Міністерство освіти і науки України
ДВНЗ “Прикарпатський національний університет імені Василя Стефаника”
Кафедра комп’ютерної інженерії та електроніки

Методичний посібник
до виконання лабораторних робіт з дисципліни
“ПРИКЛАДНІ ПРОГРАМИ КОМП’ЮТЕРНОЇ ІНЖЕНЕРІЇ”

Затверджено
на засіданні кафедри радіофізики і електроніки
протокол № 4 від 8 листопада 2017 р

Розроблено:
доц. Голота В.І.

Івано-Франківськ 2017

Зміст

1. Бібліотека Qt і її модулі ... 3
2. Елементи керування ... 12
3. Кнопки, прапорці, перемикачі, елементи налаштувань ... 23
4. Автоматичне розміщення елементів ... 34
5. Елементи введення ... 45
6. Елементи вибору ... 56
7. Архітектура “модель-подання” ... 65
8. Технологія “модель-подання” для ієрархічних даних ... 72
9. Обробка подій ... 85
10. Qt Designer. Швидке розроблення прототипів ... 93
11. Інтернаціоналізація застосувань ... 98

Лабораторна робота № 1

1. Бібліотека Qt і її модулі

Мета роботи: вивчення складу бібліотеки Qt-4.8 і отримання практичного досвіду роботи з модулями.

1. Модулі Qt

Бібліотека **Qt** – це множина класів (більше 500). Люба Qt-програма використовує хоча б один із цих модулів, у більшості випадків це `QtCore` і `QtGui`. Для використання інших модулів в проектах необхідно їх перерахувати у проектному файлі, наприклад:

```
QT += opengl network sql
```

Перелік найбільш використовуваних модулів наведено в табл. 1.

Таблиця 1 – Модулі Qt

Бібліотека	Позначення в проектному файлі	Призначення
QtCore	core	Основний модуль, який складається з класів не зв'язаних з графічним інтерфейсом
QtGui	gui	Модуль для програмування графічного інтерфейсу
QtNetwork	network	Модуль для програмування мережі
QtOpenGL	opengl	Модуль для програмування графіки OGL
QtSql	sql	Модуль для програмування баз даних
QtSvg	svg	Модуль для роботи з SVG (Scalable Vector Graphics, масштабована векторна графіка)
QtXml	xml	Модуль підтримки XML, класи, які відносяться до SAX і DOM
QtScript	script	Модуль підтримки мови сценаріїв
Phonon	phonon	Модуль мультимедіа
QtWebKit	webkit	Модуль для створення Web-додатків
QtScriptTools	scripttools	Модуль підтримки додаткових можливостей мови сценарії (зневадник)
QtTest	test	Модуль, який містить класи для тестування коду

2. Простір імен Qt

Простір імен Qt має ряд констант і перечислень, які використовуються при програмуванні. Якщо необхідно отримати доступ до якої-небудь константи простору необхідно вказати префікс Qt, наприклад: `Qt::red`. Для доступу до змінної без префіксу необхідно на початку файлу добавити наступну директиву:

```
using namespace Qt;
```

3. Модуль QtCore

Модуль **QtCore** є базовим для застосувань і не містить класів, які мають відношення до інтерфейсу користувача. В модуль QtCore входить більш 200 класів, от деякі з них:

• контейнерні класи QList, QVector, QMap;
• класи для введення і виведення QIODevice, QTextStream, QFile;

• класи процесу QProcess і для програмування багатопоточності QThread, QWaitCondition, QMutex;

• класи для роботи з таймером QBasicTimer і QTimer;

• класи для роботи з датою і часом QDate і QTime;

• клас QObject, є *наріжним каменем* об’єктної моделі Qt;

• базовий клас подій QEvent;

• клас для зберігання налаштувань застосування QSettings;

• клас застосування QCoreApplication, з об’єкта якого, якщо потрібно, можна запустити цикл подій.

4. Модуль QtGui

Модуль **QtGui** містить в собі класи, необхідні для програмування графічного інтерфейсу користувача. В цей модуль входить до 300 класів. Ось деякі з них:

• клас QWidget – базовий клас для всіх елементів керування бібліотеки Qt. За своїм зовнішнім видом це заповнений чотирикутник, але за зовнішньою простотою приховується великий потенціал непростих функціональних можливостей. Клас нараховує 254 методи і 53 властивості;

• класи для автоматичного розміщення елементів QVBoxLayout, QHBoxLayout;

• класи елементів відображення QLabel, QLCDNumber;

• класи кнопок QPushButton, QCheckBox, QRadioButton;

• класи елементів налаштувань QSlider, QScrollBar;

• класи елементів введення QLineEdit, QSpinBox;

• класи елементів вибору QComboBox, QToolBox;

• класи меню QMainWindow и QMenu;

• класи вікон повідомлень і діалогових вікон QMessageBox, QDialog;

• класи для малювання QPainter, QBrush, QPen, QColor;

• класи для растрових зображень QImage, QPixmap;

• класи стилів QMotifStyle, QWindowsStyle і др. Як окремому елементу, так і всьому застосуванню може бути присвоєний певний стиль, який змінює їх зовнішній вигляд;

• клас застосування QApplication, який відповідає за цикл подій.

Об’єкт класу QApplication є центральним контрольним пунктом всіх Qt-застосувань, які мають інтерфейс користувача. Даний об’єкт використовується для отримання подій клавіатури, миші, таймера і інших подій, на які застосування повинно реагувати відповідним чином.

В призначення QApplication входить:

• встановлення стилю застосування. Так можна встановити стиль Motif, Windows, а також інші *види і поведінки* (Look & Feel) застосування, включаючи і свої власні;

• отримання вказівника на об’єкт *робочого столу* (desktop);

• отримання доступу до буферу обміну;

• керування глобальними маніпуляціями з мишею (наприклад, встановлення інтервалу подвійного клацання кнопкою миші) і реєстрація руху миші в межах і за межами вікна застосування;

• видача попереджувального звукового сигналу;

• забезпечення правильного завершення працюючого застосування при завершенні роботи ОС;

• ініціалізація необхідних налаштувань застосування, наприклад палітри для забарвлення елементів керування.

5. Створення виконуваних модулів

Qt проекти створюються з використанням утиліти `qmake` або вручну. При створенні виконуваного модуля використовуються проектний файл, сирцеві файли і файли ресурсів. Схема створення виконуваного модуля показана на рис. 1.

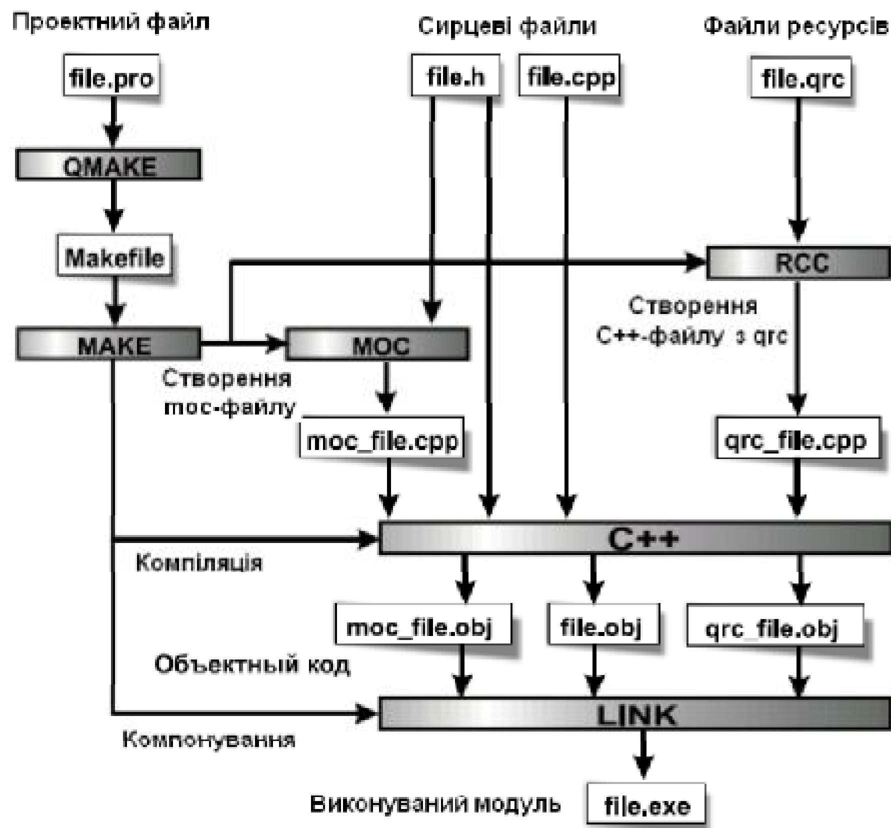

Рисунок 1 – Схема створення виконуваного модуля

Маючи в каталозі файли `*.cpp`, `*.h`, виконуваний файл можна отримати послідовно виконавши наступні команди:

```
>qmake –project
>qmake
>make
```

Команди можна виконати в одному рядку:

```
>qmake –project; qmake; make
```

6. Перша програм на Qt

Програма створює графічно вікно і виводить туди надпис “Hello world”.

Приклад 1. Виведення повідомлення в графічне вікно.

```
#include <QtGui>
int main(int argc, char** argv)
{
QApplication app(argc, argv); // об’єкт для керування застосуванням
QLabel lbl("Hello, World !"); // об’єкт класу QLabel  - надпис
lbl.show();                   // показати елемент на екран
return app.exec();            // запуск застосування + цикл обробки подій
}
```

7. Об’єктна модель Qt

Об'єктна модель Qt припускає, що все побудовано на об'єктах. Фактично, клас `QObject` – основний, базовий клас. Переважаюча більшість класів Qt є його нащадками. Класи, які мають сигнали і слоти, повинні бути успадковані від цього класу.

Клас `QObject` містить у собі підтримку:

• сигналів і слотів (`signal/slot`);
• таймера;
• механізму об'єднання об'єктів в ієрархії;
• подій і механізму їх фільтрації;
• організації об'єктних ієрархій;
• метаоб'єктної інформації;
• перетворення типів;
• властивостей.

Сигнали і слоти – це засоби, які дозволяють ефективно обмінюватися інформацією про події, які виробляються об'єктами.

Підтримка таймера дає можливість кожному з класів, успадкованих від класу `QObject`, не створювати додаткового об'єкта таймера. Тим самим економиться час на розробку.

Механізм об'єднання об'єктів в ієрархічні структури дозволяє різко скоротити часові затрати при розробці застосувань, не турбуючись про звільнення пам'яті створюваних об'єктів, так як об'єкти-предки самі відповідають за знищення своїх потомків.

Механізм фільтрації подій дозволяє здійснити їх перехоплення. Фільтр подій може бути встановлений у любому класі, успадкованому від класу `QObject`, завдяки цьому можна змінювати реакцію об'єктів на виникаючі події без зміни сирцевого коду класу.

Метаоб'єктна інформація включає в себе інформацію про успадкування класів, що дозволяє визначати, чи є класи безпосередніми нащадками, а також взнати ім'я класу.

Для *перетворення типів* Qt надає шаблонну функцію `qobject_cast<T>()`, яка базується на метаінформації, яку створює метаоб'єктний компілятор MOC, для класів, успадкованих від `QObject`.

Властивості – це поля, для яких обов'язково мають існувати методи читання. За їх допомогою можна отримати доступ до атрибутів об'єктів зовні, наприклад з мови сценарію `QtScript`. Властивості також широко використовуються у візуальному середовищі розробки Qt Designer. Цей механізм реалізований в Qt за допомогою директив препроцесора. Задаються властивості макросом `Q_PROPERTY`. Визначення властивостей у загальному виді наступне:

```
Q_PROPERTY(type name             // ім'я властивості
           READ getFunction      // ім'я методу читання
           [WRITE setFunction]   // ім'я методу запису
           [RESET resetFunction] // ім'я методу скидування значення
           [DESIGNABLE bool]     // логічний (булевий) вираз, який вказує чи
// повинна властивість появлятися в інспекторі властивостей Qt Designer
           [SCRIPTABLE bool]     // логічний (булевий) вираз, який вказує чи
// буде властивість доступна у мові сценаріїв Qt Script

           [STORED bool]  // логічний (булевий) вираз, який керує серіалізацією,
// тобто вказує, чи буде властивість запам'ятовуватися при зберіганні об'єкта

)
```

Приклад визначення властивості `m_bReadOnly` для читання в класі:

```
class MyClass : public QObject {
Q_OBJECT
Q_PROPERTY(bool readOnly
           READ isReadOnly
           WRITE setReadOnly)
private:
      bool m_bReadOnly;
public:
     MyClass(QObject* pobj = 0) : QObject(pobj), m_bReadOnly(false)
```

```
    {
    }
public:
    void setReadOnly(bool bReadOnly)
    {
        m_bReadOnly = bReadOnly;
    }
    bool isReadOnly() const
    {
        return m_bReadOnly;
    }
}
```

З програми можна змінити значення властивості і отримати поточне значення:

```
pobj->setProperty("readOnly", true);
bool bReadOnly = pobj->property("readOnly").toBool();
```

8. Механізми сигналів і слотів

Сигнал визначається у класі, як звичайний метод, але без реалізації. Сигнал не повертає значень, тому перед ним ставлять `void`. Для відправлення сигналу використовується конструкція `emit doIt();` Сигнал не обов'язково з'єднується зі слотом. Якщо з'єднання не відбулося, то він просто не буде оброблятися.

В бібліотеках є велика кількість сигналів для існуючих елементів керування, але іноді виникає необхідність в реалізації нових сигналів у класах користувача.

Приклад реалізації сигналу без і з параметром:

```
class MySignal {
Q_OBJECT
public: void sendSignal()
        { emit doIt(); }
signals:
        void doIt();
...
};
```

```
class MySignal {
Q_OBJECT
public: void sendSignal()
  { emit sendString("Hello"); }
signals:
   void sendString(const QString&);
...
};
```

Слоти (slots) – це методи, які приєднуються до сигналів. По суті, вони є звичайними методами. Найбільша їх відмінність полягає у тому, що вони можуть приймати сигнали. Як і звичайні методи, вони визначаються у класі як `public`, `private` або `protected`. Відповідно, перед кожною групою слотів має стояти: `private slots:`, `protected slots:` або `public slots:`. Слоти можуть бути і віртуальними.

Класи бібліотек містять цілий ряд вже реалізованих слотів, але часто виникає необхідність реалізації слотів для своїх класів.

Приклад реалізації слоту:

```
class MySlot : public QObject {
Q_OBJECT
public:
    MySlot();
public slots:
    void slot()
    {
      qDebug() << "Це слот";
    }
};
```

Всередині слоту викликом метода `sender()` можна взнати, від якого об'єкта був висланий сигнал. Він повертає вказівник на об'єкт типа `QObject`. Наприклад, у цьому випадку на консоль буде виведено ім'я об'єкта, який вислав сигнал:

```
void slot()
{
   qDebug() << sender()->objectName();
}
```

Для з'єднання об'єктів використовується статичний метод `connect()`, який визначений в класі `QObject`. У загальному випадку виклик метода `connect()` має наступний вид:

```
QObject::connect(const QObject* sender,
                 const char* signal,
                 const QObject* receiver,
                 const char* slot,
                 Qt::ConnectionType type = Qt::AutoConnection
                 );
```

Йому передаються наступні параметри:

• `sender` – вказівник на об'єкт, який відправляє сигнал;

• `signal` – це сигнал, з яким здійснюється з'єднання. Прототип (ім'я і аргументи) метода сигналу повинен бути поміщений у спеціальний макрос `SIGNAL(method());`

• `receiver` – вказівник на об'єкт, який має слот для оброблення сигналу;

• `slot` – слот, який викликається при отриманні сигналу. Прототип слоту повинен бути поміщений в спеціальний макрос `SLOT(method());`

• `type` – керує режимом оброблення (`Qt::AutoConnection` (за замовчуванням), `Qt::DirectConnection`, `Qt::QueuedConnection`).

Приклад з'єднання об'єктів в програмі:

```
void main()
{
    QObject::connect(pSender, SIGNAL(signalMethod()),
                     pReceiver, SLOT(slotMethod())
                     );
}
```

Якщо виклик відбувається з класу, успадкованого від `QObject`, тоді `QObject::` можна опустити:

```
MyClass::MyClass() : QObject()
{
...
connect(pSender, SIGNAL(signalMethod()),
        pReceiver, SLOT(slotMethod())
       );
...
}
```

При з'єднанні сигналів і слотів, з передачею значень, необхідно щоб тип значення яке передається співпадав із типом значення яке приймається.

У випадку, якщо слот міститься у класі, з якого здійснюється з'єднання, то можна опустити третій параметр (`pReceiver`), вказуючий на об'єкт-отримувач.

```
MyClass::MyClass() : QObject()
{
   connect(pSender, SIGNAL(signalMethod()), SLOT(slot()));
}
void MyClass::slot()
{
   qDebug() << "Це слот";
}
```

Іноді об'єкт не обробляє сигнал, а просто передає його далі, тоді можна просто з'єднати сигнали один з одним. Сигнал для відправлення має міститися у визначенні класу.

```
MyClass::MyClass() : QObject()
{
   connect(pSender, SIGNAL(signalMethod()), SIGNAL(mySignal()));
}
```

У Qt, при знищенні об’єктів, всі зв'язані з ним з’єднання знищуються автоматично. У деяких випадках може виникнути необхідність знищення цих об’єктів вручну. Для цього існує статичний метод `disconnect()`, параметри якого аналогічні параметрам метода `connect()`. Приклад роз’єднання об’єктів у програмі:

```
void main()
{
... QObject::disconnect(pSender, SIGNAL(signalMethod()),
                        pReceiver, SLOT(slotMethod())
                        );
...
}
```

Приклад 2. Застосування з двох вікон. В першому вікні знаходиться кнопка для натискання, а у другому віджеті надпис. При клацанні на кнопці ADD збільшується відображуване значення надпису на 1, рис.2.

```
// ==========================================================================
//  Counter.h  - оголошення класу
// ==========================================================================
#ifndef _Counter_h_
#define _Counter_h_
#include <QObject>

// ==========================================================================
class Counter : public QObject {
    Q_OBJECT
private:
    int m_nValue;   // лічильник

public:
    Counter();      // конструктор

public slots:
    void slotInc();  // слот лічильник

signals:
    void goodbye();  // сигнал завершення
    void counterChanged(int);  // сигнал зміни лічильника
};
#endif  //_Counter_h_

 // ==========================================================================
//  Counter.cpp – реалізація методів
// ==========================================================================
#include "Counter.h"

// --------------------------------------------------------------------------
Counter::Counter() : QObject(), m_nValue(0) { }

// --------------------------------------------------------------------------
void Counter::slotInc()
{
    emit counterChanged(++m_nValue);

    if (m_nValue == 5) {
        emit goodbye();
    }
}

// ==========================================================================
//  main.cpp – основна програма
// ==========================================================================
```

```
#include <QtGui>
#include "Counter.h"

// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);  // створення графічного вікна

    QLabel      lbl("0");          // створення об'єкту надпису
    QPushButton cmd("ADD");        // створення об'єкту кнопки
    Counter     counter;           // створення об'єкту лічильника

    lbl.show();                    // показати надпис
    cmd.show();                    // показати кнопку

    // сигнал cliked з'єднується зі слотом slotInc
    QObject::connect(&cmd, SIGNAL(clicked()),
                     &counter, SLOT(slotInc())
                    );

   // сигнал conterChanged з'єднується зі слотом setNum для відображення
   // змін лічильника
    QObject::connect(&counter, SIGNAL(counterChanged(int)),
                     &lbl, SLOT(setNum(int))
                    );
   // сигнал завершення роботи лічильника goodbye() з'єднується зі слотом об'єкта
app
   // quit(), який завершує роботу
    QObject::connect(&counter, SIGNAL(goodbye()),
                     &app, SLOT(quit())
                    );

    return app.exec();
}
```

Створення виконуваного файлу:

```
> qmake -project
> qmake
> make
```

Запуск на виконання:

```
> ./Counter
```

Рисунок 2 – Програма лічильник

Запитання.

1. Модулі Qt і їх призначення.
2. Для чого призначений модуль `QtCore`.
2. Для чого призначений модуль `QtGui`.
3. Як створюються виконувані модулі для Qt-проектів .
4. Об'єктна модель Qt. Яку підтримку надає клас `QObject`.
5. Механізм сигналів і слотів в Qt.

Завдання.

1. Запустити на виконання програму прикладу 1.

2. Запустити на виконання програму прикладу 2, в якій використовується механізм сигналів і слотів. Прокоментувати сирцевий код програми.

3. Модифікувати програму так, щоб із сигналом передати тестове повідомлення, яке висвічується у додатковому віджеті `QLabel`.

Лабораторна робота № 2

2. Елементи керування

Мета роботи: отримання практичних навиків роботи з елементами керування.

1. Короткі теоретичні відомості

Клас `QWidget()` є фундаментальним для усіх віджетів і має наступних нащадків:

Таблиця 1 – Ієрархія класів віджетів

QWidget			
QDialog QPrintDialog QColorDialog QErrorMessage QFileDialog QFontDialog QInputDialog QMessageBox QPageSetupDialog QProgressDialog QWizard	QFrame QAbstractScrollArea QAbstractItemView QHeaderView QListView QListWidget QUndoView QTableView QTableWidget QTreeView QTreeWidget QColumnView	QAbstactSpinBox QDateTimeEdit QDateEdit QTimeEdit QDoubleSpinBox QSpinBox QComboBox QFontComboBox QAxWidget QCalendarWidget QColumnViewGrip	QLineEdit QMainWindow QMdiSubWindow QMenu QMenuBar QPrintPreviewWg QProgressBar QRubberBund QSizeGrip QSplashScreen QSplitterHandle QStatusBar
QAstractButton QCheckBox QPushButton QRadioButton QToolButton	QGraphicsView QMdiAre QPlainTextEdit QScrollArea QTextEdit QTextBrowser	QDesignerAction QDesignerForm QDesignerObject QDesignerProperty QDesignerWidget	QSvgWidget QTabBar QTabWidget QToolBar QWSEmbeddedWg QWizardPage
QAbstractSlider QDial QScroolBar QSlider	QLCDNumber QLabel QSplitter QStackedWidget QToolBox	QDesktopWidget QDialogButtonBox QDockWidget QFocusFrame QGLWidget QGroupBox	QWorkSpace QWorkSpaceTitleBar QX11EmbedContainer QX11EmbedWidget

Для відображення віджетів на екрані викликається метод `show()`, а для приховування – `hide()`.

Клас `QWidget` і успадковані від нього класи мають конструктор з двома параметрами

```
QWidget(QWidget* pwgt = 0, Qt::WindowFlags f = 0 );
```

Якщо конструктор викликається без параметрів, то створений віджет буде віджетом верхнього рівня. Другий параметр `Qt::WindowFlags` задає властивості вікна. Аналогічного результату можна добитися викликом метода `setWindowFlags()`, наприклад:

```
wgt.setWindowFlags(Qt::Window | Qt::WindowTitleHint |
Qt::WindowStaysOnTopHint);
```

Слот-метод `setWindowTitle()` встановлює надпис вікна

```
wgt.setWindowTitle("My Window");
```

Слот-метод `setEnable()` з параметром `true` встановлює вікно у доступний стан, а з параметром `false` – у недоступний. Стан вікна повертає метод `isEnabled()`.

1.1. Розміри віджету

Методи `size()`, `height()` і `width()` повертають розміри віджету.
Методи `x()`, `y()` і `pos()` повертають координати віджету.
Метод `geometry()` повертає координати і розмір віджету.
Метод `move()` змінює координати, а метод `resize()`– розміри віджету.
Метод `setGeometry()` змінює координати і розміри віджету.

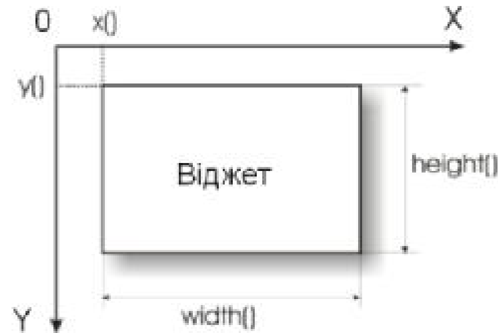

Рисунок 1 – Віджет в області екрану

1.2. Встановлення фону віджета

Віджет має властивість `autoFillBackground` (за замовчування `false`), при якій не заповнюється фон. При встановленні цієї властивості в `true`, віджет автоматично заповнить фон (кольором або растровим зображенням):

```
wgt.setAutoFillBackground(true);
```

Приклад 1. Віджет з фоном.

```
// =============================================================
//  main.cpp
// =============================================================
#include <QtGui>
// -------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget wgt;    // відмет верхнього рівня

    QWidget* pwgt1 = new QWidget(&wgt); // віджет1 нижнього рівня
    QPalette pal1;                      // створення палітри 1
    pal1.setColor(pwgt1->backgroundRole(), Qt::blue);// встановлення кольору
палітри
    pwgt1->setPalette(pal1); // передача палітри у віджет1
    pwgt1->resize(100, 100); // зміна розміру віджета1
    pwgt1->move(25, 25);     // зміна координати віджета1
    pwgt1->setAutoFillBackground(true); // заповнення фону віджета1

    QWidget* pwgt2 = new QWidget(&wgt);  // віджет2 нижнього рівня
    QPalette pal2;   створення палітри 2
    // встановлення растрового зображення для палітри
    pal2.setBrush(pwgt2->backgroundRole(), QBrush(QPixmap("stone.jpg")));
    pwgt2->setPalette(pal2);  // передача палітри у віджет2
    pwgt2->resize(100, 100);  // зміна розміру віджета2
    pwgt2->move(75, 75);      // зміна координати віджета2
    pwgt2->setAutoFillBackground(true);  // заповнення фону віджета2
```

```
    wgt.resize(200, 200);    // зміна розміру віджета верхнього рівня
    wgt.show();              // відображення віджета верхнього рівня
    return app.exec();
}
```

```
> qmake -project
> qmake
> make
```

Запуск на виконання:

```
> ./BackGround
```

Рисунок 2 – Віджет з фоном

1.3. Зміна вказівника миші

Клас вказівника миші `QCursor` визначений у файлі `QCursor`. Вказівник миші встановлюється методом `setCursor()` з параметром значення `shapeCursor` (визначене в просторі імен Qt):

`ArrowCursor` – стандартний вказівник;
`UpArrowCorsor` – ↑;
`CrossCursor` – +;
`WaitCursor` – пісчаний годинник;
`SizeHorCursor` – ↔;
`SizeVerCursor` - ↕;
…
`Blank Cursor` – пустий вказівник, вказує на неможливість використання миші.

