Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ

А. В. Бородин, А. В. Бородина

СРЕДСТВА РАЗРАБОТКИ ГРАФИЧЕСКИХ ИНТЕРФЕЙСОВ ПОЛЬЗОВАТЕЛЯ

Учебное пособие

Петрозаводск
Издательство ПетрГУ
2012

ББК 32.973
УДК 004
Б833

Печатается по решению редакционно-издательского совета Петрозаводского государственного университета

Издается в рамках реализации комплекса мероприятий Программы стратегического развития ПетрГУ на 2012–2016 г.г.

Р е ц е н з е н т ы:
канд. техн. наук Р. В. Сошкин;
канд. физ.-мат. наук, доцент К. А. Кулаков

Бородин А. В.

Б833 Средства разработки графических интерфейсов пользователя : учебное пособие / А. В. Бородин, А. В. Бородина. — Петрозаводск : Изд-во ПетрГУ, 2012. — 77 с.

ISBN 978-5-8021-1537-4

Учебное пособие предназначено для студентов математического факультета первого курса направлений подготовки «Прикладная математика и информатика», «Информационные системы и технологии».

ББК 32.973
УДК 004

ISBN 978-5-8021-1537-4

Содержание

Введение

Развитие информационных технологий и рост числа пользователей компьютеров предъявляют новые требования к пользовательским интерфейсам. Перед разработчиками программного обеспечения стоит непростая задача обеспечить низкий порог вхождения для широкого круга не обладающих достаточной квалификацией и не прошедших специальную подготовку в использовании ЭВМ лиц за счет упрощения средств диалога с машиной, использования интуитивно понятных графических метафор, унификации интерфейсов.

Таким образом, овладение средствами разработки современных графических интерфейсов пользователя является необходимым элементом подготовки разработчика прикладного программного обеспечения.

Пособие представляет собой обзор основных концепций, лежащих в основе графических интерфейсов, и краткое введение в программирование с использованием фреймворка Qt, предоставляющего набор библиотек и утилит для создания прикладных программ, в том числе, приложений с графическим интерфейсом.

Выбор Qt среди широкого круга альтернатив обусловлен двумя основными причинами. Во-первых, Qt является продуктом с открытым кодом, таким образом его использование не может быть ограничено изменением политики компании-разработчика. Во-вторых, Qt изначально разрабатывался как кроссплатформенный инструмент: написанные с использованием этого фреймворка приложения переносимы на уровне исходного кода между всеми популярными настольными операционными системами, более того, Qt можно использовать и для разработки для мобильных устройств, таких как смартфоны и планшетные компьютеры. Немаловажным является факт наличия в свободном доступе весьма подробной документации разработчика (см. [6]).

Материалы пособия используются в Петрозаводском государственном университете на математическом факультете в модуле «Средства разработки графических интерфейсов» дисциплины «Информатика», читаемой студентам I курса направлений подготовки «Прикладная математика и информатика» и «Информационные системы и технологии».

§1. Графические интерфейсы пользователя

Графическими интерфейсами пользователя (англ. Graphical User Interface, GUI) считают такие, в которых элементы интерфейса представлены в виде визуальных объектов, т. е. графических изображений.

Элементы интерфейса, используя метафоры, несут информацию об их назначении и способе использования, обеспечивая обучение пользователя в процессе работы. Интерфейсы, обладающие этим свойством, принято называть интуитивно понятными.

Ко всем видимым элементам интерфейса обеспечивается произвольный достум посредством широкого спектра устройств ввода — клавиатуры, указателя «мышь», сенсорного экрана и др.

Первые разработки в области графических интерфейсов относятся к семидесятым годам XX века и, как правило, связываются с компьютером Alto, разработанным в исследовательских лабораториях компании XEROX в Пало Альто, Калифорния. Тогда же впервые появляется аббревиатура WIMP (Windows, Icons, Menus, Pointing Device), концептуально характеризующая графический интерфейс пользователя.

Коммерческий успех к компьютерам, операционные системы которых предоставляли графический интерфейс, пришел в восьмидесятые годы XX века и связан с появлением таких разработок, как Star компании Xerox и Lisa компании Apple.

Тогда же были предпринты первые попытки упорядочить разработку графических интерфейсов пользователя, в частности, следует отметить руководство IBM Common User Access Specification.

Одной из первых оконных систем, обеспечивающей стандартные инструменты и протоколы для построения графических интерфейсов, стала X Window System, разрабатываемая с 1984 года и используемая по настоящее время в UNIX-подобных ОС. X Window System обеспечивает базовые функции графической среды: отрисовку и перемещение окон на экране, взаимодействие с устройствами ввода, такими как, например, мышь и клавиатура.

X Window System не определяет деталей интерфейса пользователя — этим занимаются менеджеры окон. С другой стороны, в этой системе предусмотрена сетевая прозрачность: графические приложе-

ния могут выполняться на другой машине в сети, а их интерфейс при этом будет передаваться по сети и отображаться на локальной машине пользователя. В контексте X Window System термины «клиент» и «сервер» имеют непривычное для многих пользователей значение: «сервер» означает локальный дисплей пользователя (дисплейный сервер), а «клиент» — программу, которая этот дисплей использует (она может выполняться на удалённом компьютере).

Архитектура X Window System проиллюстрирована на рис. 1.

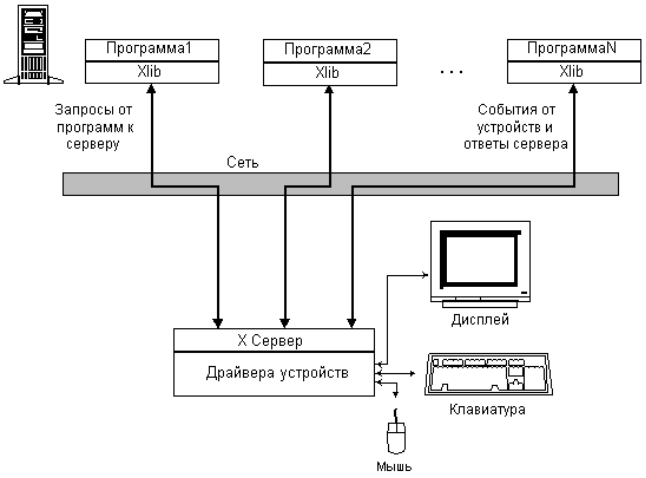

Рис. 1: Архитектура X Window System

X Window System предоставляет программный интерфейс для создания приложений с графическим интерфейсом посредством использования библиотеки XLib. Однако, программирование на уровне XLib слишком многословно и перегружает программиста необходимостью реализовывать вручную даже примитивные детали интерфейса.

Рассмотрим достаточно простую XLib-программу, изображающую на экране простое окно с надписью.

Пример 1.1

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define X 0
#define Y 0
#define WIDTH 400
#define HEIGHT 300
#define WIDTH_MIN 50
#define HEIGHT_MIN 50
#define BORDER_WIDTH 1
#define TITLE "Example"
#define ICON_TITLE "Example"
#define PRG_CLASS "Example"

/* Сообщает оконному менеджеру параметры окна */
static
void SetWindowManagerHints(Display * display,
        char *PClass, char *argv[], int argc,
        Window window, int x, int y,
        int win_wdt, int win_hgt,
        int win_wdt_min, int win_hgt_min,
        char *ptrTtl, char *ptrITtl, Pixmap pixmap)
{
    XSizeHints size_hints;
    XWMHints wm_hints;
    XClassHint class_hint;
    XTextProperty winnm, icnm;

    if (!XStringListToTextProperty (&ptrTtl, 1, &winnm)
      || !XStringListToTextProperty (&ptrITtl, 1, &icnm)) {
        puts ("No memory!\n");
        exit (1);
    }
```

```
    size_hints.flags = PPosition | PSize | PMinSize;
    size_hints.min_width = win_wdt_min;
    size_hints.min_height = win_hgt_min;
    wm_hints.flags = StateHint | IconPixmapHint | InputHint;
    wm_hints.initial_state = NormalState;
    wm_hints.input = True;
    wm_hints.icon_pixmap = pixmap;
    class_hint.res_name = argv[0];
    class_hint.res_class = PClass;

    XSetWMProperties (display, window, &winnm,
                      &icnm, argv, argc, &size_hints,
                      &wm_hints, &class_hint);
}

/* Обработчик ошибки */
int MyXlibIOErrorHandler(Display *d)
{
    XCloseDisplay(d);
    exit(0);
}

/* Основная программа */
int main(int argc, char *argv[])
{
    Display *display;
    int ScreenNumber;
    GC gc;
    XEvent report;
    Window window;

    if ((display = XOpenDisplay(NULL)) == NULL) {
        puts ("Can not connect to the X server!\n");
        exit (1);
    }

    ScreenNumber = DefaultScreen(display);
```

```
    window = XCreateSimpleWindow(display,
        RootWindow(display, ScreenNumber),
        X, Y, WIDTH, HEIGHT, BORDER_WIDTH,
        BlackPixel(display, ScreenNumber),
        WhitePixel(display, ScreenNumber));

    SetWindowManagerHints (display, PRG_CLASS, argv, argc,
        window, X, Y, WIDTH, HEIGHT, WIDTH_MIN,
        HEIGHT_MIN, TITLE, ICON_TITLE, 0);

    XSelectInput (display, window, ExposureMask
        | StructureNotifyMask);

    XMapWindow (display, window);
    XSetIOErrorHandler (&MyXlibIOErrorHandler);

    while (1) {
        XNextEvent (display, &report);
        switch (report.type) {
          case Expose:
            if (report.xexpose.count != 0) break;
            gc = XCreateGC(display, window, 0, NULL);
            XSetForeground(display, gc,
                BlackPixel(display, 0));
            XDrawString (display, window, gc, 20, 50,
                "Hello, world!", strlen ("Hello, world!"));
            XFreeGC (display, gc);
            XFlush (display);
            break;
          default:
            fprintf (stderr, "%d\n", report.type);
        }
    }
    return 0;
}
```

Используя XLib, программист полностью контролирует свойства оконного приложения, именно этим объясняется большое количество параметров в вызовах функций. Также программист самостоятельно выполняет проверку наступления тех или иных событий, организуя бесконечный цикл опроса обработки. На полотне окна программист также самостоятельно вынужден отрисовывать необходимые элементы.

Из приведенного примера можно извлечь следующие выводы.

- Поскольку любая программа с графическим интерфейсом является событийно-ориентированной, то есть большую часть времени проводит в состоянии «сна», выполняя те или иные функции лишь в ответ на активность пользователя, то она, так или иначе, будет включать цикл обработки событий, подобный приведенному в примере. Следовательно, необходимости в включении этого цикла вручную в каждой программе нет — его можно реализовать в виде отдельной функции библиотеки, предоставляющей программные средства создания графического интерфейса. При этом код обработки событий оформляется в виде отдельных функций, регистрируемых до входа в цикл обработки событий. Такие функции называют функциями обратного вызова (callback), так как в программе отсутствуют явные вызовы этих функций, они вызываются из функции, реализующей абстракцию основного цикла при наступлении соответствующего события.

- Большая часть параметров оконного приложения либо контролируется оконным менеджером, либо имеет стандартные значения (например, цветовая гамма может определяться темой оформления). Поэтому программный интерфейс можно существенно упростить.

Исходя из представленных соображений, над XLib был разработан набор высокоуровневых библиотек Xt (так называемый «тулкит»), скрывающий сложность XLib за оберточными вызовами. Рассмотрим пример Xt-программы, предоставляющий ту же функциональность, что и пример 1.1.

Пример 1.2

