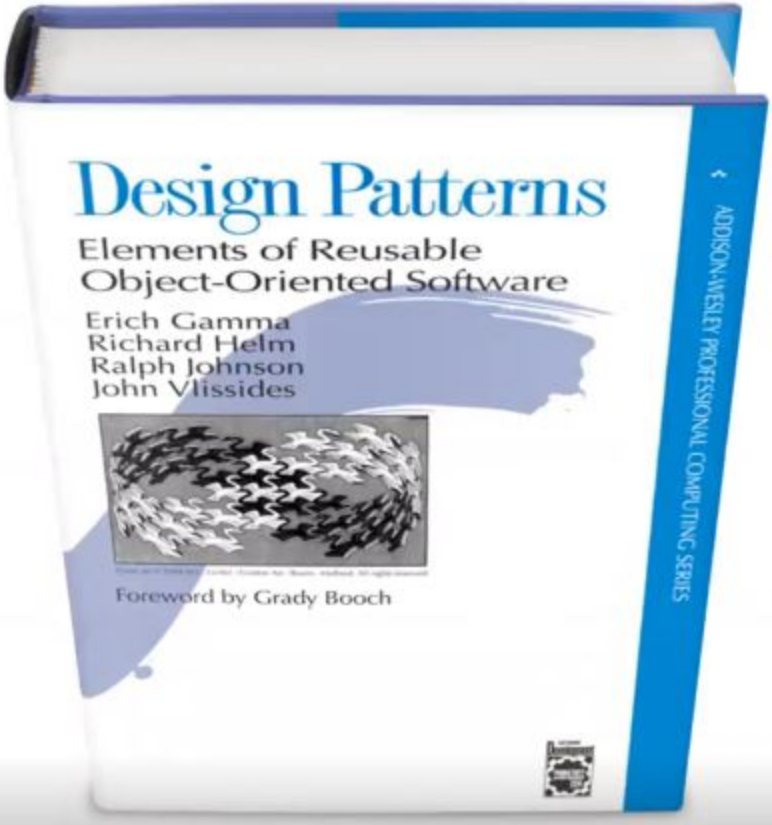
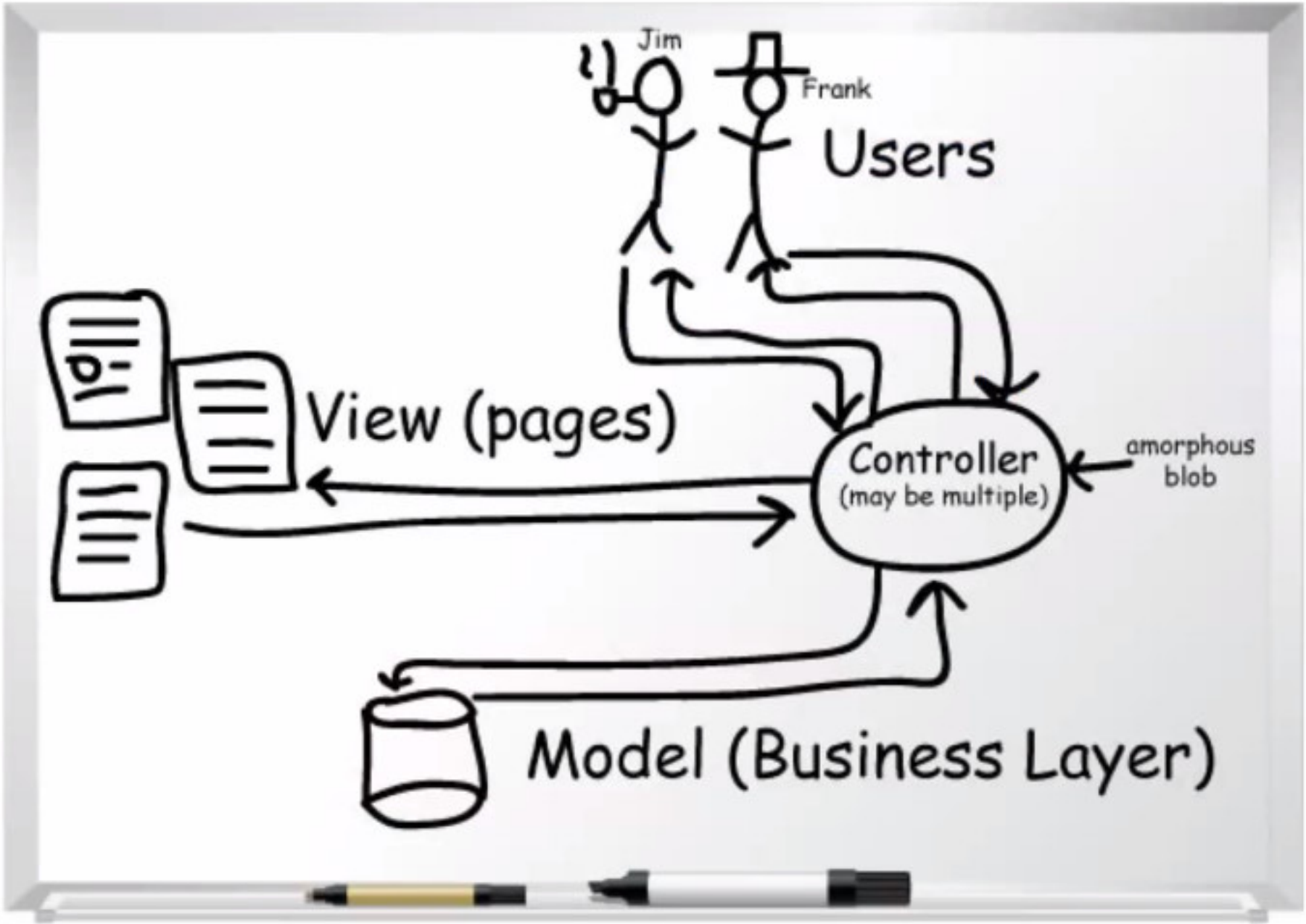
GANG OF FOUR


