(showIdsCheckBox, SIGNAL(toggled(bool)),
            this, SLOT(showIds(bool)));
    connect(runningState, SIGNAL(entered()), this, SLOT(start()));
    connect(pausedState, SIGNAL(entered()), this, SLOT(pause()));
    connect(&stateMachine, SIGNAL(finished()), this, SLOT(close()));
}

```

Первое соединение вполне традиционно, оно служит для переключения режима показа/сокрытия идентификаторов клеток.

Остальные три соединения относятся к работе конечного автомата. Соединение с сигналом `entered()` состояния «работает» означает, что помимо надлежащей установки свойств при входе в это состояние еще необходимо вызвать слот `start()`, который настроит модель и запустит ее выполнение. Аналогично при входе в состояние «приостановлено» в дополнение к установке свойств вызывается слот `pause()` (единственное назначение которого – сделать окно более прозрачным, что хорошо заметно в Windows и очень плохо в Mac OS X). Последнее соединение срабатывает при завершении работы автомата (то есть при входе в конечное состояние), соответствующий слот просто закрывает приложение.

Подсистема конечных автоматов в Qt обладает еще массой возможностей, помимо продемонстрированных в этом примере. Но, как бы то ни было, мы рассмотрели основные идеи и показали, как они применяются на практике. В следующем разделе мы создадим два конечных автомата и объединим один из них с подсистемой анимации, чтобы проиллюстрировать более сложные эффекты.

Сочетание анимации с конечными автоматами

В этом разделе мы рассмотрим приложение Find Dialog (`finddialog`), типичный пример диалогового окна, используемого для поиска. В этом окне применяются два независимых конечных автомата: один активирует или деактивирует кнопку Find (Найти) в зависимости от того, ввел ли пользователь искомый текст, а другой, работая в сочетании с подсистемой анимации, показывает или скрывает дополнительные виджеты в зависимости от состояния кнопки More (Больше).

При первом появлении диалогового окна кнопка More находится в состоянии «отжата» и дополнительные виджеты не видны. Когда пользователь щелкает по этой кнопке, она переходит в состояние «нажата», а размер окна при этом увеличивается, чтобы стали видны дополнительные виджеты. Сразу после увеличения окна из-под кнопки More «выезжают» дополнительные виджеты, поначалу очень маленькие и почти прозрачные. Но постепенно их шрифт и размер увеличиваются, они становятся все менее прозрачными и сдвигаются влево и вниз, пока не окажутся на своих конечных местах.

К моменту достижения нужного положения виджеты уже имеют правильный размер, изображаются правильным шрифтом и полностью непрозрачны. Этот процесс показан на рис. 13.3. Если пользователь еще раз нажмет кнопку More, то она возвратится в состояние «отжата», а дополнительные виджеты «уедут» под кнопку, постепенно сжимаясь в размерах, меняя шрифт на более мелкий и становясь прозрачными. В заключение диалоговое окно уменьшается, оставляя место только для видимых виджетов.

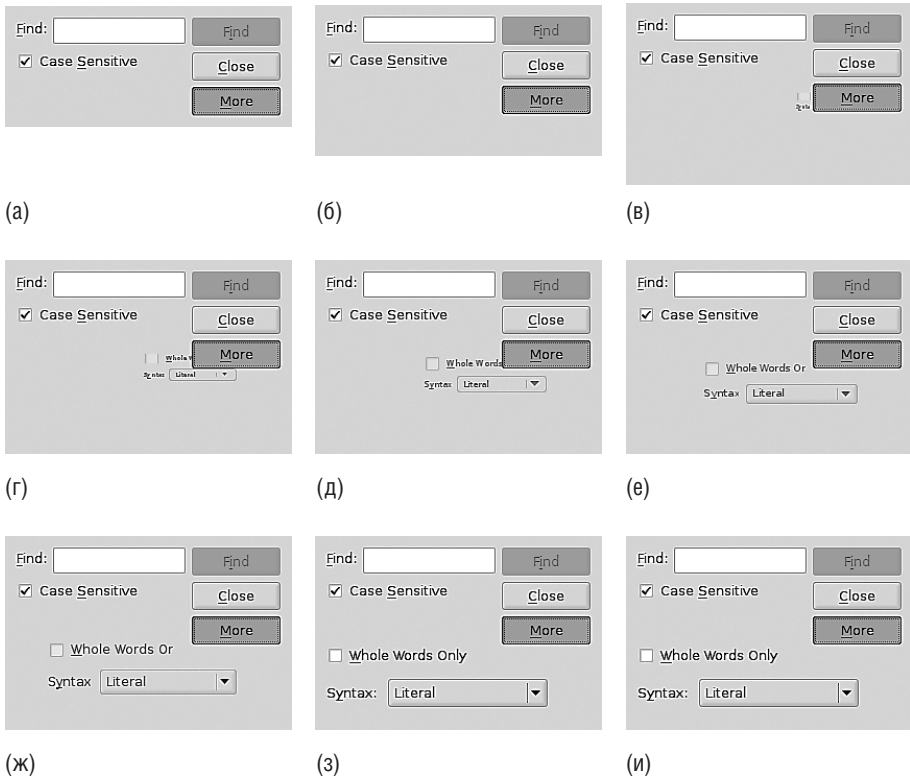


Рис. 13.3. Нажатие кнопки *More* в приложении *FindDialog*

Сначала приведем короткую выдержку из заголовочного файла диалогового окна, чтобы ознакомиться с закрытыми данными-членами.

```
QCheckBox *wholeWordsCheckBox;
QLabel *syntaxLabel;
QComboBox *syntaxComboBox;
QList<QWidget*> extraWidgets;
QStateMachine *findStateMachine;
QState *nothingToFindState;
QState *somethingToFindState;
QStateMachine *extraStateMachine;
QState *showExtraWidgetsState;
QState *hideExtraWidgetsState;
```

Мы опустили виджеты, которые видны всегда (поле ввода текста, кнопку Find и пр.), но показали дополнительные виджеты, которые видны или скрыты в зависимости от состояния кнопки More. Кроме того, мы храним список дополнительных виджетов, чтобы их можно было обрабатывать вместе.

У каждого конечного автомата всего два состояния, их имена не нуждаются в пояснениях.

Конструктор (не показан) устроен, как обычно: создает виджеты, компокует их, устанавливает соединения, а потом создает конечные автоматы и переходы. Вот фрагмент метода `createWidgets()`, в котором показаны создание и настройка дополнительных виджетов:

```
wholeWordsCheckBox = new QCheckBox(tr("Whole Words Only"), this);
wholeWordsCheckBox->setChecked(false);

syntaxLabel = new QLabel(tr("Syntax:"), this);

syntaxComboBox = new QComboBox(this);
syntaxLabel->setBuddy(syntaxComboBox);
syntaxComboBox->addItem(tr("Literal"), QRegExp::FixedString);
syntaxComboBox->addItem(tr("Regex"), QRegExp::RegExp2);
syntaxComboBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
syntaxComboBox->setCurrentIndex(0);
extraWidgets << wholeWordsCheckBox << syntaxLabel << syntaxComboBox;

foreach (QWidget *widget, extraWidgets) {
    QGraphicsOpacityEffect *effect = new QGraphicsOpacityEffect;
    effect->setOpacity(1.0);
    widget->setGraphicsEffect(effect);
}
```

Класс `QGraphicsEffect` и его подклассы `QGraphicsBlurEffect`, `QGraphicsColorizeEffect`, `QGraphicsDropShadowEffect` и `QGraphicsOpacityEffect` появились в версии Qt 4.6. Их можно использовать и с элементами `QGraphicsItem` на сцене `QGraphicsScene`, и с обычными виджетами `QWidget`, как в данном случае. Для каждого дополнительного виджета мы добавили эффект изменения прозрачности, установив начальное значение 1.0 (абсолютно непрозрачный).

```
void FindDialog::createStateMachines()
{
    findStateMachine = new QStateMachine(this);
    createFindStates();
    createFindTransitions();
    findStateMachine->setInitialState(nothingToFindState);
    findStateMachine->start();
    extraStateMachine = new QStateMachine(this);
    createShowExtraWidgetsState();
    createHideExtraWidgetsState();
    createShowExtraWidgetsTransitions();
    createHideExtraWidgetsTransitions();
    extraStateMachine->setInitialState(hideExtraWidgetsState);
    extraStateMachine->start();
}
```

Этот метод служит для создания, настройки и запуска обоих конечных автоматов. Для автомата `extraStateMachine`, управляющего дополнительными виджетами, мы определяем состояния «видны» и «скрыты» и переходы между ними в отдельных вспомогательных методах, чтобы каждый метод оставался обозримым.

```
void FindDialog::createFindStates()
{
    nothingToFindState = new QState(findStateMachine);
    nothingToFindState->assignProperty(findButton, "enabled", false);
    somethingToFindState = new QState(findStateMachine);
    somethingToFindState->assignProperty(findButton, "enabled", true);
}
```

Состояния, относящиеся к поиску, очень просты: кнопка Find либо активна, либо нет.

```
void FindDialog::createFindTransitions()
{
    nothingToFindState->addTransition(this,
        SIGNAL(findTextIsNonEmpty()), somethingToFindState);
    somethingToFindState->addTransition(this,
        SIGNAL(findTextIsEmpty()), nothingToFindState);
}
```

Переходы между этими состояниями производятся в зависимости от того, есть в поле критерия поиска какой-нибудь текст или нет; они тоже очень просты.

Можно было бы реализовать управление свойством `enabled` кнопки Find в слоте `updateUi()`, как мы делали во всех предыдущих примерах, и при этом потребовалось бы написать гораздо меньше кода. Но применение конечного автомата позволяет четко разнести логику по отдельным состояниям, что делает задачу сопровождения и добавления новых состояний существенно более простой и менее подверженной ошибкам по сравнению с запутанным кодом соединений между сигналами и слотами.

```
void FindDialog::createConnections()
{
    connect(findTextLineEdit, SIGNAL(textEdited(const QString&)),
        this, SLOT(findTextChanged(const QString&)));
    connect(moreButton, SIGNAL(toggled(bool)),
        this, SLOT(showOrHideExtra(bool)));
    connect(buttonBox, SIGNAL(rejected()), this, SLOT(close()));
}
```

При изменении искомого текста мы должны генерировать сигнал `findTextIsEmpty()` или `findTextIsNonEmpty()`. Для этого мы соединяем сигнал `textEdited()` поля ввода текста с написанным нами слотом `findTextChanged()`, который уже генерирует соответствующий сигнал в зависимости от содержимого поля.

У кнопок нет сигналов «нажата» и «отжата», а только сигналы `clicked` (`bool`) и `toggled`(`bool`), но нам-то нужны сигналы `showExtra()` и `hideExtra()`, генерируемые в зависимости от того, в каком состоянии находится кнопка `More`. Поэтому мы соединяем сигнал `toggled()` от кнопки `More` с нашим слотом `showOrHideExtra()`, который генерирует нужный сигнал в зависимости от состояния кнопки.

Третий сигнал используется для закрытия диалогового окна.

```
void FindDialog::findTextChanged(const QString &text)
{
    if (text.isEmpty())
        emit findTextIsEmpty();
    else
        emit findTextIsNonEmpty();
}
```

Этот слот преобразует сигнал `QLineEdit::textEdited()` в один из сигналов `findTextIsEmpty()` или `findTextIsNonEmpty()`, а они уже применяются конечным автоматом, управляющим кнопкой поиска, для перехода из состояния «нечего искать» в состояние «есть что искать».

```
void FindDialog::showOrHideExtra(bool on)
{
    if (on)
        emit showExtra();
    else
        emit hideExtra();
}
```

Этот слот преобразует сигнал `QPushButton::toggled()` в один из сигналов `showExtra()` или `hideExtra()` в зависимости от состояния кнопки `More`. Как используются эти сигналы, мы увидим ниже, при рассмотрении переходов автомата, управляющего показом дополнительных виджетов. Но сначала посмотрим, как задаются состояния этого автомата.