```
#include <X11/X.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Core.h>

void DrawHelloString (Widget prWidget, XtPointer pData,
    XEvent *prEvent, Boolean *pbContinue)
{
    Display *prDisplay = XtDisplay (prWidget);
    Window nWindow = XtWindow (prWidget);
    GC prGC;

    if (prEvent->type == Expose) {
       prGC = XCreateGC (prDisplay, nWindow, 0, NULL);
       XDrawString (prDisplay, nWindow, prGC, 10, 50,
               "Hello, world!", strlen ("Hello, world!"));
       XFreeGC (prDisplay, prGC);
    }
}

void main (int argc, char **argv)
{
    Arg        args[2];
    Widget toplevel, prCoreWidget;

    toplevel = XtInitialize(argv[0], "GUI Example",
                            NULL, 0, &argc, argv);
    prCoreWidget = XtCreateWidget ("Core", widgetClass,
                                   toplevel, NULL, 0);

    XtSetArg(args[0], XtNwidth, 400);
    XtSetArg(args[1], XtNheight, 300);
    XtSetValues(prCoreWidget, args, 2);

    XtManageChild (prCoreWidget);
    XtAddEventHandler(prCoreWidget, ExposureMask,
                      False, DrawHelloString, NULL);
```

```
    XtRealizeWidget(toplevel);
    XtMainLoop();
}
```

Типовой цикл обработки событий заменен единственным вызовом XtMainLoop(). При этом обработчики событий, реализованные в виде функций обратного вызова, связываются с сигналами событий (регистрируются) заранее посредством вызова XtAddEventHandler().

Также в Xt введено понятие виджета (widget = visual gadget) — типового элемента графического интерфейса. Однако, отрисовка строки по-прежнему осуществляется с помощью низкоуровневого XLib-вызова XDrawString().

Современные инструменты разработки графических интерфейсов также реализуют абстракцию основного цикла обработки событий и предоставляют разработчику богатую библиотеку виджетов. Следующий пример демонстрирует использование средств библиотеки GTK+.

Пример 1.3

```
#include <gtk/gtk.h>

int main (int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *label;

    /* Инициализируем подсистему GTK+ */
    gtk_init(&argc, &argv);

    /* Создаем окно */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* Привязываем обработчик нажатия кнопки закрытия окна */
    g_signal_connect(G_OBJECT (window), "destroy",
                     G_CALLBACK (gtk_main_quit), NULL);

        /* Создаем кнопку с надписью */
    label = gtk_label_new("Hello World");
```

```
    /* Помещаем кнопку в окно (контейнер) */
    gtk_container_add(GTK_CONTAINER (window), label);

    /* Отрисовываем окно и все принадлежащие ему объекты */
    gtk_widget_show_all(window);

    /* Основной цикл обработки событий */
    gtk_main();

    return 0;
}
```

§2. Семейство библиотек Qt

2.1 Первая Qt-программа

По сложившейся традиции руководств по программированию, обзор возможностей семейства библиотек Qt начнем с программы, выводящей на экран приветствие «Hello, world!».

Пример 2.1

```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    // Создаем экземпляр приложения
    QApplication a(argc, argv);

    // Создаем надпись
    QLabel label("Hello World");

    // Отображаем надпись на экране
    label.show();

    // Основной цикл обработки событий
    return a.exec();
}
```

Текст программы содержит включения файлов заголовков с определениями необходимых классов (QApplication и QLabel).В Qt на каждый класс приходится в точности один заголовочный файл.

Выполнение программы начинается с создания объекта класса QApplication, представляющего абстракцию графического приложения. Ни одна графическая программа не обходится без QApplication, так как именно в этом классе реализован цикл обработки событий, определяющий основной алгоритм работы графического приложения. Мы передаем управление в цикл обработки событий вызовом метода exec() класса QApplication. Именно в exec() программа проводит

большую часть времени работы, ожидая пользовательской реакции или другого события.

В конструктор QApplication мы передаем аргументы командной строки. Qt-приложение способно распознавать некоторые параметры, заданные в командной строке и адаптировать поведение в соответствии с ними.

После создания объекта QApplication создается объект QLabel, представляющий простую текстовую надпись. Текст для отображения передается через конструктор класса.

Метод show() заставляет Qt-приложение отрисовать надпись на экране, при этом, так как надпись не имеет родительского объекта, то сама формирует окно.

Цикл обработки событий прекращается при выполнении метода quit() класса QApplication, например, если пользователь закрывает окно. В этом случае приложение завершается.

Так как Qt поддерживается на многих различных платформах, отличающихся организацией процесса сборки, разработчиками Qt предложен специальный проектно-ориентированный инструмент сборки — qmake.

Программа qmake генерирует файлы сборки Makefile для каждой конкретной платформы на основе специального файла проекта (*.pro), который описывает приложение в независимой от операционной системы и среды программирования форме.

Сохраним нашу первую программу под именем main.cpp в каталоге myFirstQtProject и сгенерируем в нем шаблон файла проекта с помощью той же программы qmake:

Пример 2.2

```
user@linux:~/myFirstQtProject> qmake -project
```

В результате будет построен шаблон файла проекта:

Пример 2.3

```
#myFirstQtProject/myFirstQtProject.pro
```

```
#####################################
# Automatically generated by qmake
#####################################

TEMPLATE = app
CONFIG -= moc
DEPENDPATH += .
INCLUDEPATH += .

# Input
SOURCES += main.cpp
```

Файл проекта состоит из директив, в данном примере директива TEMPLATE указывает, что проект представляет собой приложение (app), а SOURCES содержит перечень файлов исходного кода проекта (в нашем примере только main.cpp).

Повторный вызов qmake (теперь без параметров) создаст зависимый от платформы файл сборки (Makefile), который можно использовать для получения исполняемого файла:

Пример 2.4

```
user@linux:~/myFirstQtProject> qmake
user@linux:~/myFirstQtProject> make
user@linux:~/myFirstQtProject> ./myFirstQtProject
```

2.2 Обзор средств Qt

2.2.1 Виджеты и конструирование интерфейсов

Основу библиотеки графического интерфейса составляют виджеты — классы, представляющие визуальные объекты. Виджеты образуют иерархию классов. Базовым классом для всех виджетов в Qt является QWidget. При этом имеется более шестидесяти производных от QWidget классов, реализующих типовые элементы управления. Дополнительно программист имеет возможность создания собственных виджетов, расширяя функциональность элементов стандартного набора.

Визуальная структура интерфейсов формируется посредством контейнерных классов, позволяющих организовывать виджеты в иерархии. Таким образом, виджеты могут содержать в себе другие виджеты.

Харахтерными особенностями виджетов являются следующие:

- виджет занимает прямоугольную область экрана;
- виджет может получать сообщения о событиях, например, от устройств ввода;
- виджет может отправлять сигналы об изменении состояния.

Все виджеты имеют набор общих свойств, унаследованных от базового класса QWidget:

- enabled — возможна ли реакция на ввод пользователя;
- visible — видим ли виджет (или скрыт).

Действие этих свойств распространяется на вложенные виджеты.

В примере 2.1 интерфейс представляен единственным виджетом QLabel, поэтому проблем с его расположением нет. Однако интерфейс приложения может быть сконструирован из десятков виджетов, и проблема их взаимного расположения окажется весьма серьезной.

Qt предоставляет так называемые средства компоновки, облегчающие процесс взаимной расстановки виджетов в пространстве окна. Следующий пример организовывает две надписи вертикально:

Пример 2.5

```
#include <QApplication>
#include <QVBoxLayout>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    // Создаем окно
    QWidget window;
```

```
    // Создаем основной компоновщик окна -
    // вертикальный компоновщик
    QVBoxLayout* mainLayout = new QVBoxLayout(&window);

    // Создаем две надписи
    QLabel* label1 = new QLabel("One");
    QLabel* label2 = new QLabel("Two");

    // Организуем надписи с помощью компоновщика
    mainLayout->addWidget(label1);
    mainLayout->addWidget(label2);

    // Отображаем окно
    window.show();

    return a.exec();
}
```

Подробнее вопросы компоновки и виды компоновщиков рассмотрены в четвертом параграфе. К рассмотренному примеру следует заметить, что только для окна был вызван метод show(), надписи, вложенные в окно, автоматически становятся для него «дочерними» объектами и визуализируются автоматически при отображении родителя.

2.2.2 Класс QObject и метаобъектный компилятор

Базовым классом большинства классов семейства библиотек Qt, в частности, всех виджетов, является QObject. Исключениями являются только классы, для которых важна легковесность (например, графические примитивы и контейнеры данных). Именно QObject реализует большую часть возможностей, характерных для Qt: обработку событий, сигналы и слоты, свойства объектов, управление памятью.

Каждый унаследованный от QObject имеет так называемый метаобъект, содержащий информацию об имени класса, связях и т. д. Таким образом Qt вводит механизмы интроспекции в язык C++.

Метаданные собираются на этапе трансляции специальной программой — метаобъектным компилятором moc. Программа moc явля-

ется препроцессором, обрабатывающим файлы заголовков проекта и конструирующим вспомогательные файлы исходного кода (moc_*.cpp), также включаемые в проект. При использовании системы qmake генерация этих файлов осуществляется прозрачно для программиста.

Программа moc извлекает из файлов заголовков специальные макросы (например, в следующем примере, Q_OBJECT, Q_CLASSINFO), а также ключевые слова Qt, не являющиеся элементами языка C++: signals, slots, emit и др. Обработанные файлы содержат только конструкции C++ и могут быть скомпилированы обычным компилятором C++.

Макрос Q_OBJECT должен присутствовать в любом классе, наследуемом от QObject.

Пример 2.6

```
class MyClass : public QObject
{
  Q_OBJECT
  Q_CLASSINFO("author", "John Doe")
  public:
    MyClass(const Foo &foo, QObject *parent=0);
    Foo foo() const;
  public slots:
    void setFoo( const Foo &foo );
  signals:
    void fooChanged( Foo );
  private:
    Foo m_foo;
};
```

Класс QObject позволяет создавать свойства с методами чтения и записи. Свойство не просто представляет некоторое поле класса, предназначенное для хранения значения. В результате установки значения свойства визуальный объект, возможно должен быть модифицирован (например, при изменении цвета). Поэтому свойству соответствует пара методов чтения и установки значения, выполняющие и соответствующий дополнительный код — так называемые «геттер» и «сеттер». В соответствии с соглашениями имя геттер-метода должно

совпадать с именем свойства. Имя сеттер-метода начинается со слова set, за которым следует имя свойства с заглавной буквы.

Пример 2.7

```
class QLabel : public QFrame
{
  Q_OBJECT
  // Создаем свойство text с геттером text
  // и сеттером setText
  Q_PROPERTY(QString text READ text WRITE setText)
  public:
    QString text() const;
  public slots:
    void setText(const QString &);
};
```

В примере 2.5 объект класса QWidget, представляющий окно, создан на стеке, а под компоновщик и надписи память выделена динамически. Это не случайно. Вообще говоря, C++ не предусматривает автоматического управления памятью, однако в Qt некоторые подобные возможности введены. Объекты, наследующие от класса QObject (в частности, виджеты) могут организовываться в древовидные иерархии, имея родителя (parent object) и дочерних объектов (child objects). Если такой объект уничтожается (вызывается деструктор), то Qt автоматически уничтожает все дочерние объекты. Те, в свою очередь, уничтожают свои дочерние объекты и так далее. Это освобождает программиста от необходимости отслеживать и освобождать занятую дочерними объектами память вручную.

Однако для того, чтобы автоматическое управление памятью могло быть реализовано, память под дочерние объекты должна быть выделена на куче динамической памяти.

Любое нетривиальное графическое приложение должно иметь возможность отреагировать на пользовательские действия, а значит должна существовать система коммуникации между объектами. В отличие от многих инструментов создания графических интерфейсов, в Qt не используются функции обратного вызова для обеспечения реакции

на события, а применяется более совершенная технология сигналов и слотов.

И сигнал, и слот являются методами класса, за тем исключением, что тело сигнала программист не реализует, его создает метаобъектный компилятор. Тип возвращаемого значения сигнала — void. Слот может иметь отличный от void тип возвращаемого значения, но использовать возвращаемое значение можно только, если слот вызывается как обычная функция, а не по сигналу. Сигналы в определении класса объявляются в секции signals, слоты — в секции slots. Отправить сигнал из кода можно с помощью ключевого слова emit.

Сигнатуры связываемых сигнала и слота должны либо совпадать, либо в сигнале может присутствовать часть дополнительных параметров, которые будут проигнорировать в слоте. В общем случае верно правило: «Qt может игнорировать часть данных, но не может получить данные из ничего».

В следующем примере сигнал, отправляемый кнопкой в момент клика, связывается с стандартным слотом quit() объекта QApplication.

Пример 2.8

