Helion

PRENTICE HALL

NAJLEPSZE WZORCE DLA C++ I QT!

C++ i Qt

Wprowadzenie do wzorców projektowych

Wydanie II

Alan Ezust • Paul Ezust

Tytuł oryginału: An Introduction to Design Patterns in C++ with Qt (2nd Edition)

Tłumaczenie: Justyna Walkowska

ISBN: 978-83-246-8246-1

Authorized translation from the English language edition, entitled: Introduction to Design Patterns in C++ with Qt, Second Edition; ISBN 0132826453; by Alan Ezust and Paul Ezust; published by Pearson Education, Inc; publishing as Prentice Hall.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cppqtw
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

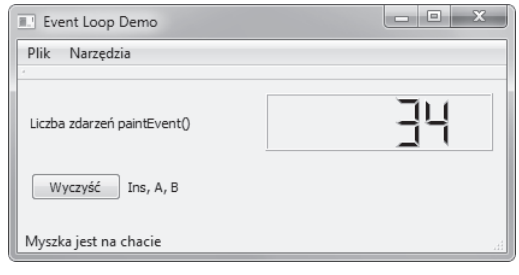
Spis treści

ROZDZIAŁ 10

Główne okna i akcje

Większość aplikacji `QApplication` opiera się na pojedynczym oknie `QMainWindow`. Jak pokazuje rysunek 10.1, `QMainWindow` zawiera elementy niezbędne w większości aplikacji desktopowych:

- centralny widżet,
- pasek menu,
- paski narzędzi,
- pasek stanu,
- obszary łączenia.



RYSUNEK 10.1. Główne okno

W większości aplikacji `QMainWindow` jest rodzicem (lub dziadkiem) obiektów `QAction`, `QWidget` i `QObject` na stercie. Często stosowaną praktyką jest rozszerzanie tej klasy, jak na przykładzie 10.1.

PRZYKŁAD 10.1. src/widgets/mainwindow/mymainwindow.h

```
[...]
class MyMainWindow : public QMainWindow {
    Q_OBJECT
 public:
```

```
    explicit MyMainWindow(QWidget* parent=0);
    void closeEvent(QCloseEvent* event);                    ///1

 protected slots:
    virtual void newFile();
    virtual void open();
    virtual bool save();
[...]
```

///1 Metoda nadpisana, by przechwycić zdarzenie zamknięcia okna przez użytkownika.

10.1. QAction, QMenu, QMenuBar

Klasa `QAction`, dziedzicząca z `QObject`, jest klasą bazową dla akcji wybranych przez użytkownika. Ma ona bardzo bogaty interfejs, o czym wkrótce się przekonasz. Interfejs `QWidget` pozwala każdemu widżetowi na przechowywanie listy `QList<QAction*>`.

Każdemu widżetowi może towarzyszyć zestaw akcji. Niektóre widżety wyświetlają listę akcji w menu kontekstowym, inne na pasku menu. Jeśli chcesz zapoznać się z kompletem informacji na temat tworzenia menu kontekstowych dla widżetów, zajrzyj do dokumentacji `setContextMenuPolicy()`.

Obiekty `QMenu` to również widżety `QWidget`, oferujące szczególny widok na kolekcję akcji `QAction`. Obiekt `QMenuBar` to zbiór menu, najczęściej spotykany wewnątrz `QMainWindow`.

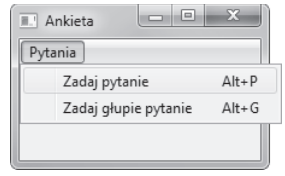
Jeśli rodzicem `QMenu` jest `QMenuBar`, `QMenu` jest widoczne jako menu rozwijane o znajomym interfejsie. Jeśli jego rodzicem nie jest `QMenuBar`, może ono wyskakiwać jak okno dialogowe — w takim przypadku jest nazywane **menu kontekstowym**[1]. Rodzicem `QMenu` może również być inne `QMenu`. Wówczas mamy do czynienia z **podmenu**.

W celu ułatwienia użytkownikowi podjęcia wyboru każda akcja `QAction` jest wyposażona w następujące elementy:

- tekst lub ikonka pojawiające się w menu lub na przycisku;
- akcelerator, czyli klawisz skrótu;
- podpowiedź „Co to jest?”;
- możliwość przestawiania stanu akcji pomiędzy następującymi parami właściwości: widoczna/niewidoczna, dostępna/niedostępna, zaznaczona/niezaznaczona;
- sygnały `changed()` (zmiana), `hovered()` (najechanie myszą), `toggled()` (zmiana właściwości), `triggered()` (uruchomienie).

Okno `QMainWindow` na rysunku 10.2 zawiera jeden pasek menu zawierający jedno menu z dwiema akcjami do wyboru.

[1] Menu kontekstowe najczęściej aktywuje się poprzez kliknięcie prawym przyciskiem myszy lub wciśnięcie przycisku menu. Nazywamy je *kontekstowym*, ponieważ jego zawartość zawsze jest zależna od kontekstu (który `QWidget` jest aktualnie zaznaczony lub aktywny).



RYSUNEK 10.2. QMenu

Przykład 10.2 przedstawia kod użyty do stworzenia paska narzędzi z rysunku 10.2. Funkcja `QMainWindow::menuBar()` zwraca wskaźnik na `QMenuBar` będący dzieckiem `QMainWindow`. Funkcja `menuBar()` tworzy i zwraca wskaźnik na pusty `QMenuBar`, jeśli pasek nie został jeszcze utworzony.

PRZYKŁAD 10.2. src/widgets/dialogs/messagebox/dialogs.cpp

```
[...]

    /* Umieść menu na pasku */
    QMenu* menu = new QMenu(tr("&Pytania"), this);

    QMainWindow::menuBar()->addMenu(menu);

    /* Kilka opcji w menu */
    menu->addAction(tr("&Zadaj pytanie"),
                    this, SLOT(askQuestion()), tr("Alt+Z"));
    menu->addAction(tr("Zadaj &głupie pytanie"),
                    this, SLOT(askDumbQuestion()), tr("Alt+G"));
}
```

Każde wywołanie `QMenu::addAction(text, target, slot, shortcut)` tworzy nienazwany obiekt `QAction` i dodaje go do `QMenu`. Następnie wywołuje ukrytą funkcję z klasy bazowej, `QWidget::addAction(QAction*)`. Dodaje ona nowo powstałą akcję do listy akcji, które zostaną użyte w menu kontekstowym. Akcja zostanie dodana do listy `QList<QAction*>` przechowywanej przez menu.

10.1.1. QAction, QToolbar, QActionGroup

Ponieważ aplikacje pozwalają użytkownikom na wydawanie tych samych komend na różne sposoby (np. poprzez menu, przyciski, skróty klawiszowe), enkapsulacja takiej komendy do postaci *akcji* jest dobrym sposobem zapewnienia spójnego zachowania w całej aplikacji. Akcje `QAction` mogą emitować sygnały i podłączać je do slotów, jeśli jest to potrzebne.

W aplikacjach Qt GUI akcje są z reguły „odpalane” na jeden z następujących sposobów:

- użytkownik klika opcję w menu,
- użytkownik wprowadza skrót klawiszowy,
- użytkownik klika przycisk na pasku narzędzi.

Istnieje kilka przeładowanych postaci funkcji `QMenu::addAction()`. W przykładzie 10.3 korzystamy z wersji odziedziczonej z `QWidget`, `addAction(QAction*)`.

Przykład 10.3 pokazuje, jak dodawać akcje do menu, grup akcji i pasków narzędzi. Na początku tworzymy klasę dziedziczącą z `QMainWindow` i dodajemy do niej kilka składowych `QAction`, a także grupę `QActionGroup` i pasek narzędzi `QToolBar`.