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


Jim
Frank
Users
View (pages)
Controller
(may be multiple)
amorphous
blob
Model (Business Layer)

Model–view–controller

From Wikipedia, the free encyclopedia

Model–view–controller (**MVC**) is a software pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.[1][2] The central component, the *model*, consists of application data, business rules, logic, and functions. A *view* can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The third part, the *controller*, accepts input and converts it to commands for the model or view.[3]

Contents [hide]

Model-View-Controller (MVC) [edit]

A pattern often used by applications that need the ability to maintain multiple views of the same data. The model-view-controller pattern was until recently a very common pattern especially for graphic user interface programming, it splits the code in 3 pieces. The model, the view, and the controller.

The Model is the actual data representation (for example, Array vs Linked List) or other objects representing a database. The View is an interface to reading the model or a fat client GUI. The Controller provides the interface of changing or modifying the data, and then selecting the "Next Best View" (NBV).

Newcomers will probably see this "MVC" model as wasteful, mainly because you are working with many extra objects at runtime, when it seems like one giant object will do. But the secret to the MVC pattern is not writing the code, but in maintaining it, and allowing people to modify the code without changing much else. Also, keep in mind, that different developers have different strengths and weaknesses, so team building around MVC is easier. Imagine a View Team that is responsible for great views, a Model Team that knows a lot about data, and a Controller Team that see the big picture of application flow, handing requests, working with the model, and selecting the most appropriate next view for that client.

Model/View Programming

The model/view architecture

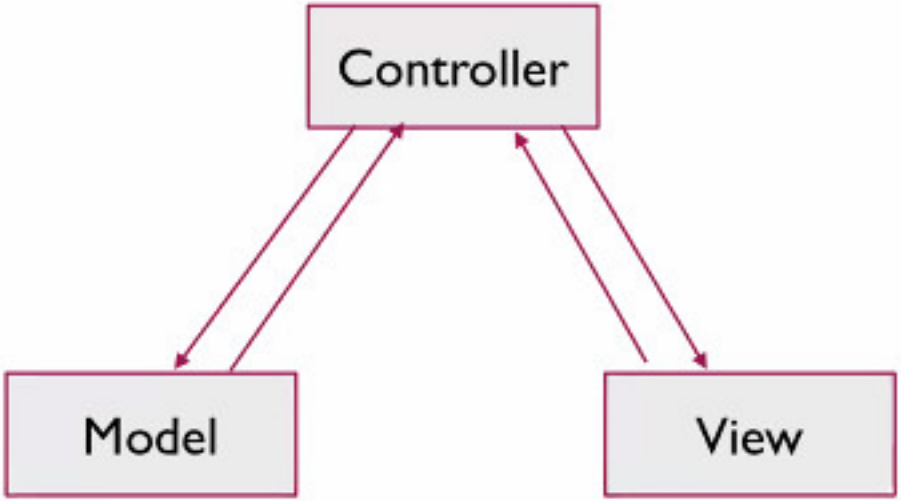
Model-View-Controller (MVC) is a design pattern originating from Smalltalk that is often used when building user interfaces. In Design Patterns, Gamma et al. write:

MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.

If the view and the controller objects are combined, the result is the model/view architecture. This still separates the way that data is stored from the way that it is presented to the user, but provides a simpler framework based on the same principles. This separation makes it possible to display the same data in several different views, and to implement new types of views, without changing the underlying data structures. To allow flexible handling of user input, we introduce the concept of the *delegate*. The advantage of having a delegate in this framework is that it allows the way items of data are rendered and edited to be customized.

