



ODKRYJ TAJEMNICE BIBLIOTEKI QT!

Biblioteki Qt

Zaawansowane
programowanie

przy użyciu C++



Mark Summerfield

Tytuł oryginału: Advanced Qt Programming: Creating Great Software with C++ and Qt 4

Tłumaczenie: Radosław Meryk

ISBN: 978-83-246-8233-1

Authorized translation from the English language edition, entitled: ADVANCED QT PROGRAMMING: CREATING GREAT SOFTWARE WITH C++ AND QT 4; ISBN 0321635906; by Mark Summerfield; published by Pearson Education, Inc; publishing as Prentice Hall. Copyright © 2011 Qtrac Ltd.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Polish language edition published by HELION S.A., Copyright © 2014.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Wydawnictwo HELION dołożyło wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/bibqtc.zip>

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

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to!» Nasza społeczność](#)

Spis treści

Słowo wstępne	9
Wprowadzenie	11
Rozdział 1. Aplikacje hybrydowe desktopowo-internetowe	17
Widżety internetowe	18
Korzystanie z WebKit	32
Generyczny komponent przeglądarki internetowej	33
Tworzenie aplikacji specyficznych dla witryny WWW	41
Osadzanie widżetów Qt na stronach WWW	54
Rozdział 2. Audio i wideo	63
Korzystanie z klas QSound i QMovie	64
Framework obsługi multimediiów Phonon	70
Odtwarzanie muzyki	74
Odtwarzanie wideo	89
Rozdział 3. Modele tabel model-widok	97
Architektura model-widok frameworka Qt	98
Wykorzystanie modeli QStandardItemModel dla tabel	100
Zmianianie modelu tabeli za pośrednictwem interfejsu użytkownika	101
Podklasa QStandardItemModel dla tabel	112
Model QSortFilterProxyModel do filtrowania duplikatów	116
Model QSortFilterProxyModel do filtrowania pożądaných wierszy	118
Tworzenie własnych modeli tabel	122
Zmianianie modelu tabeli za pośrednictwem interfejsu użytkownika	122
Niestandardowa podklasa QAbstractTableModel dla tabel	125
Rozdział 4. Modele drzew w architekturze model-widok	139
Wykorzystanie klasy QStandardItemModel dla drzew	141
Zmianianie modelu drzewa za pośrednictwem interfejsu użytkownika	142
Podklasa QStandardItem dla elementów drzewa	151
Podklasa QStandardItemModel dla drzew	152
Tworzenie niestandardowych modeli drzew	160
Zmiana modelu drzewa za pomocą interfejsu użytkownika	161
Niestandardowa klasa opisująca element drzewa	165
Niestandardowa podklasa klasy QAbstractItemModel dla drzew	168

Rozdział 5. Delegaty w architekturze model-widok	193
Edytory specyficzne dla typów danych	194
Delegaty specyficzne dla typów danych	196
Delegat tylko do odczytu dla kolumn lub wierszy	197
Delegat dla kolumn lub wierszy, które można edytować	201
Delegaty specyficzne dla modelu	208
Rozdział 6. Widoki w architekturze model-widok	215
Podklasy klasy QAbstractItemView	216
Widoki wizualizacji specyficzne dla modelu	232
Widżet wizualizatora	233
Zagregowany widżet nagłówka w wizualizatorze	239
Zagregowany widżet widoku w wizualizatorze	243
Rozdział 7. Wielowątkowość z wykorzystaniem przestrzeni nazw QtConcurrent	253
Uruchamianie funkcji w wątkach	256
Zastosowanie metody QtConcurrent::run()	260
Wykorzystanie podklasy klasy QRunnable	265
Filtrowanie i mapowanie w wątkach	268
Wykorzystanie funkcji przestrzeni nazw QtConcurrent do filtrowania	278
Wykorzystanie funkcji przestrzeni nazw QtConcurrent do filtrowania z redukcją	285
Wykorzystanie funkcji przestrzeni nazw QtConcurrent do mapowania	289
Rozdział 8. Obsługa wielu wątków z wykorzystaniem klasy QThread	295
Przetwarzanie niezależnych elementów	296
Przetwarzanie współdzielonych elementów	310
Rozdział 9. Tworzenie edytorów tekstu sformatowanego	325
Klasa QTextDocument — wprowadzenie	326
Tworzenie własnych edytorów tekstu	328
Uzupełnianie w polach tekstowych oraz polach kombi	329
Uzupełnianie i podświetlanie składni dla edytorów tekstu	330
Jednowierszowy edytor sformatowanego tekstu	350
Wielowierszowy edytor sformatowanego tekstu	361
Rozdział 10. Tworzenie sformatowanych dokumentów	367
Jakość obiektu QTextDocument wyeksportowanego do pliku	369
Tworzenie dokumentów QTextDocument	372
Tworzenie dokumentów QTextDocument za pomocą HTML	373
Tworzenie dokumentów QTextDocument za pomocą obiektów klasy QTextCursor	375
Eksportowanie i drukowanie dokumentów	379
Eksportowanie dokumentów QTextDocument	380
Drukowanie i przeglądanie dokumentów QTextDocument	384
Rysowanie stron	387
Rysowanie dokumentów PDF lub PostScript	394
Rysowanie dokumentów SVG	395
Rysowanie dokumentów z grafiką rastrową	395
Rozdział 11. Tworzenie okien w architekturze grafika-widok	397
Architektura grafika-widok	398
Widżety i układ w architekturze grafika-widok	401
Wprowadzenie do elementów graficznych	407

Rozdział 12. Tworzenie scen w architekturze grafika-widok	417
Sceny, elementy i akcje	419
Tworzenie głównego okna	420
Zapisywanie, ładowanie, drukowanie i eksportowanie scen	423
Wykonywanie operacji na elementach graficznych	431
Ulepszanie widoku QGraphicsView	447
Tworzenie przyborników w postaci widżetów doku	448
Tworzenie własnych elementów graficznych	454
Ulepszanie klasy QGraphicsTextItem	455
Ulepszanie istniejących elementów graficznych	463
Tworzenie własnych elementów graficznych od podstaw	466
Rozdział 13. Frameworki obsługi animacji i maszyn stanów	475
Wprowadzenie do frameworka animacji	476
Wprowadzenie do frameworka maszyny stanów	480
Połączenie animacji z maszynami stanów	487
Epilog	497
Wybrana bibliografia	501
O autorze	505
Skorowidz	507

11 Tworzenie okien w architekturze grafika-widok

W tym rozdziale:

- Architektura grafika-widok
- Widżety i układ w architekturze grafika-widok
- Wprowadzenie do elementów graficznych

Poprzez utworzenie własnej implementacji procedury obsługi `paintEvent()` podklasy klasy `QWidget` i zastosowanie klasy `QPainter` możemy narysować wszystko, co chcemy. Metoda ta idealnie nadaje się do rysowania własnych widżetów, ale nie jest wygodna, gdybyśmy chcieli rysować wiele pojedynczych elementów, zwłaszcza gdyby zależało nam na zapewnieniu użytkownikom możliwości interakcji z elementami. W przeszłości niektórzy programiści tworzyli na przykład aplikacje graficzne, korzystając dosłownie z tysięcy własnych widżetów spełniających funkcję elementów graficznych. Choć rysowanie widżetów przebiega bardzo szybko, obsługa jednokrotnego kliknięcia myszą w takich sytuacjach mogła z łatwością zużyć prawie całą moc obliczeniową procesora. Na szczęście w wersji 4.2 frameworka Qt wprowadzono architekturę grafika-widok, która doskonale wypełnia potrzeby wysokowydajnego rysowania i interakcji na poziomie elementów.

Chociaż architektura grafika-widok pierwotnie była pomyślana jako następcą klasy `QCanvas` z frameworka Qt w wersji 3., funkcjonalność architektury grafika-widok Qt w wersji 4. znacznie wykracza poza funkcjonalność klasy `QCanvas`. W rzeczywistości w niektórych aplikacjach obiekt `QGraphicsView` odgrywa rolę centralnego widżetu okna, w którym są umieszczone wszystkie inne widżety. W ten sposób powstaje interfejs użytkownika wewnątrz widoku w postaci własnych elementów graficznych.

Pierwszy podrozdział tego rozdziału rozpoczniemy od zwięzłego omówienia architektury grafika-widok. Zamieścimy również ramkę opisującą znaczące zmiany wprowadzone w Qt 4.6. Następnie, w drugim podrozdziale, przeanalizujemy aplikację, w której centralnym widżetem w głównym oknie jest obiekt `QGraphicsView` zawierający zarówno widżety, jak i konwencjonalne elementy graficzne. Na koniec, w trzecim podrozdziale tego rozdziału, omówimy prostą podklasę klasy `QGraphicsItem` oraz opiszemy API klasy `QGraphicsItem`.

W następnym rozdziale przyjrzymy się konwencjonalnej aplikacji w architekturze grafika-widok — prostemu programowi do rysowania. Omówimy większość klas architektury grafika-widok i zaprezentujemy więcej przykładów tworzenia własnych elementów graficznych. Nawiasem mówiąc, do przykładów przedstawionych w tym i w następnym rozdziale wrócimy w rozdziale 13., w którym utworzymy ich zmodyfikowane wersje, korzystające ze specyficznych własności frameworka Qt 4.6.

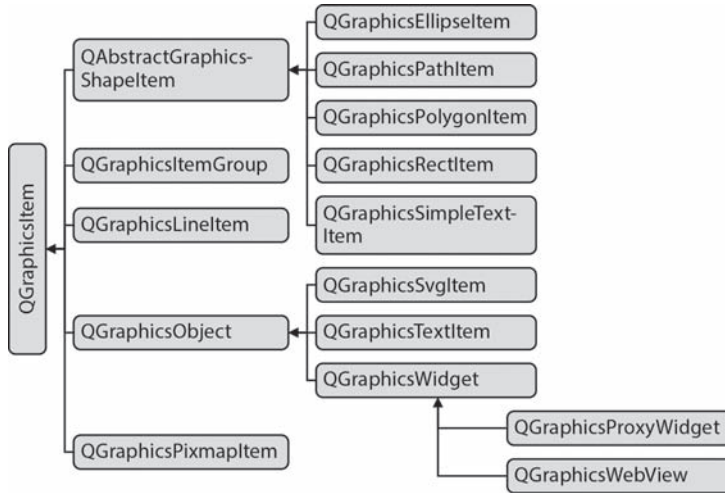
Architektura grafika-widok

Podobnie jak w architekturze model-widok frameworka Qt, w architekturze grafika-widok występuje niez wizualizowana klasa służąca do przechowywania danych w postaci modelu (QGraphicsScene) oraz klasa do wizualizacji danych (QGraphicsView). Jeśli jest taka potrzeba, to tę samą scenę można zwizualizować w wielu różnych widokach. Scena graficzna zawiera elementy, które wywodzą się z abstrakcyjnej klasy QGraphicsItem.

Począwszy od pojawienia się architektury grafika-widok frameworka Qt po raz pierwszy, deweloperzy podejmowali olbrzymie wysiłki zmierzające do poprawienia zarówno szybkości działania, jak i możliwości tego mechanizmu. Sceny można skalować, obracać i drukować, a renderowanie można realizować za pomocą silnika renderowania frameworka Qt albo za pomocą OpenGL. Architektura pozwala także na tworzenie animacji oraz obsługuje technikę „przeciągnij i upuść”. Sceny graficzne można wykorzystywać do prezentowania dowolnej liczby elementów — od zaledwie kilku do dziesiątek tysięcy, a nawet więcej.

Framework Qt dostarcza wielu predefiniowanych, gotowych do wykorzystania typów elementów graficznych. Wyszczególniono je na rysunku 11.1. Większość nazw klas nie wymaga dodatkowych objaśnień, ale omówimy kilka spośród tych, które nie są oczywiste. Klasa QGraphicsPathItem reprezentuje obiekt klasy QPainterPath — w istocie jest to dowolna figura składająca się z prymitywów, które framework Qt potrafi rysować. Obejmuje to łuki, krzywe Béziera, cięciwy (ang. *chords*), elipsy, linie, prostokąty i tekst. Klasa QGraphicsSimpleTextItem reprezentuje fragment zwykłego tekstu, natomiast klasa QGraphicsTextItem reprezentuje fragment sformatowanego tekstu frameworka Qt (może on być określony za pomocą HTML — zagadnienia związane z tekstem sformatowanym omawialiśmy w poprzednich dwóch rozdziałach). QGraphicsWidget odgrywa rolę klasy bazowej do tworzenia niestandardowych widżetów przeznaczonych do wykorzystania na scenach graficznych. Istnieje również możliwość osadzania na scenach graficznych standardowych widżetów — pochodnych klasy QWidget — aby to zrobić, należy dodać widżet do obiektu QGraphicsProxyWidget, a następnie dodać widżet proxy do sceny. Używanie widżetów proxy (lub bezpośrednio obiektów QWidget) jest „wolne”, ale to, czy będzie to odczuwalne, zależy od aplikacji¹. Klasę QGraphicsWebView wprowadzono w Qt w wersji 4.6.

¹ Informacje dotyczące zagadnień związanych z wydajnością używania obiektów QWidget i obiektów proxy na scenach graficznych można znaleźć pod adresem labs.qt.nokia.com/blogs/2010/01/11/qt-graphics-and-performance-the-cost-of-convenience.