PRZYKŁAD 10.3. src/widgets/menus/study.h

```
[...]
class Study : public QMainWindow {
    Q_OBJECT
 public:
    explicit Study(QWidget* parent=0);
 public slots:
    void actionEvent(QAction* act);
 private:
    QActionGroup* actionGroup;                                    ///1
    QToolBar* toolbar;                                            ///2

    QAction* useTheForce;
    QAction* useTheDarkSide;
    QAction* studyWithObiWan;
    QAction* studyWithYoda;
    QAction* studyWithEmperor;
    QAction* fightYoda;
    QAction* fightDarthVader;
    QAction* fightObiWan;
    QAction* fightEmperor;
protected:
    QAction* addChoice(QString name, QString text);

};
[...]
```

///**1** Do przechwytywania sygnałów.

///**2** Do wyświetlania akcji jako przycisków.

Konstruktor naszej klasy tworzy wymagane menu i instaluje je wewnątrz paska `QMenuBar` będącego integralną częścią klasy bazowej. Odpowiedni kod został pokazany jako przykład 10.4.

PRZYKŁAD 10.4. src/widgets/menus/study.cpp

```
[...]

Study::Study(QWidget* parent) : QMainWindow(parent) {
    actionGroup = new QActionGroup(this);
    actionGroup->setExclusive(false);
    statusBar();

    QWidget::setWindowTitle( "zostać jedi chcesz?");              ///1

    QMenu* useMenu = new QMenu("&Użyj", this);
    QMenu* studyMenu = new QMenu("&Studiuj", this);
```

```
    QMenu* fightMenu = new QMenu("&Walcz", this);

    useTheForce = addChoice("useTheForce", "Użyj &Mocy");
    useTheForce->setStatusTip("To początek podróży...");
    useTheForce->setEnabled(true);
    useMenu->addAction(useTheForce);                                    ///2
[...]

    studyWithObiWan = addChoice("studyWithObiWan", "&Ucz się u Obi-Wana ");
    studyMenu->addAction(studyWithObiWan);
    studyWithObiWan->setStatusTip("Z pewnością otworzy przed Tobą "
                                  "wiele drzwi...");
    fightObiWan = addChoice("fightObiWan", "&Walcz z Obi-Wanem");
    fightMenu->addAction(fightObiWan);
    fightObiWan->setStatusTip("W ten sposób z pewnością "
                              "nauczysz się od niego wielu sztuczek!");
[...]

    QMainWindow::menuBar()->addMenu(useMenu);
    QMainWindow::menuBar()->addMenu(studyMenu);
    QMainWindow::menuBar()->addMenu(fightMenu);

    toolbar = new QToolBar("Pasek wyboru", this);                       ///3
    toolbar->addActions(actionGroup->actions());

    QMainWindow::addToolBar(Qt::LeftToolBarArea, toolbar);


    QObject::connect(actionGroup, SIGNAL(triggered(QAction*)),
            this, SLOT(actionEvent(QAction*)));                         ///4

    QWidget::move(300, 300);
    QWidget::resize(300, 300);
}
```

///1 Niektóre z użytych tu prefiksów *NazwaKlasy*:: nie są wymagane, ponieważ funkcje można wywołać na this. Nazw klas można użyć, by wywołać wersję funkcji zdefiniowaną w konkretnej nadklasie lub by pokazać czytelnikowi kodu, którą wersję wywołujemy.

///2 Akcja jest już w QActionGroup, ale dodajemy ją także do QMenu.

///3 W ten sposób uzyskujemy widoczne przyciski w dokowanym (przyłączanym) widżecie dla każdej z QAction.

///4 Zamiast podłączać sygnał każdej z akcji, łączymy się jedynie z zawierającą je wszystkie grupą ActionGroup.

Istnieje możliwość podłączenia poszczególnych sygnałów triggered() akcji do poszczególnych slotów. W przykładzie 10.5 grupujemy powiązane ze sobą akcje wewnątrz QAction ↪Group. Jeśli „odpalona” zostanie jedna z akcji z grupy, QAction wyemituje pojedynczy sygnał triggered(QAction*), który pozwala na obsługę wszystkich akcji w ujednolicony sposób. Sygnał przekazuje wskaźnik na konkretną odpaloną akcję, zatem można wybrać odpowiednią dla niej reakcję.

PRZYKŁAD 10.5. src/widgets/menus/study.cpp

```
[...]

// Funkcja wytwórcza (fabryka) produkująca akcje QAction inicjalizowane w ujednolicony sposób
QAction* Study::addChoice(QString name, QString text) {
    QAction* retval = new QAction(text, this);
    retval->setObjectName(name);
    retval->setEnabled(false);
    retval->setCheckable(true);
    actionGroup->addAction(retval);                                    ///1
    return retval;
}
```

///1 Dodaj każdą akcję do `QActionGroup`, dzięki czemu potrzebny będzie tylko jeden sygnał podłączony do jednego slotu.

Po stworzeniu akcji `QAction` jest ona dodawana (za pomocą `addAction()`) do trzech innych obiektów:

1. `QActionGroup` — w celu obsługi sygnałów.
2. `QMenu` — jednego z trzech menu rozwijanych na pasku `QMenuBar`.
3. `QToolBar` — gdzie zostanie udostępniona jako przycisk.

Dla urozmaicenia przykładu wprowadziliśmy pewne zależności logiczne pomiędzy opcjami w menu, odpowiadające fabułom kilku znanych filmów. Są one widoczne w funkcji `actionEvent()` z przykładu 10.6.

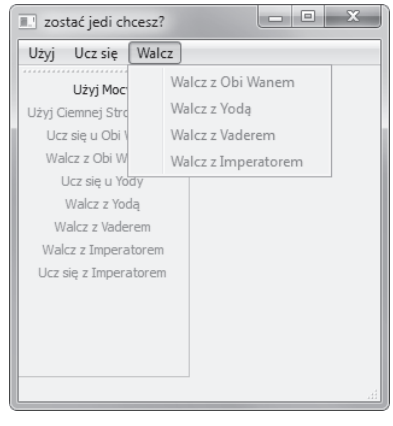
PRZYKŁAD 10.6. src/widgets/menus/study.cpp

```
[...]

void Study::actionEvent(QAction* act) {
   QString name = act->objectName();
   QString msg = QString();

   if (act == useTheForce ) { // użyj Mocy
       studyWithObiWan->setEnabled(true); // ucz się u Obi-Wana
       fightObiWan->setEnabled(true); // walcz z Obi-Wanem
       useTheDarkSide->setEnabled(true); // użyj ciemnej strony Mocy
   }
   if (act == useTheDarkSide) { // użyj ciemnej strony Mocy
       studyWithYoda->setEnabled(false); // ucz się u Yody
       fightYoda->setEnabled(true); // walcz z Yodą
       studyWithEmperor->setEnabled(true); // ucz się u Imperatora
       fightEmperor->setEnabled(true); // walcz z Imperatorem
       fightDarthVader->setEnabled(true); // walcz z Darthem Vaderem
   }
   if (act == studyWithObiWan) { // ucz się u Obi-Wana
       fightObiWan->setEnabled(true); // walcz z Obi-Wanem
       fightDarthVader->setEnabled(true); // walcz z Darthem Vaderem
       studyWithYoda->setEnabled(true); // ucz się u Yody
   }
[...]
```

```
    if (act == fightObiWan ) {
        if (studyWithEmperor->isChecked()) {
             msg = "Wygrałeś!";
        }
        else {
            msg = "Przegrałeś.";
            act->setChecked(false);
            studyWithYoda->setEnabled(false);
        }
    }
[...]

    if (msg != QString()) {
       QMessageBox::information(this, "Wynik", msg, "ok");
    }
}
```

Ponieważ wszystkie akcje należą do `QActionGroup`, sygnał `triggered(QAction*)` powoduje wywołanie `actionEvent()`.