```
void FindDialog::createShowExtraWidgetsState()
{
    QSize size = extraSize();
    size.rheight() += minimumSizeHint().height();
    size.setWidth(qMax(size.width(), minimumSizeHint().width()));

    QList<QRectF> rects;
    int y = sizeHint().height() - margin;
    rects << QRectF(margin, y,
                     wholeWordsCheckBox->sizeHint().width(),
                     wholeWordsCheckBox->sizeHint().height());
    y += wholeWordsCheckBox->sizeHint().height() + margin;
    int height = qMax(syntaxLabel->sizeHint().height(),
                     syntaxComboBox->sizeHint().height());
    int width = syntaxLabel->sizeHint().width();
    rects << QRectF(margin, y, width, height);
}
```



```

int x = margin + syntaxLabel->sizeHint().width() + margin;
width = qMin(sizeHint().width(), size.width()) - (x + margin);
rects << QRectF(x, y, width, height);

showExtraWidgetsState = new QState(extraStateMachine);
foreach (QWidget *widget, extraWidgets) {
    showExtraWidgetsState->assignProperty(
        widget, "geometry", rects.takeFirst());
    showExtraWidgetsState->assignProperty(
        widget, "font", font());
    showExtraWidgetsState->assignProperty(
        widget->graphicsEffect(), "opacity", 1.0);
}
showExtraWidgetsState->assignProperty(this, "minimumSize", size);
}

```

В состоянии `showExtraWidgetsState` мы должны задать геометрию (позицию и размер) и шрифт каждого дополнительного виджета и установить степень прозрачности равной 1.0 (абсолютно непрозрачный). Кроме того, размер диалогового окна должен быть таким, чтобы хватило места для всех дополнительных виджетов, но не больше. Все это делается в показанном выше методе.

Сначала мы вычисляем ширину и высоту, необходимые для показа дополнительных виджетов, и создаем объект `size` типа `QSize`, который учитывает эти величины и представляет минимальный размер увеличенного диалогового окна. Затем создается список объектов `QRectF`, в которых хранятся геометрические размеры каждого дополнительного виджета. Закончив вычисления, мы создаем состояние `showExtraWidgetsState`. Далее для каждого дополнительного виджета в этом состоянии мы записываем в свойство `geometry` соответствующий вычисленный прямоугольник, а в свойство `font` — шрифт диалогового окна. Кроме того, свойству `opacity`, определяющему непрозрачность окна, присваивается значение 1.0. (Как обычно, в этот момент значения свойств не устанавливаются; они будут установлены только при входе в соответствующее состояние.) Наконец мы устанавливаем минимальный размер диалогового окна, необходимый для показа всех дополнительных виджетов.

```

QSize FindDialog::extraSize() const
{
    const int Width = syntaxLabel->sizeHint().width() +
        syntaxComboBox->sizeHint().width() + (2 * margin);
    const int Height = wholeWordsCheckBox->sizeHint().height() +
        qMax(syntaxLabel->sizeHint().height(),
            syntaxComboBox->sizeHint().height()) + (2 * margin);
    return QSize(Width, Height);
}

```

Этот вспомогательный метод вычисляет необходимую дополнительную ширину, суммируя значения ширины метки `syntaxLabel`, комбинирую-

ванного списка `syntaxComboBox` и поля. Кроме того, он подсчитывает дополнительную высоту, суммируя значения высоты флажка `wholeWordsCheckBox` и максимума из высот метки `syntaxLabel` и комбинированного списка `syntaxComboBox`, опять же добавляя небольшое поле).

```
void FindDialog::createHideExtraWidgetsState()
{
    QRectF rect = QRectF(buttonBox->x() + (moreButton->width()/2),
        buttonBox->height() - (moreButton->height()/2), 1, 1);
    QFont smallFont(font());
    smallFont.setPointSizeF(1.0);
    hideExtraWidgetsState = new QState(extraStateMachine);
    foreach (QWidget *widget, extraWidgets) {
        hideExtraWidgetsState->assignProperty(
            widget, "geometry", rect);
        hideExtraWidgetsState->assignProperty(
            widget, "font", smallFont);
        hideExtraWidgetsState->assignProperty(
            widget->graphicsEffect(), "opacity", 0.0);
    }
    hideExtraWidgetsState->assignProperty(
        this, "minimumSize", minimumSizeHint());
}
```

Состояние `hideExtraWidgetsState` несколько проще, чем `showExtraWidgetsState`, поскольку для всех дополнительных виджетов задаются одинаковые значения свойств. В качестве значения свойства `geometry` мы выбираем прямоугольник 1×1, расположенный в центре кнопки `More`, в качестве значения свойства `font` — мелкий шрифт, а в свойство `opacity` записываем 0.0 (абсолютно прозрачный). И наконец в свойство `minimumSize` мы записываем минимальный предпочтительный размер диалогового окна (который не учитывает дополнительных виджетов).

```
const int Duration = 1500;

void FindDialog::createShowExtraWidgetsTransitions()
{
    QSignalTransition *transition =
        hideExtraWidgetsState->addTransition(this,
            SIGNAL(showExtra()), showExtraWidgetsState);

    createCommonTransitions(transition);

    QPropertyAnimation *animation;
    animation = new QPropertyAnimation(this, "minimumSize");
    animation->setDuration(Duration / 3);
    transition->addAnimation(animation);

    animation = new QPropertyAnimation(this, "size");
    animation->setDuration(Duration / 3);
    QSize size = extraSize();
```

```

        size = QSize(qMax(size.width(), width()),
                     sizeHint().height() + size.height());
        animation->setEndValue(size);
        transition->addAnimation(animation);
    }

```

Мы создаем переход из состояния `hideExtraWidgetsState` в состояние `showExtraWidgetsState`, который срабатывает при получении сигнала `showExtra()`. Мы сохраняем указатель на этот переход, чтобы его можно было использовать при добавлении анимаций.

При этом переходе мы хотим контролировать пять свойств: минимальный и обычный размер диалогового окна, а также геометрию, шрифт и степень прозрачности каждого дополнительного виджета. Работу, относящуюся к дополнительным виджетам, мы вынесли во вспомогательный метод, поскольку точно такая же анимация свойств применяется и для сокрытия дополнительных виджетов.

Для свойств диалогового окна `minimumSize` и `size` мы задаем анимации, продолжительность которых в три раза меньше продолжительности анимаций свойств виджетов (то есть окно достигает конечного размера быстрее). Для свойства `size` конечное значение задается достаточно большим для показа всех дополнительных виджетов.

```

void FindDialog::createCommonTransitions(
    QSignalTransition *transition)
{
    QPropertyAnimation *animation;
    foreach (QWidget *widget, extraWidgets) {
        animation = new QPropertyAnimation(widget, "geometry");
        animation->setDuration(Duration);
        transition->addAnimation(animation);
        animation = new QPropertyAnimation(widget, "font");
        animation->setDuration(Duration);
        transition->addAnimation(animation);
        if (QGraphicsOpacityEffect *effect =
            static_cast<QGraphicsOpacityEffect*>(
                widget->graphicsEffect())) {
            animation = new QPropertyAnimation(effect, "opacity");
            animation->setDuration(Duration);
            animation->setEasingCurve(QEasingCurve::OutInCirc);
            transition->addAnimation(animation);
        }
    }
}

```

Вне зависимости от того, показываем мы или скрываем дополнительные виджеты, анимации свойств одни и те же. Для свойств `geometry` и `font` (равно как для анимации свойств диалогового окна) используется линейная переходная кривая (поскольку явно мы ее не задали, а по умолчанию подразумевается линейная интерполяция), а для степени

прозрачности – кривая `QEasingCurve::InOutCirc`, которая сначала изменяется медленно, затем быстро, а в конце снова медленно.

Все заданные нами анимации изменяют свойство от начального до конечного значения. Однако те же самые анимации мы используем и для перехода из состояния, в котором дополнительные виджеты скрыты (`hideExtraWidgetsState`), в состояние, где они показываются (`showExtraWidgetsState`), и обратно. Это работает, потому что для обоих состояний мы определили значения свойств; например, в состоянии `hideExtraWidgetsState` свойство `opacity` равно `0.0`, а в состоянии `showExtraWidgetsState` – `1.0`. Поэтому анимация происходит в направлении от `0.0` к `1.0` или от `1.0` к `0.0` в зависимости от того, какой переход имеет место, то есть в одном случае начальным будет значение `0.0` (а конечным – `1.0`), а в другом – `1.0` (и соответственно `0.0`).

```
void FindDialog::createHideExtraWidgetsTransitions()
{
    QSignalTransition *transition =
        showExtraWidgetsState->addTransition(this,
        SIGNAL(hideExtra()), hideExtraWidgetsState);

    createCommonTransitions(transition);

    QPropertyAnimation *animation = new QPropertyAnimation(this,
        "size");
    animation->setDuration(Duration);
    animation->setEndValue(sizeHint());
    transition->addAnimation(animation);
}
```

При переходе из состояния `showExtraWidgetsState` в состояние `hideExtraWidgetsState` используются те же анимации, что и при переходе в обратном направлении. И, кроме того, мы анимируем свойство `size` диалогового окна – на этот раз задавая такую же продолжительность, как для анимации виджетов, чтобы размер не уменьшался слишком быстро и у виджетов было достаточно времени, чтобы сжаться и скрыться.

Выше мы уже отмечали, что все анимации свойств оперируют объектами типа `QVariant`, а сами свойства должны допускать запись. Есть также возможность задать специальные интерполяторы для тех типов `QVariant`, которые в настоящий момент не поддерживают анимацию. Одно из решений – создать подкласс класса `QVariantAnimation` (базового для класса `QPropertyAnimation`) и переопределить метод `interpolated()`. Другой подход – реализовать функцию интерполяции, принимающую начальное и конечное значения того типа, который мы собираемся интерполировать, и значение продвижения типа `qreal`, а возвращающую интерполированное значение в виде `QVariant`. Эту функцию необходимо зарегистрировать с помощью глобальной функции `qRegisterAnimationInterpolator<>()`.

Возможно, вы обратили внимание на то, что Qt не поддерживает интерполяцию объектов QFont, но это не помешало нам интерполировать свойство font дополнительных виджетов от одного пункта до размера шрифта по умолчанию. Все дело в функции fontInterpolator() которая находится в файле *finddialog.cpp* и зарегистрирована в конструкторе класса FindDialog.

```
QVariant fontInterpolator(const QFont &start, const QFont &end,
                          qreal progress)
{
    qreal startSize = start.pointSizeF();
    qreal endSize = end.pointSizeF();
    qreal newSize = startSize + ((endSize - startSize) *
                                qBound(0.0, progress, 1.0));

    QFont font(start);
    font.setPointSizeF(newSize);
    return font;
}
```

Эта функция возвращает шрифт, размер которого изменяется от начального до конечного. Значение progress обычно изменяется в диапазоне от 0.0 до 1.0, но для некоторых переходных кривых может принимать отрицательные или превышающие 1.0 значения, что необходимо учитывать. Здесь мы принудительно ограничили минимальное и максимальное значения progress, поскольку отрицательный размер шрифта не имеет смысла. Но можно было бы написать qMax(0.0, progress), то есть ограничить минимальный размер шрифта значением 0.0 и при этом разрешить ситуацию, когда промежуточный шрифт превышает конечный по размеру.

Чтобы функция интерполяции возымела эффект, ее следует зарегистрировать. Вот соответствующая строка из кода конструктора FindDialog:

```
qRegisterAnimationInterpolator<QFont>(fontInterpolator);
```

Аргумент шаблона сообщает Qt, что эта функция будет использоваться в качестве интерполятора для объектов указанного типа – в данном случае QFont.

На этом мы завершаем рассмотрение приложения Find Dialog. Было бы нетрудно добавить какой-нибудь анимационный эффект и для кнопки Find, например эффект размывания. Анимация перемещения, изменения размера и прозрачности виджетов работает хорошо, но с одним ограничением: во время перехода виджеты сохраняют присущую им прямоугольную форму. Было бы куда интереснее, если бы виджеты могли деформироваться – например, в процессе исчезновения правые края сжимались быстрее левых; тогда, пропадая, они принимали бы форму воронки, создавая «эффект джинна»¹.

¹ Дополнительные сведения об «эффекте джинна» см. в статье по адресу labs.qt.nokia.com/blogs/2008/12/15/genie-fx.

Мы увидели, как можно использовать подсистему конечных автоматов для управления состоянием виджетов и как в сочетании с подсистемой анимации она позволяет создавать плавные визуальные переходы из одного состояния в другое. Для создания анимационных эффектов нужно написать больше кода (и тем самым увеличить вероятность появления ошибок), к тому же на них тратится процессорное время. Однако подобно тому, как в современных графических интерфейсах стало нормой использование большого количества разнообразных пиктограмм (например, в меню и на панелях инструментов), растет и применение анимации в приложениях. Ну и, конечно, анимация приносит вполне конкретную прагматическую пользу: она помогает пользователям лучше понять, что происходит (как, например, в случае анимации выравнивания графических элементов в предыдущем разделе). А в случае мгновенного изменения пользователь может не понять, что операция уже выполнена, и попытается повторить ее.

Эпилог

В этой книге представлено широкое разнообразие технологий и приемов программирования с использованием библиотеки Qt, хотя мы рассмотрели только основные аспекты реализованной в ней функциональности. Все приведенные в книге примеры – *hpp*- и *cpp*-файлы с кодом, *qrc*-файлы ресурсов и *pro*-файлы проектов – были написаны в обычном текстовом редакторе, программа *Qt Designer* не использовалась вовсе. Но в наши дни разработка с использованием Qt может и не быть столь аскетической. К услугам тех, кто любит проектировать окна визуально, есть инструмент *Qt Designer*, а предпочитающие полностью интегрированную среду разработки (IDE) могут воспользоваться программой *Qt Creator* (в которую *Qt Designer* уже интегрирован). Все библиотеки и основные инструменты входят в стандартный дистрибутив Qt, а полный комплект ПО, включая *Qt Creator*, – в дистрибутив Qt SDK.

Сама библиотека Qt уже содержит богатейшую функциональность, но тем не менее для нее имеются дополнительные компоненты. Некоторые из них можно найти на сайте Qt Development Frameworks, например решения Qt Solutions, многие из которых поставляются на условиях лицензии LGPL. Там вы найдете многочисленные дополнительные виджеты и служебные классы; см. qt.nokia.com/products/appdev/add-on-products/catalog/4. Компоненты предоставляют также сторонние разработчики. Библиотека Qwt (qwt.sourceforge.net) включает виджеты и классы, особенно полезные для научных и инженерных приложений. Библиотека LibQxt library (www.libqxt.org) содержит разнообразные служебные модули и классы, в том числе интерфейс с библиотекой Berkeley DB и широкий спектр дополнительных виджетов. На сайте qt-apps.org поддерживается репозиторий сторонних подключаемых к Qt модулей, в котором хранится обширное собрание компонентов и виджетов.

Больше узнать о Qt помогут многочисленные книги – на сайте Qt Development Frameworks имеется их список по адресу qt.nokia.com/developer/books. Кроме того, компания ICS (Integrated Computer Solutions, Inc.; www.ics.com/learning/icsnetwork) регулярно публикует бесплатные онлайн-вебкасты, в которых рассказывается о различных технологиях Qt и сообщается о том, что нового появилось в последних версиях. Большая часть выступлений на конференциях Qt Developer Days, организуемых в рамках проекта Qt Development Frameworks, сняты на видео и доступны для просмотра в Сети; особый интерес представляют основные доклады и технические заседания (qt.nokia.com/

developer/learning/online/talks). Еще один полезный источник информации о Qt – бесплатный онлайн-журнал *Qt Quarterly*, в котором публикуются короткие статьи об актуальных вопросах программирования с использованием Qt (qt.nokia.com/doc/qj/index.html), хотя для ознакомления публики с самой свежей информацией есть также корпоративный блог по адресу blog.qt.nokia.com и – что еще лучше – блог разработчиков Qt Labs по адресу labs.qt.nokia.com/blogs. Вопросы можно задавать в списке рассылки *qt-interest*; трафик в нем очень высокий, и есть поистине блистательные авторы. Только не забудьте сначала «погуглить» и ознакомиться с документацией, иначе на вас могут наброситься с обвинениями!

На сайте Qt Development Frameworks имеется перспективный план (qt.nokia.com/developer/qt-roadmap), в котором описывается, в каком направлении развивается Qt. Поскольку компанией владеет Nokia, нетрудно догадаться, что в ближайшие планы входит поддержка сенсорных экранов и разработка новых API, относящихся к мобильным телефонам, в том числе для обмена сообщениями и мобильных служб. Но немало интересного ждет и разработчиков настольных приложений.

Самое крупное нововведение, планируемое для включения в версию Qt 4.7¹, – Qt Quick (Qt User Interface Creation Kit). Оно знаменует совершенно новую парадигму в создании пользовательских интерфейсов. В Qt Quick используется основанный на JavaScript декларативный язык QML (Qt Meta-Object Language), где в полной мере задействованы подсистемы анимации и конечных автоматов. Он позволяет без особого труда создавать превосходные интерфейсы. Технология Qt Quick гораздо более гибкая, чем традиционный подход на базе виджетов и компоновщиков, в частности, она позволяет легко применять к виджетам анимированные преобразования. Лучше всего Qt Quick подходит в тех случаях, когда у приложения должны быть разные интерфейсы, зависящие от устройства и форм-фактора. Традиционный же подход на основе QWidget идеален для случая, когда необходим единый интерфейс на всех платформах, который бы тесно интегрировался с платформенным внешним обликом. Таким образом, Qt Quick – решение для потребительской электроники и встраиваемых приложений, а QWidget – для приложений на рабочем столе.

В среднесрочной и долгосрочной перспективе планируется большой объем работы в области графики. Значительные усилия прилагаются к разработке более простого и ориентированного на Qt трехмерного API в виде слоя, обертывающего OpenGL API (который также останется полностью доступным). Задача состоит в том, чтобы абстрагировать насколько возможно сложность и платформенные зависимости OpenGL API.

¹ На момент подготовки русскоязычного издания книги эти планы уже были претворены в жизнь. – *Прим. науч. ред.*

В планах отсутствуют кое-какие темы, которые, пожалуй, должны там быть. Архитектура отмены и повтора операций должна быть полностью интегрирована с архитектурой графических представлений. В настоящее время реализовать отмену и повтор операций для моделей совсем не просто и, даже если это сделано, мы должны очень внимательно следить за тем, когда употреблять собственные методы для некоторых операций, а когда стандартные; если здесь допустить ошибку, то отмена и повтор операций будут работать неправильно. Также хотелось бы увидеть улучшения в поддержке баз данных. В настоящее время ни один виджет Qt не понимает, что такое `NULL`, и создается впечатление, что поддержка баз данных была не очень ловко утрамбована в жесткие рамки архитектуры модель/представление. Кроме того, поведение баз данных, особенно в части SQLite, похоже, мало изменится при выпуске очередной промежуточной версии. Будем надеяться, что в будущих версиях Qt нас ждут усовершенствования в этой области. Но есть и радостные известия. Проект Item Views NG (Next Generation), похоже, устойчиво набирает обороты и уверенно претендует на то, чтобы стать мощнее, гибче и в то же время проще в использовании, чем нынешняя архитектура модель/представление. А к тому времени, как вы будете читать эти строки, он, возможно, будет готов к встрече с публикой.

Меньше шансов на осуществление моего личного пожелания увидеть в будущих версиях Qt добавление PDF API, который поддерживал бы чтение, редактирование и запись *pdf*-файлов с учетом всех без исключения возможностей формата PDF. Хорошо бы, если бы Qt предоставила аналогичный API также для чтения и редактирования файлов в формате Open Document Format, а заодно уж улучшила бы поддержку вывода файлов в этом формате. Полезной была бы поддержка чтения и записи наиболее распространенных архивных форматов, особенно *tar*-файлов (в том числе сжатых *gzip* или *bzip2*) и *zip*-файлов (для которых в Qt уже есть внутренний API, по крайней мере для записи). Также было бы неплохо, если бы Qt расширила поддержку более крупного программного окружения – например, включила высокоуровневый API для поддержки клиент-серверного программирования. И, конечно, были бы крайне желательны дополнительные виджеты, особенно для представления двумерных и трехмерных графиков; это сделало бы Qt более удобной библиотекой для научных и инженерных приложений. В версии Qt 5 было бы хорошо отказаться от компилятора метаобъектов (Meta-Object Compiler). Библиотеки Boost уже доказали, что механизм сигналов и слотов, равно как и систему свойств, можно написать на стандартном C++, но вопрос о том, возможно ли реализовать всю объектную модель Qt (или, по крайней мере, достаточную ее часть), пока остается открытым.

Разумеется, больше нет необходимости дожидаться, пока разработчики Qt добавят все необходимые нам средства. Теперь разработка Qt ведется более открыто, чем раньше, так что любую нужную вам функциональность вы можете реализовать самостоятельно, а потом попытаться

ся объединить ее с официальной версией Qt; подробности см. на сайте qt.gitorious.org.

Qt – превосходный каркас для разработки ПО, в улучшение и расширение которого компания Nokia вложила огромные средства. Qt можно использовать не только для программирования графических приложений, но также для разработки серверов, серверной части веб-сайтов и командных утилит. Ну и, конечно, для создания графических программ со сложным, эстетически привлекательным и в высшей степени динамичным интерфейсом. Qt применяется также в программах управления устройствами в самом широком диапазоне – от тостеров, мобильных телефонов и КПК до настольных компьютеров и далее до бесконечности.