Rysunek 11.1. Hierarchia klasy `QGraphicsItem` frameworka Qt

Klasa ta dostarcza graficznej wersji klasy `QWebView`, którą omawialiśmy w rozdziale 1., i służy do prezentowania treści z internetu na scenach graficznych.

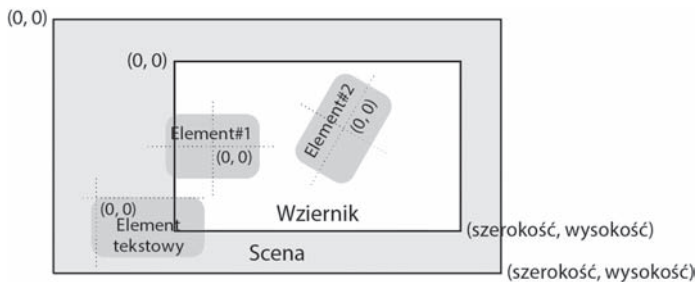
W przypadku scen z małą liczbą elementów możemy skorzystać z obiektów `QGraphicsObject` wprowadzonych w Qt 4.6. Jeśli chodzi o Qt 4.5 lub wersje wcześniejsze, podstawą niestandardowych elementów mogą być zarówno klasa `QObject`, jak i `QGraphicsItem`. Powoduje to zwiększenie kosztów związanych z elementem (tzn. elementy zużywają więcej pamięci), ale gwarantuje wygodę korzystania z sygnałów i slotów oraz systemu właściwości frameworka Qt. W przypadku scen zawierających bardzo dużo elementów zazwyczaj lepsze efekty daje skorzystanie z lekkiej klasy `QGraphicsItem` jako podstawy niestandardowych elementów, natomiast obiekty `QGraphicsObject` stosuje się tylko w odniesieniu do elementów mniej licznych.

Klasy widoków graficznych w zasadzie są dwuwymiarowe, chociaż każdy element ma współrzędną z, przy czym elementy, które mają większą wartość współrzędnej z, są rysowane przed tymi o niższych wartościach współrzędnej z. Wykrywanie kolizji bazuje na pozycjach elementu określonych parą współrzędnych (x, y). Oprócz informacji na temat kolizji scena może poinformować nas, które elementy zawierają konkretny punkt lub znajdują się w danym obszarze oraz które są zaznaczone. Sceny mają również warstwę pierwszego planu, która przydaje się na przykład do narysowania siatki nałożonej na wszystkie elementy na scenie. Mają także warstwę tła rysowaną pod wszystkimi elementami, która jest przydatna do określenia obrazu lub koloru tła.

Elementy są albo dziećmi sceny, albo dziećmi innego elementu — tak samo jak w przypadku zwykłych relacji rodzic-dziecko frameworka Qt. Kiedy do elementu zostaną zastosowane przekształcenia, będą one automatycznie zastosowane do wszystkich dzieci elementu — rekurencyjnie, aż do najdalszego potomka. Oznacza to, że jeśli element zostanie przeniesiony — gdy użytkownik na przykład go przeciągnie — to wszystkie jego dzieci (i rekurencyjnie ich dzieci) będą przeciągnięte razem z nim. Aby element potomny igno-

rował przekształcenia wykonywane na jego rodzicu, można wywołać metodę `QGraphicsItem::setFlag(QGraphicsItem::ItemIgnoresTransformations)`. Do innych, częściej używanych flag można zaliczyć te, które włączają możliwości przemieszczania, zaznaczania i nadawania fokusu (wszystkie flagi wyszczególniono w tabeli 11.3). Elementy mogą być również grupowane. W tym celu powinny stać się dziećmi obiektu `QGraphicsItemGroup`. Jest to bardzo przydatny mechanizm tworzenia doraźnych kolekcji elementów.

Klasy widoków graficznych wykorzystują trzy różne systemy współrzędnych, chociaż w praktyce uwzględniamy tylko dwa z nich. Widoki wykorzystują fizyczny układ współrzędnych. Sceny korzystają z logicznego układu współrzędnych, który definiujemy poprzez przekazanie do konstruktora obiektu `QRectF`. Framework Qt automatycznie odwzorowuje współrzędne sceny na współrzędne widoku. W istocie sceny wykorzystują współrzędne „okien” (logiczne), natomiast widoki używają współrzędnych „wziernika” (fizyczne). Tak więc, gdy pozycjonujemy elementy, rozmieszczamy je w kontekście współrzędnych sceny. Trzeci system współrzędnych jest wykorzystywany przez elementy. Jest to szczególnie wygodne, ponieważ jest to system współrzędnych logicznych, którego centralnym punktem jest punkt o współrzędnych (0, 0). Punkt (0,0) każdego elementu znajduje się w środkowym punkcie elementu na scenie (wyjątkiem są elementy tekstowe, dla których punktem (0,0) jest lewy górny róg tekstu). Oznacza to, że w praktyce zawsze możemy rysować elementy względem ich własnego punktu środkowego i nie musimy martwić się wszelkimi transformacjami, które zostały do nich zastosowane przez elementy-rodziców, ponieważ scena zrobi to za nas. Zwróćmy również uwagę na to, że we frameworku Qt współrzędne *y* wzrastają ku dołowi — na przykład punkt o współrzędnych (5, 8) jest o 6 pikseli *powyżej* punktu o współrzędnych (5, 14). Relacje zachodzące pomiędzy współrzędnymi sceny a współrzędnymi elementu zaprezentowano na rysunku 11.2.

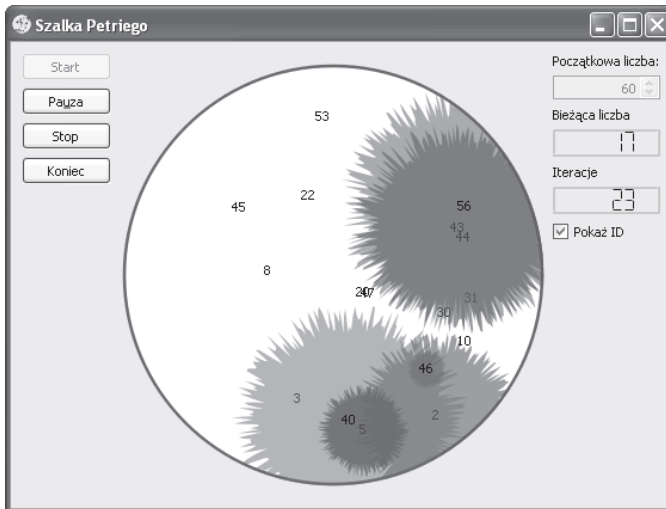


Rysunek 11.2. Elementy graficzne wykorzystują lokalne współrzędne logiczne

Pewne elementy architektury grafika-widok frameworka Qt zmieniły się pomiędzy wersjami Qt 4.5 a Qt 4.6. Zestawiono je w ramce „Zmiany działania architektury grafika-widok wprowadzone w Qt 4.6” (patrz strona 402).

Widżety i układ w architekturze grafika-widok

W tym punkcie przeanalizujemy aplikację *Szalka Petriego* (*petridish1*). Zrzut ekranu z tej aplikacji pokazano na rysunku 11.3. Aplikacja zawiera klasę `MainWindow`, która dziedziczy po klasie `QMainWindow` i wykorzystuje obiekt `QGraphicsView` w roli centralnego widżetu. *Szalka Petriego* jest aplikacją symulującą komórki — rosną one, jeśli są zbyt zatłoczone, i kurczą się, jeśli są zbyt odizolowane, zbyt zatłoczone albo za duże. Komórki mogą losowo umierać. Nie będziemy mówili zbyt wiele na temat samej symulacji czy też logiki aplikacji, ponieważ tematem tego rozdziału jest architektura grafika-widok frameworka Qt.



Rysunek 11.3. Aplikacja *Szalka Petriego*

Aby pokazać, jak można utworzyć główne okno aplikacji na bazie sceny graficznej, poniżej przeanalizujemy właściwe metody głównego okna (lub fragmenty tych metod). Natomiast w następnym punkcie przeanalizujemy elementy `Cell` (pochodne klasy `QGraphicsItem`). Skoncentrujemy się na podstawach tworzenia niestandardowych elementów graficznych oraz zaprezentowaniu interfejsu API klasy `QGraphicsItem`, natomiast pominiemy nieistotną logikę związaną z symulacją (kod źródłowy aplikacji można znaleźć w podkatalogu *petridish1*).

Aplikacja zawiera przyciski *Start*, *Pauza/Wznów*, *Stop* i *Koniec*, które pozwalają na sterowanie symulacją. Użytkownik może ustawić początkową liczbę komórek i określić, czy mają być wyświetlane identyfikatory komórek (są one przydatne w przypadku komórek, które bez wyświetlania identyfikatorów byłyby zbyt małe, aby można je było zobaczyć). Pole tekstowe początkowej liczby komórek w czasie działania symulacji jest zablokowane, co widać na zrzucie ekranu.

W interfejsie użytkownika wykorzystano kilka obiektów `QLCDNumbers`, aby pokazać, ile komórek pozostało oraz ile iteracji symulacji wykonano.

Zmiany działania architektury grafika-widok wprowadzone w Qt 4.6

W klasach architektury grafika-widok wprowadzono znaczące zmiany pomiędzy wersjami frameworka Qt 4.5 i Qt 4.6. W efekcie znacznie poprawiła się wydajność. Jedną z konsekwencji tych wprowadzonych „pod maską” zmian była konieczność dokonania pewnych zmian widocznych dla użytkownika. Było to niezbędne dla osiągnięcia najlepszej optymalizacji. Oto najważniejsze zmiany w działaniu:

- Publiczna zmienna klasy `QStyleOptionGraphicsItem` — `exposedRect` typu `QRectF` — zawiera udostępniony prostokąt elementu wyrażony we współrzędnych elementu. Zmienna ta jest jednak ustawiana tylko dla tych elementów graficznych, dla których ustawiono flagę `ItemUsesExtendedStyleOption`.
- Zmienne klasy `QStyleOptionGraphicsItem` — `levelOfDetail` i `matrix` — są przestarzałe. Prawidłowy sposób uzyskania poziomu szczegółów w Qt 4.6 bazuje na statycznej metodzie `QStyleOptionGraphicsItem::levelOfDetailFromTransform()`.
- Obiekty klasy `QGraphicsView` nie wywołują już metod `QGraphicsView::drawItems()` i `QGraphicsView::drawItem()` — o ile nie ustawimy flagi „optymalizacji” `QGraphicsView::IndirectPainting` (co jednak nie jest zalecane).
- Obiekty klasy `QGraphicsItem` nie emitują już sygnału `itemChange()` w przypadku zmian pozycji i wykonanych przekształceń. Aby uzyskać informacje o tych zmianach, należy ustawić flagę `QGraphicsItem::ItemSendsGeometryChanges` (ta flaga jest ustawiona domyślnie dla obiektów klasy `QGraphicsWidget` oraz `QGraphicsProxyWidget`).
- Nawet gdy flaga `ItemSendsGeometryChanges` jest ustawiona, sygnał `itemChange()` w przypadku wykonanej transformacji jest emitowany tylko wtedy, gdy zostanie wykorzystana metoda `setTransform()`. Począwszy od wydania Qt w wersji 4.7, oczekuje się, że jeśli będzie ustawiona ta flaga, to sygnał `itemChange()` będzie także wywołany w przypadku wywołania metod `setRotation()`, `setScale()` lub `setTransformOriginPoint()` (wszystkie wprowadzono w Qt 4.6).

To, w jakim stopniu — a nawet czy w ogóle — te zmiany wpłyną na konkretną aplikację, zależy od tego, z jakich własności architektury grafika-widok korzysta aplikacja. W przypadku przykładów przedstawionych w tej książce na aplikację *Projektant stron* z następnego rozdziału wpłynęła ostatnia zmiana działania z listy zamieszczonej powyżej.

Omawianie przykładu rozpoczniemy od analizy konstruktora głównego okna. Następnie przeanalizujemy kilka metod pomocniczych istotnych dla programowania w architekturze grafika-widok, która jest głównym tematem niniejszego rozdziału.

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), simulationState(Stopped), iterations(0)
{
    scene = new QGraphicsScene(this);
    scene->setItemIndexMethod(QGraphicsScene::NoIndex);

    createWidgets();
    createProxyWidgets();
    createLayout();
    createCentralWidget();
    createConnections();
}
```

```

startButton->setFocus();
setWindowTitle(QApplication::applicationName());
}

```

Sposób utworzenia obiektu `QGraphicsScene` jest dość osobliwy, ponieważ nie podaliśmy wymiarów sceny. Chociaż znamy potrzebną wysokość (musi być wystarczająca do tego, żeby zmieścić szalkę Petriego, plus pewien margines), to szerokość zależy od szerokości widżetów, dlatego wymiary ustawimy po utworzeniu i rozmieszczeniu widżetów.

Kiedy elementy są dodawane, przesuwane bądź usuwane ze sceny, wymagane jest wykonanie obliczeń położenia. Na przykład kiedy dodajemy element w widocznej części sceny, musimy go narysować. Z kolei kiedy widoczny element zostanie przesunięty lub usunięty, trzeba narysować ten fragment sceny, który był zakryty, a teraz jest odkryty. W przypadku scen z dużą liczbą statycznych elementów obliczenia te można znacząco przyspieszyć poprzez skorzystanie z metody indeksowania `QGraphicsScene::BspTreeIndex` (algorytm BSP — *Binary Space Partitioning*); ale w przypadku dynamicznych scen, gdzie wiele elementów jest dodawanych, przesuwanym bądź usuwanym, lepiej wyłączyć indeksowanie (tak jak zrobiliśmy w tym przykładzie), ponieważ koszty wykorzystania indeksów są zbyt wysokie w porównaniu z oszczędnościami, jakie one przynoszą.

Zgodnie ze stylem kodowania, jaki został przyjęty w całej książce, wywołujemy metody pomocnicze w konstruktorze w celu wykonania większości operacji związanych z inicjalizacją widżetu. Ponieważ używamy sceny graficznej jako głównego widżetu głównego okna, wszystkie metody pomocnicze są istotne. W związku z tym pokażemy i omówimy je wszystkie (pomijając w miarę możliwości powtarzające się fragmenty kodu).