Wszystkie opcje w menu poza jedną są początkowo nieaktywne. W miarę jak użytkownik wybiera spośród dostępnych opcji, inne stają się dostępne lub nie, co pokazuje rysunek 10.3. Zwróć uwagę na spójność pomiędzy przyciskami i opcjami w menu. Kliknięcie aktywnego przycisku powoduje zaznaczenie odpowiadającego mu elementu w menu. Akcja `QAction` przechowuje stan (dostępna/zaznaczona), a `QMenu` i `QToolBar` to widoki tej akcji.



RYSUNEK 10.3. Wybór akcji z menu i pasków narzędzi

10.1.2. Ćwiczenie: GUI dla gry w karty

Napisz grę oczko (blackjack). Możliwe są następujące akcje:

- nowa gra,
- rozdaj karty,
- tasuj,
- dobierz,

- zostań (nie dobieramy nowej karty),
- wyjdź.

Akcje powinny być dostępne poprzez pasek menu oraz przez pasek narzędzi. Aplikacja powinna przypominać tę przedstawioną na rysunku 10.4. Przejdźmy do zwięzłego omówienia reguł gry.

RYSUNEK 10.4. Zrzut ekranu gry w oczko

Na początku gry użytkownik oraz rozdający otrzymują po dwie karty. Użytkownik może zero lub więcej razy dobrać karty. Każde dobranie powoduje dodanie do ręki użytkownika jednej karty. Użytkownik kończy dobieranie, wykonując akcję Zostań.

Celem gry jest uzyskanie największej liczby punktów, przy czym nie może ona przekraczać 21.

Zasady ewaluacji ręki są następujące: figury (walet, królowa, król) mają po 10 punktów, as to 1 lub 11 — w zależności od tego, co akurat odpowiada graczowi. Pozostałe karty mają wartość punktową równą ich numerowi. Jeśli masz na ręce asa i waleta, lepiej jest policzyć asa za 11 punktów, co łącznie daje upragnione 21. Natomiast jeśli masz na ręce 7 i 8, a wyciągniesz asa, lepiej przypisać mu wartość 1.

Jeśli gracz przekroczy 21 punktów, automatycznie przegrywa i rozdanie dobiega końca.

Jeśli gracz ma na ręce pięć kart, których suma nie przekracza 21 — wygrywa.

Po tym jak gracz wygra, przegra lub zostanie (nie dobiera karty) rozdający dobiera dowolnie wiele kart tak długo, dopóki nie przekroczy liczby 18. Gdy to nastąpi, rozdający musi zostać, a gra się kończy. Wygrywa gracz, którego wynik jest bliższy (lecz nie większy niż) 21. Jeśli obaj gracze mają po tyle samo punktów, wygrywa rozdający.

Po zakończeniu rozdania użytkownik może wybierać jedynie spośród akcji: Rozdaj, Nowa gra i Wyjdź (Dobierz i Zostań są niedostępne).

Po wybraniu przez użytkownika akcji Rozdaj opcja ta staje się nieaktywna aż do zakończenia rozdania.

Aplikacja powinna pamiętać liczbę gier wygranych przez użytkownika i przez rozdającego, przyznając im jeden „duży" punkt za wygranie rozdania. Wyniki powinny być wyświetlane nad kartami gracza.

Rozdawaj karty bez resetowania talii do momentu, w którym karty się skończą lub użytkownik wybierze akcję Rozdaj.

Spróbuj wykorzystać w programie klasę `CardDeck` i inne klasy utworzone w ramach ćwiczeń i przykładów z sekcji 6.9.1. Dodaj do gry grafikę, wyświetlając wizualizacje każdej z kart, na przykład w sposób przedstawiony w podrozdziałach 9.5 i 9.6.

Stwórz menu rozwijane i pasek narzędzi z akcjami. Dobierz i Zostań mają być dostępne dopiero po rozdaniu kart.

Pokaż, ile kart zostało jeszcze w talii, korzystając z przewijanego pola `QSpinBox` w wersji tylko do odczytu.

Akcja Nowa gra powinna zerować zapamiętaną liczbę gier wygranych i przegranych przez graczy oraz resetować talię.

Sugestie projektowe

Staraj się rozdzielić klasy modelu od klas widoku — nie wprowadzaj kodu GUI do klas modelu. Rozdział modelu od widoku to dobra praktyka programistyczna, ułatwiająca utrzymywanie projektu.

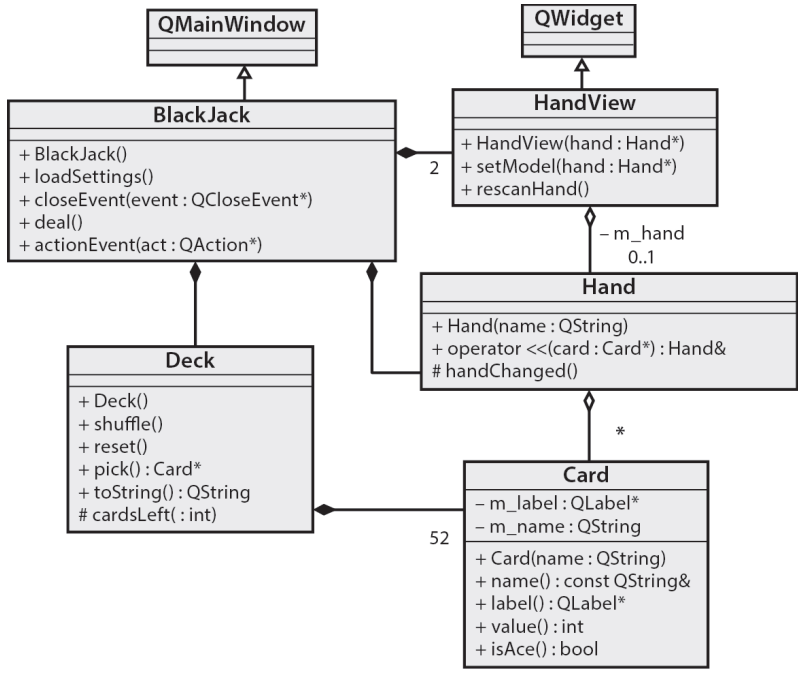
Rysunek 10.5 przedstawia jeden z możliwych projektów aplikacji.

10.2. Obszary i dokowanie

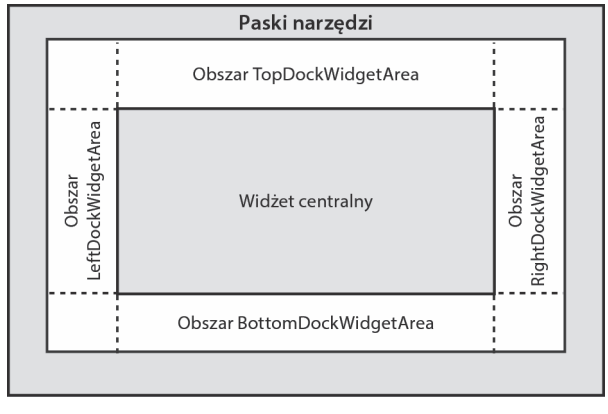
Każda klasa dziedzicząca z `QMainWindow` ma obszary dokowania okien, po jednym z każdej strony **widżetu centralnego**, jak pokazuje rysunek 10.6. Obszary te są używane do podłączania (**dokowania**) okien pomocniczych do widżetu centralnego.

O `QDockWidget` można myśleć jako o kopercie na inny widżet. Ma on pasek tytułu i obszar treści, w którym można umieścić inny widżet. W zależności od konfiguracji atrybutów `QDockWidget` może być odłączany (wówczas „pływa"), rozciągany, zamykany, przeciągany w inne miejsce lub podłączany do tego samego lub innego obszaru dokowania — na życzenie użytkownika końcowego. Dozwolona jest sytuacja, w której kilka takich widżetów zajmuje ten sam obszar dokowania.