```
#include <QApplication>
#include <QPushButton>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    // Создаем и отрисовываем кнопку с надписью "Quit"
    QPushButton button("Quit");
    button.show();

    // Связываем сигнал clicked() и слот quit()
    QObject::connect(&button, SIGNAL(clicked()),
                     &a, SLOT(quit()));

    return a.exec();
}
```

Статическая функция connect() класса QObject выполняет связывание сигнала и слота. Сигнал — функция, уведомляющая о наступлении события, слот — функция, получающая этот сигнал и корректно обрабатывающая его. Функция connect() получает четыре аргумента: объект-отправитель, сигнал, объект-получатель, слот. При связывании необходимо использовать макросы SIGNAL() и SLOT().

В примере 2.8 никакой дополнительной информации из сигнала в слот не передается. Рассмотрим более реальный пример. Пусть интерфейс содержит три виджета: QLabel, QSpinBox и QSlider. Последние два обеспечивают просто различные способы ввода целых чисел. Свяжем эти виджеты друг с другом таким образом, чтобы все они отображали одно и то же число.

Пример 2.9

```
#include <QApplication>
#include <QVBoxLayout>
#include <QLabel>
#include <QSpinBox>
#include <QSlider>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    // Создаем элементы интерфейса
    QWidget window;
    QVBoxLayout* mainLayout = new QVBoxLayout(&window);
    QLabel* label = new QLabel("0");
    QSpinBox* spinBox = new QSpinBox;
    QSlider* slider = new QSlider(Qt::Horizontal);

    // Выполняем компоновку
    mainLayout->addWidget(label);
    mainLayout->addWidget(spinBox);
    mainLayout->addWidget(slider);

    // Выполняем связывание сигналов и слотов
```

```
    QObject::connect(spinBox, SIGNAL(valueChanged(int)),
                     label, SLOT(setNum(int)));
    QObject::connect(spinBox, SIGNAL(valueChanged(int)),
                     slider, SLOT(setValue(int)));
    QObject::connect(slider, SIGNAL(valueChanged(int)),
                     label, SLOT(setNum(int)));
    QObject::connect(slider, SIGNAL(valueChanged(int)),
                     spinBox, SLOT(setValue(int)));

    window.show();

    return a.exec();
}
```

Сигналы valueChanged() отправляются виджетами ввода целых чисел при изменении состояния (хранимого значения), слот setValue() позволяет установить значение виджета ввода целых чисел вручную, слот setNum() позволяет заменить текст надписи на заданное числовое значение.

2.2.3 Прочие возможности

Контейнеры данных — классы, обеспечивающие возможности контейнеров C++ STL (абстракции структур данных), но обладающих большей функциональностью.

В качестве примера можно привести QString — контейнер для хранения строк и манипуляций над ними. В отличие от своего STL-аналога string, QString хранит и обрабатывает строки в Unicode, применяет механизм неявного совместного использования и т.д.

Другие примеры контейнеров Qt — QVector, QList, QLinkedList, QStringList, QMap, QHash, QStack, QQueue и другие.

Различные операционные системы могут иметь различные интерфейсы доступа к носителям данных. Например, в ОС Windows компоненты пути к файлу разделяются символом обратной косой черты, а в ОС Linux — обычной косой черты; в ОС Windows приложения обычно размещаются в каталоге Program Files, а в ОС Linux могут храниться, например, в /usr/bin.

Кроссплатформенная система программирования, такая как Qt, должна предоставлять программисту некий унифицированный интерфейс доступа к файлам и каталогам, не зависящий от деталей конкретной целевой системы.

Qt имеет набор классов для представления объектов на накопителях (QDir, QFile), потокового ввода-вывода (QTextStream, QDataStream), а также богатый набор функций, значительно упрощающих задачу программиста.

Следует также отметить, что возможности Qt гораздо шире библиотеки функций для создания графического интерфейса пользователя. Qt включает несколько модулей, реализующих классы для организации обмена данными по сети, обработки структурированных XML-данных, доступа к базам данных и даже работы с подсистемами мобильных устройств, такими как GPS-приемник или акселерометр.

§3. Средства визуального проектирования

Qt предоставляет разработчику не только набор библиотек, но и комплекс утилит, позволяющих, в частности, значительно сократить объем написанного вручную кода за счет использования визуальных средств.

В настоящее время программисту доступны для загрузки как библиотеки и утилиты в отдельности, так и комплект разработчика — Qt SDK, включающий и то, и другое. Комплект разработчика поставляется с интегрированной средой Qt Creator, объединяющей возможности практически всех утилит, входящих в комплект, а также предоставляющий редактор кода, средства интеграции с отладчиком и системами управления версиями и т.п. К числу основных утилит Qt следует отнести Qt Designer — визуальный редактор форм, Qt Assistant — средство организации справочной информации, Qt Linguist — инструментарий создания многоязычного интерфейса приложения.

Одним из основных принципов Qt выражен слоганом «пиши меньше — создавай больше» (code less — create more). Для наглядности поясним этот принцип на очень показательном примере создания простого приложения, заимствованном из [5], попутно рассмотрев основные возможности дизайнера форм.

Разрабатываемое приложение представляет собой простейшую телефонную книгу.

Главное окно приложения отображает перечень записей (пар имя–номер), а также предоставляет интерфейс для управления записями: кнопку «Add» для добавления новой записи, кнопку «Edit» для редактирования записи, кнопку «Del» для удаления выделенной записи и кнопку «Clear» для удаления всех записей.

По нажатию на кнопку «Add» или «Edit» открывается вспомогательное диалоговое окно, позволяющее ввести или отредактировать имя и номер. Диалоговое окно имеет две кнопки — «Ok» для принятия изменений и «Cancel» для отмены.

Схематично интерфейс приложения изображен на рис. 2.

Откроем Qt Creator. Внешний вид стартовой страницы показан на рис 3. Помимо доступа к среде разработки, начальная страница открывает программисту богатую коллекцию примеров и документацию. Для начала работы выберем вариант «Создать проект...». С помощью открывшегося мастера проекта последовательно устанавли-

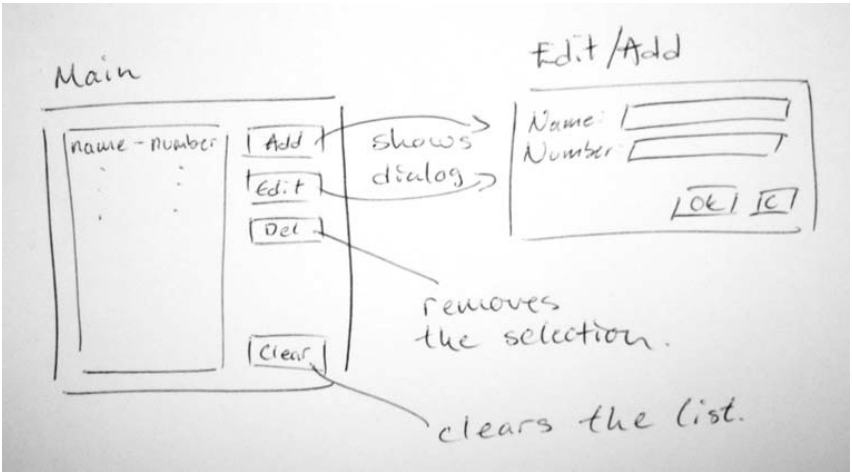

Рис. 2: Набросок интерфейса пользователя

ваем следующем параметры:

1. в окне «Новый проект» выбираем вариант «Проект Qt Widget / GUI приложение Qt»;
2. в окне «Введение и размещение проекта» задаем имя («PhoneBook») и каталог проекта;
3. в окне «Настройка цели» выбираем вариант «Desktop» (приложение для настолького компьютера);
4. в окне «Информация о классе» установим QWidget в качестве базового класса, убедимся, что отмечен флажок «Создать форму»;
5. в окне «Управление проектом» откажемся от использования системы контроля версий (вариант по умолчанию).

В результате работы мастера будет сгенерирован шаблон приложения, содержащий файлы исходного кода main.cpp (главная программа), widget.h (интерфейс класса Widget, представляющего форму

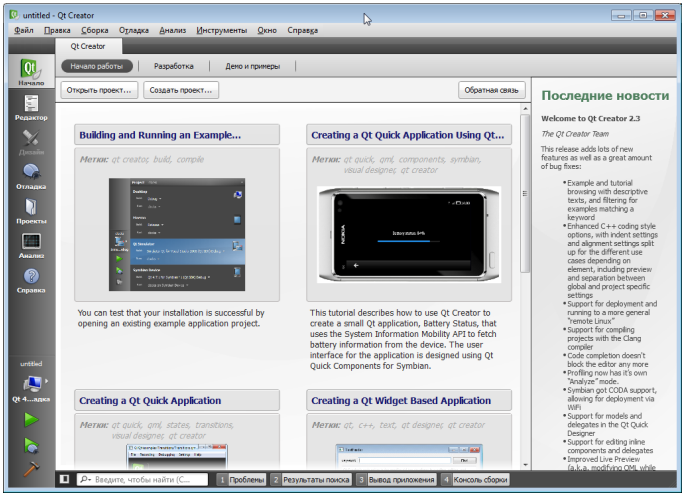

Рис. 3: Стартовое окно Qt Creator

приложения) и widget.cpp (реализация класса Widget), а также файл проекта PhoneBook.pro и форму widget.ui. При этом Qt Creator переключится в режим редактора. Дважды щелкнув по файлу формы, перейдем в режим дизайнера. Вид окна Qt Creator в режиме дизайнера показан на рис. 4.

Рассмотрим инструменты, доступные разработчику при работе с дизайнером форм.

Собственно графическая форма, представляющая проектируемый интерфейс находится в верхней части центральной области окна. На рис. 4 форма пуста.

Слева размещается палитра компонентов, представляющая библиотеку виджетов Qt. Для размещения виджета на форме достаточно перетащить его мышью.

Каждый виджет имеет набор свойств, как собственных, так и унаследованных от базовых классов (вплоть до QWidget и QObject). Программист имеет возможность модифицировать значения доступных для редактирования свойств с помощью редактора свойств. На рис. 4

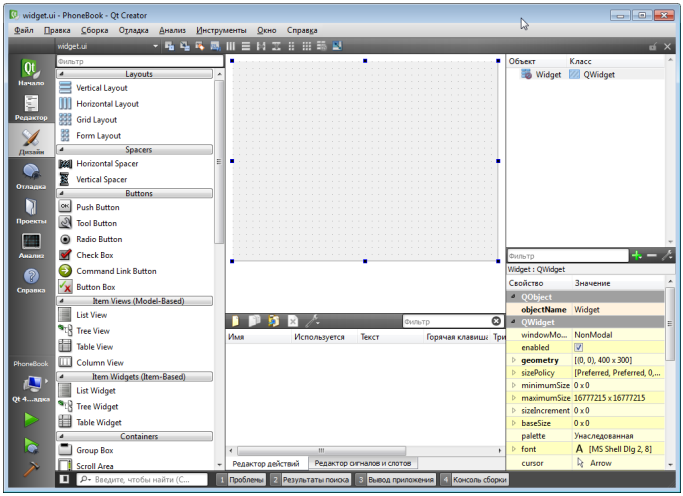

Рис. 4: Вид окна Qt Creator в режиме дизайнера

редактор свойств находится в правом нижнем углу.

Над редактором свойств на рис. 4 размещается инспектор объектов, изображающий спроектированный интерфейс в виде иерархической структуры (в соответствии с вложенностью виджетов в контейнеры и компоновщики).

В нижней части центральной области на рис. 4 находятся сгруппированные вместе редактор действий (привязка горячих клавиш, подсказок и т. п.) и редактор сигналов и слотов.

Приступим к созданию интерфейса основного окна нашего приложения. Для этого перетащим на форму четыре кнопки (Push Button) и вертикальный разделитель (Vertical Spacer). Выравнивание элементов не имеет значения (см. рис. 5 слева). Выделим все элементы и применим вертикальный компоновщик (рис. 5 справа).

Для управления компоновкой можно воспользоваться панелью компоновки в верхней части окна дизайнера (рис. 6) или выбрать соответствующий пункт контекстного меню, вызванного на выделенных объектах.

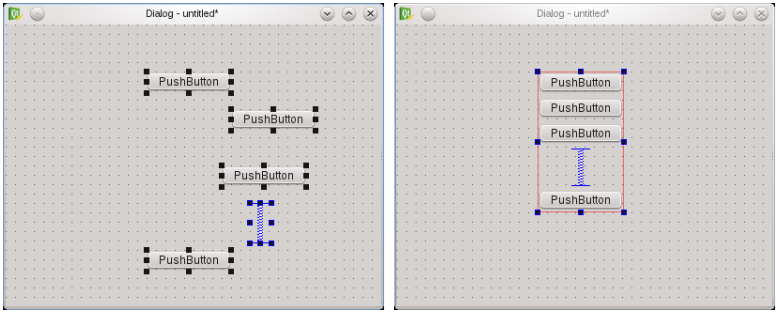

Рис. 5: Добавляем и компонуем набор кнопок

Рис. 6: Панель компоновки

Добавим виджет ListWidget и применим к форме табличный компоновщик (рис. 7).

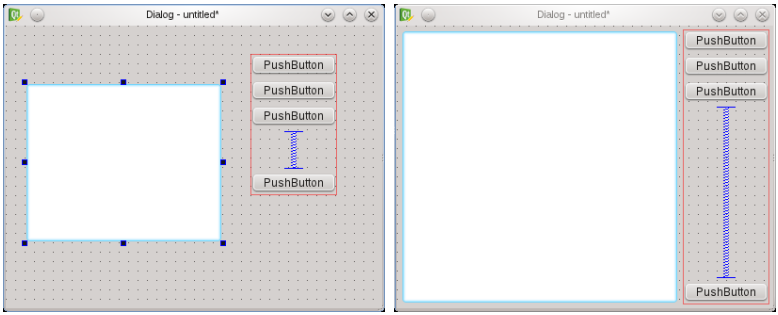

Рис. 7: Завершаем компоновку интерфейса

Используя редактор свойств, установим значения свойств objectName

и text кнопок, задав соответственно «addButton» / «Add», «editButton» / «Edit», «deleteButton» / «Delete» и «clearButton» / «Clear All». Аналогично установим значение свойства objectName для виджета списка в «list».

Все предыдущие операции выполнялись в режиме изменения виджетов. Дизайнер форм поддерживает четыре режима:

- режим изменения виджетов, предназначенный для визуального проектирования интерфейса;
- режим редактирования сигналов и слотов;
- режим изменения партнёров, предназначенный для привязки надписей к полям ввода;
- режим изменения порядка обхода, предназначенный для определения порядка передачи фокуса клавиатуры по нажатию клавиши табуляции.

Для переключения режимов в верхней части дизайнера имеется специальная панель (см. рис. 8).

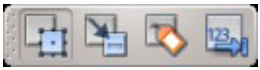

Рис. 8: Панель режимов

Переключимся в режим редактирования сигналов и слотов. Захватив мышью кнопку «Clear All», переведем курсор на виджет списка. В открывшемся диалоговом окне, для отправителя (кнопки) выберем сигнал clicked(), а для виджета списка слот clear(). Теперь при нажатии на кнопку «Clear All» все данные виджета списка будут удаляться.

Переключимся в режим изменения порядком обхода и установим нужный (см. рис. 12).

Добавим в проект еще один файл формы (editdialog.ui) и аналогично описанному выше спроектируем второе окно. Полям ввода присвоим имена nameEdit и numberEdit.

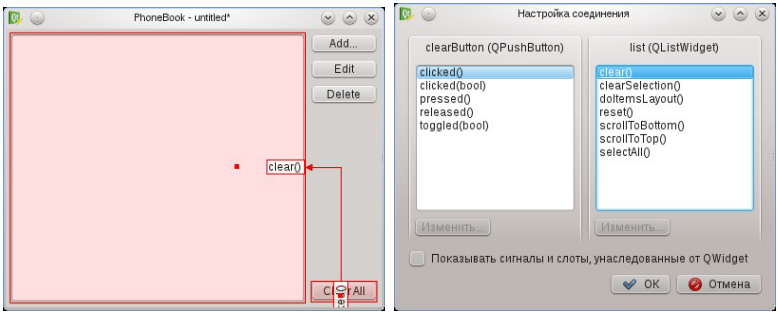

Рис. 9: Выполняем связывание сигналов и слотов

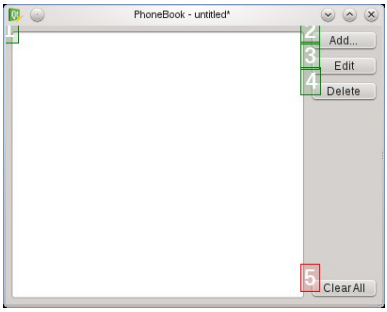

Рис. 10: Устанавливаем порядок обхода виджетов

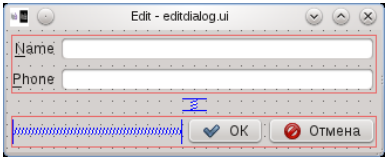

Рис. 11: Диалог редактирования

Для обеспечения возможности быстрого доступа к полям ввода с клавиатуры, настроим горячие клавиши. Для этого в тексты надписей зададим как «&Name» и «&Phone» и, переключившись в режим редактирования партнеров, свяжем надписи с полями ввода. Теперь полям ввода можно быстро передавать фокус клавиатуры, используя сочетания Alt+N и Alt+P соответственно, а в тексте надписей эти символы будут изображаться с подчеркиванием, подсказывая пользователю комбинации горячих клавиш.

На этом этапе подготовка форм завершена. Заметим, что в проекте пока еще нет ни одной написанной вручную строки кода!

Сохраненная форма представляет собой XML-файл (*.ui) с описанием интерфейса. Файлы описаний интерфейса обрабатываются специальной программой uic (user interface compiler). На выходе uic генерирует файлы исходного кода, которые включаются в проект.

Создадим обработчик нажатия кнопки «Add...». Для этого, находясь в режиме изменения виджетов, в контекстном меню на кнопке «Add...» выберем пункт «Перейти к слоту...» и в открывшамся окне выберем сигнал clicked(). Qt Creator переключится в режим редактирования файла исходного кода, соответствующего форме (см. пример **??**).

Пример 3.1