Методи `QCursor.pos()`, `QCusor.setPos()` повертають і встановлюють поточну позицію вказівника миші. Для встановлення власного зображення вказівника миші можна використати об'єкт класу `QPixmap`.

Приклад 2. Власне зображення вказівника миші.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;    // створення віджета
    QPixmap      pix("clock.png");  // створення об'єкта растрового зображення
    QCursor      cur(pix);   // створення вказівника миші з растрового зображення

    wgt.setCursor(cur);      // встановлення курсора у віджеті
    wgt.resize(180, 100);  // зміна розмірів віджета
    wgt.show();            // відображення віджета

    return app.exec();
}
```

```
> qmake -project
```

```
> qmake
> make
```

Запуск на виконання:

```
> ./MouseCursor
```

1.4. Віджет видової прокрутки

Базовий клас для видової прокрутки `QAbstractScrollArea` успадкований від класу `QFrame` і є вікном для прогляду тільки частини інформації. Сам віджет видової прокрутки реалізує клас `QScrollArea`. Поміщення горизонтальної і вертикальної смуг прокрутки у віджет:

```
QScrollArea sa;
sa.setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
sa.setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
```

Віджет видової прокрутки є сукупністю декількох віджетів, які працюють одночасно. Вказівники на ці віджети можна отримати методами, показаними на рис. 3.

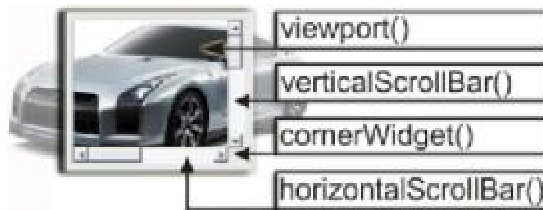

Рисунок 3 – Структура віджета видової прокрутки

Встановити віджет у віджет видової прокрутки можна методом `setWidget()`, передавши у нього вказівник на віджет. Ця операція автоматично зробить переданий віджет нащадком віджета області прогляду. Вказівник встановленого віджета завжди можна отримати методом `widget()`. Вилучає віджет з `QScrollArea` метод `removeChild()`.

Приклад 3. Програма видової прокрутки.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QScrollArea  sa;  // створення віджета видової прокрутки

    QWidget*     pwgt = new QWidget;  // створення звичайного віджета
    QPixmap      pix("img.jpg");      // створення об'єкта растрового зображення
                                      // і ініціалізація його файлом img.jpg
    QPalette pal;                     // створення об'єкта палітри
    pal.setBrush(pwgt->backgroundRole(), QBrush(pix));  // задній фон
    pwgt->setPalette(pal);     // встановлення заднього фону віджета
    pwgt->setAutoFillBackground(true); // авто заповнення заднього фону
    pwgt->setFixedSize(pix.width(), pix.height()); // розміри віджета приводяться
у
                               // відповідність до розмірів растрового зображення

    sa.setWidget(pwgt);        // віджет видової прокрутки добавляє у своє вікно
                               // створений віджет
    sa.resize(350, 150);       // зміна розмірів віджету видової прокрутки
    sa.show();                 // відображення віджета видової прокрутки

    return app.exec();
}
```

```
> qmake -project
> qmake
> make
```

Запуск на виконання:

```
> ./ScrollArea
```

…

1.5. Елементи відображення

Елементи відображення використовуються тільки для інформування користувача. Вони можуть відображати як текстову, так і графічну інформацію.

Надписи. Клас віджета надпису `QLabel` визначений у заголовковому файлі `QLabel`. Клас успадкований від классу `Frame` і тому може мати рамку. Для відображення інформації текстового, графічного і анімаційного характеру використовуються методи `setText()`, `setPixmap()`, `setMovie()`.

Розміщенням тексту керують методом `setAlignment()` з використанням прапорів або їх комбінацій (об'єднанням за допомогою логічного АБО "|"). Значення прапорів `AlignmentFlag` простору імен `Qt: AlignLeft, AlignRight, AlignHCenter, AlignJustify, AlignTop, AlignBottom, AlighVCenter, AlignCenter.`

Приклад 4. Віджет надпису може відображати звичайний текст і текст у форматі HTML, який містить зображення.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    // в об'єкт lbl параметром передається текст в форматі HTML
    // текст можна також передати методом setText()
    QLabel lbl("<H1><CENTER>QLabel - HTML Demo</CENTER></H1>"
               "<H2><CENTER>Image</CENTER><H2>"
               "<CENTER><IMG BORDER=\"0\" SRC=\"img.jpg\"></CENTER>"
               "<H2><CENTER>List</CENTER><H2>"
               "<OL><LI>One</LI>"
               "    <LI>Two</LI>"
               "    <LI>Three</LI>"
               "</OL>"
               "<H2><CENTER>Font Style</CENTER><H2>"
               "<CENTER><FONT COLOR=RED>"
               "    <B>Bold</B>, <I>Italic</I>, <U>Underline</U>"
               "</FONT></CENTER>"
               "<H2><CENTER>Table</CENTER></H2>"
               "<CENTER> <TABLE>"
               "    <TR BGCOLOR=#ff00ff>"
               "        <TD>1,1</TD><TD>1,2</TD><TD>1,3</TD><TD>1,4</TD>"
               "    </TR>"
               "    <TR BGCOLOR=YELLOW>"
               "        <TD>2,1</TD><TD>2,2</TD><TD>2,3</TD><TD>2,4</TD>"
               "    </TR>"
               "    <TR BGCOLOR=#00f000>"
               "        <TD>3,1</TD><TD>3,2</TD><TD>3,3</TD><TD>3,4</TD>"
               "    </TR>"
               "</TABLE> </CENTER>"
              );
    lbl.show();
    Return app.exec();
}
```

```
> qmake -project
> qmake
> make
```

Запуск на виконання:

```
> ./Label
```

Рисунок 4 – Відображення інформації віджетом надпису у форматі HTML

Приклад 5. Відображувати графічну інформацію можна і без використання формату HTML.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QPixmap pix;             // створення об’єкта растрового зображення
    pix.load("mira.jpg");    // і завантаження в нього файлу зображення

    QLabel lbl;              // створення об’єкта надпису
    lbl.resize(pix.size()); // приведення у відповідність розміру об’єкта надпису
                             // до розміру растрового зображення
    lbl.setPixmap(pix);      // встановлення зображення у віджет надпис
    lbl.show();              // відображення надпису
    return app.exec();
}
```

```
> qmake -project
> qmake
> make
```

Запуск на виконання:

```
> ./LabelPixmap
```

Рисунок 5 – Відображення графічної інформації віджетом надпису

За допомогою методу `setBuddy()` віджет надпису можна асоціювати з любим іншим віджетом. Якщо текст надпису містить знак &, то символ, перед яким він стоїть, буде підкресленим. При сумісному натисканні цього символу з клавішею <Alt> фокус перейде до віджету, встановленого методом `setBuddy()` (зв'язування віджетів).

Приклад 6. Асоціювання надпису з іншим віджетом

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget    wgt;   // створення віджету верхнього рівня
    // віджет надпису з іменем Name (N – буква швидкого доступу <ALT>+<N>)
    QLabel*    plblName = new QLabel("&Name:");
    QLineEdit* ptxtName = new QLineEdit; // віджет однорядкового текстового поля
    plblName->setBuddy(ptxtName); // зв'язування віджету надпису з текстовим полем

    // віджет надпису з іменем Age (A – буква швидкого доступу <ALT>+<A>)
    QLabel*    plblAge = new QLabel("&Age:");
    QSpinBox*  pspbAge = new QSpinBox;  // віджет лічильника SpinBox
    plblAge->setBuddy(pspbAge); // зв'язування віджету надпису з лічильником

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;  // встановлення віджету 
вертикального
                                              // розміщення віджетів нащадків
    pvbxLayout->addWidget(plblName); // добавлення віджетів у розміщення
    pvbxLayout->addWidget(ptxtName);
    pvbxLayout->addWidget(plblAge);
    pvbxLayout->addWidget(pspbAge);
    wgt.setLayout(pvbxLayout);       // встановлення вертикального розміщення

    wgt.show();
    return app.exec();
}
```

```
> qmake -project
> qmake
> make
```

Запуск на виконання:

```
> ./LabelBuddy
```

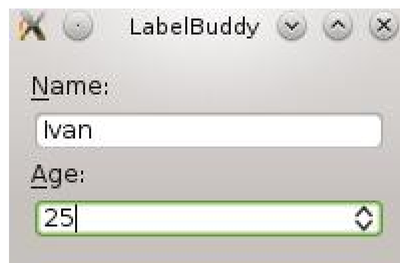

Рисунок 6 – Зв'язування віджетів і використання символу “&”

Клас `QLabel` підтримує гіпертекстові посилання. Для цього достатньо викликати метод `setOpenExternalLinks()` з параметром `true`, наприклад

```
QLabel* plbl = new QLabel("<A REF=\"www.xxx.com\">http://www.xxx.com</A>");
lbl.setOpenExternalLinks(true);
```

Приклад. 7. Підтримка віджетом надпису гіпертекстового посилання.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    // створення віджету надпису
    QLabel lbl("<CENTER><A HREF=\"LINK\">http://www.goole.com</A></CENTER>");
    // включення підтримки HTML посилань
    lbl.setOpenExternalLinks(true);
    lbl.show();

    return app.exec();
}
```

```
> qmake -project
> qmake
> make
```

Запуск на виконання:

```
> ./LinkLabel
```

Рисунок 7 – Віджет надпис з підтримкою HTML посилань

1.6. Індикатор прогресу

Індикатор прогресу – це віджет, який показує процес виконання операції шляхом заповнення смуги зліва направо. Клас `QProressBar` визначений у заголовковому файлі `QProgressBar`. Розміщення індикатора прогресу в горизонтальному або вертикальному положенні задається передачею у слот `setOrientation()` значення `QT:: Horizontal` або `QT::Vertical`.

Приклад 7. Індикатор прогресу.

```
// ======================================================================
//  Progress.h
// ======================================================================
#ifndef _Progress_h_
#define _Progress_h_

#include <QWidget>

class QProgressBar;

// ======================================================================
class Progress : public QWidget {
    Q_OBJECT
private:
    QProgressBar* m_pprb;   // вказівник на віджет
    int           m_nStep;  // номер кроку
```

```
public:
    Progress(QWidget* pobj = 0);

public slots:
    void slotStep ();      // нарощення номера кроку на 1
    void slotReset();      // скидування індикатора в 0
};

#endif  //_Progress_h_

// ======================================================================
//  Progress.cpp
// ======================================================================
#include <QtGui>
#include "Progress.h"

// ----------------------------------------------------------------------
Progress::Progress(QWidget* pwgt/*= 0*/)
    : QWidget(pwgt)
    , m_nStep(0)
{
    m_pprb = new QProgressBar;    // створення об'єкта індикатор
    m_pprb->setRange(0, 5);       // встановлення кількості кроків 5
    m_pprb->setMinimumWidth(200); // мінімальна довжина індикатора прогресу
    m_pprb->setAlignment(Qt::AlignCenter); // режим відображення % у центрі

    QPushButton* pcmdStep  = new QPushButton("&Step");  // створення кнопки Step
    QPushButton* pcmdReset = new QPushButton("&Reset"); // створення кнопки Reset
    // з'єднання кнопок із слотами індикатора прогрес
    QObject::connect(pcmdStep, SIGNAL(clicked()), SLOT(slotStep()));
    QObject::connect(pcmdReset, SIGNAL(clicked()), SLOT(slotReset()));

    // створення об'єкта горизонтального розміщення
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->addWidget(m_pprb);  // добавлення віджетів у розміщення
    phbxLayout->addWidget(pcmdStep);
    phbxLayout->addWidget(pcmdReset);
    setLayout(phbxLayout);  // ініціалізація розміщення
}

// ----------------------------------------------------------------------
void Progress::slotStep()
{
    m_pprb->setValue(++m_nStep);  // збільшення кроку на 1
}

// ----------------------------------------------------------------------
void Progress::slotReset()
{
    m_nStep = 0;
    m_pprb->reset();    // встановлення кроку в 0
}

// ======================================================================
//  main.cpp
// ======================================================================
#include <QApplication>
#include "Progress.h"

// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
```

```
    QApplication app(argc, argv);
    Progress     progress;

    progress.show();

    return app.exec();
}
```

```
> qmake -project
> qmake
> make
```

Запуск на виконання:

```
> ./Progress
```

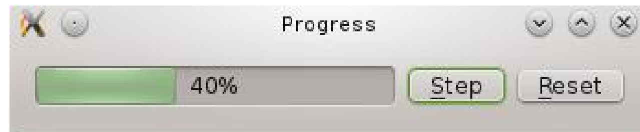

Рисунок 8 – Індикатор прогресу

1.7. Електронний індикатор

Електронний індикатор є набором сегментних вказівників, як в електронному годиннику. За допомогою цього віджета відображаються цілі числа. Допускається використання крапки між позиціями сегментів. Клас `QLCDNumber` визначений у заголовковому файлі `QLCDNumber`.

Кількість відображуваних сегментів можна задати у конструкторі або методі `setNumDigits()`. Якщо для відображення числа не вистачає числа сегментів, то відсилається сигнал `overflow()`. Для відображення крапки між позиціями сегментів використовується метод `setSmallDecimalPoint()` в який передається значення `true` або `false`. Можна задати стиль електронного індикатора `QLCDNumber::Outline`, `QLCDNumber::Filled`, `QLCDNumber::Flat`.

Електронний індикатор дозволяє відображати числа у двійковій, вісімковій, десятковій і шістнадцятковій системах числення методом `setMode()` у який передаються значення `QLCDNumber::bin`, `QLCDNumber::oct`, `QLCDNumber::dec`, `QLCDNumber::hex`.

Приклад 7. Електронний індикатор.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget     wgt;   // основний віджет
    QLCDNumber* plcd = new QLCDNumber; // створення віджета електронного
індикатора
    QSpinBox*   pspb = new QSpinBox;   // створення віджета лічильника

    plcd->setSegmentStyle(QLCDNumber::Filled); // стиль електронного індикатора
    plcd->setMode(QLCDNumber::Hex);            // 16-ва система числення

    pspb->setFixedHeight(30);     // встановлення висоти елемента лічильника
    // з'єднання сигналу лічильника зі слотом електронного індикатора
    QObject::connect(pspb, SIGNAL(valueChanged(int)),
                     plcd, SLOT(display(int))
                    );

    //   створення об'єкта вертикального розміщення
```

```
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(plcd);     // добавлення віджетів в об’єкт розміщення
    pvbxLayout->addWidget(pspb);
    wgt.setLayout(pvbxLayout);       // передача вказівника на об’єкт розміщення в
                                     // основний віджет
    wgt.resize(250, 150);            // встановлення розмірів основного віджету
    wgt.show();

    return app.exec();
}
```

```
> qmake -project
> qmake
> make
```

Запуск на виконання:

```
> ./LCD
```

Рисунок 9 – Електронний індикатор

Запитання.

1. Ієрархія класів віджетів і їх спільні методи.
2. Задання розмірів і фону віджетів.
3. Зміна вказівника миші.
4. Віджет видової прокрутки і його методи.
5. Віджет відображення інформації у текстовій і графічній формі `QLabel` і його методи
6. Асоціювання віджету надпису з іншим віджетом. Підтримка гіпертекстових посилань.
7. Індикатор прогресу і його методи.
8. Електронний індикатор і його методи.

Завдання.

1. Написати програму в якій при натисканні кнопки змінюється форма вказівника миші.
2. Написати програму, яка при натисканні кнопки розміщує текст “Привіт” у верху, у низу, з права, з ліва.
3. Написати програму в якій асоціювати один надпис з лічильником, а інший – з індикатором прогресу.
4. Написати програму, яка використовує віджет `QLCDNumber` і при натисканні кнопки відображає число у двійковій, вісімковій, десятковій і шістнадцятковій системах числення.

Лабораторна робота № 3

3. Кнопки, прапорці, перемикачі, елементи налаштувань

Мета роботи: отримання практичних навиків роботи з кнопками, прапорцями і перемикачами.

1. Короткі теоретичні відомості

Клас `QAbstractButton` – базовий для всіх кнопок. В застосуваннях використовують три типи кнопок: звичайні кнопки (`QPushButton`), прапорці (`QCheckButton`), перемикачі (`QRadioButton`). В класі `QAbstractButton` реалізовані методи і можливості, присутні у всіх класах.

Методт встановлення тексту і зображення.

`setText()` – встановити текст;

`text()` – отримати текст;

`setIconText()` – встановити зображення;

`iconSize()` – отримати максимальний розмір зображення.

`icon()` – отримати зображення.

Методи опитування стану.

`isDown()` – повертає `true`, якщо кнопка у натиснутому стані.

`setDown()` – змінює стан кнопки. Стан кнопки змінює також натискання;

`isCheked()` – повертає `true`, якщо кнопка у ввімкненому стані.

`setChecked()` – змінює стан кнопки. Стан кнопки змінює натискання;

`isEnable()` – повертає `true`, якщо кнопка доступна;

`setEnable()` – змінює доступність кнопки.

Сигнали для взаємодії з користувачем.

`clicked()` – відправляється при клацанні кнопкою миші;

`pressed()` – відправляється при натисканні на кнопку миші;

`released()` – відправляється при відпусканні кнопки миші;

`toggled()` – відправляється при зміні стану кнопки, яка має статус перемикача.

2. Кнопки

Створення кнопки

```
QPushButton* pcmd = new QPushButton("My Button");
```

Приклад рис. 1 демонструє різні види кнопок:

Нормальна кнопка (Normal Button) при відпусканні повертається у своє початкове положення.

Кнопка вимикач (Toggle Button) може знаходитися у двох станах – ввімкнено/вимкнено.

Плоска кнопка (Flat Button) ідентична до нормальної, але відрізняється від неї зовнішнім виглядом. Завдяки тому, що її контури не видимі, її можна використовувати як “секретну кнопку” діалогового вікна.

Кнопка із зображенням (Pixmap Button) є звичайною кнопкою, але містить зображення.

Приклад 1. Різні види кнопок

```
// ================================================================
//  main.cpp
#include <QtGui>
```

```
// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget      wgt;
    // віджет нормальної кнопки
    QPushButton* pcmdNormal = new QPushButton("&Normal Button");

    // віджет кнопки
    QPushButton* pcmdToggle = new QPushButton("&Toggle Button");
    pcmdToggle->setCheckable(true); // встановлення властивості вимикача
    pcmdToggle->setChecked(true);   // встановлення ввімкненого стану

    // віджет кнопки
    QPushButton* pcmdFlat = new QPushButton("&Flat Button");
    pcmdFlat->setFlat(true); // встановлення властивості плоскої кнопки

    // віджет кнопки
    QPixmap pix("aloena.jpg");  // створення об'єкту растрове зображення
    QPushButton* pcmdPix = new QPushButton ("&Pixmap Button");
    pcmdPix->setIcon(pix);  // встановлення зображення у кнопці
    pcmdPix->setIconSize(pix.size()); // встановити розмір растрового зображення

    // Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout; // об'єкт розміщення
    pvbxLayout->addWidget(pcmdNormal);         // добавлення елементів у
розміщення
    pvbxLayout->addWidget(pcmdToggle);
    pvbxLayout->addWidget(pcmdFlat);
    pvbxLayout->addWidget(pcmdPix);
    wgt.setLayout(pvbxLayout);  // передати в основний віджет об'єкт розміщення

    wgt.show();                 // показати основний віджет

    return app.exec();
}
```

```
> qmake -project
> qmake
> make
```

Запуск на виконання:

```
> ./Buttons
```

Рисунок 1 – Різні види кнопок

У нормальних кнопках можна використовувати виринаючи меню. Добавити меню можна викликавши метод `setMenu()` і передавши вказівник на об'єкт виринаюче меню.

Приклад 2. Нормальна кнопка з меню.

```
// ======================================================================
//  main.cpp
// ======================================================================
```

```
#include <QtGui>

// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QPushButton  cmd("Menu");               // віджет кнопка
    QMenu*       pmnu = new QMenu(&cmd);    // об’єкт меню
    pmnu->addAction("Item1");               // елементи меню
    pmnu->addAction("Item2");
    pmnu->addAction("&Quit", &app, SLOT(quit())); // елемент вихід з меню

    cmd.setMenu(pmnu);  // передача у віджет кнопка вказівника на меню
    cmd.show();

    return app.exec();
}
```

```
> qmake -project
> qmake
> make
```

Запуск на виконання:

```
> ./ButtonPopup
```

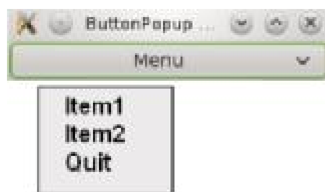

Рисунок 2 – Нормальна кнопка з меню

3. Прапорці

Прапорці використовуються у програмах для встановлення налаштувань, які змінюють поведінку програми. Для створення прапорців використовують клас `QCheckBox`. Звичайні прапорці на два стани (`Normal Check Box`) встановлюються методом `setChecked(true)`. Прапорці на три стани (з третім невизначеним станом, `Tristate Check Box`) встановлюються методом `SetTristate(true)`. Виклик метода `SetCheckState()` з параметром `Qt::PartiallyChecked` встановлює третій стан.

Приклад 3. Прапорці на два і три стани.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>

// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget    wgt;
    // віджет прапорець
    QCheckBox* pchkNormal = new QCheckBox("&Normal Check Box");
    pchkNormal->setChecked(true);  // задання 2-х станів

    // віджет прапорець
    QCheckBox* pchkTristate = new QCheckBox("&Tristate Check Box");
    pchkTristate->setTristate(true);                    // задання 3-х станів
```

```
    pchkTristate->setCheckState(Qt::PartiallyChecked); // невизначений стан

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;    // об'єкт розміщення
    pvbxLayout->addWidget(pchkNormal);            // елементи розміщення
    pvbxLayout->addWidget(pchkTristate);
    wgt.setLayout(pvbxLayout);

    wgt.show();

    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

> **./ButtonPopup**

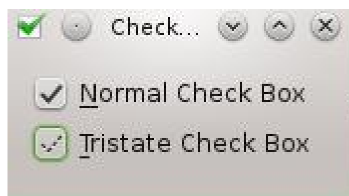

Рисунок 3 – Прапорці на два і три стани

4. Перемикачі (радіокнопки)

Перемикач (radio button) може знаходитися в одному із двох станів: ввімкнено (on) або вимкнено (off). Клас `QRadioButton` віджета перемикача визначений у заголовковому файлі `QRadioButton`. Віджети перемикачів можуть використовуватися тільки у групах. У групі можна ввімкнути тільки один перемикач (реалізується відношення один-до-багатьох). Об'єднуються перемикачі у групу методом `QGroupBox`.

Приклад 4. Перемикачі.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>

// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QGroupBox gbx("&Colors");   // створення віджета групи

    // створення перемикачів і розміщення їх на поверхні віджета групи
    QRadioButton* pradRed   = new QRadioButton("&Red");
    QRadioButton* pradGreen = new QRadioButton("&Green");
    QRadioButton* pradBlue  = new QRadioButton("&Blue");
    pradGreen->setChecked(true); // ввімкнути зелений перемикач

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout; // об'єкт розміщення
    pvbxLayout->addWidget(pradRed);            // елементи розміщення
    pvbxLayout->addWidget(pradGreen);
    pvbxLayout->addWidget(pradBlue);
    gbx.setLayout(pvbxLayout);

    gbx.show();  // показати групу
```

```
    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

> **./RadioPopup**

Рисунок 4 – Перемикачі

5. Групування кнопок

Клас `QGroupBox` використовується для об'єднання у групи різних елементів керування. Він може мати надпис із клавішею швидкого доступу, а також додатковий прапорець, який буде керувати доступністю згрупованих елементів.

Приклад 5. Групування кнопок.

```
// ======================================================================
//  Buttons.h
// ======================================================================
#ifndef _Buttons_h_
#define _Buttons_h_

#include <QGroupBox>

class QCheckBox;
class QRadioButton;

// ======================================================================
class Buttons : public QGroupBox {  // Клас Buttons успадкований від QGroupBox
    Q_OBJECT
private:
    QCheckBox*    m_pchk;       // вказівник типу прапорець
    QRadioButton* m_pradRed;     // вказівники типу перемикач
    QRadioButton* m_pradGreen;
    QRadioButton* m_pradBlue;

public:
    Buttons(QWidget* pwgt = 0);

public slots:
    void slotButtonClicked();
};

#endif  //_Buttons_h_

// ======================================================================
//  Buttons.cpp
// ======================================================================
#include <QtGui>
#include "Buttons.h"

// ----------------------------------------------------------------------
Buttons::Buttons(QWidget* pwgt/*= 0*/) : QGroupBox("QGroupBox", pwgt)
```

```
{
    resize(100, 150);
    setCheckable(true);    // встановлення прапорця
    setChecked(true);      // ввімкнення прапорця

    m_pradRed   = new QRadioButton("&Red");   // створення віджетів
    m_pradGreen = new QRadioButton("&Green");
    m_pradBlue  = new QRadioButton("&Blue");
    m_pradGreen->setChecked(true);    // ввімкнення зеленого перемикача
    // створення з'єднань сигнали - слоти
    connect(m_pradRed, SIGNAL(clicked()), SLOT(slotButtonClicked()));
    connect(m_pradGreen, SIGNAL(clicked()), SLOT(slotButtonClicked()));
    connect(m_pradBlue, SIGNAL(clicked()), SLOT(slotButtonClicked()));

    m_pchk = new QCheckBox("&Light"); // створення прапорця
    m_pchk->setChecked(true);         // встоновити прапорець
    // створення з'єднань сигнал - слот
    connect(m_pchk, SIGNAL(clicked()), SLOT(slotButtonClicked()));

    QPushButton* pcmd = new QPushButton("&Exit");  // створення кнопки вихід
    // створення з'єднань сигнали - слоти
    connect(pcmd, SIGNAL(clicked()), qApp, SLOT(quit()));

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;   // об'єкт розміщення
    pvbxLayout->addWidget(m_pradRed);            // елементи розміщення
    pvbxLayout->addWidget(m_pradGreen);
    pvbxLayout->addWidget(m_pradBlue);
    pvbxLayout->addWidget(m_pchk);
    pvbxLayout->addWidget(pcmd);
    setLayout(pvbxLayout);

    slotButtonClicked();
}

// ----------------------------------------------------------------------
void Buttons::slotButtonClicked()
{
    QPalette pal    = palette();          // створення палітри
    int      nLight = m_pchk->isChecked() ? 150 : 80;
    if(m_pradRed->isChecked()) {
        pal.setColor(backgroundRole(), QColor(Qt::red).light(nLight));
    }
    else if(m_pradGreen->isChecked()) {
        pal.setColor(backgroundRole(), QColor(Qt::green).light(nLight));
    }
    else {
        pal.setColor(backgroundRole(), QColor(Qt::blue).light(nLight));
    }
    setPalette(pal);
}

// ======================================================================
//  main.cpp
// ======================================================================

#include <QApplication>
#include "Buttons.h"

// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);
```

```
    Buttons buttons;
    buttons.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
> ./ButtonGroup
```

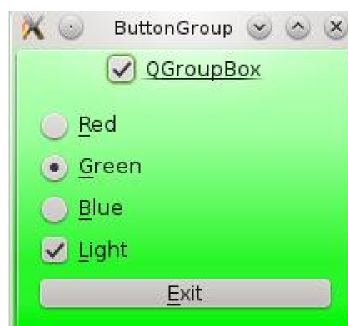

Рисунок 5 – Групування кнопок

6. Елементи налаштувань

Клас `QAbstractSlider` є базовим класом для всіх віджетів налаштувань: повзунок (`QSlider`), смуги прокрутки (`QScrollBar`) і встановлювача (`QDial`).

6.1. Зміна положення

Віджети класів, успадковані від `QAbstractSlider`, можуть бути як горизонтальними, так і вертикальними. Для зміни положення використовується слот `setOrientation` у який передається значення `Qt::Horizontal` або `Qt::Vertical`.

6.2. Встановлення діапазону

Діапазон встановлюється методом `setRange()` з параметрами нижньої і верхньої границі. Також можна використати методи `setMinimum(), setMaximum()`.