Okno `QMainWindow` poprawnie tworzy przeciągalne obszary `QSplitter` pomiędzy widżetem centralnym a widżetami `QDockWidget`. Główne okno ma dwie podstawowe funkcje do zarządzania obszarami dokowania:



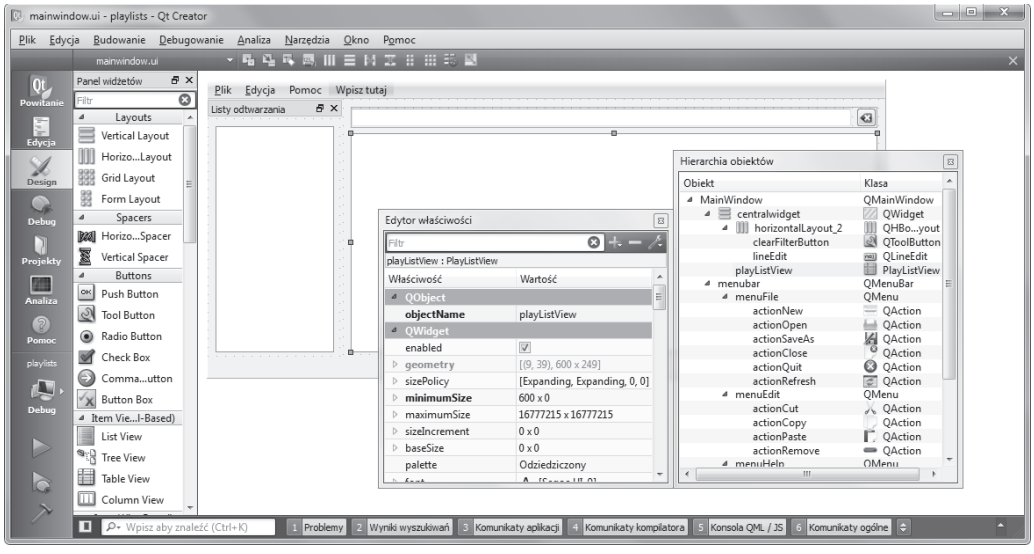
RYSUNEK 10.5. Diagram klas UML dla gry w oczko (klasy: BlackJack — oczko, Deck — talia, HandView — widok ręki, Hand — ręka, Card — karta)



RYSUNEK 10.6. Obszary dokowania QMainWindow

1. `setCentralWidget(QWidget*)` — ustawia widżet centralny;
2. `addDockWidget(Qt::DockWidgetAreas, QDockWidget*)` — dodaje podany `QDockWidget` do określonego obszaru dokowania.

Okna dokowane pełnią istotną rolę w zintegrowanych środowiskach programistycznych, ponieważ w różnych sytuacjach wymagane są różne zestawy narzędzi i widoki. Każde okno dokowane jest widżetem, który można dość łatwo „podpiąć" do głównego okna, jak pokazuje rysunek 10.7.



RYSUNEK 10.7. Pływające okna QDockWindow w Designerze

Jak większość aplikacji Qt, również Designer — narzędzie do projektowania i tworzenia interfejsów użytkownika — intensywnie korzysta z okien dokowanych. Designer składa się z Edytora widżetów (widżet centralny) i dokowanych widoków narzędzi takich jak:

- Panel widżetów,
- Inspektor obiektów,
- Edytor właściwości,
- Edytor sygnałów / slotów,
- Edytor akcji,
- Edytor zasobów.

Podłączane okna nie zawsze muszą być widoczne, dlatego można sterować ich wyświetlaniem za pomocą menu Widok (*View*). Jest ono zwracane przez funkcję `QMainWindow::create➥PopupMenu()`, co pozwala na odwoływanie się do niego z menu kontekstowych i dodawanie go do pasków narzędzi i menu rozwijanych.

10.3. QSettings: konfiguracja aplikacji

Większość aplikacji pozwala użytkownikom na konfigurację ustawień. Ustawienia (opcje, preferencje) powinny być trwałe. Właśnie w tym celu wprowadzono `QSettings`.

Obiekt `QSettings` (ustawienia) musi przed pierwszym użyciem zostać zainicjalizowany następującymi wartościami: nazwa aplikacji, nazwa organizacji, domena organizacji. Te wymagania chronią przed przypadkowym nadpisaniem ustawień jednej aplikacji przez ustawienia drugiej. Jeśli jednak informacje te są już w posiadaniu `QApplication`, jak w przykładzie 10.7, to konstruktor domyślny `QSettings` może ich użyć ponownie.

PRZYKŁAD 10.7. src/widgets/mainwindow/mainwindow-main.cpp

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

int main( int argc, char ** argv ) {
    QApplication app( argc, argv );
    app.setOrganizationName("objectlearning");
    app.setOrganizationDomain("objectlearning.net");
    app.setApplicationName("mainwindow-test");
    MyMainWindow mw;
    mw.show();
    return app.exec();
}
```

Po takiej inicjalizacji `QApplication` możliwe jest utworzenie instancji `QSettings` z wykorzystaniem konstruktora domyślnego.

Kiedy tworzysz aplikację opartą na `QMainWindow`, jednymi z pierwszych ustawień, które będziesz chciał pamiętać, będą rozmiar i położenie okna. Być może zechcesz również zapamiętać nazwy ostatnich dokumentów otwieranych przez aplikację.

Obiekt `QSettings` przechowuje trwałe mapy par klucz i wartość. Jest on także obiektem `QObject` i korzysta z rozwiązania przypominającego interfejs właściwości `QObject`: `setValue()` ustawia wartość `value`, a `value()` ją zwraca. Za jego pomocą można składować dowolne dane, które mają być pamiętane pomiędzy uruchomieniami aplikacji.

Obiekt `QSettings` wymaga podania nazwy organizacji i nazwy aplikacji, ale jest w stanie w konstruktorze domyślnym pobrać te dane od aplikacji. Każda kombinacja nazw definiuje unikatową trwałą mapę, która nie wchodzi w konflikt z ustawieniami aplikacji Qt o innych nazwach.

Wzorzec Monostate

Klasa, która pozwala wielu instancjom na współdzielenie tego samego stanu, jest implementacją **wzorca Monostate** (pojedynczy stan). Dwie instancje `QSettings` o tej samej nazwie organizacji i aplikacji mogą odwoływać się do tych samych trwałych danych. W ten sposób aplikacje mogą łatwo uzyskiwać dostęp do wspólnych ustawień z różnych plików źródłowych.

Klasa `QSettings` implementuje wzorzec Monostate.

Właściwy mechanizm trwałego przechowywania danych `QSettings` jest zależny od implementacji i dość elastyczny. Możliwe jest wykorzystanie rejestru Win32 (w Windows) i katalogu *$HOME/.settings* (w Linuksie).Więcej informacji znajdziesz w dokumentacji API klasy `QSettings`.

Funkcja `QMainWindow::saveState()` (zapisz stan) zwraca tablicę `QByteArray` zawierającą informacje o paskach narzędzi i widżetach podłączonych do głównego okna. Wykorzystana jest do tego właściwość `objectName` każdego z podwidżetów, dlatego istotne jest, by nazwy te były unikatowe. Funkcja `saveState()` ma opcjonalny parametr `int versionNumber`. Obiekt `QSettings` przechowuje tablicę `QByteArray` pod kluczem `"state"`.

Funkcja `QMainWindow::restoreState()` (przywróć stan) pobiera `QByteArray`, prawdopodobnie utworzoną przez `saveState()`, i w oparciu o zapisane w niej informacje odpowiednio rozmieszcza paski narzędzi i podłączone widżety. Ona również ma opcjonalny parametr `versionNumber`. Odczytaj ustawienia po skonstruowaniu obiektu, jak w przykładzie 10.8.

PRZYKŁAD 10.8. src/widgets/mainwindow/mymainwindow.cpp