```

void MainWindow::createWidgets()
{
    startButton = new QPushButton(tr("Start"));
    pauseOrResumeButton = new QPushButton(tr("Pa&uza"));
    pauseOrResumeButton->setEnabled(false);
    stopButton = new QPushButton(tr("Stop"));
    quitButton = new QPushButton(tr("Koniec"));

    QString styleSheet("background-color: bisque;");
    initialCountLabel = new QLabel(tr("Początkowa liczba:"));
    initialCountLabel->setStyleSheet(styleSheet);
    ...
    AQP::accelerateWidgets(QList<QWidget*>() << startButton
        << stopButton << quitButton << initialCountLabel
        << showIdsCheckBox);
}

```

Aplikacja wykorzystuje standardowe widżety `QWidget`. W sposobie ich tworzenia nie ma żadnych niespodzianek.

Jedyną niestandardową operacją, którą tu zastosowaliśmy, jest określenie dla widżetów (z wyjątkiem przycisków) arkusza stylów zapewniającego jednolity kolor tła. Dla przycisków nie określono arkuszy stylów, ponieważ wolimy, aby zachowały one wygląd zgodny z platformą i użytym motywem.

```

void MainWindow::createProxyWidgets()
{
    proxyForName["startButton"] = scene->addWidget(startButton);
    proxyForName["pauseOrResumeButton"] = scene->addWidget(
        pauseOrResumeButton);
    ...
}

```

Wszystkie widżety muszą być dodane do sceny, ponieważ widok sceny jest centralnym widżetem głównego okna. Z łatwością moglibyśmy zaadaptować inne podejście — na przykład wykorzystać w roli centralnego widżetu obiekt `QWidget` i przekazać do widżetu obiekt `QHBoxLayout`. Ten obiekt zawierałby obiekt `QVBoxLayout` z przyciskami, obiekt `QGraphicsView` oraz inny obiekt `QVBoxLayout` z innymi widżetami. Ale żeby pokazać, że można to zrobić, postanowiliśmy wykorzystać sam obiekt `QGraphicsView` jako centralny widżet i umieścić w nim wszystkie inne widżety, jak również elementy graficzne.

Aby dodać standardowe obiekty `QWidget` na scenie, należy utworzyć obiekt `QGraphicsProxyWidget` dla każdego obiektu `QWidget` i dodać obiekt proxy do sceny. W powyższej metodzie skorzystaliśmy z metody `QGraphicsScene::addWidget()`, która tworzy obiekt `QGraphicsProxyWidget` reprezentujący widżet przekazany w roli argumentu. Metoda zwraca wynik w postaci wskaźnika do widżetu proxy. Dla wygody utrzymujemy tablicę asocjacyjną, w której klucze są nazwami widżetu, a jej wartości wskaźnikami na widżety proxy. Każdy z utworzonych widżetów proxy dodajemy do tablicy asocjacyjnej (tablicę deklarujemy w pliku nagłówkowym jako `QHash<QString, QGraphicsProxyWidget*> proxyForName`).

Po utworzeniu widżetów i ich widżetów proxy możemy rozmieścić je na scenie. Przypomina to korzystanie ze standardowych układów frameworka Qt, z tym że trzeba używać klas sterowania układem specyficznych dla architektury grafika-widok. Metodę `createLayout()` przeanalizujemy w dwóch częściach. Najpierw omówimy tworzenie układów, a następnie przyjrzymy się ustawianiu wymiarów sceny.

```

const int DishSize = 350;
const int Margin = 20;

void MainWindow::createLayout()
{
    QGraphicsLinearLayout *leftLayout = new QGraphicsLinearLayout(
        Qt::Vertical);
    leftLayout->addItem(proxyForName["startButton"]);
    leftLayout->addItem(proxyForName["pauseOrResumeButton"]);
    leftLayout->addItem(proxyForName["stopButton"]);
    leftLayout->addItem(proxyForName["quitButton"]);

    QGraphicsLinearLayout *rightLayout = new QGraphicsLinearLayout(
        Qt::Vertical);
    foreach (const QString &name, QStringList(
        << "initialCountLabel" << "initialCountSpinBox"
        << "currentCountLabel" << "currentCountLCD"
        << "iterationsLabel" << "iterationsLCD"

```

```

    << "showIdsCheckBox")
    rightLayout->addItem(proxyForName[name]);

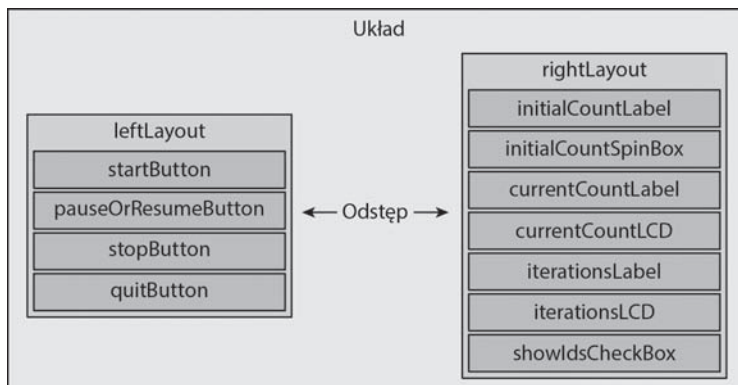
    QGraphicsLinearLayout *layout = new QGraphicsLinearLayout;
    layout->addItem(leftLayout);
    layout->setItemSpacing(0, DishSize + Margin);
    layout->addItem(rightLayout);

    QGraphicsWidget *widget = new QGraphicsWidget;
    widget->setLayout(layout);
    scene->addItem(widget);

```

`QGraphicsLinearLayout` jest klasą sterowania układem architektury grafika-widok będącej odpowiednikiem klasy `QBoxLayout`, z której wywodzą się klasy `QHBoxLayout` i `QVBoxLayout`. Interfejsy API są bardzo podobne. Różnica polega na tym, że zamiast dodawać widżety za pomocą metody `QBoxLayout::addWidget()`, korzystamy z metody `QGraphicsLinearLayout::addItem()`. Ta metoda dodaje do układu obiekt `QGraphicsLayoutItem` (który jest jednym z obiektów `QGraphicsWidget` — a tym samym obiektem klasy bazowej dla `QGraphicsProxyWidget`). Istnieje również klasa `QGraphicsGridLayout`, będąca odpowiednikiem klasy `QGridLayout`. W Qt w wersji 4.6 wprowadzono klasę `QGraphicsAnchorLayout`, która implementuje nowe podejście do sterowania układami — takie, którego nie spotykano we wcześniejszych wersjach Qt. Podejście to bazuje na rozmieszczaniu widżetów względem siebie oraz względem krawędzi i narożników prostokąta, które zajmują układ.

W tej metodzie tworzymy trzy obiekty `QGraphicsLinearLayout`. Pierwszy układ służy do utworzenia pionowej kolumny dla widżetów proxy przycisków z lewej strony, natomiast drugi do utworzenia pionowej kolumny widżetów proxy po prawej stronie. Trzeci służy do utworzenia ogólnego poziomego układu składającego się z lewej kolumny, odstępu (aby zapewnić miejsce dla właściwej szalki Petriego) oraz prawej kolumny. Układ głównego okna aplikacji schematycznie przedstawiono na rysunku 11.4.



Rysunek 11.4. Układ głównego okna aplikacji Szalka Petriego

Po utworzeniu układów tworzymy nowy „pusty” obiekt `QGraphicsWidget`. Ta klasa nie ma wizualnej reprezentacji sama w sobie i jest specjalnie zaprojektowana zarówno do odgrywania roli klasy bazowej dla niestandardowych widżetów architektury grafika-widok, jak i do tego celu, do jakiego używamy jej tutaj — jest kontenerem jednego lub większej liczby widżetów-dzieci rozmieszczonych w układzie dokumentu. Po utworzeniu widżetu konfigurujemy ogólny układ okna i dodajemy widżet do sceny. W efekcie wszystkie obiekty zarządzania układem i widżety proxy otrzymują nowego rodzica — na przykład widżety proxy stają się dziećmi sceny (widżety są przyporządkowane do swoich proxy w wywołaniach `QGraphicsScene::addWidget()`).

```
int width = qRound(layout->preferredWidth());
int height = DishSize + (2 * Margin);
setMinimumSize(width, height);
scene->setSceneRect(0, 0, width, height);
}
```

Scenę ustawiamy na taką szerokość, aby układ wyświetlał się w swojej preferowanej szerokości i wysokości, wystarczającej do tego, by wyświetlić szalkę Petriego, pozostawiając pewien margines w pionie. Ustawiamy także minimalny rozmiar głównego okna w taki sposób, by nigdy nie skurczyło się do takich wartości, które nie pozwalają na prawidłowe wyświetlanie szalki Petriego i jego widżetów.

```
void MainWindow::createCentralWidget()
{
    dishItem = new QGraphicsEllipseItem;
    dishItem->setFlags(QGraphicsItem::ItemClipsChildrenToShape);
    dishItem->setPen(QPen(QColor("brown"), 2.5));
    dishItem->setBrush(Qt::white);
    dishItem->setRect(pauseOrResumeButton->width() + Margin,
                    Margin, DishSize, DishSize);

    scene->addItem(dishItem);

    view = new QGraphicsView(scene);
    view->setRenderHints(QPainter::Antialiasing|
                       QPainter::TextAntialiasing);
    view->setBackgroundBrush(QColor("bisque"));
    setCentralWidget(view);
}
```

Ta metoda jest wywoływana po utworzeniu sceny i wypełnieniu jej widżetami (a ściślej widżetami proxy). Tworzy ona szalkę Petriego i widok, co kończy konfigurację wyglądu aplikacji.

Zaczynamy od utworzenia nowego graficznego elementu elipsy — choć w tym przypadku będzie to koło, ponieważ ustawiliśmy jego szerokość i wysokość na tę samą wartość. Dla elementu ustawiamy opcję obcinania elementów potomnych. Wszystkie symulowane komórki są tworzone jako dzieci szalki Petriego. To gwarantuje, że wszystkie komórki znajdujące się poza szalką Petriego nie są wyświetlane, a wszystkie komórki, które przekra-

czają granice naczynia, mają widoczną tylko tę część, która mieści się w naczyniu. Prostopąt szalki Petriego ustawiamy w taki sposób, aby jego współrzędna x była równa szerokości jednego z przycisków w lewej kolumnie powiększonej o pewien margines, a jego współrzędna y zapewniała niewielki margines nad szalką. Po utworzeniu elementu szalki dodajemy go do sceny.

Tworzymy standardowy obiekt `QGraphicsView` z włączoną opcją antyaliasingu oraz z takim samym kolorem tła, jaki ustawiliśmy w arkuszu stylów dla niektórych widżetów. Następnie ustawiamy go jako centralny widżet głównego okna. Na tym kończymy konfigurowanie wyglądu aplikacji.

Pod względem struktury używanie architektury grafika-widok do ustawiania widżetów głównego okna nie różni się zbytnio od bardziej konwencjonalnego podejścia. Jedyna znacząca różnica polega na tym, że musimy utworzyć i dodać widżety proxy dla właściwych widżetów oraz musimy korzystać ze specyficznych klas zarządzania układem architektury grafika-widok zamiast standardowych klas sterowania układem. Oczywiście gdybyśmy chcieli użyć obiektów wywodzących się z klasy `QGraphicsWidget`, nie musieliśmy tworzyć dla nich widżetów proxy, ponieważ można je bezpośrednio dodawać do sceny (w czasie, kiedy powstawała ta książka, jedyną dostępną podklasą klasy `GraphicsWidget` oprócz klasy `QGraphicsProxyWidget` była klasa `QGraphicsWebView`, chociaż bez trudu moglibyśmy utworzyć własne podklasy klasy `QGraphicsWidget`, gdybyśmy tego chcieli).