```
psld->setRange(1, 10);
psld->setMinimum(1);
psld->setMaximum(10);
```

6.3. Встановлення кроку

Метод `setSingleStep()` встановлює крок зсуву повзунка, `setPageStep()` задає крок для сторінки.

6.4. Встановлення і отримання значень

Значення встановлюється слотом `setValue()`, а отримується методом `value()`.

Сигнал `sliderMoved(int)` передає актуальне значення положення і відправляється при зміні користувачем вказівника поточного положення.

Сигнал `valueChanged()` посилається одночасно з сигналом `sliderMoved(int)` зразу після зміни положення повзунка і також передає змінене значення смуги прокрутки. Поведінка

сигналу змінюється викликом метода `setTracking()`. Якщо передати йому значення `false`, це приведе до того, що сигнал `valueChanged()` буде відправлятися тільки при відпусканні вказівника поточного положення. Щоб взнати, чи відпустив користувач вказівник поточного положення повзунка або все ще утримує його, можна приєднатися до сигналів `sliderPressed()` або `sliderReleased()`.

7. Повзунок

Повзунок задається класом `QSlider`. Клас має метод `setTickPosition()`, який керує розміщенням рисок. Можливі значення, які можна передати у метод:

- `NoTicks` (без рисок);
- `TicksAbove` (риски зверху);
- `TicksBelow` (риски знизу);
- `TicksBothSides` (риски з обох сторін).

Метод `setTickInterval` задає крок рисок.

Приклад 6. Віджет повзунок

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>

// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget  wgt;
    QSlider* psld = new QSlider(Qt::Horizontal); // віджет повзунка
    QLabel*  plbl = new QLabel("3");             // віджет надпис

    psld->setRange(0, 9);  // діапазон значень повзунка
    psld->setPageStep(2);  // крок сторінки
    psld->setValue(3);     // стартове значення повзунка
    psld->setTickInterval(2);  // крок рисок
    psld->setTickPosition(QSlider::TicksBelow);  // розміщення рисок знизу
    QObject::connect(psld, SIGNAL(valueChanged(int)),
                     plbl, SLOT(setNum(int))
                    );  // з'єднання сигналу повзунка зі слотом

    //Layout setup
    QHBoxLayout* phbxLayout = new QHBoxLayout;  // горизонтальне розміщення
    phbxLayout->addWidget(psld);   // добавлення у розміщення повзунка і надпису
    phbxLayout->addWidget(plbl);
    wgt.setLayout(phbxLayout);   // розміщення елементів на поверхні віджета

    wgt.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./Slider
// результат на на рис.1.6
```

8. Смуга прокрутки

Смуги прокрутки дозволяють продивлятися тестові або графічні дані, які за розміром більші від області вікна прогляду. Клас `QScrollBar` реалізує віджет прокрутки і не містить ніяких додаткових методів і сигналів, які розширюють визначення класу `QAbstractSlider`.

Окремо смуги прокрутки використовуються рідко. Вони вбудовані у віджет `QAbstractScrollArea`. Тому для їх використання потрібно використати один із віджетів, успадкованих від `QAbstractScrollArea`.

В об'єктів, успадкованих від класу `QScrollBar`, можна викликати контекстне меню з параметрами навігації за замовчуванням (рис. 1.6).

Приклад 7. Електронний індикатор зі смугою прокрутки.

```
// ==============================================================
//  main.cpp
// ==============================================================
#include <QtGui>

// --------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget     wgt;
    QLCDNumber* plcd = new QLCDNumber(4);  // створення електор. індикатора
    QScrollBar* phsb = new QScrollBar(Qt::Horizontal); // гориз. смуга прокрутки
    // з'єднання сигналу смуги прокрутки зі слотом електронного індикатора
    QObject::connect(phsb, SIGNAL(valueChanged(int)),
                     plcd, SLOT(display(int))
                    );

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout; // вертикальне розміщення
    pvbxLayout->addWidget(plcd);    // добавлення віджетів у розміщення
    pvbxLayout->addWidget(phsb);
    wgt.setLayout(pvbxLayout); // розміщення на основний віджет

    wgt.resize(250, 150); // масштабування основного віджету
    wgt.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./Slider
```

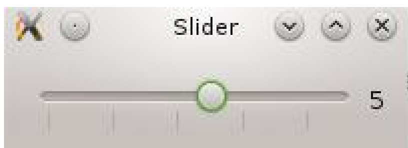

Рисунок 6 – Повзунок

Рисунок 7 – Індикатор і смуга прокрутки

9. Круговий регулятор (встановлювач)

Круговий регулятор визначається у класі `QDial`. За своїми функціональними властивостями він подібний на повзунок. Різниця в тому, що кругла форма віджета дозволяє по досягненню максимального значення перейти до мінімального. Для дозволу або заборони прокручування служить слот `setWrapping()`.

За відображення рисок відповідає метод `setNotchTarget()`, який встановлює їх кількість. Метод `setNothchesVisible()` керує їх видимістю.

Приклад 8. Круговий регулятор та індикатор прогресу.

```
// ==============================================================
//  main.cpp
// ==============================================================
#include <QtGui>

// --------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget       wgt;                        // основний віджет
    QDial*        pdia = new QDial;           // круговий регулятор
    QProgressBar* pprb = new QProgressBar;    // індикатор прогресу
    pdia->setRange(0, 100);  // встановлюється діапазон
    pdia->setNotchTarget(5);            // встановлюється крок рисок 5
    pdia->setNotchesVisible(true);      // риски видимі
    QObject::connect(pdia, SIGNAL(valueChanged(int)),
                     pprb, SLOT(setValue(int))
                    ); // з'єднання сигналу зі слотом

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;  // об'єкт вертикального розміщення
    pvbxLayout->addWidget(pdia);  // добалення віджетів у розміщення
    pvbxLayout->addWidget(pprb);  // зв'язування основного віджету із розміщенням
    wgt.setLayout(pvbxLayout);

    wgt.resize(180, 200);
    wgt.show();
    return app.exec();
}
```

`> qmake -project; qmake; make`

Запуск на виконання:

`>./Dial`

Рисунок 8 – Круговий регулятор та індикатор прогресу

Запитання.

1. Клас `ABstractButton`, методи і сигнали.
2. Клас `QButton`, методи. Створення кнопок різних видів.
3. Клас `QCheckBox`, методи. Створення прапорців на два і три положення.
4. Клас `QRadioButton`, методи. Об'єднання перемикачів у групи.
5. Клас `QGroup`, методи. Групування елементів керування.
6. Клас `QAbstractSlider`, методи.
7. Класи `QSlider, QScrollBar,QDial`, методи.

Завдання.

1. Створити вікно у якому кнопки, прапорці і перемикачі об'єднані у три різні групи. При виборі однієї групи інші групи повинні блокуватися.

2. Створити вікно у якому вибір одного з прапорців – червоного, зеленого, синього змінює колір надпису "Привіт".

3. Створити вікно у якому натискання кнопки послідовно перемикає радіокнопки у групі.

4. Створити вікно у якому при натисканні лівої клавіші миші повзунок рухається вліво, а при натискання правої клавіші миші – рухається вправо.

5. Створити вікно у якому натискання кнопки A збільшує значення кругового індикатора, а натискання кнопки B – зменшує.

Лабораторна робота № 4

4. Автоматичне розміщення елементів

Мета роботи: отримання практичних навиків роботи з автоматичним компонуванням елементів.

1. Короткі теоретичні відомості

Класи для розміщення віджетів є контейнерами, які після зміни розміру вікна автоматично приводять у відповідність розміри і координати віджетів, які знаходяться на ньому. Вони також дозволяють автоматично корегувати розміри віджетів при зміні мови.

1.1. Менеджери компонування (layout manager)

Менеджери мають можливості горизонтального, вертикального і табличного компонування як віджетів, так і вбудованих компонувань.

Базовим для всієї групи менеджерів компонування є клас `Qlayout`. Це абстрактний клас успадкований від двох класів `QObject` і `QLayoutItem`.

Від класу `QLayout` успадковані класи `QGridLayout` і `QBoxLayout` (рис. 1). Клас `QGridLayout` керує табличним розміщенням, а від `QBoxLayout` успадковані два класи `QHBoxLayout` і `QVBoxLayout` для горизонтального і вертикального розміщення.

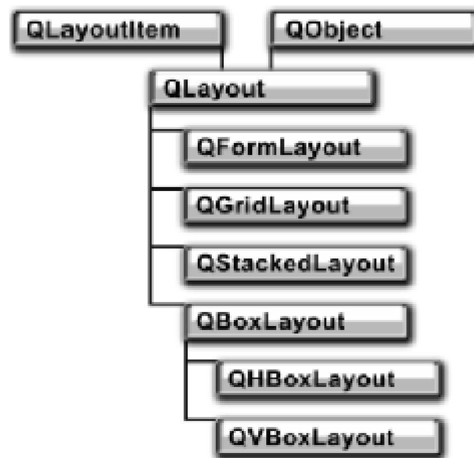

Рисунок 1 – Ієрархія класів менеджерів компонування

За замовчуванням між віджетами залишається невелика віддаль, яка необхідна для їх візуального розділення. Задати її можна за допомогою метода `setSpacing()`, передавши в нього потрібне значення у пікселах. Методом `setMargin()` можна встановити відступ віджетів від границі компонування (звичайно 5-10 пікселов). Рис. 2 показує зміст цих методів на прикладі горизонтального розміщення.

Метод `addWidget()` добавляє віджети у компонування, а метод `addLayout()` добавляє вбудовані менеджери компонування. Вилучити який-небудь віджет із компонування можна методом `removeWidget()`, якому передається вказівник на цей віджет.

Рисунок 2 – Горизонтальне розміщення віджетів

1.2. Горизонтальне і вертикальне розміщення

Для горизонтального або вертикального розміщення використовують класи `QBoxLayout` або успадковані від нього класи `QHBoxLayout` і `QVBoxLayout`.

1.2.1. Клас QBoxLayout

Об’єкт класу `QBoxLayout` може керувати як горизонтальним, так і вертикальним розміщенням. Для того щоб задати спосіб розміщення, першим параметром конструктора має бути одне із наступних значень:

- `LeftToRight` — горизонтальне розміщення із заповненням зліва направо;
- `RightToLeft` — горизонтальне розміщення, із заповненням справа наліво;
- `TopToBottom` — вертикальне розміщення із заповненням зверху вниз;
- `BottomToTop` — вертикальне розміщення із заповненням знизу вверх.

Цей класс розширює клас `QLayout` методами вставлення на задану позицію: віджета – `insertWidget()`, вбудованого компонування – `insertLayout()`, віддалі між віджетами – `insertSpacing()` і фактора розтягування – `insertStretch()`.

У компонування методом `addSpacing()` можна добавити задану віддаль між двома віджетами.

Клас `QBoxLayout` визначає свій власний метод `addWidget()` для добавлення віджетів у компонування з можливістю указання, в додатковому параметрі, фактора розтягування (за замовчуванням нуль). Демонстрація цієї можливості показана в прикладі.

Приклад 1. Кнопки з фактором розтягування.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>

// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QPushButton* pcmdA = new QPushButton("A");  // створення кнопок
    QPushButton* pcmdB = new QPushButton("B");
    QPushButton* pcmdC = new QPushButton("C");

    //Layout setup – компонування зліва направо
    QBoxLayout* pbxLayout = new QBoxLayout(QBoxLayout::LeftToRight);
    pbxLayout->addWidget(pcmdA, 1); // розміщення віджетів з фактором розтягування
    pbxLayout->addWidget(pcmdB, 2);
    pbxLayout->addWidget(pcmdC, 3);
    wgt.setLayout(pbxLayout);   // поміщення компонування в основний віджет

    wgt.resize(450, 40);
    wgt.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
> ./Strach
```

Рисунок 3 – Кнопки з факторами розтягування

Фактор розтягування можна добавити (вставити як пружинку) між двома віджетами методом `addStrech()`.

Приклад 2. Фактор розтягування між двома кнопками.

```
// ==========================================================================
//  main.cpp
// ==========================================================================
#include <QtGui>

// --------------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt; // основний віджет

    QPushButton* pcmdA = new QPushButton("A");  // дві кнопки
    QPushButton* pcmdB = new QPushButton("B");

    //Layout setup - створення компонування
    QBoxLayout* pbxLayout = new QBoxLayout(QBoxLayout::LeftToRight);
    pbxLayout->addWidget(pcmdA); // добавлення кнопок A, B у компонування
    pbxLayout->addStretch(1);    // параметр розтягування між кнопками
    pbxLayout->addWidget(pcmdB);
    wgt.setLayout(pbxLayout);    // добавлення компонування в основний віджет

    wgt.resize(350, 40);
    wgt.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
> ./AddStrach
```

Рисунок 4 – Фактор розтягування між кнопками

1.2.2. Горизонтальне і вертикальне розміщення

Об'єкти класу `QHBoxLayout` упорядковують всі віджети тільки в горизонтальному порядку, зліва направо.

Приклад 3. Горизонтальне розміщення кнопок

```
// ==========================================================================
//  main.cpp
```

```
// ==========================================================================
#include <QtGui>

// --------------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;   // основний віджет

    QPushButton* pcmdA = new QPushButton("A"); // створення кнопок
    QPushButton* pcmdB = new QPushButton("B");
    QPushButton* pcmdC = new QPushButton("C");

    //Layout setup - створення компонування
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->setMargin(10);     // встановлення товщини рамки
    phbxLayout->setSpacing(20);    // встановлення віддалі мід віджетами
    phbxLayout->addWidget(pcmdA); // добавлення кнопок у компонування
    phbxLayout->addWidget(pcmdB);
    phbxLayout->addWidget(pcmdC);
    wgt.setLayout(phbxLayout);  // зв'язування компонування з основним віджетом

    wgt.show();

    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

> **./HBoxLayout**

Рисунок 5 – Розміщення кнопок по горизонталі

Об'єкти класу `QVBoxLayout` упорядковують всі віджети тільки у вертикальному порядку, зверху вниз. В іншому вони нічим не відрізняються від класів `QBoxLayout` і `QHBoxLayout`. Якщо у прикладі 3 замінити `QHBoxLayout` на `QVBoxLayout`, то отримаємо програму, вікно якої показано на рис. 6.

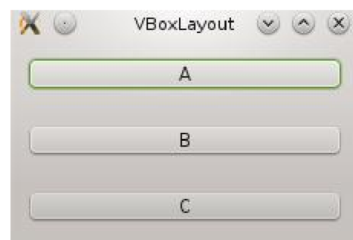

Рисунок 6 – Розміщення кнопок по вертикалі

1.3. Вкладене розміщення

Розміщуючи одне компонування в інше, можна створювати розміщення практично любої складності. Для організації вкладених розміщень існує метод `addLayout()`, у який другим параметром передається фактор розтягування для добавленого компонування. Приклад вкладеного розміщення двох менеджерів компонування показано на рис. 4.7. В компонування `QVBoxLayout` поміщається компонування `QHBoxLayout`.

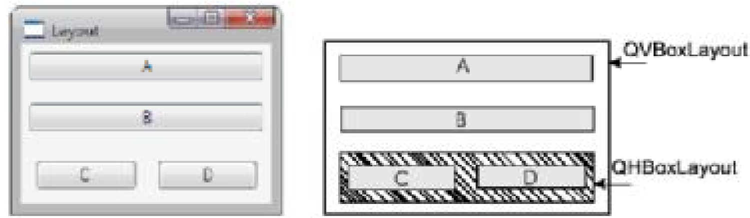

Рисунок 7 – Вкладене розміщення

Приклад 4. Вкладене розміщення компонування

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>

// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
   QApplication app(argc, argv);
   QWidget      wgt;  // основний віджет

   QPushButton* pcmdA = new QPushButton("A"); // створення кнопок
   QPushButton* pcmdB = new QPushButton("B");
   QPushButton* pcmdC = new QPushButton("C");
   QPushButton* pcmdD = new QPushButton("D");

   QVBoxLayout* pvbxLayout = new QVBoxLayout; //створення вертикального
компонування
   QHBoxLayout* phbxLayout = new QHBoxLayout;//створення горизонтального
компонування
   phbxLayout->setMargin(5);   // товщина рамки компонування
   phbxLayout->setSpacing(15); // віддаль між віджетами
   phbxLayout->addWidget(pcmdC); // добавлення віджетів у горизонтальне
компонування
   phbxLayout->addWidget(pcmdD);
   pvbxLayout->setMargin(5);   // товщина рамки компонування
   pvbxLayout->setSpacing(15);   // віддаль між віджетами
   pvbxLayout->addWidget(pcmdA);  // добавлення віджетів у вертикальне
компонування
   pvbxLayout->addWidget(pcmdB);
   pvbxLayout->addLayout(phbxLayout); // добавлення гориз. у вертикальне
розміщення
   wgt.setLayout(pvbxLayout);  // зв'язування верт. компонування з основним
віджетом

   wgt.show();

   return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

> **./HBoxLayout**

1.4. Табличне розміщення

Для табличного розміщення використовується класс `QGridLayout`, який дозволяє швидко створювати складні за структурою розміщення. Таблиця складається з комірок, позиції яких задаються рядками і стовпцями.

Метод `addWidget()` добавляє віджет у таблицю за заданою позицією. Якщо потрібно, щоб віджет займав декілька позицій, потрібно в додаткових параметрах вказати кількість рядків і стовпців які буде займати віджет. В останньому параметрі задається вирівнювання:

```
playout->addWidget(widget, 17, 1, Qt::AlignCenter);
```

Фактор розтягування встановлюється методами `setRowStretch()` і `setColumnStretch()`, але не для кожного віджету зокрема, а для рядка або стовпця. Віддаль між віджетами також встановлюється для стовпців або рядків методом `setSpacing()`.

Приклад 5. Табличне розміщення.

```
#include <QtGui>
int main(int argc, char** argv)
{
  QApplication app(argc,argv);
  QWidget wgt;  // основний віджет

  QPushButton*  pcmdA  =  new  QPushButton("A"); // створення кнопок
  QPushButton*  pcmdB  =  new  QPushButton("B");
  QPushButton*  pcmdC  =  new  QPushButton("C");
  QPushButton*  pcmdD  =  new  QPushButton("D");

  QGridLayout* pgrdLayout = new QGridLayout; .// об'єкт табличного компонування
  pgrdLayout->setMargin(5);      // товщина рамки компонування
  pgrdLayout->setSpacing(15);    // віддаль між віджетами
  pgrdLayout->addWidget(pcmdA, 0, 0); ..// добавлення кнопок у таблицю
  pgrdLayout->addWidget(pcmdB, 0, 1);
  pgrdLayout->addWidget(pcmdC, 1, 0);
  pgrdLayout->addWidget(pcmdD, 1, 1);
  wgt.setLayout(pgrdLayout);  // зв'язування основного віджету з компонуванням

  wgt.show();

  return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
> ./TableLayout
```

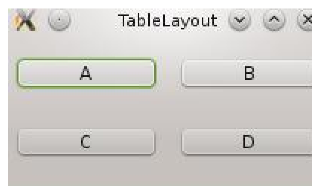

Рисунок 8 – Табличне розміщення

1.4.1. Калькулятор

Табличне розміщення можна продемонструвати на прикладі калькулятора. В програмі використовуються класи стеку `QValueStack` і регулярного виразу `QregExp`.

Приклад 6. Калькулятор.

```
// ======================================================================
//  Calculator.h
// ======================================================================
#ifndef _Calculator_h_
#define _Calculator_h_

#include <QWidget>
#include <QStack>
```

```
class QLCDNumber;
class QPushButton;

// ======================================================================
class Calculator : public QWidget {
    Q_OBJECT
private:
    QLCDNumber*     m_plcd;       // вказівник на віджет електронного індикатора
    QStack<QString> m_stk;        // стек для виконання арифметичних операцій
    QString         m_strDisplay; // стрічка в яку записуються символи натиснутих
                                  // користувачем кнопок

public:
    Calculator(QWidget* pwgt = 0);  // конструктор

    QPushButton* createButton(const QString& str);  // метод для створення кнопок
                                                    // калькулятора
    void         calculate   (); // метод для обчислення виразів зі стеку m_stk

public slots:
    void slotButtonClicked();   // викликається при натисканні на любу кнопку
};

#endif  //_Calculator_h_

// ======================================================================
//  Calculator.cpp
// ======================================================================
#include <QtGui>
#include "Calculator.h"

// ----------------------------------------------------------------------
Calculator::Calculator(QWidget* pwgt/*= 0*/) : QWidget(pwgt) // конструктор
{
    m_plcd = new QLCDNumber(12);  // створення електронного індикатора на 12 цифр
    m_plcd->setSegmentStyle(QLCDNumber::Flat); // плоский стиль індикатора
    m_plcd->setMinimumSize(150, 50);  // встановлення мін. розмірів індикатора

    QChar aButtons[4][4] = {{'7', '8', '9', '/'},  // масив надписів для кнопок
                            {'4', '5', '6', '*'},
                            {'1', '2', '3', '-'},
                            {'0', '.', '=', '+'}
                           };

    //Layout setup
    QGridLayout* ptopLayout = new QGridLayout;  // об'єкт табличного розміщення
    // розміщення ел. індикатора в позицію (0,0), розміри: 1 рядок, 4 стовпців
    ptopLayout->addWidget(m_plcd, 0, 0, 1, 4);
    // створення і добавлення кнопки CE в позицію (1,3), рядок 1, стовпець 3
    ptopLayout->addWidget(createButton("CE"), 1, 3);

    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {   // створення і добавлення кнопок під цифри
           ptopLayout->addWidget(createButton(aButtons[i][j]), i + 2, j);
        }
    }
    setLayout(ptopLayout);
}

// ----------------------------------------------------------------------
// метод отримує стрічку з надрисом і створює натискну кнопку
QPushButton* Calculator::createButton(const QString& str)
{
```

```
    QPushButton* pcmd = new QPushButton(str);
    pcmd->setMinimumSize(40, 40); // встановлення мін розмірів для кнопки
    connect(pcmd, SIGNAL(clicked()), SLOT(slotButtonClicked()));
    return pcmd;
}


// ----------------------------------------------------------------------
// метод обчислює вирази зі стеку m_stk
void Calculator::calculate()
{
    // отримання виразу з верхівки стеку і перетворення його в double
    double  dOperand2    = m_stk.pop().toDouble();
    QString strOperation = m_stk.pop(); // отримання символу операції
    // отримання останнього виразу зі стеку і перетворення його в double
    double  dOperand1    = m_stk.pop().toDouble();
    double  dResult      = 0;

    // отримання результату операцій
    if (strOperation == "+") {
        dResult = dOperand1 + dOperand2;
    }
    if (strOperation == "-") {
        dResult = dOperand1 - dOperand2;
    }
    if (strOperation == "/") {
        dResult = dOperand1 / dOperand2;
    }
    if (strOperation == "*") {
        dResult = dOperand1 * dOperand2;
    }
    m_plcd->display(dResult);  // відображення результату на електр. індикатор
}

// ----------------------------------------------------------------------
/*
У слоті slotButtonClicked() перетворюється віджет, виславший сигнал, до типу QPushButton, після чого змінній str присвоюється текст надпису на кнопці. Якщо надпис дорівнює ”CE”, то виконується операція скидування-очистка стеку і встановлення значення індикатора в 0. Якщо була натиснута цифра або крапка, то вона добавляється в кінець стрічки m_strDisplay, вона відображається індикатором з наступною актуалізацією. При натисканні любої іншої кнопки вважається, що була натиснута кнопка операції. Якщо в стеку знаходиться менше двох елементів, то відображуване число і операція заносяться в стек. Інакше в стек заноситься відображуване значення і викликається метод calculate() для обчислення виразу, який знаходиться у стеку. Після цього стек очищається за допомогою метода clear() і в нього записується значення результату, відображуване індикатором, і наступна операція. Якщо виконується операція “=”, то вона не буде добавлятися у стек.
*/
void Calculator::slotButtonClicked()
{
    QString str = ((QPushButton*)sender())->text();

    if (str == "CE") {
        m_stk.clear();
        m_strDisplay = "";
        m_plcd->display("0");
        return;
    }
    if (str.contains(QRegExp("[0-9]"))) {
        m_strDisplay += str;
        m_plcd->display(m_strDisplay.toDouble());
    }
```

```
    else if (str == ".") {
        m_strDisplay += str;
        m_plcd->display(m_strDisplay);
    }
    else {
        if (m_stk.count() >= 2) {
            m_stk.push(QString().setNum(m_plcd->value()));
            calculate();
            m_stk.clear();
            m_stk.push(QString().setNum(m_plcd->value()));
            if (str != "=") {
                m_stk.push(str);
            }
        }
        else {
            m_stk.push(QString().setNum(m_plcd->value()));
            m_stk.push(str);

            m_strDisplay = "";
            m_plcd->display("0");
        }
    }
}

// ======================================================================
//  main.cpp
// ======================================================================
#include <QApplication>
#include "Calculator.h"

// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    Calculator   calculator;     // створення віджета калькулятор

    calculator.setWindowTitle("Calculator"); // надпис калькулятора
    calculator.resize(230, 200); // зміна розмірів

    calculator.show();

    return app.exec();
}
```

\> **qmake -project; qmake; make**

Запуск на виконання:

\> **./Calculator**

Рисунок 9 – Використання табличного розміщення у калькуляторі

1.5. Порядок слідування табулятора

З віджетами можна взаємодіяти за допомогою миші і клавіатури. В останньому випадку для вибору потрібного віджета використовується клавіша табулятора – <Tab>, при натисканні якої фокус переходить від одного віджета до іншого згідно встановленого порядку. Іноді потрібно змінити цей порядок.

