Wstęp do programowania w Qt

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Wstęp do programowania w Qt

Karol Kuczyński
Paweł Mikołajczak
Marcin Denkowski
Rafał Stęgierski
Krzysztof Dmitruk
Michał Pańczyk

UMCS
UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ

Lublin 2012

Instytut Informatyki UMCS
Lublin 2012

Karol Kuczyński
Paweł Mikołajczak
Marcin Denkowski
Rafał Stęgierski
Krzysztof Dmitruk
Michał Pańczyk

Wstęp do programowania w Qt

Recenzent: Jakub Smołka

Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
FIGARO Group Sp. z o.o. z siedzibą w Rykach
ul. Warszawska 10
08-500 Ryki
www: www.figaro.pl

ISBN: 978-83-62773-26-8

Spis treści

Przedmowa (M. Denkowski) **vii**

1 Witaj świecie (M. Denkowski) **1**
1.1. Wstęp 2
1.2. Pierwszy program w Qt 2
1.3. Program qmake 8
1.4. Środowisko Qt Creator 10

2 Tworzenie graficznego interfejsu użytkownika (M. Pańczyk) **17**
2.1. Wstęp 18
2.2. Widgety 18
2.3. Okno główne, menu, pasek narzędzi i pasek statusu 32

3 Sloty, sygnały, zdarzenia (K. Kuczyński) **37**
3.1. Wstęp 38
3.2. Sloty i sygnały 38
3.3. Zdarzenia w Qt 47

4 Grafika 2D (M. Denkowski) **53**
4.1. Wstęp 54
4.2. Rysowanie rastrowe 55
4.3. Rysowanie Painterem 61
4.4. Przeglądarka obrazów 68

5 Grafika 3D (R. Stęgierski) **79**
5.1. Wstęp 80
5.2. Widget OpenGL 80

6 Operacje wejścia/wyjścia (R. Stęgierski) **87**
6.1. Wstęp 88
6.2. Styl C 88
6.3. Styl C++ 91

6.4. Dialog 94
6.5. Obsługa systemu plików 96

7 Operacje sieciowe (K. Kuczyński) **99**
7.1. Wstęp 100
7.2. Klient FTP 100
7.3. Klient HTTP 105

8 Architektura model/widok (K. Dmitruk) **109**
8.1. Wzorzec projektowy MVC 110
8.2. Implementacja architektury model/widok w Qt 111
8.3. Przegląd klas widoku 113
8.4. Komunikacja między modelem a widokiem 116
8.5. Baza danych w architekturze model/widok 131

9 Kontenery i algorytmy w Qt (P. Mikołajczak) **141**
9.1. Wstęp 142
9.2. Kontenery Qt 142
9.3. Kontenery sekwencyjne 143
9.4. Kontenery asocjacyjne 148
9.5. Iteratory STL 150
9.6. Iteratory Qt 153
9.7. Algorytmy Qt 159

Bibliografia **167**

Przedmowa

Qt jest zespołem bibliotek umożliwiających tworzenie wszechstronnych aplikacji w języku C++. Stanowią one doskonałą bazę dla wieloplatformowych aplikacji z zaawansowanym graficznym interfesjem użytkownika. Wieloplatformowość w tym kontekście należałoby rozumieć jako zdolność do kompilacji kodu źródłowego danego rozwiązania na każdej obsługiwanej platformie bez wprowadzania jakichkolwiek zmian. W chwili obecnej wśród obsługiwanych dużych systemów jest Linux, BSD, Windows, MacOS oraz wiele mniejszych mobilnych platform typu SymbianOS, MeeGo czy Windows CE.

Qt jest dostępny pod dwoma licencjami: komercyjną, wymagającą wykupienia ale umożliwiającą produkcję zamkniętego oprogramowania oraz w wersji wolnej opartej na licencji GPL. Na ten moment, chyba najbardziej rozbudowanym projektem ukazującym możliwości biblioteki Qt jest K Desktop Environment (KDE).

Celem niniejszego podręcznika jest wprowadzenie czytelnika w podstawowe pojęcia i mechanizmy bibliotek Qt oraz nauka wykorzystania tych mechanizmów przy tworzeniu bardziej skomplikowanych rozwiązań. ie wyczerpuje jednak wszystkich możliwości biblioteki a autorzy skupili się na, ich zdaniem, najistotniejszych elementach biblioteki.

Układ książki z grubsza odpowiada kolejności w jakiej czytelnik powinien się zapoznawać z kolejnymi zagadnieniami biblioteki. Pierwszy rozdział stanowi wprowadzenie do programowania w Qt razem ze standardowym programem „Witaj świecie" oraz dołączonymi do pakietu bibliotek narzędziami. Drugi rozdział opisuje jedną z najsilniejszych stron biblioteki Qt, mianowicie jej potężne możliwości budowy graficznego interfejsu użytkownika (GUI). Trzeci rozdział zawiera wprowadzenie do mechanizmu sygnałów i slotów. Jest to charakterystyczny dla tej biblioteki i jednocześnie wyróżniający ją z pośród innych rozwiązań mechanizm komunikacji pomiędzy obiektami. Po tych rozdziałach czytelnik powinien posiadać już wystarczającą wiedzę do pisania prostego oprogramowania z użyciem biblioteki Qt. Dwa następne rozdziały zawierają wprowadzenie do programowania grafiki dwuwymiarowej i trójwymiarowej z użyciem biblioteki OpenGL. Rozdział 6 opisuje klasy

i metody dostępu do zasobów dyskowych i operacji wejścia/wyjścia. Rozdział 7 zawiera omówienie metod niezbędnych przy budowie oprogramowania wykorzystującego komunikację sieciową na przykładzie protokołów http i ftp. Rozdział 8 opisuje metody spajające interfejs użytkownika z całą logiką aplikacji w modelu widok-kontroler. Ostatni rozdział stanowi niejako dodatek opisujący klasy kontenerowe i algorytmy na nich operujące zawarte w bibliotece Qt pomyślane wzór standardowej biblioteki szablonów STL.

Książka stanowi podręcznik dla studentów kierunku informatyka oraz wszystkich programistów chcących rozpocząć pisanie aplikacji przy użyciu bibliotek Qt. Do poprawnego zrozumienia podręcznika wymagana jest przynajmniej podstawowa znajomość języka C++.

Niniejszy podręcznik opiera się na wersji 4.5 bibliotek Qt.

Rozdział 1

Witaj świecie

1.1. Wstęp **2**
1.2. Pierwszy program w Qt **2**
1.2.1. Prosta aplikacja 2
1.2.2. Aplikacja konsolowa 4
1.2.3. Aplikacja z połączeniami 4
1.3. Program qmake **8**
1.3.1. Użycie qmake 8
1.3.2. Plik projektu 9
1.4. Środowisko Qt Creator **10**

1.1. Wstęp

Qt jako zespół bibliotek został stworzony w 1995 roku przez norweską firmę *Quasar Technologies*, przemianowaną następnie na *Trolltech*. Po przejęciu praw większościowych przez fińską *Nokię* spółka *Trolltech* zmieniła po raz kolejny nazwę na *Qt Software* i ostatecznie w 2009 na *Qt Development Frameworks*. Aktualna wersja 4 środowiska jest rozprowadzana w dwóch typach licencji: komercyjnej i wolnej. W skład całego pakietu poza samymi bibliotekami wchodzi jeszcze szereg narzędzi:

- qmake – program wspomagający tworzenie międzyplatformowych plików projektów,
- moc – dodatkowy preprocesor generujący pliki cpp dla specjalnych struktur Qt,
- uic – preprocesor plików form generowanych przez Designer,
- Designer – aplikacja ułatwiająca tworzenie interfejsu użytkownika,
- Linguist – aplikacja wspomagająca proces lokalizacji,
- Assistant – system pomocy.

Całą platformę spina środowisko IDE o nazwie *Qt Creator*.

1.2. Pierwszy program w Qt

1.2.1. Prosta aplikacja

Zacznijmy od najprostszego programu wykorzystującego model okienkowy, którego celem będzie wyświetlenie standardowego tekstu "Witaj świecie":

Listing 1.1. Program Witaj świecie.

```
1  #include <QApplication>
2  #include <QLabel>
3
4  int main(int argc, char* argv[])
5  {
6    QApplication app(argc, argv);
7    QLabel *label = new QLabel(QObject::tr("Witaj swiecie"));
8    label->show();
9    return app.exec();
10 }
```

Pierwsze dwie linie zawierają polecenia włączenia plików nagłówkowych z definicjami klas `QApplication` oraz `QLabel`. W środowisku Qt każdy plik nagłówkowy zawierający definicję klasy ma nazwę identyczną z nazwą tej klasy. Klasa `QApplication` stanowi trzon każdej aplikacji wykorzystującej

model zdarzeniowy Qt. Jej instancja jest tworzona jako obiekt automatyczny w linii 6:

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

Obiektowi temu przekazywane są w parametrze konstruktora parametry wywołania programu `argc` i `argv`. W całym programie Qt może istnieć co najwyżej jedna istancja `QApplication`, co więcej, jest ona dostępna zawsze w kodzie programu pod zmienną `qApp` lub zamiennie, zwracana przez statyczną metodę:

```
QCoreApplication *QCoreApplication:instance()
```

Klasa `QCoreApplication` jest klasą bazową dla `QApplication`.

W linii 7 tworzony jest dynamicznie obiekt klasy `QLabel`. Klasa ta jest pochodną tak zwanego *widgetu* (skrót od angielskiej nazwy *window gadget*), czyli wizualnego elementu interfejsu. W tym przypadku jest to prosta kontrolka wyświetlająca tekst. Użyta w konstruktorze klasy `QLabel` statyczna metoda klasy `QObject`:

```
QString QObject::tr(const char* text)
```

zwraca przetłumaczoną wersję napisu `text` w zależności od aktualnej lokalizacji programu.

Linia 8 powoduje wyświetlenie obiektu `label` na ekranie. Domyślnie wszystkie widgety są konstruowane jako niewidoczne (*hidden*).

Cały program rozpoczyna swoje działanie dopiero w linii 9, w której obiekt `app` przechodzi w stan nieskończonej pętli obsługi zdarzeń pochodzących zarówno z systemu jak i od użytkownika aplikacji. Przerwanie tej pętli jest równoznaczne z zakończeniem programu.

Tak napisany kod źródłowy należy jeszcze skompilować dołączając odpowiednie biblioteki. Platforma Qt ułatwia to zadanie dostarczając program o nazwie `qmake` generujący odpowiedni plik projektu. W celu kompilacji programu należy wydać następujące komendy:

```
qmake -project
qmake
make
```

Pierwsze polecenie utworzy plik opisujący projekt Qt w sposób niezależny od systemu. Domyślnie będzie on miał nazwę aktualnego katalogu z rozszerzeniem `.pro`. Drugie polecenie na podstawie pliku projektu utworzy odpowiedni plik `Makefile`, a trzecie polecenie wykona reguły kompilacji zawarte w tym pliku.

Uruchomienie tego programu spowoduje utworzenie nowego okna z kontrolką zawierającą napis “Witaj swiecie” (zobacz Rysunek 1.1).

Rysunek 1.1. Wynik działania programu Witaj świecie.

1.2.2. Aplikacja konsolowa

Biblioteki Qt umożliwiają również konstrukcję programów konsolowych bez obciążania systemu graficznym interfejsem ale korzystających z mechanizmu slotów i sygnałów oraz zdarzeń. W takim przypadku zamiast klasy `QApplication` należy jako trzon aplikacji użyć klasy `QCoreApplication`.

Listing 1.2. Program konsolowy.

```
1  #include <QCoreApplication>
2  #include <QFile>
3  #include <QTextStream>
4  #include <iostream>
5
6  int main(int argc, char* argv[])
7  {
8    QCoreApplication app(argc, argv);
9    QFile file("core_example.cpp");
10   file.open(QIODevice::ReadOnly);
11   QTextStream out(&file);
12   QString file_text = out.readAll();
13   std::cout << file_text.toAscii().data();
14   return app.exec();
15 }
```

Powyższy program tworzy aplikację wczytującą plik tekstowy o nazwie „core_example.cpp” do obiektu typu `QString` a następnie wyświetlającą na konsoli ten tekst za pomocą obiektu `cout` standardowej biblioteki `iostream`. Szersze omówienie metod wejścia/wyjścia znajduje się w rozdziale 6.

1.2.3. Aplikacja z połączeniami

Poniżej został przedstawiony nieco bardziej rozbudowany program wykorzystujący wspomniany już mechanizm slotów i sygnałów oraz prezentujący możliwości widgetów jako kontrolek oraz kontenerów na inne widgety.

Listing 1.3. Prosta Aplikacja.

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QPushButton>
4  #include <QSlider>
5  #include <QSpinBox>
6  #include <QHBoxLayout>
7
8  int main(int argc, char* argv[])
9  {
10   QApplication app(argc, argv);
11   QWidget *mainWidget = new QWidget();
12   mainWidget->setWindowTitle(QObject::tr(
13                              "Simple Application"));
14
15   QPushButton *exitButton = new QPushButton(
16                           QObject::tr("Exit"), mainWidget);
17
18   QSlider *slider = new QSlider(Qt::Horizontal,mainWidget);
19   slider->setRange(0, 50);
20
21   QSpinBox * spin = new QSpinBox(mainWidget);
22
23   QHBoxLayout *layout = new QHBoxLayout(mainWidget);
24   layout->addWidget(exitButton);
25   layout->addWidget(slider);
26   layout->addWidget(spin);
27
28   QObject::connect(exitButton, SIGNAL(clicked()),
29                    &app, SLOT(quit()));
30   QObject::connect(slider, SIGNAL(valueChanged(int)),
31                    spin, SLOT(setValue(int)));
32
33   mainWidget->show();
34   return app.exec();
35 }
```

Wynik działania programu został przedstawiony na rysunku 1.2.

Rysunek 1.2. Wynik działania programu 1.3.

- Linie 1-6 włączają pliki nagłówkowe z definicjami odpowiednich klas Qt.

- W linii 11 tworzony jest obiekt klasy `QWidget` stanowiącej podstawową klasę, z której wywodzą się wszystkie wizualne kontrolki Qt. W tym przypadku obiekt `mainWidget` zostanie wykorzystany w roli kontenera na inne kontrolki i jednocześnie będzie stanowił główne okno aplikacji.

- W linii 12 wywoływana jest metoda `QWidget::setWindowTitle(QString)` nadająca odpowiednią nazwę w pasku tytułu okna aplikacji. Wykorzystana została tutaj wspomniana już wyżej statyczna metoda `tr(const char*)` umożliwiająca proste lokalizowanie aplikacji.

- W linii 15 tworzony jest obiekt `exitButton` klasy `QPushButton` reprezentującej standardowy okienkowy przycisk za pomocą konstruktora:

  ```
  QPushButton(const QString &text, QWidget* parent)
  ```

  W pierwszym parametrze konstruktora został podany napis, który zostanie wyświetlony na przycisku. Drugi parametr `parent` jest tzw. rodzicem danego obiektu. W tym przypadku rodzicem przycisku będzie obiekt `mainWidget`, który staje się jednocześnie kontenerem zawierającym ten przycisk. W przypadku przekazania w parametrze `parent` wartości `NULL` dla rodzica, widget taki staje się automatycznie autonomicznym oknem z całym obramowaniem oraz belką tytułową. Niejawnie zostało to wykonane w linii 11, ponieważ konstruktor klasy `QWidget` jako pierwszy parametr przyjmuje właśnie wartość rodzica z domyślną wartością `NULL`.

  ```
  QWidget(QWidget* parent = NULL, Qt::WindowFlags f = 0)
  ```

  Nadanie niezerowej wartości parametrowi `parent` ma jeszcze jedną zaletę. Obiekt taki jest zarządzany w kontekście pamięci przez swojego rodzica. To zwalnia programistę z ręcznego usuwania takiego obiektu, gdyż zostanie on skasowany przez obiekt rodzica w momencie usuwania z pamięci samego rodzica.

- Linia 18 tworzy dynamicznie następny widget o nazwie `slider` klasy `QSlider` za pomocą konstruktora:

  ```
  QSlider(Qt::Orientation orientation, QWidget* parent = NULL)
  ```

  Kontrolka ta reprezentuje klasyczny suwak o wartościach całkowitoliczbowych. W konstruktorze, w pierwszym parametrze podawana jest wartość typu wyliczeniowego `Qt::Horizontal` decydująca o poziomym ułożeniu suwaka. W drugim parametrze ustawiony zostaje rodzic widgeta na `mainWidget`. W 19 linii zakres możliwych wartości suwaka jest ustalany na [0..50].

- W linii 21 tworzony jest następny widget klasy `QSpinBox` reprezentujący kontrolkę tekstową umożliwiającą wprowadzanie wartości całkowitoliczbowych za pomocą klawiatury lub strzałek. W jedynym parametrze konstruktora przekazany zostaje obiekt `mainWindow` jako rodzic tego widgetu.

- W linii 23 tworzony jest obiekt typu *layout*, czyli swoisty układacz widgetów na innym widgecie stanowiącym dla nich kontener. W tym przypadku jest to obiekt klasy `QHBoxLayout`, który rozkłada widgety w poziomie jeden za drugim, od lewej do prawej strony. W liniach 24-26 do tego układacza dodawane są kolejno widgety `exitButton`, `slider` i `spin` za pomocą metody:

  ```
  QHBoxLayout::addWidget(QWidget * widget,
                         int stretch = 0,
                         Qt::Alignment alignment = 0)
  ```

  Dwa ostanie parametry metody decydują o sposobie rozciagania zawieranych widgetów. Kolejność dodawania jest istotna, gdyż w takiej kolejności układacz będzie rozkładał poszczególne widgety.

- Linie 28-31 pokazują główną siłę biblioteki Qt – mechanizm slotów i sygnałów. Każdy obiekt wywodzący się z klasy `QObject` poza standardowymi mechanizmami C++ może być wyposażony w sygnały i sloty. Sygnały z reguły wysyłane są w momencie wystąpienia jakiegoś zdarzenia (np. naciśnięcie przycisku) natomiast sloty odpowiedzialne są za obsługę wysyłanych sygnałów. Dany sygnał można połączyć z odpowiednim slotem za pomocą statycznej metody klasy `QObject`:

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

  Pierwszy parametr jest obiektem wywodzącym się z klasy `QObject` wysyłającym sygnał. Sygnał jest identyfikowany przez swoją nazwę w parametrze `signal`. Analogicznie parametr `reciever` jest potomkiem klasy `QObject` odbierającym sygnał i wywołującym metodę–slot, również identyfikowaną przez nazwę w parametrze `method`. Ostatni parametr wpływa na sposób wywoływania metod–slotów w kontekście wielu wątków i kolejkowania wywołań. Domyślna wartość `Qt::AutoConnection` powoduje natychmiastowe synchroniczne wywołanie metody–slotu w przypadku wysłania odpowiedniego sygnału.
  Analizując przykładowe połączenia z omawianego listingu, w linii 28 zawierającej wywołanie:

  ```
  QObject::connect(exitButton, SIGNAL(clicked()),
  ```

```
                          &app, SLOT(quit()));
```

obiektem wysyłającym sygnał jest `exitButton`, który może wysyłać sygnał o nazwie `clicked()`, natomiast odbierającym sygnał jest obiekt aplikacji `app` w slocie `quit()`. Zarówno dla sygnałów jak i slotów w metodzie `connect()` zdefiniowane są makra `SIGNAL()` oraz `SLOT()` tłumaczące odpowiednie nazwy na wewnętrzną reprezentację Qt. Takie połączenie sygnału `clicked()` przycisku `exitButton` z metodą-slotem `quit()` aplikacji `app` sprawia, że naciśnięcie przycisku powodujące wysłanie sygnału, który spowoduje wywołanie metody `quit()`, powodując tym samym zakończenie aplikacji.
W drugim przykładzie połączenia:

```
QObject::connect(slider, SIGNAL(valueChanged(int)),
                 spin, SLOT(setValue(int)));
```

obiektem wysyłającym sygnał jest suwak `slider` a obiektem odbierającym sygnał pole tekstowe `spin`. Sygnał `valueChanged(int)` jest wysyłany (wraz z wartością typu `int`) przy zdarzeniu zmiany wartości suwaka i wywoływana jest metoda–slot `setValue(int)` obiektu `spin`. Dzięki takiemu połączeniu każda zmiana położenia suwaka spowoduje odpowiednią zmianę wartości pola tekstowego.

- Linia 33 sprawia, że główny widget `mainWidget` staje się widoczny a w linii 34 rozpoczyna się pętla reakcji na zdarzenia aplikacji.

1.3. Program qmake

1.3.1. Użycie qmake

Program `qmake` jest programem automatyzującym tworzenie niezależnych od systemu plików projektu. Najprostsze użycie zostało już zaprezentowane w poprzednim podrozdziale i polega na utworzeniu pliku projektu poleceniem:

```
qmake -project
```

Takie wywołanie przegląda aktualny katalog w poszukiwaniu plików źródłowych i dodaje je do pliku projektu.

Na podstawie tak stworzonego pliku projektu można wygenerować plik konfiguracyjny dla konkretnego środowiska. Klasyczny plik `Makefile` generowany jest poprzez wywołanie:

```
qmake plik_projektu.pro
```

Podanie nazwy pliku projektu jest opcjonalne. W przypadku braku tej nazwy zostanie użyty domyślny plik projektu.

Dla środowiska Microsoft Visual C++ odpowiedni plik projektu zostanie wygenerowany za pomocą polecenia:

```
qmake -tp vc plik_projektu.pro
```

Dla środowiska Xcode na platformie Mac OS X plik projektu zostanie wygenerowany za pomocą polecenia:

```
qmake -spec macx-xcode project.pro
```

1.3.2. Plik projektu

Plik projektu może być dosyć skomplikowanym zestawem poleceń, pętli i warunków umożliwiających kompilację projektu w zależności od konfiguracji systemu. Przeanalizujmy podstawowe możliwości pliku opisującego projekt składający się z następujących plików z kodem źródłowym:

```
main.cpp
mainwidget.h
mainwidget.cpp
panel.h
panel.cpp
```

Uruchomienie programu `qmake -project` w katalogu zawierającym te pliki źródłowe powoduje utworzenie pliku projektu o następującej treści:

```
HEADERS += mainwidget.h panel.h
SOURCES += main.cpp mainwidget.cpp panel.cpp
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .
```

Zmienna `HEADERS` zawiera listę wszystkich plików nagłówkowych a zmienna `SOURCES` listę wszystkich plików źródłowych z tego katalogu.

Zmienna `TEMPLATE` zawiera nazwę szablonu dla sposobu generowania projektu. W tym przypadku wartość `app` spowoduje utworzenie klasycznej wykonywalnej aplikacji. Inne możliwe wartości to:
– `subdir` – tworzy makefile przenoszący budowę do innego katalogu
– `lib` – tworzy makefile dla budowy bibliotek

– vcapp i vclib – odpowiednio aplikacja i biblioteka dla środowiska Miscrosoft Visual C++

Zmienna TARGET zawiera nazwę tworzonego pliku wykonywalnego lub biblioteki. Pozostawienie tej zmiennej niezdefiniowanej spowoduje nadanie plikowi wykonywalnemu nazwy identycznej z nazwą projektu.

Zmienna INCLUDEPATH zawiera listę katalogów rozdzieloną spacjami, w których należy szukać plików nagłówkowych.

Istnieje wiele zdefiniowanych zmiennych rozumianych przez program qmake. Najistotniejsze z nich to:

– QT – zmienna kontroluje używane w projekcie moduły Qt. Może przyjmować wartości: core (moduł dodany domyślnie), gui (moduł dodany domyślnie), opengl, network, sql, svg, phonon, xml, webkit, qt3support.
  Przykład:
  ```
  QT += opengl xml phonon
  ```
– LIBS – zmienna zawierająca listę katalogów oraz nazw bibliotek dołączanych do projektu. Katalogi dołącza się poprzedzając ich nazwę symbolami "-L", natomiast same biblioteki poprzedzane są symbolami "-l" i zawierają jedynie rdzeń nazwy biblioteki. Rdzeniem nazwy biblioteki jest literał pozostały po odrzuceniu z nazwy przedrostka "lib" oraz rozszerzenia. Przykładowo, jeżeli biblioteka nazywa się "libmath.so" to rdzeniem jest "math".
  Przykład:
  ```
  LIBS += -L/usr/local/include -lmath
  ```
– CONFIG – zmienna kontrolująca proces kompilacji projektu. Najistotniejsze wartości to: release, debug, debug_and_release, precompile_header, warn_on, warn_off.
  Przykład:
  ```
  CONFIG += release warn_off
  ```
– RESOURCES – zmienna zawierająca listę nazw plików zasobów projektu (.qrc)
– FORMS – zmienna zawierająca listę plików UI (formularze Qt Designera) używanych w projekcie.
– TRANSLATIONS – zmienna zawierająca listę plików z translacjami (.ts) zawierającymi odpowiednie wersje językowe tekstów używanych w aplikacji.

1.4. Środowisko Qt Creator

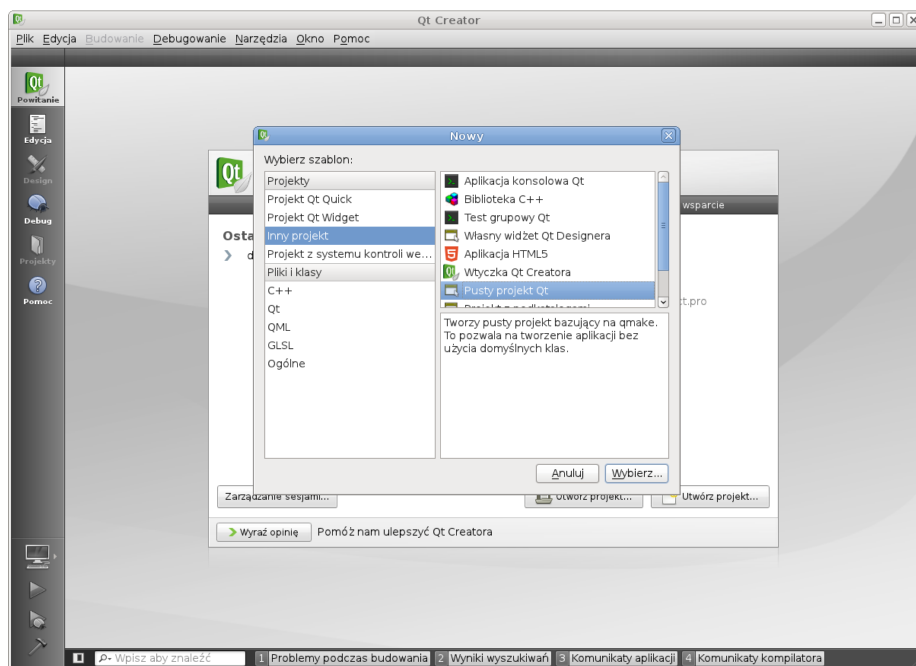
Qt Creator jest wieloplatformowym zintegrowanym środowiskiem programistycznym (ang. Integrated Development Environment (IDE)). Poza standardowymi możliwościami edycyjnymi i kompilacyjnymi zawiera wizu-

alny debugger oraz designer ułatwiający tworzenie formularzy GUI. *Qt Creator* integruje również obsługę systemów kontroli wersji takich jak GIT, Subversion czy CVS. Zostały stworzone wersje tego środowiska dla systemów Linux, Mac OS X oraz MS Windows, dzięki którym można tworzyć aplikacje na te systemy oraz na platformy mobilne. W przypadku Linuxa i jego odmian oraz systemu Mac OS X środowisko używa kompilatora GNU Compiler Collection, dla wersji MS Windows można używać MinGW lub MSVC. Dla platform mobilnych środowisko oferuje w pełni funkcjonalne emulatory.

Program po uruchomieniu wita użytkownika powitalnym ekranem z możliwością otwarcia ostatnich projektów, sesji domyślnej, otwarcia innego projektu lub utworzenia nowego projektu.

Przeanalizujmy sposób tworzenia nowego projektu i dodania do niego pliku z funkcją `main()` oraz nowej klasy Qt w wersji 2.2 środowiska *Qt Creator*.

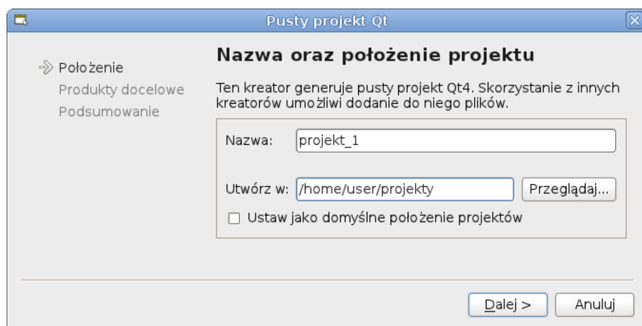
Po kliknięciu menu *Plik/Nowy plik lub projekt...* ukaże się nowe okno dialogowe:



Rysunek 1.3. Tworzenie nowego projektu Qt.

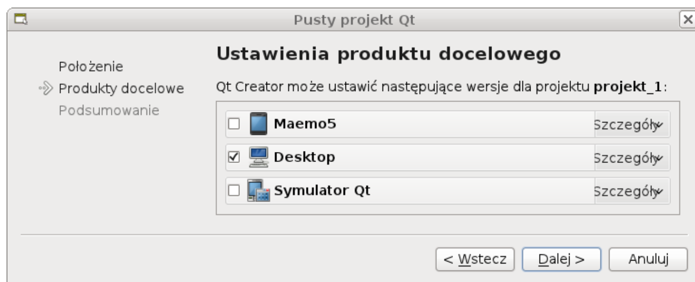
Wybieramy *Projekt/Inny projekt* i po prawej stronie *Pusty projekt Qt*. Zostanie wyświetlone nowe okno dialogowe umożliwiające zdefiniowanie na-

zwy dla projektu oraz wskazanie położenia katalogu, w którym zostanie on utworzony:



Rysunek 1.4. Tworzenie nowego projektu Qt, krok 2.

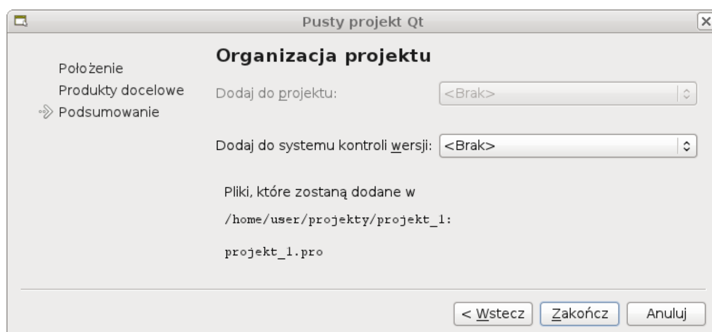
Po kliknięciu przycisku *Dalej* w oknie Ustawień projektu akceptujemy wersję projektu *Desktop*:



Rysunek 1.5. Tworzenie nowego projektu Qt, krok 3.

Tutaj można zdecydować czy ma to być wersja na komputer czy też na platformy mobilne Maemo5 lub Symbian.

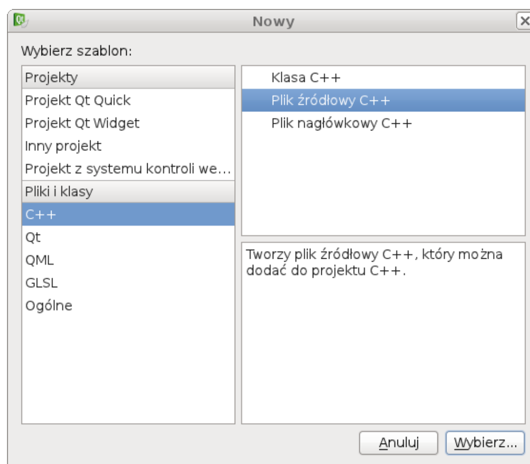
W podsumowaniu widać, że jedynym plikiem w projekcie na tym etapie jest sam plik projektu. W tym oknie można również wybrać system kontroli wersji:



Rysunek 1.6. Tworzenie nowego projektu Qt, krok 4.

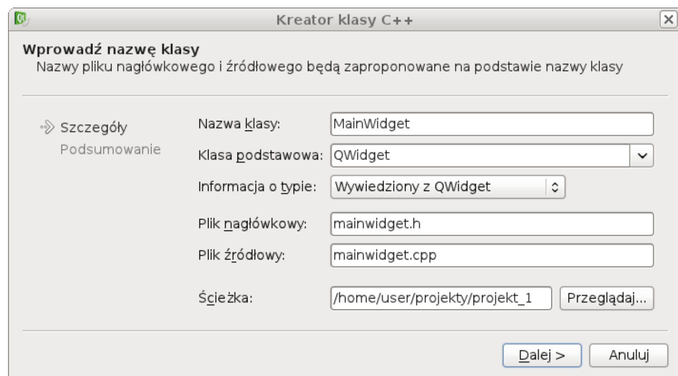
Konfiguracja systemu kontroli wersji na tym etapie spowoduje, że wszystkie pliki zarządzane przez środowisko będą objęte kontrolą.

Główny plik źródłowy zawierający funkcję `main()` można dodać do projektu klikając znowu w menu *Plik/Nowy plik lub projekt...*, wybierając w oknie dialogowym Szablon Plików i klas *C++* a następnie po prawej stronie *Plik źródłowy C++*:



Rysunek 1.7. Dodanie pliku głównego do projektu.

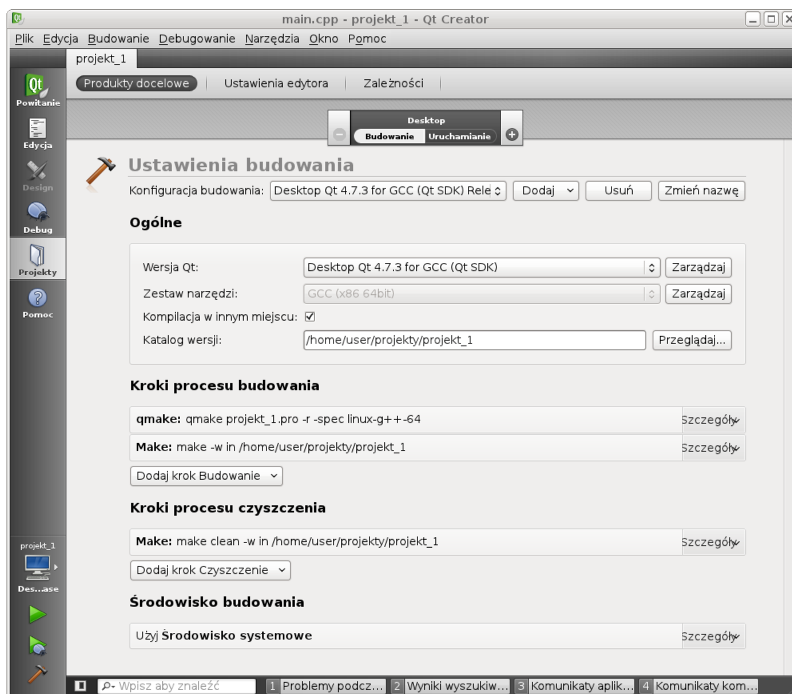
Analogicznie w menu *Plik/Nowy plik lub projekt...* można dodać nową klasę wybierając opcję *Klasa C++* w tym samym oknie. W przypadku klasy zostanie wyświetlony kreator umożliwiający zdefiniowanie pewnych własności tej klasy:



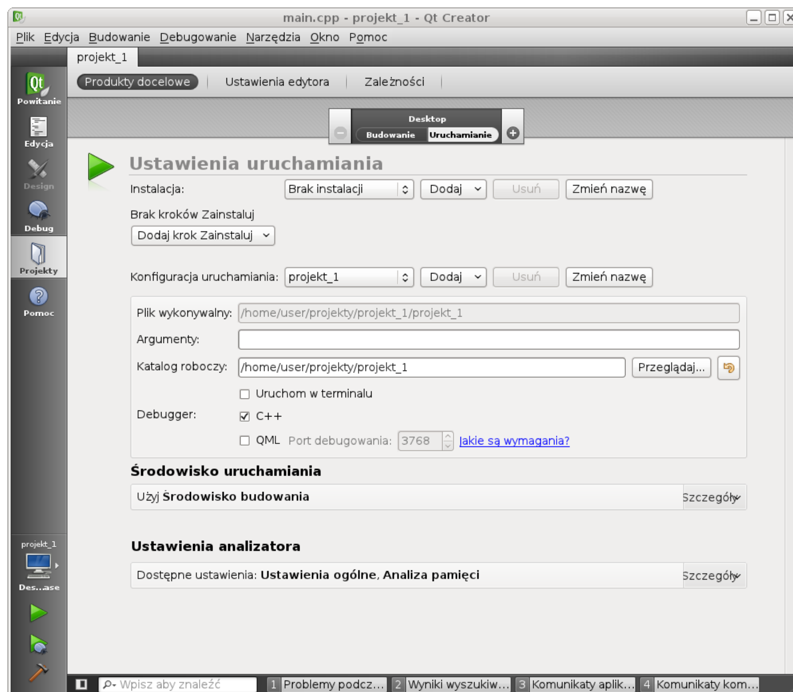
Rysunek 1.8. Dodanie klasy do projektu.

Po kliknięciu przycisku *Dalej* w oknie dialogowym utworzone zostaną dwa pliki: nagłówkowy i źródłowy zawierający szablon generowanej klasy.

Plik projektu można edytować ręcznie za pomocą wbudowanego edytora tekstowego lub za pomocą graficznego odpowiednika po kliknięciu w lewym panelu *Qt Creatora* na ikonę *Projekty*. W tym przypadku większość możliwości `qmake` jest dostępna z dwóch paneli konfiguracyjnych:



Rysunek 1.9. Panel konfiguracji budowania projektu.



Rysunek 1.10. Panel konfiguracji uruchamiania projektu.

Rozdział 2

Tworzenie graficznego interfejsu użytkownika

2.1. Wstęp **18**
2.2. Widgety **18**
2.2.1. Przegląd widgetów 18
2.2.2. Rozmieszczanie widgetów 23
2.3. Okno główne, menu, pasek narzędzi i pasek statusu . . **32**

2.1. Wstęp

Biblioteka Qt została stworzona głównie w celu łatwego tworzenia interfejsów graficznych. Podstawowym elementem wizualnym interfejsu graficznego jest wspomniany w poprzednim rozdziale „widget”. W istocie niemal wszystko, co widać na ekranie w klasycznej aplikacji Qt, jest widgetem. Każdy taki element interfejsu jest realizowany przez obiekt klasy dziedziczącej po `QWidget`.

2.2. Widgety

Funkcją widgetów jest nie tylko pojawianie się na ekranie, ale również interakcja z użytkownikiem. Przykładem może być przycisk, który można kliknąć; pole tekstowe, do którego można wprowadzać tekst itd. W niniejszym rozdziale zajmiemy się wizualną stroną widgetów. Natomiast jak zdefiniować reakcje na działania użytkownika związane z widgetami — dowiemy się z następnego rozdziału.

2.2.1. Przegląd widgetów

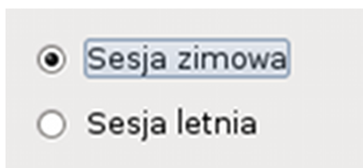
Do dyspozycji dostępnych jest wiele gotowych widgetów. Poniżej zostaną omówione niektóre z nich.

— `QPushButton`

Rysunek 2.1. Przykładowe przyciski QPushButton

Przycisk `QPushButton` służy do uruchamiania pewnej akcji poprzez kliknięcie na niego kursorem myszy.

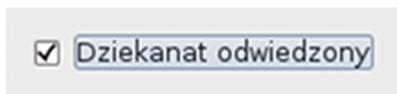
— `QRadioButton`



Rysunek 2.2. Przykładowe przyciski QRadioButton

Przycisk `QRadioButton` umożliwia użytkownikowi wybór dokładnie jednej opcji z wielu dostępnych.

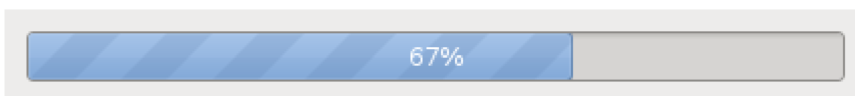
— `QCheckBox`



Rysunek 2.3. Przykładowy przycisk QCheckBox

Przycisk `QCheckBox` przyjmuje dwa stany: zaznaczony — niezaznaczony. Możliwe jest również, aby stanem było częściowe zaznaczenie.

— `QProgressBar`



Rysunek 2.4. Przykładowy pasek postępu QProgressBar

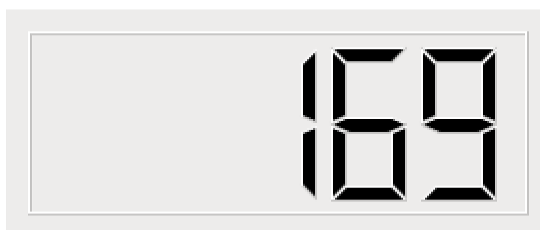
Pasek postępu `QProgressBar` umożliwia wyświetlanie w postaci wizualnej postępu wykonania dowolnej operacji trwającej dłuższy czas.

— `QSlider`

Rysunek 2.5. Przykładowy suwak QSlider

Suwak `QSlider` daje możliwość ustawiania wielkości liczbowych za pomocą myszy.

— `QLCDNumber`



Rysunek 2.6. Przykładowy wyświetlacz QLCDNumber

Wyświetlacz `QLCDNumber` pozwala wyświetlać wartości liczbowe jak na wyświetlaczu kalkulatora w bazie dziesiętnej, szesnastkowej, ósemkowej lub binarnej.

— `QLabel`

Rysunek 2.7. Przykładowa etykieta QLabel

Etykieta tekstowa `QLabel` umożliwia wyświetlenie krótkiego tekstu, który najczęściej jest podpisem do innego widgetu.

— `QLineEdit`

Rysunek 2.8. Przykładowa linia tekstu QLineEdit

Linia tekstu `QLineEdit` służy do wprowadzania jednolinijkowej informacji tekstowej.

— `QTextEdit`

Rysunek 2.9. Przykładowe pole tekstowe QTextEdit

Pole tekstowe `QTextEdit` pozwala wprowadzać i edytować teksty składające się z wielu linii. Ma możliwość wyświetlania tekstu formatowanego (zmiana kroju pisma, wielkości itd.).

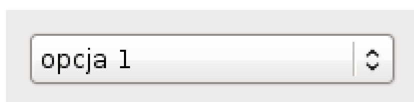
— `QDateTimeEdit`
Pole daty i czasu `QDateTimeEdit` umożliwia wprowadzanie daty i czasu. Istnieją też oddzielne klasy do wprowadzania daty: `QDateEdit` oraz czasu: `QTimeEdit`.

Rysunek 2.10. Przykładowe pole daty i czasu QDateTimeEdit

Rysunek 2.11. Przykładowy widok kalendarza QCalendarWidget

— `QCalendarWidget`
  Widget kalendarza `QCalendarWidget` pozwala w wygodny sposób wprowadzić datę.

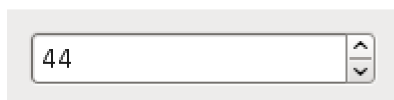
— `QComboBox`



Rysunek 2.12. Przykładowa lista rozwijana QComboBox

Lista rozwijana `QComboBox` służy do wybierania jednej opcji, wartości spośród kilku możliwych.

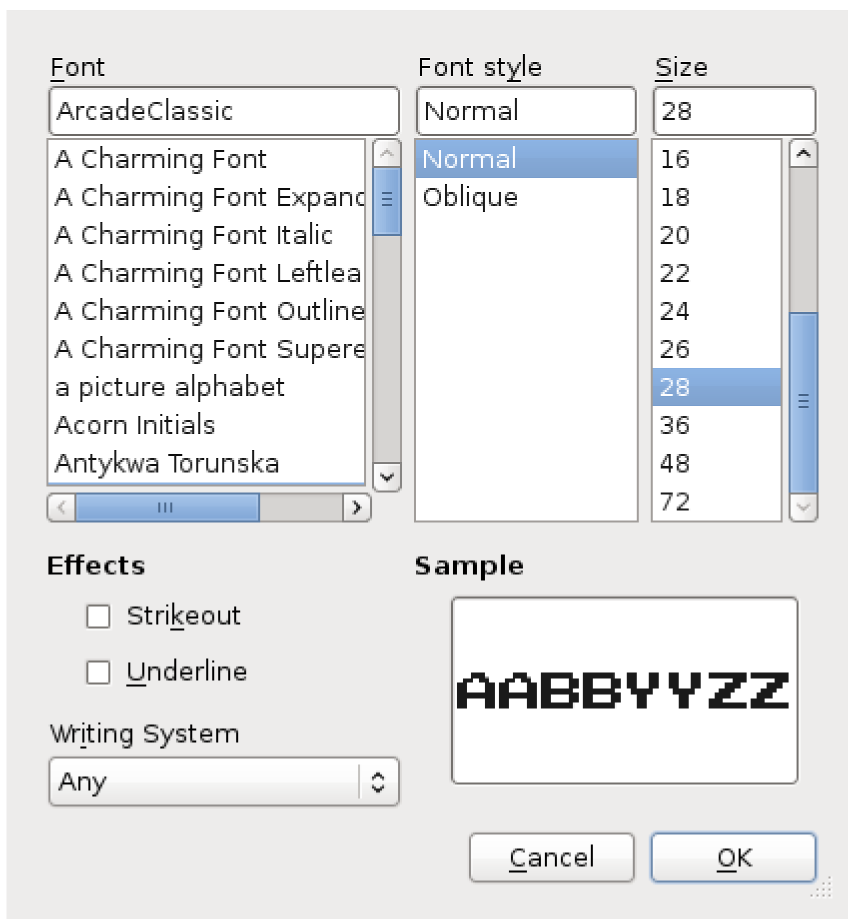
— `QSpinBox`



Rysunek 2.13. Przykładowe pole QSpinBox

Pole `QSpinBox` umożliwia wprowadzanie wartości całkowitych. Odpowiednikiem dla wartości zmiennoprzecinkowych jest widget klasy `QDoubleSpinBox`.

Oprócz gotowych widgetów dostępne są również gotowe, często stosowane typowe okna dialogowe. Przykładem może być okno wyboru kroju pisma `QFontDialog`.

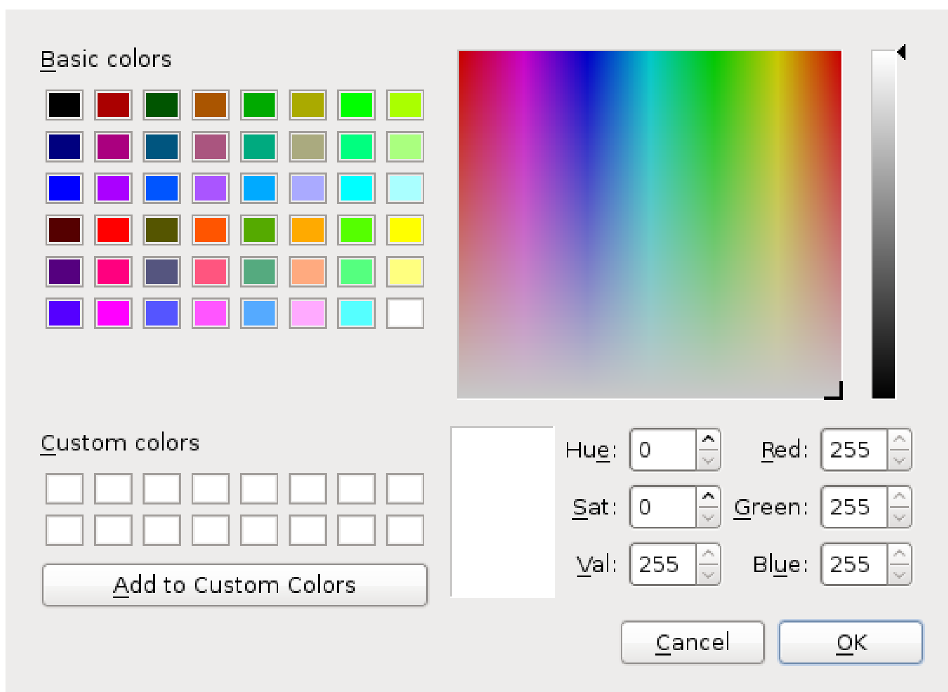


Rysunek 2.14. Przykładowe okno wyboru kroju pisma QFontDialog

W aplikacjach graficznych może być przydatne okno wyboru koloru `QColorDialog`.

Przydatne w zdecydowanej większości programów jest okno wyboru pliku — realizuje je klasa `QFileDialog`.

Oprócz tego dostępne są okna dialogowe takie jak:
- `QMessageBox` służące do wyświetlania komunikatów,



Rysunek 2.15. Przykładowe okno wyboru koloru QColorDialog

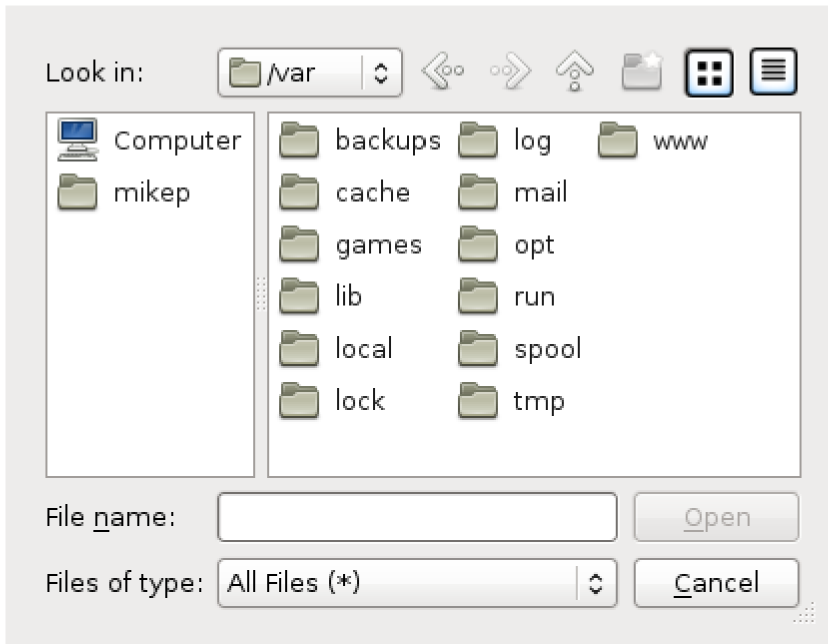
— `QPrintDialog` umożliwiające ustawienie parametrów wydruku,
— `QProgressDialog` wyświetlające poziom zaawansowania dowolnej czynności wykonywanej przez program.

2.2.2. Rozmieszczanie widgetów

Najczęściej jeden widget to zdecydowanie za mało, aby stworzyć funkcjonalną aplikację. Dlatego zajmiemy się komponowaniem widgetów w większe całości. Odbywać się to będzie poprzez „wkładanie" widgetów składowych — *dzieci* — do widgetów *rodziców*. Widget może mieć wiele widgetów-dzieci, ale tylko jeden widget-rodzica. Niedługo się przekonamy, że nawet okno programu jest widgetem, do którego zostały dodane widgety potomne. Taka hierarchia może mieć wiele poziomów, tzn. widget, który ma widgety potomne, sam może być potomkiem innego widgetu itd.

Jeśli spojrzymy na deklarację konstruktora klasy `QWidget`:

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



Rysunek 2.16. Przykładowe okno wyboru pliku QFileDialog

zobaczymy, że przyjmuje on parametr `parent`, który domyślnie przyjmuje wartość 0 (`NULL`). Standardowo w ten właśnie sposób, tworząc nowy widget, określamy rodzica — przekazujemy jego wskaźnik. Jednak jeśli nie zrobiliśmy tego tworząc widget, możemy to zrobić już w czasie jego życia, na przykład za pomocą metody `setParent(QWidget *parent)`. Widget, który nie ma rodzica, wyświetlany jest jako oddzielne okno. Wniosek z tego jest taki, że jeśli chcemy utworzyć aplikację z wieloma oknami (jak ma to miejsce np. w programie *gimp*), wystarczy zaznaczyć odpowiednie widgety jako nieposiadające rodziców.

Rozmieszczenie widgetów tak, aby automatycznie dostosowywały swój rozmiar do rozmiarów okna (ogólniej: widgetu rodzica) — co jest pożądane — uzyskamy używając layoutów. Są to obiekty klas dziedziczących po `QLayout`. Layout jest pojemnikiem, do którego wkładamy widgety. On zaś dostosowuje rozmiar swoich składowych do swojego rozmiaru. Dodatkowo layouty można zagnieżdżać: obok innych widgetów umieszczonych wewnątrz layoutu może się znaleźć kolejny layout, w którym znowu umieszczane będą widgety i layouty. Można tu zaobserwować analogię z widgetami, które również mogą być umieszczane jeden w drugim. Praktyczny przykład użycia zagnieżdżania widgetów i layoutów możemy zaobserwować w programie

widgety i layouty zaprezentowanym na rysunku 2.19, którego kod źródłowy prezentujemy w listingu 2.3.

Klasa `QLayout` jest abstrakcyjna — służy tylko jako baza dla konkretnych klas dziedziczących, z których każda charakteryzuje się specyficznym sposobem rozmieszczania elementów składowych. Mamy więc następujące klasy dziedziczące po `QLayout`:

- `QBoxLayout`, z której z kolei dziedziczą klasy
  - `QVBoxLayout`
  - `QHBoxLayout`
- `QGridLayout`
- `QFormLayout`
- `QStackedLayout`

Obiekt klasy **`QBoxLayout`** rozmieszcza swoje składowe jeden za drugim w szeregu, przy czym należy podać kierunek i zwrot dodawania składowych:

```
QBoxLayout::QBoxLayout(
      Direction dir,
      QWidget *parent = 0
    );
```

Jako `dir` możemy podać `QBoxLayout::LeftToRight`, `QBoxLayout::RightToLeft`, `QBoxLayout::TopToBottom`, `QBoxLayout::BottomToTop`, aby otrzymać orientację layoutu zgodną z argumentem.

Parametr `parent` określa widget, w którym tworzony właśnie layout ma zarządzać elementami składowymi.

Klasy **`QVBoxLayout`** oraz **`QHBoxLayout`** dziedziczące po `QBoxLayout` zostały stworzone jako uproszczenie w tworzeniu layoutów, w których elementy są umieszczane po kolei. W layoucie klasy `QVBoxLayout` widgety rozmieszczane są z góry na dół, natomiast w layoucie klasy `QHBoxLayout` — z lewej do prawej.

Aby dodać widget do layoutu klasy `QBoxLayout` lub potomnej, używamy metody

```
void QBoxLayout::addWidget (
        QWidget * widget,
        int stretch = 0,
        Qt::Alignment alignment = 0
     )
```

Parametr `stretch` określa rozmiar widgetu w kierunku zgodnym z layoutem, czyli np. dla layoutu klasy `QHBoxLayout` w poziomie. Rozmiar dodanego widgetu jest na bieżąco dostosowywany do rozmiaru całego layoutu, w ten sposób, że przy zwiększaniu layoutu rozmiar widgetu rośnie tym bardziej, im większa jest wartość parametru `stretch`. Natomiast jeśli jako wartość tego parametru podamy 0 i wszystkie inne elementy dodane do layoutu będą

również miały parametr `stretch` równy 0, to elementy będą zmieniać swój rozmiar zgodnie z własnością `QWidget:sizePolicy()`.

Parametr `alignment` określa względne ułożenie widgetu wewnątrz konkretnej komórki przydzielonej dla niego w layoucie. Domyślna wartość 0 oznacza wypełnienie jej w całości przez widget. Inne możliwe przykładowe wartości to np. `Qt::AlignLeft`, `Qt::AlignVCenter`, które oznaczają odpowiednio ułożenie po lewej i pośrodku na osi pionowej.

Opisane poniżej dwie dodatkowe metody dają nam jeszcze większą kontrolę nad rozmieszczeniem elementów w layoucie. Metoda

```
void QBoxLayout::addSpacing (int size)
```

wstawia puste miejsce do layoutu o rozmiarze `size` pikseli. Jeśli chcielibyśmy, by rozmiar przerwy dostosowywał się tak, jak rozmiary widgetów, możemy użyć metody

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

w której parametr `stretch` ma dokładnie takie znaczenie, jak w metodzie `QBoxLayout::addWidget`.

Layouty klasy **`QGridLayout`** umożliwiają umieszczanie elementów w tabeli. Spójrzmy na przykład podany na rysunku 2.17. Wszystkie widgety zostały umieszczone w wierszach i kolumnach, które numerowane są od 0, począwszy od lewego górnego rogu. Zostały do tego użyte dwie metody:

```
void  addWidget(
          QWidget * widget,
          int row,
          int column,
          Qt::Alignment alignment = 0
      )
```

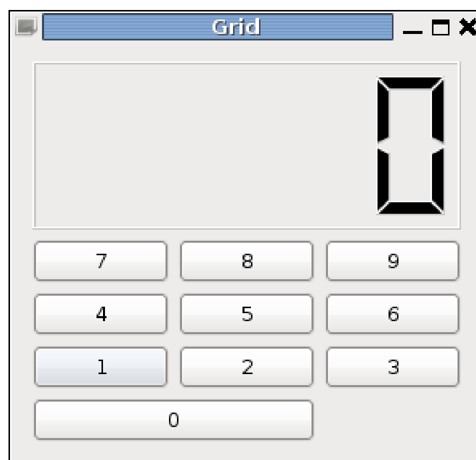
oraz

```
void  addWidget(
          QWidget * widget,
          int fromRow,
          int fromColumn,
          int rowSpan,
          int columnSpan,
          Qt::Alignment alignment = 0
      )
```

Przykład użycia pierwszej metody mamy w linii 16 listingu 2.1. W pętli umieszczane są przyciski klawiszy od 1 do 9. Ich położenie obliczane jest na

podstawie wartości umieszczonej na przycisku. Użyta metoda powoduje, że widgety zajmują pojedynczą komórkę tabeli.

Podobnie działa druga metoda, ale dodatkowo możemy określić, że widget ma zajmować kilka wierszy lub kolumn. Na przykład w linii 11 umieszczany jest widget `QLCDNumber` tak, aby zajmował 3 kolumny. Natomiast w 19 linii umieszczany w layoucie jest przycisk „0”, który zajmuje 2 kolumny.



Rysunek 2.17. Użycie layoutu `QGridLayout` — okno programu

Listing 2.1. Użycie layoutu `QGridLayout` — kod źródłowy

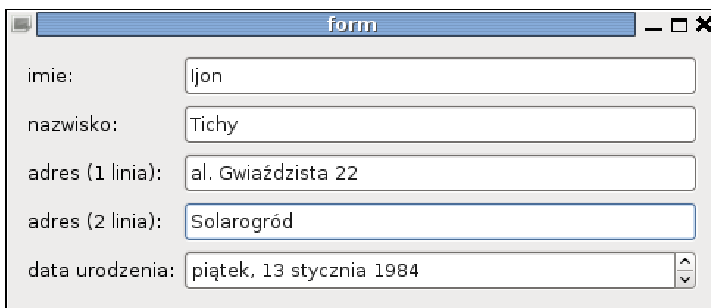
```
 1 #include <QApplication>
 2 #include <QtGui>
 3
 4 int main(int argc, char* argv[]) {
 5   QApplication app(argc, argv);
 6   QWidget *mainWidget = new QWidget();
 7
 8   QGridLayout *layout = new QGridLayout(mainWidget);
 9
10   QLCDNumber *lcd = new QLCDNumber();
11   layout->addWidget( lcd, 0, 0, 1, 3);
12
13   QPushButton *pbt[10];
14   for(int ii=1; ii<10; ++ii){
15     pbt[ii] = new QPushButton( QString("%1").arg(ii) );
16     layout->addWidget( pbt[ii], 3-(ii-1)/3, (ii-1)%3);
17   }
18   pbt[0] = new QPushButton("0");
19   layout->addWidget( pbt[0], 4, 0, 1, 2);
20
21   mainWidget->show();
```

```
22   return app.exec();
23 }
```

W tym miejscu Czytelnikowi należy się pewne wyjaśnienie. Wspomnieliśmy wcześniej, że jeśli widget nie ma określonego rodzica, jest wyświetlany jako oddzielne okno. Dlaczego zatem, skoro tworzymy widget `QLCDNumber` w linii 10 i nie podajemy konstruktorowi wskaźnika na rodzica, widget ten nie wyświetla się jako oddzielne okienko? Dzieje się tak, ponieważ w następnej linijce obiekt `lcd` zostaje dodany do layoutu. Layout zajmuje się już dodanym widgetem i ustawia mu odpowiedniego rodzica — tego, w którym sam jest umieszczony.

Przy okazji zauważmy, że wszystkie widgety, które utworzyliśmy, stały się widoczne na ekranie, mimo że metoda `show()` została tylko raz wywołana na rzecz głównego widgetu. Dzieje się tak dzięki temu, że w momencie zmiany widoczności widgetu, zmienia on również widoczność wszystkich swoich widgetów potomnych.

Layout klasy **`QFormLayout`** został zaprojektowany specjalnie do tworzenia formularzy, w których występuje ciąg etykiet po lewej i widgetów wejściowych po prawej. Niech za przykład posłuży prosty program poniżej.



Rysunek 2.18. Przykład użycia `QFormLayout` — okno programu

Listing 2.2. Przykład użycia `QFormLayout` — kod źródłowy

```
1  #include <QApplication>
2  #include <QtGui>
3
4  int main(int argc, char* argv[]) {
5    QApplication app(argc, argv);
6    QWidget *mainWidget = new QWidget();
7
8    QFormLayout *layout = new QFormLayout(mainWidget);
9
10   QLineEdit *fnameEdit = new QLineEdit();
```

```
11   QLineEdit *snameEdit = new QLineEdit();
12   QLineEdit *address1Edit = new QLineEdit();
13   QLineEdit *address2Edit = new QLineEdit();
14   QDateEdit *dobEdit = new QDateEdit();
15   dobEdit->setDisplayFormat(
16                   QObject::tr("dddd, d MMMM yyyy"));
17
18   layout->addRow( QObject::tr("imie:"),
19                   fnameEdit);
20   layout->addRow( QObject::tr("nazwisko:"),
21                   snameEdit);
22   layout->addRow( QObject::tr("adres (1 linia):"),
23                   address1Edit);
24   layout->addRow( QObject::tr("adres (2 linia):"),
25                   address2Edit);
26   layout->addRow( QObject::tr("data urodzenia:"),
27                   dobEdit);
28
29   mainWidget->show();
30   return app.exec();
31 }
```

Ostatni z wymienionych layoutów — `QStackedLayout` — umieszcza wszystkie widgety w swojej wewnętrzej liście, ale wyświetla tylko jeden z nich w danym czasie. Który widget ma być wyświetlany, można zmienić metodą

```
void  setCurrentIndex(int index),
```

gdzie parametr `index` oznacza numer kolejny widgetu, który ma się stać widoczny lub metodą

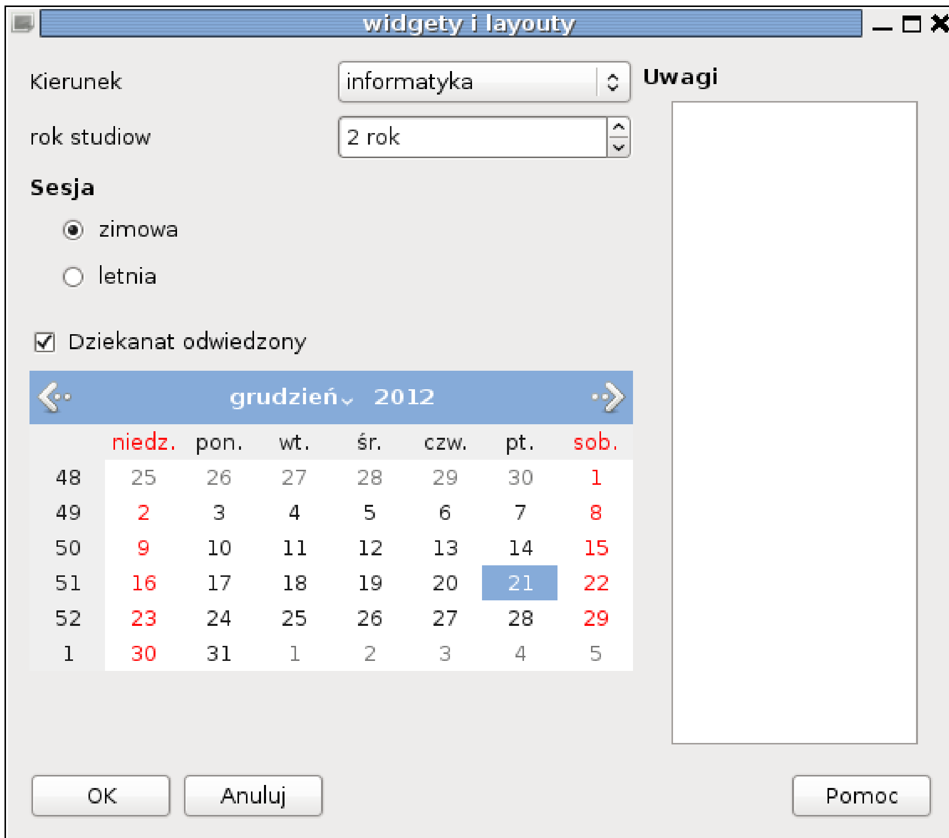
```
void  setCurrentWidget(QWidget *widget),
```

której wprost podajemy wskaźnik na widget mający się uwidocznić.

Zobaczmy przykładowe okno programu (rys. 2.19) z rozmieszczonymi za pomocą layoutów różnymi widgetami. Wykorzystaliśmy w nim poznane do tej pory widgety.

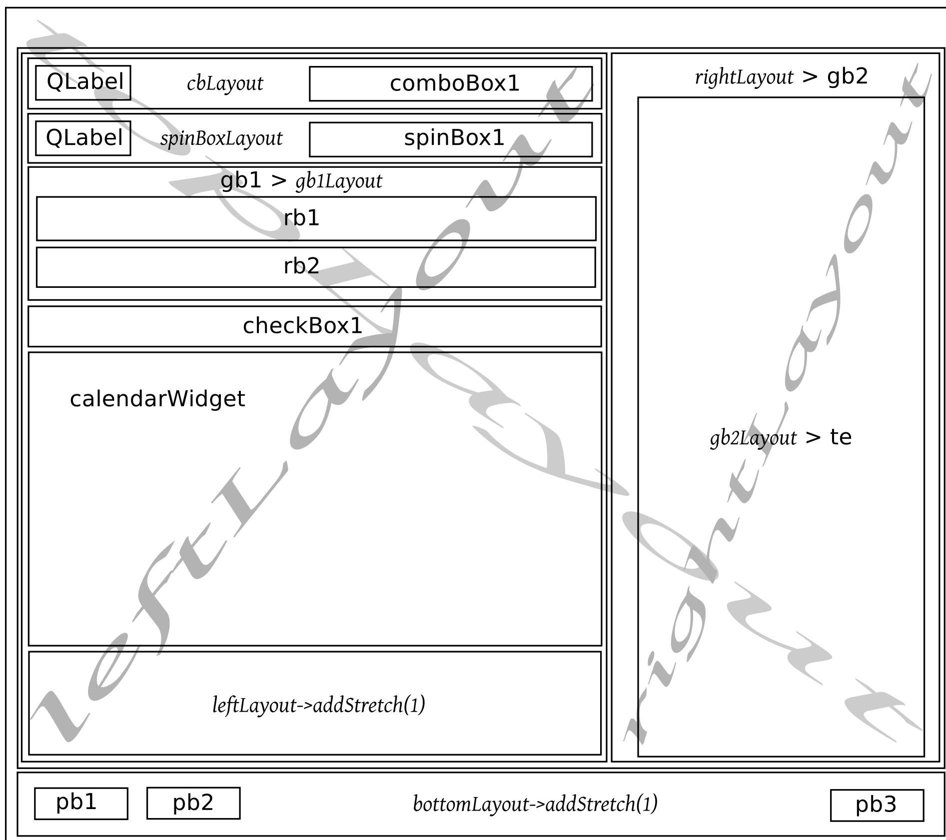
Listing 2.3. Przykładowe użycie widgetów

```
1 #include <QApplication>
2 #include <QtGui>
3
4 int main(int argc, char* argv[])
5 {
6   QApplication app(argc, argv);
7   QWidget *mainWidget = new QWidget();
8   mainWidget->setWindowTitle(
9         QObject::tr("widgety i layouty"));
```



Rysunek 2.19. Przykładowe okno programu z wieloma widgetami

```
10
11  QBoxLayout *mainLayout = new QVBoxLayout(mainWidget);
12  QBoxLayout *topLayout = new QHBoxLayout();
13  mainLayout->addLayout(topLayout);
14  QBoxLayout *bottomLayout = new QHBoxLayout();
15  mainLayout->addLayout(bottomLayout);
16  QBoxLayout *leftLayout = new QVBoxLayout();
17  topLayout->addLayout(leftLayout);
18  QBoxLayout *rightLayout = new QVBoxLayout();
19  topLayout->addLayout(rightLayout, 1);
20
21  QComboBox *comboBox1 = new QComboBox();
22  comboBox1->addItem("fizyka");
23  comboBox1->addItem("informatyka");
24  comboBox1->addItem("matematyka");
25  QBoxLayout *cbLayout = new QHBoxLayout();
26  cbLayout->addWidget(new QLabel(
27        QObject::tr("Kierunek")));
```



Rysunek 2.20. Schemat rozmieszczenia widgetów i layoutów w powyższym programie

```
28    cbLayout->addWidget(comboBox1);
29    leftLayout->addLayout(cbLayout);
30
31    QSpinBox *spinBox1 = new QSpinBox();
32    spinBox1->setRange(1, 5);
33    spinBox1->setSuffix(QObject::tr(" rok"));
34    QBoxLayout *spinBox1Layout = new QHBoxLayout();
35    spinBox1Layout->addWidget(new QLabel(
36          QObject::tr("rok studiow")));
37    spinBox1Layout->addWidget(spinBox1);
38    leftLayout->addLayout(spinBox1Layout);
39
40    QGroupBox *gb1 = new QGroupBox(QObject::tr("Sesja"));
41    leftLayout->addWidget(gb1);
42    QVBoxLayout *gb1Layout = new QVBoxLayout(gb1);
43    QRadioButton *rb1 = new QRadioButton(
44          QObject::tr("zimowa"));
```

```
45      gb1Layout->addWidget(rb1);
46      QRadioButton *rb2 = new QRadioButton(
47            QObject::tr("letnia"));
48      gb1Layout->addWidget(rb2);
49
50      QCheckBox *checkBox1 = new QCheckBox(
51            QObject::tr("Dziekanat odwiedzony"));
52      leftLayout->addWidget(checkBox1);
53
54      QCalendarWidget *calendarWidget = new QCalendarWidget();
55      leftLayout->addWidget(calendarWidget);
56      leftLayout->addStretch(1);
57
58      QGroupBox *gb2 = new QGroupBox(QObject::tr("Uwagi"));
59      rightLayout->addWidget(gb2);
60      QBoxLayout *gb2Layout = new QVBoxLayout(gb2);
61      QTextEdit *te = new QTextEdit();
62      gb2Layout->addWidget(te);
63
64      QPushButton *pb1 = new QPushButton(QObject::tr("OK"));
65      QPushButton *pb2 = new QPushButton(
66           QObject::tr("Anuluj"));
67      bottomLayout->addWidget(pb1);
68      bottomLayout->addWidget(pb2);
69      bottomLayout->addStretch(1);
70      QPushButton *pb3 = new QPushButton(
71           QObject::tr("Pomoc"));
72      bottomLayout->addWidget(pb3);
73
74      mainWidget->show();
75
76      return app.exec();
77  }
```

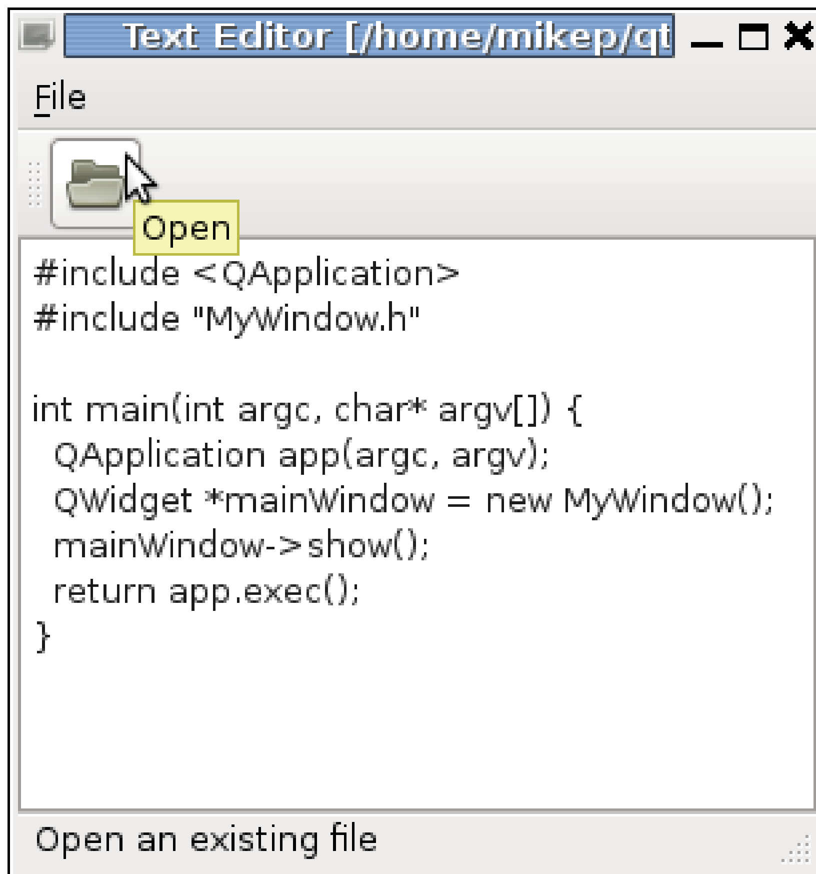
2.3. Okno główne, menu, pasek narzędzi i pasek statusu

Przedstawiony teraz zostanie sposób, w jaki można wykorzystać klasę `QMainWindow` do tworzenia głównego okna aplikacji. Utworzone zostanie menu, pasek narzędzi (toolbar) z ikoną i pasek statusu (statusbar).

Przykładem będzie prosty edytor tekstu. W celu ograniczenia i uproszczenia kodu źródłowego funkcje programu zostały ograniczone do jedynie wczytywania i edycji bez możliwości zapisywania zmian.

Poniżej zamieszczony został plik nagłówkowy zawierający deklarację klasy `MyWindow` będącej głównym widgetem aplikacji.

Listing 2.4. Plik nagłówkowy klasy `MyWindow`



Rysunek 2.21. Przykładowe okno programu wykorzystującego `QMainWindow` — prosty edytor tekstu

```
1  #ifndef MYWINDOW__H
2  #define MYWINDOW__H
3
4  #include <QMainWindow>
5  #include <QApplication>
6  #include <QtGui>
7
8  class MyWindow : public QMainWindow {
9      Q_OBJECT
10   public:
11     MyWindow(QWidget * parent = 0);
12   protected:
13     QTextEdit * textEdit;
14     QAction * openFileAction;
15     QMenu * fileMenu;
16     QToolBar * fileToolBar;
```

```
17     QString fileName;
18
19   private slots:
20     void openFile();
21   private:
22     void setTextEdit();
23     void createStatusBar();
24     void createActions();
25     void createMenu();
26     void createToolBar();
27     void setTitle();
28 };
29
30 #endif  //  MYWINDOW__H
```

Klasa `MyWindow` dziedziczy po `QMainWindow`. Zawiera utworzony przez nas slot `openFile()`, zatem wymagana jest obecność makra `Q_OBJECT`. Inicjacja całego okna następuje w konstruktorze, który wywołuje pomocnicze metody tworzące m.in. statusbar, menu itd.

Poniżej zamieszczony został kod źródłowy implementujący klasę `MyWindow`.

Listing 2.5. Implementacja klasy `MyWindow`

```
1 #include "MyWindow.h"
2
3 MyWindow::MyWindow(QWidget * parent) : QMainWindow(parent) {
4   setTextEdit();
5   createStatusBar();
6   createActions();
7   createMenu();
8   createToolBar();
9
10  setTitle();
11 }
12
13 void MyWindow::setTextEdit() {
14   textEdit = new QTextEdit();
15   setCentralWidget(textEdit);
16 }
17
18 void MyWindow::createStatusBar() {
19   statusBar()->showMessage(tr("Application ready"), 5000);
20 }
21
22 void MyWindow::createActions() {
23   openFileAction = new QAction(
24       QApplication::style()->standardIcon(QStyle::SP_DirOpenIcon),
25       tr("&Open..."),
26       this
```

```
27      );
28    openFileAction->setStatusTip(tr("Open an existing file"));
29    connect( openFileAction, SIGNAL(triggered()),
30             this, SLOT(openFile()) );
31  }
32
33  void MyWindow::createMenu() {
34    fileMenu = menuBar()->addMenu(tr("&File"));
35    fileMenu->addAction(openFileAction);
36  }
37
38  void MyWindow::createToolBar() {
39    fileToolBar = addToolBar(tr("&File"));
40    fileToolBar->addAction(openFileAction);
41  }
42
43  void MyWindow::setTitle() {
44    setWindowTitle( tr("Text Editor [%1]").arg(fileName) );
45  }
46
47  void MyWindow::openFile() {
48    fileName = QFileDialog::getOpenFileName(this,
49                 tr("Open Text File"), ".",
50                 tr("Text Files (*.txt *.cpp *.c *.h)") );
51    QFile file(fileName);
52    if( file.open(QFile::ReadOnly | QFile::Text) ){
53      QTextStream textStream( &file );
54      textEdit->setPlainText( textStream.readAll() );
55      setTitle();
56    } else {
57      statusBar()->showMessage(tr("Error in opening the file"));
58    }
59  }
```

Jak już wspomniano — inicjacja następuje w konstruktorze.

Zwróćmy uwagę na metodę `setCentralWidget()` klasy `QMainWindow` (linia 15). Ustala ona główny widget okna — w niniejszym przykładzie jest nim pole tekstowe.

W linii 19 następuje utworzenie elementu statusbar wraz z ustawieniem w nim komunikatu „Application ready" na 5 sekund.

W 23 linii tworzony jest obiekt klasy `QAction`, który później zostanie dodany do menu oraz paska narzędzi. Została mu ustawiona standardowa ikona otwierania katalogu właściwa dla aktualnego stylu wyglądu okna. W momencie, gdy kursor myszy znajduje się nad obszarem związanym z tą akcją, w pasku statusu wyświetlana jest podpowiedź „Open an existing file" (linia 28). Uaktywnienie akcji (czyli wyemitowanie z niej sygnału `triggered()`) spowoduje wywołanie slotu `openFile()` — mechanizm ten jest ustawiany

w liniach 29-30. Więcej o mechanizmie slotów i sygnałów Czytelnik się dowie z rozdziału 3.

W liniach 33-36 tworzony jest element menu i dodana do niego utworzona uprzednio akcja związana z otwieranim pliku. Czynność analogiczna, ale dla paska narzędzi, wykonywana jest w liniach 38-41.

Metoda `setTitle()` ustawia tytuł okna umieszczając w nim nazwę otwartego pliku w nawiasach kwadratowych (linia 44).

Slot `openFile()` tworzy okno dialogowe wyboru pliku do otwarcia (linie 48-50). Następnie próbuje otworzyć plik i jeśli czynność ta się powiedzie, wczytuje jego zawartość do pola edycyjnego `textEdit` oraz aktualizuje tytuł okna. W przeciwnym razie na pasku statusu wyświetlany jest komunikat o niepowodzeniu.

Rozdział 3

Sloty, sygnały, zdarzenia

3.1. Wstęp **38**
3.2. Sloty i sygnały **38**
3.3. Zdarzenia w Qt **47**

3.1. Wstęp

Sygnały i sloty zapewniają komunikację między obiektami. Przykładowo, w graficznym interfejsie użytkownika, zmiana stanu danego obiektu (np. kliknięcie przycisku przez użytkownika) może wymagać poinformowania innych obiektów i wykonania przez nie określonych funkcji (np. zamknięcia okna).

Sloty i sygnały są jednym z najważniejszych mechanizmów środowiska Qt, a jednocześnie jego cechą charakterystyczną. W konkurencyjnych środowiskach, zadanie komunikacji miedzy obiektami jest realizowane inaczej.

3.2. Sloty i sygnały

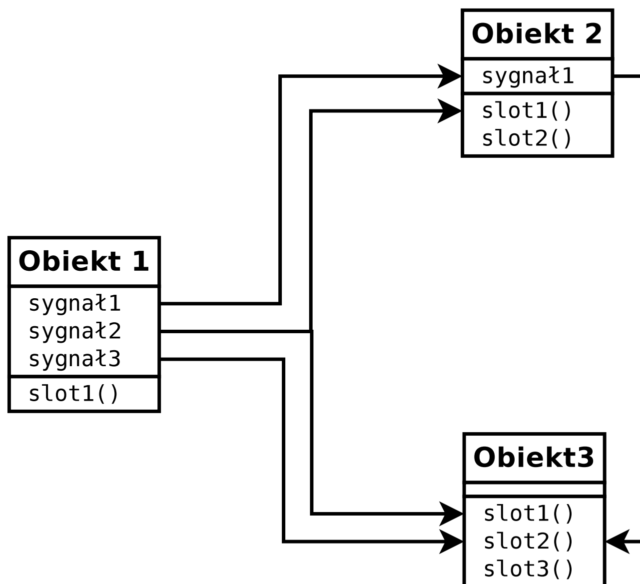
Sygnał jest emitowany w momencie wystąpienia określonego zdarzenia (np. wspomnianego już kliknięcia przycisku). Obiekty Qt emitują sygnały zdefiniowane przez ich twórców. Informacje o sygnałach tych można znaleźć w dokumentacji [1]. Tworząc własne widgety (a ściślej mówiąc, klasy dziedziczące po QObject [1]), można definiować również własne sygnały. Slot jest z kolei funkcją składową obiektu, która jest wywoływana w reakcji na określony sygnał. Funkcje te można oczywiście nadal wywoływać w standardowy dla języka C++ sposób. Analogicznie jak w przypadku sygnałów, tworząc własne widgety, można definiować własne sloty lub redefiniować istniejące.

Aby mechanizm faktycznie zadziałał, należy dokonać odpowiedniego skojarzenia (rys. 3.1). Dany sygnał może być połączony z wieloma slotami, z kolei z tym samym slotem może być skojarzonych wiele różnych sygnałów. Można także połączyć sygnał z kolejnym sygnałem. Wyemitowanie pierwszego sygnału spowoduje wówczas natychmiastową emisję drugiego. Dozwolone jest kojarzenie sygnałów i slotów tego samego obiektu, jak również różnych obiektów, różnych klas.

Obiekt, emitując sygnał, nie otrzymuje żadnej informacji zwrotnej o odbiorcach, ani o tym, czy w ogóle został on odebrany. Slot z kolei nie dysponuje informacją, czy są do niego dołączone jakiekolwiek sygnały.

Sygnały i sloty mogą posiadać argumenty dowolnego typu, np. sygnał informujący o zmianie rozmiaru okna mógłby jednocześnie podawać jego nowe rozmiary, które zostaną użyte jako argumenty wywołania slotu (funkcji). Zgodność typów argumentów musi być zachowana i jest weryfikowana przez kompilator. Slot może posiadać mniej argumentów wywołania niż skojarzony z nim sygnał i nadmiarowe argumenty są wówczas ignorowane.

Listing 3.1 przedstawia możliwie najprostszy przykład zastosowania slotów i sygnałów.



Rysunek 3.1. Przykład połączeń slotów i sygnałów

Rysunek 3.2. Wynik działania programu z listingu 3.1

Listing 3.1. Najprostszy przykład użycia slotów i sygnałów