Гигантский объем библиотеки Qt может поначалу отпугнуть, но один раз освоив фундаментальные идеи, вы уже без особого труда сможете изучать те классы и модули, которые вам нужны, и все благодаря согласованному API Qt. Великолепная документация, изобилие примеров и демонстрационных программ, наличие книг, подобных этой, вышеупомянутые онлайн-ресурсы, учебные курсы и, конечно, сам исходный код Qt – этого должно быть достаточно для любого желающего научиться. И, в отличие от некоторых библиотек, привязанных к одной платформе, Qt превращает программирование в удовольствие, позволяя работать на той платформе, к которой мы привыкли, и развертывать приложение там, где оно нужно пользователям.

Список литературы

The Art of Multiprocessor Programming

Maurice Herlihy and Nir Shavit
(Morgan Kaufmann, 2008, ISBN 9780123705914)

Подробное введение в многопоточное программирование, включающее небольшие, но полностью работоспособные примеры (на языке Java), на которых демонстрируются все основные приемы. Последняя глава содержит краткое введение в транзакционную память, одну из технологий, на которую возлагают большие надежды адепты высокоуровневого подхода к многопоточной обработке, снимающего с программиста бремя заботы о низкоуровневых деталях – неотъемлемую часть традиционных решений.

C++ GUI Programming with Qt 4, Second Edition

Jasmin Blanchette and Mark Summerfield
(Prentice Hall, 2008, ISBN 0132354160)¹

Идеальное введение в программирование на C++/Qt и прекрасное дополнение к документации по Qt. Книга научит программистов на C++ наиболее эффективно пользоваться классами и модулями Qt для создания полноценного приложения. Это официальный учебник по Qt.

C++ in a Nutshell

Ray Lischner (O'Reilly, 2003, ISBN 059600298X)²

Очень полезная книга – основательный, компактный и исчерпывающий справочник по языку C++ и его стандартным библиотекам (включая библиотеки языка C, которые не являются частью стандарта C++).

The C++ Programming Language, Third Edition

Bjarne Stroustrup (Addison-Wesley, 2000, ISBN 0201889544)³

Стандартный учебник по языку C++, написанный его автором. Может служить также полезным справочником.

¹ Жасмин Бланшет, Марк Саммерфилд «Qt 4. Программирование GUI на C++», КУДИЦ-Пресс, 2007.

² Рэй Лишнер «C++. Справочник», Питер, 2005.

³ Бьярн Страуструп «Язык программирования C++. Специальное издание», Бином, Невский диалект, 2008.

Clean Code

Robert C. Martin (Prentice Hall, 2009, ISBN 0132350882)¹

В этой книге рассматриваются многие «тактические» проблемы программирования: хороший выбор имен, проектирование функций, рефакторинг и пр. Имеется много интересных и полезных идей, которые помогут любому программисту улучшить стиль кодирования и сделать свои программы более удобными для сопровождения. (Примеры в книге написаны на языке Java.)

Code Complete: A Practical Handbook of Software Construction, Second Edition

Steve McConnell (Microsoft Press, 2004, ISBN 0735619670)²

В книге показано, как создавать надежное ПО. Изложение не ограничивается каким-то одним языком и затрагивает различные идеи, принципы и приемы. В книге множество глубоких мыслей, которые заставят любого программиста задуматься о том, что такое программирование.

Design Patterns

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides³

(Addison-Wesley, 1998, ISBN 0201633612)

Одна из книг, оказавших наиболее сильное влияние на современное программирование, хотя читать ее не всегда легко. Паттерны проектирования способны пленить любого и применяются в программировании чрезвычайно широко.

Domain-Driven Design

Eric Evans (Addison-Wesley, 2004, ISBN 0321125215)⁴

Весьма интересная книга о проектировании программного обеспечения, особенно полезная тем, кто разрабатывает крупные проекты с большим числом участников. Основная тема – разработка и уточнение моделей предметной области, которые описывают, что должна делать проектируемая система, а также создание единого языка, на котором все участники проекта – не только программисты – могли бы обмениваться мыслями.

¹ Роберт Мартин «Чистый код. Создание, анализ и рефакторинг», Питер, 2010.

² С. Макконелл «Совершенный код», Питер, Русская редакция, 2007.

³ Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влissидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования», Питер, 2001.

⁴ Эрик Эванс «Предметно-ориентированное проектирование (DDD)», Вильямс, 2010.

Effective C++, Third Edition

Scott Meyers (Addison-Wesley, 2005, ISBN 0321334876)¹

Книга, которая должна стать настольной для любого программиста на C++. В этой книге описываются многие подводные камни и дается немало полезных рекомендаций.

GUI Bloopers 2.0

Jeff Johnson (Morgan Kaufmann, 2008, ISBN 9780123706430)

Пусть вас не вводит в заблуждение шуточное название; это серьезная книга, которую должен прочитать каждый разработчик графических пользовательских интерфейсов. Наверное, вы согласитесь не со всеми предложениями, но, проработав книгу, станете более вдумчиво и с большим пониманием относиться к проектированию пользовательского интерфейса.

JavaScript: The Definitive Guide, Fifth Edition

David Flanagan (O'Reilly, 2006, ISBN 9780596101992)²

Идеальный учебник и справочное пособие по языку JavaScript/ECMAScript; эту книгу документация по Qt заслуженно рекомендует как источник для изучения QtScript. Она будет полезна также всем, кто собирается изучить язык Qt Quick QML, который войдет в версию Qt 4.7.

The Little Manual of API Design

Jasmin Blanchette (Trolltech/Nokia, 2009)

Очень короткий документ (www4.in.tum.de/~blanchet/api-design.pdf), проливающий свет на идеи и проектирование API, большая часть примеров взята из библиотеки Qt.

Mastering Regular Expressions, Third Edition

Jeffrey E. F. Friedl (O'Reilly, 2006, ISBN 0596528124)³

Стандартный учебник по регулярным выражениям – очень интересная и полезная книга.

Rapid GUI Programming with Python and Qt

Mark Summerfield (Prentice Hall, 2007, ISBN 0132354187)

Этот учебник по программированию на PyQt4 – пожалуй, самый легкий путь освоить программирование с использованием Qt вообще. На PyQt можно создавать как полноценные приложения, так и прототипы для будущих программ на C++/Qt.

¹ Скотт Мейерс «Эффективное использование C++», ДМК Пресс, 2006.

² Д. Флэнаган «JavaScript. Подробное руководство», 5-е издание, Символ-Плюс, 2008.

³ Дж. Фридл «Регулярные выражения», 3-е издание, Символ-Плюс, 2008.

Алфавитный указатель

А

`accelerateActions()` (AQP), метод, 41
`accelerateMenu()` (AQP), метод, 28, 153, 376
`accelerateWidget()` (AQP), метод, 31, 112
`accelerateWidget()` (AQP), функция, 318
`accelerateWidgets()` (AQP), метод, 31
`activeThreadCount()` (QThreadPool), метод, 282
`addActions()`, метод
 QWidget, класс, 376
`addAction()`, метод
 QActionGroup, класс, 27
 QMenu, класс, 465
`addButton()`, метод
 QMessageBox, класс, 119, 158
`addDockWidget()`, метод
 QMainWindow, класс, 448
`addItem()`, метод
 QComboBox, класс, 51, 476
 QGraphicsScene, класс, 430
`addToJavaScriptWindowObject()` (QWebFrame), метод, 56
`addTransition()` (QState), метод, 515, 520
`addWidget()` (QGraphicsScene), метод, 429
`adjusted()` (QRect/QRectF), метод, 214, 245
`adjusted()` (QRect), метод, 266
`advance()` (QGraphicsItem), метод, 436
<algorithm>, заголовочный файл, 466
`alternateBase()` (QPalette), метод, 366
Alt, клавиша, 30
`anchor()` (QTextCursor), метод, 356, 363
`appendPlainText()` (QPlainTextEdit), метод, 281
`appendRow()` (QStandardItemModel), метод, 117, 125
`appendRow()` (QStandardItem), метод, 338
`append()`, метод
 QList, класс, 311

`applicationName()` (QApplication), метод, 72
AQP, пространство имен
 `accelerateMenu()`, метод, 27, 30, 45
 `accelerateMenu()`, функция, 318
 `accelerateWidgets()`, функция, 428
 `applicationPathOf()`, метод, 49, 73
 `chunkSizes()`, метод, 278
 `chunkSizes()`, функция, 336
 Error, класс, 122, 165
 `filenameFilter()`, метод, 74
 `information()`, метод, 77, 307
 `okToClearData()`, метод, 77, 157
 `okToDelete()`, метод, 77, 161
 `question()`, метод, 312
 `radiansFromDegrees()`, метод, 215
 `sufficesForMimeTypes()`, метод, 88
 `warning()`, метод, 77, 405
`arg()` (QString), метод, 91, 321
`assignProperty()` (QState), метод, 514, 520, 523
`atBlockEnd()` (QTextCursor), метод, 356
`atBlockStart()` (QTextCursor), метод, 356
`atEnd()` (QTextCursor), метод, 356
`at()` (QList), метод, 204, 466
`at()` (QString), метод, 355
`atStart()` (QTextCursor), метод, 356

В

`baseName()` (QFileInfo), метод, 101
`beginEditBlock()` (QTextCursor), метод, 356
`beginInsertRows()` (QAbstractItemModel), метод, 138, 146, 198
`begin()` (QTextDocument), метод, 416
`beginRemoveRows()` (QAbstractItemModel), метод, 138, 189
`blockCharFormat()` (QTextCursor), метод, 356
`blockFormat()` (QTextCursor), метод, 356, 387
`block()` (QTextCursor), метод, 356
bmp, формат, 394, 407

bool volatile, спецификатор, 272
 Boost, библиотека, 79
 bottom() (QRect/QRectF), метод, 238
 boundingRect() (QGraphicsItem), метод, 433, 436, 467, 502
 Browser Window, пример приложения, 38

C

ceil() (std), функция, 470
 center() (QRect/QRectF), метод, 240, 434
 charFormat() (QTextCursor), метод, 356
 childItems() (QGraphicsItem), метод, 436
 child(), метод
 QStandardItem, класс, 166
 QTreeWidgetItem, класс, 96
 clearSelection() (QAbstractItemView), метод, 301
 clearSelection() (QGraphicsScene), метод, 454, 455
 clearSelection() (QTextCursor), метод, 356, 364
 clear(), метод
 QComboBox, класс, 51
 QGraphicsScene, класс, 451
 QList, класс, 319
 QStandardItemModel, класс, 122, 165
 QString, класс, 129
 clickedButton() (QMessageBox), метод, 120
 clipboard() (QApplication), метод, 462, 473
 clone() (QStandardItem), метод, 210
 close() (QFile), метод, 405
 collidesWithItem() (QGraphicsItem), метод, 436
 collidesWithPath() (QGraphicsItem), метод, 437
 collidingItems() (QGraphicsItem), метод, 437
 colorNames() (QColor), метод, 477
 columnCount(), метод
 QAbstractItemModel, класс, 138, 184
 QStandardItemModel, класс, 299
 columnNumber() (QTextCursor), метод, 356
 column() (QModelIndex), метод, 140, 226
 compare() (QString), метод, 361
 complete(), метод
 QCompleter, класс, 359
 completionPrefix() (QCompleter), метод, 361

connect() (QObject), метод, 43, 92, 113, 375, 385, 520
 constBegin() (QHash), метод, 333
 contains(), метод
 QGraphicsItem, класс, 437