Змінити порядок можна за допомогою статичного метода `QWidget::setTabOrder()`, отримуючого як параметри, два вказівники на віджети:

```
QWidget::setTabOrder(A, B);
QWidget::setTabOrder(B, C);
```

1.6. Розділювачі

Розділювачі дозволяють одночасно продивлятися різні частини текстових або графічних об'єктів. Розділювачі реалізовані у класі `QSplitter`. У деяких випадках розділювачі використовуються замість об'єктів компонування, так як дозволяють змінювати розміри віджетів. За допомогою розділювачів віджети можна розміщувати як вертикально, так і горизонтально. Між віджетами утворюється розділююча лінія, яку можна рухати за допомогою миші, тим самим міняючи розміри віджетів.

Якщо необхідно, щоб віджети розділювача булі інформовані про зміни розмірів, то тоді потрібно викликати метод `setOpaqueResize()`, передавши йому значення `true`.

Приклад 7. Приклад розділення двох віджетів класу `QTextEdit`.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>

// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QSplitter  spl(Qt::Vertical);    // основний віджет вертикальний розділювач
    QTextEdit* ptxt1 = new QTextEdit;  // два об'єкти QTextEdit
    QTextEdit* ptxt2 = new QTextEdit;
    spl.addWidget(ptxt1);   // розміщення об'єктів у розділювач
    spl.addWidget(ptxt2);

    ptxt1->setPlainText("Line1\n"      // встановлення тексту в QTextEdit
                        "Line2\n"
                        "Line3\n"
                        "Line4\n"
                        "Line5\n"
                        "Line6\n"
                        "Line7\n"
                       );
    ptxt2->setPlainText(ptxt1->toPlainText()); // копіювання тексту

    spl.resize(200, 220);
    spl.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
> ./SPlitter
```

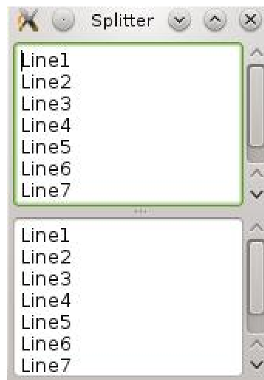

Рисунок 10 – Розділювач

Запитання.

1. Ієрархія класів менеджерів компонування.
2. Клас керування розміщенням `QBoxLayout`, методи. Фактор розтягування.
3. Класи горизонтального і вертикального розміщення `QHBoxLayout` і `QVBoxLayout`, методи.
4. Вкладене розміщення, метод `addLayout()`.
5. Табличне розміщення, клас `QgridLayout` і його методи.
6. Особливості використання табличного розміщення у калькуляторі. Класи, методи, сигнали, слоти.
7. Використання стеку для обчислень виразів. Використовувані класи, методи, сигнали, слоти.
8. Клас розділювачів `QSplitter`.

Завдання.

1. Створити вікно у якому прапорці розмістити горизонтально, а перемикачі вертикально.
2. Створити вікно у якому у горизонтальному розміщенні вкладене вертикальне розміщення кнопок.
3. Створити вікно з табличним розміщення кнопок з порядком слідування табулятора по діагоналях.
4. Створити вікно в якому використовується табличне розміщення для розташування кнопок у шаховому порядку.
5. Створити вікно з вертикальним розділювачем для 4-х кнопок.

Лабораторна робота № 5

5. Елементи введення

Мета роботи: отримання практичних навиків роботи з елементами введення

1. Короткі теоретичні відомості

Група віджетів введення використовується для введення і редагування даних – тексту і чисел. Більша частина елементів введення може працювати з буфером обміну і підтримує технологію (drag & prop).

2. Однорядкове текстове поле

Клас `QLineEdit` реалізує однорядкове поле введення. Текст з віджету повертає метод `text()`. Якщо поле змінилося то відправляється сигнал `textChanged()`. Сигнал `returnPressed()` повідомляє про натиснення клавіші <`Enter`>. Метод `setReadOnly()` з параметром `true` встановлює режим "тільки для читання". Текст для ініціалізації віджету передається в слот `setText()`.

Для однорядкового текстового поля можна ввімкнути режим введення паролю методом `setEchoMode()` з прапором `Password`.

Приклад 1. Два однорядкових текстових поля.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>

// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QLabel* plblDisplay = new QLabel;  // віджет надпису
    plblDisplay->setFrameStyle(QFrame::Box | QFrame::Raised); // стиль рамки
    plblDisplay->setLineWidth(2);      // ширина рамки
    plblDisplay->setFixedHeight(50);   // висота рамки

    QLabel*    plblText = new QLabel("&Text:");  // надпис текст
    QLineEdit* ptxt     = new QLineEdit;         // однорядкове текстове поле
    plblText->setBuddy(ptxt);                    // двійник, для зв'язування полів
    QObject::connect(ptxt, SIGNAL(textChanged(const QString&)),
                     plblDisplay, SLOT(setText(const QString&))
                    );  // відображення вводи мого тексту

    QLabel*    plblPassword = new QLabel("&Password:"); // надпис для паролю
    QLineEdit* ptxtPassword = new QLineEdit;     // однорядкове текстове поле
    plblPassword->setBuddy(ptxtPassword);        // зв'язування полів
    ptxtPassword->setEchoMode(QLineEdit::Password);     // режим введення паролю
    QObject::connect(ptxtPassword, SIGNAL(textChanged(const QString&)),
                     plblDisplay, SLOT(setText(const QString&))
                    );

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout; // вертикальне розміщення
```

```
    pvbxLayout->addWidget(plblDisplay);         // добавлення елементів у розміщення
    pvbxLayout->addWidget(plblText);
    pvbxLayout->addWidget(ptxt);
    pvbxLayout->addWidget(plblPassword);
    pvbxLayout->addWidget(ptxtPassword);
    wgt.setLayout(pvbxLayout);

    wgt.show();

    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

> **./LineEdit**

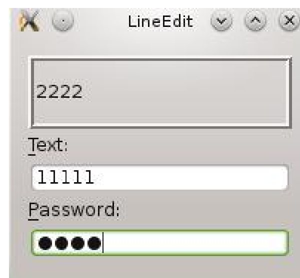

Рисунок 1 – Однорядкові поля введення

Кількість символів введення задається методом `SetMaxLength()`. Метод `maxLegth()` повертає поточне значення максимальної довжини.

Клас `QLineEdit` має наступні слоти:

- `copy()` – копіює виділений текст;
- `cut()` – копіює виділений текст і вилучає його з поля;
- `paste()` – вставляє текст (починаючи з позиції курсору), стираючи виділений текст.

Метод `undo()` відміняє останню операцію, а метод `redo()` повторює останню відмінену. Можливість використання відміни і повторення можна визначити методами `isUndoAviable()` і `isRedoAviable()`.

3. Редактор тексту

Клас QTextEdit дозволяє продивлятися і редагувати, як простий текст, так і текст у форматі HTML. Він успадкований від класу `QAbstracrScrollArea`, що дозволяє автоматично відображати смуги прокручування

Якщо потрібний редактор тільки звичайного тексту то краще скористатися класом `QPlainTextEdit`, який не підтримує формат RTF (Rich Text Format, формат збагаченого тексту) і тому є простішим.

Клас `QTextEdit` має наступні методи:

- `setReadOnly()` – встановлює або знімає режим блокування зміни тексту;
- `text()` – повертає поточний текст.

Слоти:

- `setPlainText()` – встановлює звичайний текст;
- `setHtml()` – встановлює текст у форматі HTML;
- `copy()`, `cut()` і `paste()` – робота з буфером обміну (копіювати, вирізати, вставити);
- `selectAll()` або `deselect()` – виділення або зняття виділення всього тексту;
- `clear()` – очистка поля введення.

Сигнали:

• `textChanged()` – відправляється при зміні тексту;
• `selectionChanged()` – відправляється при змінах виділення тексту.

Для роботи з виділеним текстом служить клас `QTextCursor`, об'єкт якого міститься у класі `QTextEdit`. Клас `QTextCursor` надає методи для виділення тексту, отримання виділеного тексту і його вилучення. Вказівник на об'єкт класу `QTextCursor` можна отримати викликом методу `QTextEdit::textCursor`.

Віджети класу `QTextEdit` також містять у собі об'єкт `QTextDocument`, вказівник на який можна отримати викликом метода `QTextEdit::document()`. Можна також присвоїти другий документ за допомогою метода `QTextEdit::setDocument()`. Клас `QTextDocument` надає слот `undo()` (для відміни) або `redo()` (для повторення дій). При виклику слотів `undo()` і `redo()` посилаються сигнали `undoAvailable(bool)` і `redoAvailable(bool)`, повідомляючі про успішне (або неуспішне) виконання операції. Ці сигнали відправляються як з класу `QtextDocument`, так і з `QTextEdit`. У більшості випадків зручніше використовувати сигнали класу `QTextEdit`.

Більшість методів класу `QTextEdit` є делегуючими для класу `QTextDocument`. Наприклад, клас `QTextEdit` здатний відображати файли з кодом на мові HTML, які містять таблиці і растрові зображення. Для його розміщення і показу можна скористатися методом `setHtml()`, в який передається стрічка, яка містить в собі текст у форматі HTML, або скористатися слотом `insertHtml()`. Ці методи визначені в обох класах і їх виклик з об'єкта класу `QTextEdit` приведе до того, що буде викликаний аналогічний метод з об'єкта класу `QTextDocument`.

Щоб помістити звичайний текст у область віджету можна скористатися методом `setPlainText()` або слотом `insertPlainText()`. За допомогою слота `append()` добавляється текст, причому добавлений текст не вноситься у список операцій, дію яких можна повернути за допомогою слоту `undo()`, що робить цей слот швидким і не вимагаючим додаткових затрат пам'яті. Метод `find()` використовується для пошуку і виділення заданого рядка у тексті.

Приклад 2. Відображення HTML документа.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QTextEdit    txt;

    txt.setHtml("<HTML>"
                "<BODY BGCOLOR=YELLOW>"
                "<CENTER><IMG SRC=\"agnetha.jpg\"></CENTER>"
           "<H2><CENTER>THE WINNER TAKES IT ALL</CENTER></H2>"
           "<H3><CENTER>(Copyrigth (B.Andersson/B.Ulvaeus)</CENTER><H3>"
                "<FONT COLOR=BLUE>"
                "<P ALIGN=\"center\">"
                    "<I>"
                        "I don't wanna talk<BR>"
                        "About the things we've gone through<BR>"
                        "Though it's hurting me<BR>"
                        "Now it's history<BR>"
                        "I've played all ma cards<BR>"
                        "And that's what you've done too<BR>"
                        "Nothing more to say<BR>"
                        "No more ace to play<BR><BR>"
                        "The winner takes it all<BR>"
                        "The loser standing small<BR>"
                        "Beside the victory<BR>"
                        "That's her destiny<BR>"
```

```
                                 "..."
                             "</I>"
                         "</P>"
                         "</FONT>"
                         "</BODY>"
                         "</HTML>"
                        );
        txt.resize(320, 250);
        txt.show();

        return app.exec();
    }
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
> ./TextEdit
```

Рисунок 2 – Вікно програми, яке відображає документ HTML

4. Запис у файл

Клас `QTextDocumentWriter` має три формати для запису вмісту об'єкта класу `QTextDocument` в `PlainText` (Простой текст), HTML і ODF (OpenDocument Format, відкритий формат документів для офісних застосувань, який використовує OpenOffice). Фрагмент коду для запису у форматі ODF.

```
QTextEdit* ptxt = new QTextEdit("This is a <b>TEST</b>"); // віджет редактора
QTextDocumentWriter writer;          // об'єкт запису
writer.setFormat("odf");             // задання формату
writer.setFileName("output.odf");    // вихідний файл
writer.write(ptxt->document());      // запис у файл
```

Запис у форматі PDF класом `QTextDocumentWriter` не підтримується, але це можна зробити шляхом малювання у контексті `QPrinter`.

Приклад 3. Запис документа у форматі PDF.

```
#include <QtGui>
int main(int argc, char *argv)
{
  QTextEdit* ptxt =
   new QTextEdit("<center>This is a <b>TEST</b></center>"); // віджет
редактора
```

```
    QPrinter printer(QPrinter::HighResolution);   // об'єкт прінтера з високою
                                                   // розділювальною здатністю
    printer.setOutputFormat(QPrinter::PdfFormat); // формат виводу
    printer.setOutputFileName("output.pdf");   // ім'я файлу
    ptxt->document()->print(&printer);          // запис у файл
  }
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
> ./PdfTest
```

5. Забарвлення синтаксису

Забарвлення і форматування сприяє більш зручному сприйняттю текста. Для забарвлення синтаксису в `QTextEdit` потрібно успадкувати клас `QSyntaxHighlighter` і реалізувати в успадкованому класі метод `highlightBlock()`. Приклад виділення цифр у тексті червоним кольором:

```
/*virtual*/ void MyHighlighter::highlightBlock(const QString& str)
{
   for (int i = 0; i < str.length(); ++i) {
      if (str.at(i).isNumber()) {
         setFormat(i, 1, Qt::red);
                    <стартове значення>, <к-ть символів>, <колір або шрифт>
      }
   }
}
```

Для того щоб застосувати об'єкт класу забарвлення до об'єкта `QTextEdit`, потрібно при створенні передати йому вказівник на об'єкт `QTextDocument`. Наприклад: `MyHighlighter* pHighlighter = new MyHighlighter(ptxt->document());` Як видно з прикладу, для застосування забарвлення синтаксису потрібний вказівник на об'єкт класу `QTextDocument`, а це значить, що застосувати забарвлення можна не тільки до класу `QTextEdit`, а і до всіх класів, які мають у своєму розпорядженні об'єкт класу `QTextDocument`. До таких класів належать `QTextBrowser`, `QTextFrame`, `QTextTable`, клас елемента тексту `QGraphicsTextItem` графічного подання та інші класи.

Приклад 4. Забарвлення С++ програми у стилі Borland.

```
// ======================================================================
//  SyntaxHighlighter.h
// ======================================================================
#ifndef _SyntaxHighlighter_h_
#define _SyntaxHighlighter_h_

#include <QSyntaxHighlighter>

class QTextDocument;

// ======================================================================
class SyntaxHighlighter: public QSyntaxHighlighter {  // успадкування
Q_OBJECT
private:
    QStringList m_lstKeywords; // список ключових слів
protected:
    // перечислення для поточого стану фрагмента
    enum { NormalState = -1, InsideCStyleComment, InsideCString };
    // перевантаження метода для власної реалізаціїї
    virtual void highlightBlock(const QString&);
    // аналізатор ключових слів
    QString getKeyword(int i, const QString& str) const;

public:
```

```
    SyntaxHighlighter(QTextDocument* parent = 0); // конструктор
};
#endif //_SyntaxHighlighter_h_

// ======================================================================
//  SyntaxHighlighter.cpp
// ======================================================================
#include <QtGui>
#include "SyntaxHighlighter.h"
// ----------------------------------------------------------------------
// конструктор - ініціалізація ключових слів
SyntaxHighlighter::SyntaxHighlighter(QTextDocument* parent/*= 0*/)
    : QSyntaxHighlighter(parent)
{
    m_lstKeywords
       << "foreach"   << "bool"    << "int"    << "void"    << "double"
       << "float"     << "char"    << "delete" << "class"   << "const"
       << "virtual"   << "mutable" << "this"   << "struct"  << "union"
       << "throw"     << "for"     << "if"     << "else"    << "false"
       << "namespace" << "switch"  << "case"   << "public"  << "private"
       << "protected" << "new"     << "return" << "using"   << "true"
       << "->"        << ">>"      << "<<"     << ">"       << "<"
       << "("         << ")"       << "{"      << "}"       << "["
       << "]"         << "+"       << "-"      << "*"       << "/"
       << "="         << "!"       << "."      << ","       << ";"
       << ":"         << "&"       << "emit"   << "connect" << "SIGNAL"
       << "|"         << "SLOT"    << "slots"  << "signals";
}

// ----------------------------------------------------------------------
// аналіз стрічки тексту
/*virtual*/ void SyntaxHighlighter::highlightBlock(const QString& str)
{
    int nState = previousBlockState();
    int nStart = 0;
    for (int i = 0; i < str.length(); ++i) {
        if (nState == InsideCStyleComment) { // позиція всередині коментарів
            if (str.mid(i, 2) == "*/") {
                nState = NormalState;  // використовуються барви палітри
                setFormat(nStart, i - nStart + 2, Qt::darkGray);
                i++;
            }
        }
        else if (nState == InsideCString) {  // позиція всередині стрічки
            if (str.mid(i, 1) == "\"" || str.mid(i, 1) == "\'") {
                if (str.mid(i - 1, 2) != "\\\""
                    && str.mid(i - 1, 2) != "\\\'"
                   ) {
                    nState = NormalState;
                    setFormat(nStart, i - nStart + 1, Qt::cyan);
                }
            }
        }
        else {
            if (str.mid(i, 2) == "//") {  // однорядковий коментар
                setFormat(i, str.length() - i, Qt::darkGray);
                break;
            }
            else if (str.mid(i, 1) == "#") { // директиви препроцесора
                setFormat(i, str.length() - i, Qt::green);
                break;
            }
            else if (str.at(i).isNumber()) {  // цифри
```

```
                setFormat(i, 1, Qt::cyan);
            }
            else if (str.mid(i, 2) == "/*") {
                nStart = i;
                nState = InsideCStyleComment;
            }
            else if (str.mid(i, 1) == "\"" || str.mid(i, 1) == "\'") {
                nStart = i;
                nState = InsideCString;
            }
            else {
                QString strKeyword = getKeyword(i, str);  // ключові слова
                if (!strKeyword.isEmpty()) {
                    setFormat(i, strKeyword.length(), Qt::white);
                    i += strKeyword.length() - 1;
                }
            }
        }
    }
    if (nState == InsideCStyleComment) {   // позиція всередині коментарів
        setFormat(nStart, str.length() - nStart, Qt::darkGray);
    }
    if (nState == InsideCString) { // позиція всередині стрічки
        setFormat(nStart, str.length() - nStart, Qt::cyan);
    }

    setCurrentBlockState(nState);   // встановлення поточного стану
}

// ----------------------------------------------------------------------
// отримання ключових слів
Qstring SyntaxHighlighter::getKeyword(int nPos, const Qstring& str) const
{
    Qstring strTemp = «»;

    foreach (Qstring strKeyword, m_lstKeywords) {
        if (str.mid(nPos, strKeyword.length()) == strKeyword) {
            strTemp = strKeyword;
            break;
        }
    }

    return strTemp;
}

// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
#include «SyntaxHighlighter.h»
// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    Qapplication app(argc, argv);
    QTextEdit    txt;           // віджет редактор тексту

    Qfont fnt(”Lucida Console”, 9, QFont::Normal);   // створення об’єкту шрифт
    txt.document()->setDefaultFont(fnt);    // шрифт за замовчуванням

    new SyntaxHighlighter(txt.document());  // об’єкт забарвлення синтаксису

    Qpalette pal = txt.palette();           // створення палітри
    pal.setColor(Qpalette::Base, Qt::darkBlue); // фон
```

```
    pal.setColor(Qpalette::Text, Qt::yellow);   // текст
    txt.setPalette(pal);   // застосування палітри до редактора

    txt.show();            // показ редактора
    txt.resize(640, 480);

    Qfile file(":/SyntaxHighlighter.cpp");//створення об'єкту файл
    file.open(Qfile::ReadOnly);          // відкриття файлу для читання
    txt.setPlainText(Qlatin1String(file.readAll()));//завантаження тексту у
редактор

    return app.exec();
}
```

> qmake -project; qmake; make

Запуск на виконання:

>./PdfTest

Рисунок 3 – Забарвлення синтаксису

6. Віджети лічильників

Клас абстрактного лічильника `QAbstractSpinBox` надає всім успадкованим від нього класам невелике текстове поле і дві стрілки для зменшення або збільшення числових значень. Від цього класу успадковані наступні класи:

- `QSpinBox` – лічильник;
- `QDoubleSpinBox` – елемент для введення даних з типом `double`.
- `QDataTimeEdit` – елемент для введення дати і часу;

Можна встановити циклічний режим, коли за максимально можливим значенням буде слідувати мінімально можливе, і навпаки. Цей режим встановлюється викликом методу `setWrapping()` з параметром `true`.

Реалізовані два методи покрокової зміни значень `stepUp()` і `stepDown()`, які симулюють натиснення на кнопки стрілок.

За допомогою методу `setSpecialValueText()` встановлюється текст, незалежно від числового значення, наприклад:

```
pspb->setSpecialValueText("default");
```

6.1. Лічильник

Віджет `QSpinBox` забезпечує доступ до обмеженого діапазону чисел. Всі вводимі значення перевіряються, для запобігання виходу за межі встановленого діапазону, який задається методом `setRange()`. Значення встановлюються методом `setValue()`, а отримуються методом `value()`. При зміні значень посилаються зразу два сигнали `valueChanged()`: один з параметром типу `int`, а інший – `const QString&`. Можна змінити спосіб відображення за допомогою методів `setPrefix()` і `setSuffix()`. Наприклад, виклик наступних методів відобразить число у дужках: `pspb->setPrefix("("); pspb->setSuffix(")");` Можна змінити зображення стрілок на символи + (плюс) або − (мінус), передавши методу `setButtonSymbols()` прапор `PlusMinus`.

Приклад 5. Вибір і зміна числа у діапазоні від 1 до 100.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    QSpinBox     spb;         // віджет лічильника

    spb.setRange(1, 100);     // встановлення діапазону
    spb.setSuffix(" MB");     // добавлення тесту після значення
    spb.setButtonSymbols(QSpinBox::PlusMinus); // заміна стрілок знаками ”+”, ”-”.
    spb.setWrapping(true);  // циклічний режим зміни значень
    spb.show();
    spb.resize(50, 30);

    QApplication::setStyle(new QWindowsStyle); //стиль відображення вибраного
режиму
    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./SpinBox
```

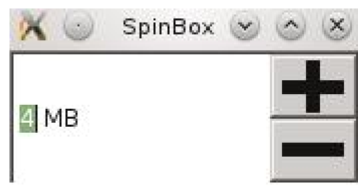

Рисунок 4 – Лічильник

6.2. Елемент введення дати і часу

Віджет `QDateTimeEdit` складається із декількох секцій, призначених для відображення і введення дати і часу. При зміні дати або часу посилається сигнал `dateTimeChanged()`. Цей сигнал передає константне посилання на об’єкт типу `QDateTime`.

Приклад 6. Відображення дати і часу запуску програми.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
```

```
    QApplication  app(argc, argv);
    // виклик поточної дати і часу
    QDateTimeEdit dateTimeEdit(QDateTime::currentDateTime());
    dateTimeEdit.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./DataTimeEdit
```

Рисунок 5 – Дата і час запуску програми

7. Перевірка введення

Об'єкт класу `QValidator` гарантує правильність введення користувача. Для встановлення об'єкта класу `QValidator` необхідно передати його в метод `setValidator()`, який міститься у класах `QComboBox` і `QLineEdit`. Для перевірки введення чисел використовуються класи `QIntValidator` і `QDoubleValidator`. Створюючи свій клас перевірки введення, потрібно успадкувати клас від `QValidator` і перезаписати метод `validate()`, в який передається вводима стрічка і позиція курсора. Метод має повертати наступні значення:

- `QValidator::Invalid` – якщо рядок не може бути прийнятий;
- `QValidator::Intermediate` – рядок не може бути прийнятий як кінцевий результат. Наприклад, якщо рядок має задавати числове значення від 50 до 100, то введене число 1 виходить за межі допустимих значень;
- `QValidator::Acceptable` – якщо рядок може бути прийнятий без змін.

Приклад 7. Поле введення любих символів крім цифр.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ======================================================================
class NameValidator : public QValidator {  // успадкування
public:
    NameValidator(QObject* parent) : QValidator(parent)
    {
    }

    virtual State validate(QString& str, int&) const  // перевизначення методу
    {
        QRegExp rxp = QRegExp("[0-9]");  // об'єкт регулярного виразу
        if (str.contains(rxp)) {         // перевірка наявності цифр
            return Invalid;              // введене не прийнято
        }
        return Acceptable;   // введене прийнято
    }
};


// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;
```

```
    // віджет надпису
    QLabel* plblText =
           new QLabel("&Name (The digits will not be accepted!):");
    // відмет однорядкового текстового поля
    QLineEdit* ptxt = new QLineEdit;
    // об'єкт контролер
    NameValidator* pnameValidator = new NameValidator(ptxt);
    // контролер зв'язується з текстовим полем
    ptxt->setValidator(pnameValidator);
    // тестове поле зв'язується з надписом
    plblText->setBuddy(ptxt);

    //Layout setup - об'єкт вертикального розміщення
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(plblText);
    pvbxLayout->addWidget(ptxt);
    wgt.setLayout(pvbxLayout);  // зв'язування основного віджету з розміщенням
    wgt.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./Validator
```

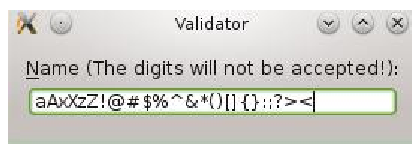

Рисунок 6 – Поле з контролем типу даних

Запитання.

1. Клас `QLineEdit`. Методи, сигнали і слоти.
2. Клас `QtextEdit`, `QplainTextEdit`, `QtextCursor`, `QTextDocument`. Методи, сигнали і слоти.
3. Клас `QtextDocumentWriter`. Методи, сигнали і слоти. Запис документа у форматі PDF.
4. Використання класів `QtextEdit`, `QSyntaxHighlighter`, `QFont` для забарвлення синтаксису.
5. Клас абстрактного лічильника `QabstractSpinBox`. Лічильник `QspinBox`, `QdoubleSpinBox`. Клас `QdataTimeEdit`. Методи, сигнали і слоти.
6. Клас QValidator. Методи, сигнали і слоти

Завдання.

1. Написати віджет з двома однорядковими полями (основним і контрольним) для введення паролю. Введене значення паролю перевірити із значенням, яке зберігається у файлі. При співпадінні паролів вивести в однорядкове поле віджету значення паролю, а при неспівпадінні – повідомлення ”Невірний пароль”.
2. Добавити у віджет класу `QTextEdit` метод запису відредагованого тексту у файл у форматі PDF.
3. Написати відмет з трьома однорядковими полями введення даних. У перше поле мають вводитися тільки цілочисельні значення, у друге – числа з плаваючою крапкою, у третє – тільки символи алфавіту.

Лабораторна робота № 6

6. Елементи вибору

Мета роботи: отримання практичних навиків роботи з елементами вибору.

1. Короткі теоретичні відомості

Елементи вибору це стандартні елементи графічного інтерфейсу користувача призначені для відображення, модифікації і вибору даних.

2. Простий список

Клас `QListWidget` – це віджет списку, який дозволяє вибрати один або декілька елементів. Елементи списку можуть містити текст і растрові зображення. Добавити елемент у список можна методом `addItem()`. У цьому методі реалізовано два його варіанти: для тексту і для об'єкта класу `QListWidgetItem`. Слот `clear()` вилучає всі елементи із списку. Клас `QListWidgetItem` – це клас для елементів списку. Об'єкти цих класів можуть створюватися неявно, наприклад, при передачі тексту в метод `QListWidget::addItem()`. Варт зауважити, що клас `QListWidgetItem` має конструктор копіювання, що дозволяє створювати копії елементів. Також для цього можна скористатися методом `clone()`.

2.1. Вставлення елементів

В список можна добавити зразу декілька текстових елементів, передавши об'єкт класу `QStringList`, який містить список стрічок, у метод `insertItems()`. Для створення текстового елемента можна скористатися методом `insertItem()`, в який потрібно передати тільки рядок тексту. Об'єкт `QListWidgetItem` також можна вставити у список за допомогою метода `insertItem()`. Відмінність його від метода `addItem()` в тому, що можна явно вказати позицію добавленого елемента. Створеному елементу можна присвоїти растрове зображення методом `QListWidgetItem::setIcon()` об'єкта елемента списку.

В елементах списку можна також встановлювати не тільки растрові зображення і текст, але і віджети. Для цього у класі `QListWidget` визначені методи `setItemWidget()` і `itemWidget()`. Першим параметром методу `setItemWidget()` потрібно передати вказівник на об'єкт елемента списку, а другим – вказівник на віджет. Для того щоб отримати вказівник на віджет, розміщений в елементі списку, необхідно передати в метод `itemWidget()` вказівник на об'єкт елемента списку.

Приклад 1. Приклад простого списку операційних систем.