TextEdit

Mini esempio di MVC



Model

```
class GraphModel {
private:
    int number; // dato logico

public:
    GraphModel(): number(1) {} // costruttore

    void increaseNumber() { number += 10; }  // scrittura/lettura

    int getNumber() const { return number; } // lettura
};
```

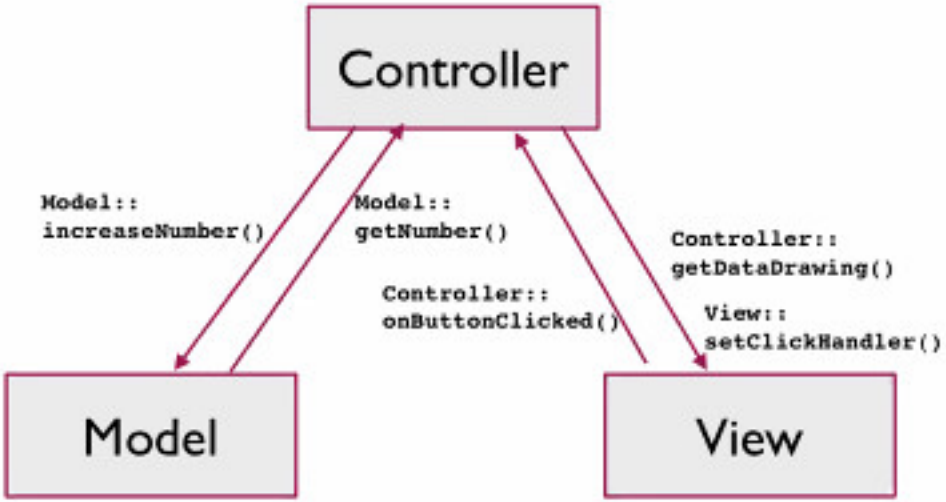
View

```
class GraphView {
private:
    Button*              button;      // view components
    GraphController*     controller;  // control

public:
    // costruisce view e control
    GraphView():
         button(new Button("Click Me")),
         controller(new GraphController(this)) {}

    ~GraphView() {
        delete button;
        delete controller;
    }

    // definisce il gestore del button click
    void setClickHandler(ButtonHandler* bh){
        button->setHandler(bh);
    }

    void drawGraph() {
        // ottiene dati logici dal model via control
        int dati = controller->getDataDrawing();
        // Disegna il grafo sui dati
        // ...
    }
};
```

Controller

```
class GraphController {
private:
    GraphModel*    model;  // model
    GraphView*     view;   // view

public:
   GraphController(GraphView* v): model(new GraphModel()), view(v) {
        // installazione del gestore sulla view
        view->setClickHandler(&onButtonClicked);
    }

    // trasmette l'input dalla view al model e modifica il model
    void onButtonClicked() {
        model->increaseNumber();
    }

    // ottiene dati dal model
    int getDataDrawing() const {
        return model->getNumber();
    }
};
```

Mini esempio di MVC



creativi(TÀ)

Open *World* Assumption

Precisione

Framework Qt

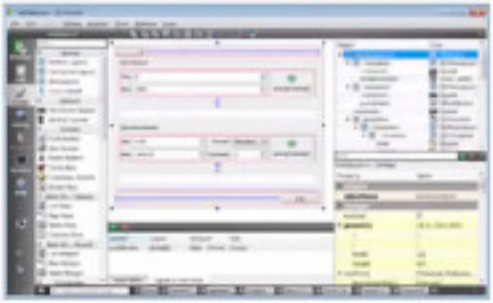
Francesco Ranzato

Qt (software)

From Wikipedia, the free encyclopedia

For the company formerly known as Qt Software, see The Qt Company.

Qt (pronounced "cute"[7][8][9]) is a cross-platform application framework and widget toolkit for creating classic and embedded graphical user interfaces, and applications that run on various software and hardware platforms with little or no change in the underlying codebase, while still being a native application with native capabilities and speed. Qt is currently being developed both by The Qt Company, a publicly listed company, and the Qt Project under open-source governance, involving individual developers and firms working to advance Qt.[10][11][12] Qt is available under both commercial licenses[4] and open source[13] GPL 2.0, GPL 3.0, and LGPL 3.0 licenses.[5][6]



GUI designing in Qt Creator using the embedded Qt Designer

Original author(s)	Haavard Nord and Eirik Chambe-Eng[1]
Developer(s)	Trolltech (1991–2008) Nokia (2008–2011) Qt Project (2011–present) Digia (2012–2014) The Qt Company (2014–present)
Initial release	20 May 1995; 22 years ago[1]
Stable release	5.9.3 (22 November 2017; 13 days ago) [+][2]
Repository	code.qt.io/qt/qt.git
Development status	Active
Written in	C++
Operating system	Android, iOS, Linux (Embedded, Wayland, X11), macOS, Windows, Windows Phone, ...[3]
Platform	Cross-platform
Type	Application framework
License	Qt Commercial License[4] GPL 2.0, 3.0[5] LGPL 3.0[6]
Website	www.qt.io

The Qt Company

From Wikipedia, the free encyclopedia

Not to be confused with the Qt Project.

