helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C++. Wykorzystaj potęgę aplikacji graficznych

Autor: Janusz Ganczarski,
Mariusz Owczarek
ISBN: 978-83-246-1473-8
Format: 158x235, stron: 448

Napisz wieloplatformowe programy w C++

- Jak korzystać z bibliotek wxWidgets oraz Qt?
- W jaki sposób implementować obsługę zdarzeń w aplikacjach?
- Jak budować aplikacje sieciowe i bazodanowe?

Okres dominacji jednego systemu operacyjnego i przeznaczonych dla niego rozwiązań powoli odchodzi do historii. Fenomen popularności różnych dystrybucji Linuksa i coraz mocniejsza pozycja komputerów Mac sprawiają, że wiele firm produkujących oprogramowanie decyduje się na tworzenie rozwiązań wieloplatformowych. W przypadku ogromnych korporacji stworzenie zespołów programistycznych pracujących równolegle nad kilkoma wersjami jednej aplikacji dla różnych systemów operacyjnych nie stanowi problemu, ale w mniejszych firmach jest to niemożliwe. Tu z pomocą przychodzą biblioteki pozwalające na tworzenie kodu źródłowego prawidłowo kompilującego się na każdej platformie, na której je zainstalowano.

Książka „C++. Wykorzystaj potęgę aplikacji graficznych" opisuje tworzenie oprogramowania z wykorzystaniem dwóch takich właśnie bibliotek – wxWidgets oraz Qt. Czytając ją, dowiesz się, jak wykorzystać język C++ i środowisko Dev-C++ do pisania programów, z których korzystać będą mogli użytkownicy systemu Windows, Linuksa i Mac OS. Nauczysz się stosować kontrolki i komponenty, budować menu i interfejsy użytkownika, obsługiwać zdarzenia i implementować operacje graficzne. Przeczytasz także o aplikacjach bazodanowych i sieciowych. Każde z zagadnień omówiono zarówno w kontekście biblioteki wxWidgets, jak i biblioteki Qt, dzięki czemu poznasz dwie metody rozwiązywania tych samych zadań programistycznych – by wybrać sobie tę, która bardziej Ci odpowiada.

- Instalacja środowiska programistycznego i bibliotek
- Struktura aplikacji i podstawowe komponenty
- Stosowanie komponentów
- Obsługa zdarzeń myszy i klawiatury
- Budowanie menu aplikacji
- Komunikacja sieciowa
- Operacje graficzne
- Połączenia z bazami danych
- Drukowanie z poziomu aplikacji

**Dzięki tej książce stworzysz aplikacje,
które docenią użytkownicy wszystkich systemów operacyjnych.**

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 12.

Struktura aplikacji Qt

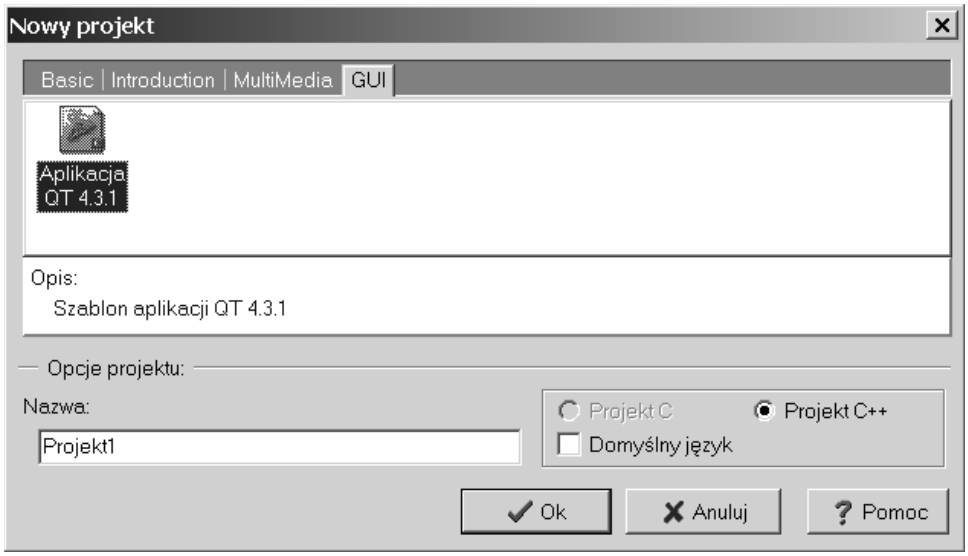
W pierwszym rozdziale poświęconym bibliotece Qt poznasz podstawowe zasady dotyczące struktury aplikacji w tejże bibliotece. Utworzymy główne okno aplikacji, umieścimy na nim pierwsze elementy graficznego interfejsu użytkownika (GUI), zmierzymy się z problemem polskich znaków, aby w końcu poznać podstawy obsługi zdarzeń w bibliotece Qt. Następnie po krótkim wprowadzeniu do Qt Designera wykonamy taką samą pracę ponownie, ale stosując to znakomite narzędzie z biblioteki Qt. Do tworzenia programów będziemy korzystać z szablonu opisanego w rozdziale 1.

Korzystamy z szablonu

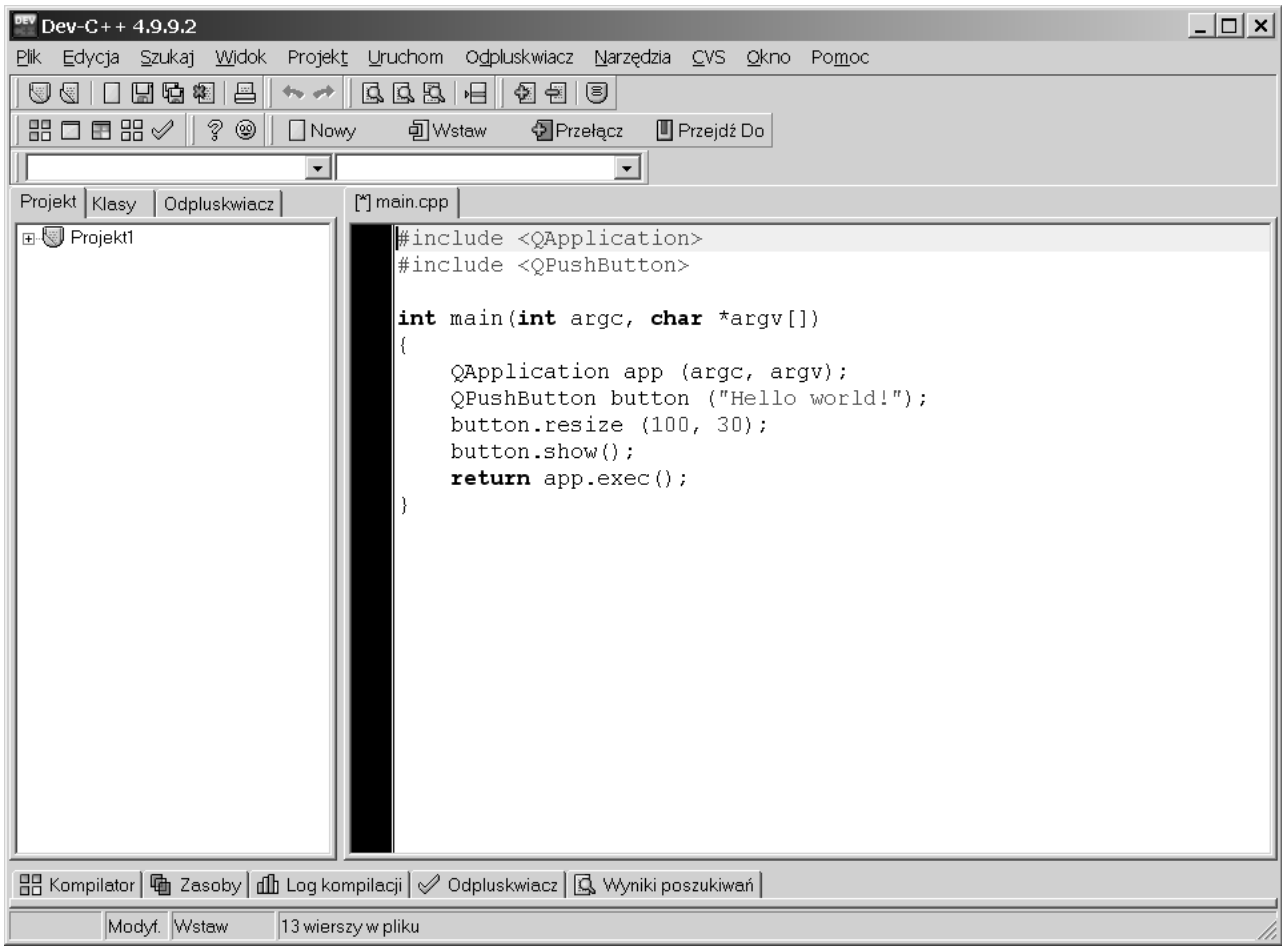
Przykład 12.1. Pierwszy program w Qt — wykorzystanie szablonu aplikacji