 createEditor() (QStyledItemDelegate), метод, 216, 222
 createIndex(), метод
 QAbstractItemModel, класс, 185, 186, 196, 200
 createList() (QTextCursor), метод, 356, 398
 createPlugin() (QWebPage), метод, 63
 CSS (Cascading Style Sheet), каскадные таблицы стилей, 346
 <cstdio>, модуль, 307
 <cstdlib>, заголовочный файл, 62
 Ctrl, клавиша, 30, 241, 375
 currentBlockState() (QSyntaxHighlighter), метод, 371
 currentCompletion() (QCompleter), метод, 355
 currentFrame() (QTextCursor), метод, 356
 currentIndex() (QAbstractItemView), метод, 157, 159, 161, 240, 244
 currentIndex() (QComboBox), метод, 53, 481
 currentItem() (QTreeWidgetItem), метод, 93
 currentList() (QTextCursor), метод, 357
 currentTable() (QTextCursor), метод, 357
 currentText() (QComboBox), метод, 115
 cursorRect() (QPlainTextEdit), метод, 359

D

dataChanged() (QAbstractItemModel), метод, 145
 dataChanged() (QAbstractItemView), метод, 231, 239
 deleteChar() (QTextCursor), метод, 357
 deleteLater() (QObject), метод, 318, 321
 deletePreviousChar() (QTextCursor), метод, 357
 desktop() (QApplication), метод, 409
 DirectShow, библиотека, 77
 DirectX, библиотека, 77
 disconnect() (QObject), метод, 72
 documentLayout() (QTextDocument), метод, 416
 document() (QPlainTextEdit), метод, 353
 document() (QTextCursor), метод, 357

DOM (Document Object Model),
объектная модель, 33
drawBackground(), метод
 QGraphicsScene, класс, 470
drawChord() (QPainter), метод, 501
drawEllipse() (QPainter), метод, 500
dropMimeData() (QAbstractItemModel),
метод, 180
dynamic_cast<>(), метод, 255, 379
dynamic_cast<>(), оператор, 458, 474
 как избежать, 458

E

edit() (QAbstractItemView), метод, 118,
160, 174
emit, метод, 262, 326, 328, 379, 493, 521
endEditBlock() (QTextCursor), метод, 357
endInsertColumns()
 (QAbstractItemModel), метод, 138
endInsertRows() (QAbstractItemModel),
метод, 138, 145, 188, 193
endMoveColumns()
 (QAbstractItemModel), метод, 180
endRemoveColumns()
 (QAbstractItemModel), метод, 138
endRemoveRows() (QAbstractItemModel),
метод, 138, 146, 189, 197
ensureVisible(), метод
 QGraphicsItem, класс, 437
Enter, клавиша, 221, 222
Error (AQP), класс, 147
errorString() (QFile), метод, 165, 405
escape() (Qt), метод, 396
eventFilter() (QObject), метод, 100, 259
event() (QWidget), метод, 281
exec(), метод
 QApplication, класс, 23
 QDialog, класс, 78
 QDialog, проблемы, 121
exists() (QFile), метод, 49, 93
expand(), метод
 QAbstractItemView, класс, 212

F

F2, клавиша, 174
filenameFilter() (AQP), метод, 102
fileName() (QFileInfo), метод, 339
filtered() (QtConcurrent), метод, 297
filteredReduced() (QtConcurrent), метод,
303, 305
find() (QPixmapCache), метод, 220

flags() (QAbstractItemModel), метод, 137,
140, 182
fontStrikeOut() (QTextCharFormat),
метод, 380
font(), метод
 QApplication, класс, 234
 QWidget, класс, 240
foreach, метод, 113, 125, 336, 377, 477
forever, метод, 338
fromHtml() (QTextDocumentFragment),
метод, 53
fromUtf8() (QString), метод, 57, 124
future() (QFutureWatcher), метод, 301,
311

G

globalInstance() (QThreadPool), метод,
283
grabWidget() (QPixmap), метод, 220
GraphicsItemFlag (QGraphicsItem),
перечисление, 441
Gstreamer, библиотека, 77

H

hasComplexSelection() (QTextCursor),
метод, 357
hasFormat() (QMimeData), метод, 473
hasSelection() (QTextCursor), метод, 357
hasText() (QMimeData), метод, 473
headerData() (QAbstractItemModel),
метод, 138
hide(), метод
 QGraphicsItem, класс, 437
 QWidget, класс, 307, 334, 339
highlightBlock() (QSyntaxHighlighter),
метод, 366, 368
highlightedText() (QPalette), метод, 265
horizontalOffset() (QAbstractItemView),
метод, 231, 241
hoursMinutesSecondsForMSec() (AQP),
метод, 91
HTML, 394
 преобразование в обычный текст, 33,
53
 ширина текста, 220
 экранирование, 383
 экспорт, 405
html() (QMimeData), метод, 464

I

idealWidth() (QTextDocument), метод,
220, 380

`indexAt()` (`QAbstractItemView`), метод, 231, 238
`index()`, метод
 `QAbstractItemModel`, класс, 130, 139, 174, 185, 235
 `QModelIndex`, класс, 212
 `QStandardItemModel`, класс, 305
 `QStandardItem`, класс, 160
`information()` (`AQP`), метод, 77, 307
`insert()`, метод
 `QList`, класс, 145, 179
 `QMultiHash`, класс, 332, 369
 `QPixmapCache`, класс, 220, 478
`insertBlock()` (`QTextCursor`), метод, 357, 402
`insertColumns()` (`QAbstractItemModel`), метод, 138, 144
`insertFragment()` (`QTextCursor`), метод, 357, 400
`insertFrame()` (`QTextCursor`), метод, 357
`insertHtml()` (`QTextCursor`), метод, 357, 399
`insertImage()` (`QTextCursor`), метод, 357, 402
`insertList()` (`QTextCursor`), метод, 358, 398
`insertRow()` (`QAbstractItemModel`), метод, 135
`insertRows()` (`QAbstractItemModel`), метод, 135, 138, 144, 146, 188
`insertTable()` (`QTextCursor`), метод, 358, 400
`insertText()` (`QTextCursor`), метод, 358, 399, 401
`intersects()` (`QRect/QRectF`), метод, 243
`invalidateFilter()` (`QSortFilterProxyModel`), метод, 128
`invisibleRootItem()` (`QStandardItemModel`), метод, 166, 339
`invokeMethod()` (`QMetaObject`), метод, 154, 284, 457
`isCanceled()` (`QFutureWatcher`), метод, 301, 311
`isCopyOf()` (`QTextCursor`), метод, 358
`isFile()` (`QFileInfo`), метод, 86, 341
`isIndexHidden()` (`QAbstractItemView`), метод, 231
`isNull()` (`QTextCursor`), метод, 358
`isNull()`, метод
 `QIcon`, класс, 33
 `QPixmap`, класс, 75

ISO 8601, формат даты и времени, 164, 211
`isObscuredBy()` (`QGraphicsItem`), метод, 437
`isRunning()`, метод
 `QThread`, класс, 337
`isSelected()` (`QGraphicsItem`), метод, 437
`isVisible()` (`QGraphicsItem`), метод, 437

J

JavaScript, язык
 и объекты `window`, 56
`joinPreviousEditBlock()` (`QTextCursor`), метод, 358

K

KDE (K Desktop Environment), среда, 36, 76
`keyboardModifiers()` (`QApplication`), метод, 508
`keyPressEvent()` (`QGraphicsItem`), метод, 437
`keyPressEvent()` (`QWidget`), метод, 262, 363, 379, 492

L

`layoutDirection()`, метод
 `QApplication`, класс, 246
 `QWidget`, класс, 246
`layout()` (`QWidget`), метод, 322
`loadFromData()` (`QPixmap`), метод, 35
`load()` (`QWebView`), метод, 43
`localeAwareCompare()` (`QString`), метод, 361
`long double`, преобразование в строку, 307

M

Mac OS X
 листки, 77
 платформенные особенности, 42
`mainFrame()` (`QWebPage`), метод, 55, 56
`mapped()` (`QtConcurrent`), метод, 308
`mapToGlobal()` (`QWidget`), метод, 380
`mapToScene()` (`QGraphicsView`), метод, 461
MD5 (алгоритм Message-Digest версии 5), 329
`mergeBlockCharFormat()` (`QTextCursor`), метод, 358
`mergeBlockFormat()` (`QTextCursor`), метод, 358

`mergeCharFormat()` (`QTextCursor`), метод, 358, 399
`mimeData()` (`QAbstractItemModel`), метод, 180, 190
`mimeTypes()` (`QAbstractItemModel`), метод, 180
`ModelTest`, класс, 133, 156
`mouseButtons()` (`QApplication`), метод, 508
`mouseDoubleClickEvent()` (`QGraphicsItem`), метод, 437
`mouseMoveEvent()` (`QGraphicsItem`), метод, 437
`mousePressEvent()`, метод
 `QAbstractItemView`, класс, 231
 `QGraphicsItem`, класс, 437
`mouseReleaseEvent()` (`QGraphicsItem`), метод, 494
`moveCursor()` (`QAbstractItemView`), метод, 231
`MoveOperation` (`QTextCursor`), перечисление, 362
`movePosition()` (`QTextCursor`), метод, 358

N

`name()` (`QColor`), метод, 478
`numeric_limits<>::max()` (`std`), метод, 209
`numeric_limits<>::min()` (`std`), функция, 471

O

`okToClearData()` (`AQP`), метод, 77
`okToDelete()` (`AQP`), метод, 77
`opaqueArea()` (`QGraphicsItem`), метод, 438
`open()`, метод
 `QDialog`, класс, 121
 `QFile`, класс, 147, 341
`operator>>()` (`QDataStream`), метод, 148, 453, 487
`operator+=()` (`QPoint/QPointF`), метод, 461
`operator()`, метод, 297, 300, 310
`operator[]()`, метод
 `QHash`, класс, 235, 429, 430
 `QMap`, класс, 322
`operator*()`, метод, 466
`operator<()`, метод, 360
 `QTreeWidgetItem`, класс, 87
 для элементов `QList`, 136

`operator<<()`, метод
 `QDataStream`, класс, 147, 451
 `QList`, класс, 225, 336, 376, 428, 477, 519
 `QSet`, класс, 127, 293
 `QVector`, класс, 401
`operator==()`, метод
 для элементов `QList`, 136

P

`paintEvent()`, метод
 `QAbstractScrollArea`, класс, 244
 `QWidget`, класс, 256, 263
`paint()` (`QGraphicsItem`), метод, 438
`palette()` (`QWidget`), метод, 256, 366
`parentItem()` (`QGraphicsItem`), метод, 438
`parent()`, метод
 `QAbstractItemModel`, класс, 139
 `QModelIndex`, класс, 119
 `QObject`, класс, 259
`popup()` (`QCompleter`), метод, 355, 363
`pos()` (`QGraphicsItem`), метод, 438
`pos()`, метод
 `QCursor`, класс, 461, 497
`prepareGeometryChange()` (`QGraphicsItem`), метод, 438, 497
`previousBlockState()` (`QSyntaxHighlighter`), метод, 368, 371
`print()` (`QTextDocument`), метод, 404
`progressMinimum()` (`QFutureWatcher`), метод, 299

Q

`QAbstractButton`, класс, 120, 276
 `setChecked()`, метод, 219
`QAbstractItemModel`, класс, 179, 182
 `beginInsertColumns()`, метод, 138
 `beginInsertRows()`, метод, 138, 188, 193
 `beginMoveColumns()`, метод, 180
 `beginMoveRows()`, метод, 180
 `beginRemoveColumns()`, метод, 138
 `beginRemoveRows()`, метод, 138, 146, 189, 197
 `columnCount()`, метод, 144, 184
 `createIndex()`, метод, 185, 186, 196, 197, 198, 200
 `dataChanged()`, метод, 138, 145, 187, 196, 200, 202
 `data()`, метод, 127, 130, 137, 183, 219, 350

- dropMimeData(), метод, 180, 193
- endInsertRows(), метод, 138, 146, 188, 193, 198
- endMoveRows(), метод, 180
- endRemoveColumns(), метод, 138
- endRemoveRows(), метод, 138, 146, 189, 197
- flags(), метод, 137, 140, 182
- headerDataChanged(), метод, 138, 145
- headerData(), метод, 141, 184
- index(), метод, 130, 139, 185
- insertColumns(), метод, 135
- insertRow(), метод, 135, 146, 188
- insertRows(), метод, 135, 146
- parent(), метод, 186
- removeColumns(), метод, 135, 138, 144
- removeRow(), метод, 146, 161
- removeRows(), метод, 135, 138, 144, 145, 188
- reset(), метод, 202
- rowCount(), метод, 135, 138, 175, 184
- setData(), метод, 138, 187
- setHeaderData(), метод, 138, 145, 187
- sort(), метод, 477
- supportedDragActions(), метод, 180, 190
- supportedDropActions(), метод, 180
- иерархия классов, 108
- QAbstractItemView::selectionModel(), метод, 114
- QAbstractItemView, класс, 230, 231
 - currentIndex(), метод, 240, 244
 - dataChanged(), метод, 231, 239
 - edit(), метод, 160, 174
 - expand(), метод, 212
 - horizontalOffset(), метод, 241
 - isIndexHidden(), метод, 237
 - resizeEvent(), метод, 232, 245
 - rootIndex(), метод, 243
 - rowsAboutToBeRemoved(), метод, 232
 - rowsInserted(), метод, 232, 239
 - scrollContentsBy(), метод, 232, 242
 - scrollDirtyRegion(), метод, 242
 - scrollTo(), метод, 155, 232
 - selectionModel(), метод, 243
 - setCurrentIndex(), метод, 155