```
1 #include <QtGui/QApplication>
2 #include <QtGui/QMainWindow>
3 #include <QtGui/QPushButton>
4
5 int main(int argc, char *argv[])
6 {
7     QApplication a(argc, argv);
8
9     QMainWindow w;
10    w.show();
11
12    QPushButton quitButton("Quit", &w);
13    quitButton.show();
14
```

```
15     QObject::connect(&quitButton, SIGNAL(clicked()),
16                                  &w, SLOT(close()));
17
18     return a.exec();
19 }
```

Okno aplikacji zawiera tylko jeden przycisk “Quit” (rys. 3.2) – obiekt klasy `QPushButton`. Zgodnie z dokumentacją [1], klasa `QPushButton` posiada wśród kilku zdefiniowanych w niej sygnałów:

```
void clicked ( bool checked = false )
```

odziedziczony po `QAbstractButton`. Jest on emitowany w momencie kliknięcia przycisku. Z kolei w przypadku `QMainWindow`, dysponujemy publiczną funkcją składową:

```
bool QWidget::close ()
```

odziedziczoną po `QWidget`. Funkcja ta jest jednocześnie slotem, a jej wywołanie skutkuje zamknięciem okna (widgetu). W naszym przypadku jest to główne okno aplikacji, więc zostanie ona zakończona.

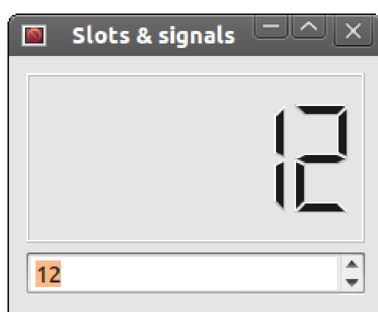
Najbardziej zagadkowym fragmentem kodu jest wywołanie funkcji `connect()`. Służy ona do powiązania sygnału ze slotem. Jej argumentami są kolejno:
— wskaźnik do obiektu emitującego sygnał,
— sygnał,
— wskaźnik do obiektu, którego slot ma zostać połączony z sygnałem,
— slot.
Do specyfikacji sygnałów i slotów konieczne jest użycie makr `SIGNAL()` i `SLOT()`, odpowiednio. Kliknięcie przycisku powinno teraz skutkować zakończeniem programu.

Kolejny przykład zilustruje sytuację, w której konieczne jest skorzystanie z argumentów slotów i sygnałów.

Listing 3.2. Sygnały i sloty z argumentami

```
1 #include <QtGui>
2
3 int main(int argc, char *argv[])
4 {
5     QApplication a(argc, argv);
6     QWidget w;
7
8     QLCDNumber* lcddisplay = new QLCDNumber();
9     QSpinBox* spinbox = new QSpinBox();
10    QVBoxLayout* layout = new QVBoxLayout();
```

```
11
12      layout->addWidget(lcddisplay);
13      layout->addWidget(spinbox);
14
15      QObject::connect(spinbox, SIGNAL(valueChanged(int)),
16                       lcddisplay, SLOT(display(int)));
17
18      w.setLayout(layout);
19      w.show();
20
21      return a.exec();
22  }
```



Rysunek 3.3. Wynik działania programu z powyższego listingu

Rys. 3.3 przedstawia okno uruchomionego programu. Zgodnie z naszymi zamierzeniami, zmiana wartości w polu `spinbox` skutkuje natychmiastową aktualizacją wartości wyświetlanej na LCD. W 15. wierszu powyższego listingu, połączono sygnał `valueChanged` obiektu klasy `QSpinBox`, ze slotem display obiektu `QLCDNumber`. `QLCDNumber` posiada kilka przeciążonych slotów `display`:

```
void  display ( const QString & s )
void  display ( double num )
void  display ( int num )
```

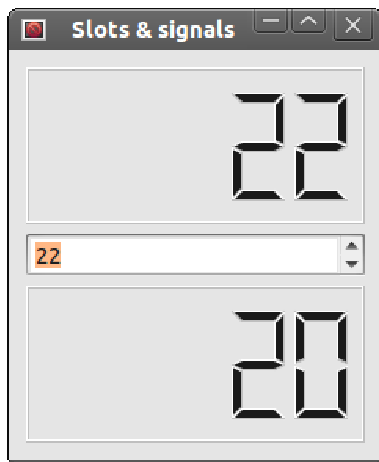
W funkcji `connect` należało użyć sygnału z typem argumentu zgodnym z argumentem sygnału:

```
void valueChanged ( int i )
```

W analogiczny sposób można przekazywać wiele argumentów.

Przedstawiony dotychczas zasób wiadomości o slotach i sygnałach jest wystarczający do budowania prostych interfejsów graficznych w Qt (przy wykorzystaniu dokumentacji). W przypadku bardziej zaawansowanych apli-

kacji, zaistnieje konieczność definiowania własnych slotów i sygnałów lub zmiany sposobu działania istniejących. Tym zagadnieniom jest poświęcona dalsza część rozdziału.



Rysunek 3.4. Wynik działania programu rozbudowanego o dodatkowe elementy

Celem kolejnych modyfikacji programu z listingu 3.2 jest dodanie drugiego wyświetlacza LCD, jak na rys. 3.4. Górny wyświetlacz będzie działał identycznie jak poprzednio, natomiast wartość wskazywana przez dolny będzie zmieniana na aktualną wartość, wpisaną w polu `spinbox` wtedy i tylko wtedy, gdy jest ona wielokrotnością liczby 10.

W tym celu zostanie stworzona nowa klasa `SpinBox10`, na bazie `QSpinBox`. Chcemy, by oprócz standardowej funkcjonalności, w razie gdy wprowadzona wartość jest wielokrotnością liczby 10, emitowany był dodatkowy (zdefiniowany przez nas) sygnał:

```
void valueChanged10( int )
```

Deklarację klasy `SpinBox10` przedstawia poniższy listing:

Listing 3.3. Deklaracja klasy SpinBox10 – plik spinbox10.h

```
1 #ifndef SPINBOX10_H
2 #define SPINBOX10_H
3
4 #include <QSpinBox>
5
6 class SpinBox10 : public QSpinBox
7 {
8     Q_OBJECT
```

```
 9
10 public:
11     SpinBox10(QSpinBox *parent = 0);
12
13 signals:
14     void valueChanged10(int);
15
16 private slots:
17     void checkValue(int);
18
19 };
20
21 #endif  //   SPINBOX10_H
```

W 8. wierszu powyższego listingu charakterystyczne jest wywołanie makra `Q_OBJECT`. Musi ono znaleźć się na początku deklaracji każdej klasy, która zawiera sloty lub sygnały. Klasa taka musi również bezpośrednio lub pośrednio dziedziczyć po QObject. Deklarację nowego sygnału zawiera 14. wiersz listingu (po słowie `signals:`).

W tym momencie widać już, że kod w przedstawionej postaci nie może zostać skompilowany standardowym kompilatorem C++, ze względu na obecność niedozwolonych słów kluczowych (`signals`, `slots`). Muszą one zostać usunięte przez preprocesor i zastąpione zwykłym kodem C++. Operacje te wykonuje *Meta-Object Compiler* (*moc*). W razie generowania plików makefile przy pomocy narzędzia qmake, wywołania *moc* zostaną automatycznie wstawione tam, gdzie jest to konieczne.

Kolejny listing przedstawia definicję klasy `SpinBox10`:

Listing 3.4. Definicja klasy SpinBox10 – plik spinbox10.cpp

```
 1 #include "spinbox10.h"
 2
 3 SpinBox10::SpinBox10(QSpinBox *parent) :
 4     QSpinBox(parent) {
 5
 6     QObject::connect(this, SIGNAL(valueChanged(int)),
 7                      this, SLOT(checkValue(int)));
 8
 9 }
10
11 void SpinBox10::checkValue(int val) {
12     if ((val !=0) && (val%10 == 0))
13         emit valueChanged10(val);
14 }
```

Do wysłania sygnału służy nowe słowo kluczowe `emit`. Emisję nowego sygnału (13. wiersz powyższego listingu) musi poprzedzić sprawdzenie, czy

wprowadzona wartość jest wielokrotnością 10 (12. wiersz). Operacje te są wykonywane przez funkcję:

```
void SpinBox10::checkValue(int val)
```

zadeklarowaną jako prywatny slot. Slot ten jest wywoływany przez sygnał `valueChanged(int)` (zgodnie z zapisem w 6. i 7. wierszu). Jak widać, nie ma przeszkód, by powiązać sygnały i sloty tego samego obiektu. Można wówczas skorzystać z prostszej postaci wywołania funkcji `QObject::connect`:

```
QObject::connect(this, SIGNAL(valueChanged(int)),
                       SLOT(checkValue(int)));
```

Plik main.cpp ma obecnie postać:

Listing 3.5. Zastosowanie klasy SpinBox10 – plik main.cpp

```
1  #include <QtGui>
2  #include "spinbox10.h"
3
4
5  int main(int argc, char *argv[])
6  {
7      QApplication a(argc, argv);
8      QWidget w;
9
10     QLCDNumber* lcddisplay = new QLCDNumber();
11     SpinBox10* spinbox = new SpinBox10();
12     QLCDNumber* lcddisplay10 = new QLCDNumber();
13
14     QVBoxLayout* layout = new QVBoxLayout();
15
16     layout->addWidget(lcddisplay);
17     layout->addWidget(spinbox);
18     layout->addWidget(lcddisplay10);
19
20     QObject::connect(spinbox, SIGNAL(valueChanged(int)),
21                      lcddisplay, SLOT(display(int)));
22     QObject::connect(spinbox, SIGNAL(valueChanged10(int)),
23                   lcddisplay10, SLOT(display(int)));
24
25     w.setLayout(layout);
26     w.show();
27
28     return a.exec();
29 }
```

Do wyświetlania wartości użyto dwóch obiektów klasy `QLCDNumber` – `lcddisplay` i `lcddisplay10`. Pierwszy z nich został skojarzony z sygnałem

`valueChanged(int)` (emitowanym przy każdej zmianie wartości; 20. wiersz powyższego listingu), natomiast drugi z `valueChanged10(int)` (emitowanym tylko gdy nowa wartość jest wielokrotnością 10; 22. wiersz powyższego listingu). Listingi 3.3–3.5 stanowią kompletne rozwiązanie postawionego problemu.

Problem ten można też rozwiązać w sposób alternatywny – pozostawiając zwykłą klasę `QSpinBox` do wprowadzania danych i modyfikując klasę `QLCDNumber`[1] tak, by jej slot `display(int)` reagował tylko w przypadku, gdy przekazywana wartość jest wielokrotnością 10. Chcemy zatem, by plik main.cpp miał postać:

Listing 3.6. Plik main.cpp dla rozwiązania zakładającego modyfikację klasy QSpinBox

```
1  #include <QtGui>
2  #include "lcdnumber10.h"
3
4  int main(int argc, char *argv[])
5  {
6      QApplication a(argc, argv);
7      QWidget w;
8
9      QLCDNumber* lcddisplay = new QLCDNumber();
10     LCDNumber10* lcddisplay10 = new LCDNumber10();
11     QSpinBox* spinbox = new QSpinBox();
12
13     QVBoxLayout* layout = new QVBoxLayout();
14
15     layout->addWidget(lcddisplay);
16     layout->addWidget(spinbox);
17     layout->addWidget(lcddisplay10);
18
19     QObject::connect(spinbox, SIGNAL(valueChanged(int)),
20                      lcddisplay, SLOT(display(int)));
21     QObject::connect(spinbox, SIGNAL(valueChanged(int)),
22                    lcddisplay10, SLOT(display(int)));
23
24     w.setLayout(layout);
25     w.show();
26
27     return a.exec();
28 }
```

Na bazie klasy `QLCDNumber` definiujemy klasę `LCDNumber10`:

[1] Pod pojęciem modyfikacji klasy rozumiemy stworzenie nowej klasy na bazie istniejącej, dodając pożądaną funkcjonalność.

Listing 3.7. Deklaracja klasy LCDNumber10 – plik lcdnumber10.h

```
1 #ifndef LCDNUMBER10_H
2 #define LCDNUMBER10_H
3
4 #include <QLCDNumber>
5
6 class LCDNumber10 : public QLCDNumber
7 {
8     Q_OBJECT
9 public:
10     LCDNumber10(QWidget *parent = 0);
11
12 signals:
13
14 public slots:
15     void display(int);
16 };
17
18 #endif // LCDNUMBER10_H
```

Musimy przedefiniować slot `void display(int)` z klasy `QLCDNumber`, zgodnie z listingiem:

Listing 3.8. Definicja klasy LCDNumber10 – plik lcdnumber10.cpp

```
1 #include "lcdnumber10.h"
2
3 LCDNumber10::LCDNumber10(QWidget *parent) :
4     QLCDNumber(parent){
5 }
6
7 void LCDNumber10::display(int val){
8     if ((val != 0) && (val % 10 == 0))
9       QLCDNumber::display(val);
10 }
```

Oryginalna funkcja `display(int)` (z klasy `QLCDNumber`) jest wywoływana w 9. wierszu, jedynie gdy parametr nowej funkcji jest wielokrotnością 10.

Alternatywą dla rozwiązania przedstawionego na listingach 3.3–3.5 są zatem listingi 3.6–3.8. W obu przypadkach, pod względem funkcjonalnym, program działa identycznie. Drugie rozwiązanie jest prostsze – wymaga reimplementacji tylko jednej funkcji składowej (slotu). W pierwszym rozwiązaniu konieczne było zdefiniowane nowego sygnału i nowego slotu. Byłoby ono preferowane, gdyby sygnał informujący o zmianie wprowadzonej wartości na nową, będącą wielokrotnością 10, był wykorzystywany nie tylko przez obiekt `lcddisplay` klasy `QLCDNumber`, lecz także do innych celów.

3.3. Zdarzenia w Qt

Pod pojęciem zdarzenia (ang. *event*) rozumiemy zaistnienie pewnej sytuacji w samej aplikacji, lub w wyniku zewnętrznych działań, o której aplikacja powinna wiedzieć. Zdarzeniami są np. naciśnięcia klawiszy klawiatury lub operacje myszą. Informacja o nich powinna zostać dostarczona do odpowiednich widgetów programu.

W Qt zdarzenia są reprezentowane przez obiekty klas dziedziczących po abstrakcyjnej klasie `QEvent`. Zdarzenia mogą być odbierane i obsługiwane przez obiekty klas dziedziczących po `QObject`, w szczególności przez widgety. W chwili wystąpienia zdarzenia, tworzony jest obiekt odpowiedniej podklasy `QEvent` i przekazywany do obiektu `QObject`, poprzez wywołanie jego funkcji `event()`. Wywołuje ona następnie funkcję obsługującą odpowiedni typ zdarzeń i zwraca informację o przyjęciu lub zignorowaniu zdarzenia.

Popularne przykłady zdarzeń związanych z systemem okienkowym to `QMouseEvent` i `QKeyEvent`. Są one generowane w przypadku działań myszy i klawiatury, w odniesieniu do danego widgetu. Inne zdarzenia mogą być związane z pracą samej aplikacji, np. `QTimerEvent` umożliwia generowanie zdarzeń w określonych przedziałach czasowych. `QPaintEvent` jest wysyłany do widgetu, gdy zachodzi konieczność aktualizacji jego zawartości (np. gdy został odsłonięty jego fragment, dotychczas przykryty innym widgetem). `QResizeEvent` jest wysyłany do widgetu, gdy zmieniany jest jego rozmiar i dostarcza mu informacji na ten temat.

Nazwy funkcji odpowiedzialnych za obsługę zdarzeń można znaleźć w dokumentacji poszczególnych klas, w sekcji *Events* [1]. Przykładowo, klasa `QWidget` zawiera między innymi funkcje:
— void QWidget::paintEvent (QPaintEvent * event)
— void QWidget::resizeEvent (QResizeEvent * event)
— void QWidget::mousePressEvent (QMouseEvent * event)
— void QWidget::mouseReleaseEvent (QMouseEvent * event)
— void QWidget::mouseDoubleClickEvent (QMouseEvent * event)
W przypadku prostych, typowych aplikacji, ich działanie powinno być wystarczające. W razie potrzeby można je jednak przedefiniować w klasach potomnych.

Kolejne listingi (3.9–3.10) przedstawiają przykład obsługi zdarzeń myszy. Celem jest stworzenie widgetu wyświetlającego informacje o współrzędnych umieszczonego nad nim kursora myszy oraz o naciśnięciu przycisków myszy (rys. 3.5).

Listing 3.9. Przykład obsługi zdarzeń myszy – deklaracja klasy MyWidget w pliku mywidget.h

```
1 #ifndef MYWIDGET_H
```

```
2  #define MYWIDGET_H
3
4  #include <QWidget>
5  #include <QLCDNumber>
6  #include <QVBoxLayout>
7  #include <QMouseEvent>
8  #include <QLabel>
9
10 class MyWidget : public QWidget
11 {
12     Q_OBJECT
13 public:
14     MyWidget(QWidget *parent = 0);
15
16 private:
17     QLCDNumber* lcddisplayX;
18     QLCDNumber* lcddisplayY;
19     QLabel* label;
20     QVBoxLayout* layout;
21
22 protected:
23     void mousePressEvent(QMouseEvent * event);
24     void mouseMoveEvent(QMouseEvent * event);
25
26 signals:
27
28 public slots:
29
30 };
31
32 #endif // MYWIDGET_H
```

Listing 3.10. Przykład obsługi zdarzeń myszy – definicja klasy MyWidget w pliku mywidget.cpp

```
1  #include "mywidget.h"
2
3  MyWidget::MyWidget(QWidget *parent) :
4      QWidget(parent)
5  {
6      lcddisplayX = new QLCDNumber();
7      lcddisplayY = new QLCDNumber();
8      label=new QLabel();
9      layout = new QVBoxLayout();
10
11     layout->addWidget(lcddisplayX);
12     layout->addWidget(lcddisplayY);
13     layout->addWidget(label);
14
15     setLayout(layout);
```

```
16 }
17
18 void MyWidget::mousePressEvent(QMouseEvent *event)
19 {
20     if (event->button() == Qt::LeftButton) {
21          label->setText("left");
22     }
23     else if (event->button() == Qt::RightButton) {
24          label->setText("right");
25     }
26     else {
27          QWidget::mousePressEvent(event);
28     }
29 }
30
31 void MyWidget::mouseMoveEvent(QMouseEvent *event)
32 {
33        lcddisplayX->display(event->x());
34        lcddisplayY->display(event->y());
35          // QWidget::mouseMoveEvent(event);
36 }
```

Listing 3.11. Przykład obsługi zdarzeń myszy – plik main.cpp

```
1 #include <QtGui>
2 #include "mywidget.h"
3
4 int main(int argc, char *argv[])
5 {
6     QApplication a(argc, argv);
7     MyWidget w;
8
9     w.show();
10
11     return a.exec();
12 }
```

Funkcja `mousePressEvent` jest wywoływana w momencie kliknięcia przycisku myszy, gdy kursor znajduje się nad widgetem. Przekazywany do niej obiekt klasy `QMouseEvent` posiada funkcję składową `button()`, która zwraca informację (typu `Qt::MouseButton`) o przycisku, którego naciśnięcie wywołało zdarzenie. Naciśnięcie lewego, prawego lub środkowego przycisku oznaczają wartości: `Qt::LeftButton`, `Qt::RightButton`, `Qt::MidButton`, odpowiednio.

Nową implementację funkcji `mousePressEvent` zawierają wiersze 18–29 listingu 3.10. Obsługę zdarzenia można w całości zaimplementować samodzielnie, lub też dodatkowo skorzystać ze standardowej funkcji obsługi, z klasy bazowej. W naszym przykładzie obsłużono jedynie naciśnięcie lewe-

Rysunek 3.5. Wynik działania programu z obsługą zdarzeń myszy

go lub prawego przycisku. Wyświetlany jest wtedy odpowiedni komunikat. Pozostałe sytuacje są obsługiwane standardowo, dzięki wywołaniu funkcji `QWidget::mousePressEvent(event)` z klasy bazowej, w 27. wierszu.

Funkcja `mouseMoveEvent` umożliwia uzyskanie informacji o współrzędnych (ekranowych lub względem widgetu) kursora myszy nad widgetem. Domyślnie (również w naszym programie) zdarzenia generowane są tylko podczas ruchu myszy z naciśniętym którymkolwiek przyciskiem. W razie potrzeby, można włączyć śledzenie myszy (korzystając z funkcji `setMouseTracking`). Widget będzie wówczas otrzymywał zdarzenia dotyczące ruchu myszy, również bez naciśniętego przycisku.

Przy tworzeniu programów realizujących operacje graficzne, istotna może okazać się znajomość funkcji obsługi zdarzenia `QPaintEvent`:

```
void QWidget::paintEvent ( QPaintEvent * event )
```

Jest ona odpowiedzialna za powtórne wyświetlenie widgetu na ekranie. Wywoływana jest w wyniku działania funkcji `repaint()` lub `update()`, odsłonięcia uprzednio przysłoniętego widgetu i w wielu innych sytuacjach. W przypadku nieskomplikowanych widgetów, zwykle w razie potrzeby powtórnie rysowany jest cały widget. Przy bardziej rozbudowanych może to być nieoptymalne. Można wówczas odświeżać tylko fragment, który tego wymaga (`QPaintEvent::region()`).

Kolejnym sposobem optymalizacji wyświetlania jest automatyczne łączenie kilku zdarzeń w jedno. Dzieje się tak, gdy wywoływana jest funkcja odświeżająca widget – `update()`, tzn. kilkukrotne jej użycie zwykle skutkuje

jednokrotnym wygenerowaniem zdarzenia. Optymalizacji takiej nie zapewnia natomiast funkcja `repaint()`. Działa ona podobnie jak `update()`, lecz funkcja `paintEvent()` jest wywoływana natychmiast.

Począwszy od wersji 4.0 Qt, `QWidget` zapewnia automatyczne podwójne buforowanie operacji swojego wyświetlania, więc samodzielna implementacja tego mechanizmu w funkcji `paintEvent()`, w celu redukcji migotania, nie jest już potrzebna.

Rozdział 4

Grafika 2D

4.1. Wstęp **54**
4.2. Rysowanie rastrowe **55**
4.2.1. Dostęp do punktów 56
4.2.2. Przykładowy program 58
4.3. Rysowanie Painterem **61**
4.3.1. Transformacje 64
4.4. Przeglądarka obrazów **68**

4.1. Wstęp

Biblioteki Qt wspierają tworzenie grafiki zarówno rastrowej jak i wektorowej. Dzięki specjalnym klasom możliwe jest również wykorzystanie biblioteki OpenGL do generowania grafiki trójwymiarowej lub tworzenie zarządzalnych dwuwymiarowych obiektów graficznych (`QGraphicsItem`) umieszczanych na dedykowanej scenie (`QGraphicsScene`). O ile możliwości Qt w zakresie wspomagania wyświetlania grafiki trójwymiarowej będą przedmiotem dyskusji następnego rozdziału to programowanie grafiki oparte na wizualnych elementach wykracza poza ramy tego podręcznika.

W Qt informacje o obrazie rastrowym przechowywane są w jednym z dwóch typów danych, klasach `QPixmap` i `QImage`. Obraz w klasie `QPixmap` jest przechowywany w wewnętrznej reprezentacji z reguły zarządzanej przez system i jest zoptymalizowany pod kątem szybkości wyświetlania. Nie udostępnia za to prostych mechanizmów dostępu do punktów. W tym przypadku kolor jest zawsze reprezentowany w sposób identyczny z aktualnymi ustawieniami systemu. Zazwyczaj będzie to 4-bajtowa struktura opisująca kolor trzema podstawowymi składowymi koloru (czerwony, zielony, niebieski) oraz przezroczystością w formacie „ARGB”. Klasa `QImage` natomiast, zapewnia bezpośredni dostęp do punktów w zamian za czas niezbędny do konwersji do struktury możliwej do wyświetlenia. W tym przypadku programista ma również o wiele więcej możliwości kodowania informacji o kolorze.

Konwersję pomiędzy obiema tymi klasami zapewniają metody klasy `QPixmap`:

```
QImage  QPixmap::toImage() const
QPixmap QPixmap::fromImage(const QImage &image,
          Qt::ImageConversionFlags flags=Qt::AutoColor)
```

Pierwsza metoda konwertuje aktualną pixmapę do obrazu `QImage` pozostawiając aktualny tryb koloru. Druga metoda jest statyczną metodą konwertującą obiekt klasy `QImage` na pixmapę wykorzystując specjalną flagę konwersji `flags` opisującą sposób konwersji reprezentacji koloru.

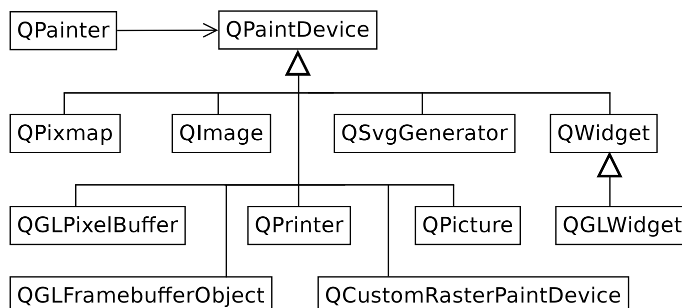
Obraz wektorowy może być reprezentowany w Qt za pomocą otwartego formatu SVG (ang. Scalable Vector Graphics). Klasą umożliwiającą tworzenie wektorowych obrazów jest `QSvgGenerator`. Konwersja do modelu rastrowego jest możliwa poprzez użycie jednej z wersji metody:

```
void QSvgRenderer::render(QPainter *painter)
```

klasy `QSvgRenderer`. Konwersji odwrotnej, tj. z typu rastrowego do wektorowego biblioteki Qt nie zapewniają.

Zarówno w przypadku obiektów klasy `QPixmap` i `QSvgGenerator` rysowanie

na obrazie jest możliwe jedynie przy użyciu tzw. *paintera* reprezentowanego przez klasę `QPainter`. Klasa ta udostępnia szereg metod rysowania prymitywów na obiektach wywodzących się z klasy `QPaintDevice`. Wspomniana pixmapa oraz generator grafiki SVG nie są jedynymi klasami pochodnymi `QPaintDevice`. Z najistotniejszych warte wspomnienia są `QImage`, `QGLFramebufferObject`, `QGLPixelBuffer` (oba obiekty enkapsułkują funkcjonalność obiektów buforowych biblioteki OpenGL) oraz sam obiekt widgetu, tj. klasa `QWidget`. Wszystkie klasy dziedziczące po `QPaintDevice` i zależność pomiędzy tymi klasami a `QPainter` są pokazane na Rysunku 4.1.



Rysunek 4.1. Diagram zależności pomiędzy klasą `QPainter` a `QPaintDevice` i pochodnymi.

4.2. Rysowanie rastrowe

Właściwy obraz rastrowy w Qt jest reprezentowany przez klasę `QImage`. Daje ona możliwość opisywania koloru za pomocą 1-, 8-, 16-, 24- i 32-bitów w przeróżnych wariantach. O głębi koloru decydujemy w chwili konstruowania obiektu tej klasy za pomocą konstruktora:

```
QImage(int width, int height, Format format)
```

gdzie parametry `width` i `height` definiują rozdzielczość poziomą i pionową obrazu a parametr `format` typu wyliczeniowego opisuje sposób kodowania koloru. Najistotniejsze będą dwie jego wartości:

1) `QImage::Format_RGB32` – obraz jest 32-bitowy, ale ostatni oktet ma zawsze wartość 255 (0xFFRRGGBB)
2) `QImage::Format_ARGB32` – pełny obraz 32-bitowy (0xAARRGGBB).

Bardzo przydatny jest również konstruktor wczytujący obraz z pliku:

```
QImage(QString fileName, const char* format=0)
```

Jeżeli parametr `format` pozostanie domyślny to loader spróbuje zgadnąć na podstawie nagłówka rzeczywisty format pliku graficznego. Głębia obrazu zostanie identyczna z głębią pliku graficznego. W przypadku istniejącego już obiektu posługując się metodą:

```
bool QImage::load(QString fileName, const char* format=0)
```

można wczytać plik `fileName` zastępując tym samym wszystkie dane przechowywane w obiekcie nowo wczytanymi. Analogicznie, do zapisu służy metoda:

```
bool QImage::save(QString fileName, const char* format=0,
                  int quality=-1)
```

W tym przypadku jeżeli nie poda się formatu pliku klasa spróbuje odgadnąć format na podstawie rozszerzenia. Opcjonalny parametr `quality` może mieć wartości -1 (domyślna kompresja) lub wartość z zakresu [0..100] i jest brany pod uwagę jedynie przy formatach umożliwiających kompresję.

Do konwersji pomiędzy formatami można wykorzystać metodę:

```
QImage QImage::convertToFormat(Format format)
```

która zwraca nowy obraz w formacie `format`.

4.2.1. Dostęp do punktów

Dostęp do pikseli zawartych w obrazie można uzyskać w dwojaki sposób. Pierwszy, prostszy polega na użyciu akcesorów:

```
void QImage::setPixel(int x, int y, unsigned int value)
QRgb QImage::pixel(int x, int y)
```

Niestety, funkcje te są dosyć kosztowne i przy transformacjach obrazu wymagających dużej wydajności nie sprawdzają się. Dużo efektywniejszym wyborem jest użycie jednej z dwóch metod:

```
char* QImage::bits();
char* QImage::scanLine(int i)
```

Pierwsza metoda jest odpowiednikiem dla wywołania metody `scanLine(0)`. W obu przypadkach metoda zwraca wskaźnik ustawiony na początek podanej w parametrze linii. Typ wskaźnika można w razie potrzeby rzutować na odpowiednią strukturę opisującą piksel. Typowy przykład wypełnienia całego obrazu konkretnym kolorem może wyglądać tak:

Listing 4.1. Wypełnienie obrazu zadanym kolorem.

```
1  QImage image(500, 600, QImage::Format_RGB32);
2  QRgb color = qRgb(255, 128, 64);
3  for(int y=0; y<image.height(); y++)
4  {
5      QRgb* p = (QRgb*)image.scanLine(y);
6      for(int x=0; x<image.width(); x++)
7      {
8          p[x] = color;
9      }
10 }
```

Kolejność pętli, czyli iteracja najpierw po liniach a wewnątrz po kolumnach ma znaczenie jedynie wydajnościowe. Klasa `QImage` przechowuje dane obrazu w jednowymiarowej tablicy o wielkości (wysokość $\times$ szerokość $\times$ ilość_bajtów_na_piksel) łącząc ze sobą kolejne linie obrazu, tzn. w bajcie o 1 dalszym niż ostatni bajt pierwszej linii jest pierwszy bajt drugiej linii. Warto przy tym pamiętać, że metoda `scanLine()` zwraca wskaźnik do pamięci zatem nie może on być użyty poza właściwym obszarem pamięci, na którym znajdują się dane obrazu. Pamiętając, że jest to obszar ciągły i jednowymiarowy w najprostszym przypadku należy tylko pilnować czy nie próbujemy zapisać jakiejś informacji przed pierwszą i za ostatnią linią obrazu. Mając to na uwadze Listing 4.1 można napisać w krótszy sposób:

Listing 4.2. Wypełnienie obrazu zadanym kolorem.

```
1  QImage image(500, 600, QImage::Format_RGB32);
2  QRgb color = qRgb(255, 128, 64);
3  QRgb* p = (QRgb*)image.bits();
4  for(int i=0; i<image.height()*image.width(); i++)
5      p[i] = color;
```

Jeszcze prostszym sposobem wypełnienia obrazu jest użycie metody:

```
void QImage::fill(unsigned int pixelValue)
```

Pomimo, że w Qt kolor jest reprezentowany przez klasę `QColor` dla niskopoziomowych przekształceń wydajniejszym rozwiązaniem będzie użyty już w powyższych przykładach typ `QRgb`, który jest ekwiwalentem typu `unsigned int`, a dokładnie:

```
typedef unsigned int QRgb
```

Funkcje ułatwiające operowanie tym typem to:

```
1  int qAlpha ( QRgb rgba )
2  int qBlue  ( QRgb rgb )
3  int qGreen ( QRgb rgb )
```

```
4 int qRed   ( QRgb rgb )
5 QRgb qRgb  ( int r, int g, int b )
6 QRgb qRgba ( int r, int g, int b, int a )
```

Pierwsze cztery pozycje pozwalają na wyłuskanie wartości konkretnej składowej koloru, a pozycje 5 i 6 ułatwiają składanie pełnego koloru z pojedynczych składowych.

4.2.2. Przykładowy program

Przeanalizujmy prosty program wczytujący obraz z pliku `"sample.png"` i wyświetlający go na formularzu. Dodatkową funkcjonalnością jest możliwość rozjaśnienia i przyciemnienia wyświetlanego obrazu za pomocą kursorów 'Góra' i 'Dół'.

Zgodnie z regułami biblioteki Qt plik z funkcją główną w najprostszej postaci może wyglądać następująco:

Listing 4.3. Plik main.cpp.

```
1 #include <QApplication>
2 #include "imagewidget.h"
3
4 int main(int argc, char* argv[])
5 {
6     QApplication app(argc, argv);
7     ImageWidget iw;
8     iw.show();
9     app.exec();
10 }
```

Klasa `ImageWidget` jest klasą użytkownika zdefiniowaną na listingu 4.4 oraz 4.5.

Listing 4.4. Plik imagewidget.h

```
1 #include <QWidget>
2 #include <QImage>
3
4 class ImageWidget : public QWidget
5 {
6     QImage image;
7     QImage resultImage;
8     int bright;
9     void processImage();
10
11   protected:
12     virtual void keyPressEvent(QKeyEvent*);
13     virtual void paintEvent(QPaintEvent*);
```

```
14
15   public:
16     ImageWidget(QWidget *parent=0);
17 };
```

Klasa `ImageWidget` dziedziczy z klasy `QWidget` i będzie stanowić formularz, na którym zostanie narysowany obraz.

Obraz `image` jest prywatną składową tej klasy i będzie przechowywał oryginalny wczytany z pliku obraz. Obiekt `resultImage` będzie zawierał przekształcony obraz. Zmienna `bright` będzie przechowywać wartość liczbową o jaką należy rozjaśnić/przyciemnić obraz.

Prywatna metoda `void processImage()` będzie odpowiadała za przetworzenie obrazu. Metoda `keyPressEvent()` będzie odpowiadała za reakcję na naciśnięcie przycisku klawiatury.

Samo rysowanie obrazu na formularzu będzie się odbywało w metodzie `paintEvent()`. Jest to wirtualna metoda wywoływana przez system zawsze gdy niezbędne jest narysowanie bądź odrysowanie formularza. Jej wywołanie można również wymusić niebezpośrednio poprzez wywołanie wirtualnej metody:

```
void QWidget::update()
```

(lub którejś z jej wersji) definiowanej klasy widgetu. Metoda ta nie powoduje jednakże natychmiastowego przerysowania a kolejkuje zdarzenie odrysowania w wątku głównym aplikacji. Dzięki temu wielokrotne wywołania w bardzo krótkim czasie metody `update()` skutkują tylko jednokrotnym wywołaniem zdarzenia `paintEvent()` optymalizując proces wyświetlania i zapobiegając migotaniu obrazu. W przypadku gdy niezbędne jest natychmiastowe odrysowanie widgetu można użyć metody:

```
void QWidget::repaint()
```

lub którejś z jej wersji.

Definicje metod tej klasy są przedstawione na poniższym listingu:

Listing 4.5. Plik imagewidget.cpp

```
1 #include "imagewidget.h"
2 #include <QPainter>
3 #include <QKeyEvent>
4 using namespace std;
5
6 ImageWidget::ImageWidget(QWidget* parent)
7     : QWidget(parent)
8 {
```

```
 9   image = QImage("sample.png");
10   resultImage = image;
11   bright = 0;
12 }
13 
14 void ImageWidget::paintEvent(QPaintEvent*)
15 {
16   QPainter painter(this);
17   painter.drawImage(0, 0, resultImage);
18 }
19 
20 void ImageWidget::processImage()
21 {
22   for(int y=0; y<image.height(); y++)
23   {
24     QRgb* pd = (QRgb*)resultImage.scanLine(y);
25     QRgb* ps = (QRgb*)image.scanLine(y);
26     for(int x=0; x<image.width(); x++)
27     {
28       int r = max(0, min(255, qRed(ps[x])   + bright));
29       int g = max(0, min(255, qGreen(ps[x]) + bright));
30       int b = max(0, min(255, qBlue(ps[x])  + bright));
31       pd[x] = qRgba(r,g,b,qAlpha(ps[x]));
32     }
33   }
34 }
35 
36 void ImageWidget::keyPressEvent(QKeyEvent* e)
37 {
38   if(e->key()==Qt::Key_Up)   bright += 1;
39   if(e->key()==Qt::Key_Down) bright -= 1;
40   processImage();
41   update();
42 }
```

W konstruktorze tworzony jest obiekt klasy `QImage` i wczytywana do niego jest zawartość pliku graficznego `"sample.png"`. W następnej linii konstruktora obiektowi `resultImage` zostanie przypisana wartość obrazu `image`. Na końcu zmienna przechowująca aktualną wartość jasności `bright` zostaje wyzerowana.

Metoda `paintEvent()` w swoim ciele tworzy obiekt typu `QPainter` i wiąże go z własną klasą poprzez podanie wskaźnika `this` do konstruktora obiektu. Klasa `QPainter` jest główną klasą rysującą w Qt dyskutowaną szerzej w następnym podrozdziale. Tu została wykorzystana do narysowania obrazu `image` na formularzu, poprzez wywołanie metody:

```
void QPainter::drawImage(int x, int y, QImage &image)
```

Parametry `x` i `y` definiują współrzędne na formularzu gdzie będzie lewy, górny róg obrazu `image`.

Metoda `processImage()` to główna funkcja przetwarzająca obraz. W liniach 22-33 zawarta została cała procedura zmiany jasności każdego punktu w obrazie. Dla każdej linii pobierany jest wskaźnik na jej początek dla obu obrazów `image` (wskaźnik `ps`) i `resultImage` (wskaźnik `pd`). Następnie dla każdej kolumny w danym wierszu obliczane są nowe wartości składowych RGB poprzez dodanie wartości `bright` do wartości kolejnych składowych pozyskanych z obrazu oryginalnego. Funkcje `min(int, int)` oraz `max(int, int)` zabezpieczają wynikową wartość składowej koloru przed przekroczeniem dopuszczalnego zakresu [0..255]. W linii 31 nowe wartości RGB składane są w ostateczny kolor i przypisywane do odpowiedniego punktu obrazu `resultImage`.

Metoda `keyPressEvent()` jest odpowiedzialna za obsługę zdarzenia naciśnięcia przycisku na klawiaturze. W parametrze `e` zawarte zostaną niezbędne informacje o stanie klawiatury, które w liniach 16 i 17 są wykorzystywane do wykrycia naciśnięcia strzałki kursorów. W przypadku naciśnięcia strzałki do góry, opisanej wartością `Qt::Key_Up` zmienna jasności `bright` zostanie zwiększona o 1. Analogicznie dla strzałki do dołu `Qt::Key_Down` zmienna `bright` zostanie zmniejszona o 1. Następnie zostanie wywołana metoda przetwarzająca obraz `processImage()` i cały widget zostanie odświeżony za pomocą metody `update()`.

4.3. Rysowanie Painterem

Klasę `QPainter` można scharakteryzować jako wysoko zoptymalizowany zestaw funkcji rysujących. Klasa ta, realizując funkcje rysowania, stara się wykorzystywać dedykowane systemowe funkcje, zwykle wspomagane sprzętowo. Przykładem może tu być możliwość rysowania za pomocą *paintera* po obiektach korzystających z biblioteki OpenGL.

Podstawową jego funkcjonalnością jest rysowanie prymitywów za pomocą szeregu metod:

```
void QPainter::drawArc(...)
void QPainter::drawLine(...)
void QPainter::drawEllipse(...)
void QPainter::drawPath(...)
void QPainter::drawPoint(...)
void QPainter::drawPolygon(...)
void QPainter::drawRect(...)
```

rysujących kolejno łuk, linię, elipsę, ścieżkę, punkt, wielokąt, prostokąt. W każdym przypadku zestaw parametrów jest trochę inny, istnieje również wiele przeładowanych metod różniących się parametrami.

Poza prymitywami *painter* umożliwia również odrysowanie na danym urządzeniu obrazu dzięki metodom:

```
void QPainter::drawImage(...)
void QPainter::drawPixmap(...)
```

oraz rendering tekstu:

```
void QPainter::drawText(...)
```

W każdym przypadku rysowanie za pomocą *paintera* odbywa się na pewnym *rysowalnym* urządzeniu (ang. *painting device*) reprezentowanym przez klasę `QPaintDevice` oraz jej pochodne. Na rysunku 4.1 zestawiono zależność pomiędzy klasą `QPainter` a urządzeniami `QPaintDevice`.

Zazwyczaj sama procedura rysująca rozpoczyna się w ciele metody `paintEvent(...)` inicjalizacją obiektu *paintera* i skojarzeniem go z danym urządzeniem.

Listing 4.6. Rysowanie za pomocą *paintera*.