```
void MainWindow::createConnections()
{
    connect(startButton, SIGNAL(clicked()), this, SLOT(start()));
    connect(pauseOrResumeButton, SIGNAL(clicked()),
           this, SLOT(pauseOrResume()));
    connect(stopButton, SIGNAL(clicked()), this, SLOT(stop()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(showIdsCheckBox, SIGNAL(toggled(bool)),
           this, SLOT(showIds(bool)));
}
```

Ta metoda jest bardzo podobna do tych, z którymi wielokrotnie spotykaliśmy się we wcześniejszych rozdziałach — zawiera połączenia pomiędzy sygnałami `clicked()` widżetów (rzeczywistych) a odpowiadającymi im slotami. Nie ma ona znaczenia dla jako takiego programowania w architekturze grafika-widok, ale została tutaj przedstawiona jako kontrast z wersją z Qt 4.6 z przykładu pokazanego w rozdziale 13. Tamta metoda korzysta z obiektu `QStateMachine` do zarządzania działaniem aplikacji, a dzięki temu zawiera mniejszą liczbę slotów i ma prostszą logikę.

Wprowadzenie do elementów graficznych

Klasa `QGraphicsItem` jest klasą bazową dla wszystkich elementów graficznych. Chociaż klasa ta zawiera bardzo wiele metod — ponad dwieście w Qt 4.6 — nie można tworzyć jej egzemplarzy, ze względu na istnienie dwóch czysto wirtualnych metod: `boundingRect()`

i `paint()`. Metoda `paint()` jest odpowiednikiem metody `QWidget::paintEvent()` i trzeba ją zaimplementować, aby narysować element. Metoda `boundingRect()` przekazuje do architektury grafika-widok prostokąt okalający element — jest on wykorzystywany do detekcji kolizji oraz do tego, by zapewnić rysowanie elementu tylko wtedy, gdy jest on widoczny we wznierniku obiektu `QGraphicsView`.

W przypadku tworzenia nieprostokątnych, niestandardowych elementów graficznych, najlepiej zaimplementować również metodę `shape()`. Metoda ta zwraca obiekt `QPainterPath`, który dokładnie opisuje obrys elementu. Jest to przydatne do dokładnego wykrywania kolizji oraz wykrywania kliknięć myszą.

Istnieje wiele metod wirtualnych, które można zaimplementować łącznie z metodami `advance()`, `boundingRect()`, `collidesWithItem()`, `collidesWithPath()`, `contains()`, `isObscuredBy()`, `opaqueArea()`, `paint()`, `shape()` i `type()`. Wszystkie metody chronione (z wyjątkiem metody `prepareGeometryChange()`) są również wirtualne, a zatem można utworzyć własną implementację wszystkich procedur obsługi zdarzeń elementów graficznych (łącznie z `contextMenuEvent()`, `keyPressEvent` oraz zdarzeniami obsługi myszy). Wszystkie je zwięźle opisano w tabeli 11.1.

Aby utworzyć niestandardową figurę, najłatwiej skorzystać ze standardowych podklas klasy `QGraphicsItem`, takich jak `QGraphicsPathItem` lub `QGraphicsPolygonItem`. A jeśli dodatkowo chcemy, aby figura charakteryzowała się własnym zachowaniem, możemy utworzyć podklasę elementu i utworzyć własną implementację niektórych chronionych procedur obsługi zdarzeń, takich jak `keyPressEvent()` i `mousePressEvent()`. Jeśli wolimy rysować samodzielnie, możemy bezpośrednio utworzyć podklasę klasy `QGraphicsItem` i zaimplementować metody `boundingRect()`, `paint()` i `shape()` oraz wszystkie procedury obsługi zdarzeń potrzebne do zapewnienia działań, które nas interesują. Dla wszystkich podklas klasy `QGraphicsItem` najlepiej zdefiniować typ wyliczeniowy `Type` oraz własną implementację metody `type()`. Zagadnienie to omówimy za chwilę.

W tym punkcie zwięźle omówimy te aspekty klasy `Cell` — bezpośredniej podklasy klasy `QGraphicsItem` — z aplikacji *Szalka Petriego*, które są związane z programowaniem w architekturze grafika-widok. Rozpoczniemy od definicji w pliku nagłówkowym, ale pominiemy sekcję prywatną.

```
class Cell : public QGraphicsItem
{
public:
    enum {Type = UserType + 1};

    explicit Cell(int id, QGraphicsItem *parent=0);

    QRectF boundingRect() const { return m_path.boundingRect(); }
    QPainterPath shape() const { return m_path; }
    void paint(QPainter *painter,
              const QStyleOptionGraphicsItem *option, QWidget *widget);
    int type() const { return Type; }
    ...
};
```

Tabela 11.1. API klasy `QGraphicsItem` (wybrane metody)

Metoda	Opis
<code>advance()</code>	Należy utworzyć własną implementację, aby wykonywać animacje. Można również stosować inne podejścia (przykłady można znaleźć w rozdziałach 12. i 13.).
<code>boundingRect()</code>	Własną implementację należy utworzyć w celu uzyskania prostokąta otaczającego element, wyrażonego we współrzędnych elementu — patrz <code>sceneBoundingRect()</code> i <code>shape()</code> .
<code>childItems()</code>	Zwraca listę bezpośrednich potomków elementu (Qt 4.4).
<code>collidesWithItem(QGraphicsItem *, Qt::ItemSelectionMode)</code>	Zwraca <code>true</code> , jeśli ten element koliduje z podanym elementem zgodnie z podanym trybem — patrz typ wyliczeniowy <code>Qt::ItemSelectionMode</code> (strona 414).
<code>collidesWithPath(QPainterPath, Qt::ItemSelectionMode)</code>	Zwraca <code>true</code> , jeśli ten element koliduje z podaną ścieżką, zgodnie z podanym trybem.
<code>collidingItems(Qt::ItemSelectionMode)</code>	Zwraca listę wszystkich elementów, z którymi koliduje ten element, zgodnie z podanym trybem.
<code>contains(QPointF)</code>	Zwraca <code>true</code> , jeśli podany punkt należy do elementu.
<code>ensureVisible()</code>	Wymusza od wszystkich widoków <code>QGraphicsView</code> powiązanych ze sceną i zawierających ten element, aby przewinęły się, jeśli zachodzi taka potrzeba, w celu wyświetlenia tego elementu.
<code>group()</code>	Zwraca obiekt <code>QGraphicsItemGroup</code> , do którego należy ten element, lub 0, jeśli nie należy on do żadnej grupy.
<code>hide()</code>	Ukrywa element — patrz <code>show()</code> i <code>setVisible()</code> .
<code>isObscuredBy(QGraphicsItem*)</code>	Zwraca <code>true</code> , jeśli prostokąt otaczający tego elementu jest całkowicie zasłonięty przez kształt podanego nieprzezroczystego elementu.
<code>isSelected()</code>	Zwraca <code>true</code> , jeśli podany element jest zaznaczony.
<code>isVisible()</code>	Zwraca <code>true</code> , jeśli ten element jest <i>logicznie</i> widoczny (nawet gdy jest całkowicie zasłonięty albo znajduje się poza wziernikiem widoku).
<code>keyPressEvent(QKeyEvent*)</code>	Należy zaimplementować tę metodę w celu obsługi naciśnięć klawiszy na elemencie. Metoda będzie wywołana tylko wtedy, gdy ustawiono flagę <code>ItemIsFocusable</code> .
<code>mouseDoubleClickEvent(QGraphicsSceneMouseEvent*)</code>	Należy zaimplementować tę metodę w celu obsługi dwukrotnych kliknięć.
<code>mouseMoveEvent(QGraphicsSceneMouseEvent*)</code>	Należy zaimplementować tę metodę w celu obsługi ruchów myszą.
<code>mousePressEvent(QGraphicsSceneMouseEvent*)</code>	Należy zaimplementować tę metodę w celu obsługi kliknięć przycisku myszy.
<code>moveBy(qreal, qreal)</code>	Przesuwa element o podane wartości w pionie i w poziomie.

Tabela 11.1. API klasy `QGraphicsItem` (wybrane metody) — ciąg dalszy

Metoda	Opis
<code>opaqueArea()</code>	Należy zaimplementować tę metodę w celu uzyskania ścieżki obiektu <code>painter</code> pokazującej miejsca, w których element jest niewidoczny. Ścieżka jest w takiej postaci, w jakiej jest używana w metodzie <code>isObscuredBy()</code> .
<code>paint(QPainter*, QStyleOptionGraphicsItem*, QWidget*)</code>	Należy zaimplementować tę metodę w celu narysowania elementu — patrz <code>boundingRect()</code> i <code>shape()</code> .
<code>parentItem()</code>	Zwraca rodzica elementu lub 0.
<code>pos()</code>	Zwraca pozycję elementu wyrażoną we współrzędnych rodzica, a jeśli element nie ma rodzica — we współrzędnych sceny (patrz <code>scenePos()</code>).
<code>prepareGeometryChange()</code>	Tę metodę <i>trzeba</i> wywołać przed modyfikacją prostokąta otaczającego element. Metoda automatycznie wywoła metodę <code>update()</code> .
<code>resetTransform()</code>	Resetuje macierz transformacji elementu na macierz tożsamościową. Powoduje to eliminację obrotów, skalowania lub przycinania.
<code>rotation()</code>	Zwraca obrót elementu wyrażony w stopniach (-360.0° , 360.0°). Wartość domyślna wynosi 0.0° (Qt 4.6).
<code>scale()</code>	Zwraca współczynnik skalowania elementu. Wartość domyślna wynosi 1.0, co oznacza, że element nie jest skalowany (Qt 4.6).
<code>scene()</code>	Zwraca scenę, do której należy element, lub 0, jeżeli element nie został dodany do sceny.
<code>sceneBoundingRect()</code>	Zwraca prostokąt otaczający elementu wyrażony we współrzędnych sceny — patrz <code>boundingRect()</code> .
<code>scenePos()</code>	Zwraca pozycję elementu wyrażoną we współrzędnych sceny — dla elementów bez rodzica wartość jest taka sama jak zwracana przez metodę <code>pos()</code> .
<code>setFlag(GraphicsItemFlag, bool)</code>	Włącza lub wyłącza flagę w zależności od wartości przekazanego parametru typu <code>Boolean</code> (domyślnie włączona).
<code>setFlags(GraphicsItemFlags)</code>	Ustawia flagi (z wykorzystaniem operatora OR) — zobacz typ wyliczeniowy <code>QGraphicsItem::GraphicsItemFlag</code> (patrz strona 414).
<code>setGraphicsEffect(QGraphicsEffect*)</code>	Ustawia podany efekt graficzny dla elementu (usuwając efekty ustawione poprzednio). Dotyczy efektów <code>QGraphicsBlurEffect</code> , <code>QGraphicsDropShadowEffect</code> i <code>QGraphicsOpacityEffect</code> (Qt 4.6).
<code>setGroup(QGraphicsItemGroup*)</code>	Dodaje ten element do podanej grupy.
<code>setParentItem(QGraphicsItem*)</code>	Ustawia (lub zmienia) rodzica elementu na podany element.

Tabela 11.1. API klasy *QGraphicsItem* (wybrane metody) — ciąg dalszy

Metoda	Opis
<code>setPos(QPointF)</code>	Ustawia pozycję elementu wyrażoną we współrzędnych rodzica. Istnieje również przeciążona wersja, która akceptuje dwa argumenty typu <code>qreal</code> .
<code>setRotation(qreal)</code>	Ustawia obrót elementu na podaną liczbę stopni (-360.0° , 360.0°) (Qt 4.6).
<code>setScale(qreal)</code>	Skaluje element. Wartość 1.0 oznacza brak skalowania (Qt 4.6).
<code>setSelected(bool)</code>	Zaznacza lub anuluje zaznaczenie elementu w zależności od argumentu typu <code>Boolean</code> .
<code>setToolTip(QString)</code>	Ustawia tekst wskazówki ekranowej dla elementu.
<code>setTransform(QTransform, bool)</code>	Ustawia macierz transformacji elementu na podaną wartość lub łączy ją z podaną wartością, jeśli argument typu <code>Boolean</code> ma wartość <code>true</code> (Qt 4.3). Istnieje również zupełnie inna metoda <code>setTransformations()</code> .
<code>setVisible(bool)</code>	Ukrywa lub pokazuje element w zależności od przekazanego argumentu typu <code>Boolean</code> .
<code>setX(qreal)</code>	Ustawia pozycję x elementu we współrzędnych jego rodzica (Qt 4.6).
<code>setY(qreal)</code>	Ustawia pozycję y elementu we współrzędnych jego rodzica (Qt 4.6).
<code>setZValue(qreal)</code>	Ustawia wartość współrzędnej z elementu.
<code>shape()</code>	Należy zaimplementować tę metodę w celu uzyskania ścieżki obiektu <code>painter</code> opisującej dokładny kształt elementu — patrz <code>boundingRect()</code> i <code>paint()</code> .
<code>show()</code>	Wyświetla element — patrz <code>hide()</code> i <code>setVisible()</code> .
<code>toolTip()</code>	Zwraca wskazówkę ekranową powiązaną z elementem.
<code>transform()</code>	Zwraca macierz transformacji elementu. Istnieje również metoda <code>transformations()</code> .
<code>type()</code>	Zwraca właściwość <code>QGraphicsItem::Type</code> elementu w postaci wartości <code>int</code> . Niestandardowe podklasy klasy <code>QGraphicsItem</code> powinny zawierać własną implementację tej metody oraz definicję typu wyliczeniowego <code>Type</code> .
<code>update()</code>	Inicjuje zdarzenie rysowania dla elementu.
<code>x()</code>	Zwraca pozycję x elementu wyrażoną we współrzędnych jego rodzica.
<code>y()</code>	Zwraca pozycję y elementu wyrażoną we współrzędnych jego rodzica.
<code>zValue()</code>	Zwraca wartość współrzędnej z elementu.

Chociaż dostarczenie własnej implementacji metody `type()` lub typu wyliczeniowego `Type` nie jest obowiązkowe, zalecamy dostarczenie obu tych implementacji dla wszystkich podklas reprezentujących niestandardowe elementy graficzne. Dzięki temu w łatwy sposób zidentyfikujemy typy niestandardowych elementów graficznych, a poza tym umożliwimy ich działanie z metodą `qgraphicsitem_cast<>()` — która rzutuje wskaźniki `QGraphicsItem` na wskaźniki na odpowiednie podklasy klasy `QGraphicsItem`. (Funkcja `qgraphicsitem_cast<>()` obsługuje tylko operacje rzutowania ze wskaźników `QGraphicsItem` na podklasy, a nie z podklas na wskaźniki `QGraphicsItem`. W przypadku operacji rzutowania z powrotem na wskaźniki `QGraphicsItem` trzeba wykorzystać inne techniki. Operacje rzutowania elementów graficznych omówimy w dalszej części tej książki — w następnym rozdziale — patrz strona 431).

W tym konkretnym przykładzie mamy prywatną zmienną składową `m_path` typu `QPainterPath` (która dynamicznie zmienia figurę w miarę postępów symulacji). Ponieważ mamy tę ścieżkę, możemy ją wykorzystać do dostarczenia zarówno prostokąta otaczającego element, jak i jego kształtu. Zwróćmy jednak uwagę, że obliczenie prostokąta otaczającego na podstawie ścieżki obiektu `painter` nie działa szczególnie szybko, choć jest wystarczająco szybkie dla aplikacji *Szalka Petriego*. W innych aplikacjach, które korzystają ze ścieżki obiektu `painter` w taki sposób, mogą być wykorzystywane mechanizmy buforowania prostokąta otaczającego ścieżki.

Metoda `shape()` jest trywialna do zaimplementowania, ponieważ jak zobaczymy za chwilę, ścieżki rysowane są nie za pomocą pióra, tylko za pomocą pędzla. Gdybyśmy rysowali ścieżkę za pomocą grubego pióra — na przykład w celu narysowania elementu o kształcie pączka na podstawie ścieżki w kształcie elipsy — to otrzymany kształt nie byłby dokładny, ponieważ nie uwzględniałby grubości obrysu. Mogłoby to znaczyć, że gdyby użytkownik kliknął obrys, nie byłoby żadnej reakcji, ponieważ jest on „poza” elipsą. W takich przypadkach można utworzyć obiekt `QPainterPathStroker`, skonfigurować go za pomocą metod do manipulowania piórem (`setWidth()`, `setJoinStyle()` itp.), a następnie wywołać metodę `QPainterPathStroker::createStroke()`, przekazując w roli argumentu ścieżkę obiektu `painter`. Wartość zwracana przez metodę `createStroke()` jest nową ścieżką obiektu `painter`, która określa obrys pierwotnej ścieżki, ale z wykorzystaniem ustawień skonfigurowanych dla pióra.

W przypadku klasy `Cell` działaniem konstruktora (którego tu nie pokazano) jest ustawienie pędzla oraz początkowego rozmiaru, a następnie wywołanie prywatnej metody (również jej nie pokazano) w celu utworzenia początkowej figury. W ten sposób metoda `paint()` staje się znacznie prostsza, niż byłaby w innej sytuacji, ponieważ jedyne jej zadania to narysowanie ścieżki i opcjonalnie identyfikatora elementu.