 - setEditTriggers(), метод, 297, 301
 - setModel(), метод, 232, 234
 - setSelection(), метод, 232, 242
 - updateGeometries(), метод, 232, 245
 - verticalOffset(), метод, 232, 242
 - viewOptions(), метод, 244
 - visualRect(), метод, 232, 235, 244
 - visualRegionForSelection(), метод, 232, 243
- QAbstractListModel, класс, 144
- QAbstractScrollArea, класс, 244
 - paintEvent(), метод, 244
 - setVerticalScrollBarPolicy(), метод, 375
 - verticalScrollBar(), метод, 233, 253, 339
 - viewport(), метод, 221, 233, 235, 238, 242
- QAbstractSocket, класс, 24
- QAbstractTableModel, класс, 135, 137
- qAbs(), глобальная функция, 62
- QActionGroup, класс
 - addAction(), метод, 28
- QAction, класс, 46, 72
 - data(), метод, 379, 460
 - setChecked(), метод, 387
 - setData(), метод, 376, 498
 - setIcon(), метод, 466
 - setMenu(), метод, 465
 - setText(), метод, 75
- qAlpha(), функция, 327
- QApplication, класс, 23
 - applicationDirPath(), метод, 49
 - clipboard(), метод, 473
 - mouseButtons(), метод, 508
 - postEvent(), метод, 280
 - restoreOverrideCursor(), метод, 87
 - sendEvent(), метод, 280
 - setApplicationName(), метод, 446
 - setQuitOnLastWindowClosed(), метод, 23
 - style(), метод, 142
- Q_ARG, макрос, 284
- Q_ASSERT, утверждение, 93, 142, 183, 185, 222, 226
- qBlue(), функция, 327
- qBound(), глобальная функция, 62, 526
- QBrush, класс, 476, 480
- QByteArray, класс, 29, 57, 74, 193, 334, 342
 - readAll(), метод, 56
- QCache, класс
 - insert(), метод, 34
 - object(), метод, 32
- QChar, класс, 252
- QCheckBox, класс, 27, 518

- QClipboard, класс
 - mimeData(), метод, 464, 473
 - setMimeData(), метод, 462
- QCloseEvent, класс, 340
- QColorDialog, класс, 386
- QColor, класс, 378, 470, 506
 - colorNames(), метод, 477
 - lighter(), метод, 502
 - name(), метод, 478
- QColumnView, класс, 106, 207
- QComboBox, класс, 106, 207, 351
 - currentIndex(), метод, 53
 - currentText(), метод, 115, 226
 - itemData(), метод, 52
 - setCurrentIndex(), метод, 119, 226, 480, 519
 - setModel(), метод, 112
- QCompleter, класс, 349, 361
 - complete(), метод, 359
 - completionPrefix(), метод, 361
 - currentCompletion(), метод, 355
 - setCompletionPrefix(), метод, 355
 - setModelSorting(), метод, 355
 - setWrapAround(), метод, 353
- QConicalGradient, класс, 266
- QCryptographicHash, класс, 342
- QDataStream, класс, 123, 462
 - operator<<(), метод, 450
 - operator>>(), метод, 148, 453, 487
 - setVersion(), метод, 122, 124
- QDateTime, класс, 163, 178
 - fromString(), метод, 194
 - toString(), метод, 166, 215
- qDebug(), глобальная функция, 62
- qDeleteAll(), метод, 178
- QDesktopServices, класс
 - openUrl(), метод, 324
- QDesktopWidget, класс
 - screenCount(), метод, 409
 - screenGeometry(), метод, 409
- QDialog, класс
 - open(), метод, 121
 - show(), метод, 121
- QDirIterator, класс, 86, 276, 341
- QDirModel, класс, 350
- QDir, класс
 - currentPath(), метод, 212
 - homePath(), метод, 212
 - home(), метод, 212
 - rootPath(), метод, 212
 - separator(), метод, 280, 350
 - tempPath(), метод, 212
 - toNativeSeparators(), метод, 339
- QDockWidget, класс, 448
- QDomDocument, класс, 31
- QDoubleSpinBox, класс
 - setRange(), метод, 209
- QEasingCurve, класс, 506
- QEventLoop, класс
 - exec(), метод, 92, 101
- QEvent, класс, 100, 280
- QFile, класс
 - exists(), метод, 49, 93
 - open(), метод, 122
 - readAll(), метод, 341
 - size(), метод, 339, 341
 - suffix(), метод, 86, 276
- QFileDialog, класс
 - getExistingDirectory(), метод, 85
 - getOpenFileName(), метод, 74, 101
 - getSaveFileName(), метод, 76
- QFileInfo
 - absolutePath(), метод, 74
- QFileInfo, класс
 - baseName(), метод, 102
 - isDir(), метод, 336
- QFileSystemWatcher, класс, 212
- QFinalState, класс, 511, 513, 515
- QFont, класс, 526
- QFontMetrics, класс
 - elidedText(), метод, 381
 - width(), метод, 235, 250
- QFormLayout, класс, 479
- QFrame, класс, 40
 - setFrameStyle(), метод, 41
- QFuture, класс, 291, 295, 309
- QFutureWatcher, класс, 291, 295
 - future(), метод, 301, 311
 - isCanceled(), метод, 301
 - progressMaximum(), метод, 299
 - result(), метод, 307
 - setFuture(), метод, 297, 304
- qFuzzyCompare(), глобальная функция, 62
- qFuzzyCompare(), функция, 300, 489
- QGraphicsAnchorLayout, класс, 430
- QGraphicsBlurEffect, класс, 519
- QGraphicsColorizeEffect, класс, 519
- QGraphicsDropShadowEffect, класс, 519
- QGraphicsEffect, класс, 439
- QGraphicsGridLayout, класс, 430
- qgraphicsitem_cast<>(), функция, 434, 458, 484
- QGraphicsItemGroup, класс, 424, 437, 451, 470

- QGraphicsItem, класс, 422, 427, 433, 450, 471
 - boundingRect(), метод, 433, 436, 467, 502
 - collidesWithItem(), метод, 436
 - collidesWithPath(), метод, 437
 - collidingItems(), метод, 437
 - contextMenuEvent(), метод, 495
 - GraphicsItemFlag, перечисление, 441
 - group(), метод, 437, 450
 - isSelected(), метод, 489, 492, 500
 - itemChange(), метод, 426, 442, 485, 486
 - mouseDoubleClickEvent(), метод, 484
 - mouseMoveEvent(), метод, 494
 - moveBy(), метод, 438, 452, 467
 - paint(), метод, 433, 498
 - pos(), метод, 487
 - resetTransform(), метод, 438
 - rotation(), метод, 438
 - scale(), метод, 438
 - sceneBoundingRect(), метод, 467
 - scenePos(), метод, 438
 - scene(), метод, 438
 - setCacheMode(), метод, 499
 - setCursor(), метод, 494
 - setFlags(), метод, 438
 - setFlag(), метод, 424, 438
 - setGraphicsEffect(), метод, 439
 - setGroup(), метод, 439
 - setParentItem(), метод, 439
 - setPen(), метод, 470
 - setPos(), метод, 439, 488
 - setRotation(), метод, 427, 439, 486
 - setScale(), метод, 427, 439, 486
 - setSelected(), метод, 439, 454
 - setToolTip(), метод, 439
 - setTransformations(), метод, 439
 - setTransformOriginPoint(), метод, 427, 486
 - setTransform(), метод, 426, 439, 486, 490
 - setX(), метод, 439
 - setY(), метод, 439
 - setZValue(), метод, 439
 - shape(), метод, 433, 440, 502
 - show(), метод, 440
 - toGraphicsObject(), метод, 458
 - toolTip(), метод, 440
 - transform(), метод, 440
 - type(), метод, 433, 440, 450, 459, 484, 492
 - update(), метод, 440, 496
 - x(), метод, 440
 - y(), метод, 440
 - z(), метод, 440
 - иерархия классов, 424
 - сравнение с QGraphicsObject, 424
- QGraphicsLayoutItem, класс, 430
- QGraphicsLinearLayout, класс, 430
- QGraphicsLineItem, класс, 470
- QGraphicsObject, класс, 458, 482, 495
- QGraphicsOpacityEffect, класс, 519, 524
- QGraphicsPathItem, класс, 422
- QGraphicsProxyWidget, класс, 423, 426, 429, 482
- QGraphicsRectItem, класс, 491
- QGraphicsScene, класс, 422, 428, 503
 - addWidget(), метод, 429
 - clearSelection(), метод, 485
 - clear(), метод, 451
 - drawBackground(), метод, 469
 - items(), метод, 450, 473
 - removeItem(), метод, 463
 - selectedItems(), метод, 462, 490
 - width(), метод, 470
 - компоновщики, 429
- QGraphicsSceneMouseEvent, класс, 493
- QGraphicsSimpleTextItem, класс, 422
- QGraphicsTextItem, класс, 422, 482
- QGraphicsView, класс, 422, 426, 474, 503
 - drawBackground(), метод, 451
 - mapToScene(), метод, 461
- QGraphicsWebView, класс, 423, 482
- QGraphicsWidget, класс, 423, 426, 430, 432, 482
- qGray(), функция, 327
- qGreen(), функция, 327
- QGridLayout, класс, 322
- QHash, класс, 86, 233, 429
 - operator[](), метод, 235, 430
 - values(), метод, 332, 333
 - value(), метод, 236
 - сравнение с QMultiHash, 331
- qHash(), метод, 26
- QImage, класс, 317
 - bits(), метод, 328
 - pixel(), метод, 327
 - save(), метод, 279, 320, 407, 419, 457
 - scanLine(), метод, 327
 - setPixel(), метод, 327
 - size(), метод, 326
 - width(), метод, 328
- QImageReader, класс, 407
- QImageWriter, класс, 407, 419

- QInputDialog, класс
 - getText(), метод, 43
- QIODevice, класс, 29, 80
- QItemDelegate, класс, 207
- QItemEditorFactory, класс, 208
- QItemSelectionModel, класс, 114, 301
- QItemSelectionRange, класс, 243
- QItemSelection, класс, 116
- QKeyEvent, класс, 262, 363, 379, 493
- QKeySequence, класс, 41, 493
- QLabel, класс, 322
 - setBuddy(), метод, 519
 - setText(), метод, 219
 - setWordWrap(), метод, 217
- QLinearGradient, класс, 266
 - highlight(), метод, 257
- QLine/QLineF, класс, 506
- QLinkedList, класс, 136
- QList, класс, 74
 - at(), метод, 141, 467, 507
 - clear(), метод, 318, 321
 - contains(), метод, 136
 - count(), метод, 136
 - first(), метод, 336
 - fromSet(), метод, 293
 - indexOf(), метод, 136
 - insert(), метод, 145
 - isEmpty(), метод, 335, 462
 - lastIndexOf(), метод, 136
 - mid(), метод, 171, 204, 283, 312
 - operator[](), метод, 145
 - operator<(), метод, 136
 - operator<<(), метод, 225, 336, 477, 519
 - operator==(), метод, 136
 - prepend(), метод, 170, 204
 - removeAll(), метод, 136, 360
 - removeAt(), метод, 146
 - removeOne(), метод, 136
 - takeAt(), метод, 179
 - takeFirst(), метод, 382
 - value(), метод, 179
- QListIterator, класс, 45, 147, 192, 301, 490
- QListView, класс, 106, 207
- QListWidget, класс, 106
- QLocale, класс, 252, 265
- QMainWindow, класс, 83, 99, 133, 153
 - addDockWidget(), метод, 448
 - setCentralWidget(), метод, 155, 173, 431, 448
 - setDockOptions(), метод, 448
 - setTabPosition(), метод, 449
- qmake, программа, 134
- qMakePair(), функция, 342, 378, 477
- QMap, класс, 51, 54, 125
 - сравнение с QMultiMap, 331
- QMapIterator, класс, 51
- qMax(), глобальная функция, 62
- qMax(), метод, 219
- QMenu, класс
 - addAction(), метод, 380
 - addAction(), метод, 377, 465
 - addSeparator(), метод, 497
 - exec(), метод, 46, 380, 498
- QMessageBox, класс, 77
 - addButton(), метод, 119
 - setText(), метод, 160
 - setWindowModality(), метод, 78
- QMetaObject, класс, 457
 - indexOfProperty(), метод, 453, 469, 474
 - invokeMethod(), метод, 154, 284, 457
- QMimeData, класс, 180, 190
 - hasText(), метод, 473
 - setData(), метод, 462
 - text(), метод, 464
- qMin(), глобальная функция, 62
- qMin(), метод, 213, 238
- QModelIndex, класс, 127, 130, 138, 145, 159, 174, 183
 - column(), метод, 224
 - index(), метод, 212
 - isValid(), метод, 160
 - row(), метод, 119, 141
- QMouseEvent, класс, 246, 261, 365
- QMovie, класс, 72, 74
- QMultiHash, класс
 - insert(), метод, 332
 - remove(), метод, 333
 - сравнение с QHash, 331
- QMultiMap, класс, 331
- QMutexLocker, класс, 293
- QMutex, класс, 292
 - сравнение с QReadWriteLock, 294
- QNetworkAccessManager, класс, 20, 25, 26
- QNetworkProxy, класс, 24
- QNetworkReply, класс, 29, 34
- qobject_cast<>(), метод, 222, 255, 259, 379
- qobject_cast<>(), функция, 458
- QObject, класс, 55, 506
 - connect(), метод, 43, 299, 520
 - deleteLater(), метод, 321
 - disconnect(), метод, 72

- eventFilter(), метод, 100, 259
- installEventFilter(), метод, 100