```
#include "widget.h"
#include "ui_widget.h"

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);
}

Widget::~Widget()
{
    delete ui;
}
```

```
void Widget::on_addButton_clicked()
{

}
```

Первые два метода построены из шаблона проекта, использующего форму. Все элементы спроектированной визуально формы доступны через объект ui, например ui->editButton. Третий метод — сгенерированный слот для обработки нажатия кнопки. Автоматически создаваемые слоты получают имена по схеме on_objectName_signal.

Добавим в тело обработчика код, создающий диалоговое окно для ввода имени и номера.

Пример 3.2

```
EditDialog dlg( this);
if( dlg.exec() == Qt::Accepted )
    ui->list->addItem( dlg.name() + " -- " + dlg.number());
```

Аналогично построим обработчик кнопки удаления.

Пример 3.3

```
void Widget::on_deleteButton_clicked()
{
    foreach (QListWidgetItem *item, ui->list->selectedItems())
        delete item;
}
```

Заметим, что кнопка «Delete» всегда активна, что не корректно — кнопка должна быть доступна только, если есть выделенные позиции в списке. Добавим слот updateDeleteEnabled():

Пример 3.4

```
void Widget::updateDeleteEnabled()
{
```

```
    ui->deleteButton->setEnabled(
        ui->list->selectedItems().count() != 0);
}
```

Дополнительно необходимо переключиться к заголовочному файлу и объявить этот слот в секции private slots.

Соединим созданный слот с сигналом об изменении выделения:

Пример 3.5

```
connect(ui->list->selectionModel(),
        SIGNAL(selectionChanged(QItemSelection,
                                QItemSelection)),
        this,
        SLOT(updateDeleteEnabled()));
```

Аналогично обработчику кнопки «Add» реализуется обработчик кнопки «Edit». Также дополнительно потребуется реализовать методы класса EditDialog, представляющего диалоговое окно редактирования записи телефонной книги. Предоставим читателю завершить работу над этим приложением.

§4. Компоновка и размещение виджетов

Типичное окно современного приложения с графическим интерфейсом может содержать десятки виджетов, некоторым образом размещенных на холсте окна.

Размещение виджетов может быть выполнено вручную. QWidget и производные классы включают метод setGeometry(), обеспечивающий возможность задать фиксированные размеры и позицию виджета (см. пример 4.1).

Пример 4.1

```
Window::Window(QWidget *parent) : QWidget(parent)
{
    setFixedSize(640, 480);
    QTextEdit *txt = new QTextEdit(this);
    txt->setGeometry(20, 20, 600, 400);

    QPushButton *btn = new QPushButton(tr("&Close"), this);
    btn->setGeometry(520, 440, 100, 20);
}
```

Однако расстановка виджетов вручную приводит к ряду сложностей. Например, при необходимости добавить новый виджет, требуется модифицировать местоположение остальных. Также возникают проблемы с изменением размеров окна пользователем — при фиксированном размещении часть элементов управления может оказаться недоступной, вне видимой области окна.

4.1 Использование компоновщиков

Qt предоставляет простой механизм автоматического размещения виджетов внутри других виджетов посредством компоновщиков. В частности, копмоновка позволяет разумно использовать пространство окна и реализовать так называемый «эластичный интерфейс», когда виджеты получают возможность адаптировать свои размеры и размещение в зависимости от содержимого и пользовательских настроек, например, автоматически растягиваться, чтобы вместить текст с увеличенным шрифтом.

Qt предоставляет три вида компоновщиков: вертикальный (класс QVBoxLayout), горизонтальный (класс QHBoxLayout), табличный (класс QGridLayout) и компоновщик форм (QFormLayout), предназначенный для формирования форм ввода, состоящих из колонки пар надпись — поле ввода. Непосредственные размеры вложенных виджетов определяются в ходе выполнения специального процесса, называемого «переговорами».

Рассмотрим пример типичного диалогового окна:

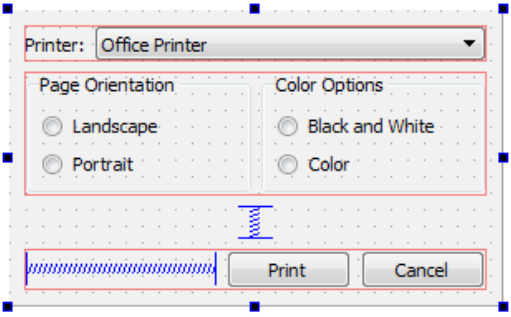

Рис. 12: Пример использования компоновщиков

Надпись «Printer» и выпадающий список в верхней части диалогового окна объединены с помощью горизонтального компоновщика. Второй горизонтальный компоновщик объединяет две группы кнопок. Третий формирует строку кнопок в нижней части окна. Для выравнивания элементов диалогового окна применены разделители (QSpacer) — невидимые виджеты-распорки. Все три горизонтальных компоновщика и один из разделителей выровнены с помощью вертикального компоновщика, примененного ко всему окну.

Следующий пример демонстрирует общую схему построения интерфейса, представленного на рис. 12 с использованием вертикальных и горизонтальных компоновщиков.

Пример 4.2