```
void Cell::paint(QPainter *painter,
                const QStyleOptionGraphicsItem *option, QWidget*)
{
    painter->setPen(Qt::NoPen);
    painter->setBrush(m_brush);
    painter->drawPath(m_path);
    if (s_showIds) {
```

```
    QPointF center = m_path.boundingRect().center();
    QString id = QString::number(m_id);
    center.setX(center.x() - (option->fontMetrics.width(id) / 2));
    center.setY(center.y() + (option->fontMetrics.height() / 4));
    painter->setPen(QPen());
    painter->drawText(center, id);
}
}
```

Zaczynamy od skonfigurowania pióra obiektu painter (Qt::NoPen oznacza, że obrys nie zostanie narysowany) oraz jego pędzla, a następnie rysujemy ścieżkę komórki (klasa Cell zawiera również statyczną zmienną Boolean `s_showIds`, z kilkoma statycznymi metodami dostępowymi, oraz zmienną składową ID — `m_id` typu `int` — żadnej z nich nie pokazano). Jeśli trzeba wyświetlić identyfikator elementu, znajdujemy środek ścieżki i używając obiektu `QPen()`, rysujemy identyfikator wyrównany do środka w poziomie, a w pionie ustawiony w jednej czwartej wysokości, licząc od góry. Domyślny konstruktor klasy `QPen` generuje *kosmetyczne pióro* (ang. *cosmetic pen*) o czarnej ciągłej linii grubości 1 piksela. Pióro jest opisywane jako „kosmetyczne”, jeśli ignoruje transformacje.

Na przykład parametr `QStyleOptionGraphicsItem*` przechowuje prostokąt udostępniony przez element, metryki czcionki — po to właśnie użyliśmy go w tym przykładzie — oraz paletę. Parametr `QWidget*` jest rzadko używany.

Implementacja metody `paint()` jest wystarczająco szybka dla aplikacji *Szalka Petriego*, ale jest daleka od optymalnej. Wydaje się, że nie warto buforować prostokąta otaczającego ścieżkę, ponieważ obiekty `Cell` kurczą się lub rozrastają przy każdej iteracji symulacji. Natomiast identyfikatory komórek nigdy się nie zmieniają. W związku z tym możemy poświęcić niewielką ilość pamięci, by zyskać na szybkości i przechowywać w niej prywatną zmienną składową `m_idString` typu `QString`, którą tworzymy w konstruktorze. W ten sposób unikamy korzystania wewnątrz metody `paint()` z metody `QString::number()`, która przy każdym wywołaniu alokuje pamięć. Obliczanie szerokości i wysokości metryk czcionki również jest wolne. Z łatwością możemy obliczyć te wartości w konstruktorze i buforować wyniki. Najlepszą zasadą jest po prostu uruchomić rysowanie, a następnie, jeśli okaże się zbyt wolne, znaleźć elementy, które można buforować. Naturalnie najlepiej mierzyć efekty wprowadzanych zmian, aby mieć pewność, że oczekiwane korzyści rzeczywiście zostały osiągnięte.

W klasie `Cell` nie zaimplementowano żadnych procedur obsługi zdarzeń ani nie ustawiono żadnych flag (na przykład `ItemIsMovable` lub `ItemIsSelectable`), dlatego użytkownicy nie mogą bezpośrednio wykonywać działań na elementach typu `Cell`. Przykłady tego, w których miejscach ustawiamy te flagi oraz gdzie implementujemy procedury obsługi zdarzeń, zaprezentujemy w następnym rozdziale. Ostatnią część tego rozdziału poświęcono na przedstawienie tabel prezentujących interfejs API klasy `QGraphicsItem`. W tabeli 11.1 zestawiono metody klasy `QGraphicsItem`, natomiast w tabelach 11.2 i 11.3 wymieniono najważniejsze typy wyliczeniowe wykorzystywane w tych metodach.

Tabela 11.2. Typ wyliczeniowy Qt::ItemSelectionMode

Typ wyliczeniowy	Opis
Qt::ContainsItemShape	Zaznaczanie elementów, których kształty w całości mieszczą się w zaznaczonym obszarze.
Qt::IntersectsItemShape	Zaznaczanie elementów, których kształty w całości mieszczą się w zaznaczonym obszarze albo przecinają ten obszar.
Qt::ContainsItemBoundingRect	Zaznaczanie elementów, których prostokąty otaczające w całości mieszczą się w zaznaczonym obszarze.
Qt::IntersectsItemBoundingRect	Zaznaczanie elementów, których prostokąty otaczające w całości mieszczą się w zaznaczonym obszarze albo przecinają ten obszar.

Tabela 11.3. Typ wyliczeniowy QGraphicsItem::GraphicsItemFlag

Typ wyliczeniowy	Opis
QGraphicsItem::ItemAcceptsInputMethod	Element obsługuje metody wprowadzania (Qt 4.6).
QGraphicsItem::ItemClipsChildrenToShape	Element obcina wszystkie swoje dzieci (rekurencyjnie) do swojego własnego kształtu (Qt 4.3).
QGraphicsItem::ItemClipsToShape	Element jest obcinany do swojego własnego kształtu niezależnie od sposobu rysowania. Poza swoim kształtem nie może także odbierać zdarzeń (np. kliknięć myszą) (Qt 4.3).
QGraphicsItem::ItemDoesntPropagate ↳OpacityToChildren	Element nie propaguje swojego pokrycia na potomków (Qt 4.5).
QGraphicsItem::ItemHasNoContents	Element nie rysuje niczego (Qt 4.6).
QGraphicsItem::ItemIgnoresParentOpacity	Pokrycie elementu ma wartość, na jaką zostało ustawione, a nie wartość w połączeniu ze swoim rodzicem (Qt 4.5).
QGraphicsItem::ItemIgnoresTransformations	Element ignoruje transformacje zastosowane do swojego rodzica (choć jego pozycja jest powiązana do rodzica). Przydatne dla elementów, które są wykorzystywane jako etykiety tekstowe (Qt 4.3).
QGraphicsItem::ItemIsFocusable	Element akceptuje naciśnięcia klawiszy.
QGraphicsItem::ItemIsMovable	Element (oraz rekurencyjnie jego dzieci) może być przeniesiony poprzez kliknięcie i przeciągnięcie.
QGraphicsItem::ItemIsPanel	Element jest panelem (Qt 4.6). Więcej informacji na temat paneli można znaleźć w dokumentacji online.
QGraphicsItem::ItemIsSelectable	Element można zaznaczyć poprzez kliknięcie, operację „spięcia gumką” (ang. <i>rubber band</i>) lub za pomocą wywołania <code>QGraphicsScene::setSelectionArea()</code> .
QGraphicsItem::ItemNegativeZStacks ↳BehindParent	Jeśli wartość współrzędnej z jest ujemna, element automatycznie chowa się za swoim rodzicem.

Tabela 11.3. Typ wyliczeniowy `QGraphicsItem::GraphicsItemFlag` — ciąg dalszy

Typ wyliczeniowy	Opis
<code>QGraphicsItem::ItemSendsGeometryChanges</code>	Element wywołuje metodę <code>itemChange()</code> dla zmian pozycji i transformacji (Qt 4.6). Zobacz też ramka „Zmiany działania architektury grafika-widok wprowadzone w Qt 4.6” (patrz strona 402).
<code>QGraphicsItem::ItemSendsScenePosition</code> ↳ <code>Changes</code>	Element wywołuje metodę <code>itemChange()</code> dla zmian pozycji (Qt 4.6).
<code>QGraphicsItem::ItemStacksBehindParent</code>	Element jest umieszczony za swoim rodzicem zamiast przed nim (co jest działaniem domyślnym). Przydatne do tworzenia efektu cienia.
<code>QGraphicsItem::ItemUsesExtendedStyle</code> ↳ <code>Option</code>	Element uzyskuje dostęp do dodatkowych atrybutów klasy <code>QStyleOptionGraphicsItem</code> .

Na tym zakończyliśmy przegląd aplikacji *Szalka Petriego* oraz API klasy `QGraphicsItem`. W kodzie źródłowym przykładu jest kilka szczegółów mniejszego znaczenia, których nie omówiliśmy. Na przykład po każdej iteracji wykorzystujemy jednorazowy timer do zainicjowania następnej iteracji. Nie możemy skorzystać z obiektu klasy `QTimer` o stałym interwale czasowym, ponieważ czas potrzebny na wykonanie obliczeń w każdej iteracji jest inny. Poza tym w momencie zatrzymania aplikacji całe okno staje się półprzezroczyste. Efekt ten najlepiej wygląda w systemie Windows.

W następnym rozdziale przeanalizujemy aplikację, która w bardziej konwencjonalny sposób korzysta z architektury grafika-widok. Zaprezentujemy też więcej przykładów tego, jak tworzyć własne elementy graficzne, a także jak zapisać i załadować sceny do i z plików, jak manipulować elementami na scenach — na przykład wykonywać transformacje, a także kopiować, wycinać i wklejać.

Skorowidz

Wszystkie funkcje nieglobalne i metody są wymienione w ramach swojej klasy (albo swojej klasy bazowej — na przykład QWidget lub QObject) oraz jako osobne pojęcia najwyższego poziomu. Tam, gdzie nazwa metody lub funkcja jest bliska określonemu pojęciu, samo pojęcie nie jest zwykle wymieniane. Na przykład nie ma terminu „łączenie listy ciągów”, ale jest termin QString::join(). Należy także zwrócić uwagę, że wiele odniesień zamieszczono wyłącznie w celu zacytowania kodu (tzn. pokazania przykładów użycia).

A

- akceleratory klawiaturowe, 26
- akcja, 438
 - editHideOrShowDoneTasksAction, 151
 - fileNewAction, 189
 - Kopiuj, 436, 444
 - Save, 445
 - triggered(), 355
 - viewShowGridAction, 443
 - wyrównania, 439
 - zoomOutAction, 36, 39
- aktualizacja wyświetlacza, 86
- akumulator, 286
- algorytm
 - BSP, 403
 - SHA1, 323
 - sumowania Kahana, 288
- animacje, 398, 475
- animowane właściwości, 495
- antialiasing, 229
- API
 - architektury model-widok, 176
 - klasy
 - QStandardItemModel, 127, 168–170, 177
 - QStandardItemView, 217, 232
 - QGraphicsItem, 401, 409–412, 415
 - QStandardItemModel, 164
 - QStyledItemDelegate, 202
 - QTextCursor, 335–338, 375
 - aplikacja
 - Dialog znajdź, 487
 - Dzingle, 64
 - Generator próbek, 369, 381, 389
 - Image2Image, 257
 - Kody pocztowe, 100
 - MPlayer, 70
 - numbergrid, 284
 - Odtwarzacz muzyki, 75, 88
 - Odtwarzacz wideo, 89, 94
 - Projektant stron, 417–74
 - przeglądarki recenzji książek, 42, 45
 - Przejście krzyżowe, 296, 301, 316
 - Rejestrator czasu, 168, 196
 - RssPanel, 31
 - Siatka liczb, 271, 277, 279
 - Szałka Petriego, 401, 415, 481
 - timelog1, 141, 163
 - timelog2, 189
 - VLC, 70
 - Weather Tray Icon, 19, 28, 32

aplikacja
 Widok folderów, 197
 Wizualizator, 233
 Wizualizator spisu, 232
 zipcodes1, 100, 124
 Znajdź duplikaty, 310, 323

aplikacje
 desktopowe, 17
 desktopowo-internetowe, 18
 Qt, 18
 specyficzne dla witryny, 41

architektura
 frameworka Phonon, 74
 grafika-widok, 397, 417, 454