- metaObject(), метод, 453, 457, 474
- sender(), метод, 222, 379, 460, 466
- setProperty(), метод, 490
- tr(), метод, 49, 161, 301
- соединение сигнала с сигналом, 251, 385
- QPageSetupDialog, класс, 408
- QPainter, класс, 406, 419
 - drawEllipse(), метод, 501
 - drawImage(), метод, 414
 - drawLine(), метод, 215, 244
 - drawPixmap(), метод, 220
 - drawText(), метод, 215
 - end(), метод, 455
 - fillRect(), метод, 213, 407, 419
 - setBrush(), метод, 214
 - setPen(), метод, 418, 435, 500
 - setRenderHints(), метод, 457
 - setViewport(), метод, 415, 419
 - viewport(), метод, 407, 412
- QPainterPathStroker, класс, 434
- QPainterPath, класс, 422, 434, 502
- QPalette, класс, 215, 224
 - alternateBase(), метод, 366
 - base(), метод, 266
- QParallelAnimationGroup, класс, 509
- QPen, класс, 245, 256, 258, 470
- QPixmapCache, класс, 219
 - find(), метод, 220
 - insert(), метод, 478
 - setCacheLimit(), метод, 478
- QPixmap, класс, 478
 - fill(), метод, 377
 - grabWidget(), метод, 220
 - isNull(), метод, 75
 - loadFromData(), метод, 35
 - save(), метод, 76
- QPlainTextEdit, класс, 351
 - appendPlainText(), метод, 281
 - document(), метод, 364
 - setTextCursor(), метод, 364
 - textCursor(), метод, 355, 361, 364
- QPoint, класс
 - rx(), метод, 238
 - ry(), метод, 238
 - setX(), метод, 239, 435
 - x(), метод, 240
- QPointF, класс, 238
- QPointer, класс, 78, 316, 320
- QPolygon/QPolygonF, класс, 496
- qPrintable(), глобальная функция, 59, 62
- QPrintDialog, класс, 455
- QPrinter, класс, 404, 446
- QPrintPreviewDialog, класс, 409
- QProcess, класс, 273
- QProgressBar, класс, 316
 - setRange(), метод, 322, 323
- Q_PROPERTY, макрос, 508, 512
- QPropertyAnimation, класс
 - setEndValue(), метод, 525
 - setStartValue(), метод, 510
 - startAt(), метод, 508
- QPushButton, класс, 428
- QRadialGradient, класс, 266
- QRadioButton, класс, 114
- qrand(), глобальная функция, 61, 62
- QReadLocker, класс, 332
- QReadWriteLock, класс, 332
 - сравнение с QMutex, 294
- QRect/QRectF, класс
 - adjusted(), метод, 214, 245, 492
 - bottom(), метод, 238
 - center(), метод, 240
 - contains(), метод, 238
 - intersects(), метод, 243
 - left(), метод, 266
 - setBottomRight(), метод, 494
 - setWidth(), метод, 214
 - top(), метод, 238
 - translated(), метод, 243
 - united(), метод, 502
 - width(), метод, 236
- qRed(), функция, 327
- QRegExpr, класс, 360
 - setPatternSyntax(), метод, 369
 - политика запоминания групп, 369
- QRegion, класс, 243
- qRegisterAnimationInterpolator<>(), функция, 525, 526
- QRgb, класс, 327
- QRgba, класс, 327
- qRound(), глобальная функция, 62
- qRound(), метод, 215, 265
- qRound(), функция, 326
- qRound64(), глобальная функция, 62
- QRunnable, класс, 282
 - run(), метод, 283
- Qscintilla, текстовый редактор, 352
- QScopedPointer, класс, 78, 120, 158, 463
- QScrollArea, класс, 254, 258, 260
 - setWidget(), метод, 253
- QScrollBar, класс
 - setPageStep(), метод, 246

- QSet, класс
 - contains(), метод, 127, 293
 - operator<<(), метод, 293, 452
- QSettings, класс, 23
- QSharedPointer, класс, 78
- QShortcut, класс, 100, 354, 375
- QSignalTransition, класс, 524
- QSize/QSizeF, класс, 220, 255, 380
 - expandedTo(), метод, 326
 - rwidth(), метод, 409
 - scale(), метод, 413
 - setWidth(), метод, 522
- QSortFilterProxyModel, класс, 126
 - filterAcceptsRow(), метод, 127
 - invalidateFilter(), метод, 129
 - setFilterKeyColumn(), метод, 130
 - setFilterRegExp(), метод, 130
 - setSourceModel(), метод, 127, 133
 - sort(), метод, 112
- qSort(), метод, 136, 148
- qSort(), функция, 360
- QSound, класс, 70, 73
- QSpinBox и QDoubleSpinBox, классы, 317
- QSpinBox, класс
 - textFromValue(), метод, 227
 - value(), метод, 226
- qrand(), глобальная функция, 61, 62
- qStableSort(), функция, 360
- QStack, класс, 167
- QStandardItemEditorCreator, класс, 208
- QStandardItemModel, класс
 - appendRow(), метод, 125
 - clear(), метод, 122, 165
 - horizontalHeaderItem(), метод, 165
 - invisibleRootItem(), метод, 162, 166, 339
 - itemChanged(), метод, 113, 156
 - item(), метод, 305
 - setColumnCount(), метод, 334
 - sort(), метод, 339
 - takeRow(), метод, 150
 - сравнение с пользовательскими моделями, 131
- QStandardItem, класс, 118, 125, 209
 - child(), метод, 162
 - clone(), метод, 210
 - data(), метод, 210
 - index(), метод, 155
 - setData(), метод, 210, 311
 - setTextAlignment(), метод, 339
 - text(), метод, 166
 - сравнение с пользовательским элементом, 177
- QState, класс, 511, 513
 - addTransition(), метод, 516, 520, 524
 - assignProperty(), метод, 514, 520, 523
- QStateMachine, класс, 511, 518
 - setInitialState(), метод, 514, 520
 - start(), метод, 511
- QStatusBar::showMessage(), метод, 72, 111, 311
- QString, класс
 - arg(), метод, 34, 119
 - clear(), метод, 130
 - compare(), метод, 87
 - endsWith(), метод, 350
 - fromUtf8(), метод, 56
 - isEmpty(), метод, 32
 - left(), метод, 252
 - localeAwareCompare(), метод, 361
 - number(), метод, 67, 435
 - prepend(), метод, 43
 - replace(), метод, 318
 - split(), метод, 360, 416
 - toLower(), метод, 88, 279
 - toUpper(), метод, 276
 - toUtf8(), метод, 124
 - trimmed(), метод, 49, 88
 - преобразование из long double, 307
- QStringList, класс, 190, 336
 - join(), метод, 88
 - removeDuplicates(), метод, 294, 360
 - sort(), метод, 477
- QStringRef, класс, 168
- QStyle, класс, 217
- QStyledItemDelegate, класс, 207
 - createEditor(), метод, 221, 225
 - paint(), метод, 216, 218, 224
 - setEditorData(), метод, 216, 221, 225
 - setModelData(), метод, 216, 222, 226
 - sizeHint(), метод, 216, 220
 - updateEditorGeometry(), метод, 216
- QStyleOptionComboBox, класс, 142
- QStyleOptionGraphicsItem, класс, 426, 435
- QStyleOptionViewItem, класс, 213, 215, 224, 244
- QSvgGenerator, класс, 406, 418, 456
- QSvgRenderer, класс, 413
- QSyntaxHighlighter, класс
 - highlightBlock(), метод, 367, 369
 - previousBlockState(), метод, 368, 371
 - setCurrentBlockState(), метод, 368
 - setFormat(), метод, 370, 371
- QSystemTrayIcon, класс, 25
 - setIcon(), метод, 32

- Qt, библиотека
 - escape(), метод, 396
 - escape(), функция, 382
 - ItemDataRole, перечисление, 139
 - ItemFlag, перечисление, 140
 - ItemSelectionMode, перечисление, 440
 - взаимодействие в JavaScript, 55
- QTableView, класс, 106, 109, 207
 - sortByColumn(), метод, 114, 130
 - verticalHeader(), метод, 112
- QTableWidgetItem, класс, 66
- QTableWidgetItem, класс, 65, 106
- QtConcurrent, пространство имен
 - filteredReduced(), метод, 303, 304
 - filtered(), метод, 296, 297
 - mapped(), метод, 308, 309
 - run(), метод, 277
- <QtDebug>, заголовочный файл, 62
- QTextBlockFormat, класс, 347, 398
- QTextBlock, класс, 356, 382, 416
- QTextCharFormat, класс, 366, 368, 383
 - fontWeight(), метод, 387
 - foreground(), метод, 382
 - intProperty(), метод, 402
 - setForeground(), метод, 368
 - setProperty(), метод, 366, 402
- QTextCursor, класс, 355
 - anchor(), метод, 363
 - API, 356
 - blockFormat(), метод, 388
 - clearSelection(), метод, 364, 366
 - createList(), метод, 398
 - insertBlock(), метод, 399, 402
 - insertFragment(), метод, 400
 - insertHtml(), метод, 399
 - insertImage(), метод, 401
 - insertList(), метод, 398
 - insertTable(), метод, 400
 - mergeBlockFormat(), метод, 399
 - movePosition(), метод, 361, 363
 - removeSelectedText(), метод, 365
 - selectedText(), метод, 355
 - selection(), метод, 348
 - setPosition(), метод, 361, 363
- QTextDocumentFragment, класс, 348, 400
 - fromHtml(), метод, 53
 - toPlainText(), метод, 33, 53
 - сравнение с QTextFragment, 348
- QTextDocumentWriter, класс, 404
- QTextDocument, класс, 53, 345, 346, 356, 364, 395, 414
 - begin(), метод, 382, 416
 - drawContents(), метод, 406, 415
 - idealWidth(), метод, 221
 - print(), метод, 404, 409
 - setDefaultFont(), метод, 398
 - setHtml(), метод, 220, 396, 415
 - setPageSize(), метод, 403
 - setUseDesignMetrics(), метод, 415
 - toPlainText(), метод, 53
 - качество файла, 391
- QTextEdit, класс, 351, 365
 - fontWeight(), метод, 380
 - mergeCurrentCharFormat(), метод, 379
 - setFontItalic(), метод, 374
 - setHtml(), метод, 385
 - setTabChangesFocus(), метод, 375
 - setTextColor(), метод, 378, 387
 - setTextCursor(), метод, 388
 - setWordWrapMode(), метод, 375
 - textColor(), метод, 387
 - toHtml(), метод, 381
 - сравнение с QPlainTextEdit, 352
- QTextFormat, класс, 366, 402
- QTextFragment, класс, 347, 416
 - сравнение
 - с QTextDocumentFragment, 348
- QTextFrame, класс, 348, 356
- QTextFrameFormat, класс, 348, 357
- QTextImageFormat, класс, 348, 357
- QTextLength, класс, 401
- QTextList, класс, 347, 356
- QTextListFormat, класс, 358
- QTextOption, класс, 215
- QTextStream, класс
 - setCodec(), метод, 405
- QTextTable, класс, 348, 357, 400
- QTextTableCell, класс, 348, 400
- QTextTableFormat, класс, 348, 358, 401
- QThreadPool, класс
 - activeThreadCount(), метод, 276, 282
 - globalInstance(), метод, 276, 283
- QThread, класс, 325, 341
 - idealThreadCount(), метод, 278, 283
 - isFinished(), метод, 324
 - isRunning(), метод, 337
 - start(), метод, 323, 336
 - wait(), метод, 320
- QTimeLine, класс, 153
- QTimer, класс, 153
 - setSingleShot(), метод, 92, 101
 - singleShot(), метод, 26, 29, 48
- QToolBar, класс, 384

- QTransform, класс, 490
 - QTreeView, класс, 106, 153, 184, 207
 - resizeColumnToContents(), метод, 87
 - setAllColumnsShowFocus(), метод, 155, 173
 - setColumnWidth(), метод, 339
 - setRowHidden(), метод, 162
 - setSortingEnabled(), метод, 334
 - QTreeWidgetItem, класс, 87
 - child(), метод, 96
 - operator<(), метод, 87
 - QTreeWidgetItem, класс, 106
 - currentItem(), метод, 93
 - invisibleRootItem(), метод, 90
 - question() (AQP), метод, 77, 312
 - QuickTime, библиотека, 77
 - QUiLoader, класс, 65
 - qUncompress(), метод, 193
 - QUrl, класс, 28
 - fromLocalFile(), метод, 61, 324
 - toString(), метод, 26
 - QVariantAnimation, класс, 525
 - QVariant, класс, 83
 - toBool(), метод, 187
 - toDate(), метод, 378
 - toDouble(), метод, 210, 299
 - toInt(), метод, 116, 145, 481
 - toPoint(), метод, 378
 - toSize(), метод, 480
 - toString(), метод, 87, 378
 - toUInt(), метод, 122
 - value<>(), метод, 378, 468, 480
 - недопустимый, 141, 143, 183, 469
 - QVector, класс, 136, 277, 336
 - append(), метод, 467
 - constBegin(), метод, 466
 - operator<<(), метод, 401
 - qVersion(), глобальная функция, 62
 - QWeakPointer, класс, 78, 316
 - QWebElement, класс, 37
 - QWebFrame, класс, 37, 55
 - evaluateJavaScript(), метод, 56
 - QWebHistory, класс, 37
 - QWebHistoryItem, класс, 37, 45
 - QWebInspector, класс, 38
 - QWebPage, класс, 37, 41, 48