The Qt Company (pronounced "cute"; formerly **Trolltech**) is a software company based in Espoo, Finland. It oversees the development of its Qt application framework within the Qt Project. It was formed following the acquisition of Qt by Digia, but was later spun off into a separate, publicly traded company.

It has core R&D in Oslo, Norway, as well as large engineering teams in Berlin, Germany and Oulu, Finland. The Qt Company operates in China, Finland, Germany, Norway, Russia, South Korea, Japan, India, and the United States.[2]

Purposes and abilities [edit]

Qt is used mainly for developing application software with graphical user interfaces (GUIs); however, programs without a GUI can be developed, such as command-line tools and consoles for servers. An example of a non-GUI program using Qt is the Cutelyst web framework.[16] GUI programs created with Qt can have a native-looking interface, in which cases Qt is classified as a *widget toolkit*.

Qt uses standard C++ with extensions including signals and slots that simplifies handling of events, and this helps in development of both GUI and server applications which receive their own set of event information and should process them accordingly. Qt supports many compilers, including the GCC C++ compiler and the Visual Studio suite. Qt also provides Qt Quick, that includes a declarative scripting language called QML that allows using JavaScript to provide the logic. With Qt Quick, rapid application development for mobile devices became possible, although logic can be written with native code as well to achieve the best possible performance. Qt can be used in several other programming languages via language bindings. It runs on the major desktop platforms and some of the mobile platforms. It has extensive internationalization support. Non-GUI features include SQL database access, XML parsing, JSON parsing, thread management and network support.

Companies using Qt

Uses [edit]

Organizations using Qt [edit]

Because of simplicity, robustness, native performance, cross-platform compatibility and both commercial and open source licenses, many organizations in many parts of the world use Qt. These include but are not limited to European Space Agency,[50] DreamWorks,[51][52] Lucasfilm,[53][54] Panasonic,[55] Philips,[56] Samsung,[57] Siemens,[58] Volvo,[59] Walt Disney Animation Studios,[60] Blizzard Entertainment[61]

Software using Qt [edit]

Main category: Software that uses Qt

Example applications using Qt are Altera Quartus, a design and simulation tool for programmable logic; Autodesk Maya;[62][63] Bitcoin Core; Cameleon (programming language); Mathematica;[64] Google Earth;[65] KDE,[66] a desktop environment for UNIX-like operating systems; Skype;[67] Spotify for Linux;[68] Ubuntu;[69] VirtualBox, an OS virtualization software package; Musescore music score writing software; and the VLC media player.[70]

Official platforms [edit]

The following platforms are officially supported by Qt:

Platform	Details
Android	Qt for Android (currently for Android 5 Lollipop and later, i.e. all currently supported and popular unsupported versions).[1] formerly known as Necessitas[2]
Embedded Linux	Qt for embedded platforms: personal digital assistant, smartphone, etc.[3]
Integrity	Qt for Integrity[4][5]
iOS	Qt for iOS platforms (iPhone, iPad); currently only for 64-bit, i.e. iOS 11 and later.[6] Support for tvOS 11 and later and watchOS 4 and later as a technology preview.
macOS	Qt for Apple macOS (64-bit platforms); supports applications on Cocoa[7]
QNX	Qt for QNX[8] Under free software license and also under "Qt Commercial" license
VxWorks	Qt for VxWorks;[9] only available under a commercial license. Qt 5.5 is currently tested and supported on VxWorks 7 release SR0480 2016-09-16.[10]
Wayland	Qt for Wayland.[11] Qt applications can switch between graphical backends like X and Wayland at load time with the -platform command line option.[12][13] This allows a seamless transition of Qt applications from X11 to Wayland.
Windows	Qt 5.13[14] for Microsoft Windows 7,[15] 8.1 and 10; Qt 5.6 version supported: Windows XP and Vista
Windows CE	Older Qt versions had support for Windows CE 6 and Windows Embedded Compact 7.[16]
Windows RT	Universal Windows Platform 10. Previous Qt versions: Support for WinRT-based Windows 8 apps and Windows Phone 8[17]
X11	Qt for X Window System (GNU/Linux); FreeBSD, NetBSD, OpenBSD, and DragonFly BSD (and other operating systems) have community support. has community support for Qt 4.6 (now no longer supported version).[18]