 model-widok, 75, 97, 139, 167, 193, 215

arkusze, sheets, 71
 arkusze CSS, 345
 asercja `Q_ASSERT()`, 208
 atak Denial of Service, 27

B

backend, 70
 bezpieczeństwo wątków, 314, 315
 biblioteka
 Boost, 72
 DirectShow, 70
 DirectX, 70
 GStreamer, 70
 Phonon, 63, 70
 QtCore, 355
 Qwt, 497
 Source-Highlight, 345
 STL, 440

biblioteki backend, 70
 blok try...catch, 144
 blokada
 odczytu, 313
 zapisu, 313

blokady mutowalne, 313
 blokowanie muteksa, 276
 błąd parsowania XML, 157
 błędy, 275
 BSP, Binary Space Partitioning, 403

C

CSS, 345, 375
 CSS, Cascading Style Sheets, 32, 325

czas trwania animacji, 476
 czcionka, 219, 391

D

degradowanie elementu, 188
 delegat, 193
 QDateTimeDelegate, 197
 ItemDelegate, 100
 QStyledItemDelegate, 98, 201
 RichTextDelegate, 202, 208

delegaty
 dla kolumn lub wierszy, 201
 niestandardowe, 230
 specyficzne dla modelu, 208
 specyficzne dla typów danych, 196
 tylko do odczytu, 197

deserializacja, 177
 dodawanie elementów, 431, 433
 dokument QTextDocument, 372
 dokumentacja biblioteki, 12
 dokumenty
 PDF, 394
 PostScript, 394
 SVG, 395
 z grafiką rastrową, 395

DOM, Document Object Model, 18

dostęp
 do elementów DOM, 54
 do obiektów QObject, 58
 do odczytu, 276
 do sieci, 23
 do zapisu, 276

drukowanie, 384
 drukowanie dokumentów, 379

drzewo
 metody obsługi ładowania, 189
 metody obsługi zapisywania, 189
 przemieszczanie elementów, 182
 TreeWidgetItem, 76

duplikaty plików, 310

dyrektywa
`#if QT_VERSION`, 13, 111
`#ifdef`, 123, 145, 161
`DEBUG`, 37

działania na macierzach, 55
 działanie architektury grafika-widok, 402

E

edytor, 194
 globalny, 194
 jednowierszowy, 350
 sformatowanego tekstu, 325, 350, 361
 SpinBox, 195
 wielowierszowy, 361
 efekt przejścia krzyżowego, 297
 eksportowanie, 370
 do formatów rastrowych, 383
 do formatu PDF, 380
 do formatu PostScript, 380
 dokumentów, 379
 dokumentów QTextDocument, 380
 w formacie HTML, 382
 w formacie ODF, 381
 w formacie SVG, 383
 elementy
 graficzne, 400, 407, 454, 466
 przechowywane na listach, 125
 współdzielone, 310
 emisja sygnału, 456
 etykieta, 303

F

filtr zdarzeń, 236
 filtrowanie, 102, 105–110, 116–119, 283
 w wątkach, 268
 z redukcją, 285
 format
 BMP, 372
 HTML, 372, 382
 ODF, 326, 379
 ODT, 325
 Ogg, 64
 PDF, 325, 380
 PNG, 372
 PostScript, 370, 380
 RTF, 325
 SVG, 371, 383, 395
 formatowanie akapitu, 327
 formaty rastrowe, 383
 framework Phonon, 70, 90
 frameworki
 maszyny stanów, 480
 obsługi animacji, 476
 funkcja, *Patrz* metoda

funkcje
 filtrowania, 278
 kryptograficzne, 311
 narzędziowe biblioteki Qt, 57
 wielobieżne, 285
 funktor, 282

G

grafika rastrowa, 390
 GUI, 102, 499

H

hierarchia
 klasy QGraphicsItem, 399
 modeli, 99
 HTML, 372

I

IDE, Integrated Development Environment, 497
 identyfikacja plików, 82
 identyfikator classId, 59
 indeks modelu, 173
 informacje o błędach, 275
 instrukcja debug(), 54
 inteligentne wskaźniki, 72, 111
 interfejs
 GUI, 102
 responsywny, 256
 użytkownika
 zmienianie modelu drzewa, 142, 161
 zmienianie modelu tabeli, 101
 interpolacja, 495

J

jakość obiektu QTextDocument, 369
 język
 JavaScript, 50
 QML, 498

K

kanał RSS, 31
 katalog domowy użytkownika, 197
 kąt, 464

klasa

- BoxItem, 464
- BrowserWindow, 36
- Cell, 412
- CensusVisualizer, 233, 244
- ConvertImageTask, 266
- CrossFader, 306
- customMatrixWidget, 56
- Error, 114
- GetMD5sThread, 321
- JingleAction, 65
- LinkFetcher, 49–52
- MainWindow, 101
- MatrixWidget, 59
- ModelTest, 124
- Phonon::Effect, 74
- QAbstractItemModel, 99, 105, 126, 164
- QAbstractItemView, 215, 219, 231
- QAbstractScrollArea, 231
- QAbstractTableModel, 125
- QAction, 65
- QAuthenticator, 18
- QBoxLayout, 405
- QCache, 22
- QCompleter, 329
- QConicalGradient, 251
- QCryptographicHash, 323
- QDataStream, 114, 455
- QDir, 197
- QDomDocument, 27, 32
- QFileInfo, 79
- QFutureWatcher, 277
- QGraphicsEffect, 489
- QGraphicsGridLayout, 405
- QGraphicsItem, 399, 407–413, 454
- QGraphicsObject, 13, 433
- QGraphicsPathItem, 398
- QGraphicsScene, 474
- QGraphicsSimpleTextItem, 398
- QGraphicsTextItem, 455
- QGraphicsView, 447, 474
- QGraphicsWidget, 407
- QHeaderView, 232
- QIODevice, 27
- QLabel, 306
- QList, 125
- QListView, 219
- QLocale, 237
- QMessageBox, 71, 148
- QMovie, 64
- QMutex, 276
- QNetworkAccessManager, 18–23, 31
- QNetworkCookieJar, 18
- QPainter, 397
- QPainterPath, 398
- QPlainTextEdit, 332
- QRadialGradient, 251
- QReadLocker, 276
- QReadWriteLock, 277, 313
- QRegExp, 347, 393
- QRunnable, 254, 257, 265
- QSortFilterProxyModel, 100, 120
- QSound, 64
- QStandardItem, 114, 140
- QStandardItemModel, 99, 100, 139, 145
- QString, 25
- QStringList, 313
- QStyledItemDelegate, 202, 208
- QStyleOptionViewItem, 209
- QSyntaxHighlighter, 345
- QTextBlock, 327
- QTextBlockFormat, 366
- QTextCursor, 342, 375
- QTextDocument, 325, 326, 360
- QTextDocumentFragment, 327
- QTextEdit, 207, 352, 361
- QTextFormat, 344, 379
- QTextFragment, 327
- QTextFrame, 328
- QTextLength, 378
- QTextOption, 201
- QTextTable, 328
- QThread, 255, 295–324
- QTimeL, 475
- QTimer, 415
- QTreeView, 141, 172
- QTreeWidgetItem, 75
- QUILoader, 59
- QWeakPointer, 298
- QWebElement, 54
- QWebPage, 33, 55
- QWebView, 36
- QWidget, 215, 398
- QXmlStreamWriter, 154
- RichTextLineEdit, 358, 360
- SmileyItem, 466
- StandardItemModel, 112
- StandardItemModel, 145, 148, 169

SurrogateItemApplier, 290
 TaskItem, 165
 TextEdit, 361, 365
 TextItemDialog, 434, 458
 Thread, 275
 ThreadSafeErrorInfo, 275
 ThreadSafeHash, 312–315, 323
 TiledListView, 217, 220, 251
 TreeModel, 168, 177
 WebPage, 56
 XmlEdit, 332
 XmlHighlighter, 350
 ZipcodeItem, 125
 ZipcodeSpinBox, 212
 klasy
 aplikacji Wizualizator, 233
 biblioteki Phonon, 73
 modułu QtWebKit, 33
 obsługi WebKit, 32
 QLabel, 306
 widoków graficznych, 399, 400
 z hierarchii modeli, 99
 klucze pamięci podręcznej, 44
 kolejka zdarzeń, 267
 kolekcja QActionGroup, 24
 kolor tekstu, 364
 kolorowanie, 200
 komentarze wielowierszowe, 350
 komponent
 okna przeglądarki, 33, 37
 TextEdit, 361
 komunikat o błędzie, 87, 114, 275, 293
 konfigurowanie
 akcji, 438
 wątków pomocniczych, 293
 widoku, 421
 konwersja pliku graficznego, 258
 kopiowanie
 elementów, 435
 podczas zapisu, 281
 korzeń drzewa, 174
 koszt obsługi wielowątkowości, 254
 krzywa dynamiki, easing curve, 478, 494

L

licencja LGPL, 497
 liczba
 komunikatów o błędach, 276
 wątków, 259

 wątków pomocniczych, 310
 licznik błędów, 275
 lista QList, 125, 152
 listy
 punktowane, 366
 uzupełniania, 339
 lokalne współrzędne logiczne, 400

Ł

ładowanie elementów tabel, 135
 łączenie animacji z maszynami stanów, 487

M

magiczne numery, 82
 makro
 CUSTOM_MODEL, 122
 foreach, 355
 forever, 319
 MODEL_TEST, 123
 manipulowanie piórem, 412
 mapa pikseli, 205, 451
 mapowanie, 289
 mapowanie w wątkach, 268
 maszyna stanów, 475, 480
 MD5, Message-Digest algorithm5, 311
 mechanizm
 blokujący, 277
 uzupełniania, 328
 menu
 kontekstowe, 351
 kontekstowe buźki, 469
 metadane pliku muzycznego, 81
 metoda
 addOneResult(), 319
 addTrack(), 80, 83
 addZipcode(), 134
 allSurrogateItems(), 286
 animateAlignment(), 477, 479
 appendRow(), 116
 applyColor(), 357
 applyTextEffect(), 354, 357
 AQP::accelerateMenu(), 24
 AQP::applicationPathOf(), 67
 AQP::chunkSizes(), 261
 AQP::filenameFilter(), 93
 AQP::okToClearData(), 147
 AQP::okToDelete(), 150

metoda

- AQP::suffixesForMimeTypes(), 80
- AQP::warning(), 69, 85
- assignProperty(), 484
- availableAudioEffects(), 74
- boundingRect(), 408
- brushSwatch(), 450
- calculateRectsIfNecessary(), 219, 222, 226
- calculateTotalsFor(), 158
- checkAnswer(), 61
- cleanUp(), 300, 302, 303
- Cleanup(), 298
- clear(), 112, 154
- closeEvent(), 421
- colorSwatch(), 364
- comparisonName(), 288
- convertFiles(), 266
- convertImages(), 267
- createActions(), 36, 66, 91
- createComboBoxModels(), 104
- createConnections(), 45, 76, 90, 104
- createIndex(), 183
- createLayout(), 362, 404, 452
- createMenuAction(), 469
- createNewTask(), 159
- createPlugin(), 56, 57
- createStates(), 483
- createToolBar(), 37
- createWidgets(), 76, 103, 362, 488
- CrossFader::stop(), 301
- data(), 129, 171
- dataChanged(), 183
- editCopy(), 437
- enableNetworkProxying(), 20
- endRemoveRows(), 185
- errors(), 276
- eventFilter(), 236, 245
- exec(), 111
- exists(), 123
- filename(), 126
- fileOpen(), 144
- filePrint(), 428
- fileSaveAs(), 148
- filterAcceptsRow(), 117
- FilterFunction(), 278
- finished(), 315
- finishedSelecting(), 282
- fontInterpolator(), 495
- generateOrCancelImages(), 299, 303
- getElementsByTagName(), 52
- handledCompletedAndSelected(), 342
- hideOrShowDoneTask(), 151
- highlightBlock(), 345
- highlightComments(), 347
- highlightCurrentLine(), 332, 345
- insertMulti(), 313
- insertRows(), 134
- interpretText(), 212
- invalidateFilter(), 119
- isFile(), 79
- isOK(), 270
- itemAccumulator(), 286
- itemChange(), 456, 462, 468
- itemForIndex(), 171
- keyPressEvent(), 343, 352
- LinkFetcher::addUrlAndName(), 52
- load(), 112, 142, 156
- main(), 21
- MainWindow::editAdd(), 159
- MainWindow::hideOrShowDoneTask(), 177
- MainWindow::createWidgets(), 329
- maleFemaleHeaderTextWidth(), 240
- map(), 270
- MapFunction(), 289
- MatrixWidget::readOnly(), 58
- mimeData(), 179
- minutesSecondsAsStringForMSec(), 83
- mousePressEvent(), 92
- moveCursor(), 226, 227
- moveDown(), 183
- moveItem(), 183
- newPage(), 387
- open(), 111
- operator()(), 282, 291
- operator*(), 440
- operator<(), 80
- PageData::paintPage(), 387
- paint(), 198, 203