 - createPlugin(), метод, 63
 - QWebSettings, класс, 37, 39, 55, 61
 - QWebView, класс, 37, 41, 423
 - load(), метод, 43
 - предопределенные действия, 41
 - QWidget, класс
 - actions(), метод, 45
 - addAction(), метод, 376
 - closeEvent(), метод, 97
 - geometry(), метод, 461
 - graphicsEffect(), метод, 524
 - hide(), метод, 334, 340
 - isWindowModified(), метод, 157, 473
 - keyPressEvent(), метод, 262, 363, 379
 - layoutDirection(), метод, 246
 - mapFromGlobal(), метод, 461
 - mapToGlobal(), метод, 380
 - minimumSizeHint(), метод, 255, 381, 522
 - mousePressEvent(), метод, 365
 - paintEvent(), метод, 256
 - palette(), метод, 245, 256, 257, 265, 366
 - pos(), метод, 461
 - resize(), метод, 409
 - restoreGeometry(), метод, 154
 - saveGeometry(), метод, 157
 - setAttribute(), метод, 120
 - setBackgroundRole(), метод, 250
 - setContextMenuPolicy(), метод, 41, 45
 - setContextMenu(), метод, 27
 - setFixedSize(), метод, 476
 - setFocus(), метод, 40, 118, 428
 - setFont(), метод, 233
 - setGraphicsEffect(), метод, 519
 - setLayout(), метод, 479
 - setMinimumSize(), метод, 259, 431
 - setPalette(), метод, 219
 - setStyleSheet(), метод, 428
 - setToolTip(), метод, 375
 - setWindowFilePath(), метод, 446, 455
 - setWindowModified(), метод, 111, 113, 153, 472
 - setWindowTitle(), метод, 153, 158, 447
 - sizeHint(), метод, 255, 260, 380, 522
 - window(), метод, 485
 - QWindow, класс
 - windowFilePath(), метод, 446, 455
 - QWriteLocker, класс, 332, 333
 - Q_WS_MAC, макрос, 42
 - Q_WS_X11, макрос, 33
 - QXmlStreamAttributes, класс, 168
 - QXmlStreamReader, класс, 168, 193, 203
 - QXmlStreamWriter, класс, 191, 192, 203
- ## R
- radiansFromDegrees() (AQP), метод, 215
 - RAII, идиома «захват ресурса есть инициализация», 79

RAND_MAX, константа, 62
 reinterpret_cast<>(), метод, 328
 render() (QGraphicsScene), метод, 455
 result() (QFutureWatcher), метод, 307
 Return, клавиша, 174
 RGB (красный, зеленый, синий), 214, 326
 RTTI (Run Time Type Information), динамическая идентификация типа данных, 255, 379, 458

S

Scintilla, текстовый редактор, 352
 SCXML (State Chart XML), язык, 506
 setAlignment(), метод, 209
 setCompleter() (QLineEdit), метод, 350
 setCompletionMode() (QCompleter), метод, 353
 setCompletionPrefix() (QCompleter), метод, 355
 setItemIcon() (QComboBox), метод, 481
 setSceneRect() (QGraphicsScene), метод, 448
 setShortcuts() (QAction), метод, 73
 setSortingEnabled() (QTreeView), метод, 334
 setTextAlignment() (QStandardItem), метод, 339
 setWordWrapMode() (QTextEdit), метод, 375
 setWordWrap() (QLabel), метод, 217
 shared_ptr() (std), метод, 79
 sizeFromContents() (QStyle), метод, 142
 snprintf(), функция, 307
 Source-Highlight Qt, библиотека, 367
 std, класс
 ceil(), функция, 471
 min_element(), функция, 466
 numeric_limits<int>::min(), функция, 471
 numeric_limits<>::max, метод, 209
 numeric_limits<>::min(), метод, 124
 numeric_limits<>::min(), функция, 470
 pow(), функция, 474
 shared_ptr(), метод, 79

T

typedef, спецификатор, 300, 310, 334, 377, 477
 typename, спецификатор, 333

U

uncaughtException() (QScriptEngine), метод, 310
 UTF-8 кодировка, 406
 UTI (Uniform Type Identifier), унифицированный идентификатор типов, 90

V

volatile bool, спецификатор, 272, 334

W

wait() (QThread), метод, 318, 335
 waitForDone() (QThreadPool), метод, 282
 waitForFinished() (QFutureWatcher), метод, 298
 warning() (AQP), метод, 76, 94, 405
 windowText() (QPalette), метод, 265

A

абзац, 347
 автозавершение, 348, 351, 352
 акселераторы, клавиши, 30
 активное ожидание, 91
 алгоритмы
 Куна–Манкреса, 30
 суммирования Кахана, 306
 альфа-канал, 327
 анимация, 154
 диалоговых окон, 517
 подсистема, 506, 517
 проскок, 509
 свойств, 506
 аудио
 категория устройства вывода, 83, 99
 метаданные, 88

Б

блокировки, 273
 блокирующие вызовы, 120
 брандмауэры, 23
 буфер обмена, 463

В

веб-приложения, 19
 взаимоблокировка, 294
 виджеты, 218
 агрегирование, 247
 координаты, 236, 238, 243
 с поддержкой Интернета, 21
 внеэкранное рисование, 499
 воспроизведение звука, 72

выделение

в документах, 355, 359, 366
текущей строки, 349

Г

градиент, 257, 266

графические элементы, координаты, 424

Д

дата/время, формат ISO 8601, 164, 211

двоичный формат файла, 122

действия, взаимно исключающие, 27

диалоговые окна, 121

анимация, 517

модальные, 77

немодальные, 121

удаление, 120

документо-ориентированные

приложения, 152

документы

вставки таблиц в, 400

выделение частей, 361

З

загрузка

данных, 124, 167

синхронная, 58

защищенные указатели, 316

звуковые подсказки, 69

значки, кэширование, 25, 32, 34, 478

И

изменение дерева, политика, 175

изменения несохраненные, 113, 118, 157

изображения

в документах, 348, 401

сохранение, 76

индикатор занятости, 323

интерполяция, 506

интерполяция свойств,

пользовательская, 525

К

Кахана алгоритм суммирования, 306

качество файла (QTextDocument), 392

клавиши

акселераторы, 30

горячие, 30, 100

клавиши-акселераторы, 30

кодеки, 70

кодировка файлов, 166, 167, 406

колесико мыши, 259

компоненты, 229, 251

конечные автоматы, подсистема, 510

конструкторы, однократный таймер, 26

контекстное меню, 380

политика, 42, 45

создание, 27, 375

координаты

логические, 424

порта просмотра, 424

порта просмотра и виджета, 246

сцены, 468

корневой узел дерева, 168

косметическое перо, 435, 470

Куна–Манкреса алгоритм, 30

кэширование

графических элементов, 499

пиктограмм, 25, 32, 34, 478

элементов, 299

Л

локализованные строки, 302

локальный цикл сообщений, 91

М

«магические числа», 89, 124

масштабирование, 413

медиаисточники, медиаузлы,

медиаустройства, 80

модели

выборка элементов, 116

для комбинированного списка, 112

допускающие изменение размера,

135, 144, 146, 179

нестроковые данные, 118, 125, 209

политика создания, 131

прокси, 112

роды, 107

сортировка, 130, 339

фильтрация, 118

модельный индекс, 108

модули

Phonon, 77, 89, 103

QtCore, 378

QtGui, 378

QtMultimedia, 104

QtNetwork, 21

QtWebKit, 20, 36

QtXml, 31

QtXmlPatterns, 68

мониторы, 294

Н

настройка принтера, 408

номера страниц, 392

О

облачные вычисления, 19
обнаружение столкновений, 433
округление чисел, 252
определение столкновений, 424
опрос, 282, 285
отложенное обновление, 233
отмена редактирования, 222
отображатели, 287, 308
отправка событий, 280

П

палитра, 218
партнеры в разработке ПО, 479
 недостатки, 70
перенаправление HTTP, 34
перетаскивание мышью, 190
переходная кривая, 506
переходы, 511
печать
 pdf- и ps-файлов, 418
 растровых изображений, 419
 спен, 454
пиктограммы, кэширование, 25, 32, 34, 478
платформенные особенности
 Mac OS X, 77, 234, 390, 409
 Windows, 280
подсветка синтаксиса, 349, 367
политики
 анимации, 69
 выбора версии сериализатора, 123
 захвата фокуса, 66, 259
 инициализации главного окна, 154
 использования if, 222
 контекстного меню, 42, 45, 375
 расстановки утверждений, 222
 сглаживания, 407, 411, 457
 сохранения параметров приложения, 35
полноэкранный режим, 100
полосы прокрутки, 242
пользовательские графические элементы, 433, 482
пользовательские события
 и вызов слота, 285
потoki, остановка, 318
потокобезопасный хеш, 331
предварительный просмотр до печати, 408
преобразование
 HTML в обычный текст, 53
 HTML в простой текст, 33

 long double в строку, 307
приложение
 завершение, 23, 97, 282
 политика сохранения параметров, 35
 системный лоток, 21
примеры
 Browser Window, приложение, 38
 Census Visualizer, приложение, 247
 Cross Fader, приложение, 314
 Find Dialog, приложение, 517
 Find Duplicates, приложение, 329, 349
 Folder View, приложение, 211
 Image2Image, приложение, 274, 349
 Matrix Quiz, приложение, 60
 Movie Jingle, приложение, 70
 New York Review of Books Viewer, приложение, 47
 Number Grid, приложение, 210, 288
 Output Sampler, приложение, 391
 Page Designer, приложение, 443, 507
 Petri Dish, приложение, 425, 511
 Play Music, приложение, 81
 Play Video, приложение, 98
 RssPanel, приложение, 35
 Text Edit, приложение, 384
 Tiled List View, приложение, 230
 Timelog (QStandardItemModel), приложение, 151, 373
 Timelog (пользовательская модель), приложение, 172
 Weather Tray Icon, приложение, 21
 Zipcodes (QStandardItemModel), приложение, 109
 Zipcodes (пользовательская модель), приложение, 132, 223
 Zipcodes (стандартная модель), приложение, 223
прозрачность, 327
прокси-модель, 106
прокси-сервер, 23
пропорции изображения, 414, 456, 475

Р

разделители компонентов пути, 73
разрыв страниц, вставка, 399
растровые форматы, 394
редактирование на месте, 118, 142, 174, 216
редукторы, 287, 303
рисование, 218
 SVG-документов, 418
 форматированного текста, 414
 эффективность, 264

С

свойства

- анимация, 506
- интерполяция пользовательская, 525
- система в Qt, 453, 484, 506, 511
- формат, 366, 402

сглаживание, 457

- политика, 407, 411, 457

сериализатор, политика выбора версии, 123

сигналы межпотокковые, 327

события

- отправка, 280
- пользовательские, 280

соединение сигнала с сигналом, 251, 386

сокрытие строк, 162

состояния, 510

сохранение данных, 122, 165

строки, кэширование, 127

стыкуемые окна, 444

суррогатные элементы, 289

сцены, 450

- загрузка, 451
- индексирование, 428
- экспорт, 454

Т

таблицы, 400

- в документах, 348

текстовый курсор, 355

типы MIME, 88, 89, 180, 190, 463

точность и эффективность, 337

тракт медийный, 81

У

удаление строк модели, 119

указатели, защищенные

- и интеллектуальные, 316

указатели, охраняемые

- и интеллектуальные, 78

указатель на метод, 157

упорядочение деревьев, 87

утверждения `Q_ASSERT()`, 222**Ф**

файлы

- bmp, 407, 419
- gif, 70
- html, 394
- jpg и jpeg, 407
- mng, 70
- ods, 393
- odt, 393, 404

oga и ogg, 70

pdf и ps, 392, 403, 418

png, 394, 407

ppm, 407

pro, 134

qm, 302

svg, 392, 406, 413, 418, 456

tiff, 407

wav, 69

xpm, 407

двоичные форматы, 122

загрузка, 28

изменение двоичного формата, 123

печать, 418

суффиксы, 74, 76, 86, 88, 89

форматы, 74

экспорт, 403, 456

фигуры пользовательские, 433

фильтры, 286, 296

событий, 100, 251, 260

формат, свойства, 402

форматирование символов, 347

функторы, 277, 297, 300, 305

функции высшего порядка, 286

функциональное программирование, 286

функция шаблонная, 158

Х

хеш, потокобезопасный, 331

Ц

циклы

- событий, 91, 101
- сообщений, 327

К

числа

- локализация, 117
- округление, 252
- с плавающей точкой
 - суммирование, 306
 - формат, 123

экранирование HTML и XML, 166, 168, 397

экспорт

- в растровых форматах, 407
- в формате ODT, 404
- в формате SVG, 406
- эластичный контур, 474