```
1 void Widget::paintEvent(QPaintEvent *e)
2 {
3   QPainter paint(this);
4   paint.setPen(QPen(QColor(0, 20, 100)));
5   paint.drawRect(QRect(10, 10, 100, 100));
6   paint.drawText(10, 20, "Testowy napis");
7 }
```

W tym przypadku został stworzony automatyczny obiekt `paint` a skojarzenie z obiektem klasy `QWidget` nastąpiło poprzez przekazanie w parametrze konstuktora wskaźnika `this`. Po inicjalizacji, w liniach 4–6 wywoływane są odpowiednie metody rysujące *paintera*.

Ten sam obiekt *paintera* może być wykorzystywany wielokrotnie do rysowania na różnych urządzeniach. Zawsze jednak procedury rysujące muszą być wywoływane pomiędzy wywołaniami metod:

```
void QPainter::begin(QPaintDevice * device)
void QPainter::end()
```

Ilustruje to poniższy przykład:

Listing 4.7. Rysowanie za pomocą *paintera*.

```
1  void Widget::paintEvent(QPaintEvent *e)
2  {
3    QPixmap pixmap(256,256);
4
5    QPainter paint(&pixmap);
6      QLinearGradient grad(QPointF(0,0), QPointF(0,256));
7      grad.setColorAt(0, Qt::black);
8      grad.setColorAt(1, Qt::white);
9      paint.fillRect(0,0,256,256, QBrush(grad));
10   paint.end();
11
12   paint.begin(this);
13     paint.drawPixmap(0,0, pixmap);
14 }
```

W linii 5 w konstruktorze klasy `QPainter`, obiekt *paintera* został skojarzony z obiektem piksmapy `pixmap`, zastępując tym samym wywołanie metody `QPainter::begin(...)`. W kolejnych liniach na tym obrazie rysowany jest wypełniony prostokąt metodą:

```
void QPainter::fillRect(int, int, int, int, const QBrush&)
```

Prostokąt zostanie wypełniony liniowym gradientem pionowym w kolorze od czarnego do białego. W linii 10 metoda `QPainter::end()` kończy rysowanie na piksmapie a w linii 12 *painter* jest kojarzony z nowym obiektem za pomocą metody `QPainter::begin(...)`. Wywołanie metody `QPainter::end()` po odrysowaniu piksmapy nie jest już konieczne, ponieważ destruktor *paintera* wywoła ją niejawnie. Nie jest dopuszczalne użycie dwóch aktywnych *painterów* jednocześnie.

Klasa `QPainter` posiada szereg metod dostosowujących styl rysowania do konkretnych wymagań. Dla prymitywów możliwe jest zdefiniowanie koloru, pióra oraz pędzla, dla tekstu dodatkowo możliwe jest wyspecyfikowanie kroju pisma. Wybór pióra, a dokładniej ustawienie egzemplarza obiektu klasy `QPen` za pomocą metody:

```
void QPainter::setPen(const QPen&)
```

definiuje sposób rysowania linii i obramowań prymitywów oraz kolor prymitywów i tekstu. Pędzel, definiowany klasą `QBrush` i ustawiany za pomocą metody:

```
void QPainter::setBrush(const QBrush&)
```

definiuje sposób wypełniania kształtów. Krój pisma jest wybierany poprzez przekazanie obiektu klasy `QFont` do klasy paintera metodą:

```
void QPainter::setFont(const QFont&)
```

4.3.1. Transformacje

Domyślnie `QPainter` wykonuje działania w układzie współrzędnych należącym do obiektu, na którym rysuje. Jednakże, dzięki wbudowanym metodom układ współrzędnych może być przetransformowany. Podstawowe funkcje realizujące transformacje to:

```
void QPainter::translate(qral dx, qreal dy)
void QPainter::rotate(qreal angle)
void QPainter::scale(qreal sx, qreal sy)
void QPainter::shear(qreal sh, qreal sv)
```

Pierwsza przesuwa globalny układ współrzędnych o wartości `dx` i `dy`. Druga obraca aktualny układ współrzędnych zgodnie z ruchem wskazówek zegara o `angle` stopni względem początku układu współrzędnych. Trzecia przeskalowuje układ współrzędnych o wartości `sx` i `sy`. Czwarta pochyla ten układ o wartości `sh` i `sv`. Wszystkie te funkcje modyfikują macierz transformacji danego *paintera*. Aktualną macierz można uzyskać dzięki metodzie:

```
const QTransform& QPainter::worldTransform()
```

zwracającej obiekt klasy `QTransform`. Umożliwia to dokonanie dowolnego afinicznego przekształcenia w przestrzeni dwuwymiarowej dzięki odpowiednim metodom tej klasy. Tak przygotowaną macierz transformacji można przypisać do danego *paintera* za pomocą metody:

```
void QPainter::setTransform(const QTransform &transform,
                            bool combine=false)
```

Jeżeli parametr `combine` będzie miał wartość `false` wtedy aktualna macierz paintera zostanie zastąpiona tą z obiektu `transform`, w innym przypadku obie macierze zostaną połączone.

Przykładowe wykorzystanie stanów *paintera* oraz możliwości transformacji przedstawione są na listingu 4.8 oraz na rysunkach 4.2.

Listing 4.8. Przykład zastosowania transformacji paintera.

```
1 void ImageWidget::paintEvent(QPaintEvent *e)
2 {
3   QPainter paint(&image);
```

```
4
5     QString string("Test String");
6     QPoint center(150, 150);
7
8     paint.save();
9     transformCoordinates(paint, center);
10    drawBox(paint, "String Testowy");
11    paint.restore();
12
13    drawWaterMark(paint, center);
14    drawCoordinates(paint, center);
15
16    paint.end();
17    paint.begin(this);
18    paint.drawImage(0,0, internalImage);
19  }
20
21  void drawBox(QPainter& paint, const QString &text)
22  {
23    paint.save();
24
25    QPoint margin(20, 5);
26
27    QPen pen(QColor(180, 20, 30));
28    paint.setPen(pen);
29
30    QFont font("Times");
31    font.setWeight(QFont::Bold);
32    font.setPointSize(24);
33    paint.setFont(font);
34    QRect rect = paint.boundingRect(0, 0, 200, 200,
35                        Qt::AlignLeft|Qt::AlignTop, text);
36    paint.drawText(margin.x(),rect.height()-margin.y(),text);
37
38    pen.setColor(QColor(100, 10, 200));
39    pen.setWidth(5);
40    paint.setPen(pen);
41    paint.drawRect(0, 0, rect.width()+margin.x()*2,
42       rect.height()+margin.y()*2);
43    paint.fillRect(0, 0, rect.width()+margin.x()*2,
44                   rect.height()+margin.y()*2,
45                   QBrush(QColor(120, 120, 220, 64)));
46
47    pen.setWidth(4);
48    pen.setColor(QColor(Qt::black));
49    paint.setPen(pen);
50    paint.drawArc(-margin.x()/2, -margin.x()/2,
51                  margin.x(), margin.x(), 0, 270*16);
52
53    paint.restore();
54  }
```

```
55
56 void drawWaterMark(QPainter &paint, QPoint &center)
57 {
58   paint.save();
59   QFont font = paint.font();
60   font.setPointSize(48);
61   font.setItalic(true);
62   font.setFamily("Times");
63   paint.setFont(font);
64
65   QPen pen = paint.pen();
66   pen.setColor(QColor(100, 100, 100, 64));
67   paint.setPen(pen);
68
69   paint.translate(center.x()-50, center.y()+120);
70   paint.drawText(0, 0, "Watermark");
71   paint.restore();
72 }
73
74 void drawCoordinates(QPainter &paint, QPoint &center)
75 {
76   paint.drawLine(0, center.y(), paint.device()->width(),
77                  center.y());
78   paint.drawLine(center.x(), 0, center.x(),
79                  paint.device()->height());
80 }
```

W metodzie `paintEvent(...)` w linii 3 zostało założone istnienie obiektu `image` klasy `QImage`. W liniach 34-35 w metodzie `drawBox(...)` została użyta metoda:

```
QRect QPainter::boundingRect(int x, int y, int w, int h,
                             int flags, const QString &text)
```

umożliwiająca oszacowanie wielkości renderowanego napisu `text` w zależności od flag `flags` formatujących tekst.

Kod ten obrazuje również użycie metod:

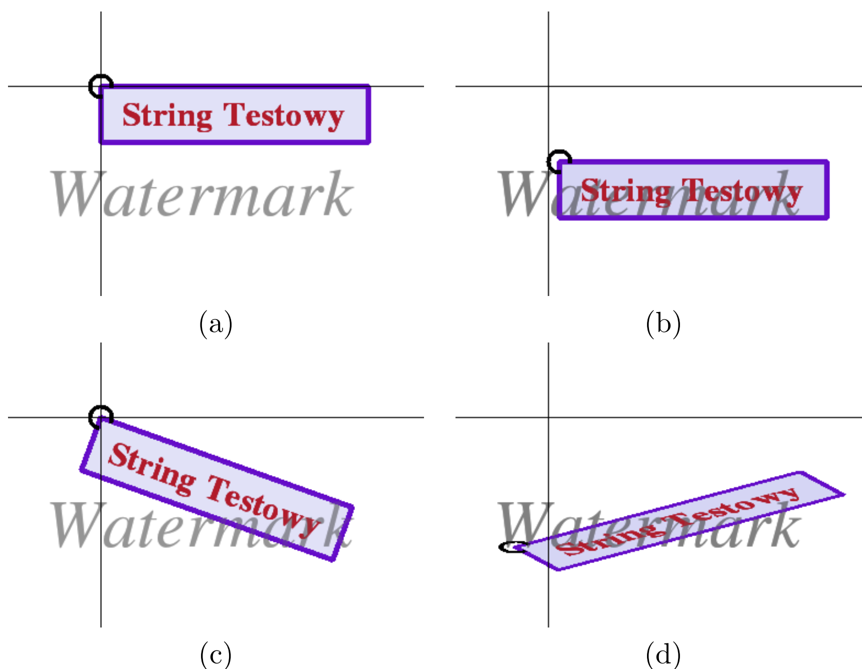
```
void QPainter::save()
void QPainter::restore()
```

Pierwsza z nich odkłada na stos aktualny stan *paintera*, druga zaś przywraca jego stan ze stosu.

W linii 9 została użyta funkcja:

```
void transformCoordinates(QPainter&, const QPoint&)
```

zdefiniowana na listingach 4.9–4.12 odpowiedzialna za konkretne przekształcenie rysowanego kształtu.



Rysunek 4.2. Wynik działania kodu z listingu 4.2 dla funkcji `void transformCoordinates(QPainter&, const QPoint&)` zdefiniowanej kolejno na listingach (a) 4.9, (b) 4.10, (c) 4.11, (d) 4.12.

Listing 4.9. Funkcja transformCoordinates w wersji 1.

```
1 void transformCoordinates(QPainter &paint, const QPoint &c)
2 {
3   paint.translate(c);
4 }
```

Listing 4.10. Funkcja transformCoordinates w wersji 2.

```
1 void transformCoordinates(QPainter &paint, const QPoint &c)
2 {
3   paint.translate(c.x()+10, c.y()+70);
4 }
```

Listing 4.11. Funkcja transformCoordinates w wersji 3.

```
void transformCoordinates(QPainter &paint, const QPoint &c)
{
  paint.translate(c);
  paint.rotate(20);
}
```

Listing 4.12. Funkcja transformCoordinates w wersji 4.

```
void transformCoordinates(QPainter &paint, const QPoint &c)
{
  paint.translate(c.x()-30, c.y()+120);
  paint.translate(margin.x(),0);
  paint.scale(1.3, 0.5);
  paint.translate(c);
  paint.rotate(-35);
}
```

4.4. Przeglądarka obrazów

Poniższy rozdział omawia przykładową, prostą realizację przeglądarki plików graficznych. Typy rozpoznawanych plików graficznych zostały ograniczone do typów obsługiwanych przez bibliotekę Qt. Domyślnie są to: BMP, GIF, JPG, PNG, PBM, PPM, TIFF, XBM, XPM. Obsługa innych typów może być dodane dzięki mechanizmowi pluginów.

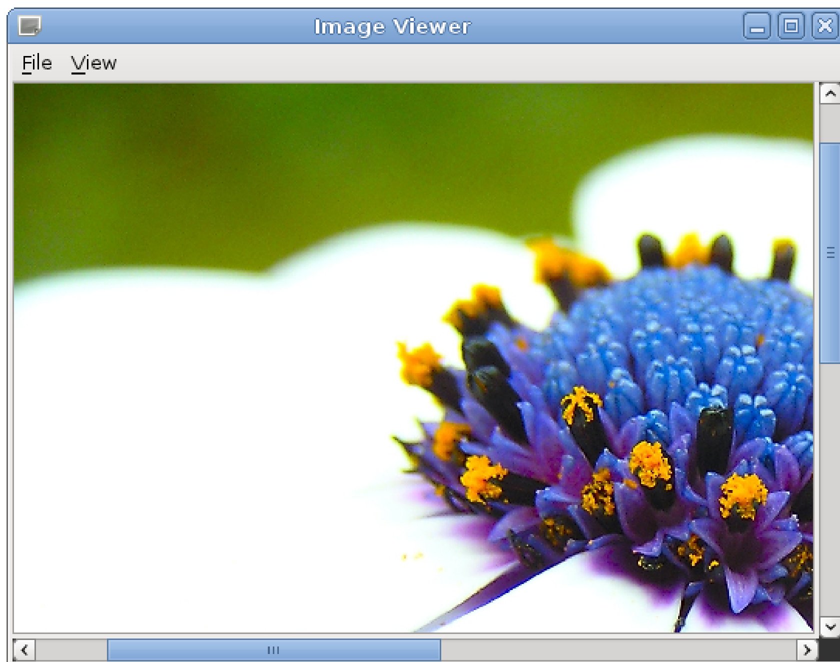
Funkcjonalność przeglądarki sprowadza się do wczytania pliku graficznego, jego wyświetlenia oraz możliwości zmiany rozmiaru wyświetlanego obrazu. Dodatkowo po wczytaniu obrazu można za pomocą myszki odczytać wartość koloru w danym punkcie. Rysunek 4.3 przedstawia aplikację w działaniu.

Plik z funkcją główną startujący aplikację jest dosyć typowy:

Listing 4.13. Plik main.cpp

```
#include <QApplication>
#include "mainwindow.h"

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  MainWindow mainWindow;
  mainWindow.show();
  app.exec();
}
```



Rysunek 4.3. Okno przeglądarki plików graficznych.

Po utworzeniu obiektu `QApplication` tworzony jest obiekt przeglądarki `mainWindow` zdefiniowanej poniżej klasy:

Listing 4.14. Plik mainwindow.h

```
1  #ifndef MAINWINDOW_H
2  #define MAINWINDOW_H
3  #include <QtGui>
4  #include "imagewidget.h"
5
6  class MainWindow : public QMainWindow
7  {
8    Q_OBJECT
9    ImageWidget *imageWidget;
10
11 public:
12   explicit MainWindow(QWidget *parent = 0);
13
14 public slots:
15   void openFile();
16   void info(QString);
17 };
18 #endif
```

Klasa ta dziedziczy po `QMainWindow` ułatwiając tworzenie menu oraz pola statusu na dole okna.

W 3 linii włączany jest metaplik nagłówkowy `QtGui` zawierający polecenie włączenia wszystkich klas modułu QtGui.

Tworzona klasa będzie korzystała z mechanizmu slotów i sygnałów zatem tuż po otwarciu definicji, w linii 8, musi wystąpić makro `Q_OBJECT` .

Użycie słowa kluczowego **`explicit`** przy specyfikowaniu konstruktora w linii 12 zabezpiecza daną klasę przed niejawną konwersją oraz przed kopiowaniem danego obiektu.

Definicje metod tej klasy zostały przedstawione na poniższym listingu:

Listing 4.15. Plik mainwindow.cpp

```
1 #include "mainwindow.h"
2
3 MainWindow::MainWindow(QWidget *parent) :
4     QMainWindow(parent)
5 {
6   setWindowTitle(tr("Image Viewer"));
7   imageWidget = new ImageWidget();
8   connect(imageWidget, SIGNAL(info(QString)),
9           this, SLOT(info(QString)));
10
11  QMenu *fileMenu = new QMenu(tr("&File"), this);
12  fileMenu->addAction(tr("&Open..."), this,
13            SLOT(openFile()), QKeySequence(tr("Ctrl+O")));
14  fileMenu->addAction(tr("E&xit"), qApp,
15            SLOT(quit()) ,QKeySequence::Quit);
16
17  menuBar()->addMenu(fileMenu);
18
19  QMenu *viewMenu = new QMenu(tr("&View"), this);
20  viewMenu->addAction(tr("Zoom In"), imageWidget,
21            SLOT(zoomIn()), QKeySequence(tr("Ctrl++")));
22  viewMenu->addAction(tr("Zoom Out"), imageWidget,
23            SLOT(zoomOut()), QKeySequence(tr("Ctrl+-")));
24  viewMenu->addAction(tr("Zoom Reset"), imageWidget,
25          SLOT(zoomReset()), QKeySequence(tr("Ctrl+=")));
26  QAction *action = viewMenu->addAction(tr("Smooth Zoom"));
27  action->setShortcut(QKeySequence("Alt+S"));
28  action->setCheckable(true);
29  connect(action, SIGNAL(toggled(bool)),
30          imageWidget, SLOT(smoothZoom(bool)));
31
32  menuBar()->addMenu(viewMenu);
33
34  QScrollArea *scrollArea = new QScrollArea(this);
35  QPalette pal = scrollArea->palette();
36  pal.setBrush(QPalette::Background,
37               QBrush(QColor(50, 50, 50)));
```

```
38   scrollArea->setPalette(pal);
39   scrollArea->setAlignment(Qt::AlignCenter);
40   scrollArea->setWidget(imageWidget);
41
42   setCentralWidget(scrollArea);
43   resize(800,600);
44 }
45
46 void MainWindow::openFile()
47 {
48   QString fileName = QFileDialog::getOpenFileName(this);
49   if(!fileName.isNull())
50   {
51     QImage image(fileName);
52     if(!image.isNull())
53     {
54       imageWidget->setImage(image);
55       imageWidget->setFileName(fileName);
56     }
57   }
58 }
59
60 void MainWindow::info(QString str)
61 {
62   statusBar()->showMessage(str);
63 }
```

- W linii 7 tworzony jest obiekt klasy `ImageWidget` zdefiniowanej na listingu 4.16. Obiekt ten będzie odpowiadał za zarządzanie wyświetlanym obrazem.

- W liniach 11-32 tworzone jest menu górne aplikacji wraz z odpowiednimi połączeniami oraz skrótami klawiszowymi. Pozycja *Smooth Zoom* z menu *View* będzie możliwa do zaznaczenia i będzie odpowiadała za możliwość wygładzenia obrazu przy powiększeniu.

- W liniach 34-40 tworzony jest obiekt klasy `QScrollArea` będący swoistym kontenerem na inne widgety. W przypadku gdy zawierany widget nie mieści się geometrycznie we wnętrzu `QScrollArea` pojawią się dodatkowe suwaki po prawej stronie i na dole umożliwiające przesuwanie widgeta dziecka.

- W liniach 35-38 przedstawiona jest jedna z możliwości zmiany domyślnego tła widgetu za pomocą klasy `QPalette` na ciemno szary.

- W linii 39 ustawiana jest opcja pozycjonowania widgeta dziecka w środku rodzica a w następnej linii przekazywany jest sam widget dziecka. W tym

wypadku będzie to obiekt klasy `ImageWidget`. Od tej pory `scrollArea` staje się rodzicem dla `imageWidget`.

- W linii 42 obiekt `scrollArea` jest ustawiany jako centralny widget całego głównego okna `mainWindow`.
- Zdefiniowana w liniach 46-58 metoda

  ```
  void openFile()
  ```

  za pomocą klasy `QFileDialog` pobiera nazwę pliku do wczytania i tworzy tymczasowy obiekt klasy `QImage`, do którego wczytuje obraz z pliku. Obraz ten jest następnie przekazywany obiektowi `imageWidget` w linii 54. W następnej linii przekazywana jest w celach informacyjnych również nazwa pliku.
- Zdefiniowany w liniach 60-63 slot

  ```
  void info(QString)
  ```

  wypisuje na pasku statusu na dole okna przekazany w parametrze napis.

Na listingu 4.16 zdefiniowana została klasa `ImageWidget` zarządzająca obrazem wywodząca się z klasy `QWidget`.

Listing 4.16. Plik imagewidget.h

```
1 #ifndef IMAGEWIDGET_H
2 #define IMAGEWIDGET_H
3 #include <QtGui>
4
5 class ImageWidget : public QWidget
6 {
7   Q_OBJECT
8   QImage internalImage;
9   QString fileName;
10  float zoom;
11  bool smooth
12
13 protected:
14  void mouseMoveEvent(QMouseEvent *);
15  void mouseReleaseEvent(QMouseEvent *);
16  void wheelEvent(QWheelEvent *);
17  void paintEvent(QPaintEvent *);
18
19 public:
20  void setImage(const QImage&);
21  void setFileName(const QString&);
```

```
22
23 signals:
24   void info(QString);
25
26 public slots:
27   void zoomIn();
28   void zoomOut();
29   void zoomReset();
30   void zoomSmooth(bool);
31 };
32 #endif
```

Jako klasa zarządzająca wyświetlanym obrazem będzie agregowała ten obraz w zmiennej `internalImage` typu `QImage`. Zastosowanie tego typu pozwoli na swobodny dostęp do punktów obrazu.

Klasa ta będzie również obsługiwała zdarzenia przesunięcia kursora myszki, obrócenia kółka myszki oraz puszczenia przycisku myszki.

Zdefiniowany został jeden sygnał `info(QString)` oraz 4 sloty realizujące operacje powiększania/pomniejszania wyświetlanego obrazu.

Metody tej klasy zostały zdefiniowane na poniższym listingu.

Listing 4.17. Plik imagewidget.cpp

```
 1 #include "imagewidget.h"
 2
 3 void ImageWidget::setImage(const QImage & im)
 4 {
 5     smooth = false;
 6     internalImage = im;
 7     zoomReset();
 8 }
 9
10 void ImageWidget::setFileName(const QString& str)
11 {
12   fileName = str.section('/', -1);
13 }
14
15 void ImageWidget::paintEvent(QPaintEvent *e)
16 {
17   if(!internalImage.isNull())
18   {
19     QPainter paint(this);
20     if(smooth)
21       paint.setRenderHint(
22             QPainter::SmoothPixmapTransform, true);
23     else
24       paint.setRenderHint(
25             QPainter::SmoothPixmapTransform, false);
26     paint.scale(zoom, zoom);
```

```
27      paint.drawImage(0,0, internalImage);
28    }
29  }
30
31  void ImageWidget::mouseMoveEvent(QMouseEvent *e)
32  {
33    if(!internalImage.isNull())
34    {
35      QPoint pos = e->pos()/zoom;
36      if(pos.x()>0 && pos.x()<internalImage.width() &&
37         pos.y()>0 && pos.y()<internalImage.height() )
38      {
39        QRgb color = internalImage.pixel(pos);
40        QString str = QString(tr("Point at [%1,%2],"
41                                 "RGB Color [%3, %4, %5]"))
42                                 .arg(pos.x()).arg(pos.y())
43                                 .arg(qRed(color))
44                                 .arg(qGreen(color))
45                                 .arg(qBlue(color));
46        emit info(str);
47      }
48    }
49  }
50
51  void ImageWidget::mouseReleaseEvent(QMouseEvent *)
52  {
53    if(!internalImage.isNull())
54    {
55      QString str = QString(tr("Image [%1], "
56                               "Size [%2 x %3]"))
57                               .arg(fileName)
58                               .arg(internalImage.width())
59                               .arg(internalImage.height());
60      emit info(str);
61    }
62  }
63
64  void ImageWidget::wheelEvent(QWheelEvent *e)
65  {
66    if(e->modifiers() == Qt::ControlModifier)
67    {
68      if(e->delta() > 0) zoomIn();
69      else if(e->delta() < 0) zoomOut();
70    }
71    else parent()->event(e);
72  }
73
74  void ImageWidget::zoomIn()
75  {
76    zoom *= 1.25;
77    if(zoom >= 8.0f) zoom = 8.0f;
```

```
78   setFixedSize(internalImage.size()*zoom);
79   update();
80 }
81
82 void ImageWidget::zoomOut()
83 {
84   zoom /= 1.25;
85   if(zoom <= 0.1f) zoom = 0.1f;
86   setFixedSize(internalImage.size()*zoom);
87   update();
88 }
89
90 void ImageWidget::zoomReset()
91 {
92   zoom = 1.0f;
93   setFixedSize(internalImage.size()*zoom);
94   update();
95 }
96
97 void ImageWidget::smoothZoom(bool v)
98 {
99   smooth = v;
100  update();
101 }
```

- Metoda `setImage(const QImage&)` zdefiniowana w liniach 3-8 kopiuje zawartość obrazu przekazanego w parametrze, zeruje wartość zmiennej wygładzania `smooth` i resetuje powiększenie za pomocą metody `zoomReset()`.
- W linii 12 w ciele metody `setFileName(const QString&)` kopiowany jest podciąg napisu zawierającego nazwę wczytanego pliku graficznego. Metoda `QString::section(...)` zwraca podciąg napisu począwszy od ostatniego wystąpienia znaku `'/'` do końca, co w powyższym przypadku usuwa ścieżkę dostępu do pliku pozostawiając samą nazwę pliku.
- Metoda `paintEvent(...)` zdefiniowana w liniach 15-29 rysuje obraz `internalImage` na powierzchni widgeta. W przypadku ustawienia wartości parametru wygładzania `smooth` w liniach 20-25 ustawiana jest odpowiednia opcja renderingu obrazu zapewniając lepszą wydajność albo lepsze efekty wizualne. W linii 26 zostaje przeskalowany układ współrzędnych paintera, skalując jednocześnie rysowany w następnej linii obraz.
- Metoda `mouseMoveEvent(QMouseEvent*)` jest wywoływana w reakcji na zdarzenie ruchu myszką w obrębie danego widgetu. W zależności od stanu Własności (Property) `mouseTracking` zdarzenie jest wywoływane zawsze (`mouseTracking==true`) lub tylko wtedy gdy równocześnie jest

wciśnięty przycisk myszy (`mouseTracking==false`). Domyślnie Własność `mouseTracking` jest wyłączona co zwalnia z potrzeby jej ręcznego ustawiania. W linii 35 odczytywane jest aktualne położenie kursora w obrębie widgetu z parametru `e`. Jeżeli współrzędne zawierają się w obrazie (linie 36-37) wtedy pobierana jest wartość koloru punktu o odpowiednich współrzędnych. Linie 40-45 pokazują jedną z wydajniejszych metod składania napisu złożonego w innych napisów i wartości numerycznych. Wykorzystana została tutaj metoda:

```
QString QString::arg(int a)
```

która zastępuje w aktualnym stringu kolejne wystąpienia znaczników (%1, %2, itd.) wartością argumentu `a`. Innym sposobem składania skomplikowanych stringów jest użycie klasy `QTextStream` analogicznie do klas strumieniowych. Zawarty w liniach 40-45 kod mógłby mieć wtedy postać:

```
QString str;
QTextStream(&str) << "Point at [" <<pos.x() << "," <<pos.y()
                  << "], RGB Color [" << qRed(color)
                  << ", " << qGreen(color) << ", "
                  << qBlue(color) << "]";
```

Na końcu tej metody w linii 46 emitowany jest sygnał `info(QString)` zawierający stworzony napis.

- Obsługa zdarzenia puszczenia przycisku myszki zdefiniowana jest w metodzie `mouseReleaseEvent(...)` w liniach 51-62. Na podobnej zasadzie tworzony jest tutaj napis zawierający nazwę pliku oraz jego rozdzielczość. Napis ten jest wysyłany w parametrze sygnału `info(QString)` w linii 60.

- W liniach 64-72 zdefiniowana jest obsługa zdarzenia obrotu kółka myszy. Funkcjonalność tej metody ograniczy się do skalowania obrazu przy obrocie kółka myszy o ile wciśnięty jest klawisz 'Ctrl'. Najpierw w 66 linii testowane są modyfikatory zdarzenia. W tym przypadku istotny jest tylko modyfikator `Qt::ControlModifier` opisujący stan przycisku 'Ctrl'. Metoda:

  ```
  int QWheelEvent::delta()
  ```

  zwraca wartość liczbową o ile zostało obrócone kółko myszy. Wartość ta jest dodatnia w przypadku obrotu od użytkownika i ujemna w przypadku obrotu do użytkownika. Jeżeli przycisk 'Ctrl' nie był wciśnięty podczas obrotu kółka myszy (linia 71) obsługa zdarzenia zostanie przekazana do rodzica, którym w tym przypadku jest obiekt klasy `QScrollArea`.

- Metody zoomowania zdefiniowane w liniach 74-95 modyfikują odpowiednio współczynnik skalowania obrazu dbając jednocześnie o to aby ten współczynnik nie przekroczył pewnych wartości granicznych. Niezbędne jest również ustalenie nowego rozmiaru dla całego widgeta na rozmiar przeskalowanego obrazu. Pominięcie tego kroku skutkowałoby widgetem o minimalnej wielkości, ponieważ nie istnieje żadne bezpośrednie powiązanie pomiędzy wielkością widgeta a wielkością wyświetlanego obrazu.

Rozdział 5

Grafika 3D

5.1. Wstęp **80**
5.2. Widget OpenGL **80**

5.1. Wstęp

Wróćmy do rysunku 4.1. Pomiędzy klasami dziedziczącymi po `QPainer` i mającymi związek z tworzeniem i zarządzaniem grafiki występuje również `QGLWidget`. *OpenGL* to niepodważalny standard tworzenia grafiki trójwymiarowej czasu rzeczywistego. Specyfikacja API wywodzi się z biblioteki utworzonej przez Silcon Graphics na początku lat dziewięćdziesiątych. Została przyjęta przez dużych producentów tak sprzętu jak i oprogramowania działających w ramach powołanego w 1992 roku Architecture Review Board. Obecnie najnowsza jej wersja 4.2 datowana jest na sierpień 2011 i zawiera około 250 funkcji pozwalających na kompleksowe kształtowanie w elastyczny sposób warstwy wizualnej aplikacji niezależnie od tego czy jest to system inżynierski, oprogramowanie do wizualizacji czy gra w środowisku trówymiarowym. *OpenGL* działa w architekturze klient-serwer co pozwala na przeniesienie renderingu na zewnętrzne maszyny. Taka budowa wymagała oparcia się przy tworzeniu systemu na architekturze maszyny stanów, w której określone przez użytkownika parametry i tryby renderingu wpływają na końcowy efekt jej działania.

5.2. Widget OpenGL

Klasa `QGLWidget` ma za zadanie udostępnienie interfejsu pozwalającego na korzystanie z API *OpenGL*. Ze względu na specyficzną architekturę istnieje konieczność specjalnego traktowania tej interakcji poczynając od konieczności inicjalizacji maszyny stanów, poprzez budowę pętli obsługi renderingu na obsłudze zdarzeń specjalnych kończąc. Jednak użycie jej gdy już poznamy zasady rządzące tworzeniem aplikacji Qt jest stosunkowo proste.

Pierwszym krokiem jest stworzenie własnej klasy widgetu, która dziedziczy po `QGLWidget` reimplementując kilka metod wirtualnych. Pierwszą z nich jest `initializeGL(...)` odpowiedzialna za ustawienie kontekstu renderowania, konfiguracji wyświetlania definicji świateł, list wyświetlania i wielu innych elementów typowych dla *OpenGL* co oczywiście wykracza poza tematykę tego skryptu dlatego osoby zainteresowane szczegółami odsyłam do skryptu temu poświęconemu w całości [4].

Kolejna z metod wymagająca implementacji to `paintGL(...)` zwierająca opis sceny i wywoływana za każdym razem gdy konieczne jest jej przerysowanie. Typowo jest ona wywoływana w ramach głównej pętli obsługi.

Jeżeli wielkość widgetu ulegnie zmianie co wpływa tak na rozmiar i proporcje kontekstu renderowania wywoływana jest reimplementowana metoda `resizeGL(...)` przekazująca informacje do maszyny stanów z użyciem funkcji *OpenGL* `glViewport(...)` i podobnych definiujących macierz projekcji.

W przypadku gdy zależy nam na ręcznym wywołaniu przerysowania sceny korzystamy z metody `updateGL(...)` zamiast bezpośredniego wywołania `paintGL(...)`.

Listing 5.1. Plik nagłówkowy klasy `NewGLWidget`.

```
1  #ifndef NEWGLWIDGET_H
2  #define NEWGLWIDGET_H
3
4  #include <QGLWidget>
5
6  class NewGLWidget : public QGLWidget
7  {
8      Q_OBJECT
9
10 public:
11     NewGLWidget(QWidget *parent = 0);
12     ~NewGLWidget();
13     QSize minimumSizeHint() const;
14     QSize sizeHint() const;
15
16 public slots:
17     void setShading(bool mode);
18     void setLighting(bool mode);
19
20 signals:
21
22 protected:
23     void initializeGL();
24     void paintGL();
25     void resizeGL(int width, int height);
26     void mousePressEvent(QMouseEvent *event);
27     void mouseMoveEvent(QMouseEvent *event);
28
29 private:
30     int shading;
31     int lighting;
32     int rotX;
33     int rotY;
34     int rotZ;
35     QPoint pos;
36 };
37
38 #endif
```

Jeżeli przyjrzymy się deklaracjom metod klasy NewGLWidget przedstawionej na listingu 5.1 to poza wspomnianymi w liniach 13 i 14 znajdziemy sugestie tyczące minimalnego i sugerowanego rozmiaru widgetu, dwa sloty odpowiedzialne za zmianę modelu cieniowania w liniach 17 i 18 oraz obsługę

zdarzeń myszy odpowiednio w liniach 26 i 27. W sekcji prywatnej dodatkowo zawarte są pomocnicze zmienne.

Przejdźmy jednak do definicji zawartych w pliku NewQGLWidget.cpp widocznym na listingu 5.2. W konstruktorze inicjalizowana jest tylko wartość zmiennych więc nie będziemy się nad nim rozwodzić. Przeanalizujmy więc metodę `initializeGL(...)`, która konfiguruje nam wstępnie maszynę stanów *OpenGL* poprzez wywołanie z odpowiednimi parametrami funkcji odpowiadających za ustawienie właściwości wykorzystanych materiałów i sposobu cieniowania (linie 24-26), pozycji i rodzaju świateł (27-30) jak i włączenia obsługi określonych mechanizmów z użyciem `glEnable(...)` widoczne w liniach 31 do 34.

Ważne rzeczy rozgrywają się też w metodzie `resizeGL(...)`, w której po wejściu w tryb ustawiania macierzy projekcji następuje zdefiniowanie parametrów płaszczyzny projekcji (od linii 45). Następnie ustawione zostaje rzutowanie ortogonalne wraz z odpowiednią kontrolą proporcji obrazu w zależności od rozmiaru widgetu. Całość zamyka wywołanie `updateGL(...)` dzięki czemu zmiany zostają uwzględnione w kontekście renderowania i scena zostaje przerysowana z nowymi ustawieniami maszyny stanów.

Metoda `paintGL(...)`, która jest wywoływana zawsze gdy scena ulega zmianie i wymusza ponowne przekazanie jej opisu do kontekstu renderowania zawiera w naszym przypadku, poza wywołaniem funkcji czyszczących bufory (linia 59), utworzenie obiektu w postaci kuli będącej kwadryką występującą w bibliotece *GLU* standardowo uwzględnianej w Qt gdy korzystamy z obsługi *OpenGL*.

Listing 5.2. Plik definicji klasy `NewGLWidget`.

```
1  #include <QtGui>
2  #include <QtOpenGL>
3
4  #include "glwidget.h"
5
6  NewGLWidget::NewGLWidget(QWidget *parent)
7      : QGLWidget(parent)
8  {
9      rotX = rotY = rotZ = 0;
10 }
11
12 NewGLWidget::~NewGLWidget()
13 {
14 }
15
16 void NewGLWidget::initializeGL()
17 {
18     GLfloat mat_specular[] = { .4, .4, .4, 1.0 };
19     GLfloat mat_shininess[] = { 100.0 };
```

```
20      GLfloat light_position[] = { 0.0, 5.0, 10.0, 0.0 };
21      GLfloat white_light[] = { .60, .60, .60, 1.0 };
22      GLfloat lmodel_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
23
24      glShadeModel(GL_SMOOTH);
25      glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
26      glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
27      glLightfv(GL_LIGHT0, GL_POSITION, light_position);
28      glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);
29      glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);
30      glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
31      glEnable(GL_LIGHTING);
32      glEnable(GL_LIGHT0);
33      glEnable(GL_DEPTH_TEST);
34      glEnable(GL_NORMALIZE);
35      glClearColor(1,1,1,1);
36  }
37
38  void NewGLWidget::resizeGL(int w, int h)
39  {
40      if(w==0)
41              w=1;
42      if(h==0)
43              h=1;
44      glMatrixMode(GL_PROJECTION);
45      glLoadIdentity();
46      glViewport(0, 0, w, h);
47      if(w<h)
48              glOrtho(-1.1f,1.1f,-1.1f*h/w,1.1f*h/w,10.0f,-10.0f);
49      else
50              glOrtho(-1.1f*w/h,1.1f*w/h,-1.1f,1.1f,10.0f,-10.0f);
51      updateGL();
52  }
53
54  void NewGLWidget::paintGL()
55  {
56      glMatrixMode(GL_MODELVIEW);
57      glLoadIdentity();
58      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
59
60      glColor3f(1,0,0);
61      GLUquadricObj *kwadryka;
62      kwadryka = gluNewQuadric();
63      gluQuadricOrientation(kwadryka, GLU_INSIDE);
64      gluSphere(kwadryka,1,64,64);
65      gluDeleteQuadric(kwadryka);
66  }
67
68  void NewGLWidget::setShading(bool mode) {
69      if(mode)
70          glShadeModel(GL_FLAT);
```

```
71      else
72          glShadeModel(GL_SMOOTH);
73      updateGL();
74  }
75
76  void NewGLWidget::setLighting(bool mode) {
77      if(mode)
78          glEnable(GL_LIGHTING);
79      else
80          glDisable(GL_LIGHTING);
81      updateGL();
82  }
```

Sloty `setShading(...)` i `setLighting(...)` zmieniają stan maszyny *OpenGL* odpowiednio wyłączając/włączając obsługę oświetlenia i zmieniając sposób domyślnego cieniowania obiektów. Obsługę myszy odkładamy na później.

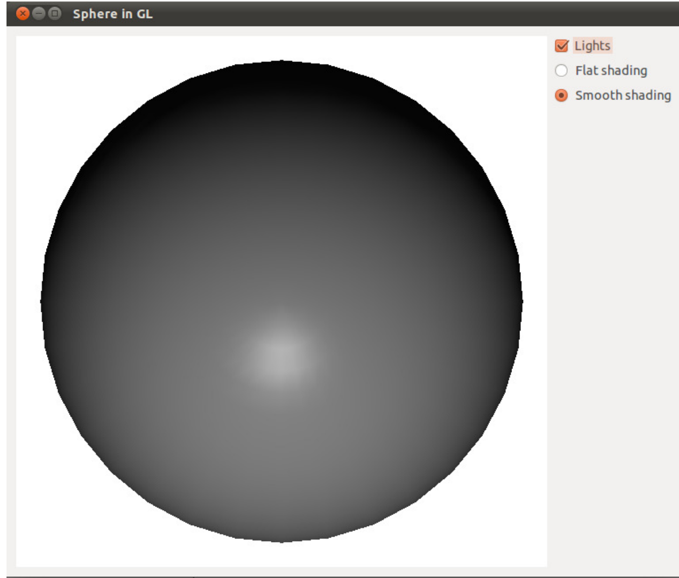
W tej sytuacji pozostaje nam użycie właśnie stworzonej klasy w ramach programu. W tym celu tworzymy okno wykorzystując kod widoczny na listingu 5.3. Składa się na nie widget obiekt naszej klasy `newglwidget`, zamknięty w układzie poziomym `mainLayout` oraz widgety w postaci checkboxu i przycisków typu radio zamknięte w układzie pionowym `menuLayout`. Ponieważ poszczególne opcje cieniowania muszą być włączane wykluczająco zostały związane ze sobą za pomocą klasy `QButtonGroup`. Całość menu została wyrównana w pionie do góry obszaru za pomocą metody `addStretch(...)`.

Listing 5.3. Zawartość okna programu.