```
// Создаем внешний вертикальный компоновщик
QVBoxLayout *outerLayout = new QVBoxLayout(this);
```

```
// Создаем верхний горизонтальный компоновщик
QHBoxLayout *topLayout = new QHBoxLayout();
// Добавляем виджеты
topLayout->addWidget(new QLabel("Printer:"));
topLayout->addWidget(c=new QComboBox());
// Добавляем в внешний компоновщик
outerLayout->addLayout(topLayout);

// Создаем средний горизонтальный компоновщик
QHBoxLayout *groupLayout = new QHBoxLayout();
...
outerLayout->addLayout(groupLayout);

// Добавляем разделитель
outerLayout->addSpacerItem(new QSpacerItem(...));

// Создаем нижний горизонтальный компоновщик
QHBoxLayout *buttonLayout = new QHBoxLayout();
buttonLayout->addSpacerItem(new QSpacerItem(...));
buttonLayout->addWidget(new QPushButton("Print"));
buttonLayout->addWidget(new QPushButton("Cancel"));
outerLayout->addLayout(buttonLayout);

// Применяем внешний компоновщик к окну
setLayout(outerLayout);
```

Каждый виджет наследует от класса QWidget свойства sizeHint, minimumSizeHint и sizePolicy. Свойство sizeHint определяет, какое пространство виджет «хотел бы» занимать в нормальных условиях. Свойство minimumSizeHint задает минимальный размер, меньше которого виджет не может занимать ни при каких обстоятельствах. Значением свойства sizePolicy является политика, определяющая отношение виджета к изменениям его размера и принимающая одно из следующих значений:

- QSizePolicy::Fixed — виджет не может иметь размер, отличный от sizeHint;

- QSizePolicy::Minimum — sizeHint представляет минимально допустимый размер виджета, но он может быть неограниченно растянут;
- QSizePolicy::Maximum — sizeHint представляет максимально допустимый размер виджета, но он может быть неограниченно ужат;
- QSizePolicy::Preferred — sizeHint представляет оптимальный размер виджета, но изменения допустимы в обе стороны;
- QSizePolicy::Expanding — аналогично Preferred, но виджет требует предоставить ему любое доступное пространство компоновщика;
- QSizePolicy::MinimumExpanding — аналогично Minimum, но виджет требует предоставить ему любое доступное пространство компоновщика;
- QSizePolicy::Ignored — sizeHint игнорируется, виджету выделяется столько пространства, сколько возможно в пределах компоновщика.

Помимо варьирования свойств sizeHint, minimumSizeHint и sizePolicy вложенных в компоновщик виджетов, управление размещением может быть выполнено и средствами самого компоновщика. Метод addWidget() содержит необязательный второй аргумент (stretch factor), равный по умолчанию нулю. Ненулевые значения коэффициента растяжения задают относительный размер пространства, выделенного компоновщиком под размещение виджета.

При использовании табличного компоновщика программист задает, по каким границам сетки выравнивается каждый вложенный виджет. Такой подход позволяет занимать под виджет любую прямоугольную подобласть таблицы, а также создавать перекрытия. Если добавляемый виджет занимает ровно одну ячейку таблицы, достаточно определить левую и верхнюю границы, в противном случае — все четыре (см. пример 4.3).

Пример 4.3

```
QPushButton *button1 = new QPushButton("One");
QPushButton *button2 = new QPushButton("Two");
QPushButton *button3 = new QPushButton("Three");
QPushButton *button4 = new QPushButton("Four");
QPushButton *button5 = new QPushButton("Five");

QGridLayout *layout = new QGridLayout;
layout->addWidget(button1, 0, 0);
layout->addWidget(button2, 0, 1);
layout->addWidget(button3, 1, 0, 1, 2);
layout->addWidget(button4, 2, 0);
layout->addWidget(button5, 2, 1);
```

Предложенный в примере код создает интерфейс, представленный на рис. 13.

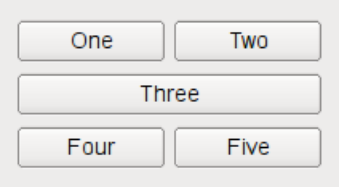

Рис. 13: Пример использования табличного компоновщика

4.2 Контейнерные виджеты

Помимо компоновщиков иерархическая структура графического интерфейса может быть построена с помощью виджетов-контейнеров — виджетов, которые могут содержать другие виджеты.

Контейнерные виджеты могут применяться как для организации высокоуровневых элементов, например, окон — QWidget, QDialog, а также QMainWindow, так и группировки виджетов внутри окон — QGroupBox, QTabWidget и т.п.

Пример 4.4 иллюстрирует применение QMainWindow для организации окна приложения и QTabWidget для создания набора вкладок. Следует отметить, что в отличие от «пустых» контейнеров QWidget и QDialog, QMainWindow включает основные элементы типового окна приложения — меню, строку состояния и контейнер для наполнения центральной области (centralWidget).

Пример 4.4

```
#include <QApplication>
#include <QMainWindow>
#include <QTabWidget>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QMainWindow *window = new QMainWindow();

    window->setWindowTitle("MainWindow");
    window->resize(250, 250);

    QWidget *centralWidget = new QWidget(window);
    QTabWidget *tabs = new QTabWidget(centralWidget);

    tabs->setFixedSize(245, 245);
    tabs->addTab(new QWidget(),"Tab 1");
    tabs->addTab(new QWidget(),"Tab 2");

    window->setCentralWidget(centralWidget);
    window->show();

    return app.exec();
}
```

§5. Обзор библиотеки виджетов Qt

В предыдущих разделах уже были приведены примеры виджетов — основным строительных элементов графического интерфейса пользователя.

Далее рассмотрим несколько основных виджетов Qt. В целях экономии все примеры построены по общей схеме: имеется приложение, состоящее из трех следующих файлов исходного кода. В главном файле main.cpp создается экземпляр приложения и собственного виджета Widget.

Пример 5.1

```
#include <QtGui/QApplication>
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

Файл widget.h содержит описание класса Widget, при этом в примерах будут варьироваться файлы заголовков и приватные компоненты.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QtGui/QWidget>

// Необходимые файлы заголовков

class Widget : public QWidget
{
```

```
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

private:
    // Необходимые приватные компоненты
};

#endif // WIDGET_H
```

Файл widget.cpp содержит реализацию методов класса Widget, при этом пользовательский интерфейс формируется в конструкторе класса. Именно код конструктора будет приведен во всех последующих примерах этого раздела.

```
#include "widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    // Конструируем интерфейс пользователя
}

Widget::~Widget()
{

}
```

5.1 Кнопки

Три класса QPushButton, QCheckBox, QRadioButton реализуют соответственно простую кнопку, флажок и радиокнопку. Базовым классом всех кнопок является QAbstractButton.

Основные сигналы для кнопок:

- clicked() отправляется в момент, когда пользователь отпускает кнопку мыши;
- toggled(bool) отправляется в момент изменения состояния кнопки.

Отметим также наиболее полезные свойства:

- checkable — истина, если кнопка может работать как флажок;
- checked — истина, если флажок имеет отметку;
- text — текст кнопки;
- icon — иконка кнопки, которая может отображаться вместе с текстом.

Следующий пример иллюстрирует использование QCheckBox. В конструкторе создадим флажок, установив отметку как начальное состояние.

Пример 5.2

```
    // Создаем флажок
    checkBox = new QCheckBox("Show Title", this);
    checkBox->setCheckState(Qt::Checked);

    // Связываем изменение состояния флажка
    // с обработчиком
    connect(checkBox, SIGNAL(stateChanged(int)),
            this, SLOT(showTitle(int)));
```

Код обработчика для изменения состояния флажка приведен ниже. В данном примере меняем заголовок окна.

```
void Widget::showTitle(int state)
{
    if (state == Qt::Checked) {
        setWindowTitle("QCheckBox");
    } else {
        setWindowTitle("");
    }
}
```

5.2 Надписи

QLabel отображает текстовый блок или картинку, которые задаются свойствами text и pixmap соответственно. Следующий пример демонстрирует использование QLabel для изображения простой текстовой надписи.

Пример 5.3

```
// Создаем компоновщик
layout = new QHBoxLayout();

// Создаем надпись
label = new QLabel("Hello");

// Добавляем надпись в компоновщик,
   центрируя разделителями
layout->addWidget(new QSplitter());
layout->addWidget(label);
layout->addWidget(new QSplitter());

// Применяем компоновщик к виджету
setLayout(layout);
```

QLabel распознает некоторое подмножество тэгов HTML-разметки.

Пример 5.4

```
// Создаем полужирную надпись красного цвета
label = new QLabel(
    "<b><font color=\"red\">Hello</font></b>");
```

Для класса QLabel предусмотрен так называемый механизм «приятелей» (buddy). На практике надписи часто появляются в связке с виджетами, обеспечивающими ввод данных, например, QLineEdit. При этом для комфортной работы пользователя разумно предоставлять возможность передачи фокуса клавиатуры в поле ввода по горячей клавише. Однако подсказку о том, какая конкретно комбинация

клавиш переводит фокус в поле ввода разместить можно только в надписи, обычно, подчеркиванием буквы, клавишу которой необходимо нажать, удерживая Control. Таким образом надпись и поле ввода работают в паре. Механизм работает, только в случае, когда в тексте надписи одной из букв (клавиша которой задействована в комбинации) предшествует амперсанд.

Пример 5.5

```
// Создаем надпись и поле ввода для ввода имени
QLineEdit *nameEdit  = new QLineEdit(this);
QLabel *nameLabel  = new QLabel("&Name:", this);
nameLabel->setBuddy(nameEdit);

// Создаем надпись и поле ввода для ввода номера
QLineEdit *phoneEdit = new QLineEdit(this);
QLabel *phoneLabel = new QLabel("&Phone:", this);
phoneLabel->setBuddy(phoneEdit);
```

5.3 Инструменты ввода текста

Виджет QLineEdit обеспечивает ввод одной строки текста.
Основные сигналы:

- textChanged() отправляется при изменении текста;
- editingFinished() отправляется при потере фокуса ввода;
- returnPressed() отправляется при нажатии клавиши «Enter».

Полезные свойства:

- text — текст в поле ввода;
- maxLength — предельная длина текста в поле ввода;
- readOnly — если истина, редактирование запрещено.

Следующий пример создает простой интерфейс для редактирования визитной карточки.

Пример 5.6

```
// Создает набор надписей
QLabel *name = new QLabel("Name:", this);
QLabel *age = new QLabel("Age:", this);
QLabel *occupation = new QLabel("Occupation:", this);

// Создаем набор полей ввода
QLineEdit *le1 = new QLineEdit(this);
QLineEdit *le2 = new QLineEdit(this);
QLineEdit *le3 = new QLineEdit(this);

// Создаем табличный компоновщик
QGridLayout *grid = new QGridLayout();

// Размещаем виджеты в ячейках таблицы
grid->addWidget(name, 0, 0);
grid->addWidget(le1, 0, 1);
grid->addWidget(age, 1, 0);
grid->addWidget(le2, 1, 1);
grid->addWidget(occupation, 2, 0);
grid->addWidget(le3, 2, 1);

setLayout(grid);
```

Классы QTextEdit, QPlainTextEdit обеспечивают возможность реализации инструментов ввода многострочных текстов.

Основные сигналы:

- textChanged() отправляется при изменении текста.

Полезные свойства:

- plainText — неформатированный текст;
- html — текст использует HTML-разметку;
- readOnly — если истина, редактирование запрещено.

5.4 Инструменты ввода чисел

Все четыре перечисленных в заголовке подраздела класса являются инструментами ввода-вывода целых чисел, но отличаются визуальным представлением. В то время как QSlider реализует бегунок для ввода числового значения, QScrollBar применяется для позиционирования виджета большого размера относительно маленькой видимой области. QDial представляется круговой шкалой, а QSpinBox позволяет вводить числа вручную или листать с помощью стрелок.

Основные сигналы:

- valueChanged() отправляется при изменении значения.

Полезные свойства:

- value — текущее значение;
- maximum — максимальное значение;
- minimum — минимальное значение.

Пример 5.7