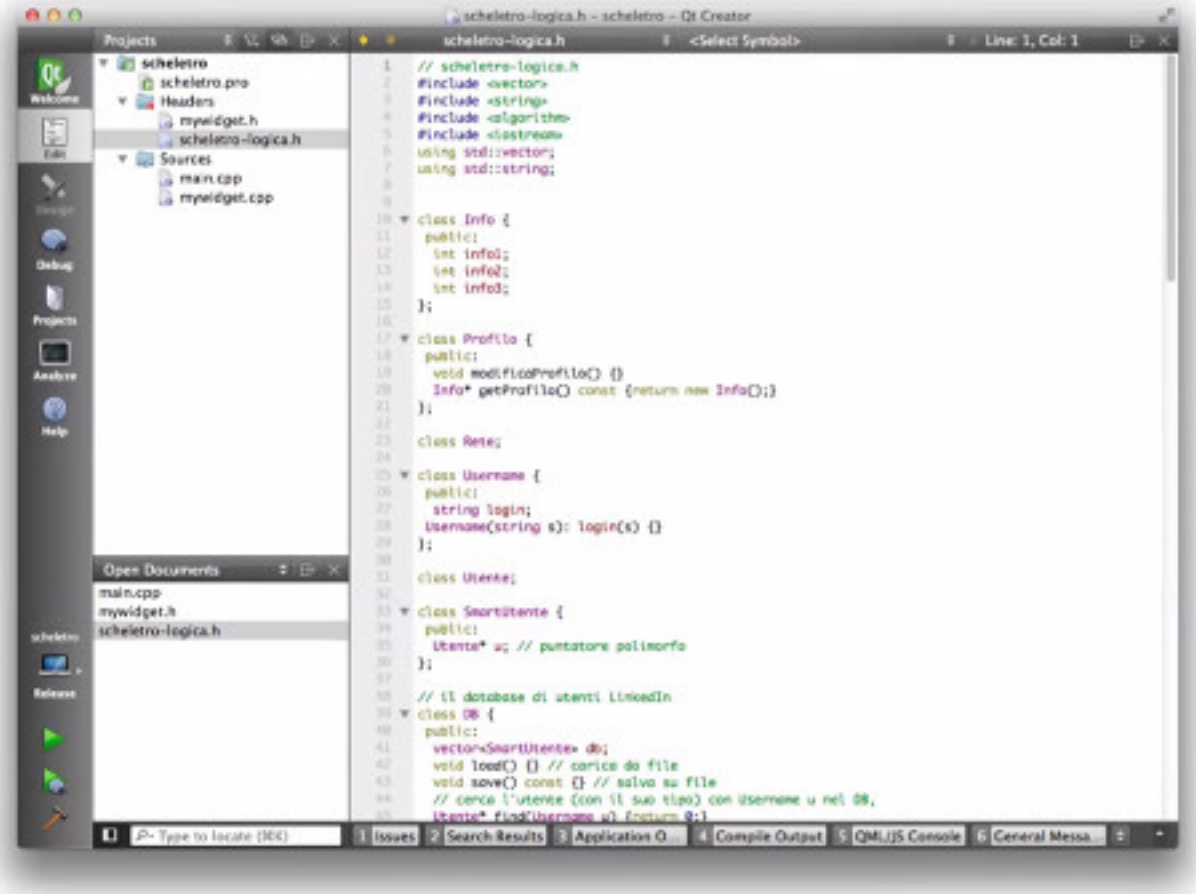
Qt Creator

From Wikipedia, the free encyclopedia

This article **relies on references to primary sources**. Please add referen

(June 2009)

Qt Creator is a cross-platform C++ integrated development environment which is part of the SDK for the Qt GUI Application development framework.[4] It includes a visual debugger and an integrated GUI layout and forms designer. The editor's features include syntax highlighting and autocompletion, but not tabs. Qt Creator uses the C++ compiler from the GNU Compiler Collection on Linux and FreeBSD. On Windows it can use MinGW or MSVC with the default install and can also use cdb when compiled from source code. Clang is also supported.



Qt Designer Manual

Contents

Qt Designer is Qt's tool for designing and building graphical user interfaces (GUIs) from Qt components. You can compose and customize your widgets or dialogs in a what-you-see-is-what-you-get (WYSIWYG) manner, and test them using different styles and resolutions.