- paintEvent(), 240, 248
- paintFooter(), 388
- paintHeaderItem(), 241
- paintItemBorder(), 242
- paintMaleFemale(), 250
- paintScene(), 428
- paintWidget(), 205
- pathForIndex(), 160, 191
- performCompletion(), 334
- performSelection(), 106

- play(), 64
- populateToolTip(), 27
- populateAComboBox(), 46
- populateArticleComboBox(), 46–48
- populateToolTip(), 30
- position(), 434, 438
- postEvent(), 263
- prepareToProcess(), 316
- processDirectories(), 315, 317
- QAbstractItemModel::beginInsertRows(), 176
- QAbstractItemModel::endRemoveRows(), 176
- QAbstractItemModel::reset(), 189
- QAbstractItemModel::data(), 330
- QAbstractItemView::scrollDirtyRegion(), 227
- QApplication::applicationDirPath(), 44
- QApplication::font(), 219
- QApplication::postEvent(), 263
- QApplication::topLevelWidgets(), 458
- QBoxLayout::addWidget(), 405
- QCache::object(), 28
- QColor::colorNames(), 450
- QComboBox::setModelColumn(), 104
- QComboBox::findData(), 452
- QCryptographicHash::hash(), 323
- QDesktopServices::storageLocation(), 78
- QDesktopServices::openUrl(), 305
- QDialog::open(), 111
- QDir::homePath(), 197
- QDir::toNativeSeparators(), 262, 320
- QDomDocument::setContent(), 27
- QDomNode::nodeValue(), 29
- QFile::readAll(), 51
- QFile::open(), 323
- QFileDialog::getOpenFileName(), 68
- QFileSystemModel::setRootPath(), 198
- QFontMetrics::elidedText(), 41
- qFuzzyCompare(), 283
- QGraphicsItem::sceneBoundingRect(), 441
- QGraphicsScene::render(), 429
- QGraphicsScene::removeItem(), 437
- qHash(), 23
- QImage::save(), 262, 305
- QImage::scaled(), 307
- QImage::pixel(), 308
- QImage::bits(), 309
- QImage::save(), 431
- qMakePair(), 323
- QMediaObject::totalTime(), 81
- QMenu::exec(), 41
- QMetaObject::invokeMethod(), 267, 431
- QMovie::supportedFormats(), 68
- QMovie::setFileName(), 68
- QMovie::currentPixmap(), 69
- QNetworkAccessManager::get(), 25
- QObjectFrom(), 432
- QObject::sender(), 25, 356
- QObject::property(), 442
- QPainter::drawImage(), 390
- QPixmap::loadFromData(), 30
- QPixmap::save(), 69
- QPixmap::grabWidget(), 205
- QPixmapCache::setCacheLimit(), 451
- QPixmapCache::find(), 451
- QPlainTextEdit::cursorRect(), 338
- QPrintPreviewDialog::open(), 386
- QPropertyAnimation::setKeyValueAt(), 478
- QRect::adjusted(), 199
- QScrollArea::ensureVisible(), 238
- qsort(), 125, 339
- QSortFilterProxyModel::lessThan(), 117
- qrand(), 56
- qStableSort(), 339
- QStandardItem::parent(), 159
- QStandardItem::data(), 195
- QState::assignProperty(), 484
- QString::fromUtf8(), 51
- QString::arg(), 288
- QString::compare(), 339
- QString::localeAwareCompare(), 339
- QStringList::mid(), 293
- QStringList::sort(), 450
- QSvgRenderer::render(), 390
- Qt::escape(), 374
- QtConcurrent::run(), 257, 265, 268
- QtConcurrent::filtered(), 278, 280
- QtConcurrent::mapped(), 290
- QtConcurrent::mappedReduced(), 285
- QTextCursor::movePosition(), 340
- QTextCursor::mergeBlockFormat(), 376
- QTextCursor::mergeCharFormat(), 376
- QTextCursor::insertText(), 377
- QTextCursor::insertTable(), 377
- QTextCursor::insertImage(), 379
- QTextDocument::drawContents(), 392
- QTextDocumentFragment::fromHtml(), 29
- QTextDocumentFragment::toPlainText(), 29
- QTextDocumentWriter::supportedFormats(), 381

metoda

QTextEdit::toHtml(), 208, 358
 QTableWidgetItem::firstCursorPosition(), 379
 QThread::idealThreadCount(), 261
 QThread::wait(), 301
 QTimer::singleShot(), 27
 QUiLoader::createWidget(), 59
 QUrl::fromLocalFile(), 305
 QWebFrame::evaluateJavaScript(), 51
 QWebSettings::testAttribute(), 35
 QWebView::findText(), 41
 QWidget::focusInEvent(), 35
 QWidget::event(), 264
 QWidget::window(), 458
 QDomStreamReader::attributes(), 157
 QDomStreamReader::hasError(), 157
 QDomStreamWriter::writeAttribute(), 155
 readItems(), 425, 428
 readTasks(), 190
 removeRow(), 162
 repopulateMatrix(), 61
 requestXml(), 23
 RichTextDelegate::sizeHint(), 358
 save(), 112
 setAngle(), 459, 468
 setBrush(), 468
 setColor(), 364
 setCompleter(), 330
 setCurrentBlockState(), 346
 setDirty(), 104, 109, 434, 440
 setEditorData(), 207
 setFace(), 469
 setFixedSize(), 449
 setFontItalic(), 352
 setFontWeight(), 352
 setModel(), 215
 setMusicDirectory(), 80
 setPen(), 468
 setShear(), 459, 468
 setTextColor(), 355
 shape(), 412
 show(), 34
 SimpleHtml(), 358
 StandardTableModel::load(), 136
 StandardTreeModel::itemForPath(), 162
 std::ceil(), 444
 stopThreads(), 315, 319
 takeChild(), 167
 QTableWidgetItem::swapChildren(), 183

QTableWidgetItem::insertChild(), 187
 QTableWidgetItem::addChild(), 189
 textForTag(), 28
 textFromValue(), 212
 toHtml(), 358
 tooltipField(), 30
 toPlainText(), 48
 toSimpleHtml(), 359
 tr(), 108, 155, 284
 TreeModel::indexPath(), 162
 update(), 221
 updateColor(), 453
 updateColorSwatch(), 364
 updateContextMenuActions(), 357
 updateGeometries(), 219, 230
 updateSwatches(), 453
 updateTransform(), 461
 updateUi(), 68, 436
 viewport(), 219
 viewportRectForRow(), 222
 visualRect(), 223
 windowFilePath(), 420, 429
 writeItems(), 424
 writeTaskAndChildren(), 155, 190

metody

klasy QFileInfo, 79
 obsługi ładowania, 189
 obsługi ścieżek zadań, 189
 obsługi tabel, 126
 obsługi zapisywania, 189
 prywatne, 169
 przemieszczania elementów, 182
 wirtualne, 229, 408

metody-adaptery, 362

migawka, 69

model

DOM, 18
 proxy, 109, 119
 QAbstractItemModel, 169, 170, 177
 QDirModel, 329
 QSortFilterProxyModel, 116, 118
 QStandardItemModel, 99, 121, 160, 166
 StandardTreeModel, 150, 179
 TreeModel, 172
 UniqueProxyModel, 104

modele

drzew, 127, 139
 drzew niestandardowe, 160
 edytowalne, 127

list, 97
 niestandardowe, 121
 o zmiennych rozmiarach, 127
 tabel, 97, 100, 122
 tabel i drzew, 127
 wyborów, 105
 moduł
 aqp, 80
 Phonon, 74, 94
 QtMultimedia, 95, 454
 QtNetwork, 19
 QtScript, 18
 QtWebKit, 18, 33
 QtXml, 27
 QtXmlPatterns, 62
 modyfikowanie współrzędnej z, 459
 muteks, 275

N

narzędzia translatorskie, 284
 narzędzie Web Inspector, 34
 nawiasy wychwytyjące, 347
 nazwy kolorów, 450
 niestandardowe modele drzew, 160

O

obiekt
 BrowserWindow, 39
 CensusVisualizer, 236, 240, 246
 CensusVisualizerView, 239, 243
 completer, 334
 CrossFader, 299, 304
 errorInfo, 291
 GetMD5sThread, 318
 LinkChecker, 44
 LinkFetcher, 49, 51
 ModelTest, 123
 QAbstractButton, 111
 QAbstractTableModel, 192
 QAction, 24
 QActionGroup, 25
 QApplication, 21
 QByteArray, 179, 323
 QCache, 23
 QDataStream, 135
 QDirIterator, 322
 QFileSystemModel, 198

QFuture<T>, 268, 274
 QGraphicsItemGroup, 420
 QGraphicsView, 397
 QHBoxLayout, 316
 QItemEditorFactory, 194
 QLineEdit, 211
 QList, 269
 QModelIndex, 172, 173
 QMutexLocker, 276
 QNetworkAccessManager, 18, 30
 QNetworkReply, 27
 QPainter, 370, 387
 QPair, 323
 QPixmap, 31
 QPrinter, 381
 QReadLocker, 313
 QScrollArea, 239
 QSettings, 24
 QStandardItem, 109
 QStandardItemModel, 192
 QStringList, 369
 QStyleOptionComboBox, 131
 QTableWidgetItem, 60
 QTextCursor, 367, 376
 QTextDocument, 367, 369, 391
 QTextDocumentFragment, 48
 QTextStream, 51
 QTransform, 461
 QTreeWidgetItem, 74
 QUrl, 30
 QVariant, 129, 131
 QWebFrame, 50
 QWebInspector, 34
 QWebPage, 33
 QWebView, 36, 58
 QWidget, 219, 232
 QWriteLocker, 314
 QXmlStreamReader, 190
 resultLabel, 62
 TextItemDialog, 458
 TreeModel, 173
 TreeWidgetItem, 75
 obliczanie sygnatury MD5, 311
 obraz SVG, 370, 371
 obserwator futury licznika, 287
 obsługa
 animacji, 475
 blokad, 324
 JavaScript, 50

- obiekt
 - klawiatury, 465
 - klawiszy, 227
 - list punktowanych, 366
 - multimediów, 70, 95
 - niestandardowych widżetów, 55
 - plików cookies, 18
 - powiększania, 421
 - serwerów proxy, 20
 - tekstu sformatowanego, 325
 - wątków, 253–256, 293, 295
 - zdarzeń, 332, 464
 - ODF, Open Document Format, 326, 370, 379
 - ODT, Open Document Text, 325
 - odtworzenie
 - muzyki, 74
 - wideo, 89
 - okno
 - główne aplikacji, 420
 - TextItemDialog, 434
 - Znajdź, 488
 - operacje
 - na elementach graficznych, 431, 441
 - na próbkach, 450
 - na zaznaczonych elementach, 438
 - operator &, 205
 - organizacja IANA, 82
 - osadzanie widżetów Qt, 54
- P**
- pakiet QtWebKit, 18
 - pamięć podręczna, 471
 - parser kodu JavaScript, 291
 - pasek
 - postępu, 303
 - przewijania, 231
 - stanu, 304
 - PDF, Portable Document Format, 325
 - pędzel, 451
 - pętla zdarzeń, 93
 - plik
 - jingleaction.cpp, 65
 - mainwindow.cpp, 481
 - mime.types, 82
 - pliki
 - .odt, 370
 - .pro, 19, 70, 370
 - .ps, 380
 - dźingli, 67
 - PDF, 369
 - SVG, 395
 - PNG, Portable Network Graphics, 372
 - pochylenie, 464
 - podgląd wydruku, 386
 - podklasa
 - CrossFader, 304
 - QAbstractTableModel, 116, 125
 - QGraphicsPathItem, 408
 - QGraphicsPolygonItem, 408
 - QSortFilterProxyModel, 112
 - QStandardItem, 151, 195
 - QStandardItemModel, 112, 122, 152
 - StandardTableModel, 116
 - podświetlenie
 - komentarzy, 348
 - składni, 325, 330, 344
 - połączenia przełącznika, 105
 - połączenie
 - aboutToFinish(), 77
 - stateChanged(), 77
 - sygnał-sl, 47
 - sygnał-slot, 84, 277, 334, 362
 - tick(), 77
 - porównywanie kodów pocztowych, 126
 - powiązanie sygnał-slot, 50
 - powiększanie, 421
 - prefiks uzupełnienia, 334
 - procedury obsługi zdarzeń, 464
 - program
 - evince, 368
 - OpenOffice.org, 370
 - QScintilla, 331
 - promowanie elementu, 187
 - próbka penJoinSwatch(), 450
 - próbki kolorów, 354
 - prywatne składowe widżetów, 362
 - przeciąganie, 169, 177, 447
 - przeciążanie metod, 117
 - przełączarka
 - evince, 370
 - gv, 370
 - recenzji książek, 42
 - przejścia pomiędzy stanami, 482