1. Aby skorzystać z szablonu, wybierz z menu opcję *Plik\Nowy Projekt* i dalej w zakładce *GUI* wskaż szablon aplikacji Qt, tak jak to przedstawiamy na rysunku 12.1. Do ewentualnej zmiany pozostaje nazwa projektu. Środowisko Dev-C++ przyjmuje, że nowy projekt nazywa się po prostu *Projekt* i do tej nazwy dodaje mu kolejny numer.
2. Po wybraniu szablonu pliki projektu zapisz w wybranym folderze i w efekcie powinieneś uzyskać program analogiczny do przedstawionego na rysunku 12.2. Musisz jeszcze zapisać plik źródłowy programu, ewentualnie zmieniając przy tym jego nazwę z domyślnego *main.cpp*.

   Warto wspomnieć jeszcze o jednej ważnej zasadzie obowiązującej przy zapisie projektów. Z uwagi na generowanie przy każdym projekcie pliku *Makefile.win* zawierającego instrukcje dla narzędzia `make`, koniecznie zapisuj pliki projektów w odrębnych katalogach. W przeciwnym wypadku może dojść do przypadkowej utraty zawartości tego pliku.



Rysunek 12.1. *Okno wyboru nowego projektu z szablonem aplikacji Qt 4.3.1*



Rysunek 12.2. *Widok środowiska Dev-C++ po wygenerowaniu szablonu programu Qt*

3. Oto tekst źródłowy całego programu:

```
#include <QApplication>
#include <QPushButton>

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

```
    QPushButton button ("Hello world!");
    button.resize (100, 30);
    button.show();
    return app.exec();
}
```

4. Kompilacja programu wymaga wybrania z menu opcji *Projekt\Kompiluj i uruchom* lub użycia klawisza *F9*. W efekcie otrzymamy okno przedstawione na rysunku 12.3. Program nawiązuje do klasycznej postaci pierwszego programu, który w swojej karierze napisało wielu programistów. Efekt jego działania wprawdzie nie jest imponujący, ale pokazuje potęgę biblioteki Qt, która w kilku wierszach kodu umożliwia wyświetlenie okna z przyciskiem.

Rysunek 12.3. *Początkowy wygląd okna pierwszego programu*

Pierwszy program krok po kroku

Przeanalizujemy teraz krok po kroku kod źródłowy programu wygenerowanego przez szablon. Dwa początkowe wiersze programu:

```
#include <QApplication>
#include <QPushButton>
```

włączają pliki nagłówkowe biblioteki Qt zawierające definicje klas `QApplication` i `QPushButton`. Pierwszej z nich używa każdy program, który korzysta z graficznych elementów biblioteki Qt. Jej zadaniem jest inicjalizacja i zakończenie działania programu, przetwarzanie pętli zdarzeń i umożliwienie ich obsługi przez elementy graficznego interfejsu. Klasa ta dziedziczy po klasie `QCoreApplication`, która z kolei jest klasą bazową dla programów pracujących w trybie konsoli (tekstowym). Klasa ta jest potomkiem klasy `QObject`, która jest bazą wszystkich klas w bibliotece Qt. Druga klasa wykorzystywana w programie reprezentuje przycisk. Warto zauważyć, że nazwa pliku nagłówkowego odpowiada nazwie danej klasy. Zasada ta obowiązuje dla każdej klasy biblioteki Qt należącej do tzw. publicznej części biblioteki i trzeba przyznać, że znakomicie ułatwia pracę.

Jak wyżej wspominaliśmy, `QApplication` jest bazową klasą dla wszystkich programów korzystających z biblioteki Qt, które pracują w trybie graficznym. Parametry użytego w programie konstruktora tej klasy:

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

odpowiadają standardowym argumentom funkcji `main`:

```
int main(int argc, char *argv[])
```

Także w ostatnim wierszu programu korzystamy z klasy `QApplication`:

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

Metoda `exec` przekazuje obsługę pętli zdarzeń, a tym samym dalsze sterowanie programem, do biblioteki Qt. Poza takimi wyjątkami jak okna komunikatów generowane przy użyciu klasy `QMessageBox`, wywołanie `exec` jest niezbędne do wyświetlenia graficznych elementów programu, czyli najczęściej całego interfejsu aplikacji.

Klasa `QPushButton` reprezentuje standardowy przycisk. Przycisk może, oprócz tekstu, zawierać także ikonę. W programie tworzony jest przycisk zawierający tekst wskazany jako parametr konstruktora klasy:

```
QPushButton button ("Hello world!");
```

Po utworzeniu przycisku zmieniamy jego wymiary tak, aby napis w nim zawarty był w całości widoczny w wyświetlanym oknie:

```
button.resize (100, 30);
```

Metoda `resize`, która to wykonuje, pochodzi z klasy `QWidget` będącej bazą wszystkich wizualnych elementów GUI w bibliotece Qt. Z tej samej klasy pochodzi metoda `show`, która wyświetla bieżącą kontrolkę i wszystkie elementy pochodne:

```
button.show();
```

Z obiektów klas dziedziczących pośrednio lub bezpośrednio po klasie `QWidget` będziemy wielokrotnie korzystać. Ich cechą charakterystyczną, którą przedstawimy jeszcze w tym rozdziale, jest zdolność zarówno do samodzielnej pracy jako okno aplikacji (jak ma to miejsce w pierwszym przykładzie), jak też jako jeden z elementów podrzędnych okna. W tym drugim przypadku za wyświetlenie całego okna odpowiedzialny jest obiekt będący najwyżej w hierarchii elementów okna. Warto także wiedzieć, że każdy obiekt klasy pochodnej po `QWidget` dziedziczy możliwość pełnej obsługi zdarzeń generowanych przez użytkownika, w tym — oczywiście — obsługę myszy i klawiatury.

Wyświetlamy polskie znaki

Niezbędną cechą programów jest poprawna obsługa znaków charakterystycznych dla języka polskiego. Niestety, prosta zamiana napisu na przycisku z pierwszego przykładu z angielskiego *Hello world* na polski odpowiednik *Witaj Świecie* da efekt przedstawiony na rysunku 12.4, na którym widoczny jest brak polskich liter.

Rysunek 12.4.
Przykład nieprawidłowego kodowania polskich znaków

Brak polskich znaków spowodowany jest sposobem obsługi napisów przez bibliotekę Qt, a szczególnie metodą konwersji znaków. Napisy w bibliotece Qt obsługuje specjalizowana klasa `QString` (przedstawimy ją bliżej w następnym rozdziale), która przechowuje dane, korzystając ze standardu Unicode 4.0. Jednak przy konwersji napisów klasa ta domyślnie używa standardu Latin 1, znanego także jako norma ISO 8859-1, który nie zawiera polskich znaków.

Zmiana domyślnego standardu kodowania wymaga użycia statycznej metody `setCodec ➥ForCStrings` klasy `QTextCodec:`, której parametrem jest nazwa wybranego standardu kodowania znaków. W przypadku języka polskiego najważniejsze są dwa standardy kodowania znaków: Windows-1250 stosowany w systemach z rodziny Microsoft Windows oraz ISO 8859-2 używany w systemach Linux/Unix. Wybór jednego z tych dwóch standardów kodowania wygląda następująco:

```
QTextCodec::setCodecForCStrings (QTextCodec::codecForName ("Windows-1250"));
QTextCodec::setCodecForCStrings (QTextCodec::codecForName ("ISO-8859-2"));
```

Biblioteka Qt obsługuje wiele standardów kodowania znaków, oprócz tak ważnych standardów jak ISO z rodziny 8859 (poza alfabetem tajskim) oraz standardów systemów z rodziny Windows, Qt obsługuje także standard Unicode w wersjach kodowania UTF-8 i UTF-16. Dzięki tak obszernej liczbie obsługiwanych standardów kodowania znaków biblioteka Qt umożliwia tworzenie programów wykorzystujących wszystkie najważniejsze języki używane przez ludzkość. Pełny wykaz obsługiwanych standardów kodowania znaków z ewentualnymi ograniczeniami dotyczącymi niektórych platform znajduje się w dokumentacji biblioteki.

Powyższą wiedzę wykorzystamy w kolejnym przykładzie.

Przykład 12.2. Wyświetlenie polskich znaków w bibliotece Qt

1. Utwórz nowy projekt, korzystając z szablonu aplikacji Qt, tak jak w poprzednim przykładzie.
2. Listę plików nagłówkowych uzupełniamy tak, aby można było użyć klasy `QTextCodec`:

   ```
   #include <QTextCodec>
   ```

3. Wybór standardu kodowania polskich znaków najlepiej umieścić bezpośrednio po utworzeniu obiektu `app`:

   ```
   QApplication app (argc, argv);
   QTextCodec::setCodecForCStrings (QTextCodec::codecForName ("Windows-1250"));
   ```

4. W następnym wierszu programu możemy już wpisać nowy tekst przycisku:

   ```
   QPushButton button ("Witaj Świecie!");
   ```

5. Po kompilacji i uruchomieniu programu pojawi się okno, takie jak na rysunku 12.5.

Rysunek 12.5. *Program poprawnie wyświetlający polskie znaki*

Podstawy hierarchii elementów interfejsu użytkownika

W typowym programie korzystającym z graficznego interfejsu użytkownika okna aplikacji zawierają wiele różnego rodzaju elementów. Biblioteka Qt jest tak skonstruowana, że elementy GUI stanowiące część okna ułożone są w odpowiedniej hierarchii. W pierwszej kolejności tworzony jest obiekt główny, potem budujemy obiekty klas będących częścią składową danego okna. W dwóch pierwszych programach rolę głównego i zarazem jedynego elementu GUI stanowił przycisk — obiekt klasy `QPushButton`. Ideę hierarchii obiektów w bibliotece Qt najlepiej zilustruje następny przykład.

Przykład 12.3. Hierarchia elementów GUI

1. Tradycyjnie utwórz nowy projekt, korzystając z szablonu aplikacji Qt (możesz także wykorzystać projekt utworzony w drugim przykładzie).
2. Dodaj plik nagłówkowy klasy `QMainWindow`, która będzie klasą bazową dla większości naszych programów przykładowych:

   ```
   #include <QMainWindow>
   ```

3. Po wierszach tworzących obiekt klasy `QApplication` i definiujących obsługę wybranego standardu polskich znaków:

   ```
   QApplication app (argc, argv);
   QTextCodec::setCodecForCStrings (QTextCodec::codecForName ("Windows-1250"));
   ```

   umieść obiekt klasy `QMainWindow` reprezentujący okno aplikacji:

   ```
   QMainWindow window;
   ```

   Klasa `QMainWindow` potrafi obsługiwać wszystkie typowe elementy głównego okna aplikacji, w tym menu, paski narzędziowe oraz pasek statusu, ale można ją także zastosować do utworzenia innych rodzajów okien. W programie użyliśmy wartości domyślnych konstruktora, który ma następującą postać:

   ```
   QMainWindow::QMainWindow (QWidget * parent = 0, Qt::WindowFlags flags = 0)
   ```

   Pierwszy parametr — `parent` — określa wskaźnik do obiektu nadrzędnego (rodzica) w hierarchii elementów GUI. Domyślna wartość oznacza, że dany obiekt jest głównym obiektem w hierarchii. Drugi parametr, czyli `flags`, opiszemy dokładniej w następnym rozdziale.

4. Dalej określ rozmiar i położenie okna oraz podaj jego tytuł:

   ```
   window.setGeometry (400,300,300,200);
   window.setWindowTitle ("Przykład 3");
   ```

   Obie powyższe metody pochodzą z klasy `QWidget`. Krótkiego wyjaśnienia wymaga tylko `setGeometry`, której dwa pierwsze parametry określają położenie lewego górnego narożnika kontrolki względem elementu nadrzędnego. Gdy tworzymy główne okno, będą to współrzędne położenia okna na pulpicie. Dwa ostatnie parametry określają rozmiary okna lub innego elementu GUI.

Zwróć jeszcze uwagę, że w poprzednich programach tytuł okna określała automatycznie biblioteka Qt na podstawie nazwy pliku wykonywalnego programu.

5. Następnie utwórz przycisk i umieść go we wcześniej utworzonym oknie:

```
QPushButton button ("Wyjście",&window);
button.setGeometry (100,120,100,40);
```

Jeżeli spojrzymy na definicję użytego w tym przypadku konstruktora klasy `QPushButton`:

```
QPushButton::QPushButton (const QString & text, QWidget * parent = 0)
```

to zobaczymy, że drugi parametr jest odpowiednikiem pierwszego parametru konstruktora klasy `QMainWindow`. W dwóch pierwszych programach przyciski były elementami głównymi, stąd drugi parametr konstruktora `QPushButton` miał wartość domyślną, jednak w tym przypadku tworzymy element podrzędny i w tym miejscu podajemy wskaźnik do głównego okna. Oczywiście, także położenie przycisku określone jest we współrzędnych okna nadrzędnego, a nie pulpitu.

6. Aby napis przycisku odpowiadał wykonywanej przez niego funkcji, musisz jeszcze odpowiednio połączyć sygnał (zdarzenie) generowany po naciśnięciu przycisku z odpowiednim gniazdem (ang. *slot*), czyli metodą, która będzie wywołana w wyniku naciśnięcia przycisku. Służy do tego metoda `connect` klasy `QObject`. W naszym przykładzie naciśnięcie przycisku zakończy pracę programu poprzez wywołanie metody `quit` klasy `QApplication`:

```
QObject::connect (&button,SIGNAL (clicked()),&app,SLOT (quit()));
```

Parametry tej funkcji tworzą dwie pary. Pierwsza para określa źródło i rodzaj sygnału (w naszym przykładzie obiekt `button` i jego metodę `clicked`), druga — jego odbiorcę i wywoływane gniazdo (w przykładzie obiekt `app` i jego metodę `quit`). Bliżej mechanizm sygnałów i gniazd opiszemy w rozdziale 15.

7. Ostatnią czynnością związaną z oknem jest jego wyświetlenie, co realizuje poznana już wcześniej metoda `show`:

```
window.show ();
```

Zauważ, że wykonujemy metodę `show` dla głównego obiektu w hierarchii elementów okna, czyli w tym przypadku dla obiektu klasy `QMainWindow`. Zadanie wyświetlenia elementów podrzędnych wykona główny element w hierarchii.

8. Po uzupełnieniu o znane już z poprzednich programów zakończenie:

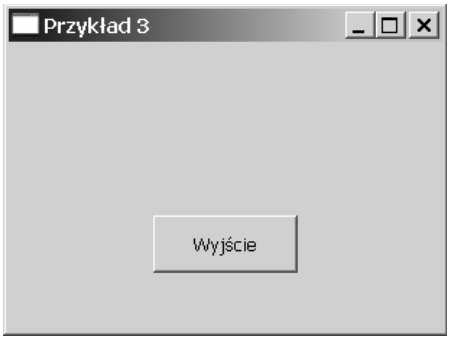
```
return app.exec();
```

całość tekstu źródłowego programu wygląda następująco:

```
#include <QApplication>
#include <QPushButton>
#include <QTextCodec>
#include <QMainWindow>
```

```
int main(int argc, char *argv[])
{
    QApplication app (argc, argv);
    QTextCodec::setCodecForCStrings (QTextCodec::codecForName ("Windows-1250"));
    QMainWindow window;
    window.setGeometry (400,300,300,200);
    window.setWindowTitle ("Przykład 3");
    QPushButton button ("Wyjście",&window);
    button.setGeometry (100,120,100,40);
    QObject::connect (&button,SIGNAL (clicked()),&app,SLOT (quit()));
    window.show ();
    return app.exec();
}
```

Po kompilacji i uruchomieniu otrzymamy okno przedstawione na rysunku 12.6.



Rysunek 12.6. *Przycisk umieszczony w oknie aplikacji*

Tworzenie własnej klasy okna

Ostatnim krokiem do poznania struktury aplikacji Qt jest utworzenie własnej klasy okna. Podobnie jak w poprzednim przykładzie, jako klasę bazową wykorzystamy `QMainWindow`, ale nic nie stoi na przeszkodzie, aby elementy GUI tworzyć na bazie klasy `QWidget`. Wybór `QMainWindow` podyktowany jest głównie jej przygotowaniem do obsługi standardowych elementów głównych okien programów. Jeżeli taka funkcjonalność nie jest potrzebna, klasę, którą poniżej opisujemy, można także zbudować na bazie `QWidget`. Popatrzmy zatem na kolejny przykład.

Przykład 12.4. Własna klasa okna

1. Ponownie skorzystaj z szablonu aplikacji Qt i zapisz projekt w nowym folderze.
2. Pliki nagłówkowe pozostają takie, jak w poprzednim przykładzie:

```
#include <QApplication>
#include <QPushButton>
#include <QTextCodec>
#include <QMainWindow>
```

3. Klasę reprezentującą okno nazwij `MyWindow`. Zawiera ona jedynie konstruktor, destruktor oraz jedno prywatne pole `button` — wskaźnik na obiekt klasy `QPushButton`:

```
class MyWindow: public QMainWindow
{
public:
    MyWindow ();
    ~MyWindow ();
private:
    QPushButton *button;
};
```

4. Konstruktor klasy `MyWindow` realizuje takie same zadanie jak program z poprzedniego przykładu. Tworzone okno ma rozmiary `300` na `200` pikseli i zawiera jeden przycisk, którego naciśnięcia zamyka okno i kończy działanie całego programu:

```
MyWindow::MyWindow (): QMainWindow ()
{
    setGeometry (400,300,300,200);
    setWindowTitle ("Przykład 4");
    button = new QPushButton ("Wyjście",this);
    button -> setGeometry (100,120,100,40);
    connect (button,SIGNAL (clicked()),qApp,SLOT (quit()));
}
```

   Warto zwrócić uwagę na to, że obiekt reprezentujący przycisk tworzymy dynamicznie, a wskaźnikiem do obiektu klasy bazowej względem przycisku jest `this`. Drugą zmianą jest wykorzystanie w wywołaniu funkcji `connect` jako trzeciego parametru makra `qApp`, które przekazuje wskaźnik do obiektu klasy `QApplication`. Specjalne mechanizmy zawarte w bibliotece Qt uniemożliwiają utworzenie więcej niż jednego obiektu tej klasy.

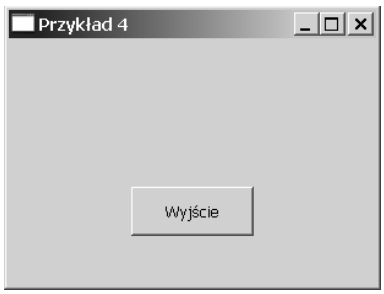
5. Destruktor klasy `MyWindow` pozostaw pusty. Usunięcie elementów GUI tworzonych dynamicznie zostanie zrealizowane automatycznie przez obiekt będący najwyżej w hierarchii.

```
MyWindow::~MyWindow ()
{
}
```

6. Prezentowana poniżej funkcja `main` osiągnęła postać, którą będziemy stosować w większości programów przykładowych:

```
int main (int argc, char *argv[])
{
    QApplication app (argc, argv);
    QTextCodec::setCodecForCStrings (QTextCodec::codecForName ("Windows-1250"));
    MyWindow window;
    window.show ();
    return app.exec();
}
```

   Po wpisaniu całości, kompilacji i uruchomieniu programu uzyskamy efekt niemal identyczny z efektem w poprzednim przykładzie. Jedyna różnica to inny tytuł okna. Okno programu przedstawiamy na rysunku 12.7.

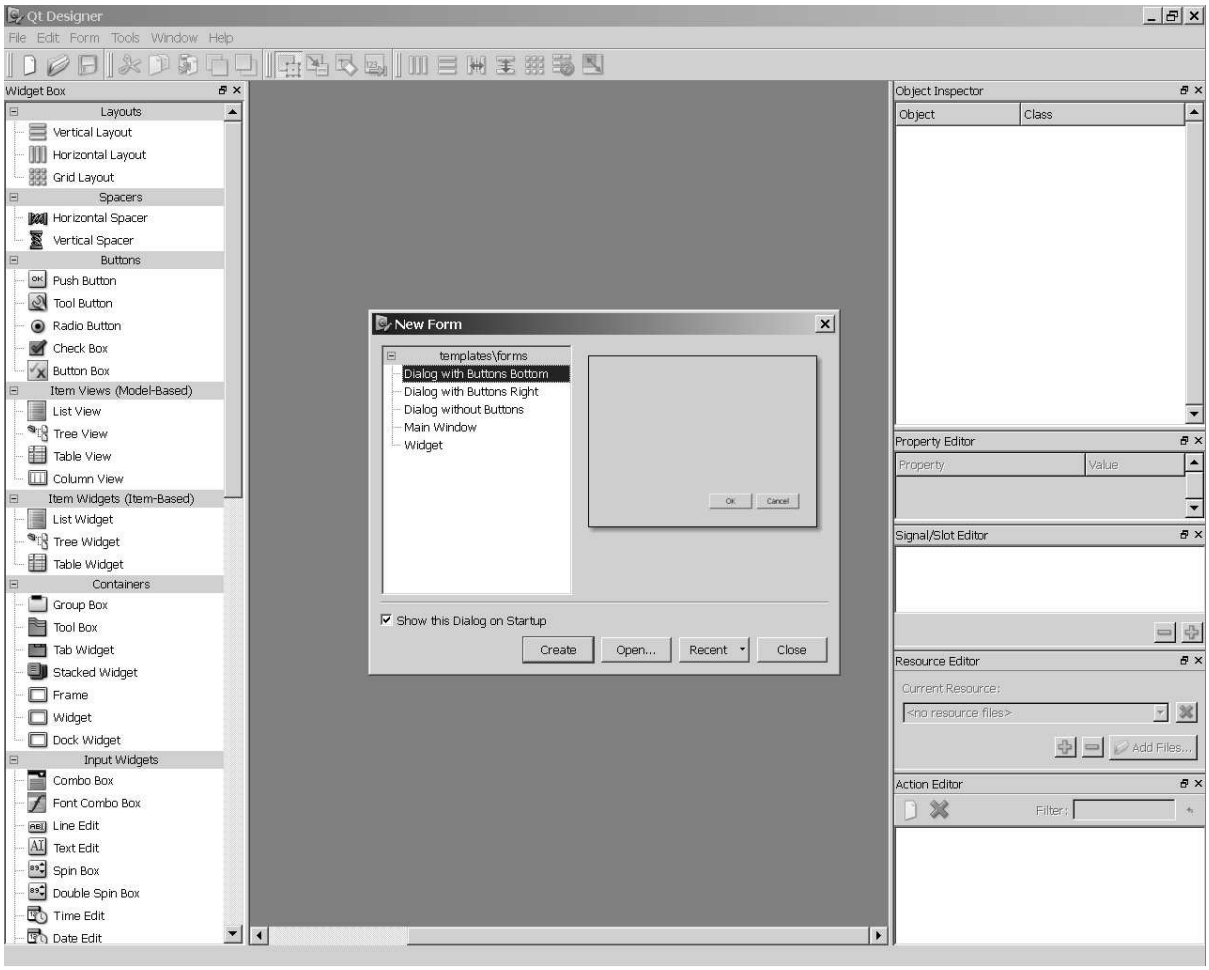


Rysunek 12.7. *Przycisk umieszczony w oknie reprezentowanym przez nową klasę*

Qt Designer

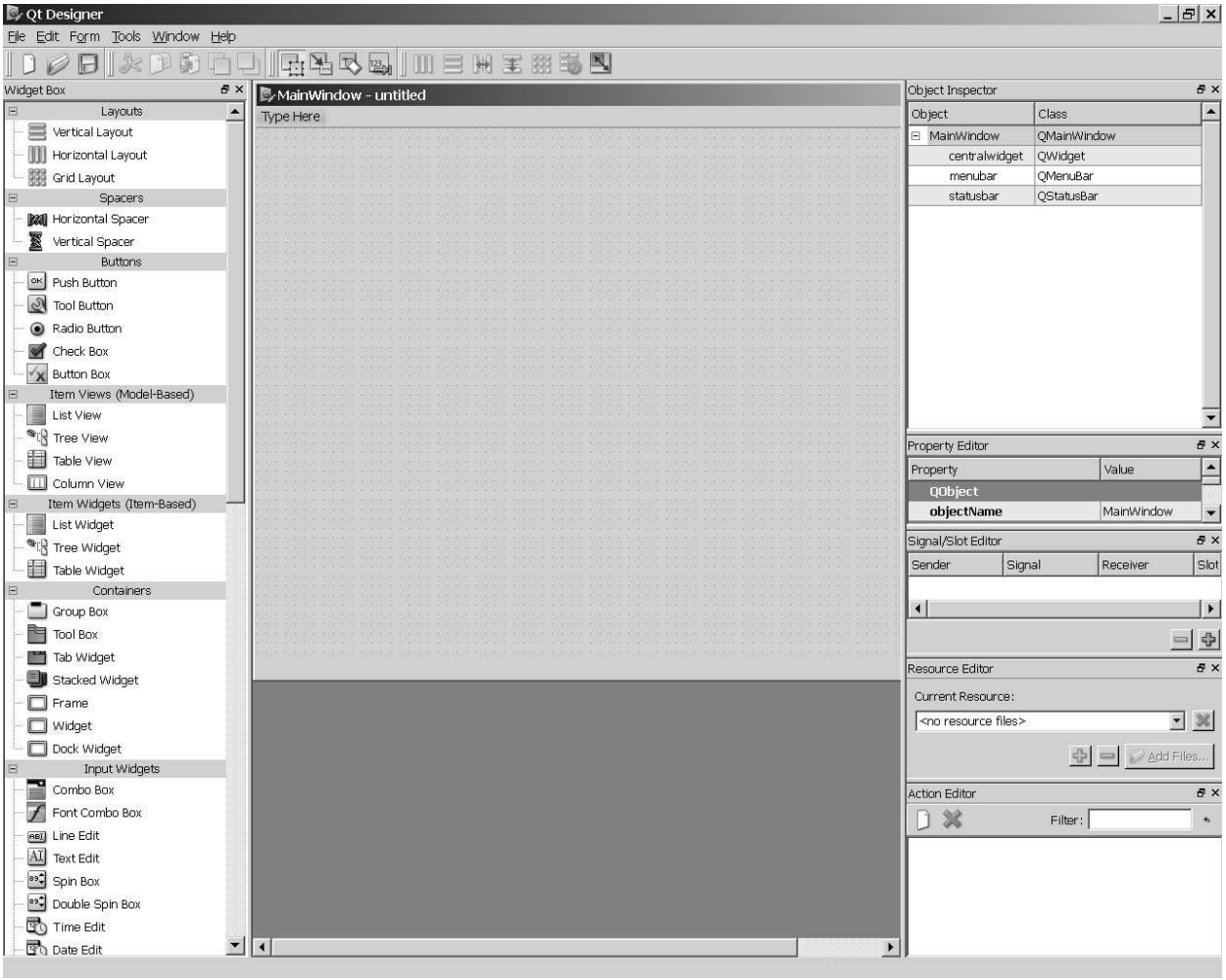
Filozofia pracy i ogólny wygląd Qt Designera są zbliżone do innych wizualnych edytorów graficznego interfejsu użytkownika. Osoby znające takie środowiska jak Borland Delphi czy Microsoft Visual Basic znajdą tu wiele podobieństw.

Przy uruchomieniu Qt Designer standardowo wyświetla okno kreatora rodzaju tworzonego formularza — tak jak to przedstawiamy na rysunku 12.8. Możemy wybrać okna dialogowe z przyciskami lub bez przycisków, okno główne oparte na klasie `QMainWindow` oraz dowolną kontrolkę GUI na bazie klasy `QWidget`. Jeżeli wyłączymy opcję wyświetlania kreatora nowego formularza, można go uruchomić z opcji menu *File\NewForm*.



Rysunek 12.8. *Qt Designer z otwartym oknem wyboru rodzaju tworzonego formularza*

Po wyborze odpowiedniego rodzaju tworzonego formularza (w naszym przykładzie jest to *Main Window*) otrzymujemy gotowy do pracy graficzny edytor interfejsu użytkownika, którego przykładowy wygląd przedstawiony jest na rysunku 12.9. Warto poświęcić chwilę czasu na wygodne ustawienie elementów edytora. Na początku można zamknąć część okien — proponuję pozostawić *Widget Box*, *Property Editor* oraz *Object Inspector*.

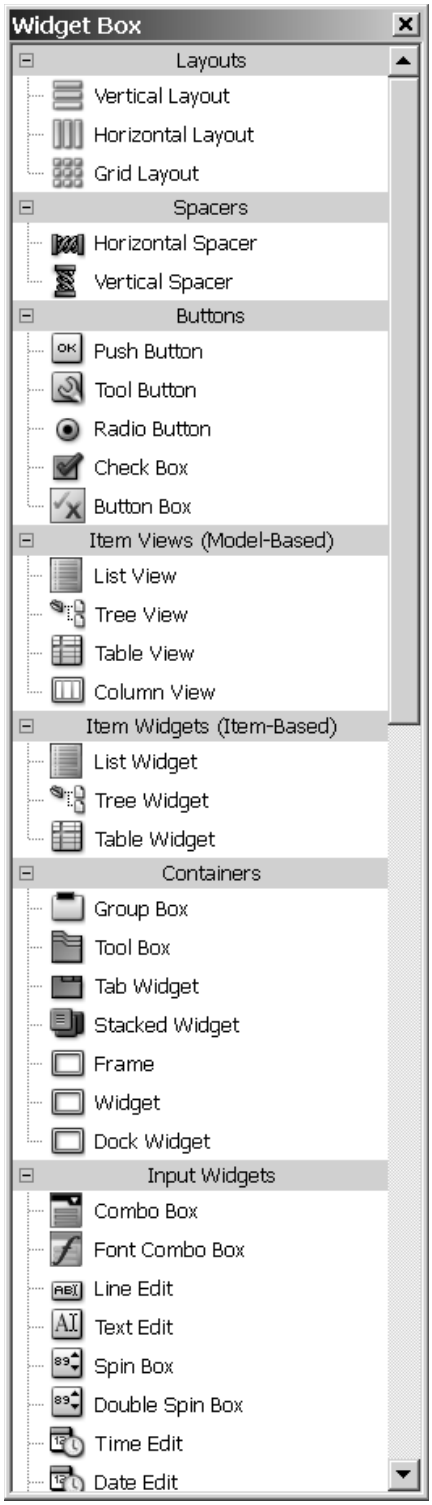


Rysunek 12.9. *Qt Designer w trakcie edycji formularza typu Main Window*

Przedstawione na rysunku 12.10 okno *Widget Box* zawiera paletę elementów GUI (okno widoczne jest także w lewej części rysunku 12.9). Wybraną kontrolkę przenosimy na okno formularza, korzystając z mechanizmu przeciągnij i upuść. Widoczna na oknie siatka punktów ułatwia rozmieszczanie poszczególnych elementów okna.

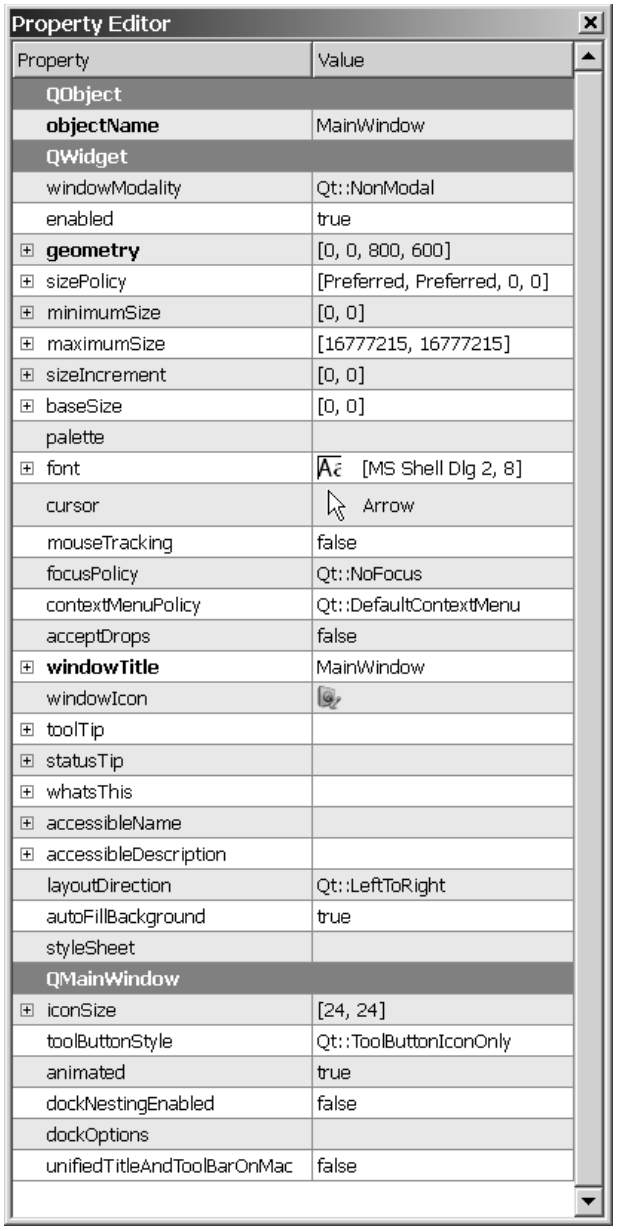
Właściwości wybranego elementu wyświetlane są w oknie *Property Editor*, które przedstawiamy na rysunku 12.11. Okno to zawiera informacje o klasie reprezentującej dany element oraz jej klasach bazowych. Na rysunku 12.12 widzimy, że wybranym elementem jest główne okno aplikacji (klasa `QMainWindow`, która dziedziczy po klasach `QWidget` i `QObject`). Wszystkie właściwości wyświetlane w oknie *Property Editor* mają — oczywiście — swoje odpowiedniki w polach klasy.