Widgets and forms created with *Qt Designer* integrated seamlessly with programmed code, using Qt's signals and slots mechanism, that lets you easily assign behavior to graphical elements. All properties set in *Qt Designer* can be changed dynamically within the code. Furthermore, features like widget promotion and custom plugins allow you to use your own components with *Qt Designer*.

stackoverflow About Products For Teams

Home

PUBLIC

Qt Designer vs Handcoding

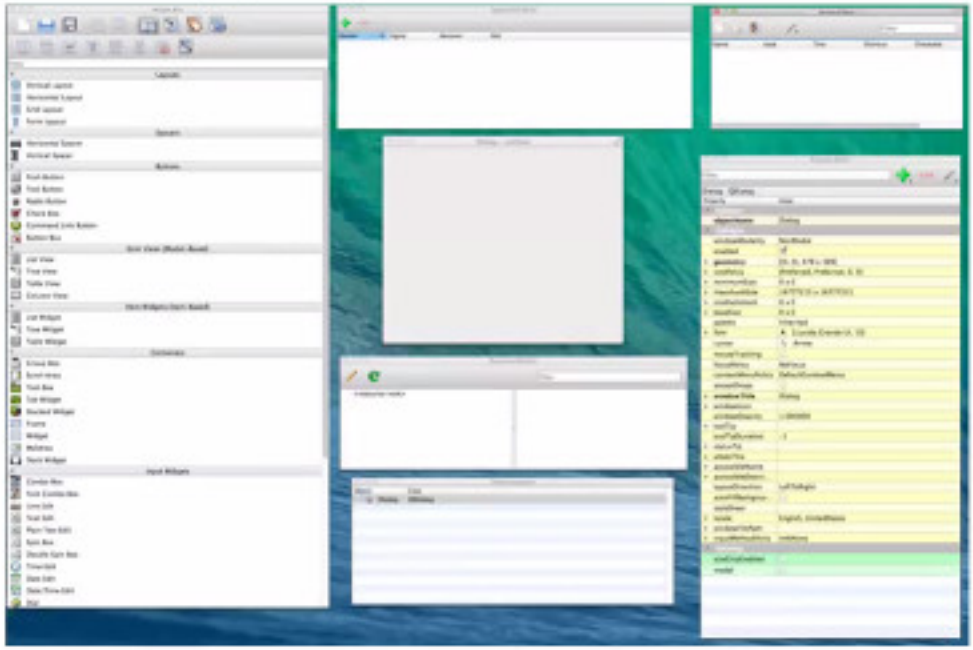
Asked 10 years, 1 month ago Active 3 months ago Viewed 15k times

Qt Designer

- Good
  - Exploration. Discover what widgets are available, the names for those widgets, what properties you can set for each, etc.
  - Enforces separation of UI logic from application logic.
- Bad
  - If you need to add or remove widgets at run-time, you have to have that logic in code. I think it's a bad idea to put your UI logic in two places.
  - Making changes to nested layouts. When a layout has no widgets in it, it collapses, and it can be really hard to drag and drop a widget in to the location you want.

Hand coding

- Good
  - Fast if you are very familiar with Qt.
  - Best choice if you need to add or remove widgets at run-time.
  - Easier than Qt Designer if you have your own custom widgets.
  - With discipline, you can still seperate UI layout from behavior. Just put your code to create and layout widgets in one place, and your code to set signals and slots in another place.
- Bad
  - Slow if you are new to Qt.
  - Does not **enforce** separation of layout from behavior.



Qt Designer Manual

Contents

Qt Designer is Qt's tool for designing and building graphical user interfaces (GUIs) from Qt components. You can compose and customize your widgets or dialogs in a what-you-see-is-what-you-get (WYSIWYG) manner, and test them using different styles and resolutions.

Widgets and forms created with *Qt Designer* integrated seamlessly with programmed code, using Qt's signals and slots mechanism, that lets you easily assign behavior to graphical elements. All properties set in *Qt Designer* can be changed dynamically within the code. Furthermore, features like widget promotion and custom plugins allow you to use your own components with *Qt Designer*.