```
// ======================================================================
//  main.cpp
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication     app(argc, argv);
    QStringList      lst;
    QListWidget      lwg;
    QListWidgetItem* pitem = 0;

    lwg.setIconSize(QSize(48, 48));  // встановлення розміру растрових зображень
    lst << "Linux" << "Windows" << "MacOS" << "OS2";
    foreach(QString str, lst) {      // заповнення списку зображеннями
        pitem = new QListWidgetItem(str, &lwg);
```

```
        pitem->setIcon(QPixmap(str + ".jpg"));
    }
    lwg.resize(125, 175);
    lwg.show();

    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

>**./ListWidget**

Рисунок 1 – Простий список

2.2. Вибір елементів користувачем

Взнати, який елемент вибрав користувач, можна за допомогою метода `QListWidget::currentItem()`, який повертає вказівник на вибраний елемент. Якщо вибраних елементів декілька, то потрібно викликати метод `selectedItems()`, який верне список вибраних елементів. Щоб дозволити режим множинного виділення, необхідно встановити методом `setSelectionMode()`, реалізованим у базовому класі `QAbstractItemView`, значення `QAbstractItemView::MultiSelection`. У цей метод можна передавати і інші значення, наприклад, для того щоб заборонити виділення зовсім, у нього потрібно передати `QAbstractItemView::NoSelection`, а для можливості виділення тільки одного з елементів – `QAbstractItem::SingleSelection`. Після клацання на елементі списку відправляється сигнал `itemClicked()`. При подвійному клацанні миші на елементі відправляється сигнал `itemDoubleClicked()` з параметром `QListWidgetItem*`. Після кожної зміни виділення елементів відсилається сигнал `itemSelectionChanged()`.

2.3. Зміна елементів користувачем

Щоб надати можливість користувачу міняти текст елемента, необхідно викликати з потрібного об'єкта елемента метод `QListWidgetItem::setFlags()` і передати в нього значення `Qt::ItemIsEditable` і інші потрібні значення. Наприклад: `pitem->setFlags(Qt::ItemIsEditable | Qt::ItemIsEnabled);` Перейменування здійснюється подвійним клацанням миші на елементі списку або натисканням клавіші <F2>. Після перейменування віджет `QListWidget` відправляє сигнали `itemChanged(QListWidgetItem*)` і `itemRenamed(QListWidgetItem*)`.

2.4. Режим піктограм

Віджет списку можна перевести у режим піктограм (режим подання у вигляді значків або піктограм), який дозволяє вибирати елементи, здійснювати над ними операції перетягування і відпускання (drag & drop).

Приклад 2. Приклад піктограм.

```
// ======================================================================
//  main.cpp
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication     app(argc, argv);
    QListWidget      lwg;   // віджет списка
    QListWidgetItem* pitem = 0;
    QStringList      lst;

    lwg.setIconSize(QSize(48, 48));
    // вибір декількох елементів
    lwg.setSelectionMode(QAbstractItemView::MultiSelection);
    lwg.setViewMode(QListView::IconMode);   // режим піктограм
    lst << "Linux" << "Windows" << "MacOS" << "OS2"; // заповнення списку іменами
    foreach(QString str, lst) {  // створення елементів списку
        pitem = new QListWidgetItem(str, &lwg);
        pitem->setIcon(QPixmap(str + ".jpg"));
        // встановлення властивостей елементів списку
        pitem->setFlags(Qt::ItemIsEnabled | Qt::ItemIsSelectable |
                        Qt::ItemIsEditable | Qt::ItemIsDragEnabled);

    }
    lwg.resize(135, 135);
    lwg.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./IconMode
```

Рисунок 2 – Режим піктограм

2.5. Сортування елементів

Элементи списку можна впорядкувати викликом метода `sortItems()`. При передачі в цей метод значення `Qt::AscendingOrder` елементи будуть відсортовані у зростаючому порядку, а при `Qt::DescendingOrder` – в спадаючому. Якщо виконати сортування, а потім добавляти нові елементи, вони не будуть сортуватися. Сортування виконується в алфавітному порядку, а якщо потрібно відсортувати по даті або по числовому значенню то необхідно успадкувати клас елемента `QListWidgetItem` і перезаписати у ньому `operator<()`.

3. Ієрархічні списки

Віджет `QTreeWidget` відображає елементи списку в ієрархічній формі і підтримує можливості вибору користувачем одного або декількох з них. Його часто застосовують для

показу вмісту дисків і каталогів. Якщо область відображення не в змозі розмістити всі елементи, то появляються смуги прокрутки. За допомогою метода `setItemWidget()` в ієрархічному списку можна розмістити віджети.

Елементи списку є об'єктами класу `QTreeWidgetItem` і надають можливість відображати декілька стовпців з даними. Клас `QTreeWidgetItem` містить конструктор копіювання і метод `clone()` для створення копій елементів. За допомогою методів `addChildren()` і `insertChildren()` можна добавляти і вставляти зразу декілька елементів. Якщо необхідно добавити до елементів невеликі растрові зображення, то вони встановлюються за допомогою метода `QTreeWidgetItem::setIcon()`, а текст елемента – за допомогою `QTreeWidgetItem::setText()`. Перший параметр обох методів відповідає номеру стовпця.

Приклад 3. Список з вкладеними елементами каталогами.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QTreeWidget  twg;  // віджет ієрархічного списку
    QStringList  lst;  // віджет список

    lst << "Folders" << "Used Space"; // назви елементів списку
    twg.setHeaderLabels(lst);         // заголовки
    twg.setSortingEnabled(true);

    // зв'язування першого елемента списку з відметом ієрархічного списку
    QTreeWidgetItem* ptwgItem = new QTreeWidgetItem(&twg); // віджет
    ptwgItem->setText(0, "Local Disk(C)");
    ptwgItem->setIcon(0, QPixmap("drive.bmp"));

    // створення 20 тотомків для верхівки списку
    QTreeWidgetItem* ptwgItemDir = 0;
    for (int i = 1; i < 20; ++i) {
        ptwgItemDir = new QTreeWidgetItem(ptwgItem);
        // текст стовпців
        ptwgItemDir->setText(0, "Directory" + QString::number(i));
        ptwgItemDir->setText(1, QString::number(i) + "MB");
        // добавлення до першого стовпця растрового зображення
        ptwgItemDir->setIcon(0, QPixmap("folder.bmp"));
    }
    twg.setItemExpanded(ptwgItem, true); // розгортає елемент верхівки
    twg.resize(250, 110);
    twg.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./TreeWidget
```

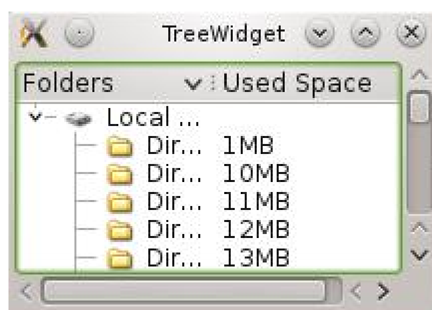

Рисунок 3 – Ієрархічний список

При натисканні на заголовок стовпця його елементи сортуються методом setSortingEnabled(arg) у спадаючому (true) або зростаючому порядку (false).

Метод setSelectionMode() базового класу QAbstractItemView з параметром QAbstractItemView::MultiSelection встановлює режим множинного виділення. Для перебору всіх елементів віджета ієрархічного списку QtreeWidget, використовується ітератор, наприклад:

```
QTreeWidgetItemIterator it(&twg, QTreeWidgetItemIterator::All);
while (*(++it)) { qDebug() << (*it)->text(0); }
```

Для перебору тільки виділених елементів у конструктор потрібно передати значення QTreeWidgetItemIterator::Selected.

Клас QTreeWidget містить наступні сигнали:

- itemSelectionChanged() повідомляє про зміни вибраних елементів;
- itemClicked(QTreeWidgetItem*, int) відправляється після клацанні на елементі;
- itemDoubleClicked(QTreeWidgetItem*, int) відправляється при подвійному клацанні миші;
- itemActivated(QTreeWidgetItem*, int) відправляється при подвійному клацанні миші, а також при натисканні клавіші <Enter> на елементі.

Другий параметр останніх трьох сигналів містить номер стовпця, на якому відбулося клацання.

Класс QTreeWidget підтримує технологію перетягування (drag & drop). Для її реалізації необхідно викликати метод setFlags() для тех елементів, для яких потрібно включити підтримку перетягування з параметром Qt::ItemIsDragEnabled, можливо об'єднану з іншими значеннями. Наприклад: pitem->setFlags(Qt::ItemIsDragEnabled | Qt::ItemIsEnabled);

4. Таблиці

Клас QTableWidget реалізує таблицю. Об'єкт комірки реалізований у класі QTableWidgetItem. Створену комірку можна вставити у таблицю викликом метода QTableWidget::setItem(). Першим параметром метода setItem() є номер рядка, а другим – номер стовпця. Таким чином, ці параметри задають місцезнаходження комірки у таблиці. Встановити текст у самій комірці можна методом QTableWidgetItem::setText(), а для розміщення растрового зображення можна скористатися методом QTableWidgetItem::setIcon(). Якщо у комірці встановлені як текст, так і растрове зображення, то растрове зображення займе місце зліва від тексту. Клас комірки QTableWidgetItem подає конструктор копіювання, що дозволяє створювати копії елементів. Також для цього можна скористатися методом clone().

В таблицю допускається, крім тексту і кнопок, поміщати і віджети. Для цього використовується метод setCellWidget().

При подвійному клацанні кнопкою миші на полі комірки вона переходить у режим редагування, при цьому використовується віджет QLineEdit.

Приклад 4. Таблиця.

```
// ==========================================================
//  main.cpp
// ==========================================================
#include <QtGui>
// ----------------------------------------------------------
int main(int argc, char** argv)
{
    const int n = 3;
    QApplication      app(argc, argv);
    QTableWidget      tbl(n, n);  // віджет таблиці розміром n x n
```

```
    QTableWidgetItem* ptwi = 0;   // вказівник на об'єкт комірку
    QStringList       lst;

    lst << "First" << "Second" << "Third";  // список заголовків
    tbl.setHorizontalHeaderLabels(lst);     // заголовки стовпців
    tbl.setVerticalHeaderLabels(lst);       // заголовки рядків

    for (int i  = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) { // створення і заповнення об'єктів комірок
            ptwi = new QTableWidgetItem(QString("%1,%2").arg(i).arg(j));
            tbl.setItem(i, j, ptwi);
        }
    }
    tbl.resize(350, 125);
    tbl.show();
    return app.exec();
}
```

> qmake -project; qmake; make

Запуск на виконання:

>./TableWidget

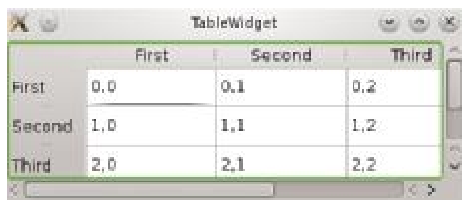

Рисунок 4 – Таблиця

5. Випадаючий список

Клас QComboBox надає можливість вибору одного елемента з декількох. Його функціональне призначення співпадає з віджетом простого списку QListWidget. Основна перевага випадаючого списку у можливості відображення тільки одного (вибраного) елементу, завдяки чому для його розміщення не вимагається багато місця. Відображення всього списку (розкриття) відбувається тільки на деякий проміжок часу, щоб користувач міг зробити вибір, а потім список повертається у свій початковий стан (згортається).

Як елемент можна добавити текст і/або картинку. Для цього служить метод addItem(). Можна добавити зразу декілька текстових елементів, передавши вказівник на об'єкт класу QStringList в метод addItems(). Викликом методу setDuplicatesEnabled(false), можна включити режим, запобігаючий повторення елементів зі списку. Викликом слоту clear() можна вилучити всі елементи випадаючого списку.

Метод currentIndex() повертає порядковий номер поточного елементу. Щоб дозволити користувачу самому добавляти елементи у список, необхідно викликати метод setEditable(true). Після зміни користувачем тексту вибраного елемента, відправляється сигнал editTextChanged(const QString&), і новий елемент добавляється у список.

Після вибору елемента відправляються зразу два сигнали activated() – один з параметром типу int (індекс вибраного елемента), а інший з параметром типу const QString& (його значенням). Ці сигнали відправляються, навіть якщо користувач вибрав раніше вибраний елемент; інформують про реальні зміни два сигнали currentIndexChanged(), які відправляються з параметрами int і const QString& кожний.

Приклад 5. Випадаючий список.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
```

```
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QComboBox    cbo;  // віджет випадаючого списку
    QStringList  lst;  // об’єкт список
    // елементи списку
    lst << "John" << "Paul" << "George" << "Ringo";
    cbo.addItems(lst);  ..// добавлення списку у випадаючий список
    cbo.setEditable(true); // режим редагування списку
    cbo.show();

    return app.exec();
}
```

> qmake -project; qmake; make

Запуск на виконання:

>./ComboBox

Рисунок 5 – Випадаючий список

6. Вкладки

Основне призначення вкладок – розвантажити складне діалогове вікно, яке має велику кількість опцій, розділивши його на серію логічно скомпонованих діалогових підвікон. Вкладки можуть містити як текст, так і растрове зображення.

Сторінки можна роботи доступними і недоступними. Щоб зробити сторінку недоступною, потрібно викликати метод `setTabEnabled()` і передати йому значення `false`. Викликом слота `setCurrentIndex()` можна зробити вкладку поточною.

Приклад 6. Вікно з чотирма вкладками.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QTabWidget   tab;    // створення віджету вкладок
    QStringList  lst;

    lst << "Linux" << "Windows" << "MacOS" << "OS2"; // список назв
    foreach(QString str, lst) {  // добавлення вкладок у циклі
// параметри: вказівник на відмет, який відображається при виборі вкладки
//            растрове зображення,
              текст вкладки
        tab.addTab(new QLabel(str, &tab), QPixmap(str + ".jpg"), str);
    }

    tab.resize(300, 100);
    tab.show();

    return app.exec();
}
```

> qmake -project; qmake; make

Запуск на виконання:

>./TabWidget

Рисунок 6 – Вкладки

7. Панелі інструментів

Клас `QToolBox` реалізовує вкладки, розміщені вертикально. Зв'язані (з вкладками) віджети відображаються безпосередньо під ними. Текст вкладки добавляється разом з віджетом при виклику метода `addItem()`. Якщо потрібно вставити вкладку на певну позицію, то викликається метод `insertItem()`. Кількість вкладок можна взнати викликом методу `count()`. Вкладки вилучаються методом `removeItem()`.

Викликом методу `currentWidget()`, можна отримати вказівник на закріплений за поточною вкладкою віджет.

Віджет панелі інструментів має тільки один сигнал `currentChanged(int)`, який відсилається при виборі однієї з вкладок.

Приклад 7. Панель інструментів з чотирма вкладками і закріпленими за ними віджетами надписів.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QToolBox     tbx;   // створення панелі інструментів
    QStringList  lst;

    lst << "Linux" << "Windows" << "MacOS" << "OS2"; // список назв
    foreach(QString str, lst) {  // добавлення вкладок у циклі
        tbx.addItem(new QLabel(str, &tbx), QPixmap(str + ".jpg"), str);
    }
    tbx.resize(100, 80);
    tbx.show();

    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

>**./ToolBox**

Рисунок 7 – Панель інструментів

Запитання.

1. Клас `QListWidget`. Вставлення елементів у список.
2. Клас `QListWidget`. Вибір і зміна елементів користувачем.
3. Клас `QListWidget`. Режим піктограм. Сортування елементів.
4. Клас `QTreeWidget`. Методи і сигнали для роботи з ієрархічними списками..
5. Клас `QTableWidget`. Методи для роботи з таблицями.
6. Клас `QComboBox`. Методи для роботи з випадаючим списком.
7. Класи `QTabWidget, QToolWidget`. Методи відображення вкладок і панелі інструментів.

Завдання.

1. Добавити у приклад 1 можливість множинного виділення елементів простого списку.
2. Добавити у приклад 3 можливість множинного виділення елементів ієрархічного списку.
3. У прикладі 4 відобразити у таблиці растрові зображення.
4. Добавити у випадаючий список можливість відображення тексту і растрового зображення.
5. Добавити у вкладки можливість роботи сторінки доступними і недоступними.

Лабораторна робота № 7

7. Архітектура “модель-подання”

Мета роботи: отримання практичних навиків розроблення архітектури “модель-подання”

1. Короткі теоретичні відомості

Архітектура “модель-подання” дозволяє відділити дані від їх подання. Використання цієї технології дає наступні переваги:

- можливість показати дані у різному поданні без дублювання;
- можливість внесення змін з мінімальними часовими затратами;
- зручність редагування програмного коду;
- спрощення інтеграції баз даних.

Вказана архітектура реалізована в Qt як технологія “модель-подання”.

2. Структура технології “модель-подання”

Взаємодія компонентів технології “модель-подання” показана на рис. 1.

Рисунок 1 – Взаємодія компонентів технології “модель-подання”

Компоненти технології виконують наступні функції:

• *модель* – відповідає за керування даними і надає інтерфейс для читання і запису даних;

• *подання* – відповідає за подання даних користувачу і за їх розміщення;

• *виділення елемента* – спеціальна модель, яка відповідає за централізоване використання виділених елементів;

• *делегат* – відповідає за окреме малювання кожного елементу, а також за його редагування.

2.1. Модель

Модель – це оболонка навколо початкових даних, яка надає стандартний інтерфейс для доступу до них. Інтерфейс моделі є основною одиницею, яка забезпечує зв’язок між моделлю і поданням, що дозволяє розробляти моделі незалежно одна від одної і при необхідності замінити одну на іншу. Інтерфейс любої Qt-моделі базується на класі `QAbstractItemModel` (рис. 2.). Для створення власної моделі потрібно успадкувати цей клас або один із його нащадків.

Сам клас `QAbstractItemModel` є узагальненою таблицею, за кожною коміркою якої можна закріпити підтаблицю. Завдяки такій властивості можна створювати моделі для складних структур даних. Наприклад, для деревовидної структури, яка описує вміст каталогу, деяка комірка рядка, що подає каталог, буде мати підтаблицю, рядки якої будуть відповідати файлам і підкаталогам. Підкаталоги, в свою чергу, можуть також мати підтаблицю з рядками, в яких будуть появлятися файли і підкаталоги.

Деякі класи, успадковані від `QabstractItemModel`, показані на рис.2.

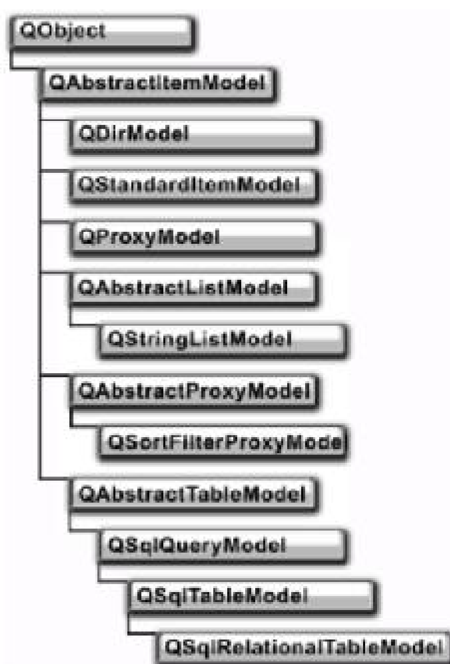

Рисунок 2 – Ієрархія класів моделі

Клас `QStandardItemModel` дозволяє напряму зберігати дані у моделі. Використовується у випадку невеликих обсягів даних.

Клас `QAbstractListModel` реалізовує одновимірний список, а клас `QAbstractTableModel` – двовимірну таблицю.

Клас `QStringListModel` – це реалізація `QAbstractListModel`, яка подає одновимірну модель, призначену для работи зі списком стрічок. Джерело даних – це список стрічок (`QStringList`). Ця модель надає можливість редагування, тобто, якщо користувач за допомогою подання змінить один із записів, то старий запис буде заміщений новим. Кожний запис відповідає одному рядку.

```
QStringListModel model;
model.setStringList(QStringList() << "Перший рядок"
                                  << "Другий рядок"
                                  << "Третій рядок" );
```

Основна ідея класу `QAbstractProxyModel` полягає у видобуванні даних з моделі, здійснення з ними деяких маніпуляцій і повернення їх як нової моделі. Таким чином можна виконувати вибірку і сортування даних. Для виконання цих операцій можна скористатися успадкованим класом `QSortFilterProxyModel`.

Клас `QDirModel` є готовим класом ієрархії файлової системи.

Дані, надані моделями, можуть сумісно використовуватися, через інтерфейс, різними поданнями (віджетами, успадкованими від `QAbstarctItemView`), рис. 3. Для того щоб модель і подання могли розуміти один одного, модель інформована про основні властивості подання: кожний запис займає в ній один рядок і стовпчик, а також може мати індекс, який відграє важливу роль у вкладених структурах.

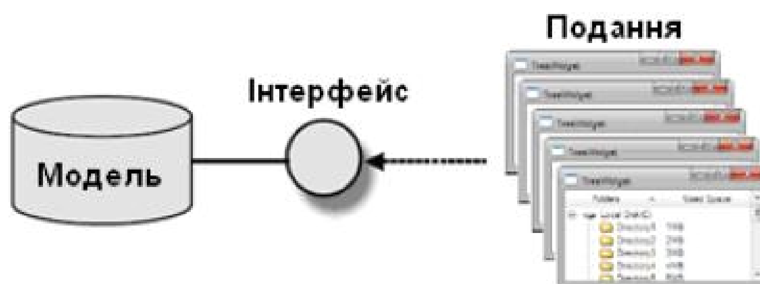

Рисунок 3 – Зв'язок моделі з різними поданнями

2.2. Подання

Базовим класом більшості подань є `QAbstactScrollArea`, що дозволяє, у тих випадках, коли відображувана інформація займає більше місця, ніж область вікна, скористатися смугами прокрутки, рис.4.

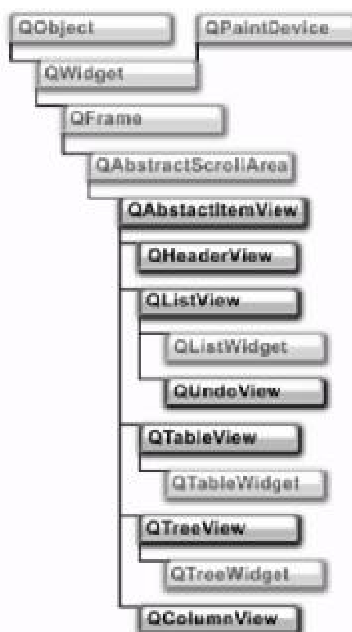

Рисунок 4 – Класи подань

Поданням може бути і клас `QComboBox`, який напряму успадкований від класу `QWidget`. Клас `QComboBox` надає метод `setModel()` для встановлення моделей, як і всі далі описані класи подань.

Ці класи подань успадковуються від класу `QAbstractItemView`, який дає такі базові можливості для всіх подань, як, наприклад, встановлення моделей у поданні, методи для прокрутки зображення та інші. Цей клас містить метод `setEditTriggers()`, який задає параметри перейменування елементів.

В цей метод можна передати наступні значення:

- `NoEditTriggers` – перейменувати неможливо;
- `DoubleClicked` – перейменувати при подвійному клацанні на елементі мишею;
- `SelectedClicked` – перейменувати при одному клацанні на елементі мишею.

Для подання даних в Qt використовуються, в основному, три класи:

• `QListView` є одновимірний список. Цей клас також має режим піктограм (відображення значків);

• `QTreeView` відображає ієрархічні списки. Цей клас також може відображати стовпці;

• `QTableView` відображає дані у таблиці. Клас `QHeaderView` успадкований безпосередньо від `QAbstractItemView`, але призначенйи для самостійного відображення даних, а використовується сумісно з класами `QTableView` і `QTreeView` для відображення заголовків стовпців і рядків.

2.3. Виділення елементу

Керування виділенням елементів здійснюється за допомогою спеціальної моделі, реалізованої у класі `QitemSelectionModel`, рис. 5.

Рисунок 5 – Клас виділення `QItemSelectionModel`

Для отримання моделі виділення елементів, встановленої у поданні, потрібно викликати метод `QAbstractItemView::selectionModel()`, а встановити нову модель можна викликом методу `QAbstractItemView::setSelectionModel()`.

Індекси поточних виділених позицій можна отримати викликом методу `QItemSelectionModel::selectedIndexes()`. Виділяти елементи програмно можна за допомогою методу `QItemSelectionModel::select()`. При змінах виділення модель відсилає сигнали `currentChanged()`, `selectionChanged()`, `currentColumnChanged()` і `currentRowChanged()`.

Приклад 1. Розділення виділення елементу між трьома поданнями.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QStringListModel model;  // створення об’єкта списку і його ініціалізація
    model.setStringList(QStringList() << "Item1" << "Item2" << "Item3");

    QTreeView* pTreeView = new QTreeView;     // подання 1
    pTreeView->setModel(&model);

    QListView* pListView = new QListView;     // подання 2
    pListView->setModel(&model);

    QTableView* pTableView = new QTableView;  // подання 3
    pTableView->setModel(&model);

    QItemSelectionModel selection(&model); // модель виділення і її ініціалізація
// модель виділення встановлюється у всі три подання
    pTreeView->setSelectionModel(&selection);
    pListView->setSelectionModel(&selection);
    pTableView->setSelectionModel(&selection);

    //Layout setup
```

```
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->addWidget(pTreeView);
    phbxLayout->addWidget(pListView);
    phbxLayout->addWidget(pTableView);
    wgt.setLayout(phbxLayout);

    wgt.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./SelectionSharing
```

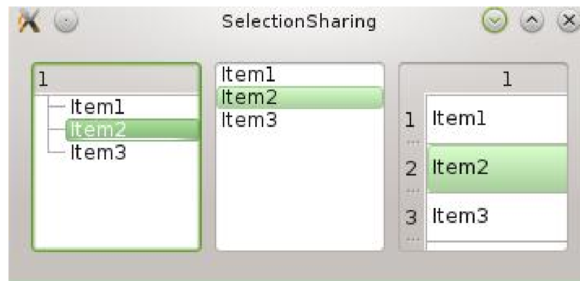

Рисунок 6 – Розділення виділення елементу між поданнями

2.4. Делегат

Для стандартних подань списків і таблиць відображення елементів виконується через делегування. Це дозволяє дуже просто створювати подання для любих потреб без написання великої кількості нового коду. Делегат відповідає за малювання кожного елементу і за його редагування (зміну користувачем). В Qt є готовий клас делегата `QItemDelegate` (рис. 7), який надає методи редагування кожного запису за допомогою елемента однорядкового текстового поля, і для більшості випадків його вповні достатньо. Для особливого контролю над відображенням і редагуванням даних потрібно створити власний делегат. Для цього необхідно успадкувати свій клас або від `QAbstractItemDelegate`, або від `QItemDelegate`.