Rysunek 12.10.
Okno Widget Box

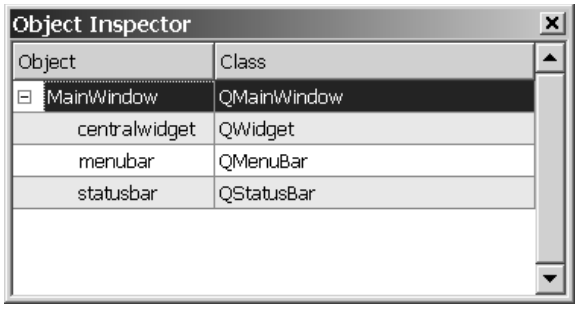


Ostatnim z podstawowych okien Qt Designera jest okno *Object Inspector*, które przedstawiamy na rysunku 12.12. Zawiera ono hierarchię klas bieżącego projektu. W oknie prezentowanym na rysunku 12.12 zamieściliśmy początkową hierarchię klas szablonu *Main Window*. Widzimy tu elementy klasy `QMainWindow`, takie jak centralna kontrolka okna (`centralwidget`), pasek menu (`menubar`) oraz pasek statusu (`statusbar`), które bliżej omówimy w następnym rozdziale.

Rysunek 12.11.
Okno Property Editor



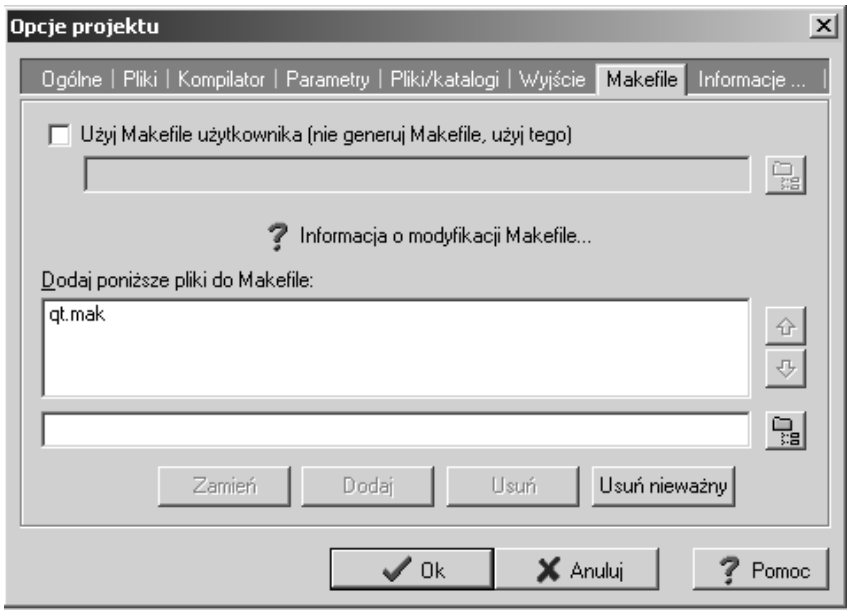
Rysunek 12.12.
Okno Object Inspector



Integracja Qt Designer i Dev-C++

Qt Designer zapisuje opis formularzy w plikach z rozszerzeniem *ui*, które wykorzystują składnię XML. Biblioteka Qt zawiera specjalne narzędzie — kompilator interfejsu użytkownika UIC (ang. *user interface compiler*) — które konwertuje pliki *ui* do

plików nagłówkowych w języku C++. Do integracji w środowisku Dev-C++ kompilatora UIC i innych specjalistycznych narzędzi biblioteki Qt wykorzystamy narzędzie `qt-moc-ui-rcc`. Plik wykonywalny *qt-moc-ui-rcc.exe* kopiujemy do folderu *C:\Dev-Cpp*
↪bin, a plik *qt.mak*, zawierający dodatkowe polecenia dla narzędzia `make`, umieszczamy w folderze z projektem. Ostatnim etapem konfiguracji jest dodanie pliku *qt.mak* w opcjach projektu Dev-C++ (menu *Projekt\Opcje projektu*) w zakładce *Makefile*, tak jak to przedstawiamy na rysunku 12.13. W przypadku plików formularzy automatycznie wygenerowany przez UIC plik nagłówkowy będzie miał nazwę taką samą jak plik *ui* (bez rozszerzenia) uzupełnioną dodatkowo o przedrostek *ui_*.



Rysunek 12.13. *Dodanie pliku qt.mak do projektu*

Wykorzystanie formularzy w programach

Wygenerowane za pomocą Qt Designera pliki formularzy można w stosunkowo łatwy sposób wykorzystać w aplikacji. Dwa sposoby najczęściej stosowne w praktyce przećwiczymy na przykładach. Pierwszym z nich będzie użycie wielodziedziczenia.

Przykład 12.5. Formularz dołączony za pomocą wielodziedziczenia

1. Korzystając z szablonu aplikacji Qt, utwórz nowy projekt. Funkcja `main` będzie miała taką samą zawartość jak w poprzednim przykładzie. Możesz także wykorzystać pliki z tego przykładu. Klasę `MyWindow` ogranicz do minimum:

```
class MyWindow: public QMainWindow
{
public:
    MyWindow ();
    ~MyWindow () {}
};
```

Podobnie ograniczona do minimum jest implementacja konstruktora:

```
MyWindow::MyWindow ():
    QMainWindow (0,Qt::Window)
{
}
```

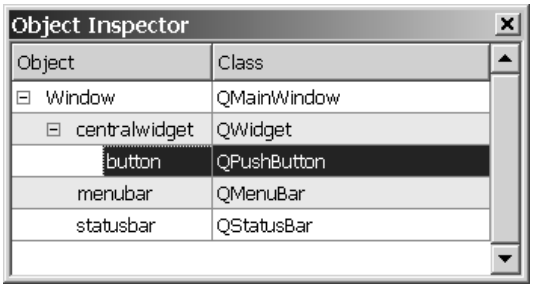
2. W Qt Designerze utwórz nowy projekt formularza oparty na szablonie *Main Window*. Korzystając z edytora właściwości (okno *Property Editor*), zmodyfikuj nazwę klasy (właściwość `objectName`) na `Window`, rozmiary okna (właściwość `geometry`) do `300` na `200` pikseli oraz dobierz odpowiedni tytuł okna (właściwość `windowTitle`). Wstępnie przygotowany projekt zapisz pod nazwą *window.ui* w folderze z projektem. Okno *Property Editor* z właściwościami naszej klasy `Window` przedstawiamy na rysunku 12.14. Warto jednocześnie zauważyć, że te właściwości klasy, których wartość po modyfikacji odbiega od stanu początkowego, wyróżniane są w oknie *Property Editor* pogrubioną czcionką.

Rysunek 12.14. *Oko Property Editor z wyróżnionymi zmianami właściwości objectName, geometry i windowTitle*

Property Editor

Property	Value
QObject	
objectName	Window
QWidget	
windowModality	Qt::NonModal
enabled	true
⊞ **geometry**	[0, 0, 300, 200]
⊞ sizePolicy	[Preferred, ...erred, 0, 0]
⊞ minimumSize	[0, 0]
⊞ maximumSize	[16777215, 16777215]
⊞ sizeIncrement	[0, 0]
⊞ baseSize	[0, 0]
palette	
⊞ font	[MS Shell Dlg 2, 8]
cursor	Arrow
mouseTracking	false
focusPolicy	Qt::NoFocus
contextMenuPolicy	Qt::DefaultContextMenu
acceptDrops	false
⊞ **windowTitle**	QtDesigner - test
windowIcon	
⊞ toolTip	
⊞ statusTip	
⊞ whatsThis	
⊞ accessibleName	
⊞ accessibleDescription	
layoutDirection	Qt::LeftToRight
autoFillBackground	true
styleSheet	
QMainWindow	
⊞ iconSize	[24, 24]
toolButtonStyle	Qt::ToolButtonIconOnly
animated	true
dockNestingEnabled	false
dockOptions	
unifiedTitleAndToolBarOnMac	false

3. Na formularzu okna umieszczamy przycisk — obiekt klasy `PushButton`. Jego nazwę zmieniamy na `button`, rozmiary i położenie to kolejno: `100`, `120`, `100` i `40`, a wyświetlany tekst (właściwość `text`) to `Wyjście`. Struktura klasy formularza powinna wyglądać tak, jak na rysunku 12.15.



Rysunek 12.15. *Struktura klas formularza zawierającego okno z jednym przyciskiem*

4. Do ukończenia prac nad pierwszym formularzem pozostało jeszcze wyłączenie niepotrzebnego paska statusu oraz paska menu. Wybierz odpowiednie klasy w oknie *Object Inspector* i właściwość `enabled` ustaw na `false`. Dodatkowo dla paska statusu musisz wyłączyć jeszcze brzeg dla łapki kursora myszy (właściwość `sizeGripEnabled`). W efekcie otrzymaliśmy projekt formularza dokładnie odpowiadający oknu z poprzedniego przykładu.

   Plik *window.ui* wygląda następująco:

```
<ui version="4.0" >
 <class>Window</class>
 <widget class="QMainWindow" name="Window" >
  <property name="geometry" >
   <rect>
    <x>0</x>
    <y>0</y>
    <width>300</width>
    <height>200</height>
   </rect>
  </property>
  <property name="windowTitle" >
   <string>QtDesigner - test</string>
  </property>
  <widget class="QWidget" name="centralwidget" >
   <property name="enabled" >
    <bool>true</bool>
   </property>
   <widget class="QPushButton" name="button" >
    <property name="geometry" >
     <rect>
      <x>100</x>
      <y>120</y>
      <width>100</width>
      <height>40</height>
     </rect>
    </property>
    <property name="text" >
     <string>WyjĹ›cie</string>
    </property>
   </widget>
```

```
  </widget>
  <widget class="QMenuBar" name="menubar" >
   <property name="enabled" >
    <bool>false</bool>
   </property>
   <property name="geometry" >
    <rect>
     <x>0</x>
     <y>0</y>
     <width>300</width>
     <height>23</height>
    </rect>
   </property>
  </widget>
  <widget class="QStatusBar" name="statusbar" >
   <property name="enabled" >
    <bool>false</bool>
   </property>
   <property name="sizeGripEnabled" >
    <bool>false</bool>
   </property>
  </widget>
 </widget>
 <resources/>
 <connections/>
</ui>
```

5. Do projektu dołącz plik *window.ui*, a w opcjach projektu umieść plik *qt.mak* i następnie skompiluj całość. Przy pierwszej kompilacji w folderze z projektem zostanie utworzony dodatkowy plik *ui_window.h*, który został wygenerowany przez kompilator UIC. Plik ten dołącz do projektu.

   Plik *ui_window.h* zawiera jednocześnie deklarację i implementację klasy `Ui_Window`:

```
class Ui_Window
{
public:
    QWidget *centralwidget;
    QPushButton *button;
    QMenuBar *menubar;
    QStatusBar *statusbar;

    void setupUi(QMainWindow *Window)
    {
    if (Window->objectName().isEmpty())
        Window->setObjectName(QString::fromUtf8("Window"));
    Window->resize(300, 200);
    centralwidget = new QWidget(Window);
    centralwidget->setObjectName(QString::fromUtf8("centralwidget"));
    centralwidget->setEnabled(true);
    button = new QPushButton(centralwidget);
    button->setObjectName(QString::fromUtf8("button"));
    button->setGeometry(QRect(100, 120, 100, 40));
    Window->setCentralWidget(centralwidget);
    menubar = new QMenuBar(Window);
```

```
        menubar->setObjectName(QString::fromUtf8("menubar"));
        menubar->setEnabled(false);
        menubar->setGeometry(QRect(0, 0, 300, 23));
        Window->setMenuBar(menubar);
        statusbar = new QStatusBar(Window);
        statusbar->setObjectName(QString::fromUtf8("statusbar"));
        statusbar->setEnabled(false);
        statusbar->setSizeGripEnabled(false);
        Window->setStatusBar(statusbar);

        retranslateUi(Window);

        QMetaObject::connectSlotsByName(Window);
        } // setupUi

        void retranslateUi(QMainWindow *Window)
        {
        Window->setWindowTitle(QApplication::translate("Window", "QtDesigner - test",
        ↪0, QApplication::UnicodeUTF8));
        button->setText(QApplication::translate("Window", "Wyj\305\233cie", 0,
        ↪QApplication::UnicodeUTF8));
        Q_UNUSED(Window);
        } // retranslateUi

    };
```

Dodatkowo plik ten zawiera przestrzeń nazw `Ui` z klasą `Ui_Window`:

```
namespace Ui {
    class Window: public Ui_Window {};
} // namespace Ui
```

6. W definicji klasy dodaj dziedziczenie prywatne klasy wygenerowanej *Ui::Window*:

```
class MyWindow: public QMainWindow, private Ui::Window
```

Oczywiście, na początku pliku *mywindow.h* trzeba jeszcze dołączyć odpowiedni plik nagłówkowy:

```
#include "ui_window.h"
```

7. W konstruktorze naszej klasy wywołaj metodę *setupUi* znajdującą się w klasie `Ui::Window`:

```
setupUi (this);
```

Parametrem tej metody jest wskaźnik do obiektu klasy `QMainWindow`, czyli w naszym przypadku `this`.

Łączymy także sygnał wysyłany przez przycisk `button` w dokładnie taki sam sposób jak w poprzednim przykładzie.

```
connect (button,SIGNAL (clicked()),qApp,SLOT (quit()))
```

8. Po kompilacji projektu i uruchomieniu programu uzyskamy okno, takie jak na rysunku 12.16.

Rysunek 12.16. *Okno QMainWindow wygenerowane przy użyciu Qt Designera*

Drugim obok wielodziedziczenia typowym sposobem wykorzystania w programie formularzy wygenerowanych przez Qt Designer jest dołączenie pola będącego obiektem klasy wygenerowanej przez kompilator UIC. Różnice w stosunku do wielodziedziczenia pokażemy na przykładzie.

Przykład 12.6. Formularz dołączony jako pole klasy

1. Skopiuj wszystkie pliki z poprzedniego przykładu i otwórz plik projektu w Dev-C++.
2. W definicji klasy `MyWindow` dodaj pole `window` będące obiektem klasy `Ui::Window`. Jednocześnie usuń dziedziczenie po klasie `Ui::Window`:

```
class MyWindow: public QMainWindow
{
public:
    MyWindow ();
    ~MyWindow () {}
private:
    Ui::Window window;
};
```

3. Konstruktor klasy `MyWindow`, podobnie jak w poprzednim przykładzie, musi wywołać metodę `setupUi` klasy `Ui::Window`. Jedyna różnica polega na innym sposobie dostępu do tej metody. Metodę, zamiast bezpośrednio, wywołujemy za pośrednictwem pola `window`:

```
window.setupUi (this);
```

   Analogicznie modyfikujemy pierwszy parametr funkcji `connect`:

```
connect (window.button,SIGNAL (clicked()),qApp,SLOT (quit()))
```

4. Po kompilacji i uruchomieniu programu otrzymujemy efekt analogiczny do efektu z przykładu 12.5.

Z dwóch przedstawionych metod wykorzystania formularzy generowanych przez Qt Designer najbardziej uniwersalna jest metoda druga. W przeciwieństwie do wielodziedziczenia, pozwala ona na bezproblemowe dołączenie do jednej klasy aplikacji wielu formularzy. Jednak w typowych sytuacjach zastosowanie wielodziedziczenia jest najwygodniejszą metodą, głównie dzięki bezpośredniemu dostępowi do wszystkich elementów formularza.

Test

Wśród pytań testowych dotyczących powyższego rozdziału poprawna jest co najmniej jedna odpowiedź.

1. Bazową klasą okna aplikacji może być:
   - **a)** `QPushButton`,
   - **b)** `QMainWindow`,
   - **c)** `QWidget`,
   - **d)** `QApplication`.
2. Ile obiektów klasy `QApplication` może wystąpić w programie:
   - **a)** dowolna ilość,
   - **b)** tylko jeden,
   - **c)** ilość obiektów zależy od ilości kontrolek.
3. Jakie domyślne kodowanie znaków obsługuje klasa `QString`:
   - **a)** Latin 1,
   - **b)** Windows 1250,
   - **c)** ICO 8859-2.
4. Łączenie sygnałów i gniazd umożliwia metoda `connect` klasy:
   - **a)** `QWidget`,
   - **b)** `QMainWindow`,
   - **c)** `QApplication`,
   - **d)** `QObject`,
   - **e)** `QString`.
5. Domyślna nazwa obiektu klasy `QMainWindow` formularza tworzonego przez Qt Designer to:
   - **a)** `MyWindow`,
   - **b)** `MainWindow`,
   - **c)** `Window`,
   - **d)** `Widget`.