stackoverflow About Products For Teams

Home

PUBLIC

Qt Designer vs Handcoding

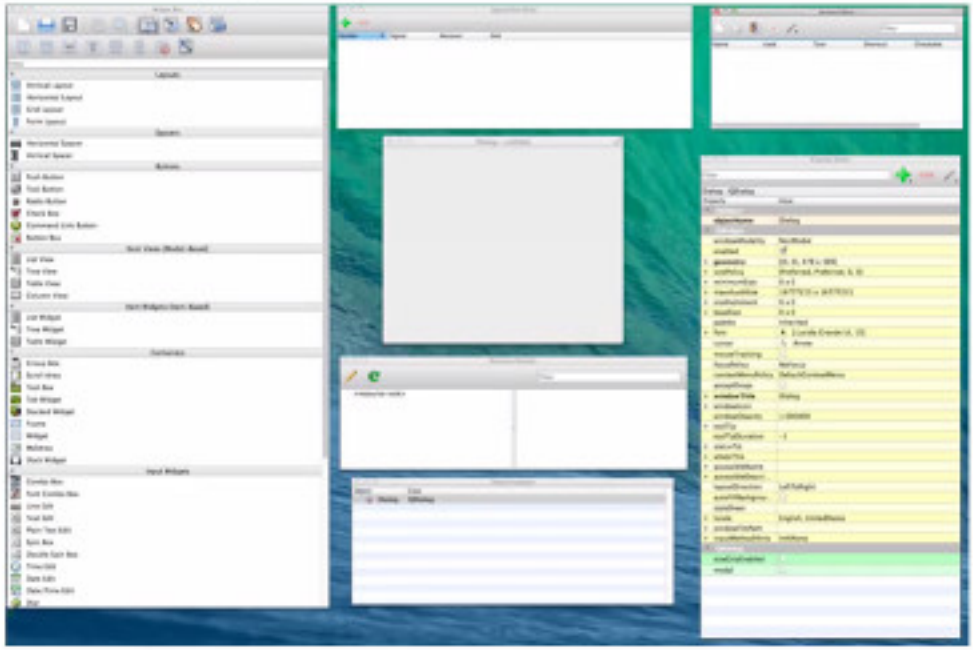
Asked 10 years, 1 month ago Active 3 months ago Viewed 15k times

Qt Designer

- Good
  - Exploration. Discover what widgets are available, the names for those widgets, what properties you can set for each, etc.
  - Enforces separation of UI logic from application logic.
- Bad
  - If you need to add or remove widgets at run-time, you have to have that logic in code. I think it's a bad idea to put your UI logic in two places.
  - Making changes to nested layouts. When a layout has no widgets in it, it collapses, and it can be really hard to drag and drop a widget in to the location you want.

Hand coding

- Good
  - Fast if you are very familiar with Qt.
  - Best choice if you need to add or remove widgets at run-time.
  - Easier than Qt Designer if you have your own custom widgets.
  - With discipline, you can still seperate UI layout from behavior. Just put your code to create and layout widgets in one place, and your code to set signals and slots in another place.
- Bad
  - Slow if you are new to Qt.
  - Does not **enforce** separation of layout from behavior.



- Qt è completamente ad oggetti ed event driven
- Qt estende C++ con segnali e slot utilizzando il Meta Object Compiler (moc)

The Meta-Object Compiler, moc, is the program that handles Qt's C++ extensions.

The moc tool reads a C++ header file. If it finds one or more class declarations that contain the Q_OBJECT macro, it produces a C++ source file containing the meta-object code for those classes. Among other things, meta-object code is required for the signals and slots mechanism, the run-time type information, and the dynamic property system.

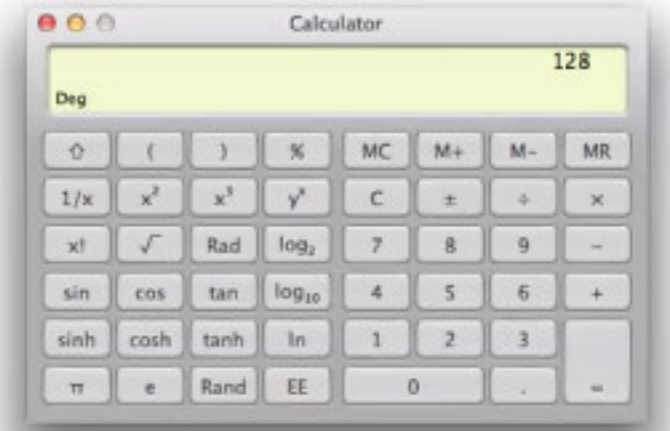
The C++ source file generated by moc must be compiled and linked with the implementation of the class.