 - przejście typu duszek, 496
 - przełącznik, 105
 - przełącznik Filter, 108
 - przenoszenie elementu, 184
 - przepływ sterowania, 28, 45

przeźren nazw
 AQP, 12
 QtConcurrent, 254, 270, 285, 289
 przesunięcia współrzędnych, 239
 przesuwane elementów, 447, 465
 przetwarzanie
 danych, 296
 niezależnych elementów, 296
 w wątkach, 257
 współdzielonych elementów, 310
 przewijanie, 448
 przybornik, 448

Q

QML, Qt Meta-Object Language, 498

R

RAII, Resource Acquisition Is Initialization, 72
 referencja zmienna, 205
 relacja rodzic-dziecko, 399
 relacje między elementami, 174
 renderowanie, 398
 elementów, 202
 poza ekranem, 471
 sceny, 429
 treści dokumentu, 391
 RGB, 308
 rodzaje elementów graficznych, 418
 rodzic elementu, 175
 rola
 Qt::SizeHintRole, 130
 Qt::DisplayRole, 130, 171
 Qt::FontRole, 130
 Qt::EditRole, 171
 Qt::CheckStateRole, 171
 Qt::DecorationRole, 171
 rozmiar
 obrazka, 451
 okna, 406
 sceny, 421
 widżetu, 245
 RSS, Really Simple Syndication, 31
 RTF, Rich Text Format, 325
 RTTI, Run Time Type Information, 240, 356,
 432, 462
 rysowanie, 210, 241, 249, 408
 akapitu HTML, 392

buźki, 470
 dokumentów PDF, 394
 dokumentów PostScript, 394
 dokumentów SVG, 395
 dokumentów z grafiką rastrową, 395
 elipsy, 200
 kapelusza, 473
 obrazu SVG, 390
 prostokąta, 199, 230
 słowa, 393
 stron, 387
 tekstu, 388
 tła, 249
 widżetów, 205
 rzutowanie, 240
 rzutowanie wskaźników, 456

S

scena, 406, 419
 drukowanie, 428
 eksportowanie, 428
 ładowanie, 425
 renderowanie, 429
 zapisywanie, 423
 serializacja, 177
 serwer proxy, 20
 sformatowane dokumenty tekstowe, 367
 siatka, 419, 443
 silnik WebKit, 32
 silniki renderowania, 398
 skalowanie, 448, 462
 składowe widżetów, 362
 skrót kryptograficzny, 323
 skróty klawiaturowe
 Ctrl+B, 353
 Ctrl+M, 330
 skrypt
 fetch_article_links.js, 52
 fetch_issue_links.js, 53
 slot
 aboutToFinish(), 86, 88
 addUrlAndName(), 51
 addZipcode(), 102
 checkIfDone(), 261
 chooseVideo(), 92
 closeAndCommitEditor(), 202
 convertFiles(), 264
 convertOrCancel(), 258

slot

currentIssueIndexChanged(), 47
 currentItemChanged(), 84
 currentSourceChanged(), 85
 editAlign(), 438, 477
 editHideOrShowDoneTasks(), 151
 editMoveUp(), 165
 editSelect(), 279
 fetchLinks(), 51
 fileExport(), 430
 fileNew(), 421
 fileOpen(), 425
 fileSave(), 423
 findOrCancel(), 316
 load(), 144
 playOrPause(), 84, 94
 playTrack(), 85
 populateCache(), 45
 populateIssueComboBox(), 46
 readIcon(), 29, 30
 requestXml(), 23
 setAirport(), 25
 setAngle(), 461
 setBrush(), 452
 setDirty(), 437
 setMusicDirectory(), 77
 setShear(), 461
 setValue(), 39
 start(), 481
 startOrStop(), 68
 stateChanged(), 86, 93
 stop(), 481
 tick(), 86
 updateUi(), 102, 146, 439, 490
 viewShowGrid(), 421

słowo kluczowe volatile, 256

sortowanie, 105, 117, 121, 126, 320, 333

stan aplikacji, 481

- hideExtraWidgetsState, 492
- pauza, 485
- początkowy, 484
- showExtraWidgetsState, 491
- uruchomiona, 484
- zatrzymana, 485

standard ISO 8601, 153

sterowanie układem, 405

STL, Standard Template Library, 440

struktura MatchCriteria, 274

strumień QDataStream, 113

sumowanie liczb zmiennoprzecinkowych, 288

SVG, Scalable Vector Graphics, 32, 371

sygnał

- brushChanged(), 427, 449
- clicked(), 247, 486
- colorSelected(), 364
- currentSourceChanged(), 77, 88
- customContextMenuRequested(), 352
- dataChanged(), 133, 175
- DataChanged(), 124
- dirty(), 459
- editAddAction(), 146
- findTextIsNonEmpty(), 490
- finished(), 91, 277
- headerDataChanged(), 134
- itemChanged(), 109
- LinkFetcher::finished(), 46
- loadProgress(), 39
- paintRequested(), 386
- progress(), 308
- QGraphicsScene::selectionChanged(), 442
- QStandardItemModel::rowsInserted(), 109
- readOneFile(), 318
- readyRead(), 27
- returnPressed(), 352, 357
- sectionClicked(), 120
- setDirty(), 109
- showExtra(), 491
- stateChanged(), 64
- stopTiming(), 180
- tick(), 76
- triggered(), 65, 421
- valueChanged(), 39

sygnatura MD5, 310

symbol DEBUG, 34

Ś

śledzenie postępów przetwarzania, 268
 środowisko IDE, 497

T

tablica asocjacyjna, 221, 225, 313, 348
 technika

- kopiuj i wklej, 189
- przeciągnij i upuść, 181

 technologia Qt Quick, 13
 tekst sformatowany, 325

test `Q_ASSERT`, 188
 timery, 143
 transformacje elementów graficznych, 460
 tryb domyślny przeciągania, 447
 tworzenie

- animacji, 398
- dokumentów `QTextDocument`, 372, 375
- edytorów tekstu, 325, 328
- elementów graficznych, 408, 454, 466
- modeli drzew, 139, 160
- obiektu `QGraphicsScene`, 403
- obiektu uzupełniającego, 330
- okien, 397, 420
- przyborników, 448
- scen, 417
- sformatowanych dokumentów, 367
- widżetów potomnych, 332
- własnych modeli tabel, 122

 typ

- MIME, 80, 82, 178, 436
- T, 278, 289
- wyliczeniowy, 66
 - `QGraphicsItem::GraphicsItemFlag`, 414
 - `Qt::ItemDataRole`, 128
 - `Qt::ItemSelectionMode`, 414
 - `QTextCursor::MoveOperation`, 341

 typy modeli, 99

U

układ, 401, 405
 ulepszenie

- elementów graficznych, 463
- klasy `QGraphicsTextItem`, 455
- widoku `QGraphicsView`, 447

 unikatowe identyfikatory typów, UTI, 82
 upuszczanie, 169, 177
 uruchamianie funkcji w wątkach, 256
 urządzenia multimedialne, 73
 ustawienia czcionki, 389, 391
 usuwanie

- elementów, 437
- etykiet, 302
- widżetów, 303

 UTI, Uniform Type Identifiers, 82
 utrata dokładności liczb, 288
 uzupełnianie tekstu, 325, 328

- dla edytorów wielowierszowych, 332
- w polach kombi, 329

w polach tekstowych, 329
 w trybie `popup`, 329
 w trybie `inline`, 329

V

VoIP, Voice over Internet Protocol, 76

W

wątki, 253
 wątki pomocnicze, 293, 310
 WebKit, 32
 węzły multimediiów, 73
 widok, 215

- `CensusVisualizer`, 232
- `QColumnView`, 215
- `QGraphicsView`, 447
- `QListView`, 215, 216
- `QTableView`, 215, 233
- `QTreeView`, 215
- `TiledListView`, 216, 224, 251

 widoki

- standardowe, 98, 215
- wizualizacji, 232

 widżet

- `CensusVisualizerHeader`, 239
- komfortowy, 121
- `MatrixWidget`, 61
- nagłówka, 239
- paska postępu, 302
- `Phonon::VideoPlayer`, 94
- `progressWidget`, 298
- proxy, 398
- `QWebView`, 62
- `RichTextLineEdit`, 207, 351
- `TransformWidget`, 461
- widoku, 243
- wizualizatora, 233
- `XmlEdit`, 330
- zagregowany, 243

 widżety

- internetowe, 18
- komfortowe, 98
- niestandardowe, 240
- przybornika, 445
- Qt, 54
- `QWidget`, 403
- wielowątkowość, 253

- wizualizator, 232
- wklejanie elementu, 186
- własne modele tabel, 122
- włączanie widżetów, 444
- wskaźnik
 - na muteks, 276
 - QPointer, 72, 298
 - QPrinter, 419
 - QSharedPointer, 111
- współbieżne opcje redukcji, 285
- współrzędne, 239
 - elementu, 400
 - logiczne, 400
 - sceny, 400
 - wziernika, 222
- wstawianie list, 375
- wstrzykiwanie kodu JavaScript, 52
- wybieranie, 110
- wybór kolorów, 355
- wydajność metod, 281
- wydłużenia dolne, descenders, 391
- wyjątek, 158, 190
- wyrównywanie, 439
- wyświetlanie
 - linii siatki, 443
 - okien informacyjnych, 71
 - okna, 492

Z

- zakładanie blokad, 315
- zamykanie
 - okna, 490
 - okna głównego, 191
- zapisywanie
 - elementów tabel, 135
 - scen, 423
- zaplecze, backend, 70
- zatrzymywanie wątku, 280
- zdarzenia niestandardowe, 264

- zdarzenie
 - keyPressEvent(), 342
 - rysowania, 248
 - wheelEvent(), 448
- zegar, 200
- zmienianie modelu
 - drzewa, 142, 161
 - tabeli, 101, 122
- zmienna volatile, 256
- znacznik
 - , 361
 - <h1>, 374
 - <head>, 373
 - <html>, 373
 - , 374
 - <p>, 149
 - a, 52
 - TASK, 153
 - WHEN, 153
- znaczniki XML, 331
- znaki
 - nowego wiersza, 373
 - specjalne, 360

Ż

- źródła danych multimedialnych, 73

Ż

- żądania HTTP, 18
- żądanie
 - GET, 53
 - zatrzymania wątku, 280

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

Wykorzystaj zaawansowane funkcje Qt!

Qt to popularny zestaw bibliotek pozwalający na tworzenie atrakcyjnego interfejsu użytkownika dla różnych platform, w tym dla systemów Windows, Mac OS X i Linux. Pierwsza wersja ukazała się w 1992 roku i od tego czasu jest intensywnie rozwijana. Za pomocą najnowszej wersji Qt można tworzyć aplikacje internetowe i mobilne. Zakres jej możliwości jest tak szeroki, że nawet doświadczeni programiści wykorzystują zaledwie ich ułamek. To może się zmienić dzięki tej książce!

W trakcie lektury odkryjesz funkcje i możliwości, z których istnienia nie zdawałeś sobie sprawy. Ponadto nauczysz się pisać wydajne programy wielowątkowe, korzystać z silnika WebKit oraz współpracować z biblioteką Phonon. Dowiesz się również, jak sprawnie przy użyciu Qt tworzyć sformatowane dokumenty, a następnie eksportować je do różnych formatów (między innymi PDF, HTML i SVG). Znajdziesz tu przykłady kodu, przetestowanego przy użyciu Qt 4.6 na platformach Windows, MacOS X i Linux. Wstęp do książki napisał sam współtwórca Qt - Eirik Chambe-Eng.

Sięgnij po doskonałe źródło informacji dla programistów!

helion.pl
księgarnia
internetowa

Nr katalogowy: 17376



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900

**PRENTICE
HALL**



Helion

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/nowosci>

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

Informatyka w najlepszym wydaniu



Dzięki tej książce:

- poznasz zaawansowane zastosowania biblioteki Qt
- wykorzystasz możliwości silnika WebKit
- przygotujesz atrakcyjny wizualnie dokument i wyeksportujesz go do popularnych formatów
- zgłębisz tajniki biblioteki Qt
- wykorzystasz przykładowy kod, gotowy do użycia w najpopularniejszych systemach operacyjnych

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-246-8233-1



9 788324 682331

Cena: 79,00 zł