```
[...]

void MyMainWindow::readSettings() {
    QSettings settings;
    QPoint pos = settings.value("pos", QPoint(200, 200)).toPoint();
    QSize size = settings.value("size", QSize(400, 400)).toSize();
    QByteArray state = settings.value("state", QByteArray())
                                                    .toByteArray();
    restoreState(state);
    resize(size);
    move(pos);
}
```

Ustawienia należy zapisać po tym, gdy użytkownik wyrazi chęć wyjścia z aplikacji, ale przed zamknięciem okna. Odpowiednim na to miejscem jest kod obsługi zdarzeń, gdyż można tam obsłużyć zdarzenie, zanim zrobi to sam widżet. Przykład 10.9 przedstawia funkcję obsługi zdarzenia zamknięcia okna, która zapisuje informacje konfiguracyjne w `QSettings`.

PRZYKŁAD 10.9. src/widgets/mainwindow/mymainwindow.cpp

```
[...]

void MyMainWindow::closeEvent(QCloseEvent* event) {
    if (maybeSave()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}


void MyMainWindow::writeSettings() {
    /* zapisz położenie i rozmiar głównego okna */
    QSettings settings;
    settings.setValue("pos", pos());
    settings.setValue("size", size());
    settings.setValue("state", saveState());
}
```

10.4. Schowek i transfer danych

Czasami niezbędne jest pobranie danych z pewnego miejsca i przeniesienie ich gdzie indziej. Można w tym celu zastosować metodę „kopiuj i wklej” (schowek) lub „przeciągnij i upuść”. W obu przypadkach wykorzystywane są te same klasy.

Każda aplikacja Qt ma dostęp do **schowka** systemowego poprzez wywołanie `qApp->clip` ↪`board()`. Schowek przechowuje dane określonego typu (tekst, grafika, URL, typy zdefiniowane przez użytkownika). Jeśli chcesz umieścić dane w schowku, stwórz obiekt `QMimeData`, zakoduj dane i wywołaj `QClipBoard->setMimeData()`. Do zapisywania danych poszczególnych typów służą następujące funkcje:

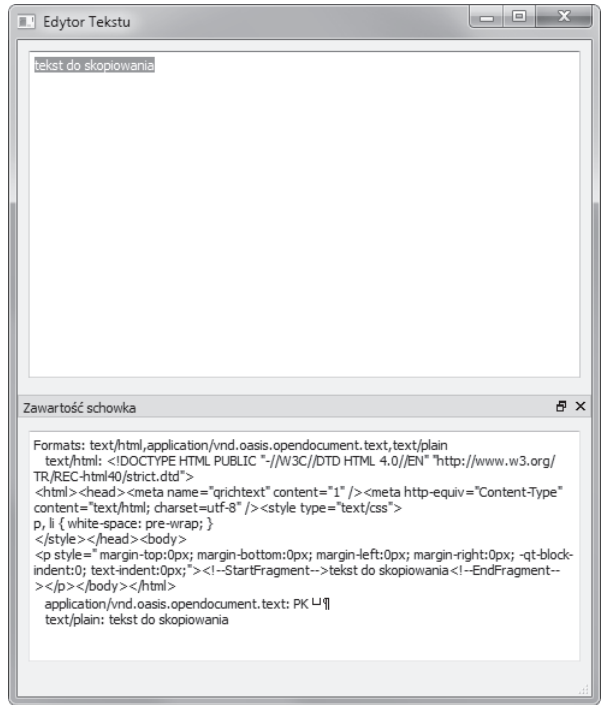
- `setText()` — dane tekstowe;
- `setHtml()` — tekst ze znacznikami;
- `setImageData()` — dane graficzne;
- `setUrls()` — lista adresów URL lub nazw plików.

W przykładzie 10.10 podłączamy sygnał `changed` (zmieniony) schowka systemowego do slotu `MainWindow clipboardChanged()`. Sygnał jest emitowany za każdym razem, gdy dowolna aplikacja skopiuje coś do schowka. Możesz uruchomić przykład i sprawdzić, jakie typy danych są dostępne podczas kopiowania danych pochodzących z `QTextEdit` lub z innych aplikacji.

PRZYKŁAD 10.10. src/clipboard/mainwindow.cpp

```
[...]
MainWindow::MainWindow(QWidget* parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    connect (qApp->clipboard(), SIGNAL(changed(QClipboard::Mode)),
             this, SLOT(clipboardChanged(QClipboard::Mode)));
}

void MainWindow::clipboardChanged(QClipboard::Mode) {
    QStringList sl;
    QClipboard *cb = qApp->clipboard();
    const QMimeData *md = cb->mimeData();
    sl << "Formaty: " + md->formats().join(",");
    foreach (QString format, md->formats()) {
        QByteArray ba = md->data(format);
        sl << "   " + format + ": " + QString(ba);
    }
    ui->clipboardContentsEdit->setText(sl.join("\n"));
}
[...]
```

Rysunek 10.8 pokazuje, co się dzieje, gdy tekst jest kopiowany z `QTextEdit`. W tym konkretnym przypadku dane w schowku zostały zakodowane na trzy sposoby: jako czysty tekst,



RYSUNEK 10.8. Demonstracja schowka systemowego

HTML oraz w otwartym formacie OASIS. Ostateczny wybór jest zależny od tego, jaki format danych jest oczekiwany w miejscu docelowym (tam gdzie tekst zostanie wklejony lub upuszczony).

10.5. Wzorzec Polecenie

Dodanie do programu akcji „cofnij" uspokaja użytkownika i zachęca go do eksperymentowania z możliwościami programu. Takie rozwiązanie może okazać się lepsze niż okienko wymuszające na użytkowniku zatwierdzenie wprowadzonych zmian. Qt oferuje gotowe klasy ułatwiające tworzenie aplikacji pozwalających na cofanie zmian.

Wzorzec Polecenie

Wzorzec Polecenie (ang. *Command*), opisany w [Gamma95], opakowuje operacje w postaci obiektów o wspólnym interfejsie wykonawczym. Taka architektura umożliwia ustawianie operacji w kolejce, logowanie operacji oraz cofnięcie wyników operacji już wykonanej. Poprawna implementacja tego wzorca może współdzielić kod obsługi błędów i sytuacji wyjątkowych.

Klasy Qt, które w mniejszym lub większym stopniu implementują ten wzorzec, to `QUndoCommand`, `QRunnable` i `QAction`.

Implementacja wzorca Polecenie w Qt jest wyjątkowo prosta:

- Możesz stworzyć kilka poleceń i ustawić je w kolejce w ramach odpowiedniego kontenera (np. kolejki `QQueue`).
- Polecenie cofnięcia `QUndoCommand` można wykonać, wkładając je na stos `QUndoStack`.
- Jeśli polecenia mają być wykonywane równolegle, możesz dodać dziedzicznie z `QRunnable` i zaplanować ich uruchomienie w ramach puli wątków za pomocą `QtConcurrent::Run()`[2].
- Możesz zapisać polecenia w pliku (serializacja) i odczytać je później (może nawet na innej maszynie). Przydaje się to do pracy wsadowej i rozproszonej[3].

10.5.1. Polecenie QUndoCommand i przetwarzanie obrazów

Kolejny przykład wiąże się z zastosowaniem polecenia cofnięcia `QUndoCommand`. Nasz przykładowy program służy do obróbki zdjęć[4]. Wykorzystuje on klasę `QImage`, niezależną od sprzętu reprezentację obrazu pozwalającą na manipulację na poziomie pojedynczych pikseli. Klasa `QImage` obsługuje wiele spośród najpopularniejszych formatów plików graficznych, w tym JPEG[5], stratny system kompresji, którego użyjemy na potrzeby przykładu.

Na początku, co pokazuje przykład 10.11, wprowadzimy kilka typowych operacji na obrazach. Wszystkie będą dziedziczyły z `QUndoCommand`. Pierwsza operacja zmienia kolor pikseli, mnożąc wartości składowych RGB (czerwony, zielony, niebieski) przez podane wartości `double`. Druga operacja zastępuje połowę obrazu lustrzanym odbiciem drugiej połowy, poziomo lub pionowo w zależności od wyboru użytkownika. Konstruktory obu operacji pobierają referencję do źródłowej grafiki `QImage` oraz tworzą pustą instancję `QImage` o tym samym rozmiarze i formacie.

PRZYKŁAD 10.11. src/undo-demo/image-manip.h