```
    // Создаем экземпляр поля ввода чисел
    spinBox = new QSpinBox();

    // Обрабатываем сигнал изменения значения
    connect(spinBox, SIGNAL(valueChanged(int)),
            this, SLOT(setTitle(int)));
}
```

В обработчике будем печатать в отладочную консоль новое значение, установленное в виджете:

```
void Widget::setTitle(int val)
{
  qDebug() << QString::number(val);
}
```

§6. Создание собственных виджетов

Несмотря на обилие стандартных виджетов в Qt, программисту может потребоваться особенный элемент управления, не представленный в библиотеке.

6.1 Создание виджета композицией имеющихся

В качестве примера рассмотрим виджет, состоящий из элементов QDial, QLabel и QSlider, организованных в один столбец вертикальным компоновщиком.

Создадим заголовочный файл ComposedGauge.h, опишем интерфейс класса (см. пример 6.1). Так как наш класс является виджетом, наследуем его от QWidget. QWidget является потомков QObject, следовательно мы получим и механизм сигналов и слотов. Для корректной обработки класса метаобъектным компилятором добавляем макрос Q_OBJECT.

Объявляем свойство value, доступ к которому на чтение и запись обеспечивается соответственно функциями value() и setValue(). Функция setValue() объявлена слотом и будет связываться с сигналами. Мы также определяем собственный сигнал valueChanged() для информирования о том, что данные виджета изменились.

Для хранения фактического значения виджета требуется объект данных, импользуем в этом качестве приватную переменную m_slider.

Пример 6.1

```
class ComposedGauge : public QWidget
{
  Q_OBJECT
  Q_PROPERTY(int value READ value WRITE setValue)
  public:
    explicit ComposedGauge(QWidget *parent = 0);
    int value() const;
  public slots:
    void setValue(int);
  signals:
    void valueChanged(int);
```

```
  private:
    QSlider *m_slider;
};
```

В конструкторе класса построим интерфейс и свяжем сигналы и слоты.

```
ComposedGauge::ComposedGauge(QWidget *parent) :
QWidget(parent)
{
    // Создаем виджеты, упаковываем в компоновщик
    QVBoxLayout *layout = new QVBoxLayout(this);
    QDial *dial = new QDial();
    QLabel *label = new QLabel();
    m_slider = new QSlider();
    layout->addWidget(dial);
    layout->addWidget(label);
    layout->addWidget(m_slider);

    // Устанавливаем дополнительные параметры виджетов
    m_slider->setOrientation(Qt::Horizontal);
    label->setAlignment(Qt::AlignCenter);
    dial->setFocusPolicy(Qt::NoFocus);
    dial->setValue(m_slider->value());
    label->setNum(m_slider->value());

    // Связываем сигналы и слоты
    connect(dial, SIGNAL(valueChanged(int)),
            m_slider, SLOT(setValue(int)));
    connect(m_slider, SIGNAL(valueChanged(int)),
            dial, SLOT(setValue(int)));
    connect(m_slider, SIGNAL(valueChanged(int)),
            label, SLOT(setNum(int)));
    connect(m_slider, SIGNAL(valueChanged(int)),
            this, SIGNAL(valueChanged(int)));
```

Реализуем методы установки и чтения значения виджета (так называемые «геттер» и «сеттер»).

```
// Возвращает константную ссылку
// на хранимое значение
int ComposedGauge::value() const
{
    return m_slider->value();
}

// Модифицирует хранимое значение
// в соответствии с параметром
void ComposedGauge::setValue(int v)
{
    m_slider->setValue(v);
}
```

Сконструированный таким образом виджет можно легко встроить в проектируемый интерфейс.

```
ComposedGauge *gauge = new ComposedGauge();
layout->addWidget(gauge);
connect(gauge, SIGNAL(valueChanged(int)), ... );
```

Создание истинно собственного виджета требует от программиста самостоятельно:

- управлять отрисовкой виджета;
- обрабатывать события (клавиатура, мышь, оконный менеджер и другие);
- управлять подсказками и политиками размера виджета (для компоновки).

6.2 Отрисовка на поверхности виджета

Отрисовка реализуется через метод paintEvent(), который должен быть переопределен в классе. В методе paintEvent() необходимо получить экземпляр класса QPainter, соответствующий виджету и выполняющий отрисовку.

Пример 6.2

```
class MyWidget : public QWidget
{
  ...
  protected:
    void paintEvent(QPaintEvent*);
  ...
};

...

void MyWidget::paintEvent(QPaintEvent *ev)
{
    QPainter p(this);
    ...
}
```

Отрисовка выполняется в координатной плоскости относительно левого верхнего угла холста. Для работы с координатами применяются следующие классы Qt:

- QPoint представляет точку (пару координат);
- QSize представляет размеры прямоугольника (ширина, высота);
- QRect представляет прямоугольник (координаты левого верхнего угла, ширина и высота).

Аналогично QPointF, QSizeF, QRectF представляют те же величины, но в формате с плавающей точкой.

Тривиальная отрисовка выполняется с применением стилей контура (QPen), заливки (QBrush), для хранения цвета определен класс QColor, присутствует набор функций, реализующих простые геометрические примитивы. Следующий пример иллюстрирует использование заливки для изображение прямоугольника и эллипса.

Пример 6.3

```
void RectWithCircle::paintEvent(QPaintEvent *ev)
{
    QPainter p(this);
    p.setBrush(Qt::green);
    p.drawRect(10, 10, width()-20, height()-20);
    p.setBrush(Qt::yellow);
    p.drawEllipse(20, 20, width()-40, height()-40);
}
```

Класс QPen обеспечивает широкий набор стилей линий:

- Qt::SolidLine — сплошная линия;
- Qt::DashLine — штриховая линия;
- Qt::DotLine — пунктирная линия;
- и т. п.

Также определены стили соединения и концов линий.

Инструмент заливки также может использовать не только сплошной цвет, но и шаблоны, текстуры и градиенты.

Большинство функций рисования перегружены, чтобы обеспечить программисту возможность выбора наиболее удобного программного интерфейса:

Пример 6.4

```
    // Рисуем прямоугольник (три способа)
    drawRect(QRectF r);
    drawRect(QRect r);
    drawRect(int x, int y, int w, int h);

    //Рисуем точку (три способа)
    drawPoint(QPointF p);
    drawPoint(QPoint p);
    drawPoint(int x, int y);
```

Перечислим функции отрисовки некоторых простых примитивов:

- QPainter::drawPoint() — точка;
- QPainter::drawLine() — отрезок прямой;
- QPainter::drawRect() — прямоугольник;
- QPainter::drawRoundedRect() — скругленный прямоугольник;
- QPainter::drawEllipse() — эллипс;
- QPainter::drawArc() — участок дуги эллипса;
- QPainter::drawText() — произвольный текст.

При выводе текста на экран можно использовать класс QFont для установки параметры вывода текста (гарнитура шрифта, размер, и т. п.).

Пример 6.5

```
QPainter p(this);

QFont font("Helvetica");
p.setFont(font);
p.drawText(20, 20, 120, 20, 0, "Hello World!");

font.setPixelSize(10);
p.setFont(font);
p.drawText(20, 40, 120, 20, 0, "Hello World!");

font.setPixelSize(20);
p.setFont(font);
p.drawText(20, 60, 120, 20, 0, "Hello World!");

QRect r;
p.setPen(Qt::red);
p.drawText(20, 80, 120, 20, 0, "Hello World!", &r);
```

Отрисовывая примитивы, можно использовать базовые трансформации QPainter:

- QPainter::scale — масштабирование;

- QPainter::translate — перенос;
- QPainter::rotate — поворот;
- QPainter::shear — параллельные искажения.

Трансформации применяются к координатной сетке. Используя save() и restore() трансформации можно сохранять на стеке QPainter и позже восстановить.

Пример 6.6

```
QPoint rotCenter(50, 50);
qreal angle = 42;

// Сохраняем состояние трансформации
p.save();

// Выполняет поворот
p.translate(rotCenter);
p.rotate(angle);
p.translate(-rotCenter);

// Отрисовываем первый (повернутый) квадрат
p.setBrush(Qt::red);
p.setPen(Qt::black);
p.drawRect(25,25, 50, 50);

// Восстанавливаем состояние трансформации
p.restore();

// Отрисовываем второй квадрат
p.setPen(Qt::NoPen);
p.setBrush(QColor(80, 80, 80, 150));
p.drawRect(25,25, 50, 50);
```

6.3 Пример создания виджета с нуля

Создадим виджет, позволяющий визуализировать спидометр. Помимо графического изображения определим реакцию на события кла-

виатуры и мыши.

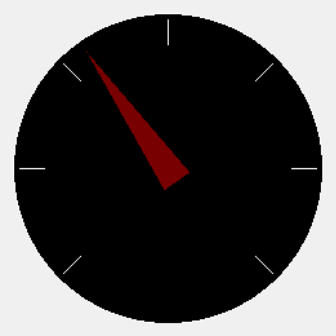

Рис. 14: Внешний вид виджета

Следующий код выполняет отрисовку спидометра.

Пример 6.7

```
void CircularGauge::paintEvent(QPaintEvent *ev)
{
        QPainter p(this);

        // Отрисовка фона
        int extent;
        if (width()>height())
        extent = height()-20;
        else
        extent = width()-20;
        p.translate((width()-extent)/2, (height()-extent)/2);
        p.setPen(Qt::white);
        p.setBrush(Qt::black);
        p.drawEllipse(0, 0, extent, extent);

        // Отрисовка отсечек
        p.translate(extent/2, extent/2);
```

```
        for(int angle=0; angle<=270; angle+=45)
        {
        p.save();
        p.rotate(angle+135);
        p.drawLine(extent*0.4, 0, extent*0.48, 0);
        p.restore();
        }

        // Отрисовка стрелки
        p.rotate(m_value+135);
        QPolygon polygon;
        polygon << QPoint(-extent*0.05, extent*0.05)
        << QPoint(-extent*0.05, -extent*0.05)
        << QPoint(extent*0.46, 0);
        p.setPen(Qt::NoPen);
        p.setBrush(QColor(255,0,0,120));
        p.drawPolygon(polygon);
}
```

Для определения реакции на нажатие клавиши на клавиатуре переопределим keyPressEvent():

Пример 6.8

```
void CircularGauge::keyPressEvent(QKeyEvent *ev)
{
    switch(ev->key())
    {
        case Qt::Key_Up:
        case Qt::Key_Right:
            setValue(value() + 1);
            break;
        case Qt::Key_Down:
        case Qt::Key_Left:
            setValue(value() - 1);
            break;
        case Qt::Key_PageUp:
            setValue(value() + 10);
```

```
            break;
        case Qt::Key_PageDown:
            setValue(value() - 10);
            break;
        default:
           QWidget::keyPressEvent(ev);
    }
}
```

Аналогичным образом отслеживаются события мыши.

Пример 6.9

```
// Обрабатываем нажатие кнопки мыши
void CircularGauge::mousePressEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}
// Обрабатываем освобождение кнопки мыши
void CircularGauge::mouseReleaseEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}
// Обрабатываем перемещение указателя мыши
void CircularGauge::mouseMoveEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}
```

В некоторых случаях, необходимо предусмотреть реакцию на событие, не связанное с пользовательских вводом, а происходящее по истечении определенного интервала времени. Для этого воспользуемся классом QTimer.

Пример 6.10

```
MyClass(QObject *parent) : QObject(parent)
{
```

```
    // Создаем экземпляр таймера
    QTimer *timer = new QTimer(this);

    // Взводим таймер на 5 секунд
    timer->setInterval(5000);

    // Связываем сигнал таймера с обработчиком
    connect(timer, SIGNAL(timeout()),
            this, SLOT(doSomething());
    // Запускаем обратный отсчет
    timer->start();
}
```

В приведенном выше коде сигнал таймера будет приходить каждые пять секунд. В некоторых случаях необходимо запланировать однократное выполнение некоторого кода в определенный момент в будущем.