```
1   class Window : public QWidget
2   {
3       Q_OBJECT
4
5   public:
6       Window();
7
8   protected:
9
10  private:
11      NewGLWidget *glWidget;
12      QCheckBox *lighting;
13      QRadioButton *flat;
14      QRadioButton *smooth;
15  };
16
17  Window::Window()
18  {
19      glWidget = new NewGLWidget;
20      lighting = new QCheckBox("Lights");
21      lighting->setChecked(true);
```

```
22      flat = new QRadioButton("Flat shading");
23      smooth = new QRadioButton("Smooth shading");
24      smooth->setChecked(true);
25
26      QHBoxLayout *mainLayout = new QHBoxLayout;
27      QVBoxLayout *menuLayout = new QVBoxLayout;
28
29      mainLayout->addWidget(glWidget);
30
31      QButtonGroup *shading = new QButtonGroup;
32      shading->addButton(flat);
33      shading->addButton(smooth);
34
35      menuLayout->addWidget(lighting);
36      menuLayout->addWidget(flat);
37      menuLayout->addWidget(smooth);
38      menuLayout->addStretch();
39      mainLayout->addLayout(menuLayout);
40
41      setLayout(mainLayout);
42
43      connect(lighting, SIGNAL(toggled(bool)),
44        glWidget, SLOT(setLighting(bool)));
45      connect(flat, SIGNAL(toggled(bool)),
46        glWidget, SLOT(setShading(bool)));
47
48      setWindowTitle(tr("Sphere in GL"));
49  }
```

W liniach od 43 do 46 mamy stworzone połączenie pomiędzy sygnałem `toggled` i odpowiednimi slotami zdefiniowanymi w naszej klasie. Wynik działania przedstawia ilustracja 5.1.



Rysunek 5.1. Program wykorzystujący widget klasy `NewGLWidget`.

Rozdział 6

Operacje wejścia/wyjścia

6.1. Wstęp **88**
6.2. Styl C **88**
6.3. Styl C++ **91**
6.4. Dialog **94**
6.5. Obsługa systemu plików **96**

6.1. Wstęp

Nie da się ukryć, że dla poprawnego funkcjonowania większości aplikacji konieczne jest zrealizowanie mechanizmu dostępu do urządzeń zewnętrznych. W najprostszym przypadku będzie to komunikacja z lokalnymi nośnikami danych oraz odczyt i zapis na nich. Najbardziej banalnym przykładem może być proste zachowanie konfiguracji. Dużo jednak istotniejszy jest fakt, że oczekujemy możliwości łatwego dostępu do danych, które w aplikacji będą przetwarzane, a wyniki naszej pracy będą zachowane dla potomności.

Wprawdzie w C++ jest możliwość operowania na plikach z wykorzystaniem mechanizmów dostępnych za pośrednictwem `iostream`, jednak oparcie się na rozwiązaniach alternatywnych oferowanych przez bibliotekę Qt w sytuacji gdy nasza aplikacja z niej korzysta niesie ze sobą szereg korzyści na czele z niezależnością od sposobu reprezentacji ścieżek i nazw plików w różnych systemach operacyjnych.

Podstawą wszelkich działań na plikach jest klasa `QFile`. Możemy korzystać z niej tworząc kod przypominający mechanizmy znane z języka C (`stdio.h`), oczywiście z uwzględnieniem obiektowości Qt, lub pracując z użyciem strumieni na sposób znany z C++. Ta druga metoda jest bardziej skomplikowana, ale jednocześnie jest bardziej elastyczna oraz spójna z paradygmatami programowania obiektowego. W następnych sekcjach zaprezentowane zostaną oba podejścia.

6.2. Styl C

Pierwsza z metod korzystania z plików na dysku w założeniu jest łatwa do opanowania przez osoby, które do tej pory programowały głównie w C lub używały w C++ bibliotek obsługi wejścia i wyjścia z C wywodzących się. Przyjrzyjmy się następującemu kodowi:

```
QFile plik("config.txt");
QByteArray konfiguracja[3];
int i=0;

if (!plik.open(QIODevice::ReadOnly | QIODevice::Text))
  return -1;

while (!plik.atEnd() && i<3)
{
  konfiguracja[i++] = plik.readLine();
}

plik.close();
```

Jego zadaniem jest wczytanie do tablicy trzech kolejnych linii pliku **config.txt**, zawierającego konfigurację. Oczywiście w rzeczywistości dużo lepszym rozwiązaniem byłoby przechowywanie tej informacji w systemie przykładowo w postaci listy, ale ze względu na prostotę skorzystamy z metody suboptymalnej.

W pierwszej kolejności jest tworzony obiekt klasy `QFile` gdzie w ramach konstruktora podana zostaje ścieżka, względna (jak ma to miejsce w tym przypadku) lub bezwzględna oraz nazwa pliku. Nazwę można też zmienić później przekazując ją do obiektu za pośrednictwem metody `setFileName(...)`.

Następnie ma miejsce otworzenie dostępu do pliku za pośrednictwem przeciążonej metody `open(...)`, która w najprostszym przypadku przyjmuje tylko argument opisujący tryb dostępu do pliku o wartościach zdefiniowanych w przestrzeni nazw `QIODevice`. Trzy podstawowe to `QIODevice::ReadOnly` - tylko odczyt, `QIODevice::WriteOnly` - tylko zapis i `QIODevice::ReadWrite`. Dodatkowo gdy chcemy dopisywać dane na końcu pliku możliwe jest skorzystanie z `QIODevice::Append`. Zaś `QIODevice::Truncate` powoduje nadpisanie istniejącego pliku w momencie otwarcia.

Specjalne, i nie posiadające odpowiednika w `stdio`, jest użycie trybów `QIODevice::Text` oraz `QIODevice::Unbuffered`. Ten pierwszy powoduje, że podczas odczytu pliku z dysku znaczniki końca linii tekstu zamieniane są do postaci `\n`. Drugi zaś wymusza pominięcie bufora urządzenia przy zapisie (o ile taki występuje). Oba łączone są z sześcioma wcześniej wymienionymi poprzerz operator alternatywny.

Po otworzeniu pliku tekstowego, jak ma to miejsce w przykładzie, wchodzimy do pętli, której działanie zakończy się (w zależności od tego co nastąpi pierwsze) gdy dotrzemy do końca pliku, co możemy sprawdzić wywołując metodę `atEnd(...)`, lub wczytamy trzy linie konfiguracji. Odczyt pojedynczej linii zwracanej w postaci `QByteArray` odbywa się z użyciem metody `readLine(...)`, która wraz z zakończeniem odczytu przestawia kursor pliku na następną linię. Jej wywołanie może dodatkowo zawierać parametr określający maksymalną długość odczytu w bajtach, ale domyślnie jest to cała linia niezależnie od jej długości. Niestety nie mamy w tym wypadku kontroli błędów, a zwrócenie pustej linii może świadczyć zarówno o tym, że taka zapisana była w pliku, jak i o błędzie odczytu. Alternatywnie możemy skorzystać z przeciążonego wariantu tej metody, która zwraca długość odczytanego łańcucha lub -1 w przypadku niepowodzenia, ale niestety konieczne jest wtedy korzystanie z bufora odczytu w postaci tablicy typu `char`.

W przypadku dostępu tekstowego możliwe jest również skorzystanie z odczytu pojedynczych znaków oferowane przez metodę `getChar`. Jej jedyny parametr wywołania to wskaźnik do zmiennej typu `char`, w której zostanie

zapisany wynik odczytu z dysku. Wartość zwracana mówi o sukcesie lub porażce operacji.

Gdy cały odczyt zostanie zakończony wywoływana jest metoda zamknięcia dostępu - `close(...)`. Jej znaczenie w tym wypadku jest mniejsze niż miałoby to miejsce przy zapisie. Nawet gdybyśmy zapomnieli ją umieścić w kodzi programu nie spowoduje to utraty informacji zawartej w pliku.

Dużo bardziej rozbudowana jest operacja odczytu danych w postaci binarnej.

```
QFile plikWejsciowy("20120110.dat");
QFile plikWyjsciowy("statystyki_dzienne.dat");

float *pomiary;
float suma = 0.0, srednia;
int liczbaPomiarow;

if (!plikWejsciowy.open(QIODevice::ReadOnly))
  return -1;

liczbaPomiarow = plikWejsciowy.size() / sizeof(float);
pomiary = new float[liczbaPomiarow];

plikWejsciowy.read
  ((char *) pomiary, liczbaPomiarow * sizeof(float));
plikWejsciowy.close();

for(int i=0; i<liczbaPomiarow; i++)
{
  suma += pomiary[i];
}

srednia = suma / liczbaPomiarow;

if (!plikWyjsciowy.open
  (QIODevice::WriteOnly | QIODevice::Append))
  return -1;

plikWyjsciowy.write((char *) &srednia, sizeof (float));
plikWyjsciowy.close();
```

W powyższym przykładzie będziemy operowali na dwóch plikach. Pierwszy zawiera informację wejściową w postaci kolejnych wartości typu **`float`**, które mogą pochodzić z odczytu z czujnika w aparaturze pomiarowej dla danego dnia. Drugi jest zbiorem wartości wyjściowych średniej wartości dla danych wyjściowych dla kolejnych dni.

Po otworzeniu pliku wejściowego sprawdzana jest jego wielkość za pomocą metody `size(...)` a następnie w oparciu o nią dokonywane jest wyliczenie liczby odczytów w nim zawartych. Z wykorzystaniem tej informacji alokowa-

na jest tablica przechowująca pomiary. Do niej do niej będą zapisane dane odczytane z użyciem metody `read(...)`. Posiada ona dwa argumenty - wskazanie do miejsca pamięci, do którego odbywał się będzie odczyt i wielkość odczytu wyrażona w bajtach.

Po wyliczeniu średniej otwierany jest plik wyjściowy do zapisu. Jeżeli taki plik nie istnieje to zostanie utworzony pusty. Dodatkowo użyta w wywołaniu stała `QIODevice::Append` spowoduje automatyczne ustawienie kursora pliku na jego koniec. Wartość średnia zostanie dopisana z wykorzystaniem `write(...)`, które przyjmuje analogiczną do `read(...)` listę parametrów. Warto jeszcze wspomnieć o tym, że do odczytu możemy skorzystać z metody `peek(...)`, której użycie jest tożsame z `read(...)` za wyjątkiem braku zmiany położenia kursora pliku.

Ważne jest zakończenie całej operacji poprzez zamknięcie dostępu do pliku. W przypadku zapisu spowoduje to uprzednie wywołanie metody `flush(...)`, która opróżniena bufor urządzenia co daje nam pewność, że dane znajdą się tam gdzie powinny.

Dodatkowo obsługa kursora pozycji w pliku realizowana jest z użyciem szeregu metod. Obecne położenie możliwe jest do sprawdzenia z wykorzystaniem `pos(...)`, a jego zmiana jest dokonywana w oparciu o `seek(...)`. Powrót do pozycji początkowej można wymusić poprzez wywołanie `reset(...)`. Metody te nie są dostępne dla urządzeń, które nie są blokowe lecz sekwencyjne jednak ich obsługa wykracza poza zakres tego skryptu.

6.3. Styl C++

Jedną z ważnych nowości, które wynikły z wprowadzenia języka C++ w porównaniu do bazy, z której został wyprowadzony, obok rozszerzenia C o paradygmat obiektowy, było zaproponowanie rozbudowanej obsługi strumieni oraz odpowiednich operatorów znaczenie tę obsługę ułatwiający.

W Qt do dyspozycji programistów standardowo dano dwa typy obsługiwanych strumieni. Tekstowy w postaci klasy `QTextStream` i binarny `QDataStream`. Zakres ich działania jest szerszy niż ma to miejsce w ramach `iostream` znanego z C++.

Spójrzmy na następujący przykład:

```
QFile plik("config.txt");
QString konfiguracja[3];
int i=0;

if (!plik.open(QIODevice::ReadOnly | QIODevice::Text))
  return -1;

QTextStream strumienZPliku(&plik);
```

```
while (!strumienZPliku.atEnd() && i<3)
{
    konfiguracja[i++] = strumienZPliku.readLine();
}

plik.close();
```

Jego działanie jest analogiczne do pierwszego z przedstawionych w niniejszym rozdziale. Pomijając użycie elementu pośredniego w postaci obiektu strumienia `strumienZPliku` i zapisu poszczególnych linii konfiguracji jako `QString` jest prawie identyczny. Okazuje się jednak, że pod pozorami podobieństwa ukryte są zasadnicze różnice. W prezentowanym przykładzie znaki odczytane z pliku przechowywane są zakodowane w postaci **Unicode**, a nie pojedynczych bajtów. Jeżeli plik, z którego miał miejsce odczyt jest zachowany w kodowaniu ośmiobitowym to znaki są domyślnie konwertowane do szesnastobitowej reprezentacji. Za pomocą metody `setCodec(...)` możliwe jest zdefiniowanie kodowania pliku źródłowego w sytuacji gdy jest ona odmienna od domyślnie ustawionej w systemie.

Możliwe jest też odczytywanie i zapis danych do i z plików tekstowych z użyciem operatorów strumienia << i >> dla wszystkich typów standardowych zdefiniowanych w ramach tak C++ jak i samego Qt. Operacja taka odbywa się po pojedynczym słowie. Jeżeli założymy, że każda linia nasze konfiguracji składa się pary atrybut i jego wartość to odpowiednio zmodyfikowany kod będzie miał postać:

```
QFile plik("config.txt");
QString konfiguracja[3][2];
int i=0, j;

if (!plik.open(QIODevice::ReadOnly | QIODevice::Text))
  return -1;

QTextStream strumienZPliku(&plik);

while (!strumienZPliku.atEnd() && i<3)
{
  j=0;
  strumienZPliku >> konfiguracja[i][j++];
  strumienZPliku >> konfiguracja[i++][j];
}

plik.close();
```

Ważną częścią klasy `QTextStream` są też metody definiowania długości pola, w którym będą zapisywane kolejne słowa, oraz definiowania notacji liczb.

W pierwszym wypadku korzystamy z `setFieldWidth(...)`, które przyjmuje jeden parametr w wywołaniu mówiący jak długie jest pole dla kolejnych słów umieszczonych w strumieniu. Przekazanie do tej metody wartości 0 powoduje, że wartość będzie dobierana automatycznie w zależności od długości słowa.

Jeżeli teraz wykonamy poniższy kod to łańcuch '0123' przesłany do zdefiniowanego strumienia tekstowego zostanie sformatowany w sposób przedstawiony w tabeli 6.1.

```
strumienDoPliku.setFieldWidth(16);
strumienDoPliku.setPadChar('-');
```

Tabela 6.1. Parametry metody `setFieldAlignment()`

Enumerator	Wynik działania
`QTextStream::AlignLeft`	`0123------------`
`QTextStream::AlignCenter`	`------0123------`
`QTextStream::AlignRight`	`------------0123`

Metoda `setIntegerBase(...)` pozwala na ustawienie podstawy w reprezentacji liczb całkowitych. Domyślnie jest ona dziesiętna, jednak jest możliwe zmienienie jej na binarną, ósemkową i szesnastkową. Metoda ta dotyczy zarówno zapisu jak i odczytu strumienia z urządzenia.

Sposób reprezentacji liczby zmiennoprzecinkowych możliwy jest z użyciem `setRealNumberNotation(...)`, za pomocą którego możemy określić używaną notację, i `setRealNumberPrecision(...)` definiującą precyzję. Praktyczne wykorzystanie ich prezentuje następujący następujący przykład:

```
QFile formaty("formaty.txt");
QTextStream strumienDoPliku(&formaty);

if (!formaty.open(QIODevice::WriteOnly | QIODevice::Text))
  return -1;

strumienDoPliku << "Podstawa dziesietna:   " << 27 << endl;
strumienDoPliku.setIntegerBase(2);
strumienDoPliku << "Podstawa dwojkowa:     " << 27 << endl;
strumienDoPliku.setIntegerBase(8);
strumienDoPliku << "Podstawa osemkowa:     " << 27 << endl;
strumienDoPliku.setIntegerBase(16);
strumienDoPliku << "Podstawa szesnastkowa: " << 27 << endl;

strumienDoPliku << "----------------" << endl;

strumienDoPliku << 0.123456789 << endl;
strumienDoPliku.setRealNumberPrecision(2);
```

```
strumienDoPliku << 0.123456789 << endl;
strumienDoPliku.setRealNumberPrecision(6);

strumienDoPliku << "----------------" << endl;

strumienDoPliku << 0.123456789 << endl;
strumienDoPliku.setRealNumberNotation
  (QTextStream::ScientificNotation);
strumienDoPliku << 0.123456789 << endl;

formaty.close();
```

W wyniku działania powyższego programu otrzymamy plik **formaty.txt**, którego zawartość przedstawia się tak:

```
Podstawa dziesietna:   27
Podstawa dwojkowa:     11011
Podstawa osemkowa:     33
Podstawa szesnastkowa: 1b
----------------
0.123457
0.12
----------------
0.123457
1.234568e-01
```

W przypadku dostępu do plików binarnych korzystamy z klasy `QDataStream` i operatorów strumieniowych. Dodatkowo dostępne są metody, które umożliwiają nam odczyt i zapis bloków danych oraz operację zmiany kolejności bajtowej dla danych znajdujących się w ramach strumienia. Ta druga funkcjonalność jest dostępna z użyciem `setByteOrder` wywołanej z parametrem `QDataStream::BigEndian` lub `QDataStream::LittleEndian`.

6.4. Dialog

W ramach modułu **QtGui** biblioteka Qt oferuje standardowy dialog służący do wyboru położenia pliku oraz katalogu tak do odczytu jak i zapisu. Dialog realizowany jest przez obiekt klasy `QFileDialog`, która dodatkowo zawiera szereg parametrów i metod umożliwiających konfigurację sposobu komunikacji z użytkownikiem.

Wywołanie dialogu jest możliwe na dwa sposoby. Pierwszy wykorzystuje statyczne wywołanie funkcji `getOpenFileName(..)` wraz z odpowiednimi parametrami. Takie podejście ma mniejszą możliwość kształtowania wyglądu i zachowania okna dialogowego jednak w większości przypadków jest ono w

pełni zadowalające. Druga droga korzysta z utworzonego obiektu i metod klasy `QFileDialog`. Poniższy kod przedstawia to rozwiązanie:

```
QFileDialog dialogPlikowy(this, tr("Plik"));
dialogPlikowy.setFileMode(QFileDialog::AnyFile);

QStringList filtry;
filtry << tr("Obrazy (*.jpg *.png *.gif)")
  << tr("Pliki tekstowe (*.txt)")
  << tr("Wszystkie pliki (*)");

dialogPlikowy.setNameFilters(filtry);

dialogPlikowy.setViewMode(QFileDialog::Detail);

QStringList listaPlikow;

if (dialogPlikowy.exec())
  listaPlikow = dialogPlikowy.selectedFiles();
```

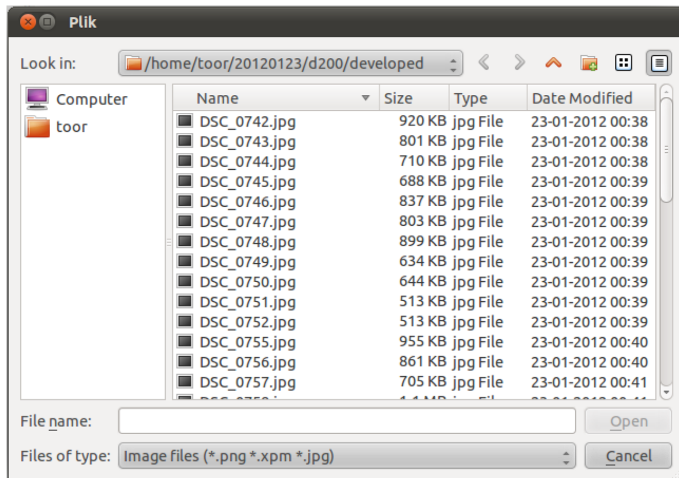
W ramach konfiguracji ustalamy tryb selekcji plików w ramach dialogu za pomocą metody `setFileMode(...)`. Możliwe są cztery opcje przekazywane do niej jako parametr:

1) `QFileDialog::AnyFile` – pojedynczy plik niezależnie od tego czy istnieje
2) `QFileDialog::ExistingFile` – pojedynczy istniejący plik
3) `QFileDialog::ExistingFiles` – dowlona, w tym zerowa, liczba istniejących plików
4) `QFileDialog::Directory` – nazwa katalogu

Następnie tworzymy listę łańcuchów `QStringList` i wypełniamy regułami filtrowania listy plików. Każda z nich zawiera nazwę dla każdego filtra oraz maski wyświetlanych plików zamknięte w nawiasie okrągłym. W gotowym dialogu będzie możliwość wybrania ich z listy rozwijalnej. Filtry przekazywane są w wywołaniu metody `setNameFilters(...)`.

Metoda `setViewMode(...)` pozwala na ustalenie czy pliki mają być reprezentowane w formie szczegółowej - `QFileDialog::Detail` z podaną datą modyfikacji i wielkością, czy też uproszczonej - `QFileDialog::List`. Okno wywołuje metoda `exec()`, która jeżeli wszystko odbyło się prawidłowo zwraca `TRUE`. W takiej sytuacji możemy pobrać nazwy wybranych plików lub katalogów w postaci listy łańcuchów za pomocą metody `selectedFiles(...)`.

Wynik działania prezentowanego kodu widoczny jest na ilustracji 6.1.



Rysunek 6.1. Wywołane okienko dialogowe widgetu `QFileDialog`

6.5. Obsługa systemu plików

Biblioteka Qt oferuje również szerszą możliwość pracy z system plików. Z pomocą idzie tu poznana już klasa `QFile` oraz całkiem nowe dla naszych rozważań `QFileInfo` i `QDir` pozwalają na kopiowanie, zmianę nazwy, pobieranie szczegółowych informacji tak o plikach jak i katalogach.

Mając utworzony obiekt klasy `QFile` możliwe jest wykonanie z pomocą metody `rename(...)` zmiany nazwy wskazanego pliku, zaś z użyciem `remove(...)` usunięcie go z dysku, oczywiście pod warunkiem posiadania odpowiednich uprawnień. Sprawdzenie praw dostępu odbywa się w oparciu o metodę `permissions(...)` zwracającą połączone operatorem logicznym "lub" stałe opisujące poszczególne prawa z uwzględnieniem charakterystyki danego systemu. Tyczą się one kolejno właściciela, bieżącego użytkownika, grupy oraz pozostałych. Dla każdego z nich opisują prawo odczytu, zapisu i wykonania pliku.

Szerszą wiedzę na temat pliku możemy pozyskać korzystając z metod zawartych w `QFileInfo`. Przyjrzyjmy się następującemu kodowi:

```
#include <QtCore/QCoreApplication>
#include <QFileInfo>
#include <QDebug>
#include <QDateTime>

int main(int argc, char *argv[])
{
    QFileInfo informator("/home/toor/qt/skrypt_qt/skrypt.pdf");
    qDebug() << "Sciezka:"
```

```
    << informator.absolutePath();
  qDebug() << "Link symboliczny:" << informator.isSymLink();
  qDebug() << "Sciezka oryginalna dla linku:"
    << informator.symLinkTarget();
  qDebug() << "Nazwa:" << informator.baseName();
  qDebug() << "Rozszerzenie:" << informator.suffix();
  qDebug() << "Wlasciciel:" << informator.owner();
  qDebug() << "Grupa:" << informator.group();
  qDebug() << "Rozmiar:" << informator.size() << "B";
  qDebug() << "Data utworzenia:"
    << informator.created().toString("dd-MM-yyyy");
  qDebug() << "Data ostatniej zmiany:"
    << informator.lastModified().toString("dd-MM-yyyy");
  qDebug() << "Data ostatniego odczytu:"
    << informator.lastModified().toString("dd-MM-yyyy");

  return 0;
}
```

Wynik działania przykładu przedstawiono poniżej. Widać na nim, że stosunkowo łatwo możemy pozyskać szeroką wiedzę o wybranym pliku od informacji o jego rozmiarze, właścicielu, ścieżkach, nazwie aż po dane na temat daty i czasu utworzenia, zmiany czy ostatniego dostępu. Oczywiście część z tych informacji jest zależna od możliwości systemu operacyjnego, w którym uruchamiana jest aplikacja nie mniej jednak widoczna jest duża uniwersalność tego rozwiązania.

```
Sciezka: "/home/toor/qt/skrypt_qt"
Link symboliczny: false
Sciezka oryginalna dla linku: ""
Nazwa: "skrypt"
Rozszerzenie: "pdf"
Wlasciciel: "toor"
Grupa: "toor"
Rozmiar: 2608146 B
Data utworzenia: "20-02-2012"
Data ostatniej zmiany: "20-02-2012"
Data ostatniego odczytu: "20-02-2012"
```

Ostatnia omawiana w niniejszym rozdziale klasa, `QDir`, oferuje nam interfejs umożliwiający działanie na strukturze katalogów w systemie plików. Poniższy program przedstawia użycie części z metod oferowanych przez klasę.

```
#include <QtCore/QCoreApplication>
#include <QDir>
#include <QDebug>
#include <QFileInfoList>
```

```
int main(int argc, char *argv[])
{
    QDir dirInfo("../../qt/Qdir-build-desktop");
    qDebug() << "Sciezka: " << dirInfo.absolutePath();
    qDebug() << "Sciezka kanoniczna: "
      << dirInfo.canonicalPath();
    qDebug() << "Nazwa katalogu: " << dirInfo.dirName();
    qDebug() << "Czy podana sciezka jest bezwzgledna: "
      << dirInfo.isAbsolute();
    qDebug() << "Czy uzytkownik ma prawo odczytu z niej: "
      << dirInfo.isReadable();
    qDebug() << "Liczba elementow w katalogu: "
      << dirInfo.count();

    dirInfo.setFilter(QDir::Files);
    dirInfo.setSorting(QDir::Name | QDir::Reversed);

    qDebug() << "Lista plikow: nazwa / rozmiar";
    QFileInfoList listaPlikow = dirInfo.entryInfoList();
    for (int i = 0; i < listaPlikow.size(); ++i) {
            QFileInfo iPlik = listaPlikow.at(i);
            qDebug() << iPlik.baseName() << " / "
             << iPlik.size() << "B";
    }
}
```

Po utworzeniu obiektu klasy `QDir`, w konstruktorze którego wskazaliśmy interesujący nas katalog mamy możliwość sprawdzić czy użytkownik ma prawo odczytu danych w nim zawartych za pomocą metody `isReadable(...)`. Informacje o względnej i kanonicznej ścieżce dla katalogu dostępne są za pomocą `relativePath(...)` i `canonicalPath(...)`. Sprawdzenie liczby elementów zawartych w danym katalogu odbywa się poprzez wywołanie metody `count(...)`, a pobranie szczegółów tyczących się każdego z nich wykorzystuje `entryInfoList(...)` zwracające listę elementów poznanego wcześniej typu `QFile`. Możliwe jest przefiltrowanie i posortowanie zawartości dzięki metodom `setFilter` i `setSorting`.

W wyniku otrzymamy następujące dane:

```
Sciezka:  "/home/toor/qt/Qdir-build-desktop"
Sciezka kanoniczna:  "/home/toor/qt/Qdir-build-desktop"
Nazwa katalogu:  "Qdir-build-desktop"
Czy podana sciezka jest bezwzgledna:  false
Czy uzytkownik ma prawo odczytu z niej:  true
Liczba elementow w katalogu:  5
Lista plikow: nazwa / rozmiar
"main"  /  194336 B
"Qdir"  /  158452 B
"Makefile"  /  6790 B
```

Rozdział 7

Operacje sieciowe

7.1. Wstęp **100**
7.2. Klient FTP **100**
7.3. Klient HTTP **105**

7.1. Wstęp

Qt4 jest wyposażone w moduł sieciowy, oferujący wiele różnorodnych możliwości, od podstawowych po zaawansowane, jak np. internacjonalizacja nazw domen, czy wsparcie dla IPv6. Klasy `QTcpSocket` i `QUdpSocket` implementują dwa podstawowe protokoły warstwy transportowej. Pozwalają one zarówno na odbieranie, jak i wysyłanie datagramów. `QTcpServer` umożliwia nasłuchiwanie na zadanym porcie serwera i emituje sygnał `newConnection()`, w razie próby połączenia ze strony klienta. Dalsza komunikacja jest następnie realizowana przy pomocy `QTcpSocket`.

Dostępne są także klasy realizujące operacje na wyższym poziomie. Klasa `QFtp` jest implementacją klienta FTP. Implementację klienta FTP oraz HTTP zapewnia także nowsza klasa – `QNetworkAccessManager`. Z kolei klasa `QHttp`, której istnienia można się domyślać, jest już przestarzała. W Qt 4.7 jest dostępna dla zachowania zgodności ze starszymi programami, jednak korzystanie z niej jest nierekomendowane.

Aby swobodnie korzystać z klas modułu sieciowego Qt, można użyć dyrektywy:

```
#include <QtNetwork>
```

natomiast do pliku .pro dla qmake należy dodać wpis:

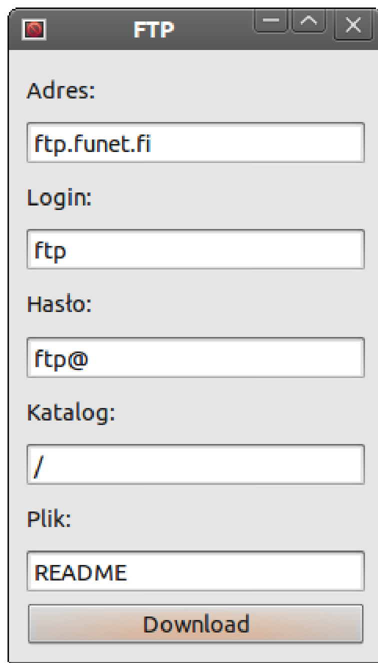
```
QT += network
```

7.2. Klient FTP

W bieżącym podrozdziale przedstawiony zostanie sposób stworzenia klienta FTP, z wykorzystaniem klasy `QFtp`. Poszczególne funkcje składowe klasy umożliwiają wykonywanie typowych operacji FTP:
- `connectToHost()` – łączenie z serwerem,
- `login()` – logowanie,
- `close()` – zamykanie połączenia,
- `list()` – listowanie zawartości katalogu,
- `cd()` – zmiana katalogu,
- `get()` – pobieranie plików,
- `put()` – wysyłanie plików,
- `remove()` – usuwanie plików z serwera,
- `mkdir()` – tworzenie katalogów na serwerze,
- `rmdir()` – usuwanie katalogów na serwerze,
- `rename()` – zmiana nazwy pliku na serwerze,

— `rawCommand()` – wysłanie dowolnego polecenia do serwera (ma zastosowania, gdy brakuje funkcji `QFtp` do wykonania określonej czynności).

Funkcje te są nieblokujące, tzn. tuż po wywołaniu program jest kontynuowany. Dzięki temu interfejs graficzny nie jest "zamrażany" na czas wykonywania poszczególnych operacji. Ich status można natomiast śledzić dzięki wysyłanym sygnałom.



Rysunek 7.1. Okno klienta FTP

Rys. 7.1 przedstawia okno bardzo prostego klienta FTP. Po kliknięciu przycisku "Download", ze wskazanego miejsca na wybranym serwerze FTP pobierany jest plik o podanej nazwie. Musimy zatem znać lokalizację pliku, który ma zostać pobrany. Jest on następnie zapisywany w bieżącym katalogu, w pliku o nazwie `ftp.out`.

Deklarację klasy reprezentującej okno klienta FTP przedstawia listing 7.1, a definicje funkcji składowych – 7.2. Za pobranie pliku z serwera odpowiedzialna jest funkcja `FTPWindow::downloadFile()`. Jest w niej tworzony obiekt klasy `QFtp`, a następnie przy jego pomocy wykonywane są odpowiednie operacje FTP (sekwencja poleceń w wierszach 59–63 listingu 7.2). Poszczególnym operacjom automatycznie nadawane są kolejne identyfikatory. Rozpoczęciu wykonywania danej operacji towarzyszy wysłanie sygnału `commandStarted(int)`, przekazującego wartość identyfikatora. O zakończeniu wykonywania operacji informuje z kolei sygnał `commandFinished(int,bool)`.

Pierwszym parametrem, podobnie jak poprzednio, jest identyfikator operacji. Drugi parametr przyjmuje wartość *true* w przypadku wystąpienia błędu, a w przeciwnym razie *false*. W prezentowanym programie sygnały te zostały połączone ze slotami `infoStart(int)` i `infoStop(int,bool)`. Wyświetlają one na konsoli (by dodatkowo nie komplikować przykładowego programu) informacje o przebiegu sesji FTP. Jeżeli wykonywanie jednej z operacji w sekwencji zakończy się z błędem, kolejne operacje są porzucane (nie ma sensu próba pobierania pliku, jeżeli np. nie udało się zalogować). Po wykonaniu całej sekwencji instrukcji, emitowany jest sygnał `done(bool)` z parametrem o wartości *false*, jeżeli wszystkie operacje wykonano bezbłędnie, lub *true*, w przeciwnym razie. W przykładowym programie sygnał ten jest połączony ze slotem `infoSeq(bool)`. Oprócz wyświetlania informacji na konsoli, jego zadaniem jest ponowne uaktywnienie przycisku “Download”, który jest blokowany na czas trwania sesji FTP.

Listing 7.1. Deklaracja klasy klienta FTP

```
1  #ifndef FTPWINDOW_H
2  #define FTPWINDOW_H
3
4  #include <QtGui>
5
6  class FTPWindow : public QWidget
7  {
8      Q_OBJECT
9  public:
10     FTPWindow(QWidget *parent = 0);
11
12 private:
13     QLineEdit* addrEdit;
14     QLineEdit* loginEdit;
15     QLineEdit* passEdit;
16     QLineEdit* dirEdit;
17     QLineEdit* filenameEdit;
18     QPushButton* goButton;
19     QFile* file;
20
21 signals:
22
23 public slots:
24
25 private slots:
26     void downloadFile();
27     void infoStart(int);
28     void infoStop(int,bool);
29     void infoSeq(bool);
30 };
31 #endif  //  FTPWINDOW_H
```

Listing 7.2. Definicja klasy klienta FTP

```
1  #include <QFtp>
2  #include <QFile>
3  #include <iostream>
4  #include "ftpwindow.h"
5
6  using namespace std;
7
8  FTPWindow::FTPWindow(QWidget *parent) :
9      QWidget(parent) {
10
11     setWindowTitle("FTP");
12
13     QLabel* addrLabel = new QLabel("Adres:");
14     addrEdit = new QLineEdit("ftp.funet.fi");
15     QLabel* loginLabel = new QLabel("Login:");
16     loginEdit = new QLineEdit("ftp");
17     QLabel* passLabel = new QLabel("Haslo:");
18     passEdit = new QLineEdit("ftp@");
19     QLabel* dirLabel = new QLabel("Katalog:");
20     dirEdit = new QLineEdit("/");
21     QLabel* filenameLabel = new QLabel("Plik:");
22     filenameEdit = new QLineEdit("README");
23
24     goButton = new QPushButton("Download");
25
26     QVBoxLayout* layout = new QVBoxLayout();
27     layout->addWidget(addrLabel);
28     layout->addWidget(addrEdit);
29     layout->addWidget(loginLabel);
30     layout->addWidget(loginEdit);
31     layout->addWidget(passLabel);
32     layout->addWidget(passEdit);
33     layout->addWidget(dirLabel);
34     layout->addWidget(dirEdit);
35     layout->addWidget(filenameLabel);
36     layout->addWidget(filenameEdit);
37     layout->addWidget(goButton);
38     setLayout(layout);
39
40     connect(goButton, SIGNAL(clicked()),
41             this, SLOT(downloadFile()));
42 }
43
44 void FTPWindow::downloadFile() {
45
46     goButton->setEnabled(FALSE);
47
48     QFtp *ftp = new QFtp();
49     connect(ftp, SIGNAL(commandStarted(int)),
50             this, SLOT(infoStart(int)));
```

```
51     connect(ftp, SIGNAL(commandFinished(int,bool)),
52             this, SLOT(infoStop(int,bool)));
53     connect(ftp, SIGNAL(done(bool)),
54             this, SLOT(infoSeq(bool)));
55
56     file = new QFile("ftp.out");
57     file->open(QIODevice::WriteOnly);
58
59     ftp->connectToHost(addrEdit->text());                  // id1
60     ftp->login(loginEdit->text(),passEdit->text());        // id2
61     ftp->cd(dirEdit->text());                              // id3
62     ftp->get(filenameEdit->text(), file);                  // id4
63     ftp->close();                                          // id5
64 }
65
66 void FTPWindow::infoStart(int i) {
67    cout << "start " << i << endl;
68 }
69
70 void FTPWindow::infoStop(int i,bool error) {
71     cout << "stop " << i << (error?" !":" ok") << endl;
72 }
73
74 void FTPWindow::infoSeq(bool error) {
75    cout << "STOP " << (error?"!":"ok") << endl;
76    file->close();
77    goButton->setEnabled(TRUE);
78 }
```

Listing 7.3. Klient FTP – funkcja main

```
1 #include <QtGui>
2 #include "ftpwindow.h"
3
4 int main(int argc, char *argv[])
5 {
6     QApplication a(argc, argv);
7     FTPWindow w;
8
9     w.show();
10
11     return a.exec();
12 }
```

W razie pobierania pliku zakończonego sukcesem, na konsoli zostaną wyświetlone następujące komunikaty:

```
start 1
stop 1 ok
```

```
start 2
stop 2 ok
start 3
stop 3 ok
start 4
stop 4 ok
start 5
stop 5 ok
STOP ok
```

Liczby są identyfikatorami operacji z wierszy 59–63 listingu 7.2. Komunikat STOP ok informuje o poprawnym zakończeniu całej sekwencji (a więc o pobraniu pliku z serwera).

W razie gdy zostanie podana niepoprawna nazwa katalogu, zobaczymy komunikaty:

```
start 1
stop 1 ok
start 2
stop 2 ok
start 3
stop 3 !
STOP !
```

Operacja nr 3 (`ftp->cd()`) zakończyła się błędem, przerywając całą sekwencję.

W przedstawionym powyżej, przykładowym programie wykorzystano jedynie podstawowe możliwości klasy `QFtp`. Bardziej szczegółowych informacji o wykonywaniu poszczególnych informacji dostarcza sygnał `stateChanged()`. Z kolei sygnał `dataTransferProgress()` umożliwia wyświetlanie paska postępu podczas transmisji plików.

7.3. Klient HTTP

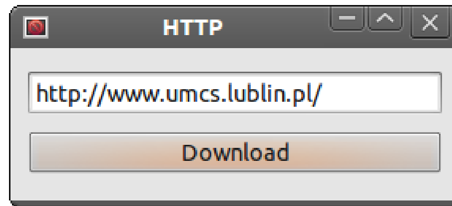
Najważniejszą klasą Qt, umożliwiającą wysyłanie zapytań do serwerów sieciowych i odbieranie odpowiedzi od nich, jest `QNetworkAccessManager`. Zazwyczaj jeden obiekt tej klasy jest wystarczający dla całej aplikacji, gdyż zapytania są kolejkowane i uruchamiane równolegle. Duża liczba opcji konfiguracyjnych pozwala realizować zarówno proste, jak też zaawansowane operacje. Ich przebieg może być monitorowany dzięki emitowanym sygnałom.

Odpowiedzi na zapytania wysyłane do sieci zwracane są w postaci obiektów klasy `QNetworkReply`. Umożliwiają one dostęp do otrzymanych danych i metadanych (np. nagłówków) . Dane te są przechowywane tylko do chwili ich odczytania. Skasowanie niepotrzebnego już obiektu `QNetworkReply` jest

zadaniem programisty, przy czym nie należy do tego celu używać słowa kluczowego **delete**, lecz funkcji `deleteLater()`.

Poniżej przedstawiony jest prosty przykład wykorzystania klasy `QNetworkAccessManager`, do budowy klienta pobierającego dane ze wskazanego adresu URL (ang. *Uniform Resource Locator*) (rys. 7.2, listingi 7.4–7.6). Pobieranie danych uruchamia funkcja `get` (26. wiersz listingu 7.5). Jej parametrem jest obiekt klasy `QUrl`, reprezentujący adres URL zasobu. Funkcje tej klasy umożliwiają weryfikację poprawności adresów i szereg innych operacji.

Sygnał `finished()`, emitowany przez obiekt `QNetworkAccessManager` po zakończeniu wykonywania zadania, został połączony z naszym slotem `downloadFinished()` (wiersz 19. listingu 7.5), który jest odpowiedzialny za zapisanie pobranych danych (przechowywanych w obiekcie `QNetworkReply`) do pliku `http.out`, w bieżącym katalogu. Wiersz 35. zawiera instrukcję skasowania niepotrzebnego już obiektu.



Rysunek 7.2. Okno prostego klienta sieciowego

Listing 7.4. Deklaracja klasy klienta HTTP

```
 1 #ifndef HTTPWINDOW_H
 2 #define HTTPWINDOW_H
 3
 4 #include <QtGui>
 5 #include <QtNetwork>
 6
 7 class HTTPWindow : public QWidget
 8 {
 9     Q_OBJECT
10
11 public:
12     HTTPWindow(QWidget *parent = 0);
13
14 private:
15     QLineEdit* addrEdit;
16     QPushButton* goButton;
17     QFile* file;
18     QNetworkAccessManager *manager;
19
20 signals:
```

```
21
22 public slots:
23
24 private slots:
25     void download();
26     void downloadFinished(QNetworkReply*);
27 };
28
29 #endif  //  HTTPWINDOW_H
```

Listing 7.5. Definicja klasy klienta HTTP

```
1  #include "httpwindow.h"
2
3  HTTPWindow::HTTPWindow(QWidget *parent) :
4      QWidget(parent){
5      setWindowTitle("HTTP");
6
7      addrEdit = new QLineEdit("http://www.umcs.lublin.pl/");
8      goButton = new QPushButton("Download");
9
10     QVBoxLayout* layout = new QVBoxLayout();
11     layout->addWidget(addrEdit);
12     layout->addWidget(goButton);
13     setLayout(layout);
14
15     connect(goButton, SIGNAL(clicked()),
16             this, SLOT(download()));
17
18     manager = new QNetworkAccessManager(this);
19     connect(manager, SIGNAL(finished(QNetworkReply*)),
20             this, SLOT(downloadFinished(QNetworkReply*)));
21 }
22
23 void HTTPWindow::download(){
24    goButton->setEnabled(FALSE);
25
26    manager->get(QNetworkRequest(QUrl(addrEdit->text())));
27 }
28
29 void HTTPWindow::downloadFinished(QNetworkReply* reply){
30     file = new QFile("http.out");
31     file->open(QIODevice::WriteOnly);
32     file->write(reply->readAll());
33     file->close();
34
35     reply->deleteLater();
36     goButton->setEnabled(TRUE);
37 }
```

Listing 7.6. Klient HTTP – funkcja main

```
1 #include <QtGui/QApplication>
2 #include "httpwindow.h"
3
4 int main(int argc, char *argv[])
5 {
6     QApplication a(argc, argv);
7     HTTPWindow w;
8     w.show();
9
10    return a.exec();
11 }
```

Można sprawdzić, że przedstawiony program umożliwia pobieranie danych nie tylko z serwerów WWW, w protokole HTTP. Przykładowo, wpisanie adresu w postaci ftp://ftp.funet.fi/pub/README spowoduje pobranie pliku z serwera FTP (o ile istnieje).

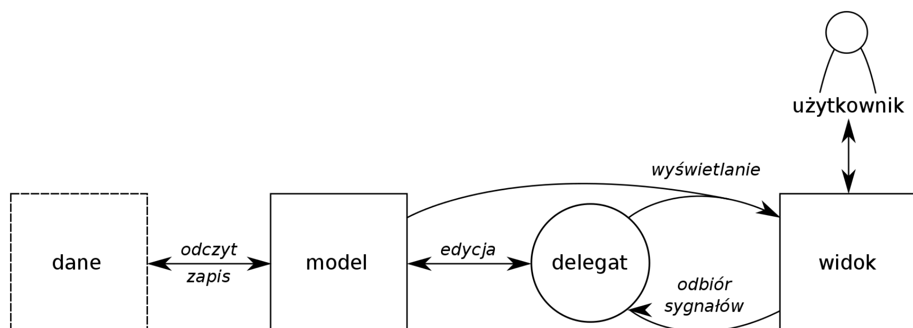
Rozdział 8

Architektura model/widok

8.1. Wzorzec projektowy MVC **110**
8.2. Implementacja architektury model/widok w Qt **111**
8.2.1. Model 111
8.2.2. Widok 112
8.3. Przegląd klas widoku **113**
8.3.1. Lista 113
8.3.2. Tabela 114
8.3.3. Drzewo 115
8.4. Komunikacja między modelem a widokiem **116**
8.4.1. Poruszanie w modelu 116
8.4.2. Delegat 118
8.4.3. Modyfikacja delegata 120
8.4.4. Selekcja 126
8.5. Baza danych w architekturze model/widok **131**
8.5.1. Połączenie z bazą danych 132
8.5.2. Korzystanie z bazy danych 133
8.5.3. Model zapytania i tabeli 135
8.5.4. Model tabeli relacyjnej 136

8.1. Wzorzec projektowy MVC

Omówienie architektury model/widok w Qt należy rozpocząć od przypomnienia wzorca projektowego model-widok-sterownik (ang. model-view-controller, MVC), którego jest ona implementacją. Wzorzec ten zakłada podział programu na trzy części. W pierwszej z nich – modelu, znajduje się logika progamu oraz dane (lub, gdy dane nie są bezpośrednią częścią programu, mechamizmy dostępu do danych). Druga część architektury – widok, zawiera w sobie wszystkie elementy interejsu użytkownika. Sterownik, zwany w Qt delegatem, ma za zadanie łączyć obie warstwy udostępniając informacje o zmianach i modyfikacjach modelu w jedną stronę, a o działaniach użytkownika w drugą. Jak widać na



Rysunek 8.1. Schemat wzorca projektowego MVC z modyfikacją nomenklatury używaną w Qt.

schemacie 8.1 w przypadku, gdy działania podejmowane w warstwie widoku nie wpływają na model można połączyć te dwie warstwy bezpośrednio (choć jednostronnie) z pominięciem delegata. W rzeczywistości, zostanie wówczas wykorzystany standardowy delegat użytego widoku.

O ile w przypadku prostych, kilkuklasowych projektach programowanie wykorzystujące wzorzec MVC wydaje się niecelowe – zysk jest niewspółmierny do dodatkowo włożonej pracy, o tyle w bardziej złożonych aplikacji jego zastosowanie przynosi konkretne korzyści:
— przejrzystość kodu poprzez rodzielenie warstwy modelu od warstwy interfejsu,
— uproszczenie synchronizacji między różnymi widokami opisującymi ten sam model,
— możliwość podmiany bibliotek widoku bez naruszenia warstwy modelu (wieloplatformowość),

Architektura MVC jest bardzo popularna i wykorzystywana w progamowaniu aplikacji webowych. Widok (strona internetowa) jest w jednoznaczny

sposób oddzielona od programu działającego na serwerze. Korzystają z niego popularne frameworki sieciowe m.in. *Java Server Faces*, *Django*, *Ruby on Rails*.

8.2. Implementacja architektury model/widok w Qt

Qt udostępnia kilkadziesiąt klas wspomagających programowanie z wykorzystaniem architektury model/widok. Pozwalają one zarówno na zachowanie podstawowej funkcjonalności typowych sposobów reprezentacji, jak lista, czy drzewo, jak również na zbudowanie całkiem nowych połączeń model/widok z klasami organizującymi reprezentację danych z jednej strony, a samodzielną konstrukcję klasy widoku z wykorzystaniem metody `paintEvent` i przechwytywaniem zdarzeń myszy i klawiatury z drugiej.

8.2.1. Model

Najbardziej podstawową klasą modelu jest `QAbstractItemModel`. Mimo, że jako klasa abstrakcyjna, sama nie jest bezpośrednio używana, definiuje standardowy interfejs dla wszystkich klas modeli. Tworzenie własnego modelu implementującego `QAbstractItemModel` omówione zostanie w podrozdziale **??**. Póki co, spróbujmy się zaznajomić z predefinowanymi klasami modeli.

Dziedzicząca z `QAbstractItemModel` klasa `QStandardItemModel` zapewnia ogólny model do przechowywania danych w złożonej postaci. Dane w modelu są zapisywane w postaci przypominającej tablicę dwuwymiarową - ich położenie określane jest za pomocą pary wiersz-kolumna. Pojedyncze pole danych jest reprezentowane za pomocą obiektu klasy `QStandardItem`. Obiekty te mogą zawierać napis, ikonę lub pole wyboru (dwu- i trzystanowe). Można modyfikować ich wygląd m.in. kolor napisu, kolor tła, czcionkę. Dają one również możliwość przechowywania dodatkowych danych w obiekcie klasy `QVariant`. Obiekty klasy `QStandardItem` można łączyć w struktury drzewiaste – każdy obiekt dysponuje wskaźnikiem do rodzica i listą wskaźników do dzieci.

Część z nich jest wąsko specjalizowana jak np. `QFileSystemModel` przechowująca informację o strukturze plików i katalogów, czy `QStringListModel` służąca do obsługi listy napisów. Z drugiej strony do dyspozycji są klasy abstrakcyjne, bardzo ogólnego zastosowania (`QAbstractItemModel`), wykorzystywane do dziedziczenia własnych modeli.

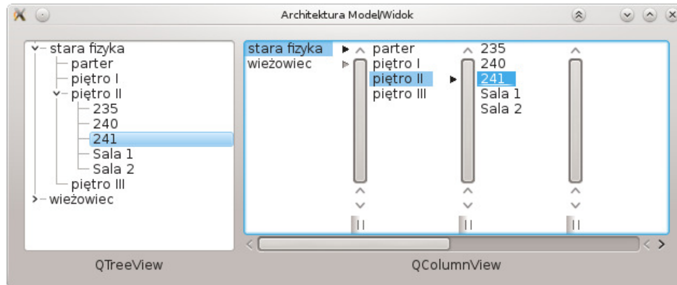
8.2.2. Widok

Qt udostępnia pięć klas widoku. Cztery służą prezentacji modelu o określonej strukturze odzwierciedlonej w ich nazwie:
— `QListView` - do prezentacji modelu w postaci listy,
— `QTableView` - do prezentacji modelu w postaci tabeli,
— `QTreeView` - do prezentacji modelu w postaci drzewa
— `QColumnView` - do prezentacji modelu drzewa w postaci wielu, kaskadowo związanych list.

Piątą klasą jest `QAbstractItemView` stanowiąca interfejs dla wszystkich klas widoku. Dziedziczą po niej cztery wcześniej wymienione klasy. Dowolna, samodzielnie stworzona klasa będąca widokiem w rozumieniu architektury M/W będzie implementowała interfejs opisany w klasie `QAbstractItemView`. Jedną z konsekwencji takiego pojęcia jest konieczność zaimplementowania wirtualnych funkcji:

```
virtual void setModel (QAbstractItemModel *model).
```

Oznacza to, że każda klasa widoku będzie mogła korzystać wyłącznie z modelu implementującego interfejs `QAbstractItemModel`.



Rysunek 8.2. Porównanie prezentacji tego samego modelu za pomocą obiektów klas `QTreeView` oraz `QColumnView`.

8.3. Przegląd klas widoku

W niniejszym podrozdziale przedstawione zostaną sposoby zastosowania architektury model/widok w aplikacjach Qt.

8.3.1. Lista

Najprostszym przykładem architektury model/widok jest lista. Do jej stworzenia, jako klasy modelu, można użyć `QStandardItemModel`. Klasą widoku będzie oczywiście `QListView`.

Jako punkt wyścia traktowany jest projekt zawierający pusty widget. W pliku nagłówkowym należy zadeklarować wskaźniki na model oraz widok.

Listing 8.1. Lista – widget.h

```
1 QStandardItemModel *model;
2 QListView *listView;
```

Kolejnym krokiem jest stworzenie modelu i dodanie do niego kilku pozycji. Klasa `QStandardItemModel` udostępnia kilka metod pozwalających na dodanie do jej obiektu nowego elementu. Wykorzystana zostanie metoda `appendRow` przyjmująca w argumencie wskaźnik na obiekt typu `QStandardItem` dodająca do modelu kolejny wiersz i umieszczającą w nim przekazany obiekt.

Listing 8.2. Lista – widget.cpp

```
1 model=new QStandardItemModel(this);
2 model->appendRow(new QStandardItem("Element 1"));
3 model->appendRow(new QStandardItem("Element 2"));
4 model->appendRow(new QStandardItem("Element 3"));
```

Nie ma potrzeby tworzenia wskaźników do poszczególnych elementów. Jeżeli zajdzie potrzeba dostępu do któregokolwiek z nich można uzyskać za pomocą metody:

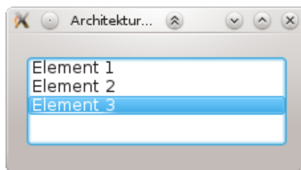
```
QStandardItem *item (int row, int column=0)
```

Nie będą one również potrzebne przy kasowaniu obiektu modelu – destruktor automatycznie kasuje jego wszystkie elementy. Ostatnim krokiem jest stworzenie obiektu klasy widoku oraz ustawienie mu stworzonego modelu.

Listing 8.3. Lista – widget.cpp

```
1 listView=new QListView(this);
2 listView->setModel(model);
```

Po dodaniu stworzonego obieku widoku do widgetu np. poprzez użycie jednej z klas dziedziczącej po `QLayout` wynik powinien być zbliżony do przedstawionego na rysunku 8.3.



Rysunek 8.3. Wykorzystanie widoku `QListView` z modelem `QStandardItemModel`.

8.3.2. Tabela

Klasa `QStandardItemModel` pozwala tworzyć modele o więcej niż jednym wymiarze. Użyta na zasadzie analogii do poprzedniego przypadku metoda `appendColumn` dodałaby element w pierwszym wierszu następnej pustej kolumny. Aby ustawić element w dowolnym miejscu można posłużyć się metodą:

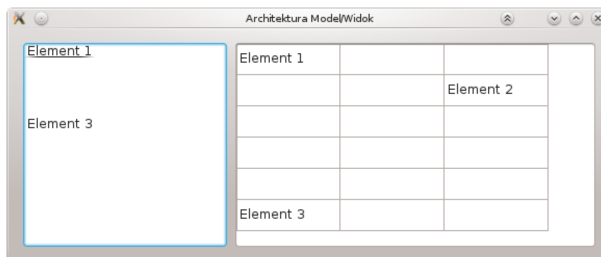
```
void setItem (int row, int column, QStandardItem *item)
```

Nowy obiekt klasy `QStandardItemModel` posiada rozmiar 0×0. Można go modyfikować za pomocą metod `setRowCount` oraz `setColumnCount`. W przypadku ustawienia nowego elementu w polu o współrzędnych przekraczających rozmiar modelu, jest on dostosowywany tak, aby go pomieścić. Jeżeli we wskazanym miejscu element już się znajduje, zostanie skasowany. Po zastąpieniu linii 2–4 w listingu 8.2 następującymi:

Listing 8.4. Tabela – widget.cpp

```
1 model->setItem(0,0,new QStandardItem("Element 1"));
2 model->setItem(1,2,new QStandardItem("Element 2"));
3 model->setItem(5,0,new QStandardItem("Element 3"));
```

otrzymany wynik powinien przedstawiać się podobnie do zrzutu z rysunku 8.4. Nietrudno zauważyć, że widok listy uwzględnia tylko pierwszą kolumnę. Po zamianie wszystkich wystąpień typu `QListView` na `QTableView` wyświetlony zostanie widget tabeli, w której znajdzie się prezentacja wcześniej zmodyfikowanego modelu.



Rysunek 8.4. Porównanie prezentacji tego samego modelu za pomocą obiektów klas `QListView` oraz `QTableView`.

8.3.3. Drzewo

Mimo, że klasa `QStandardItemModel` oferuje jedynie modyfikację ilości wierszy i kolumn istnieje możliwość dodania kolejnych wymiarów. Dokonuje się tego poprzez zagnieżdżenie elementów w sobie. Klasa `QStandardItem`, której obiektami są wspomniane elementy posiada metody:

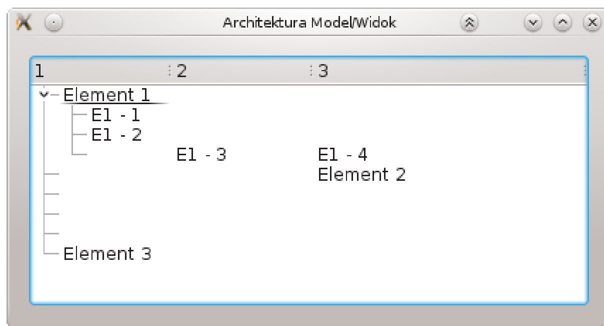
```
void setChild (int row, int column, QStandardItem * item),
void setChild (int row, QStandardItem * item),
```

pozwalające na dodanie nowych elementów, podrzędnych względem obiektu, na rzecz którego wywołana została metoda. Druga z przedstawionych metod przyjmuje indeks kolumny jako 0. Po modyfikacji fragmentu kodu odpowiedzialnego za zapełnienie modelu danymi w następujący sposób:

Listing 8.5. Tabela – widget.cpp

```
1  itemModel.setItem(0,0,new QStandardItem("Element 1"));
2  itemModel.item(0,0)->setChild(0,  new QStandardItem("E1 - 1"));
3  itemModel.item(0,0)->setChild(1,0,new QStandardItem("E1 - 2"));
4  itemModel.item(0,0)->setChild(2,1,new QStandardItem("E1 - 3"));
5  itemModel.item(0,0)->setChild(2,2,new QStandardItem("E1 - 4"));
6
7  itemModel.setItem(1,2,new QStandardItem("Element 2"));
8  itemModel.item(1,2)->setChild(0,  new QStandardItem("E2 - 1"));
9  itemModel.item(1,2)->setChild(1,  new QStandardItem("E2 - 2"));
10
11 itemModel.setItem(5,0,new QStandardItem("Element 3"));
```

otrzymany efekt będzie podobny do przedstawionego na rysunku 8.5. Uwidacznia on ograniczenie klasy `QTreeView`, która umożliwia tworzenie gałęzi jedynie od elementów znajdujących się w pierwszej kolumnie. Elementy podrzędne względem położonego w trzeciej kolumnie „Elementu 2” nie zostały pokazane.



Rysunek 8.5. Ograniczenia widoku `QTreeView`.

8.4. Komunikacja między modelem a widokiem

W poprzednim rozdziale omówione zostały klasy służące do wyświetlenia gotowego modelu. Poza możliwością prezentacji posiadają one również szeroki wachlarz metod pozwalających zaprogramować interakcję z użytkownikiem.

8.4.1. Poruszanie w modelu

Wszystkie klasy dziedziczące po `QAbstractItemView` udostępniają zestaw sygnałów zawiadamiających o działaniach użytkownika, m.in. `clicked`, `doubleClicked`, `pressed`. Sygnały te przyjmują jako argument obiekt klasy `QModelIndex`. Klasa ta określa położenie elementu w modelu. Istotną cechą funkcji wydających na wyjściu obiekt klasy `QModelIndex`, jest zwracanie go jako stałej (`const`). Jest to spowodowane tymczasowym charakterem indeksowania w modelach danych. Po wprowadzeniu zmian w modelu zadeklarowany wcześniej indeks może zawierać zupełnie inne dane. Można temu zapobiec stosując indeksy klasy `QPersistentModelIndex`. Indeksów nie należy utożsamiać ze wskaźnikami na element modelu.

Kanwą do pracy będzie stworzony, podobnie, jak we wcześniejszej części rozdziału, widok listy. Model, do którego będziemy mieć dostęp za pomocą tego widoku będzie miał strukturę drzewiastą. Jak zostało to pokazane w podrozdziale 8.3.1 niemożliwe jest wyświetlenie zagnieżdżeń w jednowymiarowej liście. Z pomocą przychodzi obecna w klasach dziedziczących po `QAbstractItemView` metoda:

```
void setRootIndex (const QModelIndex &index).
```

Przyjmuje ona wspomniany obiekt `QModelIndex` określający miejsce w modelu, które ma być odtąd traktowane jako element nadrzędny wszystkich elementów znajdujących się w widoku.

Celem przykładu jest stworzenie widoku listy, który przy podwójnym kliknięciu będzie pokazywał elementy podrzędne względem wskazanego. Jednym ze sposobów uzależnienia wykonania się metody zmiany korzenia jest połączenie jej w sygnałem kliknięcia. Metoda `setRootIndex` nie jest slotem, więc konieczne będzie stworzenie pomocniczego slotu.

Listing 8.6. Lista z zagnieżdżeniami – widget.h

```
1 private slots:
2     void updateRoot(QModelIndex);
```

Slot powinien, warunkując swoje działanie od posiadania przez wskazany element elementów podrzędnych, wywoływać metodę `setRootIndex`.

Listing 8.7. Lista z zagnieżdżeniami – widget.cpp

```
1 void Widget::updateRoot(QModelIndex index) {
2     if(listView->model()->hasChildren(index))
3     listView->setRootIndex(index);
4 }
```

Przekazany w agrumencie indeks definiuje położenie wskazanego elementu. Zostaje on ustawiony jako nowy element nadrzędny (korzeń) listy. Mimo różnic widocznych dla końcowego użytkownika, żaden element modelu nie został zmodyfikowany, ani nie zmienił swojego położenia. Warunek sprawdzający istnienie elementów podrzędnych (potomków) względem obiektu wskazanego indeksem przeciwdziała sytuacji, w której prezentowana w widoku lista potomków byłaby pusta. Aby slot mógł zostać użyty należy go, w konstruktorze głównego widgetu, połączyć z sygnałem podwójnego kliknięcia.

Listing 8.8. Lista z zagnieżdżeniami – widget.cpp (konstruktor)

```
1 connect(listView, SIGNAL(doubleClicked(QModelIndex)),
2       this, SLOT(updateRoot(QModelIndex)));
```

Pobieżna analiza napisanego programu ukazuje brakującą funkcjonalność – po zejściu na głębszy poziom niemożliwy staje się powrót. Dodanie do listy pola pozwalającego na cofnięcie się wymaga stworzenia nowej klasy widoku lub modelu dziedziczącej z aktualnie wykorzystywanych. Łatwiej jest to rozwiązać dodając przycisk `QPushButton`. Konieczne będzie też napisanie

nowego slotu odpowiadającego za cofnięcie się w hierarchii drzewiastego modelu,

Listing 8.9. Lista z zagnieżdżeniami – widget.h

```
private slots:
    void updateRootUpToParent();
```

oraz implementacji:

Listing 8.10. Lista z zagnieżdżeniami – widget.cpp

```
void Widget::updateRootUpToParent() {
  updateRoot(listView->rootIndex().parent());
}
```

W slocie `updateRootUpToParent` wykorzystana została zaprogramowana wcześniej metoda `updateRoot` przyjmująca w argumencie obiekt klasy `QModelIndex`. Taki obiekt jest zwracany przez metodę `parent` tej klasy. Obiektem, którego rodzic (element nadrzędny) jest poszukiwany, jest oczywiście najwyższy w hierarchii, aktualny element widoku. Dostęp do niego realizowany jest poprzez metodę `rootIndex`. Ostatnim krokiem jest połączenie stworzonego slotu z sygnałem wciśnięcia przycisku.

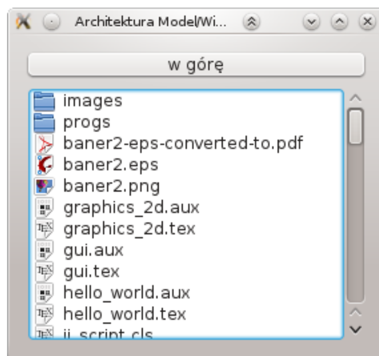
Listing 8.11. Lista z zagnieżdżeniami – widget.cpp (konstruktor)

```
connect(upButton, SIGNAL(clicked()),
      this, SLOT(updateRootUpToParent()));
```

Qt posiada klasę modelu `QFileSystemModel` reprezentującą strukturę systemu plików. Może być on wyświetlony w tak stworzonym widoku listy w miejsce ręcznie stworzonego modelu klasy `QStandardItemModel`. Wynik działania przedstawia rysunek 8.6.

8.4.2. Delegat

Przy zastosowaniu modelu klasy `QFileSystemModel` uwagę zwracają ikony odpowiadające typom plików oraz katalogom. Dotychczas przyjęte uproszczenie, zakładające wyświetlanie danych z modelu za pomocą widoku okazuje się niepełne. W obiekcie widoku, poza zmianą modelu nie zaszła żadna zmiana, a mimo to dane wyświelane są w inny sposób. Za prezentację poszczególnych elementów w widoku odpowiada obiekt klasy delegata, a w analizowanym przykładzie konkretnie `QStyledItemDelegate`. Delegat zawiera informacje w jaki sposób dane są wyświetlane, jak należy zareagować



Rysunek 8.6. Widok klasy `QListView` prezentujący model klasy `QFileSystemModel`.

na próbę ich zmiany i jak zinterpretować wprowadzone przez użytkownika dane. Widok jest jedynie siatką, na której osadzone są obiekty delegatów.

Standardowym delegatem dla takich widoków jak `QListView` jest obiekt klasy `QStyledItemDelegate`. Pozwala on modyfikować wygląd odpowiadającego mu elementu modelu za pomocą danych zapisanych w nim za pomocą metody klasy `QStandardItem`:

```
void setData (const QVariant &value, int role).
```

W polu `value` możliwe jest wstawienie danej określonego typu, podczas, gdy wartość w `role` definiuje zastosowanie tej danej. Wygodną formą określenia wartości `role` jest zastosowanie typu wyliczeniowego `Qt::ItemDataRole`. Przykładowo, wykonanie następującego kodu spowoduje stworzenie elementu modelu, w którym tekst będzie wyświetlany białymi literami na czerwonym tle.

Listing 8.12. Modyfikacja wyglądu delegata za pomocą ustawień modelu

```
1 QStandardItem *item=new QStandardItem("czerwony");
2 item->setData(QBrush(0xFF0000),Qt::BackgroundRole);
3 item->setData(QBrush(0xFFFFFF),Qt::ForegroundRole);
```

Można także wykorzystać indeksowanie modyfikując kod utworzonego wcześniej slotu `updateRoot` spowodować zaznaczenie czerwonym kolorem elementu po podwójnym kliknięciu, jeżeli nie ma on potomków.

Listing 8.13. Lista z zagnieżdżeniami – widget.cpp

```
void Widget::updateRoot(QModelIndex index) {
    if(listView->model()->hasChildren(index))
        listView->setRootIndex(index);
    else {
        if(!listView->model()
      ->data(index,Qt::BackgroundColorRole).isValid())
          listView->model()
        ->setData(index,QBrush(0xFF0000),Qt::BackgroundRole);
      else
          listView->model()
        ->setData(index,NULL,Qt::BackgroundRole);
    }
}
```

Jeżeli oznaczony już kolorem element zostanie kliknięty ponownie, wtedy wartość danej informującej o kolorze tła jest kasowana. O tym, czy dana zapisana w obiekcie klasy `QVariant` (zwracanej przez metodę `data` klasy `QAbstractItemModel`) jest poprawna informuje wynik działania metody `isValid`. W podobny sposób można modyfikować m.in. rozmiar, czcionkę tekstu czy symbol graficzny znajdujący się obok tekstu elementu, pamiętając jednak o umieszczeniu w zmiennej typu `QVariant` wartości zgodnej z typem w oczekiwanej roli. Ustawienie jako kolor tła napisu, choć formalnie możliwe, nie przyniesie oczekiwanych efektów.

8.4.3. Modyfikacja delegata

Określenie wyglądu reprezentacji elementu w widoku za pomocą nadawania mu wartości w określonych rolami wykorzystuje funkcjonalność klasy `QStyledItemDelegate`. Można jednak pójść dalej i zmienić w całości działanie delegata dziedzicząc własną klasę po jednej z klas delegatów: `QAbstractItemDelegate`, `QItemDelegate` oraz poznany już `QStyledItemDelegate`.

Przykładem demonstrującym tworzenie własnego delegata będzie tabela, w której kolejnym osobom określonym za pomocą tytułu, imienia i nazwiska odpowiadać będzie ilość punktów w przedziale od 0 do 100. Punktem wyjścia jest pusty widget. Należy stworzyć i osadzić w nim widok klasy `QTableView` oraz połączyć go z modelem klasy `QStandardItemModel` analogicznie jak w przykładzie z podrozdziału 8.3.2. Dodatkowo można opisać nagłówki poszczególnych kolumn za pomocą metody modelu:

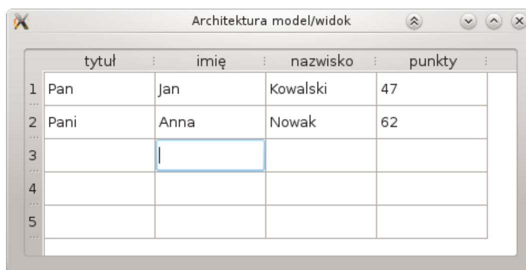
```
bool setHeaderData (int section, Qt::Orientation orientation,
          const QVariant &value, int role).
```

Jej użycie w programie może mieć następującą postać:

Listing 8.14. Modyfikacja delegata – widget.cpp

```
1 model->setHeaderData(2,Qt::Horizontal,QString("nazwisko"));
```

Widok tabeli automatycznie łączy podwójne kliknięcie z udostępnieniem możliwości edycji wybranego pola. Domyślnie odbywa się to poprzez stworzenie w polu widoku linii edycji tekstu. W wielu przypadkach to wystarcza. W analizowanym programie załóżmy ograniczenie wartości tytułu do



Rysunek 8.7. Standardowe działanie widoku w reakcji na podwójne kliknięcie

napisów „Pan" i „Pani". Wygodnie byłoby zastąpić ręczne ich wpisywanie wyborem z listy rozwijanej. Konieczne będzie to tego stworzenie użycie, nowego, dostosowanego delegata. Niech nowa klasa nazywa się `TitleDelegate` i dziedziczy po `QStyledItemDelegate`.

Delegat musi reimplementować trzy funkcje. Pierszą z nich:

```
QWidget *createEditor(QWidget *parent,
            const QStyleOptionViewItem &option,
                  const QModelIndex &index) const
```

można interpretować jako „konstruktor" dla edytora. Właściwy konstruktor klasy w tym przypadku pozostaje pusty. Wywołuje się go tylko raz dla danego widoku, zaś metodę `createEditor` za każdym razem, kiedy widok otrzyma żądanie edycji pola. Argument `parent` określa widget wewnątrz którego ma pojawić się edytor. Jest to obiekt klasy widoku i jest on jednakowy dla wszystkich pól. W przypadku nadania mu wartości `NULL` edytor pokazałby się w osobnym oknie. Stosuje się to, gdy edytorem jest rozbudowany widget potrzebujący więcej miejsca niż kratka w tabeli, czy wiersz w liście. Argument `option` przechowuje informacje na temat formatowania pola, a `index` o indeksie elementu w modelu, którego reprezentację ma modyfikować edytor.

Implementacja metody `createEditor` dla listy rozwijanej wygląda następująco:

Listing 8.15. Modyfikacja delegata – titledelegate.cpp

```
1 QWidget *TitleDelegate::createEditor(QWidget *parent,
2     const QStyleOptionViewItem &,
3     const QModelIndex &) const {
4     QComboBox *widget=new QComboBox(parent);
5     QStringList itemList; itemList<<""<<"Pan"<<"Pani";
6     widget->addItems(itemList);
7     return widget;
8 }
```

Nie ma potrzeby przekazywania opcji stylu i indeksu – edytor ma działać niezależnie od pola, w którym zostanie wywołany. Metoda tworzy obiekt typu `QComboBox` i dodaje do niego trzy wartości do wyboru, a następnie zwraca go jako wynik działania funkcji. Tak stworzony obiekt zostanie wyświetlony w polu widoku. Drugą metodą jest:

```
void setEditorData(QWidget *editor,
           const QModelIndex &index) const;
```

Służy ona ustawieniu wartości w nowo stworzonym edytorze na podstawie danych zawartych w elemencie modelu. Widgetem `editor` jest uprzednio stworzony widget znajdujący się w polu odpowiadającym elementowi o indeksie `index`. Zaimplementowana jest następująco:

Listing 8.16. Modyfikacja delegata – titledelegate.cpp

```
1 void TitleDelegate::setEditorData(QWidget *editor,
2     const QModelIndex &index) const {
3     QComboBox *widget=static_cast<QComboBox *>(editor);
4   int value=index.model()->data(index,Qt::EditRole).toInt();
5     widget->setCurrentIndex(value);
6 }
```

Funkcja przyjmuje argument `editor` jako wskaźnik na obiekt klasy `QWidget`. Korzystanie z niego jako listy rozwijanej wymaga zrzutowania na typ `QComboBox`. Przyjęty edytor jest widgetem dopiero co stworzonym metodą `createEditor` i musi być mu ustawiona wartość na podstawie danych znajdujących się w elemencie modelu. Dysponując indeksem można, za pomocą jego metody `model` odwołać się do modelu, a następnie na rzecz modelu wywołać metodę `data` przyjmującą indeks i rolę, zwracającą dane jako obiekt klasy `QVariant`. Znajduje się tam zapisana pozycja w liście rozwijanej. Aby można było jej użyć w metodzie `setCurrentIndex` należy zwrócić jej war-

tość jako liczbę całkowitą, za co odpowiada metoda `toInt` klasy `QVariant`. Ostatnia metoda służy zapisowi danych wprowadzonych w edytorze przez użytkownika do elementu modelu:

Listing 8.17. Modyfikacja delegata – titledelegate.cpp

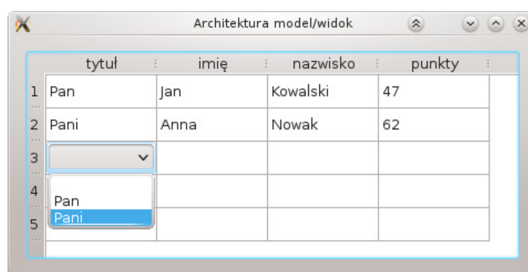
```
1 void TitleDelegate::setModelData(QWidget *editor,
2     QAbstractItemModel *model,
3         const QModelIndex &index) const {
4   QComboBox *widget = static_cast<QComboBox*>(editor);
5   int value = widget->currentIndex();
6     model->setData(index, widget->currentText(), Qt::DisplayRole);
7     model->setData(index, value, Qt::UserRole);
8 }
```

Analogicznie jak w przypadku poprzedniej metody konieczne jest zrzutowanie przyjętego w argumencie widgetu na klasę listy rozwijanej. Następnie w modelu ustawiane są dwie wartości. Napis, który ma zostać wyświetlony po zamnkięciu edytora powinien być zgodny z napisem wybranego pola elementu listy. Wartość `value` zawierająca indeks jest również zapisywana, aby mogła być wykorzystana w metodzie `setEditorData` do określenia aktywnego elementu listy.

Aby korzystać z delegata w określonej kolumnie widoku należy na rzecz widoku wywołać metodę:

```
void setItemDelegateForColumn (int column,
     QAbstractItemDelegate *delegate).
```

Analogiczna funkcja dla wierszy ma nazwę `setItemDelegateForRow`. W przypadku jednolitego modelu, można wszystkim elementom przypisać tego samego delegata metodą `setItemDelegate`. Działanie tak zaprogramowanego delegata pokazuje rysunek 8.8.



Rysunek 8.8. Wykorzystanie delegata z edytorem w postaci listy rozwijanej.

O ile w przypadku imienia i nazwiska można pozostawić standardowy

edytor, o tyle w kolumnie punktów należałoby zastosować widget specjalizowany do przyjmowania wartości liczbowych. Takim widgetem jest `QSpinBox`. Wykorzystanie go wymusza powtórzenie czynności wykonanych przy tworzeniu poprzedniego delegata. Należy stworzyć nową klasę. Niech nazywa się ona `PointsDelegate`. Podobnie jak w poprzednim przypadku zostaną zmodyfikowane trzy metody:

Listing 8.18. Modyfikacja delegata – pointsdelegate.cpp

```
 1 QWidget *PointsDelegate::createEditor(QWidget *parent,
 2     const QStyleOptionViewItem &,
 3     const QModelIndex &) const {
 4     QSpinBox *widget=new QSpinBox(parent);
 5   widget->setRange(0,100);
 6     return widget;
 7 }
 8
 9 void PointsDelegate::setEditorData(QWidget *editor,
10     const QModelIndex &index) const {
11     QSpinBox *widget=static_cast<QSpinBox *>(editor);
12     widget->setValue(index.model()
13     ->data(index,Qt::EditRole).toInt());
14 }
15
16 void PointsDelegate::setModelData(QWidget *editor,
17     QAbstractItemModel *model,
18     const QModelIndex &index) const {
19   QSpinBox *widget = static_cast<QSpinBox*>(editor);
20   int value = widget->value();
21     model->setData(index, value, Qt::EditRole);
22
23     QColor color; color.setHsv(1.2*value,255,255);
24   model->setData(index,QBrush(color),Qt::BackgroundRole);
25 }
```

Zmiany w stosunku do klasy `TitleDelegate` w dwóch pierwszych metodach ograniczają się do modyfikacji wartości widgetu innego typu. W dwóch ostatnich liniach funkcji `setModelData` zostało zaprogramowane formatowanie warunkowe. Dzięki temu pola, których wartości są bliskie liczby 0 mają czerwone tło, znajdujące się w połowie przedziału – żółte, zaś bliskie liczby 100 – zielone. Doskonale sprawdza się w tym zastosowaniu przestrzeń barw HSV. Jej składowa H (ang. hue, barwa), widoczna na rysunku 8.9, zmienia swoje wartości od 0 dla koloru czerwonego, do 120 dla zielonego. Użyta jest proporcja $\frac{\texttt{color.hue}}{120} = \frac{\texttt{value}}{100}$, gdzie znając wartość zmiennej `value` można obliczyć składową barwy. Dzięki temu przejścia są płynne. Po ustawieniu delegata na ostanią kolumnę widoku rezultat powinien przypominać zrzut

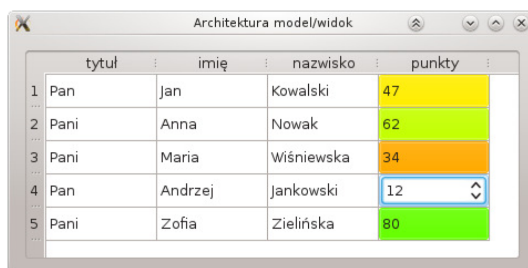
Rysunek 8.9. Składowa barwy przestrzeni barw HSV.

na rysunku 8.10. Zastąpienie ostatniej linii metody `setModelData` spowoduje ustawienie tła wszystkim elementom w rzędzie.

Listing 8.19. Modyfikacja delegata – pointsdelegate.cpp