```
[...]
class AdjustColors : public QUndoCommand {
public:
   AdjustColors(QImage& img, double radj, double gadj, double badj)
     : m_Image(img), m_Saved(img.size(), img.format()), m_RedAdj(radj),
     m_GreenAdj(gadj), m_BlueAdj(badj)   {setText("dostosuj kolory"); }
   virtual void undo();
   virtual void redo();

private:
    QImage& m_Image;
    QImage m_Saved;
    double m_RedAdj, m_GreenAdj, m_BlueAdj;
```

[2] Wątkom poświęcony zostanie podrozdział 17.2.

[3] Wzorzec Serializator został omówiony w sekcji 7.4.1.

[4] Inspiracją dla przykładu były prace Guzdiala i Ericsona [Guzdial07] i ich projekt MediaComp.

[5] *http://jpeg.org/*

```
    void adjust(double radj, double gadj, double badj);

};

class MirrorPixels : public QUndoCommand {
public:
   virtual void undo();
   virtual void redo();
[...]
```

Każda z operacji tworzy kopię oryginalnej grafiki przed rozpoczęciem wprowadzania zmian. Przykład 10.12 przedstawia implementację operacji `AdjustColor` modyfikującej kolory. Jej konstruktor przechodzi przez wszystkie piksele `QImage` i wywołuje na każdym z nich funkcję `pixel()`. Funkcja ta zwraca kolor jako czwórkę ARGB zapisaną jako ośmiobajtowa wartość `int` (bez znaku) w postaci `AARRGGBB`, gdzie każda para bajtów reprezentuje odpowiednią składową koloru. Poszczególne składowe z czwórki (o aliasie `typedef QRgb`) możemy pobrać przy użyciu funkcji `qRed()` (czerwony), `qGreen()` (zielony) i `qBlue()` (niebieski). Wartości należą do przedziału od 0 do 255[6]. Następnie wartości RGB piksela zostają zastąpione wartościami zmodyfikowanymi przez mnożenie. Pamiętaj, że wynik mnożenia `double` i `int` zostanie zapisany w zmiennej typu `int`, co spowoduje jego obcięcie. Konsekwencją tego obcięcia jest nieodwracalność tej operacji — mnożenie wartości wynikowej przez odwrotność parametru nie przywróci stanu wyjściowego.

PRZYKŁAD 10.12. src/undo-demo/image-manip.cpp

```
[...]

void AdjustColors::adjust(double radj, double gadj, double badj) {
   int h(m_Image.height()), w(m_Image.width());
   int r, g, b;
   QRgb oldpix, newpix;
   m_Saved = m_Image.copy(QRect()); // zapisz kopię całego obrazka
   for(int y = 0; y < h; ++y) {
      for(int x = 0; x < w; ++x) {
         oldpix = m_Image.pixel(x,y);
         r = qRed(oldpix) * radj;
         g = qGreen(oldpix) * gadj;
         b = qBlue(oldpix) * badj;
         newpix = qRgb(r,g,b);
         m_Image.setPixel(x,y,newpix);
      }
   }
}

void AdjustColors::redo() {
   qDebug() << "AdjustColors::redo()";
   adjust(m_RedAdj, m_GreenAdj, m_BlueAdj);
}
```

[6] Czwarta para to komponent *alfa*, określający przezroczystość piksela. Jego domyślna wartość to `ff` (255) — nieprzezroczysty.

```
void AdjustColors::undo() {
    qDebug() << "AdjustColors::undo()";
    m_Image = m_Saved.copy(QRect());
}
```

Metoda `undo()` (cofnij) przywraca zapisaną kopię obrazu, a `redo()` (ponów) znów wywołuje funkcję modyfikującą piksele.

GUI zaprojektowaliśmy w Qt Creatorze. Grafika `QImage` jest konwertowana do postaci `QPixmap` i wyświetlana na ekranie wewnątrz `QLabel`. Rysunek 10.9 przedstawia oryginalne zdjęcie, przed jakimikolwiek przekształceniami.

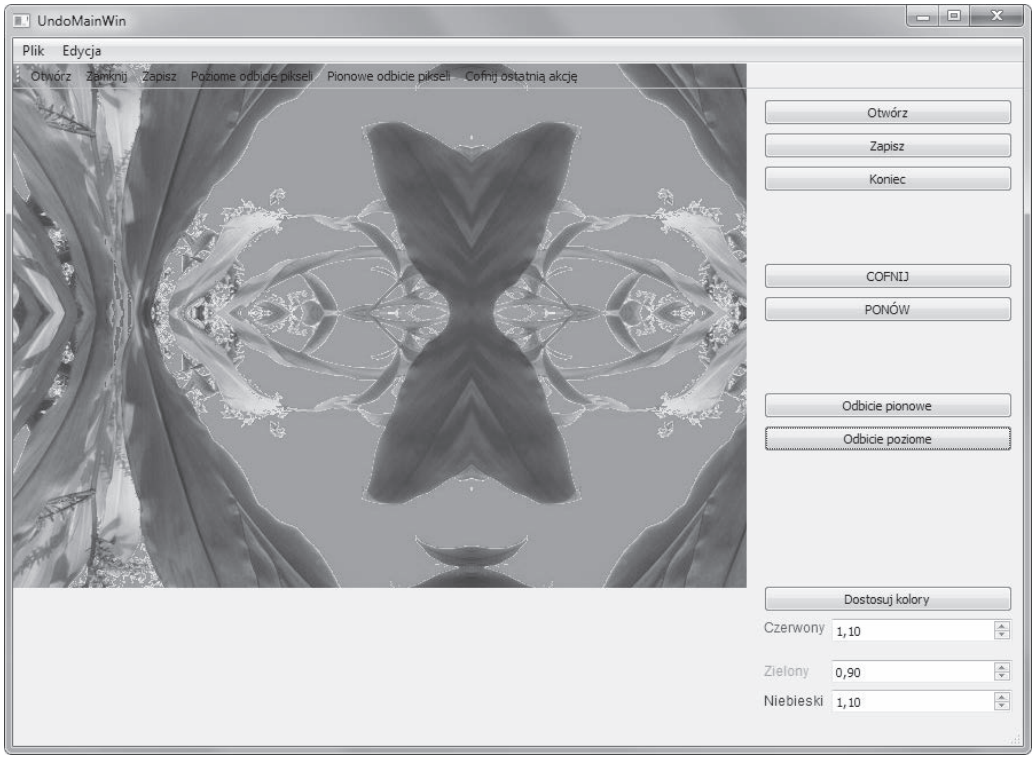
RYSUNEK 10.9. Oryginalna grafika

Rysunek 10.10 przedstawia przykry dla oka efekt operacji `AdjustColors` oraz obu wersji `MirrorPixels`.

Klasa `UndoMainWin` dziedziczy z `QMainWindow` i używa `QUndoStack`. Domyślnie Qt Creator osadza klasę *Ui* jako składową wskaźnikową `UndoMainWin`. W przykładzie 10.13 sloty prywatne początkowo były jedynie zalążkami wygenerowanymi przez Qt Creator, gdy w Designerze wybraliśmy opcję *Idź do slotu*.

PRZYKŁAD 10.13. src/undo-demo/undomainwin.h