```
    // Функция doSomething() будет выполнена через 1.5 с
    QTimer::singleShot(1500, dest, SLOT(doSomething());
```

Следует также упомянуть два класса Qt для работы с изображениями: QImage, оптимизированный для выполнения манипуляций с картинкой, и QPixmap, оптимизированный для быстрой отрисовки на экране.

§7. Работа с 2D-графикой

7.1 Graphics View Framework

В §6 был предложен способ отрисовки двухмерных изображений на поверхности виджета переопределением метода paintEvent().

Пример 7.1

```
void Widget::paintEvent(QPaintEvent *)
{
    // Получаем доступ к холсту
    QPainter painter(this);

    // Выполняем отрисовку
    painter.setPen(QPen(Qt::black, 2));
    painter.setBrush(Qt::green);
    painter.drawRect(20, 20, 60, 60);
    ...
}
```

Этот подход обладает рядом существенных ограничений. В частности, поверхность виджета всегда имеет прямоугольную форму, спозиционированные с помощью компоновщиков виджеты не перекрываются, отрисовка ведется с применением целочисленных координат пикселей.

С другой стороны, программист может быть заинтересован в создании композиции из нескольких, возможно перекрывающих друг друга элементов произвольной формы, произвольно позиционируемых с использованием вещественнозначных (субпиксельных) координат. При этом может требоваться независимое перемещение элементов друг относительно друга, а также применение графических эффектов и анимаций.

Для реализации подобных сценариев (например, при разработке компьютерных игр) в Qt предусмотрен набор классов под общим названием Graphics Vew Framework, обеспечивающий работу с 2D-графикой на основе понятия сцены.

Graphics Vew Framework стоит на трех китах: экземпляры классов QGraphicsItem и QGraphicsObject выступают «актерами» — составляющими графической композиции, объект QGraphicsScene объединяет и структурирует их, а объект QGraphicsView обеспечивает отрисовку сцены.

Пример 7.2

```
void Widget::setupScene()
{
    // Создаем представление для отображения сцены
    QGraphicsView *view = new QGraphicsView();

    // Создаем сцену
    QGraphicsScene *scene = new QGraphicsScene(0, 0
        300, 200, this);

    // Выполняем отрисовку примитивов
    scene->addRect(20, 20, 60, 60, QPen(Qt::red, 3),
        QBrush(Qt::green));
    scene->addEllipse(120, 20, 60, 60, QPen(Qt::red, 3),
        QBrush(Qt::yellow));
    scene->addPolygon(QPolygonF() << QPointF(220, 80)
        << QPointF(280, 80) << QPointF(250, 20),
        QPen(Qt::blue, 3), QBrush(Qt::magenta));

    // Связываем сцену с представлением
    view->setScene(scene);
    view->show();
}
```

На рис. 15 приведен результат выполнения кода примера 7.2.

Класс QGraphicsScene представляет сцену: содержит все элементы и работает как интерфейс между ними и представлением, распределяет события, а также содержит методы для обхода и поиска элементов.

Каждая сцена имеет ограничивающий прямоугольник (sceneRect), определяющий ее границы. Если ограничивающий прямоугольник не указан, в качестве него выбирается самый большой прямоугольник,

Рис. 15: Результат отрисовки примера 7.2

содержащий все элементы. Ограничивающий прямоугольник не обязательно должен начинаться в точке с координатами (0, 0).

Пример 7.3

```
QGraphicsScene *scene = new QGraphicsScene(this);
// Явно определяем ограничивающий прямоугольник
scene->setSceneRect(-100, -100, 201, 201);
```

Класс QGraphicsScene имеет методы, позволяющие добавить простые примитивы. Поддерживаются отрезки прямых линий, эллипсы, прямоугольники, многоугольники, текст, изображения. Тип каждого элемента сцены представлен отдельным классом, унаследованным от QGraphicsItem.

Пример 7.4

```
QGraphicsItem *item = scene->addRect(20, 20, 60, 60,
    QPen(Qt::red, 3), QBrush(Qt::green));
```

Виджет QGraphicsView отображает прямоугольную часть сцены (не обязательно сцену целиком). Непосредственная привязка сцены обеспечивается методом setupScene().

Пример 7.5

```
QGraphicsView *view = new QGraphicsView();
QGraphicsScene *scene = setupScene();
view->setScene(scene);
```

По умолчанию сцена центрирована. Следует отметить, что класс QGraphicsView наследует от класса QAbstractScrollArea, в том числе свойства horizontalScrollBarPolicy и verticalScrollBarPolicy. Таким образом, можно организовать прокрутку сцены внутри QGraphicsView.

И представление, и сцена, и каждый отдельный элемент имеют локальную систему координат, при этом для представления и элементов определены методы для перехода от одной системы к другой (mapFromScene / mapToScene, mapFromParent / mapToParent, а также mapFromItem / mapToItem). Сцена всегда использует свою систему координат.

7.2 Управление элементами сцены

Элементы сцены могут подчиняться другим элементам, образуя иерархии.

Пример 7.6

```
QGraphicsRectItem *rootItem =
    new QGraphicsRectItem(-50, -50, 101, 101);
rootItem->setBrush(Qt::green);

QGraphicsEllipseItem *item;

// Следующие четыре эллипса подчинены прямоугольнику
item = new QGraphicsEllipseItem(-40, -40, 30, 30, rootItem);
item->setBrush(Qt::yellow);
item = new QGraphicsEllipseItem(10, -40, 30, 30, rootItem);
item->setBrush(Qt::blue);
item = new QGraphicsEllipseItem(-40, 10, 30, 30, rootItem);
item->setBrush(Qt::red);
item = new QGraphicsEllipseItem(10, 10, 30, 30, rootItem);
item->setBrush(Qt::magenta);

scene->addItem(rootItem);
```

Результат выполнения кода примера 7.6 приведен на рис. 16.

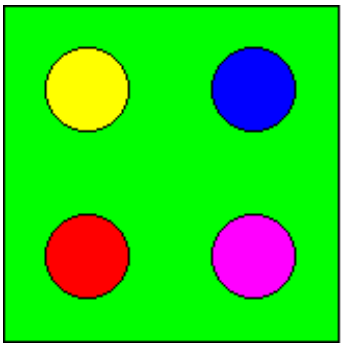

Рис. 16: Пример иерархии элементов сцены

Элементы сцены могут быть спозиционированы относительно друг друга с помощью компоновщиков (аналогично компоновке виджетов). Базовый класс для компоновщиков сцены — QGraphicsLayout. Программисту при этом доступны:

- QGraphicsLinearLayout организует элементы в строку или столбец;
- QGraphicsGridLayout организует элементы в виде таблицы;
- QGraphicsAnchorLayout привязывает элементы к соседям.

При создании линейного компоновщика необходимо выбрать ориентацию (в строку или столбец), элементы можно добавить в компоновщик с помощью методов addItem() и insertItem().

Пример 7.7

```
QGraphicsLinearLayout(Qt::Orientation orientation,
                      QGraphicsLayoutItem * parent = 0);
void addItem (QGraphicsLayoutItem *item);
void insertItem (int index, QGraphicsLayoutItem *item);
```

Пример 7.8 демонстрирует привязку левого края элемента b к правому краю элемента a.

Пример 7.8

```
QGraphicsAnchor *addAnchor(
    QGraphicsLayoutItem *firstItem,
    Qt::AnchorPoint firstEdge
    QGraphicsLayoutItem *secondItem,
    Qt::AnchorPoint secondEdge);

layout->addAnchor(b, Qt::AnchorLeft, a, Qt::AnchorRight);
```

Элементы сцены (QGraphicsItem) может подвергаться преобразованиям масштабирования, переноса, поворота, линейным искажениями. При этом преобразование родительского элемента применяется и к дочерним. В следующем примере строится набор подчиненных друг другу прямоугольников и выполняется преобразование поворота для каждого из них.

Пример 7.9

```
QGraphicsRectItem *rootItem =
    new QGraphicsRectItem(...);
rootItem->setBrush(Qt::darkGray);
rootItem->setRotation(17);

QGraphicsRectItem *midItem =
    new QGraphicsRectItem(..., rootItem);
midItem->setBrush(Qt::gray);
midItem->setRotation(17);

QGraphicsRectItem *topItem =
    new QGraphicsRectItem(..., midItem);
topItem->setBrush(Qt::lightGray);
topItem->setRotation(17);
```

Результат выполнения кода представлен на рис. 17

Анимация объектов QGraphicsItem обеспечивается возможностями QGraphicsItemAnimation. При этом изменения матрицы преобразований объекта планируются в определенные моменты времени, а

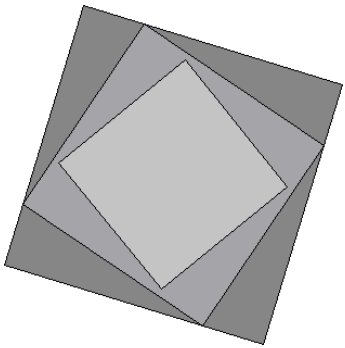

Рис. 17: Преобразование подчиненных элементов

изменения свойств от момента к моменту вычисляются с помощью линейной интерполяции. Временные отсчеты задаются на интервале $[0.0 \ldots 1.0]$, где нулевое значение соответствует состоянию объекта перед стартом анимации, а единица — по завершению. Отсчет времени обеспечивается классом QTimeLine, сигнал valueChanged() которого связывается со слотом setStep() класса QGraphicsItemAnimation (см. пример 7.10).

Пример 7.10

```
// Создаем элемент и таймер
QGraphicsItem *ball =
    new QGraphicsEllipseItem(0, 0, 20, 20);

// Создаем таймер
QTimeLine *timer = new QTimeLine(5000);
timer->setFrameRange(0, 100);

// Создаем анимацию
QGraphicsItemAnimation *animation =
    new QGraphicsItemAnimation;
animation->setItem(ball);
```

```
animation->setTimeLine(timer);

// Выставляет временные отсчеты
for (int i = 0; i < 200; ++i)
animation->setPosAt(i / 200.0, QPointF(i, i));

// Создаем и визуализируем сцену
QGraphicsScene *scene = new QGraphicsScene();
scene->setSceneRect(0, 0, 250, 250);
scene->addItem(ball);

QGraphicsView *view = new QGraphicsView(scene);
view->show();

timer->start();
```

В интерактивных приложениях, таких как компьютерные игры, достаточно часто приходится выполнять проверку касания или наложения элементов сцены — отслеживать так называемые коллизии. Например, может быть необходимо контролировать невозможность перемещения управляемого игроком персонажа сквозь стену или обрабатывать столкновение с виртуальным противником.

Далее перечислим набор методов QGraphicsItem, предоставляющих удобный интерфейс для контроля коллизий.

Пример 7.11

```
// Задает контур объекта для контроля пересечения
// если не задан, ограничивающий прямоугольник
virtual QPainterPath shape() const

// Проверяет, попадает ли указанная точка в контур
virtual bool contains(const QPointF & point) const

// Проверяет, имеет ли место коллизия с заданным элементом
virtual bool collidesWithItem(const QGraphicsItem * other,
Qt::ItemSelectionMode mode = Qt::IntersectsItemShape) const
```

```
// Проверяет, имеет ли место коллизия с заданным контуром
virtual bool collidesWithPath(const QPainterPath & path,
Qt::ItemSelectionMode mode = Qt::IntersectsItemShape) const