```
1 QModelIndex rowIndex;
2 for(int i=0; i<model->columnCount(); i++) {
3     rowIndex=model->index(index.row(),i);
4     model->setData(rowIndex,QBrush(color),Qt::BackgroundRole);
5 }
```

W omówionym przykładzie jako klasę bazową do dziedziczenia zastosowano `QStyledItemDelegate`. Pojawiła się ona w wersji Qt 4.4 jako alternatywa dla klasy `QItemDelegate`. Różni się ona użyciem aktualnego stylu programu. Z kolei obie te klasy dziedziczą po `QAbstractItemDelegate`. Zasadne jest pytanie, jakiej klasy należy użyć tworząc własnego delegata. `QAbstractItemDelegate` posiada funkcje wirtualne które muszą zostać zaprogramowane. Szczególnie pracochłonna jest implementacja metody `paint` wyświetlającej dane użytkownikowi. W przypadku reimplementacji całej funkcjonalności delegata jest to dobry wybór. `QItemDelegate` oraz `QStyledItemDelegate` zapewniają domyślną implementację metod potrzebnych do podstawowego działania. Dokumentacja Qt zaleca dziedziczenie po `QStyledItemDelegate` przy tworzeniu własnych delegatów i właśnie ta klasa jest domyślnie użyta jako delegat standardowych klas widoku jak omawiane `QListView` czy `QTableView`.



Rysunek 8.10. Delegat z edytorem klasy `QSpinBox` oraz warunkowym ustawieniem koloru tła.

8.4.4. Selekcja

Selekcja jest cechą właściwą warstwie widoku. Porównanie stworzonych do tej pory przykładów, pokazuje różnice w zaznaczaniu elementów. W przypadku widoku klasy `QListView` domyślnie istnieje możliwość zaznaczenia tylko jednego wiersza, podczas gdy widok `QTableView` pozwala na jednoczesne zaznaczenie wielu komórek.

Każdy obiekt klasy dziedziczącej z `QAbstractItemView` posiada dwie metody definiujące sposób zaznaczania elementów. Pierwsza z nich, `setSelectionBehavior` definiuje w jakie grupy elementów mają być zaznaczane. Przyjmuje w argumencie jedną z trzech wartości typu wyliczeniowego `QAbstractItemView::SelectionBehavior`:

— `QAbstractItemView::SelectItems` – zaznaczane są poszczególne elementy,
— `QAbstractItemView::SelectRows` – zaznaczane są wiersze,
— `QAbstractItemView::SelectColumns` – zaznaczane są kolumny.

Drugą z nich jest metoda `setSelectionMode` przyjmującą wartości typu `QAbstractItemView::SelectionMode`:

— `QAbstractItemView::NoSelection` – brak możliwości zaznaczenia,
— `QAbstractItemView::SingleSelection` – możliwość zaznaczenia pojedynczego elementu,
— `QAbstractItemView::MultiSelection` – możliwość zaznaczenia wielu elementów,
— `QAbstractItemView::ExtendedSelection` – możliwość zaznaczenia wielu elementów; przy zaznaczeniu nowych elementów poprzednie zaznaczenie jest unieważniane,
— `QAbstractItemView::ContiguousSelection` – możliwość zaznaczenia wielu kolejnych elementów.

Aby możliwe było zaprogramowanie działania w reakcji na zdarzenie zaznaczenia elementów konieczne jest połączenie sygnału informującego o tym, z napisanym slotem. O ile klasa widoku dostarcza sygnały informujące o kliknięciu w określonym miejscu, to brak jest sygnałów powiązanych z selekcją. Znajdują się one w innej klasie – `QItemSelectionModel`. Każdy obiekt klasy dziedziczącej po `QAbstractItemModel` posiada model selekcji. Można uzyskać do niego dostęp za pomocą metody:

```
QItemSelectionModel *selectionModel() const.
```

Klasa `QItemSelectionModel` posiada 4 synały informujące o zmianie zaznaczenia, z których każdy informuje o innej klasie zaznaczenia. Trzy z nich: `currentChanged` (odpowiadający zmianie zaznaczenia pojedynczego elementu), `currentColumnChanged` oraz `currentRowChanged` (odpowiadające odpowiednio zmianie kolumny i zmianie wiersza) przyjmują dwa obiekty klasy `QModelIndex`. Pierwszy z tych indeksów odpowiada nowo zaznaczo-

nemu elementowi, drugi zaś poprzednio zaznaczonemu. Czwarty sygnał, `selectionChanged` odnosi się do całej zaznaczonej powierzchni. Przyjmuje on w argumentach, podobnie jak w poprzednich przypadkach dwie zmienne `current` i `previous`.

Klasa `QModelIndex` indeksująca pojedynczy element jest w tym przypadku niewystarczająca, potrzebna jest możliwość podania przedziału. Daje ją klasa `QItemSelection`, której obiektami są wspomniane argumenty. Przechowują one informację o selekcji w dowolnej postaci i udostępniają, poprzez metodę `indexes`, listę indeksów typu `QModelIndexList`, Demonstruje to przykład: tworzony jest widok tablicy korzystający ze standardowego modelu zawierającego dane jak na rysunku 8.11. dodane zostało również pole tekstowe, w którym pojawi się zawartość zaznaczonych pól. Potrzebny będzie również slot reagujący na zaznaczenie przedziału danych.

Listing 8.20. Selekcja – widget.h

```
1     TableView *tableView;
2     QTextEdit *edit;
3     QStandardItemModel *model;
4     QItemSelectionModel *selectionModel;
5 private slots:
6     void readSelected();
```

Następnie w konstruktorze, po utworzeniu obiektów klas modelu i widoku oraz połączeniu ich ze sobą, należy przypisać model selekcji do wskaźnika oraz połączyć jego sygnał ze stworzonym slotem `readSelected`.

Listing 8.21. Selekcja – widget.cpp (konstruktor)

```
1 selectionModel=tableView->selectionModel();
2 connect(selectionModel,
3   SIGNAL(selectionChanged(QItemSelection,QItemSelection)),
4   this, SLOT(readSelected()));
```

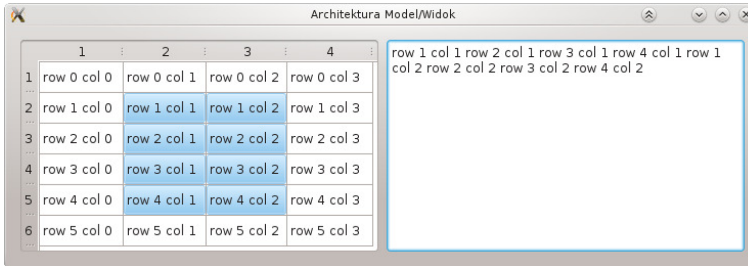
Działanie slotu `readSelected` polega na przepisaniu danych z flagą `Qt::DisplayRole` (która jest domyślną wartością metody `data` klasy `QModelIndex`) do okna tekstowego. Jej implementacja wygląda następująco:

Listing 8.22. Selekcja – widget.cpp

```
1 void Widget::readSelected() {
2     QString text;
3     foreach(QModelIndex index,
4       selectionModel->selectedIndexes()) {
5         text+=index.data().toString()+" ";
6     }
7     edit->setText(text);
```

```
8 }
```

Metoda data zwraca wartość typu QVariant, dlatego konieczne jest jej przekształcenie do napisu, za pomocą metody toString. Wynik działania programu, przedtawia rysunek 8.11. W zaprezentowanej implementacji nie zostały



Rysunek 8.11. Wylistowanie zaznaczonych elementów.

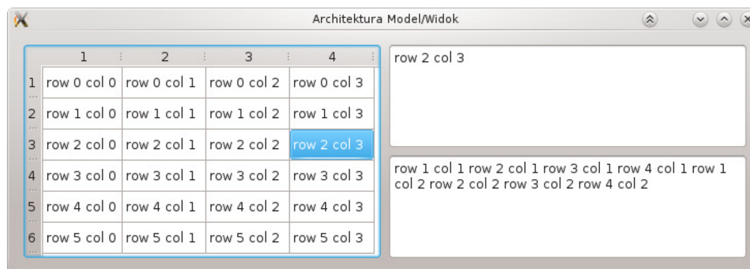
wykorzystane argumenty przekazywane przez sygnał: aktualne i poprzednie zaznaczenie. Możliwa jest modyfikacja programu, w której w dwóch oknach tekstowych prezentowane będą dane indeksów z obu przekazanych argumentów. W tym celu należy zmodyfikować slot readSelected:

Listing 8.23. Selekcja – widget.cpp

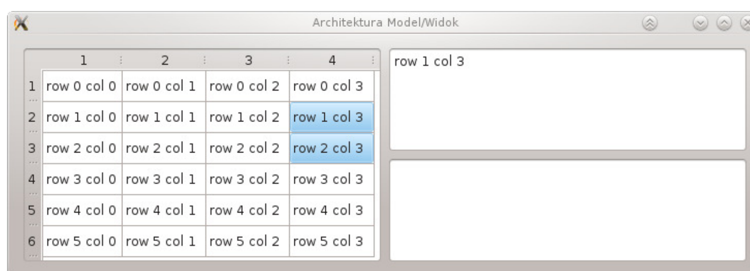
```
1  void Widget::readSelected(QItemSelection current,
2          QItemSelection previous) {
3      QString text;
4      foreach(QModelIndex index, current.indexes())
5          text+=index.data().toString()+" ";
6      selectedEdit->setText(text);
7      text.clear();
8      foreach(QModelIndex index, previous.indexes())
9          text+=index.data().toString()+" ";
10     deselectedEdit->setText(text);
11 }
```

Istotny jest fakt, że obie zmienne przechowują informacje tylko o ostatniej zmianie zaznaczenia, a nie o całym, chwilowym zaznaczeniu. Ilustruje to rysunek 8.13.

Selekcje mogą być też wykonywane z poziomu kodu programu. Klasa QItemSelectionModel udostępnia dwie metody select: jedną przyjmującą pojedynczy indeks (QModelIndex), drugą przyjmującą selekcję (QItemSelection). Obie z nich przyjmują również zmienną typu wyliczeniowego QItemSelectionModel::SelectionFlags. Pozwalają one określić sposób selekcji elementów widoku.



Rysunek 8.12. Dane przekazane przez sygnał `selectionChanged`. Wylistowanie danych indeksów selekcji `current` znajduje się na górze, `previous` na dole. Punktem wyjściowym jest sytuacja pokazana na rysunku 8.11.



Rysunek 8.13. Dane indeksów selekcji `current`. Punktem wyjściowym jest sytuacja pokazana na rysunku 8.12.

Oto ich przykładowe wartości i znaczenie:

— `QAbstractItemView::NoUpdate` – ignoruje bieżącą selekcję,
— `QAbstractItemView::Clear` – czyści selekcję,
— `QAbstractItemView::Select` – zaznacza wskazane pierwszym argumentem elementy (najbardziej typowe działanie),
— `QAbstractItemView::Deselect` – odznacza wskazane elementy,
— `QAbstractItemView::Toggle` – odwraca zaznaczenie elementów.

Wykorzystanie tej metody pozwala uzależnić zaznaczenie tabeli od czynników zewnętrznych. Poniższy przykład demonstruje zaznaczenie jednej z tabel na podstawie selekcji dokonanej w dwóch poprzednich. Potrzebne będą trzy widoki tabel: `t1View`, `t2View` oraz `t3View`. Wszystkie będą korzystać z tego samego modelu (w ogólnym przypadku nie jest to konieczne – selekcja odbywa się w warstwie widoku). W pliku nagłówkowym konieczne będzie stworzenie wskaźników i slotu:

Listing 8.24. Selekcja warunkowa – widget.h

```
1 QTableView *t1View, *t2View, *t3View;
```

```
2 private slots:
3     void toggleSelection();
```

W konstruktorze należy stworzyć obiekty tych widoków, ustawić im model oraz połączyć sygnały selekcji ze stworzonym slotem:

Listing 8.25. Selekcja warunkowa – widget.cpp (konstruktor)

```
1 t1View=new QTableView(this); t1View->setModel(model);
2 t2View=new QTableView(this); t2View->setModel(model);
3 t3View=new QTableView(this); t3View->setModel(model);
4 connect(t1View->selectionModel(), SIGNAL(
5   selectionChanged(QItemSelection,QItemSelection)),
6   this, SLOT(toggleSelection()));
7 connect(t2View->selectionModel(), SIGNAL(
8   selectionChanged(QItemSelection,QItemSelection)),
9   this, SLOT(toggleSelection()));
```

W slocie należy ustawić stosowne zaznaczenie modelu. Wykorzystanie zaznaczenia z pierwszej tabeli (za pomocą flagi `Select`), a następnie odwrócenie ich zaznaczenia (dzięki fladze `Toggle`) względem drugiej tabeli sposoduje powstanie w trzeciej z nich zaznaczenia będącego funkcją XOR (ang. exclusive or, alternatywa wykluczająca) ręcznie wprowadzonych selekcji.

Listing 8.26. Selekcja warunkowa – widget.h

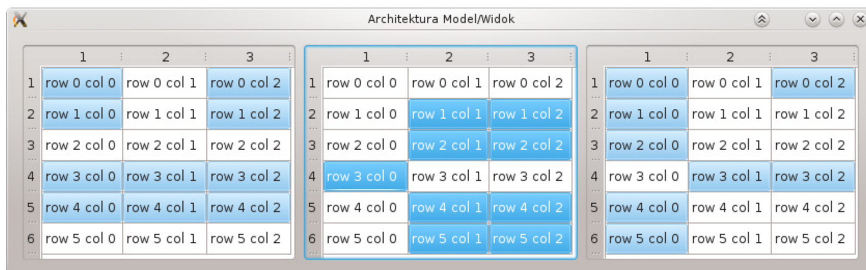
```
1 void Widget::toggleSelection() {
2     QItemSelectionModel *model=t3View->selectionModel();
3     model->select(t1View->selectionModel()->
4     selection(),QItemSelectionModel::Select);
5     model->select(t2View->selectionModel()->selection(),
6     QItemSelectionModel::Toggle);
7 }
```

Wynik prezenture rysunek 8.14.

Każdy widok ma swój oddzielny model selekcji. W wielu przypdkach wygodne jest uwspólnienie modelu selecji między wiele widoków. Jest to szczególnie przydatne przy pracy z różnymi reprezentacjami tego samego modelu (danych). Obiekt klasy dziedziczącej po `QAbstractItemView` posiada metodę `setSelectionModel`, którą należy użyć w tym celu. W przypadku wykorzystującym widoki `t1View` i `t2View` może to wyglądać następująco:

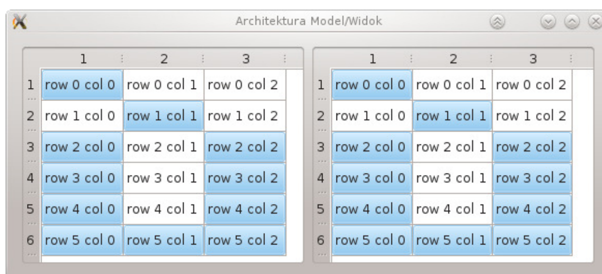
Listing 8.27. Selekcja warunkowa – widget.h

```
1 t2View->setSelectionModel(t1View->selectionModel());
```



Rysunek 8.14. Warunkowe zaznaczenie prawej tabeli jako altertnatywa wykluczająca zaznaczeń dwóch pozostałych tabel.

Rezultat działania widać na rysunku 8.15. Nie ma potrzeby zwrotnego wiązania modelu selekcji widoku `t1View` z modelem widoku `t2View`. Gdyby zaszła potrzeba odtworzenia modelu selekcji widoku `t2View` należy w argumencie metody `setSelectionModel` należy podać pusty, nowostworzony model.



Rysunek 8.15. Użycie tego samego modelu selekcji w dwóch widokach.

8.5. Baza danych w architekturze model/widok

Qt posiada moduł `QtSql` umożliwiający wykorzystanie baz danych. Praca z bazami w Qt wymaga umiejętności posługiwania się języka zapytań. Autor w tym rozdziale zakłada znajomość języka SQL u czytelnika na poziomie składni zapytań i budowania relacyjnych połączeń między tabelami.

Aby możliwe było korzystanie z modułu baz danych, do pliku projektu należy dodać linię:

```
QT += sql
```

Korzystanie z klas i funkcji tego modułu wymaga włączenia do kodu programu pliku `QtSql` agregującego wszystkie pliki nagłówkowe tego modułu, lub poszczególnych plików nagłówkowych użytych klas.

8.5.1. Połączenie z bazą danych

Pierwszą czynnością, którą należy wykonać przed przystąpieniem do pracy z bazą jest oczywiście połączenie się z nią. Klasą reprezentującą połączenie z bazą danych jest `QSqlDatabase`. Nowe połączenie tworzy się nie poprzez konstruktor, a za pomocą wywołania jej statycznej metody

```
QSqlDatabase addDatabase (const QString &type,
          const QString &connectionName)).
```

Wartość `type` określa nazwę sterownika, który jest używany do połączenia z bazą. Np. dla bazy PostgreSQL, będzie to `QPSQL`, zaś dla SQLITE `QSQLITE`. Listę obsługiwanych baz można znaleźć w dokumentacji. Może zdarzyć się, że posiadana kompilacja Qt nie udostępnia wszystkich sterowników. Na wielu platformach systemowych istnieje możliwość pobrania i zainstalowania pakietu zawierającego odpowiednią bibliotekę. W przeciwnym razie konieczna jest ręczna kompilacja ze źródeł. W wersji 4.7 Qt znajdują się one w katalogu `/src/plugins/sqldrivers/`. Istnieje też możliwość samodzielnego napisania sterownika do bazy danych nie przewidzianej przez twórców Qt. Wówczas do dodania nowej bazy wykorzystana będzie metoda:

```
QSqlDatabase addDatabase (QSqlDriver * driver,
          const QString &connectionName)).
```

Obie z przedstawionych metody jako drugi argument przyjmują napis `connectionName`. Pozwala on na określenie i różnych połączeń działających w ramach jednego programu. Można jednocześnie korzystać z wielu połączeń do tej samej i różnych baz danych. Jeżeli `connectionName` nie zostanie zdefiniowana, połączenie staje się standardowym (`defaultConnection`), co pozwoli odnosić się do niego bez podania nazwy.

Aby połączyć się z bazą, należy zdefiniować położenie, nazwę bazy oraz użytkownika i hasło jakie będzie użyte do zalogowania. Dla bazy PostgreSQL może to wyglądać następująco:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL");
db.setHostName("example.org");
```

```
db.setPort(5432);
db.setDatabaseName("foobar");
db.setUserName("user");
db.setPassword("secret");
```

Z kolei, w przypadku bazy SQLITE, która zapisywana jest w pliku wystarczające jest podanie jego nazwy:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName("database.sql");
```

Korzystając z podanych danych można połączyć się z bazą. Służy do tego metoda `open` klasy `QDatabaseConnection`. Zwraca ona wartość logiczną w zależną od powodzenia próby połączenia.

8.5.2. Korzystanie z bazy danych

W niniejszym rozdziale analizowany będzie fragment bazy zawierającej o dane o uczelni – dwie tabele, wydziały oraz kierunki prowadzone na wydziałach, połączone relacją jeden do wielu.

Rysunek 8.16. Tabele `kierunek` i `wydzial` we wspólnej relacji.

Najbardziej oczywistym sposobem pobrania danych z tabeli bazy danych jest skonstruowanie i wywołanie stosownego zapytania. Klasą opisującą zapytanie jest `QSqlQuery`. W konstruktorze jej obiektu należy podać połączenie, które ma być użyte do realizacji jego zapytań, lub pozostawić puste w przypadku ustanowienia połączenia domyślnego. Możliwe jest też podanie, w postaci napisu, zapytania w języku SQL, które ma zostać zrealizowane. Stworzenie obiektu typu `QSqlQuery` dla domyślnego połączenia może mieć postać:

```
QSqlQuery query("SELECT * FROM wydzial");
```

Wydanie komendy do bazy odbywa się za pomocą metody `exec`. Może pozostać ona bez argumentów. Wówczas wywołane zostanie ostatnie przygotowane zapytanie. Może również przyjąć napis będący zapytaniem.

Stąd zapisy:

```
QSqlQuery query("SELECT * FROM wydzial");
query.exec();
```

oraz

```
QSqlQuery query();
query.exec("SELECT * FROM wydzial");
```

są równoważne. Kolejnym sposobem przygotowania zapytania w obiekcie klasy `QSqlQuery` jest użycie metody `prepare`.

```
QStringList list;
list << "matematyka" << "fizyka" << "informatyka";
foreach(QString course, list) {
    query.prepare("INSERT INTO kierunek (nazwa, id_wydzial)"
        "VALUES (:nazwa, :wydzial)");
    query.bindValue(":nazwa", course);
    query.bindValue(":wydzial", "7");
}
```

Wygodna okazuje się medota `bindValue`, która wiąże ze sobą konkretne wartości z zastępczą wartością wpisaną po dwukropku. Zapytania można też, oczywiście, składać jak zwykłe napisy.

Wspomniana wcześniej metoda `exec`, w zależności od powodzenia wywołania zapytania zwróci wartość logiczną prawdy bądź fałszu. Zawsze warto sprawdzać poprawność wywołanej komendy. Połączenia sieciowe nie zawsze są stabilne i potwierdzenie da pewność, że dane faktycznie zostały przez bazę zarejestrowane. W czasie testów programu pomogą też odnaleźć błędy (np. składniowe) w tworzonych zapytaniach. Medota `exec` nie zwraca bezpośrednio wyników swojego działania. Są one zapisywane w obiekcie zapytania i można uzyskać do nich dostęp za pomocą metody `next`. Przegląda ona kolejne wiersze otrzymanego wyniku, a gdy dojdzie do końca, zwraca `NULL`. Można zbudować pętlę:

```
QSqlQuery query("SELECT * FROM wydzial");
while (query.next()) {
  qDebug()<<query.value(1).toString();
}
```

Wyświetli ona wszystkie nazwy wydziałów znajdujące się w tabeli. Zgodnie ze schematem 8.16 nazwa jest drugą kolumną w tabeli, więc posiada indeks równy 1. Po tym indeksie można do niej sięgnąć metodą `value`. Jako, że zwraca ona dane typu `QVariant`, konieczne jest przekształcenie ich na napis. Dodatkowo, po wykonaniu zapytania obiekt klasy `QSqlQuery` będzie posiadał

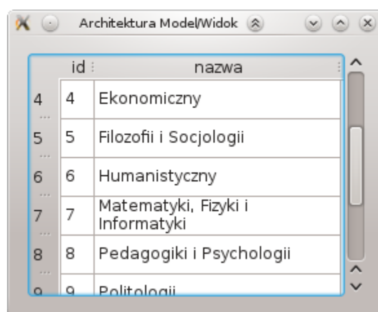
obiekt klasy `SqlRecord`, zwracany przez metodę `record`. Zawiera on m.in. informacje o liczbie kolumn i ich nazwie. Mając te wszystkie informacje, można by stworzyć model i przepisać tam wszystkie odczytane w pętli dane.

8.5.3. Model zapytania i tabeli

Aby odciążyć programistę od ręcznego przepisywania danych do tabeli powstała klasa `QSqlQueryModel`. Jej użycie ogranicza się do przekazania w metodzie `setQuery` zapytania w postaci napisu.

```
QSqlQueryModel *model = new QSqlQueryModel;
model->setQuery("SELECT * FROM wydzial");
QTableView *tableView=new QTableView(this);
tableView->setModel(model);
```

Sformułowane w ten sposób zapytania mogą być dowolnie skomplikowane i nie muszą odwoływać się do pojedynczej tablicy. Model zapytania nie pracuje bezpośrednio na bazie, a na kopii stworzonej w chwili przekazania danych programowi. Wszystkie zmiany wprowadzone w modelu przez programistę lub użytkownika pozostaną bez wpływu na bazę danych. Przykład działania takiego programu prezentuje rysunek 8.17.



Rysunek 8.17. Widok modelu zapytania o wszystkie elementy tablicy `wydzial`.

Możliwy jest również bardziej bezpośredni dostęp do tabeli bazy danych – poprzez model tabeli. Służy to tego klasa `QSqlTableModel`. W jej konstruktorze można zdecydować, które połączenie ma reprezentować dana tabela. Pominięcie tego argumentu spowoduje wykorzystanie domyślnego połączenia, o ile takie jest dostępne. Obiekt modelu tabeli SQL wymaga podania nazwy tej tabeli. Ostatnim krokiem jest faktyczne pobranie danych z bazy, za pomocą metody `select`. Fragment określający model należy zmodyfikować następująco względem poprzedniego przypadku. Warstwa widoku pozostaje bez zmian.

```
QSqlTableModel *model = new QSqlTableModel();
model->setTable("wydzial");
model->select();
```

Tak stworzony widget będzie miał wygląd jednakowy z widokiem modelu zapytania przedstawionym na rysunku 8.17.

W odróżnieniu od niego, za pomocą modelu tabeli możliwa będzie modyfikacja danych w tabeli. Postać delegata jest zależna od typu kolumny, której pole jest edytowane. Próba edycji elementu kolumny `id` typu całkowitoliczbowego spowoduje wywołanie edytora będącego obiektem klasy `QSpinBox`, zaś kolumny `nazwa` zawierającej napisy, obiektu klasy `QLineEdit`. Domyślnie, taka edycja spowoduje natychmiastową zmianę danej w bazie. Za pomocą metody `setEditStrategy` możliwa jest zmiana tego zachowania. Metoda ta przyjmuje trzy możliwe wartości typu wyliczeniowego `QSqlTableModel::EditStrategy`:

— `QSqlTableModel::OnFieldChange` – natychmiastowa aktualizacja w bazie danych,
— `QSqlTableModel::OnRowChange` – aktualizacja po zmianie aktywnego wiersza,
— `QSqlTableModel::OnManualSubmit` – wymaga ręcznego potwierdzenia.

W dowolnym momencie można zaktualizować wszystkie zmiany oczekujące na wprowadzenie do bazy metodą `submitAll`. Można je też wycofać wywołując metodę `revertAll`. W przypadku strategii `OnManualSubmit` jest to jedyny sposób zapisu danych w bazie.

Możliwe jest także filtrowanie oraz sortowanie wyników tabeli. Filtrowania dokonuje się z pomocą metody `setFilter`, przyjmującej napis zawierający warunek zapisany w składni języka SQL. Kod:

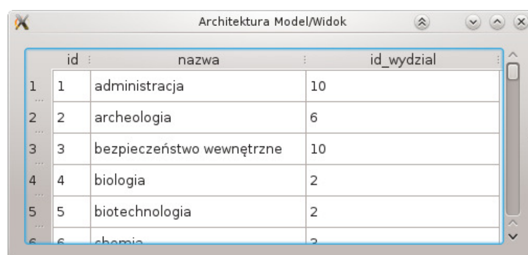
```
model->setFilter("nazwa ILIKE '%ma%'")
```

ograniczy wyświetlane wyniki, do wierszy, w których kolumna `nazwa` zawiera napis „ma”, bez uwzględnienia wielkości liter. Argument tej metody jest równoważny fragmentowi zapytania SQL, które pojawiłoby się po słowie `WHERE`. Z kolei ustawienie sortowania wymaga użycia zmiennej

8.5.4. Model tabeli relacyjnej

Klasa `QSqlTableModel` pozwala reprezentować dane z wyłącznie jednej tabeli. Jest to poważne ograniczenie, gdyż esencją pracy z relacyjnymi bazami jest właśnie wykorzystanie relacji między danymi, a kluczami obcymi wielu tabel. Tą lukę zapełnia klasa `QSqlRelationalTableModel`. Jako klasa dziedzicząca po `QSqlTableModel`, posiada tak samo działającą metodę `setTable`. Jej użycie dla tabeli `kierunek` spowoduje wyświetlenie jej w widoku tabeli

połączonym z modelem. Wszystkie dane zostaną wprost przepisane z bazy. Rezultat widoczny jest na rysunku 8.18.



Rysunek 8.18. Tabela `kierunek` w modelu klasy `QSqlRelationalTableModel`. Indeksy wydziałów są podane jawnie.

Klasa `QSqlRelationalTableModel` rozszerza `QSqlTableModel` o możliwość ustanowienia relacji między tabelami. Służy do tego metoda:

```
void setRelation (int column, const QSqlRelation &relation)
```

Argument `column` określa kolumnę w tabeli zawierającej indeks równoważny indeksowi obcej tabeli. Z kolei obiektowi klasy `QSqlRelation`, w konstruktorze, należy wskazać kolumny zewnętrznej tabeli:

```
QSqlRelation (const QString &tableName,
        const QString &indexColumn, const QString &displayColumn)
```

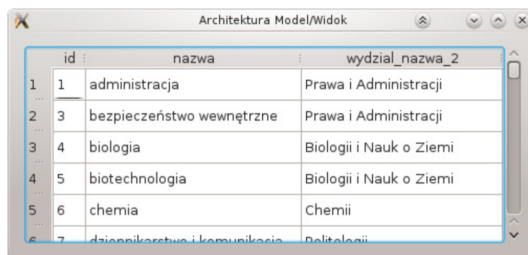
Zmienna `tableName` określa tabelę, z którą nawiązana zostanie relacja. `indexColumn` kolumnę tej tabeli, której wartości są równe z kolumną określoną zmienną `column` metody `setRelation` tabeli, na rzecz której ta metoda została wywołana. Ostatnia zmienna, `displayColumn` wskazuje kolumnę, której zawartością ma być zastąpiona kolumna określona przez `column`. Stworzenie relacji dla istniejącego modelu tabeli `kierunek` będzie wyglądać następująco:

```
model->setRelation(2,QSqlRelation("wydzial","id","nazwa"));
```

Dla kolumny o indeksie 2, tabeli `kierunek`, czyli `id_wydzial`, została stworzona relacja z tabelą `wydzial`. Dane z kolumny `id_wydzial` zostały zastąpione danymi z kolumny `nazwa` tabeli `wydzial` określonych indeksem `id` równym `id_wydzial`. Wynik jest równoważny wywołaniu zapytania:

```
SELECT kierunek.id, kierunek.nazwa, wydzial.nazwa
        FROM kierunek, wydzial
        WHERE kierunek.id_wydzial=wydzial.id;
```

Wynik działania przedstawia rysunek 8.19. Qt nie wymaga, by tabele, między którymi zachodzi relacja były połączone kluczem obcym.

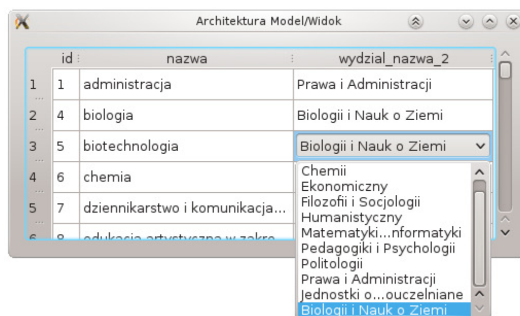


Rysunek 8.19. Indeksy wydziałów zostały zastąpione nazwami pochodzącymi z tabeli `wydzial`.

O ile, podobnie jak w przypadku modelu klasy `QSqlTableModel`, możliwa jest edycja danych pochodzących z tabeli określonej metodą `setTable`, o tyle próba edycji danych z kolumny relacyjnej skończy się powrotem do poprzedniego stanu. Rozwiązaniem tego problemu jest zastosowanie specjalizowanego delegata klasy `QSqlRelationalDelegate`. Do jego stworzenia wystarczy typowy konstruktor, przyjmujący jako rodzica widok, na którym jest osadzony. Należy go ustawić jako delegat tego widoku metodą `setItemDelegate`. Następująca implementacja będzie podobna dla wszystkich zastosowań tego delagata:

```
tableView->setItemDelegate(new QSqlRelationalDelegate(tableView));
```

Tak stworzony delegat pozostawia edytor klasy `QLineEdit` dla elementów lokalnej tabeli, a w przypadku kolumn połączonych relacją tworzy listę rozwijaną `QComboBox`, którą zapełnia wszystkimi wartościami obcej kolumny. Wynik działania został przedstawiony na rysunku 8.20



Rysunek 8.20. Delegat klasy `QSqlRelationalDelegate` wykorzystuje jako edytor listę rozwijaną.

Rozdział 9

Kontenery i algorytmy w Qt

9.1. Wstęp **142**
9.2. Kontenery Qt **142**
9.3. Kontenery sekwencyjne **143**
9.4. Kontenery asocjacyjne **148**
9.5. Iteratory STL **150**
9.6. Iteratory Qt **153**
9.7. Algorytmy Qt **159**

9.1. Wstęp

Dla Qt opracowano własne kontenerowe klasy, dzięki czemu użytkownik może w swoich programach pisanych w języku C++ wykorzystywać klasyczne kontenery STL a także kontenery Qt. Głównym argumentem za stosowaniem kontenerów Qt jest gwarancja, że wszystkie platformy i wersje Qt mają identyczne biblioteki oraz że współpraca z innymi elementami Qt jest zoptymalizowana. Dodatkowo Qt wprowadziło nowy typ iteratora wzorowany na koncepcji Javy, który według producentów nie generuje tak wiele problemów jak iterator STL. Koncepcja klas kontenerowych Qt spełnia wszystkie wymogi nałożone na klasy kontenerowe STL, oczywiście mamy do dyspozycji także iteratory i algorytmy dostosowane do obsługi kontenerów Qt. Użytkownik ma wybór, mówi się, że STL jest bardziej rozbudowany i jej elementy działają szybciej, natomiast klasy kontenerowe Qt są bardziej proste w użyciu.

9.2. Kontenery Qt

Kontenery Qt są zoptymalizowane ze względu na szybkość działania, zapotrzebowanie na pamięć, dają mniejszy kod wykonywalny. Przeglądanie kontenera wykonywać można przy pomocy dwóch typów iteratorów: iteratorów w stylu Javy oraz iteratorów w stylu STL. Twierdzi się , że iteratory typu Java są bardziej proste w użyciu, iteratory STL są bardziej wydajne. Iteratory STL mogą być wykorzystane w generycznych algorytmach Qt. Możemy także wykorzystywać konstrukcję *foreach* do łatwego przeglądania zawartości kontenerów.

Qt dostarcza następujące kontenery sekwencyjne: `Qlist`, `QLinkedList`, `QVector`, `QStack` i `QQueue`. Najczęściej wykorzystywany jest kontener `QList`. Pakiet Qt dostarcza następujące kontenery asocjacyjne: `QMap`, `QMultiMap`, `QHash`, `QMultiHash`, i `QSet`. Sa jeszcze specjalne klasy takie jak `QCache` oraz `QContiguousCache`. W tabeli 9.1 pokazano kontenery Qt i ich krótkie charakterystyki.

Kontenery mogą być zagnieżdżane. Przykładem może być następująca konstrukcja:

```
QMap<QString, QList<int> >
```

gdzie typem klucza jest typ `QString`, a typem wartości jest `Qlist<int>`. Należy zawsze pamiętać, aby pomiędzy ostatnimi znakami większości (> >) umieścić spację, bo inaczej spowoduje to błąd krytyczny. Wykorzystywanie kontenerów wymaga dołączenia plików nagłówkowych o tych samych nazwach co kontener. Na przykład do obsługi kontenera `QList` musimy dołączyć plik:

Tabela 9.1. Klasy kontenerowe Qt

Klasa Qt	Opis
QList<T>	Przechowuje listę wartości typu T, które są obsługiwane najczęściej przez indeks.
QLinkedList<T>	Przechowuje listę wartości typu T, które są obsługiwane najczęściej przez iterator.
QVector<T>	Przechowuje tablicę wartości.
QStack<T>	Jest to podklasa QVector do realizacji koncepcji LIFO. Posiada funkcje: push(), pop() i top().
QQueue<T>	Jest to podklasa QList do realizacji koncepcji FIFO. Posiada funkcje enqueue(), dequeue() i head().
QSet<T>	Kontener umożliwia obsługę zbiorów.
QMap<Key,T>	Kontener, którego elementami są pary klucz-wartość. Każdy klucz obsługuje jedną wartość.
QMultiMap<Key,T>	Jest to podklasa QMap. Klucz może obsługiwać wiele wartości.
QHash<Key,T>	Jest to szybszy odpowiednik QMap.
QMultiHash<Key,T>	Jest to podklasa QHash, dla realizacji złożonego haszowania (multi-valued hashes).