Рисунок 7 – Класи делегатів

Приклад 2. Приклад делегата, який виділяє елемент, при наведенні на нього курсору миші.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ======================================================================
class SimpleDelegate : public QItemDelegate {
public:
    SimpleDelegate(QObject* pobj = 0) : QItemDelegate(pobj)
    {
    }
    // метод малювання
    void paint(QPainter*         pPainter,  // вказівник на Художника
```

```
         const QStyleOptionViewItem& option,// посилання на структуру в класі
QStyle
         const QModelIndex& index           // модельний індекс
         ) const
    {
        if (option.state & QStyle::State_MouseOver) {
            QRect            rect = option.rect;
            QLinearGradient gradient(0, 0, rect.width(), rect.height());

            gradient.setColorAt(0, Qt::white);
            gradient.setColorAt(0.5, Qt::blue);
            gradient.setColorAt(1, Qt::green);
            pPainter->setBrush(gradient);
            pPainter->drawRect(rect);
        }
        QItemDelegate::paint(pPainter, option, index);
    }
};

// ----------------------------------------------------------------------
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QStringListModel model;    // об'єкт список і його ініціалізація
    model.setStringList(QStringList() << "Item1" << "Item2" << "Item3");

    QListView listView;   // подання і його ініціалізація
    listView.setModel(&model);
    // встановлення у поданні делегата
    listView.setItemDelegate(new SimpleDelegate(&listView));
    // встановлення у вікні прогляду атрибути WA_Hover для реакції на переміщення
    // курсору
    listView.viewport()->setAttribute(Qt::WA_Hover);
    listView.show();

    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

>**./SimpleDelegate**

Рисунок 8 – Делегат, який виділяє елемент

В класі `SimpleDelegate` у методі для малювання `paint()` три аргументи. Перший аргумент – вказівник на об'єкт класу `QPainter`. Другий – це посилання на структуру `QStyleOptionViewItem`, визначену у класі `QStyle`. Третій – модельний індекс. Для перемальовування элементу кажний раз, коли над ним переміщається миша, перевіряються прапори стану встановлених бітів об'єкта структури `option`, щоб визначити чи знаходиться миша на елементі чи ні. Якщо біти прапора `QStyle::State_MouseOver` встановлені, це значить, що курсор миші знаходиться над елементом і, у цьому випадку, його фон заповнюється лінійним градієнтом.

Для зміни стандартного способу редагування, потрібно було б реалізувати в успадкованому класі методи `createEditor()`, `setEditorData()` і `setModelData()`. Метод `createEditor()` створює віджет для редагування. Метод `setEditorData()` встановлює дані у

віджеті редагування. Метод `setModelData()` видобуває дані з віджету редагування і передає їх моделі.

2.5. Індекс моделі

Індекс моделі – це невеликий список об'єктів, який використовується для адресації комірки в таблиці, яка має ієрархію. Індекс моделі є інформацією, яка складається з трьох частин: рядка, стовпця і внутрішнього ідентифікатора. Внутрішній ідентифікатор залежить від реалізації; це може бути вказівник або цілочисельний індекс. Кожна комірка таблиці має унікальний індекс, який подається класом `QModelIndex`. Індекси класу `QModelIndex` запам'ятовувати у програмі не має змісту, так як вони можуть змінитися, наприклад, після сортування. Однак ними зручно користуватися для отримання поточних значень комірок таблиці методом `QAbstractItemModel::data()`. Отримати індекс моделі для любої комірки можна методом `QAbstractItemModel::index()`, за допомогою якого можна рухатися по всій структурі даних. Наприклад, взнати значення комірки з координатами (2, 5) можна так: `QModelIndex index = pModel->index(2, 5, QModelIndex()); QVariant value = pModel->data(index);` Якщо підтаблиця не має елементу з координатами (2, 5), то метод `index()` поверне порожній індекс (`invalid index`). Чи індекс дійсно є невірним або порожнім, можна перевірити викликом методу `QModelIndex::isValid()`.

Клас `QModelIndex` має метод `data()`. Використовуючи об'єкт цього класу можна отримати доступ до даних без звернення до моделі.

Запитання.

1. Архітектура і структура технології “модель-подання”.
2. Компонента “Модель”. Клас і методи її реалізації.
3. Компонента “Подання”. Клас і методи її реалізації.
4. Компонента “Виділення елементу”. Клас і методи її реалізації.
5. Компонента “Делегат”. Клас і методи її реалізації.
6. Компонента “Індекс моделі”. Клас і методи її реалізації.

Завдання.

1. Написати програму в якій розділюється виділення елементу CheckBox між двома поданнями.
2. Написати програму в якій делегат виділяє елемент меню при наведенні на нього вказівника миші.
3. Написати програму в якій делегат виділяє комірку таблиці і висвічує значення індексу моделі.

Лабораторна робота № 8

8. Технологія “модель-подання” для ієрархічних даних

Мета роботи: отримання практичних навиків роботи з технологією “модель-подання” для ієрархічних даних. Робота з кольоровою палітрою.

1. Кольорова палітра

Кольорова палітра елементів керування – це таблиця, в якій містяться кольори, які використовуються віджетом при відображенні на екрані. Кольори віджетів не визначені остаточно і в любой момент можуть бути змінені передачею відповідного кольора тексту, кольора фону.

Кожний з віджетів містить у собі об’єкт палітри, доступ до якого можна отримати за допомогою методу `palette()` класу `QWidget`. Сама палітра – це клас `QPalette`, який складається з трьох основних груп об’єктів. Ці групи визначають три можливих стани віджету: *активний* (Active), *неактивний* (Inactive) і *недоступний* (Disabled). Кожна з цих груп складається з різних кольорових ролей (color roles). Кожна роль має щітку (QBrush) і колір (QColor).

Приклад 1. Віджет лічильника із зміненою палітрою.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QSpinBox     spb;    // віджет лічильника
    // присвоєння об’єкту палітра значення палітри лічильника
    QPalette pal = spb.palette();
    // зміна об’єкта палітри
    pal.setBrush(QPalette::Button, QBrush(Qt::red, Qt::Dense3Pattern)); // щітка
    pal.setColor(QPalette::ButtonText, Qt::blue);  // колір
    pal.setColor(QPalette::Text, Qt::magenta);
    // колір активного стану
    pal.setColor(QPalette::Active, QPalette::Base, Qt::green);

    spb.setPalette(pal); // застосування зміненої палітри до лічильника
    spb.resize(50, 50);
    spb.show();

    app.setStyle(new QWindowsStyle);  // застосування нового стилю

    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

>**./WidgetPalette**

Рисунок 1 – Зміна палітри віджету

Якщо необхідно змінити палітру для всіх віджетів зразу то потрібно це робити централізовано. Для цього необхідно передати об'єкт палітри у статичний метод `QAppilcation::setPalette()`.

Приклад зміни палітри для всього застосування.

```
int main (int argc, char** argv)
{
   QApplication app(argc, argv); QPalette pal(Qt::red, Qt::blue);
   QApplication::setPalette(pal);
}
```

2. Ієрархічні дані

Кожна комірка в таблиці може мати дочірні таблиці. Для створення і відображення ієрархії використовується клас модель `QStandardItemModel`. Спочатку створюється об'єкт цього класу, а потім методом `setData()` встановлюються дані для кожного елементу. Надалі ці дані можна буде отримати методом `data()`.

Приклад 1. Відображення ієрархічних даних.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>

// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication       app(argc, argv);
    QStandardItemModel model(5, 3);         // об'єкт класу

    for (int nTopRow = 0; nTopRow < 5; ++nTopRow) {
        QModelIndex index = model.index(nTopRow, 0);// отримання індекса для
елемента
        model.setData(index, "item" + QString::number(nTopRow + 1)); // дані
елемента
        // для кожного елемента вставлення підтаблиці 4x3
        model.insertRows(0, 4, index);
        model.insertColumns(0, 3, index);
        // заповнення елементів підтаблиці
        for (int nRow = 0; nRow < 4; ++nRow) {
            for (int nCol = 0; nCol < 3; ++nCol) {
                QString strPos = QString("%1,%2").arg(nRow).arg(nCol);
                model.setData(model.index(nRow, nCol, index), strPos);
            }
        }
    }

    QTreeView treeView;     // створення об'єкта проглядача деревовидної моделі
    treeView.setModel(&model);
    treeView.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./HierarchicalModel
```

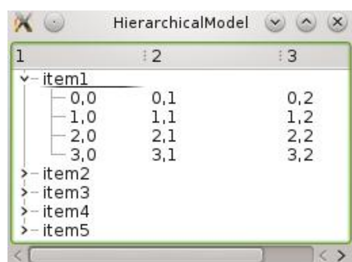

Рисунок 2 – Відображення ієрархічних даних

Для роботи з ієрархією каталогів використовується модель `QDirModel`.

Приклад 2. Відображення каталогів і файлів

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QDirModel    model;     // створення об’єкта моделі
    QTreeView    treeView;  // створення об’єкта для прогляду ієрархічних даних

    treeView.setModel(&model);
    treeView.show();

    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання**:**

>**./HierarchicalModel**

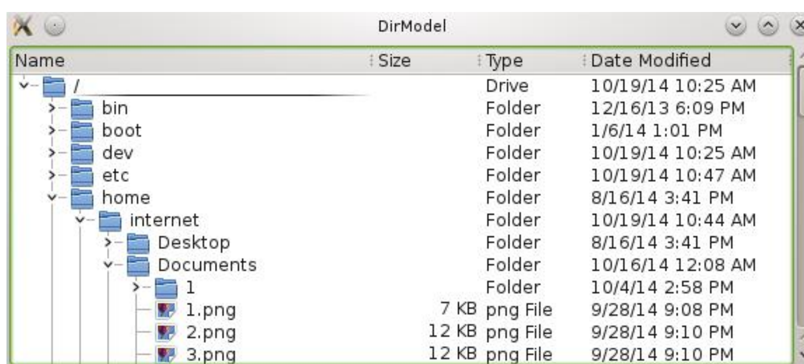

Рисунок 3 – Відображення каталогів і файлів

Для показу певного шляху використовується метод `index()`, який має три параметри: стовпчик, рядок і індекс предка, але можна обійтися і без них передавши стрічку із шляхом. Так, щоб побачити вміст тільки поточного каталогу в приклад 2 потрібно добавити наступні рядки:

```
QModelIndex index = model.index(QDir::currentPath());
treeView.setRootIndex(index);
```

Скориставшись слотом `setRooIndex()` і ще декількома слотами і сигналами можна об’єднати ієрархічне і табличне подання, реалізувавши таким чином проглядач.

Приклад 3. Проглядач.

```
// ======================================================================
//  main.cpp
// ======================================================================
```

```
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QSplitter    spl(Qt::Horizontal);  // об’єкт вертикальний розділювач
    QDirModel    model;                // об’єкт модель

    QTreeView* pTreeView = new QTreeView;  // оглядач деревовидних моделей
    pTreeView->setModel(&model);

    QTableView* pTableView = new QTableView;   // оглядач таблиць
    pTableView->setModel(&model);

    QObject::connect(pTreeView,  SIGNAL(clicked(const QModelIndex&)),
                     pTableView, SLOT(setRootIndex(const QModelIndex&))
                    );
    QObject::connect(pTableView, SIGNAL(activated(const QModelIndex&)),
                     pTreeView, SLOT(setCurrentIndex(const QModelIndex&))
                    );
    QObject::connect(pTableView, SIGNAL(activated(const QModelIndex&)),
                     pTableView,  SLOT(setRootIndex(const QModelIndex&))
                    );

    spl.addWidget(pTreeView);
    spl.addWidget(pTableView);
    spl.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./HierarchicalModel
```

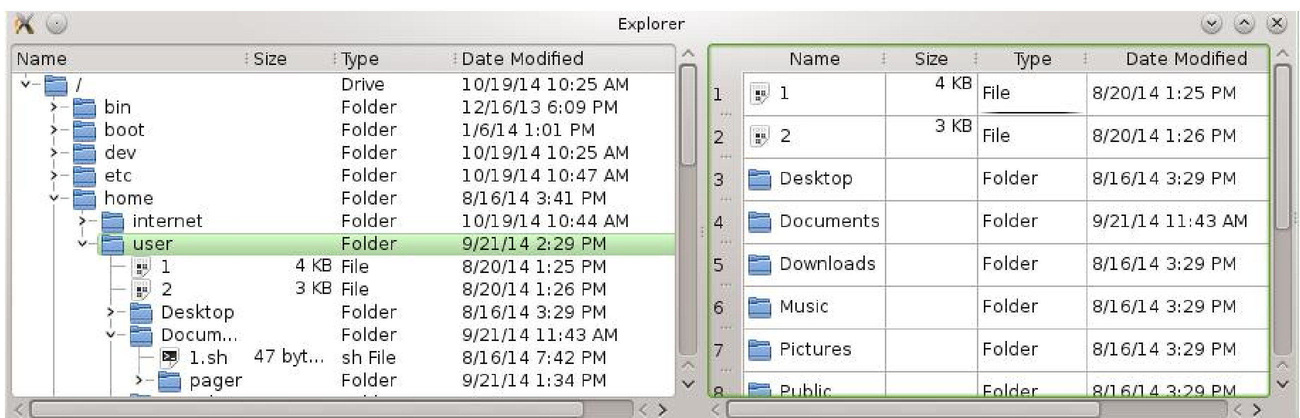

Рисунок 4 – Вікно проглядача

В програмі використовуються сигнально-слотові з’єднання. Перше з’єднання в табличному поданні встановлює каталог, вибраний у ієрархічному поданні, як вузловий. Друге з’єднання потрібно для того, щоб при виборі одного з каталогів табличного подання виділявся цей каталог в ієрархічному поданні. Останнє з’єднання показує вміст каталогу при роботі у табличному поданні. Таким чином, подвійне клацання миші або натискання на клавішу <Enter> на каталозі вишле з табличного подання сигнал `activated()`, який буде відібраний самим табличним подання, і слот `setRootIndex()` встановить даний каталог як базовий. А це значить, що табличне подання дозволяє входить тільки всередину каталогів, а не виходити з них.

2. Ролі елементів

Завдяки індексу (QModelIndex) модель може посилатися на потрібні дані і, тим самим, передати ці дані поданню. Для того щоб дані були правильно показані на екрані, подання звертається через об'єкт індекс (QModelIndex) до так званих ролей.

Кожний елемент в моделі може містити різні дані, які прив'язані до різних значень ролей. Дані заданої ролі можна отримати за допомогою метода QAbstractItemModel::data(), передавши в нього індекс і значення потрібної ролі, наприклад DisplayRole. Якщо для заданої ролі не буде знайдено відповідного значення, то метод data() поверне об'єкт класу QVariant, який не містить ніяких даних.

Елементи, крім тексту, можуть мати і растрові зображення, а також і додатковий текст. До цих даних потрібно якимось чином звертатися. При малюванні елементів використовується делегат і тому можна скористатися додатковою інформацією, яка закладена в елементі з використанням ролей.

Існуючі подання і делегати розуміють багато ролей. Найбільш часто використовуються:

- DisplayRole – текст для показу:
- DecorationRole – растрове зображення;
- FontRole – шрифт для тексту;
- ToolTipRole - текст для підказки (ToolTip);
- WhatThisRole - текст для підказки "Що це?";
- TextColorRole - колір тексту;
- BackgroundColorRole – колів фону елемента.

Приклад 4. Встановлення ролей Qt::DisplayRole, Qt::ToolTipRole і Qt::DecorationRole. В прикладі отримано елемент з текстом, растровим зображенням і виринаючою підказкою.

```
// ===================================================================
//  main.cpp
// ===================================================================
#include <QtGui>
// -------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QStringList  lst;   // об'єкт список
    lst << "Linux" << "Windows" << "MacOS" << "OS2"; // ініціалізація списку

    QStandardItemModel model(lst.size(), 1);   // створення стандартної моделі
    for (int i = 0; i < model.rowCount(); ++i) {
        QModelIndex index = model.index(i, 0);  // створення індексів моделі
        QString     str   = lst.at(i);  // список тексту
        // роль для відображення списку тексту
        model.setData(index, str, Qt::DisplayRole);
        // роль для відображення виринаючих підказок
        model.setData(index, "ToolTip for " + str, Qt::ToolTipRole);
        // роль для відображення декорацій
        model.setData(index, QIcon(str + ".jpg"), Qt::DecorationRole);
    }
    QListView listView;     // проглядач списку
    listView.setViewMode(QListView::IconMode);  // режим поглядання списку
    listView.setModel(&model);   // зв'язування списку з моделлю
    listView.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./HierarchicalModel
```

Рисунок 5 – Ролі

3. Створення власної моделі даних

Для створення власної моделі даних потрібно успадкувати клас `QAbstractItemModel`, або один із його нащадків.

В діаграмі класів (рис.4. Класи подань) відсутня модель для цілих чисел. В наступному прикладі показано як реалізувати власну модель для цілих чисел.

Приклад 5. `IntListModel.c` – створення власної моделі

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
#include "IntListModel.h"
// ----------------------------------------------------------------------
int main( int argc, char** argv ) {
    QApplication app( argc, argv );
    // об'єкт власної моделі, якому передаються цілі числа
    IntListModel model(QList<int>() << 123 << 2341 << 32 << 5342 << 723);

    QListView list;          // створення об'єкту список
    list.setModel(&model);   // зв'язування списку з моделлю
    list.show();

    QTableView table;        // створення об'єкту таблиця
    table.setModel(&model);  // зв'язування таблиці з моделлю
    table.show();

    return app.exec();
}
```

Заголовковий файл моделі.

```
// ======================================================================
//  IntListModel.h
// ======================================================================
#ifndef _IntListModel_h_
#define _IntListModel_h_

#include <QAbstractListModel>
// ======================================================================
class IntListModel : public QAbstractListModel {
Q_OBJECT
private:
    QList<int> m_list;

public:
    IntListModel(const QList<int>& list, QObject* pobj = 0);

    QVariant data(const QModelIndex& index, int nRole) const;

    bool setData(const QModelIndex& index,
                 const QVariant&    value,
                 int                nRole
                );
```

```
    int rowCount(const QModelIndex& parent = QModelIndex()) const;

    QVariant headerData(int             nSection,
                        Qt::Orientation orientation,
                        int             nRole = Qt::DisplayRole
                       ) const;

    Qt::ItemFlags flags(const QModelIndex &index) const;
};
#endif  //_IntListModel_h_
```

Для реалізації моделі на базі класу `QAbstractListModel` необхідно реалізувати методи `rowCount()` і `data()`. Метод `rowCount()` буде повідомляти про кількість рядків моделі, а метод `data()` відповідає за доставлення даних, які він повертає в об'єктах класу `QVariant`.

Також перевизначається метод `headerData()`. Це потрібно для того, щоб модель працювала з заголовками `QTableView` і `QTreeView`. Модель допускає зміни даних, і саме тому перевизначаються методи `flags()` і `setData()`.

Методи класу `IntListModel`.

Конструктор `IntListModel` ініціалізує атрибут `m_list` списком цілих чисел і передає вказівник на об'єкт предка успадкованому класу:

```
IntListModel::IntListModel(const QList<int>& list, QObject* pobj/*=0*/)
              : QAbstractListModel(pobj) , m_list(list) {}
```

Метод `data()`. Модель не повинна поставляти дані, це робота інтерфейсу моделі, яка визначена для зв'язку з структурою даних, яку опитує подання. Метод `data()` повертає інформацію про елемент, в якій зацікавлене подання, в об'єкті класу `QVariant`. Крім індексу в цей метод передаються значення ролей. В даному випадку, якщо вони призначені для відображення (`Qt::DisplayRole`) або редагування (`Qt::EditRole`), повертається значення, записане в атрибуті, який є списком цілих чисел, на позиції рядка. Якщо роль не призначена для відображення або редагування, або ж подання запитує дані, яких не має, то повертається порожній об'єкт `Qvariant`, що вказує на відсутність даних.

```
QVariant IntListModel::data(const QModelIndex& index, int nRole) const
{
    if (!index.isValid()) { return QVariant(); }
    return (nRole == Qt::DisplayRole || nRole == Qt::EditRole)
           ? m_list.at(index.row())  : QVariant();
}
```

Значення у методі `setData()` встановлюють три параметри: індекс, значення і роль. Спочатку перевіряється індекс викликом методу `isValid()`. Якщо індекс не порожній, а роль призначена для редагування, то замінюється існуюче значення в атрибуті списку (`m_list`) новим, використовуючи метод `replace()`. Перед заміною перетворюється атрибут `value` класу `QVariant` до цілого типу за допомогою шаблонного методу `QVariant::value<T>()`, параметризувавши його типом `int`. Цю конструкцію можна замінити методом `QVariant::toInt()`. Після заміни відправляється сигнал `dataChanged()`, це необхідно для того, щоб всі під'єднані до моделі подання могли зразу оновити свій вміст. Значення `true`, яке повертається з методу, повідомляє про успішне виконання операції встановлення даних, а `false` повідомляє подання про помилку.

```
bool IntListModel::setData(const QModelIndex& index,
                           const QVariant&    value,
                           int                nRole )
{
    if (index.isValid() && nRole == Qt::EditRole) {
        m_list.replace(index.row(), value.value<int>());
        emit dataChanged(index, index);
        return true;
    }
```

```
        return false;
    }
```

Метод `rowCount()` повідомляє про кількість рядків. Кількість рядків моделі відповідає кількості елементів, які містяться в атрибуті списку цілих чисел (m_list).

```
int IntListModel::rowCount(const QModelIndex& parent/*=QModelIndex()*/
                           ) const
{
    Q_UNUSED(parent);
    return m_list.size();
}
```

Метод `headerData()` потрібний для того, щоб модель була в стані підписувати горизонтальні і вертикальні секції заголовків, які мають ієрархічне і табличне подання. Ієрархічне подання має тільки горизонтальні заголовки, табличне подання має обидва типи заголовків (горизонтальні і вертикальні). Якщо подання запитає надпис для горизонтального заголовку, то повертається рядок "Number", а для вертикальних секцій заголовків повертається номер переданної секції.

```
QVariant IntListModel::headerData(int             nSection,
                                  Qt::Orientation orientation,
                                  int             nRole/*=DisplayRole*/
                                 ) const
{
    if (nRole != Qt::DisplayRole) {
        return QVariant();
    }
    return (orientation == Qt::Horizontal) ? QString("Number")
                                           : QString::number(nSection);
}
```

Для надання можливості редагування елементів, метод `flags()` повертає для кожного елементу успадковане значення з добавленням прапору `Qt::ItemIsEditable`. Якщо індекс порожній, то повертається значення без добавлень.

```
Qt::ItemFlags IntListModel::flags(const QModelIndex& index) const
{
    Qt::ItemFlags flags = QAbstractListModel::flags(index);
    return index.isValid() ? (flags | Qt::ItemIsEditable)
                           : flags;
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./IntListModel
```

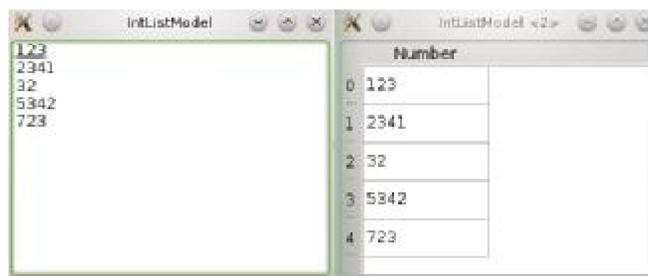

Рисунок 6 – Відображення даних моделі списка цілих чисел

Для створення табличної моделі на базі класу `QAbstractTableModel` можна поступити так як для випадку класу `QAbstractListModel`. Додатково до цього, в успадкованому від `QAbstractTableModel` класі необхідно реалізувати метод `columnCount()`, який надає інформацію про кількість стовпців таблиці.