```
#ifndef UNDOMAINWIN_H
#define UNDOMAINWIN_H

#include <QMainWindow>
#include <QUndoStack>
```



RYSUNEK 10.10. Grafika po modyfikacji

```
class QWidget;
class QLabel;
class QImage;
class QEvent;
namespace Ui {
    class UndoMainWin;
}

class UndoMainWin : public QMainWindow {
    Q_OBJECT
 public:
    explicit UndoMainWin(QWidget* parent = 0);
    ~UndoMainWin();

 public slots:
    void displayImage(const QImage& img);

 private:
    Ui::UndoMainWin* ui;
    QLabel* m_ImageDisplay;
    QImage m_Image;
    QUndoStack m_Stack;

private slots:
    void on_redoButton_clicked();
    void on_openButton_clicked();
    void on_actionAdjust_Color_triggered();
```

```
    void on_actionUndo_The_Last_Action_triggered();
    void on_actionHorizontal_Mirror_triggered();
    void on_actionVertical_Mirror_triggered();
    void on_actionQuit_triggered();
    void on_actionSave_triggered();
    void on_actionClose_triggered();
    void on_saveButton_clicked();
    void on_quitButton_clicked();
    void on_adjustColorButton_clicked();
    void on_undoButton_clicked();
    void on_verticalMirrorButton_clicked();
    void on_horizontalMirrorButton_clicked();
    void on_actionOpen_triggered();
};

#endif // UNDOMAINWIN_H
```

W przykładzie 10.14 masz okazję zaobserwować styl implementacji stosowany przez Qt Creator w celu połączenia ze sobą poszczególnych elementów. Zwróć uwagę na zwięzłe definicje prywatnych slotów z przykładu 10.13.

PRZYKŁAD 10.14. src/undo-demo/undomainwin.cpp

```
[...]
#include "image-manip.h"
#include "ui_undomainwin.h"
#include "undomainwin.h"

UndoMainWin::UndoMainWin(QWidget *parent)
: QMainWindow(parent), ui(new Ui::UndoMainWin),
  m_ImageDisplay(new QLabel(this)), m_Image(QImage()) {
  ui->setupUi(this);
  m_ImageDisplay->setMinimumSize(640,480);
}

UndoMainWin::~UndoMainWin() {
  delete ui;                                                    ///1
}

void UndoMainWin::displayImage(const QImage &img) {
    m_ImageDisplay->setPixmap(QPixmap::fromImage(img));
}

void UndoMainWin::on_actionOpen_triggered() {
    m_Image.load(QFileDialog::getOpenFileName());
    displayImage(m_Image);
}

void UndoMainWin::on_horizontalMirrorButton_clicked() {
    MirrorPixels* mp = new MirrorPixels(m_Image, true);
    m_Stack.push(mp);
    displayImage(m_Image);
}


void UndoMainWin::on_adjustColorButton_clicked() {
    double radj(ui->redSpin->value()), gadj(ui->greenSpin->value()),
```

```
    badj(ui->blueSpin->value());
    AdjustColors* ac = new AdjustColors(m_Image, radj, gadj, badj);
    m_Stack.push(ac);
    displayImage(m_Image);
}

[...]
```

///1 To nie `QObject` ani dziecko, więc musi zostać ręcznie usunięty.

Klasa `QImage` została zoptymalizowana pod kątem operacji na pikselach. Klasa `QPixmap` korzysta z pamięci wideo i jest używana przez widżety wyświetlające grafiki na ekranie. Jak wspomnieliśmy wcześniej, obraz `QImage` można przekształcić na `QPixmap` i wyświetlić na `QLabel`.

10.5.1.1. Ćwiczenia: polecenie QUndoCommand i przetwarzanie obrazów

Dodaj do kodu z przykładu 10.11 więcej operacji modyfikujących obraz. Oto kilka pomysłów:

1. **Obraz monochromatyczny**. Przekształć obraz zdefiniowany z użyciem składowych RGB do postaci monochromatycznej, opartej na odcieniach szarości. Kolor szary uzyskasz, nadając wszystkim trzem składowym koloru tę samą wartość. Niestety, jeśli ograniczysz się do ustawienia dla nowego piksela wartości średniej z trzech składowych, wynik okaże się zbyt ciemny. Standardowy sposób radzenia sobie z tym problemem opiera się na założeniu, że niebieski jest postrzegany jako „ciemniejszy" niż czerwony. Lepszy efekt uzyskasz, przemnażając wartości kolorów przez współczynniki wagowe[7]:

   ```
   czer *= 0.30; ziel *= 0.59; nieb *=0.11;
   ```

   Po tej operacji możesz już wyznaczyć luminancję piksela. Luminancja (natężenie oświetlenia) to wartość `int` równa ważonej średniej trzech składowych koloru. W naszym przypadku wagi już zostały nadane, pozostaje więc tylko wyliczenie średniej:

   ```
   luminancja = (czer + ziel + nieb) / 3
   ```

   Na koniec zastąp wartość każdej składowej nowo wyznaczoną wartością.
2. **Negatyw**. Przekształć obraz zdefiniowany z użyciem składowych RGB do postaci negatywu. Wystarczy zamienić wartość składowej `v` na `255 – v`.
3. **Zamiana kolorów**. Przesuń składowe koloru tak, by składowa czerwona otrzymała wartość niebieskiej, zielona czerwonej, a niebieska zielonej.
4. **Trzy kolory**. Wyznacz intensywność `ci` każdego piksela (w prosty sposób jako zwykłą średnią składowych). Jeśli wartość `ci` jest mniejsza niż 85, zmniejsz wartości

[7] Zagadnienie luminancji zostało omówione np. tutaj: *http://nemesis.lonestar.org/reference/internet/web/color/luminance.html*.

składowych czerwonej i niebieskiej do 0. Jeśli jest większa od 85, ale niższa od 170, wyzeruj składowe niebieską i zieloną. Jeśli `ci` to 170 lub więcej, wyzeruj czerwony i zielony.

5. **Wyodrębnienie krawędzi**. Porównaj natężenie każdego piksela z natężeniem piksela leżącego dokładnie pod nim. Jeśli wartość absolutna różnicy przekracza wyznaczoną (przekazaną jako argument) wartość progową, zmień kolor górnego piksela na czarny (wszystkie składowe mają wartość 0), a w przeciwnym razie — na biały (same wartości 255).

10.6. Umiędzynarodawianie i funkcja tr()

Gdyby Twój program miał zostać przetłumaczony na inny język (**umiędzynarodowiony**), mógłbyś liczyć na wsparcie Qt Linguist i narzędzi tłumaczeniowych Qt, które rozwiązują problemy związane z organizacją i składowaniem przetłumaczonych fraz. Możesz przygotować kod do tłumaczenia, otaczając każdą frazę do przetłumaczenia wywołaniem funkcji `QObject::tr()`. Gdy zostanie użyta w sposób niestatyczny, użyje nazwy klasy obiektu (pobranej z `QMetaObject`) jako kontekstu do grupowania przetłumaczonych fraz.

Funkcja `tr()` pełni dwie role:

1. Umożliwia narzędziu `lupdate` wydobycie wszystkich fraz, które mają być tłumaczone.
2. Jeśli dostępne jest tłumaczenie i wybrano język, zwraca przetłumaczoną frazę.

Jeśli nie podano tłumaczenia, funkcja zwróci oryginalny tekst.

UWAGA

Ważne jest, by każda fraza, która ma być tłumaczona, znajdowała się w całości wewnątrz funkcji `tr()` i by była dostępna na etapie kompilacji. W przypadku łańcuchów znaków z parametrami należy użyć funkcji `QString::arg()` w celu umieszczenia parametrów w wersji przetłumaczonej. Na przykład:

```
statusBar()->message(tr("Zakończono %1 z %2. Postęp:
%3%").arg(processed).arg(total).arg(percent));
```

Dzięki temu możliwa jest zmiana kolejności parametrów w tłumaczeniu, co bywa konieczne dla niektórych par języków.

Do tłumaczenia używane są następujące narzędzia:

1. `lupdate` — przeszukuje pliki *.ui* Designera i pliki C++ pod kątem fraz do przetłumaczenia. Generuje plik *.ts*.
2. `linguist` — edytuje pliki *.ts*, pozwala użytkownikowi na podanie tłumaczeń.
3. `lrelease` — czyta pliki *.ts* i na ich podstawie generuje pliki *.qm* używane przez aplikację do zastosowania tłumaczeń.

Więcej informacji na temat narzędzi do umiędzynarodawiania znajdziesz w dokumentacji[8].

10.7. Ćwiczenia: Główne okna i akcje

1. Napisz aplikację do edycji tekstu. Niech `QTextEdit` pełni rolę widżetu centralnego.
   - Tytuł okna ma pokazywać nazwę pliku oraz informację o niezapisanych zmianach.
   - Menu *Plik*: dodaj akcje *Otwórz*, *Zapisz jako* i *Zakończ*.
   - Menu *Pomoc*: dodaj akcje *O programie* i *O Qt*.
   - Jeśli są niezapisane zmiany, przed opuszczeniem programu poproś użytkownika o potwierdzenie.
2. Napisz aplikację, która umożliwia użytkownikowi otwarcie pliku tekstowego (lub pliku ze znacznikami) i przeglądanie oraz przewijanie jego zawartości. Widok powinien pokazywać co najmniej 10 linii naraz.

   Użytkownik może wyszukać w pliku interesujący go łańcuch znaków. Jeśli łańcuch zostanie odnaleziony, plik powinien zostać przewinięty do odpowiedniej linii, by użytkownik widział kontekst odnalezionego fragmentu. Jeśli plik nie zawiera danego tekstu, pokaż komunikat na pasku statusu.

   Daj użytkownikowi możliwość przechodzenia do następnych i poprzednich wystąpień szukanego łańcucha znaków.

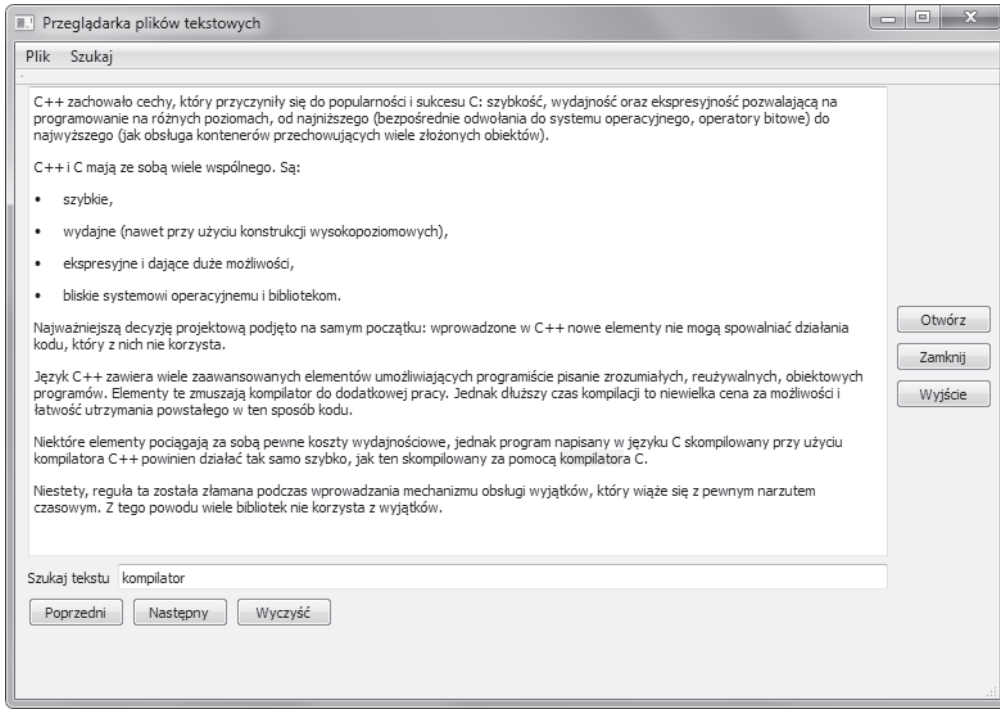
   Dodaj przycisk *Zamknij*, którego użycie usuwa z widoku dotychczas wyświetlany plik i zachęca użytkownika do otwarcia innego lub wyjścia z aplikacji.

   Rysunek 10.11 przedstawia zrzut ekranu z analogicznego programu. Dwa menu zawierają akcje duplikujące akcje dostępne w postaci przycisków. Wyjątkiem jest *Wyczyść*, która jedynie usuwa tekst z pola wejściowego.
3. Widżet `QTextEdit` ma metody pozwalające na cofanie i przywracanie. Przyjrzyj się im i znajdź odpowiedź na pytanie, co dokładnie można cofnąć lub przywrócić.

10.8. Powtórka

1. Jakie są główne funkcje `QMainWindow`?
2. Jak można zainstalować pasek `QMenuBar` w oknie `QMainWindow`?
3. W jaki sposób można zapisać, a później przywrócić rozmiar, położenie i układ widżetów w aplikacji GUI?
4. Czym jest widżet centralny?
5. Jak sprawić, by dany widżet został widżetem centralnym?
6. Do czego służą widżety dokowalne?

[8] *http://qt-project.org/doc/qt-5.1/qtdoc/index.html*



RYSUNEK 10.11. Przeglądarka plików tekstowych

7. Ile takich widżetów może mieć aplikacja?
8. Jak się ich używa?
9. Co to jest akcja?
10. Jak używać akcji?
11. Czym jest grupa akcji? Po co miałbyś dodawać ją do aplikacji?
12. Jak napisać aplikację dostępną dla użytkowników posługujących się różnymi językami?

Skorowidz

E

F

G

H

I

J

K

L

P

Q

R

S

Ś

T

U

W

X

Z

ź

Wykorzystaj moc duetu C++ i Qt!

Język C++ przez długie lata był liderem wśród języków programowania i wciąż jest rozwijany. Qt to zestaw bibliotek pozwalający na tworzenie atrakcyjnego interfejsu użytkownika, działającego w różnych systemach operacyjnych — Windows, Mac OS X oraz Linux. Projekt jest intensywnie rozwijany od 1992 roku, a najnowsza wersja pozwala nawet na tworzenie aplikacji internetowych i mobilnych. Wykorzystanie możliwości C++ oraz potencjału Qt może dać niezwykłe efekty!

W trakcie lektury nauczysz się podstaw języka C++ i zgłębisz tajniki bibliotek Qt. Zdobędziesz interesujące informacje na temat wykorzystania kontenerów, metaobiektów, metawłaściwości i mechanizmu refleksji. Dowiesz się, jak wykorzystać wzorzec MVC (ang. Model-View-Controller) oraz w jaki sposób opanować problemy związane z programowaniem współbieżnym. Ponadto nauczysz się panować nad wskaźnikami i unikać typowych problemów związanych z dostępem do pamięci. Książka ta jest doskonałą pozycją dla wszystkich programistów C++, którzy chcą wzbogacić swój warsztat o bibliotekę Qt.

Dzięki tej książce:

- poznasz podstawy C++ oraz Qt
- opanujesz programowanie współbieżne
- zrozumiesz wzorzec MVC
- zbudujesz interesującą aplikację przy użyciu C++ i Qt

ISBN 978-83-246-8246-1

9 788324 682461

cena: 89,00 zł