```
#include <QList>
```

Niektóre kontenery wymagają dostarczenia odpowiednich funkcji, dlatego zawsze należy sprawdzić dokumentację opisującą konkretną klasę kontenerową. Z drugiej strony, gdy żądane warunki nie są spełnione, kompilator wygeneruje komunikat błędu.

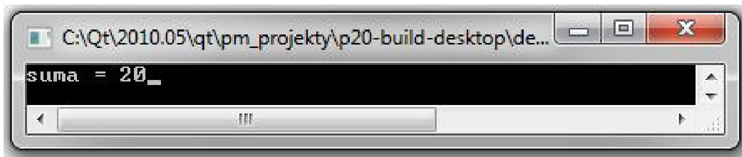
9.3. Kontenery sekwencyjne

Bardzo prostym kontenerem jest `QVector`. `QVector<T>` jest strukturą bardzo podobną do tablicy, umieszcza elementy w ciągłym fragmencie pamięci. Zasadnicza różnica między kontenerem `vector` i tablicą C++ jest fakt, że `vector` zna własny rozmiar i w miarę potrzeby może być powiększany aby pomieścić nowe elementy. Kolejny program demonstruje użycie kontenera `QVector`. W programie inicjalizujemy wektor wartościami i obliczamy ich sumę.

Listing 9.1. Kontener QVector

```
1 #include <QtCore/QCoreApplication>
2 #include <iostream>
3 #include <QVector>
4 using namespace std;
5
6 int main(int argc, char *argv[])
7 {   QCoreApplication a(argc, argv);
8     QVector<int> v1(10);
9     v1[0]=1;
10    v1[1]=2;
11    v1.append(2);
12    v1.append(4);
13    v1 << 5 << 6 ;
14    int sum = 0;
15    for(int i=0; i <v1.count(); ++i)
16        sum += v1[i];
17    cout << "suma = "<< sum;
18    return a.exec();
19 }
```

Wynik wykonania programu ma postać przedstawioną na rys. 9.1.



Rysunek 9.1. Wynik działania programu

W linii

```
QVector<int> v1(10);
```

deklarujemy wektor `v1` o rozmiarze 10, który będzie przechowywał elementy typu **int**. Gdy na początku wiemy ile elementów będzie przechowywał wektor, możemy ustalić jego rozmiar, w przeciwnym przypadku tego nie robimy. Kontener wektor obsługiwany jest przez operator indeksu `[ ]`.

W programie pokazano trzy możliwości inicjalizacji wektora `v1`:

```
v1[0]=1;
v1.append(2);
v1 << 5 << 6 ;
```

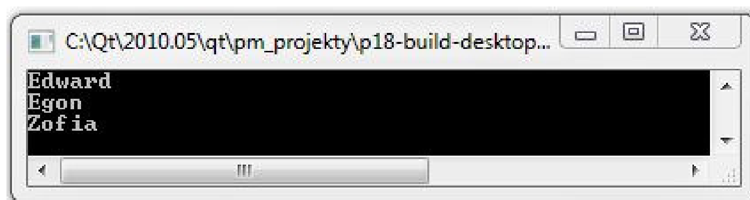
Funkcja `append()` umieszcza wartość na końcu wektora. Możemy użyć także przeciążonego operatora `<<` zamiast funkcji `append()`. W pętli for w

wyrażeniu warunkowym wykorzystaliśmy metodę `count()`. Zwraca ona liczbę wartości umieszczonych w kontenerze. Okazuje się, że umieszczanie elementów na początku wektora oraz usuwanie elementów ze środka wektora może być procesem mało wydajnym. W takim przypadku poleca się stosowanie kontenera `QLinkedList` lub `QList`. Obsługa kontenera `QList` jest także prosta. To zagadnienie ilustruje następny program. W programie wprowadzamy na listę trzy imiona, algorytm `qSort` sortuje listę.

Listing 9.2. Kontener QList

```
1 #include <QtCore/QCoreApplication>
2 #include <QTextStream>
3 int main(int argc, char *argv[])
4 {   QCoreApplication a(argc, argv);
5     QTextStream out(stdout);
6     QList<QString> list;
7     list <<"Edward"<<"Egon"<<"Zofia";
8     qSort(list);
9     for(int i =0; i<list.size(); ++i)
10        out << list.at(i) << endl;
11    return a.exec();
12 }
```

Po uruchomieniu programu otrzymujemy następujący wynik:



Rysunek 9.2. Wynik działania programu

W programie do obsługi tekstu na konsoli wykorzystaliśmy specjalną klasę Qt4 o nazwie `QTextStream`. Ta klasa dostarcza wygodny interfejs do odczytu i wydruku tekstu. Po włączeniu pliku:

```
#include <QTextStream>
```

definiujemy obiekt `out`:

```
QTextStream out(stdout);
```

dzięki któremu dane mogą być wprowadzone do standardowego wyjścia. W programie utworzono listę o nazwie `list` oraz wywołano algorytm sortujący `qSort()`:

```
list <<"Edward"<<"Egon"<<"Zofia";
qSort(list);
```

Formalnie należałoby do programu włączyć następujące linie:

```
#include <iostream>
#include <QList>
```

W naszej instalacji korzystamy z Qt 4.7.0 (32 bit) i do edytowania programu mamy Qt Creator 2.0.1, potrzebne narzędzia włączane są przez domniemanie (tak jak w pokazanym przykładzie plik `<QList>`). W pętli:

```
for(int i =0; i<list.size(); ++i)
        out << list.at(i) << endl;
```

drukujemy na ekranie uporządkowane elementy naszej listy. Metoda listy `at(i)` zwraca element o indeksie `i` w dopuszczalnym zakresie, tzn. `0 <= i < size()`.

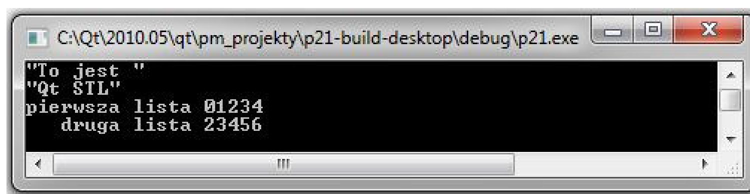
Kolejny przykład pokazujący obsługę kontenera `QList` jest bardziej skomplikowany, ponieważ wykorzystujemy kilka elementów typowych dla biblioteki Qt.

Listing 9.3. Kontener QList; QDebug

```
1 #include <QtCore/QCoreApplication>
2 #include <QDebug>
3 #include <iostream>
4 using namespace std;
5 int main(int argc, char *argv[])
6 {
7     QCoreApplication a(argc, argv);
8     QList<QString> list;
9     list <<"To jest "<<"Qt STL";
10    QListIterator<QString> i(list);
11    while(i.hasNext())
12        qDebug() << i.next();
13    QList<int>::iterator p;
14    QList<int> lst;
15    cout << "pierwsza lista ";
16    for (int k=0;k<5;++k)
17      {lst.push_back(k);
18       cout << lst[k];
19      }
20    cout << "\n   druga lista ";
21    for (p = lst.begin(); p != lst.end(); ++p)
```

```
22          *p += 2;
23      for (p = lst.begin(); p != lst.end(); ++p)
24          cout << *p;
25      return a.exec();
26  }
```

Wynikiem wykonania programu jest komunikat na ekranie monitora:



Rysunek 9.3. Wynik działania programu

W programie tworzymy listę o nazwie `list`, w której umieszczamy dwa napisy

```
QList<QString> list;
    list <<"To jest "<<"Qt STL";
```

W celu wydrukowania elementów listy wykorzystamy iterator Qt (styl Javy) oraz obsługę komunikatów Qt (qDebug):

```
QListIterator<QString> i(list);    // iterator styl Javy
while(i.hasNext())
   qDebug() << i.next();
```

Wykorzystana metoda `hasNext()`:

```
bool QListIterator::hasNext() const
```

zwraca **true** jeżeli znajduje się jeden element przed iteratorem, tzn. iterator nie jest na końcu kontenera, w przeciwnym przypadku zwraca **false**. Tego typu iterator (zwany iteratorem w stylu Javy będzie omówiony później). Metoda `next()`:

```
const T & QListIterator::next()
```

zwraca następny element i powiększa iterator o jedna pozycję.

Obsługą wydruku teksu zajmuje się funkcja `qDebug()`:

```
void qDebug(const char* msg, ......)
```

Formalnie wywołanie funkcji `qDebug()` skutkuje otrzymaniem obiektu `QDebug` użytecznego do zapisywania informacji w obsłudze błędów. W praktyce `qDebug` można używać zamiast obiektu `cout`. Proste wykorzystanie funkcji `qDebug()` ma postać:

```
int x = 10;
qDebug() << "nasz wynik = " << x;
```

W następnym fragmencie programu:

```
QList<int> lst;
cout << "pierwsza lista ";
for (int k=0;k<5;++k)
   {lst.push_back(k);
     cout << lst[k];
   }
```

tworzymy listę o nazwie `lst` do przechowywania elementów typu int. Umieszczanie elementów w kontenerze wykonuje metoda `push_back()`. Do obsługi elementów listy wykorzystujemy operator indeksowania. Korzystając ze zdefiniowanego iteratora `p` w stylu STL:

```
QList<int>::iterator p;
```

w kolejnym fragmencie następuje nadpisanie elementów listy:

```
for (p = lst.begin(); p != lst.end(); ++p)
    *p += 2;
for (p = lst.begin(); p != lst.end(); ++p)
    cout << *p;
```

Funkcja `begin()` zwraca iterator reprezentujący początek elementów w kontenerze, jest to pierwsza pozycja. Funkcja `end()` zwraca iterator reprezentujący koniec elementów w kontenerze. Jest to pozycja za ostatnim elementem.

9.4. Kontenery asocjacyjne

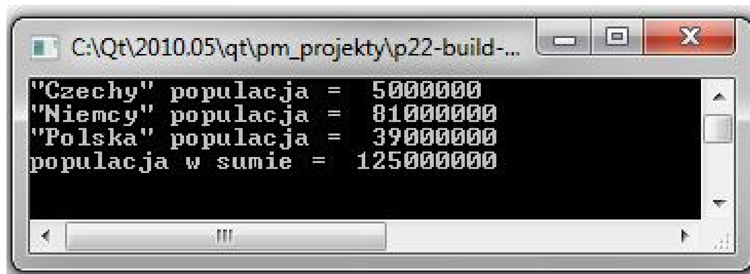
Kontenery asocjacyjne przechowują elementy tego samego typu indeksowane kluczem. Qt obsługuje dwa główne kontenery asocjacyjne: `QMap<K,T>` oraz `QHash<K,T>`.

Kontener `QMap` przechowuje uporządkowane pary klucz-wartość. Kolejny przykład ilustruje obsługę kontenera `QMap`. Program przechowuje listę państw (klucz) i ich populacje (wartość).

Listing 9.4. Kontener QMap

```
1 #include <QtCore/QCoreApplication>
2 #include <QMap>
3 #include <QDebug>3
4 int main(int argc, char *argv[])
5
6 {
7     QCoreApplication a(argc, argv);
8     int suma =0;
9     QMap<QString,int> kraj;
10    kraj["Polska"]= 39000000;
11    kraj["Niemcy"]= 81000000;
12    kraj["Czechy"]=  5000000;
13    QMapIterator<QString,int> i(kraj);
14    while(i.hasNext())
15     {i.next();
16      qDebug() << i.key()<<"populacja = "<< i.value();
17     }
18    i.toFront();
19    while(i.hasNext())
20        suma += i.next().value();
21    qDebug()<< "populacja w sumie = "<< suma;
22    return a.exec();
23 }
```

Wynikiem uruchomienia programu jest komunikat:



Rysunek 9.4. Wynik działania programu

W instrukcjach:

```
QMap<QString,int> kraj;
kraj["Polska"]= 39000000;
kraj["Niemcy"]= 81000000;
kraj["Czechy"]=  5000000;
```

tworzymy mapę o nazwie `kraj`. Program przechowuje pary klucz/wartość

pokazując kraj jego populację. Kluczem jest wartość typu `string`, a przechowywana wartość jest typu **int**.

Do inicjalizacji mapy można też wykorzystać metodę `insert()`:

```
kraj.insert("Polska",39000000);
kraj.insert("Niemcy",81000000);
kraj.insert("Czechy",5000000);
```

W celu wydrukowania zawartości mapy korzystamy z iteratora w stylu Javy:

```
QMapIterator<QString,int> i(kraj);
```

oraz metod `hasNext()` i `next()`:

```
    while(i.hasNext())
     {i.next();
      qDebug() << i.key()<<"populacja = "<< i.value();
     }
```

W celu wykonania ponownego przeglądania zawartości mapy musimy iterator ustawić na początku mapy. Należy w tym celu wykorzystać metodę `toFront()`

```
i.toFront();
```

Jeżeli zachodzi potrzeba utworzenia mapy z parami klucz/wartość, w których każdy z kluczy jest powiązany z kilkoma wartościami to należy użyć multimapy `QMultiMap<K,T>`.

9.5. Iteratory STL

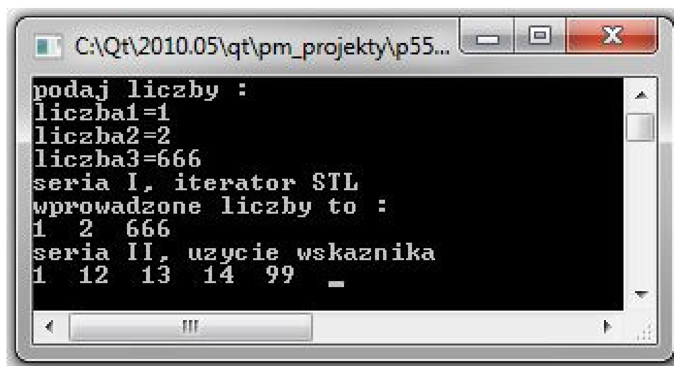
Przy pomocy iteratorów możemy prosto manipulować elementami w kontenerach STL, szczególnie przydatne są iteratory w obsłudze algorytmów STL. Iteratory STL działają bardzo podobnie do wskaźników. Dzięki temu można stosować algorytmy STL obsługiwane przez wskaźniki C++. To zagadnienie ilustruje kolejny program. W programie przypomniano także zastosowanie klasycznych iteratorów kontenera wektor.

Listing 9.5. Iteratory STL i wskaźniki

```
1 #include <QtCore/QCoreApplication>
2 #include <iostream>
3 #include <algorithm>
4 #include <vector>
5 using namespace std;
6 int main(int argc, char *argv[])
```

```
 7 {
 8     QCoreApplication a(argc, argv);
 9     vector<int> liczby(3);
10     cout <<"podaj liczby :"<<endl;
11     for(int i =0; i<3; i++)
12     {cout << "liczba" << i+1 << "=";
13         cin >> liczby[i];
14     }
15     cout << "seria I, iterator STL"<<endl;
16     vector<int>::iterator p;
17     cout << "wprowadzone liczby to :"<< endl;
18     p=liczby.begin();
19     while(p!=liczby.end())
20     {cout << *p << "  ";
21       p++;
22     }
23     cout << "\nseria II, uzycie wskaznika"<<endl;
24     int ar[5]={12,13,14,1,99};
25     int *begin = &ar[0];
26     int *end = &ar[5];
27     sort(begin,end);
28     for(int i=0; i<5; i++)
29         cout << ar[i]<<"  ";
30     return a.exec();
31 }
```

Wynikiem wykonania programu jest następujący komunikat:



Rysunek 9.5. Wynik działania programu

W programie tworzymy wektor liczb całkowitych i z klawiatury wprowadzamy trzy elementy. W celu wydrukowania wartości elementów tworzymy iterator `p` do kontenera `vector` i wykorzystujemy metody `begin()` i `end()`:

```
vector<int>::iterator p;   // iterator p
cout << "wprowadzone liczby to :"<< endl;
```

```
p=liczby.begin();  // ustawia  iterator  na  1
    elemencie
while(p!=liczby.end())   // metoda  end ()
{cout << *p << "  ";   // dereferencja  iteratora
  p++;
}
```

Następnie tworzymy tablicę elementów typu `int` i definiujemy dwa wskaźniki tej tablicy (`begin` i `end`), wykorzystujemy algorytm STL o nazwie `sort()` do uporządkowania elementów tablicy:

```
int ar[5]={12,13,14,1,99};
int *begin = &ar[0];
int *end = &ar[5];
sort(begin,end);
```

Widzimy, że algorytm STL może także być obsługiwany przez zwykle wskaźniki (nie tylko przez odpowiednie iteratory). Tabela 9.2 pokazuje podstawowe operacje wykonywane na iteratorach.

Tabela 9.2. Opis podstawowych operacji wykonywanych na iteratorach STL

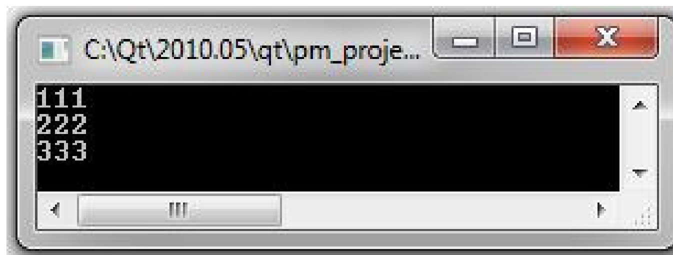
Operacja	Opis
*p	Zwraca wartość bieżącego element (dereferencja).
++p	Przemieszcza iterator do następnego elementu.
p+= n	Przesuwa iterator o n elementów.
– –p	Przesuwa iterator z powrotem o jeden element .
p –= n	Przesuwa iterator z powrotem o n elementów.
p – p1	Zwraca liczbę elementów pomiędzy p i p1.

Iteratory STL są bardzo dobre w obsłudze kontenerów Qt. W kolejnym programie wykorzystamy iteratory STL do obsługi listy Qt.

Listing 9.6. Kontener Qt; iteratory STL

```
1  #include <QtCore/QCoreApplication>
2  #include <QDebug>
3  int main(int argc, char *argv[])
4  {   QCoreApplication a(argc, argv);
5      QList<int> lista;
6      lista << 111 << 222 << 333;
7      QList<int>:: iterator p = lista.begin();
8      while(p!=lista.end())
9      {qDebug()<< *p;
10      ++p;
11     }
12     return a.exec();
13 }
```

Wynikiem program jest następujący wydruk:



Rysunek 9.6. Wynik działania programu

W programie utworzona została lista zawierająca trzy elementy typu **int**. Następnie został zdefiniowany iterator `p` w stylu STL dla kontenera `QList` i ustawiony na pierwszy element listy. W pętli **while** korzystając z dereferencji wskaźnika (`*p`) spowodowaliśmy wyprowadzenie wartości elementów na standardowe wyjście (ekran monitora):

```
QList<int>:: iterator p = lista.begin();
 while(p!=lista.end())
 {qDebug()<< *p;
  ++p;
 }
```

9.6. Iteratory Qt

W Qt 4 do obsługi kontenerów Qt opracowano nowy typ iteratora (dodatkowo do istniejących iteratorów STL) nazywany iteratorem w stylu Java (ang. *Java style iterators*). Ten typ iteratora jest standardem w obsłudze kontenerów Qt. Dla każdego kontenera mamy dwa typy iteratorów Qt : jeden służący tylko do odczytu i jeden służący do odczytu i zapisu. W tabeli 9.3 pokazane są kontenery i opracowane dla nich iteratory.

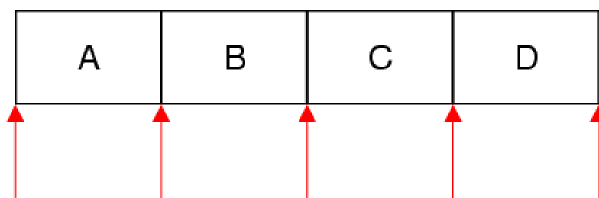
Iteratory Qt działają całkiem inaczej niż iteratory STL, wskazują na miejsce pomiędzy dwoma elementami a nie na konkretny element. Na rysunku 9.7 naszkicowana jest zasada działania iteratorów Qt.

Typowe przeglądanie elementów w kontenerze `QList` aby je wydrukować na konsoli ma postać (na podstawie opisu menu Help Qt):

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QListIterator<QString> i(list);
while (i.hasNext())
  qDebug() << i.next();
```

Tabela 9.3. Iteratory kontenerów Qt

Kontener Qt	Iterator-Odczyt	Iterator-Odczyt/zapis
QList<T>, QQueue<T>	QListIterator<T>	QMutableListIterator<T>
QLinkedList<T>	QLinkedListIterator<T>	QMutableLinkedListIterator<T>
QVector<T>, QStack<T>	QVectorIterator<T>	QMutableVectorIterator<T>
QSet<T>	QSetIterator<T>	QMutableSetIterator<T>
QMap<Key, T>, QMultiMap<Key, T>	QMapIterator<Key, T>	QMutableMapIterator<Key, T>
QHash<Key, T>, QMultiHash<Key, T>	QHashIterator<Key, T>	QMutableHashIterator<Key, T>



Rysunek 9.7. Pozycje na które może wskazywać iterator w kontenerze z 4 elementami. Pozycje wskazują strzałki.

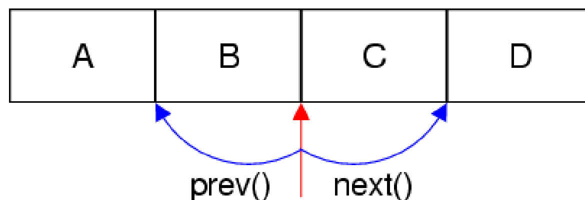
Po utworzeniu listy `list`, jest ona przekazywana do konstruktora `QListIterator`. Iterator `i` wskazuje na pozycję przed pierwszym elementem. Następnie wywoływana jest funkcja `hasNext()`, której zadaniem jest sprawdzenie, czy za tą pozycja jest element. Jeżeli tak, to następnie wywoływana jest metoda `next()`, która powoduje przesuniecie iteratora do pozycji za tym elementem. Funkcja `next()` zwraca wartość elementu ,który przeskoczyła. Funkcja `qDebug()` powoduje wydruk wartości elementu. Przeglądanie listy może odbywać się w dwóch kierunkach. Takie zadanie może realizować następujący fragment programu:

```
QListIterator<QString> i(list);
i.toBack();
while (i.hasPrevious())
  qDebug() << i.previous();
```

W tym fragmencie kodu, funkcja `toBack()` przesunie iterator na pozycję za ostatnim elementem kolekcji. Funkcja `hasPrevious()` sprawdza czy istnieje element na pozycji przed iteratorem a następnie jest wywołana funkcja `previous()`. Rysunek 9.8 ilustruje działanie funkcji `next()` i `previous()`.

W tabeli 9.4 pokazane są funkcje do obsługi iteratora Qt `QListIterator`.

Iteratory służące do wykonywania operacji związanych z odczytem i za-



Rysunek 9.8. Szkic działania funkcji next() i previous()

Tabela 9.4. Funkcje iteratorów Qt

Funkcja	Opis
toFront()	Przesuwa iterator na początek listy.
toBack()	Przesuwa iterator na koniec listy.
hasNext()	Zwraca true jeżeli iterator nie jest na końcu listy.
next()	Zwraca następny element i przesuwa iterator o jedną pozycję.
peekNext()	Zwraca następny element bez przesuwania iteratora.
hasPrevious()	Zwraca true jeżeli iterator nie jest na początku listy.
previous()	Zwraca poprzedzający element i przesuwa iterator z powrotem o jedną pozycję.
peekPrevious()	Zwraca poprzedni element bez przesuwania iteratora.

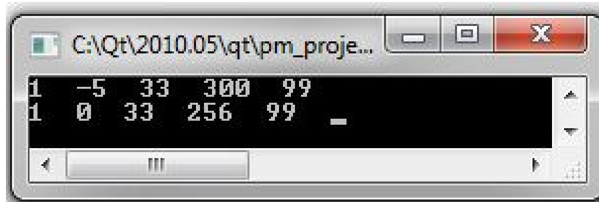
pisem mają specyfikator `Mutable` w nazwie (np. `QMutableListIterator<T>`). Tego typu iteratory umożliwiają wykonywanie operacji wstawiania, modyfikowania i usuwania elementów z kolekcji podczas jej przeglądania. Kolejny program ilustruje to zagadnienie. W programie tworzymy listę z różnymi elementami typu **int**, zadaniem programu jest wyselekcjonowanie wartości mniejszych niż 0 i zastąpieni ich wartością równą 0 oraz wyselekcjonowanie wartości większych niż 256 i zastąpienie ich wartością równą 256.

Listing 9.7. Kontener Qt; iteratory mutujące Qt

```
1 #include <QtCore/QCoreApplication>
2 #include<iostream>
3 using namespace std;
4 int main(int argc, char *argv[])
5 {   QCoreApplication a(argc, argv);
6     QList<int> z1;
7     z1 <<1<<-5<<33<<300<<99;
8     QMutableListIterator<int> p(z1);
9     while(p.hasNext())
10       cout  << p.next() << "  ";
11    cout << endl;
12    p.toFront();
```

```
13      while(p.hasNext())
14      { if(p.next() > 256)
15            p.setValue(256);
16        if (p.peekPrevious()< 0)
17             p.setValue(0);
18      }
19      p.toFront();
20      while(p.hasNext())
21          cout  << p.next() << "  ";
22      return a.exec();
23  }
```

Wynikiem tego programu jest komunikat na ekranie monitora:



Rysunek 9.9. Wynik działania programu

Na początku inicjalizujemy listę `z1` oraz deklarujemy mutujący iterator Qt o nazwie `p`, który będzie obsługiwał listę `z1` przechowującą elementy typu **int**:

```
QMutableListIterator<int> p(z1);
```

W pętli **while**:

```
while(p.hasNext())
{ if(p.next() > 256)
     p.setValue(256);
  if(p.peekPrevious()< 0)
     p.setValue(0);
}
```

Iterator kolejno przesuwa się od początku listy a za pomocą konstrukcji **if** sprawdzane są warunki. Jeżeli pierwszy warunek jest spełniony, funkcja `setValue(256)` wstawia żądaną wartość. W tym samym przebiegu pętli sprawdzamy też drugi warunek, musimy skorzystać z funkcji `peekPrevious()` aby zbadać element bieżący, ponieważ funkcja `next()` przesunęła iterator do przodu. Jeżeli drugi warunek jest spełniony, funkcja `setValue(0)` wstawia żądaną wartość. Należy pamiętać, że iterator zawsze znajduje się na ostat-

niej pozycji. W celu wydrukowania wartości elementów listy możemy użyć funkcji `previous()` lub przesunąć iterator na początek listy:

```
p.toFront();
```

W celu usuniecie żądanego elementu możemy wykorzystać funkcję `remove()`, na przykład w następujący sposób:

```
QMutableListIterator<int> i(lista);
i.toBack();
while (i.hasPrevious()) {
     if (i.previous() % 2 = 0)
     i.remove();
}
```

Iteratory kontenerów `QVector`, `QSet`, `QLinkedList` zachowują się tak samo jak iteratory `QList`. Zgodnie z oczekiwaniami, iteratory kontenera `QMap` będą zachowywały się inaczej ze względu na występowanie par klucz/wartość. Iteratory `QMap` obsługiwane są także funkcjami `toFront()`, `toBack()`, `hasNext()`, `next()`, `peekNext()`, `hasPrevious()`, `previous()`, i `peekPrevious()`. Do obsługi klucza potrzebujemy funkcji `key()`, a do obsługi wartości potrzebujemy funkcji `value()`. Te funkcje działają na obiektach zwróconych przez funkcje `next()`, `peekNext()`, `previous()` oraz `peekPrevious()`.

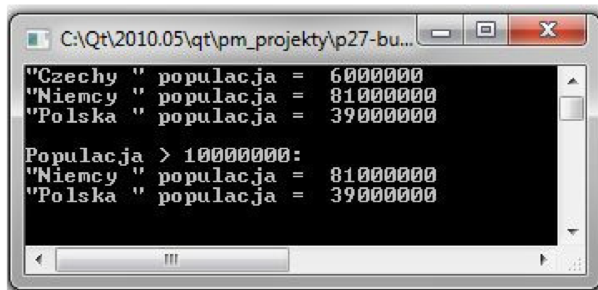
Zagadnienie obsługi kontenera `QMap` przy pomocy mutujących iteratorów ilustruje kolejny przykład. W pokazanym programie tworzymy kontener przechowujący pary kraj/populacja. Zadaniem programu jest usuniecie z mapy kraju, którego populacja jest mniejsza niż zadana wartość. W naszym programie wartością graniczną jest populacja 10 milionów.

Listing 9.8. Kontener QMap; iteratory mutujące

```
1 #include <QtCore/QCoreApplication>
2 #include <QDebug>
3
4 int main(int argc, char *argv[])
5 {
6     QCoreApplication a(argc, argv);
7     QMap<QString,int> kraj;
8     kraj["Polska "]= 39000000;
9     kraj["Niemcy "]= 81000000;
10    kraj["Czechy "]=  6000000;
11    QMutableMapIterator<QString,int> i(kraj);
12    while(i.hasNext())
13     {i.next();
14      qDebug() << i.key()<<"populacja = "<< i.value();
15     }
16     i.toFront();
17     while(i.hasNext())
```

```
18          {i.next();
19           if(i.value()<10000000)
20              i.remove();
21          }
22      qDebug()<<"\nPopulacja > 10000000:";
23      i.toFront();
24      while(i.hasNext())
25       {i.next();
26        qDebug() << i.key()<<"populacja = "<< i.value();
27       }
28      return a.exec();
29  }
```

Efektem wykonania programu jest następujący komunikat:



Rysunek 9.10. Wynik działania programu

Mutujący iterator kontenera `QMap` zadeklarowany jest następująco:

```
QMutableMapIterator<QString,int> i(kraj);
```

Po przesunięciu iteratora na początek mapy i korzystając z pętli **while** sprawdzamy ustalony warunek. Gdy populacja badanego kraju jest mniejsza niż zadana wartość, kraj jest usuwany z naszej mapy:

```
i.toFront();
while(i.hasNext())
{ i.next();
  if(i.value()<10000000)
    i.remove();
}
```

Do wydrukowania nowej mapy wykorzystujemy funkcje `key()` i `value()`:

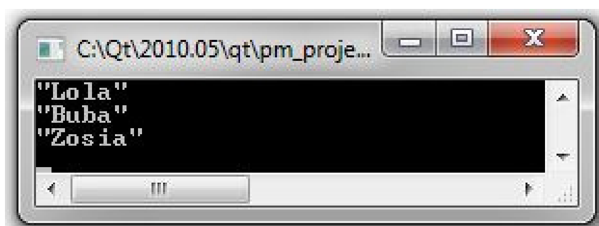
```
while(i.hasNext())
{i.next();
 qDebug() << i.key()<<"populacja = "<< i.value();
}
```

W Qt istnieje bardzo wygodna pętla do iterowania po elementach w kontenerach sekwencyjnych. Jest to pętla `foreach`. Krótki program ilustruje działanie tej pętli.

Listing 9.9. Kontener QList; pętla foreach

```
1 #include <QtCore/QCoreApplication>
2 #include <QDebug>
3 int main(int argc, char *argv[])
4 {
5     QCoreApplication a(argc, argv);
6     QList<QString> list;
7     list <<"Lola"<<"Buba"<<"Zosia";
8      QString str;
9      foreach (str, list)
10         qDebug() << str;
11     return a.exec();
12 }
```

Po uruchomieniu programu otrzymujemy następujący komunikat:



Rysunek 9.11. Wynik działania programu

Kod pętli `foreach` jest znacznie krótszy niż kod z użyciem iteratorów, pętla obsługuje wszystkie kontenery Qt.

9.7. Algorytmy Qt

W Qt możemy stosować klasyczne algorytmy STL (plik nagłówkowy `<algorithm>`) a także opracowane dla tej platformy własne algorytmy (umieszczone w pliku nagłówkowym `<QtAlgorithms>`). Sam producent Qt zauważa, że gdy nie ma specjalnej potrzeby rekomendowane jest używanie klasycznych algorytmów STL. Algorytmy Qt są globalnymi funkcjami szablonowymi stosowanymi do obsługi kontenerów i elementów kontenerów, większość algorytmów stosuje iteratory w stylu STL. Qt, jak to widać w tabeli 9.5 dostarcza ograniczoną ilość algorytmów. W tabeli 9.5 przedstawione są wszystkie algorytmy Qt z krótkimi opisami. W celu poprawnego

stosowania algorytmów Qt należy zapoznać się z dość dobrą dokumentacją techniczną zamieszczoną w menu pomocy “Help”.

Tabela 9.5. Algorytmy Qt, b – begin, e – end, v – wartość, c – kontener

Funkcja	Opis
qBinaryFind (b,e,v)	Wykonuje przeszukiwanie binarne.
qBinaryFind (b,e,v,less)	Wykonuje przeszukiwanie binarne.
qBinaryFind (c,v)	Wykonuje przeszukiwanie binarne.
qCopy (b1,e1,b2)	Kopiuje elementy.
qCopyBackward (b1,e1,e2)	Kopiuje elementy.
qCount (b,e,v,n)	Zwraca liczbę elementów (n) o wartości v.
qCount (c,v,n)	Zwraca liczbę elementów (n) o wartości v.
qDeleteAll (b,e)	Usuwa elementy z zakresu [b,e].
qDeleteAll (c)	Usuwa elementy z kontenera.
qEqual (b1,e1,b2)	Porównuje elementy z zakresów.
qFill (b,e,v)	Wypełnia zakres wartościami.
qFill (c,v)	Wypełnia kontener wartościami.
qFind (b,e,v))	Znajduje pierwszą wartość v.
qFind (c,v)	Znajduje pierwszą wartość v.
qGreater ()	Zwraca funktor do qSort().
qLess ()	Zwraca funktor do qSort().
qLowerBound (b,e,v)	Szuka pierwszej pozycji wartości v.
qLowerBound (b,e,v,less)	Szuka pierwszej pozycji wartości v.
qLowerBound (c,v)	Szuka pierwszej pozycji wartości v.
qSort (b,e) end)	Sortuje elementy w zakresie.
qSort (b,e,less)	Sortuje elementy w zakresie.
qSort (c)	Sortuje elementy w kontenerze.
qStableSort (b,e)	Sortuje elementy w zakresie.
qStableSort (b,e,less)	Sortuje elementy w zakresie.
qStableSort (c)	Sortuje elementy w kontenerze.
qSwap (v1,v2)	Wymienia elementy.
qUpperBound (b,e,v)	Szuka pozycji wartości v.
qUpperBound (b,e,v,less)	Szuka pozycji wartości v.
qUpperBound (c,v)	Szuka pozycji wartości v.

W menu pomocy zamieszczona jest dokumentacja techniczna oraz proste przykłady stosowania konkretnych algorytmów. Opisane algorytmy mogą być wykorzystane we wszystkich typach kontenerów, dla których dostępne są iteratory w stylu STL, włącznie z takimi kontenerami jak: `QList`, `QLinkedList`, `QVector`, `QMap` oraz `QHash`. Co ciekawsze, algorytmy generyczne mogą być wykorzystane poza kontenerami. Możliwe jest to dlatego, że iteratory STL są modelowane podobnie jak wskaźniki w C++. W tej sytuacji

możliwe jest wykorzystanie algorytmów STL i Qt z prostymi strukturami danych, takimi jak np. statyczne tablice. Do obsługi algorytmów potrzebujemy odpowiednich iteratorów.

Jak już pokazaliśmy, do dyspozycji mamy następujące typy iteratorów:
— Iteratory wejściowe (ang. *input iterators*)
— Iteratory wyjściowe (ang. *output iterators*)
— Iteratory postępujące (ang. *forward iterators*)
— Iteratory dwukierunkowe (ang. *bidirectional iterators*)
— Iteratory dostępu swobodnego (ang. *random access iterators*)
Najczęściej używane są iteratory dostępu swobodnego. Na tych iteratorach możemy wykonać operacje pokazane w tabeli 9.6.

Tabela 9.6. Dozwolone operacje na iteratorach dostępu swobodnego

Funkcja	Opis
i += n	Przesuwa w przód iterator o n pozycji.
i –= n	Przesuwa w tył iterator o n pozycji.
i + n lub n + i	Zwraca iterator na pozycji i+n lub n+I (i – iterator, n – liczba elementów).
i – n	Zwraca iterator na pozycji i–n.
i – j	Zwraca liczbę elementów pomiędzy iteratorem i oraz j.
i[n]	To samo co *(i + n).
i ¡ j	Zwraca true gdy iterator j jest za iteratorem i.

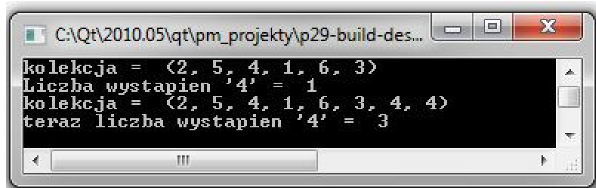
Stosowanie algorytmów Qt jest stosunkowo proste. Mają one jednak pewne specyficzne cechy, dlatego przed ich zastosowaniem najlepiej jest sprawdzić ich opis techniczny. Algorytm `qCount()` zwraca liczbę elementów o zadanej wartości `value` w kolekcji. Ilustruje to poniższy przykład.

Listing 9.10. Algorytm qCount()

```
1 #include <QtCore/QCoreApplication>
2 #include<QDebug>
3
4 int main(int argc, char *argv[])
5 {
6     QCoreApplication a(argc, argv);
7     QList<int> v;
8     int cn = 0;
9     v <<2<<5<<4<<1<<6<<3;
10    qDebug() << "kolekcja = " <<v;
11    qCount(v.begin(),v.end(),4,cn);
12    qDebug() <<"Liczba wystapien '4' = " << cn;
13    cn = 0;
14    v<<4<<4;
```

```
15     qDebug() << "kolekcja = " <<v;
16     qCount(v.begin(),v.end(),4,cn);
17     qDebug() <<"teraz liczba wystapien '4' = " << cn;
18  return a.exec();
19 }
```

Po uruchomieniu programu otrzymujemy następujący komunikat:



Rysunek 9.12. Wynik działania programu

Algorytm `qCount()` wystepuje w dwóch modyfikacjach:

```
void qCount(InputIterator Begin, Input Iterator end,
            const T & value, Size & n)

void qCount(const Container & container,
            const T & value, Size & n)
```

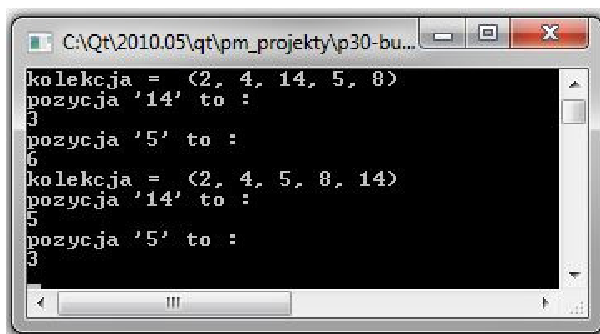
Pierwszą postać algorytmu wykorzystujemy, gdy chcemy przeglądać fragment kolekcji w zakresie `[begin, end]`, drugą postać – gdy chcemy przeglądać całą kolekcję. Należy pamiętać aby zadeklarować i zainicjalizować wartość `n`.

Listing 9.11. Algorytm qBinaryFind(); qSort()

```
1 #include <QtCore/QCoreApplication>
2 #include<QDebug>
3 int main(int argc, char *argv[])
4 {    QCoreApplication a(argc, argv);
5     QList<int> v;
6     v << 2 << 4<< 14<< 5<< 8;
7      qDebug() << "kolekcja = " << v;
8     QList<int>::iterator p;
9      p = qBinaryFind(v.begin(),v.end(), 14);  // error
10     qDebug() <<"pozycja '14' to :";
11     qDebug() << p - v.begin() + 1;
12     p = qBinaryFind(v.begin(),v.end(), 5);
13     qDebug() <<"pozycja '5' to :";
14     qDebug() << p - v.begin() + 1;
15    qSort(v);                    // poprawnie
16     qDebug() << "kolekcja = " << v;
17     p = qBinaryFind(v.begin(),v.end(), 14);
```

```
18      qDebug() <<"pozycja '14' to :";
19      qDebug() << p - v.begin() + 1;
20      p = qBinaryFind(v.begin(),v.end(), 5);
21      qDebug() <<"pozycja '5' to :";
22      qDebug() << p - v.begin() + 1;
23    return a.exec();
24 }
```

W kolejnym programie ilustrujemy sposób wykorzystania algorytmu `qBinaryFind()`. Wynik wykonania tego programu ma postać:



Rysunek 9.13. Wynik działania programu

Funkcja qBinaryFind() ma trzy realizacje:

```
RandomAccessIterator qBinaryFind(RandomAccessIterator begin,
                                 RandomAccessIterator end,
                                 const T & value)

RandomAccessIterator qBinaryFind(RandomAccessIterator begin,
                                 RandomAccessIterator end,
                                 const T & value,
                                 LessThan lessThan)

Container :: const_iterator qBinaryFind(const Container &
                                             container,
                                        const T & value)
```

Metodą przeszukiwania binarnego w przedziale `[begin, end]` algorytm szuka wyspecyfikowanej wartości `value` i zwraca pozycję znalezionej wartości. Gdy nie znajduje żadnej wartości, zwraca `end`. Jest ważne, aby przeszukiwany zbiór był uporządkowany wzrastająco (można to tego celu wykorzystać algorytm `qSort()`). Druga postać funkcji wykonuje wyszukiwanie w zbiorze posortowanym zgodnie z funktorem `lessThan`.

W pokazanym przykładzie przy pomocy zdefiniowanego iteratora `p` szukamy wartości "14" w nieuporządkowanym zbiorze:

```
QList<int>::iterator p;
p = qBinaryFind(v.begin(),v.end(), 14);  // error
qDebug() <<"pozycja '14' to :";
qDebug() << p - v.begin() + 1;
```

Wynik jest prawidłowy. Kolejne szukanie wartości "5" daje błędny wynik. Jest to spowodowane faktem przeglądania nieuporządkowanego zbioru. Prawidłowy wynik uzyskamy gdy uporządkujemy zbiór wartości algorytmem `qSort()`. Przy pomocy algorytmu `qCopy()` możemy kopiować wartości elementów jednego kontenera do innego kontenera. Algorytm przyjmuje trzy argumenty:

```
OutputIterator qCopy(InputIterator begin1,
                     InputIterator end1,
                     OutputIterator begin2)
```

Ten algorytm kopiuje element z zakresu `[begin1, end1]` do zakresu `[begin2,...]`. Bardzo użytecznym algorytmem jest algorytm `qEqual()`. Algorytm sprawdza czy elementy w wyspecyfikowanych zakresach są równe. Algorytm wymaga podania trzech argumentów:

```
bool qEqual (InputIterator1 begin1,
             InputIterator1 end1,
             InputIterator2 begin2)
```

Gdy elementy są równe zwraca wartość `true`, w przeciwnym przypadku wartość `false`. Kolejny program ilustruje wykorzystanie omówionych algorytmów.

Listing 9.12. Algorytm qCopy(); qEqual()

```
 1 #include <QtCore/QCoreApplication>
 2 #include<QStringList>
 3 #include<QDebug>
 4 int main(int argc, char *argv[])
 5 {   QCoreApplication a(argc, argv);
 6     QStringList m1;
 7     m1 << "Bach"<< "Vivaldi" << "Wagner"<< "Mozart";
 8     QVector<QString> m2(4);
 9     qCopy(m1.begin(),m1.end(),m2.begin());
10     qDebug()<< " " << m2;
11     m2[3] = "Brahms";
12     qDebug()<< " " << m2;
13     bool wynik;
14     wynik = qEqual(m1.begin(),m1.begin()+2,m2.begin());
```

```
15      qDebug()<<"3 pierwsze rowne ? " << wynik;
16      wynik = qEqual(m1.begin(),m1.end(),m2.begin());
17      qDebug()<<"wszystkie rowne ?  " << wynik;
18      return a.exec();
19  }
```

W programie tworzymu listę `m1` (dla typów `QString`) i inicjalizujemy ją nazwiskami znanych kompozytorów. Deklarujemy także wektor `m2` (także dla typów `QString`). Przy pomocy algorytmu `qCopy()` elementy listy `m1` kopiujemy do wektora `m2`:

```
QStringList m1;
m1 << "Bach"<< "Vivaldi" << "Wagner"<< "Mozart";
QVector<QString> m2(4);
qCopy(m1.begin(),m1.end(),m2.begin());
```

Czwarty element wektora jest zmieniony:

```
m2[3] = "Brahms";
```

W tym miejscu mamy nastepującą sytuację: w obu kontenerach trzy pierwsze pozycje są jednakowe. Przy pomocy algorytmu qEqual() sprawdzamy elementy w obu kontenerach:

```
bool wynik;
wynik = qEqual(m1.begin(),m1.begin()+2,m2.begin());
qDebug()<<"3 pierwsze rowne ? " << wynik;
wynik = qEqual(m1.begin(),m1.end(),m2.begin());
qDebug()<<"wszystkie rowne ?  " << wynik;
```

Po uruchomieniu programu mamy następujący wynik:

```
C:\Qt\2010.05\qt\pm_projekty\p31-build-desktop\debug\p31....
 QVector("Bach", "Vivaldi", "Wagner", "Mozart")
 QVector("Bach", "Vivaldi", "Wagner", "Brahms")
3 pierwsze rowne ?  true
wszystkie rowne ?   false
```

Rysunek 9.14. Wynik działania programu

Bibliografia

[1] Qt Reference Documentation, http://doc.qt.nokia.com/4.7/index.html.

[2] Fitzek Frank, Mikkonen Tommi, Torp Tony, *Qt for Symbian*, Wiley, 2010.

[3] Molkentin Daniel, *The Book of Qt 4: The Art of Building Qt Applications*, No Starch Press, 2007.

[4] Stęgierski Rafał, *Programowanie OpenGL*, Instytut Informatyki UMCS, Lublin 2011, http://informatyka.umcs.lublin.pl/files/stegierski.pdf.

[5] Summerfield Mark, *Advanced Qt Programming: Creating Great Software with C++ and Qt 4*, Prentice Hall, 2010.