Приклад 6. Відображення даних табличної моделі даних.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ======================================================================
class TableModel : public QAbstractTableModel {
private:
    int                          m_nRows;
    int                          m_nColumns;
    QHash<QModelIndex, QVariant> m_hash;   // хеш для зберігання даних

public:
    // конструктор з ініціалізацією даних
    TableModel(int nRows, int nColumns, QObject* pobj = 0)
        : QAbstractTableModel(pobj)
        , m_nRows(nRows)
        , m_nColumns(nColumns)
    {
    }

    // задання даних комірок. Дані комірок можна міняти, зо задає роль EditRole
    QVariant data(const QModelIndex& index, int nRole) const
    {
        if (!index.isValid()) {
            return QVariant();
        }
        QString str =
           QString("%1,%2").arg(index.row() + 1).arg(index.column() + 1);
        return (nRole == Qt::DisplayRole || nRole == Qt::EditRole)
               ? m_hash.value(index, QVariant(str))
               : QVariant();
    }

    // сигнал про зміну даних в комірці
    bool setData(const QModelIndex& index,
                 const QVariant&    value,
                 int                nRole
                )
    {
        if (index.isValid() && nRole == Qt::EditRole) {
            m_hash[index] = value;
            emit dataChanged(index, index);
            return true;
        }
        return false;
    }
    // номер стовпця
    int rowCount(const QModelIndex&) const
    {
        return m_nRows;
    }
    // номер рядка
    int columnCount(const QModelIndex&) const
    {
        return m_nColumns;
    }
    // встановлення пропору елементів таблиці
    Qt::ItemFlags flags(const QModelIndex& index) const
    {
        Qt::ItemFlags flags = QAbstractTableModel::flags(index);
        return index.isValid() ? (flags | Qt::ItemIsEditable)
                               : flags;
    }
```

```
};

// -----------------------------------------------------------------------
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    TableModel model(20, 20);  // створення табличної моделі

    QTableView tableView;      // створення табличного проглядача
    tableView.setModel(&model); // зв'язування проглядача і моделі
    tableView.show();

    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

>**./TableModel**

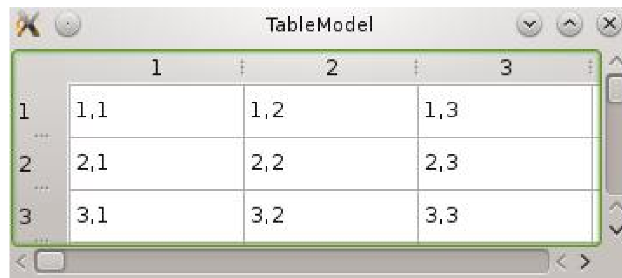

Рисунок 7 – Відображення табличної моделі даних

Реалізація моделі на основі класу `QAbstractItemModel` виглядатиме аналогічно до реалізації табличної моделі `QAbstractTableModel`. Для цього додатково в успадкованому класі потрібно реалізувати методи `QAbstractItemModel::index()` і `QAbstractItemModel::parent()`, а для створення індексів перевантажити метод `QAbstractItemModel::createIndex()`.

4. Проміжна модель

В оригінальній моделі може трапитися так, що який-небудь з елементів розміщений першим, а його потрібно помістити в кінець або в середину. Зміна розміщення даних в оригінальній моделі даних викликає зміни у всіх приєднаних поданнях, що може бути небажаним. У таких випадках використовується проміжна модель. *Проміжна модель* – це модель, яка знаходиться між моделлю даних і поданням (рис. 6). Така модель надає можливість маніпуляцій з даними, при цьому не змінюються дані оригінальної моделі.

Рисунок 8 – Схема використання проміжної моделі

За її допомогою можна сортувати або переставляти дані. Таким чином можна створити два подання: одне для змінених даних, а інше для оригінальних.

Інша корисна операція, яку можна виконати за допомогою проміжної моделі – відбір елементів даних. Для цього у проміжній моделі необхідно встановити критерії відбору. Клас `QSortFilterProxyModel` є узагальненою реалізацією проміжної моделі, який дозволяє

виконувати сортування і відбір. Для задання критеріїв відбору може бути використаний слот `QSortFilterProxyModel::setFilterRegExp()`, в який передається об'єкт регулярного виразу класу `QRegExp`.

При відборі модель повертає індекси тільки тих рядків, для яких текст у стопці відповідає вказаному критерію. При сортуванні порядок розміщення здійснюється у відповідності із значеннями елементів, розміщених у кожному стовпці. Сортувати кожний стовпець можна за зростанням або спаданням.

Приклад 7. Відбір елементів, імена яких починаються на букву "E".

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QStringListModel model;      // створення моделі і ініціалізація її списком
    model.setStringList(QStringList() << "Xandria"
                                      << "Epica"
                                      << "Therion"
                                      << "Evanescence"
                                      << "Nightwish" );

    QSortFilterProxyModel proxyModel;   // створення проміжної моделі
    proxyModel.setSourceModel(&model);  // зв'язування проміжної і початкової
моделі
    proxyModel.setFilterWildcard("E*"); // фільтр для відбору елементів

    QListView* pListView1 = new QListView;  // спискове подання 1
    pListView1->setModel(&model);           // зв'язування з основною моделлю

    QListView* pListView2 = new QListView;  // спискове подання 2
    pListView2->setModel(&proxyModel);      // зв'язування з проміжною моделлю

    //Layout setup
    QHBoxLayout* phbxLayout = new QHBoxLayout;    // горизонтальне розміщення
    phbxLayout->addWidget(pListView1);
    phbxLayout->addWidget(pListView2);
    wgt.setLayout(phbxLayout);

    wgt.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./ProxyModel
```

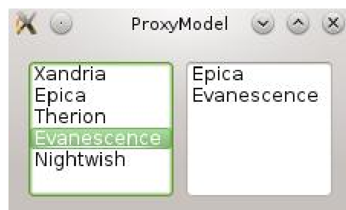

Рисунок 9 – Відбір елементів

5. Модель на основі елементно-базованих класів

Класи з елементно-базованим підходом `QListWidget`, `QTreeWidget`, `QTableWidget` працюють за принципом: створити елемент з даними і вставити його у подання (дані містяться в самих елементах).

В дійсності, ці класи основані на архітектурі модель-подання і успадковані від класів `QListView`, `QTreeView`, `QtableView`. Однак, на відміну від цих класів, вони мають свою власну, вбудовану модель даних. А це значить, що дані елементно-базованих класів можна розділити з іншими класами, для чого потрібно лише отримати вказівник на цю модель даних, який повертає метод `QAbstractItemView::model()`.

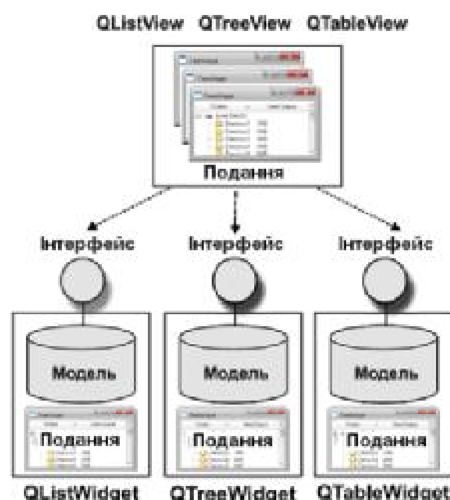

Рисунок 10 – Розділення моделей класів з елементно-базованим підходом між поданнями

Подібний механізм розділення моделей даних віджетів елементно-базованого підходу з поданнями не рекомендується з позиції "модель-подання", але для простих ситуацій він вповні прийнятний і може допомогти зекономити час, якщо у програмі вже є реалізація елементно-базованих класів.

На рис. 11 показаний приклад розділення моделі віджета класу `QListWidget` (зліва) з поданням класу `QListView` (справа), тобто, обидва віджети дивляться на одну і ту ж модель даних.

Приклад 8. Отримання доступу до моделі.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication     app(argc, argv);
    QStringList      lst;           // об'єкт список
    QListWidget      lwg;          // створення об'єкта елементно-базованого класу
    QListWidgetItem* pitem = 0;    // вказівник на елементи

    lwg.setIconSize(QSize(48, 48));
    lst << "Linux" << "Windows" << "MacOS" << "OS2"; // заповнення списку
    foreach(QString str, lst) {
        pitem = new QListWidgetItem(str, &lwg);
        pitem->setIcon(QPixmap(str + ".jpg"));
    }
    lwg.setWindowTitle("ListWidget");
    lwg.show();
```

```
    QListView listView;      // об'єкт подання списку
    listView.setModel(lwg.model());  // передача вказівника у модель списку і
                                     // розділення моделі
    listView.setSelectionModel(lwg.selectionModel()); // розділеня моделі
    listView.setWindowTitle("ListView");
    listView.show();

    return app.exec();
}
```

```
> qmake -project; qmake; make
```

Запуск на виконання:

```
>./WidgetAndView
```

Рисунок 11 – Розділення моделі даних

Запитання.

1. Віджет кольорова палітра `QPalette`. Група об'єктів і їх ролі.
2. Клас для створення ієрархічної моделі даних `QStandardItem`.
3. Клас модель для відображення каталогів і файлів `QDirModel`. Реалізація проглядача.
4. Клас `QModelIndex`. Ролі елементів. Делегати.
5. Створення власної моделі даних.
6. Проміжна модель. Клас `QSortFilterProxyModel`, методи.
7. Модель на основі елементно-базованих класів `QListWidget`, `QTreeWidget`, `QTableWidget`.

Завдання.

1. Реалізувати модель відображення цілих чисел на основі класу `QabstractItemModel`.
2. Реалізувати проміжну модель з використанням класу `QTableView`.
3. Реалізувати розділення моделі віджету класу `QTableWidget` з поданням `QTableView`.

Лабораторна робота № 9

9. Обробка подій

Мета роботи: отримання практичних навиків обробки подій.

1. Ієрархія класів подій

Обробка подій лежить в основі роботи кожного застосування. Події можна охарактеризувати як механізм оповіщення застосування про яку-небудь подію. Механізми сигналів і слотів, порівняно з подіями, є механізмом більш високого рівня, призначений для зв’язку об’єктів. Події можуть бути оброблені лише одним методом, а сигнали можуть оброблятися необмеженою кількістю зв’язаних із ними слотів. Сигнали можуть базуватися на подіях.

Qt надає цілий ряд класів для різного роду подій. Ієрархія цих класів показана на рис.1.

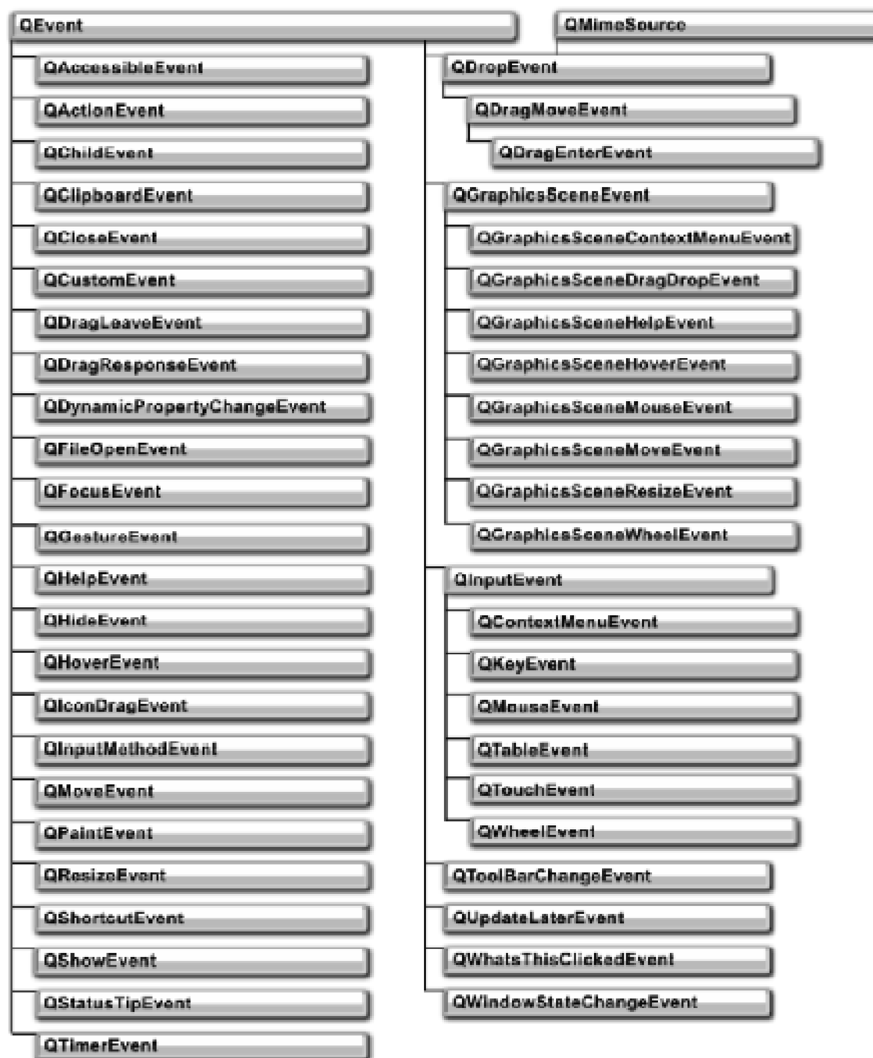

Рисунок 1 – Ієрархія класів подій

Клас QEvent є базовим для всіх категорій подій. Його об’єкти містять інформацію про тип події, яка відбулася. А для кожного типу подій є цілочисельний ідентифікатор, який встановлюється у конструкторі і може бути отриманий методом `QEvent::type()`.

Клас події `QEvent` містить методи `accept()` і `ignore()`, за допомогою яких встановлюється або скидується спеціальний прапор, регулюючий подальшу обробку події предком об'єкта. Якщо був викликаний метод `ignore()`, то при поверненні з метода обробки події подія буде передано далі на обробку об'єкту-предку.

Клас `QInputEvent` є базовим для подій, зв'язаних с введенням користувача. Цей клас реалізує всього лише один метод `modifiers()`. За його допомогою успадковані від нього класи можуть отримувати стан клавіш-модифікаторів `<Ctrl>`, `<Shift>` і `<Alt>`, які можуть бути натиснуті в момент виникнення події. Деякі їх значення вказані в табл. 1.

Таблиця 1 – Деякі значення модифікаторів

Константа	Hex значення	Описання
NoModifier	0	Клавіші модифікаторів не натиснуті
ShiftModifier	2000000	Натиснута клавіша <Shift>
ControlModifier	4000000	Натиснута клавіша <Ctrl>
AltModifier	8000000	Натиснута клавіша <Alt>

Обробка подій починається з момента виклику в основній програмі метода `QCoreApplication::exec()`. Події доставляються всім об'єктам, створеним від класів, які успадковані від класу `QObject`. Деякі події можуть бути доставлені зразу, а деякі попадають в чергу і можуть бути оброблені тільки при поверненні управління циклу обробки події `QCoreApplication::exec()`. Qt використовує спеціальні механізми для оптимізації деяких типів подій. Так, наприклад, серія подій перемалювання `QPaintEvent` з метою збільшення продуктивності може бути "упакована" (об'єднана) в одну подію, з регіоном малювання, складеним з регіонов всіх подій малювання, які знаходяться у черзі. Тим самим метод обробки подій малювання (`paintEvent()`) буде викликаний тільки один раз.

2. Перевизначення спеціалізованих методів обробки подій

Перевизначення спеціалізованих методів є самим поширеним способом їх обробки. Для того щоб обробити певну подію, потрібно успадкувати необхідний клас і перевизначити потрібний метод обробки події. В цей метод передається вказівник на об'єкт події, який містить інформацію про неї. Кожний метод отримує об'єкт відповідного типу, наприклад, методи `keyPressEvent()` і `keyReleaseEvent()` отримують вказівник на об'єкт класу `QKeyEvent`.

3. Події клавіатури. Клас QkeyEvent

Клас `QKeyEvent` містить дані про події клавіатури. З його допомогою можна отримати інформацію про клавішу, яка викликала подію, а також ASCII-код відображуваного символу. Об'єкт подія передається в методи `QWidget::keyPressEvent()` і `QWidget::keyReleaseEvent()`, визначені в класі `QWidget`. Подія може викликатися натисканням любої клавіші на клавіатурі, включаючи `<Shift>`, `<Ctrl>`, `<Alt>`, `<Esc>` і `<F1>–<F12>`. Виняток складають клавіша табулятора `<Tab>` і її сумісне натискання з клавішею `<Shift>`, які використовуються методом обробки `QWidget::event()` для передачі фокуса наступному віджету.

Метод `keyPressEvent()` викликається кожний раз при натисканні однієї з клавіш на клавіатурі, а метод `keyReleaseEvent()` – при відпусканні. В методі обробки події за допомогою методу `QKeyEvent::key()` можна визначити, яка з клавіш ініціювала його. Цей метод повертає значення цілого типу, яке можна порівняти з константами клавіш, визначеними в класі Qt.

Метод для обробки подій клавіатури класу, успадкованого від класу `QWidget`, може виглядати наступним чином.

```
void MyWidget::keyPressEvent(QKeyEvent* pe) {
   switch (pe->key()) {
```

```
      case Qt::Key_Z:
         if (pe->modifiers() & Qt::ShiftModifier) {
            // Виконати які небудь дії
         }
         else {
            // Виконати які-небудь дії
         }
          break;
      default: QWidget::keyPressEvent(pe); // Передати подію далі
   }
}
```

В даному прикладі перевіряється, чи натиснуті сумісно клавіші `<Z>` і `<Shift>`. Для перевірки статусу значення, який повертається методом `modifiers()`, використовуються значення, вказані в табл. 1.

4. Клас QFocusEvent

Коли користувач набирає що-небудь на клавіатурі, інформацію про натиснуті клавіші може приймати тільки один віджет. Якщо віджет в даний момент вибраний для введення з клавіатури, то говорять, що він *знаходиться у фокусі*. Об'єкт події фокуса `QFocusEvent` передається у методи обробки повідомлень `focusInEvent()` і `focusOutEvent()`. Цей об'єкт не містить значимої інформації. Основне призначення класу `QFocusEvent` – повідомити про отримання або втрату віджетом фокуса, для того щоб можна було, наприклад, змінити його зовнішній вид. Ці методи викликаються у тому випадку, коли віджет отримує (`focusInEvent()`) або втрачає (`focusOutEvent()`) фокус.

5. Подія оновлення контексту малювання. Клас QPaintEvent

Qt підтримує *подвійну буферизацію* (double buffering). Її можна відключити викликом методу `QWidget::setAttribute(Qt::WA_PaintOnScreen)`. Але при цьому можуть пропадати частини графічної інформації при перекритті і звільненні вікон. Щоб цього не відбулося, необхідно отримати і обробити подію `QPaintEvent`. В об'єкті класу `QPaintEvent` передається інформація для перемалювання всього зображення або його частини. Подія виникає тоді, коли віджет вперше відображується на екрані явним або неявним викликом методу `show()`, а також в результаті виклику методів `repaint()` і `update()`. Об'єкт подія передається в метод `paintEvent()`, у якому реалізується відображення самого віджета. У більшості випадків цей метод використовується для повного перемалювання віджета. Для малих віджетов це прийнятно, але для віджетів великих розмірів краще перемалювати тільки окрему потрібну область. Для отримання координат і розмірів такої області викликається метод `region()`. Викликом методу `contains()` можна перевірити, чи знаходиться об'єкт в заданій області. Наприклад:

```
MyClass::paintEvent(QPaintEvent* pe) {
   QPainter painter(this);
   QRect r(40, 40, 100, 100);
   if (pe->region().contains(r)) {  painter.drawRect(r); }
}
```

6. Події миші. Клас QMouseEvent

Об'єкт цього класу містить інформацію про події, викликані дією миші, і зберігає в собі інформацію про позицію вказівника миші в момент виклику події, статус кнопок миші і навіть деяких клавіш клавіатури. Цей об'єкт передається в методи `mousePressEvent()`, `mouseMoveEvent()`, `mouseReleaseEvent()` і `mouseDoubleClickEvent()`. Метод

mousePressEvent() викликається тоді, коли відбулося натискання на одну з кнопок миші в області віджета. При переміщенні миші буде викликатися метод mouseMoveEvent(), а при відпусканні кнопки викличеться метод mouseReleaseEvent(). За замовчуванням метод mouseMoveEvent() викликається при переміщенні вказівника миші, тільки якщо одна з її кнопок натиснута. Це дозволяє не створювати лишніх подій під час простого переміщення вказівника. Якщо ж необхідно відслідковувати всі переміщення вказівника миші, то потрібно використати метод setMouseTracking() класу QWidget, передавши йому параметр true. Метод mouseDoubleClickEvent() викликається при подвійному клацанні кнопкою миші в області віджета.

Для визначення місцезнаходження вказівника миші в момент виникнення події можна скористатися методами globalX(), globalY(), x() і y(), які повертають цілі значення. Також можна скористатися методами pos() або globalPos(). Метод pos() класу QMouseEvent повертає позицію вказівника миші в момент виникнення події відносно лівого верхнього кута віджету. Якщо потрібна абсолютна позиція (відносно лівого верхнього кута екрану), то її отримують за допомогою методу globalPos(). Викликавши метод button(), можна взнати, яка з кнопок миші була натиснута в момент виникнення події (табл. 2). Метод buttons() повертає бітову комбінацію значень, табл. 2. Як видно з таблиці, значення не перекриваються, тому можна застосовувати операцію | (АБО) для їх об'єднання.

Якщо необхідно взнати, чи були в момент виникнення події миші натиснуті клавіші-модифікатори <Ctrl>, <Shift> і/або <Alt>, то це можна перевірити за допомогою методу modifiers(), реалізованого у базовому класі QInputEvent.

Таблиця 2 – Значення перечислення MouseButton з простору імен Qt

Константа	Значення	Описання
NoButton	0	Кнопки миші не натиснуті
LeftButton	1	Натиснута ліва кнопка
RigthButton	2	Натиснута права кнопка
MidButton	4	Натиснута середня кнопка

При виклику mouseDoubleClickEvent() метод mousePressEvent() викликається двічі, так як подвійне клацання обробляється як два простих натискання. За замовчуванням інтервал подвійного клацання становить 400 мс, а для зміни цього інтервалу потрібно викликати метод setDoubleClickInterval() класу QApplication.

Приклад 1. Обробка подій миші.

```
// ==========================================================================
//  MouseObserver.h
// ==========================================================================
#ifndef _MouseObserver_h_
#define _MouseObserver_h_
// ==========================================================================
#include <QtGui>
// ==========================================================================
class MouseObserver : public QLabel {
public:
    MouseObserver(QWidget* pwgt = 0);
protected:
    virtual void mousePressEvent  (QMouseEvent* pe); // метод для натискання
кнопки
    virtual void mouseReleaseEvent(QMouseEvent* pe); // метод відпускання кнопки
    virtual void mouseMoveEvent   (QMouseEvent* pe); // метод переміщення миші

    void dumpEvent(QMouseEvent* pe,const QString& strMessage); // стан подій миші
    QString modifiersInfo (QMouseEvent* pe); // інфо про клавіші модифікатори
    QString buttonsInfo   (QMouseEvent* pe); // інфо про кнопки миші
```

```
};
#endif //_MouseObserver_h_

// ======================================================================
//  MouseObserver.cpp
// ======================================================================
#include "MouseObserver.h"
// ----------------------------------------------------------------------
MouseObserver::MouseObserver(QWidget* pwgt /*= 0*/) : QLabel(pwgt)
{
    setAlignment(Qt::AlignCenter);  // центрування інформації
    setText("Mouse interactions\n(Press a mouse button)");
}
// ----------------------------------------------------------------------
/*virtual*/void MouseObserver::mousePressEvent(QMouseEvent* pe)
{
    dumpEvent(pe, "Mouse Pressed"); // відслідковування подій миші
}
// ----------------------------------------------------------------------
/*virtual*/void MouseObserver::mouseReleaseEvent(QMouseEvent* pe)
{
    dumpEvent(pe, "Mouse Released"); // відслідковування подій миші
}
// ----------------------------------------------------------------------
/*virtual*/ void MouseObserver::mouseMoveEvent(QMouseEvent* pe)
{
    dumpEvent(pe, "Mouse Is Moving"); // відслідковування подій миші
}
// ----------------------------------------------------------------------
void MouseObserver::dumpEvent(QMouseEvent* pe, const QString& strMsg)
{                  // інфо про події
    setText(strMsg
            + "\n buttons()=" + buttonsInfo(pe)
            + "\n x()=" + QString::number(pe->x())
            + "\n y()=" + QString::number(pe->y())
            + "\n globalX()=" + QString::number(pe->globalX())
            + "\n globalY()=" + QString::number(pe->globalY())
            + "\n modifiers()=" + modifiersInfo(pe)
           );
}

// ----------------------------------------------------------------------
QString MouseObserver::modifiersInfo(QMouseEvent* pe)//інфо про клавіші-
модифікатори
{
    QString strModifiers;
    if(pe->modifiers() & Qt::ShiftModifier) {
        strModifiers += "Shift ";
    }
    if(pe->modifiers() & Qt::ControlModifier) {
        strModifiers += "Control ";
    }
    if(pe->modifiers() & Qt::AltModifier) {
        strModifiers += "Alt";
    }
    return strModifiers;
}

// ----------------------------------------------------------------------
QString MouseObserver::buttonsInfo(QMouseEvent* pe) // інфо про натиснуті кнопки
{
    QString strButtons;
    if(pe->buttons() & Qt::LeftButton) {
```

```
        strButtons += "Left ";
    }
    if(pe->buttons() & Qt::RightButton) {
        strButtons += "Right ";
    }
    if(pe->buttons() & Qt::MidButton) {
        strButtons += "Middle";
    }
    return strButtons;
}
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
#include "MouseObserver.h"
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication  app(argc, argv);
    MouseObserver wgt;     // створення обєкта

    wgt.resize(250, 130);
    wgt.show();

    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

>**./MouseEvent**

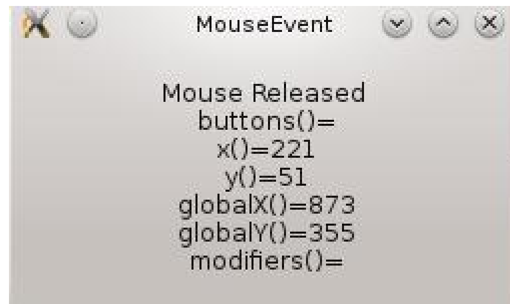

Рисунок 2 – Віджет, який отримує події миші

6.1. Класс QwheelEvent

Об’єкт класу `QWheelEvent` містить інформацію про події, викликані колесом миші. Об’єкт подія передається в метод `wheelEvent()` і містить інформацію про кут і напрям повернення колеса, а також про позицію вказівника миші, статусі кнопок миші і деяких клавіш клавіатури. Поряд з методами `buttons()`, `pos()` і `globalPos()`, які повністю ідентичні методам класу подія `QMouseEvent`, в класі `QWheelEvent` є метод `delta()`, за допомогою якого можна взнати кут повернення колеса миші (позитивне значення – колесо було повернуто від себе, а негативне значення – на себе).

6.2. Методи enterEvent() і leaveEvent().

Ці методи викликаються у тому випадку, коли вказівник миші попадає або покидає область віджета. Їх можна перевизначити, наприклад, у тому випадку, якщо потрібно змінити зовнішній вид віджета. Метод `enterEvent()` отримує об’єкт подію типу `QEvent` і викликається кожний раз, коли вказівник миші входить в область віджета. Метод `leaveEvent()` отримує

об'єкт подію типу `QEvent` і викликається, коли вказівник миші виходить за межі області віджета.

6.3. Події таймера. Клас QTimerEvent

Об'єкт класу `QTimerEvent` містить інформацію про події, ініціалізовані таймером. Цей об'єкт передається у метод обробки подій `timerEvent()`. Об'єкт події містить ідентифікаційний номер таймера. Наприклад, для класу, успадкованого від класу `QWidget`, метод оброки цієї події може виглядати так:

```
void MyClass::timerEvent(QTimerEvent* e) {
  if (event->timerId() == myTimerId) {
     // Виконати які-небудь дії
  }
  else {
      QWidget::timerEvent(e); // Передати подію далі
  }
}
```

6.4. Інші події

Клас QChildEvent. Ця подія відбувається у момент створення або вилучення об'єкта-потомка. Об'єкт подія передається у метод `childEvent()`, який визначений у класі `QObject`. Викликом методу `QChildEvent::child()` можна отримати вказівник на цей об'єкт. За допомогою методів `QChildEvent::added()` і `QChildEvent::removed()` можна взнати про створення або вилучення об'єкта-потомка.

Клас QCloseEvent Подія класу `QCloseEvent` створюється при закритті вікна віджета. Вона може бути викликана користувачем або методом `QWidget::close()`. Об'єкт класу `QCloseEvent` передається у метод `closeEvent()`, в якому можна запитати користувача, чи дійсно він хоче закрити застосування. Це має зміст у тих випадках, коли користувач не зберіг свої дані.

За допомогою методів `accept()` і `ignore()` встановлюється прапор, повідомляючий про згоду користувача закрити вікно. Виклик `accept()` приведе до того, що після повернення з цього методу вікно буде сховане методом `hide()`. Виклик `ignore()` залишить вікно без змін.

Клас QHideEvent. Дана подія створюється при натисканні користувачем кнопки згортання застосування. Також, ця подія може бути викликана методом `hide()`, яка робить віджет невидимим. Об'єкт подія класу `QHideEvent` передається у метод `hideEvent()`.

Клас QMoveEvent. Подія класу `QMoveEvent` виникає при переміщенні віджету. Для віджетів верхнього рівня це відповідає переміщенню його вікна. Об'єкт подія класу `QMoveEvent` передається у метод `moveEvent()` і містить інформацію про старі і нові координати віджета, які можна отримати викликом методів `pos()` і `oldPos()`.

Клас QShowEvent. Подія генерується при створенні віджету і при виклику метода `show()`. Об'єкт подія `QShowEvent` передається у метод `showEvent()`.

Класс QResizeEvent. Користувач може змінювати розміри вікна за допомогою миші. При цьому створюється об'єкт подія `QResizeEvent`. Об'єкт передається у метод `resizeEvent()` і містить інформацію про старі і нові розміри віджета, які можна отримати викликом методів `size()` і `oldSize()`. Реакції на зміну розмірів віджета можуть бути наступними: перемалювання вмісту вікна; зміна розмірів віджетів-потомків.

Приклад 2. Перезапис методу `resizeEvent()`.