// Возвращает список элементов, с которыми имеется коллизия
QList<QGraphicsItem *> collidingItems(
Qt::ItemSelectionMode mode = Qt::IntersectsItemShape) const
```

7.3 Создание собственных элементов сцены

Программист может создавать собственные производные от класса QGraphicsItem классы. При этом, как минимум, необходимо переопределить метод boundingRect(), определяющий ограничивающий прямоугольник и метод paint(), отвечающий за отрисовку элемента.

Пример 7.12

```
class BasicItem : public QGraphicsItem
{
  public:
    BasicItem(QGraphicsItem *parent=0);
    QRectF boundingRect() const;
    void paint(QPainter *painter,
        const QStyleOptionGraphicsItem *option,
        QWidget *widget);
};

QRectF BasicItem::boundingRect() const
{
    return QRectF(0, 0, 100, 100);
}
void BasicItem::paint(QPainter *painter,
    const QStyleOptionGraphicsItem *option,
    QWidget *widget)
{
    painter->setPen(Qt::NoPen);
    QRadialGradient gradient = radialGradient();
```

```
    painter->setBrush(gradient);
    painter->drawEllipse(boundingRect());
}
```

Метод paint() вызывается автоматически, однако при необходимости можно вызвать метод update() для перерисовки. Следует помнить, что ограничивающий прямоугольник должен содержать все элементы (размеры которых могут динамически изменяться).

Интерактивные возможности элементов, то есть способы, которыми с ними можно взаимодействовать, определяются флагами:

- ItemIsMovable —- элемент можно перемещать мышью;
- ItemIsSelectable —- элемент может быть выделен посредством методов setSelected() и QGraphicsScene::setSelectionArea();
- ItemIsFocusable —- элемент может получить фокус клавиатуры.

Пример 7.13 демонстрирует применение флага ItemIsMovable. При перемещении элемент получает события об изменении положения.

Пример 7.13

```
QGraphicsItem *item;
item = scene->addRect(...);
item->setFlag(QGraphicsItem::ItemIsMovable, true);
```

В следующем примере продемонстрировано, как можно использовать отрисовку для подсвечивания выделенных элементов. При выделении и снятии выделения элемент получает автоматический запрос на перерисовку.

Пример 7.14

```
item->setFlag(QGraphicsItem::ItemIsSelectable, true);

void BasicItem::paint(...)
{
    if(isSelected())
```

```
        painter->setPen(Qt::black);
    else
        painter->setPen(Qt::NoPen);
}
```

Возможность перемещения и выделения элементов реализована по умолчанию. Чтобы получить полный контроль, можно напрямую переопределить обработку событий:

- setAcceptHoverEvents — наведение курсора мыши;
- setAcceptTouchEvents — касание;
- setAcceptedMouseButtons — допустимые кнопки мыши.

В следующем примере элемент отслеживает события наведения и нажатия кнопки мыши: при наведении элемент расширяется, при нажатии меняется изображение.

Пример 7.15

```
class InteractiveItem : public QGraphicsItem
{
  public:
    InteractiveItem(QGraphicsItem *parent=0);
    QRectF boundingRect() const;
    void paint(...);
  protected:
    // Переопределяемые обработчики событий
    void hoverEnterEvent(QGraphicsSceneHoverEvent*);
    void hoverLeaveEvent(QGraphicsSceneHoverEvent*);
    void mousePressEvent(QGraphicsSceneMouseEvent*);
    void mouseReleaseEvent(QGraphicsSceneMouseEvent*);
  protected:
    // Переменные для хранения состояния элемента
    bool m_pressed;
    bool m_hovered;
};

InteractiveItem::InteractiveItem(QGraphicsItem *parent) :
```

```
    QGraphicsItem(parent),
    m_pressed(false), m_hovered(false)
{
    // В конструкторе разрешим обработку событий наведения
    setAcceptHoverEvents(true);
}

QRectF InteractiveItem::boundingRect() const
{
    // Ограничивающий прямоугольник меняется
    // при наведении курсора
    if(m_hovered)
        return QRectF(-50, -50, 101, 101);
    else
        return QRectF(-30, -30, 61, 61);
}

void InteractiveItem::paint(QPainter *painter,
    const QStyleOptionGraphicsItem *option, QWidget *widget)
{
    QRadialGradient gradient;
    if(m_hovered)
        ... // Отрисовываем элемент в состоянии наведения
    if(m_pressed)
        ... // Отрисовываем элемент в состоянии нажатия
        ...
}

// Реализация обработчиков событий

void InteractiveItem::mousePressEvent(
    QGraphicsSceneMouseEvent*)
{
    m_pressed = true;
    update();
}
void InteractiveItem::mouseReleaseEvent(
    QGraphicsSceneMouseEvent*)
```

```
{
    m_pressed = false;
    update();
}

void InteractiveItem::hoverEnterEvent(
    QGraphicsSceneHoverEvent*)
{
    if(!m_hovered) {
        prepareGeometryChange();
        m_hovered = true;
    }
}

void InteractiveItem::hoverLeaveEvent(
    QGraphicsSceneHoverEvent*)
{
    if(m_hovered) {
        prepareGeometryChange();
        m_hovered = false;
    }
}
```

Класс QGraphicsItem не наследует от QObject, т. е. не можеть имеет свойств, не определяет слотов, не порождает сигналов и т. д. Так сделано из соображений экономии памяти и производительности, Однако, на практике могут возникать ситуации, в которых элементам сцены полезно было бы обладать возможностями QObject. Для этих целей определен класс QGraphicsObject, наследующий от QObject и QGraphicsItem.

Модифицируем предыдущий пример: заменим базовый класс на QGraphicsObject и в обработчике нажатия кнопки мыши будем отправлять сигнал, если в момент нажатия курсор находился в вписанном в ограничивающий прямоугольник эллипс.

Пример 7.16

```
// Фрагмент конструктора
class InteractiveObject : public QGraphicsObject
{

  Q_OBJECT

  public:
    explicit InteractiveObject(QGraphicsItem *parent = 0);
    ...
}

// Фрагмент обработчика нажатия кнопки мыши
QPointF delta = boundingRect().center()-ev->pos();
qreal radius = boundingRect().width()/2.0;
if (delta.x() * delta.x() + delta.y() * delta.y()
    <= radius * radius)
emit clicked();
...
```

§8. Интернационализация приложений

Большинство современных программ создается в расчете на широкую аудиторию, включающую пользователей из различных стран. Поэтому важно предусмотреть механизмы простой адаптации интерфейса программы к особенностям другого языка. Первостепенной задачей интернационализации приложения является перевод всех видимых пользователю текстов.

Qt обеспечивает поддержку интернационализации, обеспечивая соответствующие программные интерфейсы и предоставляя программисту и переводчику набор специальных утилит: приложение с графическим интерфейсом Qt Linguist и консольные программы lupdate и lrelease.

Общий алгоритм подготовки многоязычной версии приложения можно сформулировать следующим образом:

1. написать код программы, подготовленный к переводу;
2. извлечь строки текста из кода программы;
3. перевести извлеченные строки;
4. скомпилировать перевод в специальный бинарный формат.

На первом этапе необходимо все переводимые строки провести через вызов функции tr(). Следует заметить, что все строки в объектах, сконструированных визуально, с помощью дизайнера форм будут автоматически подготовлены к локализации.

Пример 8.1

```
// Было
trayIcon.showMessage("Message", "Mail!");
// Заменим на
trayIcon.showMessage(tr("Message"), tr("Mail!"));
```

Функция tr() возвращает QString. Подготовить символьный массив для последующей передачи в tr() можно с помощью макроса QT_TR_NOOP

Пример 8.2

```
char *const EDITOR = QT_TR_NOOP("emacs");

void function() {
    ...
    QString editor = tr(EDITOR);
    ...
}
```

Для того, чтобы извлечь строки из программы, добавим в файл проекта директиву TRANSLATIONS, перечислив все желаемые файлы переводов.

Пример 8.3

```
TRANSLATIONS = myProject_de.ts \
               myProject_fr.ts \
               myProject_ru.ts
```

Запустив программу lupdate в каталоге программы, получим шаблоны файлов переводов:

Пример 8.4

```
user@linux:~/myProject> lupdate myProject.pro
```

Файлы переводов можно открыть в программе Qt Linguist и для каждой извлеченной из исходного кода строки предложить перевод. После завершения перевода строк, необходимо выполнить команду lrelease (можно из меню Qt Linguist). Будут сгенерированы файлы с расширением «.qm». Это бинарные файлы переводов, они должны быть доступны при загрузке приложения.

Пример 8.5

```
user@linux:~/myProject> lrelease myProject.pro
```

Для того, чтобы не заботиться об этом, бинарные файлы переводов могут быть внедрены в исполняемый файл с помощью ресурсной системы Qt.

В коде программы необходимо обеспечить возможность подгрузки файлов переводов, внедрив код переводчика.

Пример 8.6

```
// Необходимые файлы заголовков
#include <QLocale>
#include <QTranslator>

// Создаем экземпляр переводчика
QApplication app;
QTranslator myTranslator;
myTranslator.load("myProject_" + QLocale::system().name(),
                  QCoreApplication::applicationDirPath());
app.installTranslator(&myTranslator);
```

Предложенный код подключает файл переводов, соответствующий системной локали, что, как правило, соответствует предпочтениям пользователя. В некоторых случаях, приложение может предоставлять пользователю возможность изменения языковых настроек «на лету». В этом случае построенный в дизайнере форм интерфейс можно переключить на другой язык вызвав метод retranslateUi() в обработчике события LanguageChange.

Пример 8.7

```
void Widget::changeEvent(QEvent *e)
{
    if (e->type() == QEvent::LanguageChange) {
        ui->retranslateUi(this);
    }
    QWidget::changeEvent(e);
}
```

В некоторых случаях бывает необходимо локализовать ресурсы, например, если приложение сопровождается несколькими наборами иконок для разных языковых настроек. Ресурсная система Qt позволяет указывать локаль для каждого присоединяемого к исполняемой программе файла ресурсов.

Пример 8.8

```
<qresource lang="ru">
    <file>File.png</file>
</qresource>
```

Инструменты перевода позволяют задействовать контекст, чтобы различать строки, переводимые по-разному в разных обстоятельствах. Например, английское слово «Edit» может быть как существительным («Правка»), так и глаголом («Редактировать»). При подготовке текста к переводу в вызове tr() для определения контекста можно задействовать второй аргумент (см. пример **??**.

Пример 8.9

```
tr("edit", "noun");
tr("edit", "verb");
```

Контекст будет доступен переводчику в Qt Linguist, помогая выбрать корректный вариант перевода.

Список литературы

[1] Ж. Бланшет, М. Саммерфилд. Qt 4: программирование GUI на C++. Пер. с англ. 2-е изд., доп. — М.: КУДИЦ-ПРЕСС, 2008. — 736 с.

[2] М. Шлее М. Qt4.5. Профессиональное программирование на C++. Пер. с англ. — СПб.: БХВ-Петербург, 2010. - 896 с.

[3] A. Ezust, Alan. An introduction to design patterns in C++ with QT / Alan Ezust, Paul Ezust. — Pearson Education, 2012. — 764 c.

[4] D. Molkentin. The Book of Qt 4: The Art of Building Qt Applications. — Open Source Press, 2007. — 430 c.

[5] J. Thelin. Foundations of Qt Development. — APress, 2007. — 534 c.

[6] Qt Developer Network [Электронный ресурс]. 2012. Режим доступа: http://qt-project.org/doc/

[7] Qt in Education Course Materials [Электронный ресурс]. 2011. Режим доступа: http://qt.nokia.com/learning/education/course-materials/

Подписано в печать 03.12.12. Формат 60×84 1/16
Бумага офсетная. Уч.-изд. л. 4. Тираж 100 экз. Изд. №175

Отпечатано в типографии Издательства ПетрГУ
185910, г. Петрозаводска, пр. Ленина, 33