```
// ======================================================================
//  main.cpp
// ======================================================================
#include <QtGui>
// ======================================================================
class ResizeObserver : public QLabel {
```

```
public:
    ResizeObserver(QWidget* pwgt = 0) : QLabel(pwgt)
    {
        setAlignment(Qt::AlignCenter);
    }

protected:
    virtual void resizeEvent(QResizeEvent* pe)
    {
        setText(QString("Resized")
                + "\n width()=" + QString::number(pe->size().width())
                + "\n height()=" + QString::number(pe->size().height())
               );
    }
};

// ----------------------------------------------------------------------
int main(int argc, char** argv) {
    QApplication   app(argc, argv);
    ResizeObserver wgt;
    wgt.resize(250, 130);
    wgt.show();
    return app.exec();
}
```

> **qmake -project; qmake; make**

Запуск на виконання:

>**./ResizeEvent**

Рисунок 3 – Перевизначення методу resizeEvent()

Запитання.

1. Ієрархія класів подій.
2. Події клавіатури.
3. Зміна фокуса. Оновлення контексту малювання. Класи `QFocusEvent, QPaintEvent.`
4. Події миші. Клас `QMouseEvent`, методи.
5. Інші події вікна. Класи `QChildEvent, QCloseEvent, QHideEvent, QMoveEvent, QShowEvent, QResizeEvent.`

Завдання.

1. Написати програму, як обробляє події таймера і видає повідомлення “Привіт” кожну хвилину.
2. Написати програму, яка при збільшенні розмірів вікна замальовує фон вікна світлішими кольорами, а при зменшенні – темнішими.

Лабораторна робота № 10

10. Qt Designer. Швидке розроблення прототипів

Мета роботи: отримання практичних навиків роботи з Qt designer.

Qt Designer – це засіб швидкої розробки застосувань (Rapid Application Development, RAD). Він дозволяє швидко створювати прототипи застосувань, які базуються на діалогових вікнах, а також можуть мати головне вікно, меню, рядок стану і панель інструментів. Створені в програмі Qt Designer файли описання інтерфейсу можна конвертувати в початковий код на мові C++. Крім віджетів, які містяться в бібліотеці Qt, Qt Designer може доповнюватися віджетами, створеними самим розробником.

1. Створення нової форми

Вікно програми Qt Designer містить меню і панелі інструментів. Панель інструментів надає загальні операції для редагування форми. Вікно Qt Designer містить наступні вікна:

• **Widget Box** (об'єкти)– віджети згруповані у групи.

• **Object Inscpector** (редактор властивостей) – список використовуваних віджетів.

• **Property Editor** (Редактор сигналів і слотів) – вікно редагування сигналів і слотів.

• **Resource Editor** (Редактор ресурсів) – вікно для створення нових або завантаження існуючих ресурсів.

• **Action Editor** (Редактор дій) – вікно для створення, вилучення і керування діями команд створюваної форми.

Після запуску Qt Designer появиться нове вікно **New Form**, у якому буде запропоновано на вибір один із шаблонів форми:

• **Dialog with button**;

• **Main Window**;

• **Widget**.

При виборі Widget буде створено новий віджет вікна. Qt Designer підтримує наступні режими редагування віджетів форми:

• **Edit Widgets** (Редагування віджетів);

• **Edit Signals/Slots** (Редагування сигналів/слотів);

• **Edit Buddies** (Редагування доручень виду відмет-надпис);

• **Edit Tab Order** (Редагування порядку слідування табулятора).

• **Property Editor** (Редактор властивостей). В полі objectName (імя об'єкту) рекомендується задати ім'я `MyForm`, яке буде використане при з'єднанні з сигналами і слотами інших компонентів. При обробленні утилітою `uic` файлу `MyForm.ui` буде створено код на мові C++, в якому ім'я форми буде використано для імені класу – `Ui_MyForm`, а імена розміщених на ній елементів – для назв атрибутів цього класу.

2. Добавлення віджетів

Віджети добавляються у режимі **Edit/Edit Widgets**. Потрібний віджет вибирається з вікна **Widget Box** і перетягується в область форми. З вкладки **Button** перетягуються два віджети **PushButton** (Кнопка). З вкладки **InputWidget** перетягується **Horizontal Slider** (Горизонтальний повзунок), а з вкладки **Display Widget** – **LCD Number** (Електронний індикатор).

Розміщені на формі віджеті появляться у вікні **Object Inspector**. Клацнути у вікні **Object Inspector** лівою клавішею миші на одній кнопці. Ввести в полі **text** вікна **Property Editor** текст **"&Reset"**, а в полі **object name** – **"m_pcmdReset"**. Таку саму операцію повторити з іншою

кнопкою, але в полі **text** ввести текст **“&Quit”**, а в полі **object name** – **“m_pcmdQuit”**. Для віджету **LCD Number** у полі **object name** ввести **“m_plcd”**, а для **Horizontal Slider – “m_psld”**.

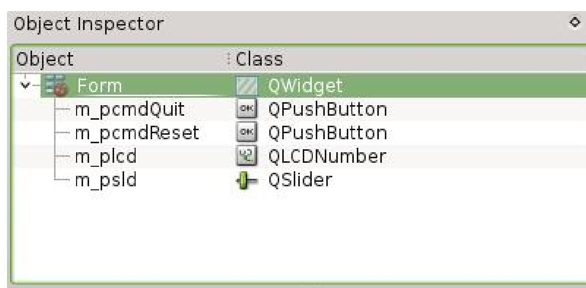

Рисунок 1 – Вікно Object Inspector

3. Компонування (lay out)

Після добавлення всіх віджетів необхідно їх розмістити. Для цього, натиснути ліву кнопку миші, і перемістити вказівник так, щоб рамка охопила повзунок і електронний індикатор. Якщо відпустити ліву клавішу миші, то ці віджети будуть виділені. При натисканні на виділених віджетах правої клавіші, появиться контекстне меню в якому вибрати команду **Lay out | Lay out Vertically** (розміщення по вертикалі). Те ж саме повторити з кнопками **Reset** і **Quit**. Під саму нижню кнопку помістити із закладки **Spacers** віджет **Vertical Spacer** (Вертикальний заповнювач).

Після цього вказати діалоговому вікну розмістити всі елементи горизонтально. Для цього клацнути на порожній області вікна вказівником миші, натиснути праву кнопку і видрати з контекстного меню **Lay out | Lay out Horizontally.** Вікно зміниться і матиме наступний вигляд.

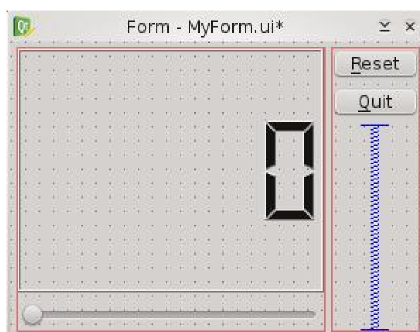

Рисунок 2 – Горизонтальне розміщення вертикальних груп елементів

4. Порядок слідування табулятора

Порядок слідування табулятора потрібен для навігації по вікну за допомогою клавіатури. Натискання на клавішу <Tab> переміщає фокус від одного віджету до іншого. Для визначення порядку слідування табулятора потрібно встановити відповідний режим командою **Edit | Edit Tab Order** (Редагування | Порядок слідування). Натискаючи вказівником миші на віджети, виділені голубим кольором, можна міняти порядок слідування табулятора.

5. Сигнали і слоти

Для з’єднання сигналів одного віджету з сигналами іншого потрібно встановити відповідний режим командою меню **Edit | Edit Signals/Slots**. Для з’єднання потрібно затримати вказівник миші на потрібному елементі, поки він не буде виділений, а потім натиснути ліву

кнопку миші і перемістити до іншого елементу, з яким має бути з'єднання. При цьому появиться червона стрілка, яка вказує на елемент з'єднання. Після відпускання клавіші миші появиться діалогове вікно **Configure Connection**, яке пропонує вибрати сигнали і слоти для з'єднуваних віджетів.

Так для з'єднання віджетів повзунка і електронного індикатора потрібно вказати наступну пару сигнал-слот: `sliderMoved(int) – display(int)`.

Щоб закрити вікно віджету натисканням кнопки `Quit`, необхідно з'єднати цю кнопку із смугою форми де міститься її ім'я і задати пару сигнал-слот: `clicked() – close()`.

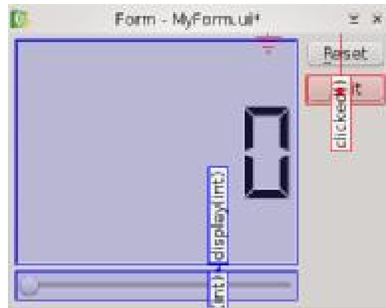

Рисунок 3 – Форма із вибраним з'єднанням слотів і сигналів

Перевірити створений віджет у дії можна за допомогою команди меню **Form/Preview.** Переміщення повзунка збільшує значення електронного індикатора, а натискання кнопки `Quit` закриває форму.

Рисунок 4 – Віджет у дії

6. Використання у формах власних віджетів

Іноді необхідно інтегрувати у форму віджети, які були створені в окремому ui-файлі або написані на C++. У такому випадку потрібно помістити у форму любий з віджетів, виділити його на формі, викликати контекстне меню і вибрати пункт **Promote to**…. В діалоговому вікні яке відкриється, потрібно вписати ім'я віджета, який потрібно інтегрувати у форму.

7. Використання форм у проектах

Форми у проектах можна використовувати наступними методами:
- прямим;
- успадкуванням;
- множинним успадкуванням;
- динамічним завантаженням.

У прямому методі потрібно створити віджет і встановити у методі `Ui::Form::setupUi()`

Приклад 1. `main.cpp` – прямий спосіб використання форми.

```
#include "ui_MyForm.h"
#include <QtGui>
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QWidget* form = new QWidget;
    Ui::MyForm ui;
    ui.setupUi(form);
    form->show();
    return app.exec();
}
>qmake -project
>qmake
>make
```

Недолік такого подходу в тому, що не можна ініціалізувати дані і доповнювати кодом використовуваний клас. Якщо у формі є, наприклад, віджети, які потрібно з'єднувати зі слотами, реалізованими у інших класах, то даний спосіб цього не забезпечує.

Другий спосіб реалізується успадкуванням. У цьому випадку успадковується клас від класу, за базу якого була взята розроблювана форма (`QWidget`). Форма `Ui_MyForm` використовується як атрибут нового класу і встановлюється в конструкторі методом `Ui::MyForm::setupUi()`.

Якщо у формі потрібно добавити нові елементи, то вони добавляються програмою QT Designer, а слоти реалізуються у програмі. Таким чином дизайн графічного інтерфейсу відокремлений від програмної реалізації.

Приклад 2. `MyForm1.cpp` – успадкування.

```
#include "ui_MyForm.h"

class MyForm : public QWidget {
Q_OBJECT
private:
   Ui::MyForm m_ui;
public:
    MyForm(QWidget* pwgt = 0) : QWidget(pwgt)
    {
       m_ui.setupUi(this);
       connect(m_ui.m_pcmdReset, SIGNAL(clicked()), SLOT(slotReset()));
    }
public slots:
    void slotReset()
    {
       m_ui.m_psld->setValue(0);
       m_ui.m_plcd->display(0);
    }
};
#include "main.cpp"
     >qmake -project
     >qmake
     >make
```

Третій спосіб множинного успадкування, подібний до другого, але дозволяє напряму звертатися до всіх віджетів форми.

Приклад 3. `MyForm2.cpp` – множинне успадкування.

```
#include "ui_MyForm.h"
class MyForm : public QWidget, public Ui::MyForm {
    Q_OBJECT
public:
    MyForm(QWidget* pwgt = 0) : QWidget(pwgt)
    {
       setupUi(this);
       connect(m_pcmdReset, SIGNAL(clicked()), SLOT(slotReset()));
```

```
    }
public slots:
    void slotReset()
    {
        m_psld->setValue(0);
        m_plcd->display(0);
    }
};
#include "main.cpp"
     >qmake -project
     >qmake
     >make
```

8. Компіляція проектів з формами

Перед компіляцією проектів потрібно створити проектний файл (*.pro).

```
TEMPLATE = app
HEADERS += MyForm.h
FORMS += MyForm.ui
SOURCES += main.cpp
```

Після цього у каталогах, які містять проектний файл, файл форми (MyForm.ui) та інші файли команди

```
qmake
make
```

В процесі компіляції з `ui`-файлу буде створено `h`-файл з префіксом `ui_`. Після компіляції буде створена виконувана програма.

Запитання.

1. Призначення програми Qt Designer. Меню і панелі інструментів.
2. Створення нової форми і добавлення віджетів.
3. Компонування віджетів. Порядок слідування табулятора.
4. Сигнали і слоти. Використання власних віджетів.
5. Використанням форм у проектах.
6. Компіляція проектів з формами.

Лабораторна робота № 11

11. Інтернаціоналізація застосувань

Мета роботи: отримання практичних навиків з інтернаціоналізації застосувань.

1. Створення нової форми

Інтернаціоналізація – це можливість вибору мови інтерфейсу програми.

Для підтримки інтернаціоналізації в `Qt`-застосуваннях потрібно виконати наступні кроки:

1. Підготовити застосування до інтернаціоналізації.
2. Запустити утиліту `lupdate`.
3. Перевести команди створюваної програми (в цьому може допомогти `Qt Linguist`).
4. Запустити утиліту `lrelease` для генерації двійкових файлів перекладів, які надалі будуть завантажуватися в об'єкт класу `QTranslator`.

2. Підготовка застосування до інтернаціоналізації

Для підготовки програми до інтернаціоналізації потрібно всі стрічки, які будуть виводитися на екран, помістити у статичний метод `tr()`, визначений у класі `QObject()`.

Текст переданий у цей метод є джерелом перекладу. Звичайно, як джерело перекладу використовується текст на англійській мові. Першим параметром у метод `tr()` передається текстова стрічка. Другий параметр – коментар, який використовується для надання додаткової інформації перекладачу.

```
setText(tr("Yes"));
QLabel lbl(tr("Location", "On the map"));
```

Метод `tr()` також можна використовувати у підставленнях з методом `String::arg()`:

```
setText(tr("User Name: %1").arg(strName));
```

У функцію `tr()` можна поміщати не тільки текст, призначений для відображення на екрані, але і позначення валюти, комбінації "гарячих" клавіш і інші відмінні особливості, характерні для кожної країни. Наприклад: `QKeySequence keyseq(tr("CTRL+L"));`

Для передачі чисел в однині або множині призначений третій параметр:

```
int n = getDays();
QLabel lbl(tr("day(s):", "Plural or singular", n));
```

Інтернаціоналізація застосувань, це більше ніж просто переклад текстів з однієї мови на іншу, вона включає в себе також складний процес локалізації. Під локалізації попадає адаптація форматів дат, часу і цифр, наприклад:

```
QLocale english(QLocale::English, QLocale::UnitedKingdom);
QString str = english.toString(2876.56); // str = 2,876.56
QLocale german(QLocale::German, QLocale::Germany);
str = german.toString(2876.56); //str = 2.876,56
```

Для деяких мов (арабська) потрібно змінити напрямок написання. А це значить, що після встановлення перекладу потрібну буде ще перевернути розміщення віджетів у зворотну сторону. Це можна зробити викликом методу з глобального об'єкта застосування наступним чином:

```
qApp->setLayoutDirection(Qt::RightToLeft);
```

3. Утиліта lupdate

Після передачі в метод `tr()` усіх потрібних рядків тексту можна приступати до створення файлів перекладу. Для цього необхідно скористатися спеціальною утилітою `lupdate`. У даному випадку призначення метода `tr()` зводиться до вказівки утиліті тексту в програмному коді, який потрібно перекласти. Із стрічок, переданих в метод `tr()`, будуть створені окремі TS-файли (Translation Source, джерело перекладу) – файли перекладів. Для створення файлів українського і німецького перекладів і *локалізації* програми, потрібно виконати наступний виклик:

```
lupdate main.cpp -ts main_ru.ts main_de.ts
```

Приклад програми і TS-файлів:

```
#include <QtGui>
int main(int argc, char** argv)
{
QApplication app(argc, argv);
QLabel lbl(QObject::tr("Hello"));
lbl.show();
return app.exec();
}

<!DOCTYPE TS><TS>
<context>
  <name>QObject</name>
  <message>
    <source>Hello</source>
    <translation type="unfinished"></translation>
  </message>
</context>
</TS>
```

В цей файл можна вручну внести переклад. Для цього в тегу `translation` потрібно вилучити атрибут `type` разом з його значенням і добавити переклад в текстову зону тега. Наприклад, для українського варіанта тег `translation` матиме такий вигляд: `<translation>Привіт</translation>`. Підготовлені файли перекладів необхідно включити в файл проекту. Для цього в нього потрібно внести наступний рядок:

```
TRANSLATIONS = main_ua.ts main_de.ts.
```

Для перекладу стрічок можна використати програму `Qt Linguist`. Вона дозволяє завантажувати і перекладати декілька файлів одночасно. Приклад запуску програми `Linguist` і завантаження в неї файлу перекладу:

```
linguist main_ru.ts
```

Після завершення роботи з файлом його необхідно зберегти.

4. Утиліта lrelease

Для конвертування файлів перекладу у завантажувані застосуванням двійкові QM-файли можна скористатися однією з трьох утиліт `lupdate, Qt Linguist и lrelease`. Але краще всього цю операцію виконувати за допомогою утиліти `lrelease`, так як вона забезпечує найкращий коефіцієнт стискання для QM-файлів. Наступна команда створить для кожного файлу перекладу (TS-файлу), вказаного у файлі проекту, свій QM-файл:

```
lrelease myproject.pro
```

Отримані QM-файли передаються в об'єкт класу `QTranslator`. Об'єкт класу `QTranslator` відповідає за переклад текстів з однієї мови на іншу. Цей переклад виконується через QM-файл, завантажений в об'єкт класу `QTranslator`, викликом методу `load()`. Завантажити інші QM-файлів можна в любий момент виконання програми. Наступна програм відображає на екрані повідомлення на українській мові, яке відповідає англійському "Hello".

```
int main(int argc, char** argv)
{
   QApplication app(argc, argv);
   QTranslator translator;
   ptranslator.load("main_ua.qm", ".");
   app.installTranslator(&translator);
   QLabel lbl(QObject::tr("Hello"));
   lbl.show();
   return app.exec();
}
```

В програмі створюється об'єкт класу `QTranslator`. Викликом методу `load()` у нього завантажується файл перекладу `main_ua.qm`, вказаний у першому параметрі цього методу. Другий параметр задає каталог, який містить файли перекладів (у даному випадку крапка задає поточний каталог). Виклик методу `installTranslator()` з об'єкта класу `QApplication` застосовує створений об'єкт перекладача до всього застосування. У складних проектах можуть використовуватися стандартні діалогові вікна, контекстні меню і т.п. Всі ці компоненти також повинні бути перекладені в застосуванні. Для цього QM-файли перекладів Qt також необхідно завантажити додатково до або після завантаження власних перекладів. Ці файли знаходяться в каталозі `translations`. Їх завантажувати в застосування можна так: спочатку переклади для Qt, а потім переклади для застосування:

```
QTranslator qtTranslator;
qtTranslator.load(QString(":/translations/qt_") + QLocale::system().name());
qApp->installTranslation(&qtTranslator);
QTranslator appTranslator;
appTranslator.load(QString(":/translations/app_") +
                   QLocale::system().name());
appTanslator.load("main_ru.qm", ".");
qApp->installTranslator(&appTranslator);
```

5. Зміна перекладу в процесі роботи

Є три варіанти зміни перекладу в процесі роботи програми. Перший – самий простий. Після того як користувач вибрав потрібну йому мову, її можна записати в об'єкт налаштувань `QSettings` і попросити запустити програму знову. При новому запуску програми потрібно просто зчитати нову мову з об'єкта налаштувань і завантажити потрібний переклад.

Другий спосіб трохи складніший, і він не вимагає завершення роботи застосування з явним його перезапуском користувачем. Він полягає у тому, що, що саме застосування після зміни мови завантажує його і знищує основне вікно віджета програми і створює нове. Для такої операції потрібний додатковий об'єкт, який буде отримувати від віджета основного вікна сигнал про зміну мови і у відповідь на це знищить старе і створить нове вікно. Самий складний момент полягає у тому, що б користувач не зауважив підміни, а значить, нове вікно має повністю відповідати знищеному вікну і містити ті ж дані. Цього можна досягти з використанням об'єкта налаштувань класу `QSettings`.

```
// Присвоєння стрічкам актуально вибраного перекладу
void MyLabel::changeEvent(QEvent* pe)
{
   if (pe->type() == QEvent::LanguageChange)
   {
      retranslateUi();
   }
 }

void MyLabel::retranslateUi()
{
   setWindowTitle(tr("Current Language"));
   setText(tr("Hello"));
```

```
}
```

При запуску нового об'єкта перекладача буде викликаний метод подія `QWidget::changeEvent()`, з якого буде викликаний метод `retranslateUi()`. У цьому методі відбудеться присвоєння нових стрічок перекладу для заголовка вікна і для тексту надпису.

Для перемикання мов можна в спеціально призначеному для цього віджеті створити окремий метод:

```
void MyProgram::switchLanguage(int n)
{
   QTranslator translator;
   switch (n)
   {
     case UKRAINIAN: translator.load("myprogram_ua.qm",".");
     break;
     case GERMAN: translator.load("myprogram_de.qm",".");
     break;
   }
   qApp->installTranslator(&translator);
}
```

У випадку, якщо користувач не задав мову (наприклад, при першому запуску програми), можна встановити поточну мову у відповідності з локалізацією платформи. Локалізацію можна взнати за допомогою статичного метода `QLocate::system()`. Цей метод повертає об'єкт класу `QLocate`, а виклик його методу `name()` повертає стрічку виду "мова_КРАЇНА".

У відповідності з цим рядком можна буде завантажити необхідний файл перекладу. Наступний фрагмент коду показує як це зробити:

```
QTranslator translator;
QString str = QLocale::system().name();
if (str == "en_US") {
    translator.load("myprogram_us.qm","."); // завантаження перекладу для США
}
else if (str == "de_CH") {
translator.load("myprogram_de.qm","."); // німецький переклад для Швейцарії
}
else if (str == "ua") {
translator.load("myprogram_ru.qm","."); // український переклад
}
qApp->installTranslator(&translator);
```

Запитання.

1. Що таке інтернаціоналізація і локалізація програм.
2. Які кроки потрібно виконати для інтернаціоналізації `Qt` програми.
3. Призначення методу `QObject.tr()`.
4. Призначення утиліти `lupdate`.
5. Призначення програми `Qt Linquist`.
6. Призначення утиліти `lrelease`.
7. Які є методи зміни мови в процесі роботи програми.

Завдання.

1. Запустити програму `Qt Linquist` і виконати переклад стрічки "Good bay" на українську мову з використанням файлу `main_ua.ts`.
2. Написати програму зміни мови Qt програми із знищення старого і створенням нового вікна.
3. Написати програму зміни мови Qt програми із використанням події `QEvent::LanguageChange`.