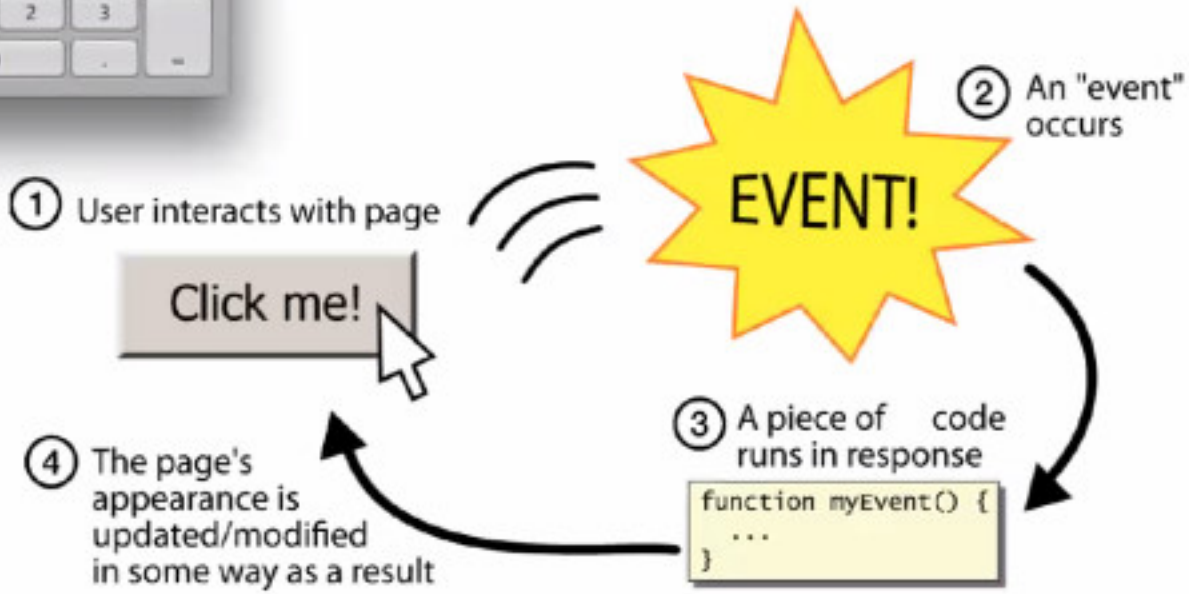
If you use qmake to create your makefiles, build rules will be included that call the moc when required, so you will not need to use the moc directly. For more background information on moc, see Why Doesn't Qt Use Templates for Signals and Slots?

- Qt è completamente ad oggetti ed event driven
- Qt estende C++ con segnali e slot utilizzando il Meta Object Compiler (moc)
- Varie funzionalità nonGUI:
  - accesso a database SQL
  - Qcontainers
  - JavaScript
  - XML and JSON support
  - concurrent programming (threads)
  - network programming
  - etc

Event Driven Programming

Standard example: the calculator





- In Qt tutte le classi ereditano da QObject
- QObject ha varie caratteristiche interessanti:
  - Relazione di parent/siblings (con gestione automatica della distruzione parentale)
  - Signals/Slots

Detailed Description

The QObject class is the base class of all Qt objects.

QObject is the heart of the Qt Object Model. The central feature in this model is a very powerful mechanism for seamless object communication called signals and slots. You can connect a signal to a slot with connect() and destroy the connection with disconnect().

Object Trees & Ownership

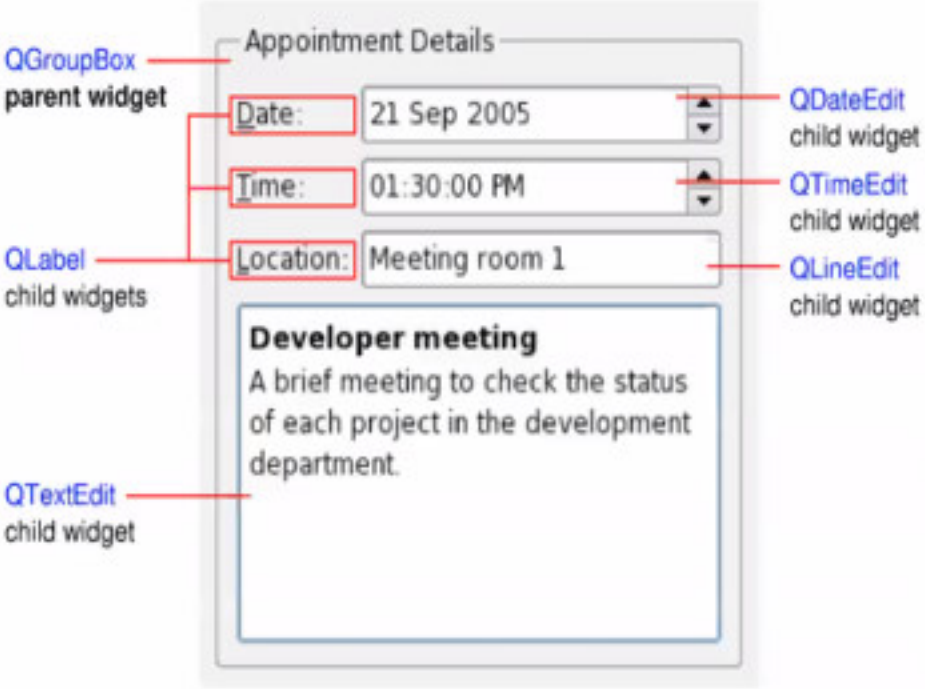
Overview

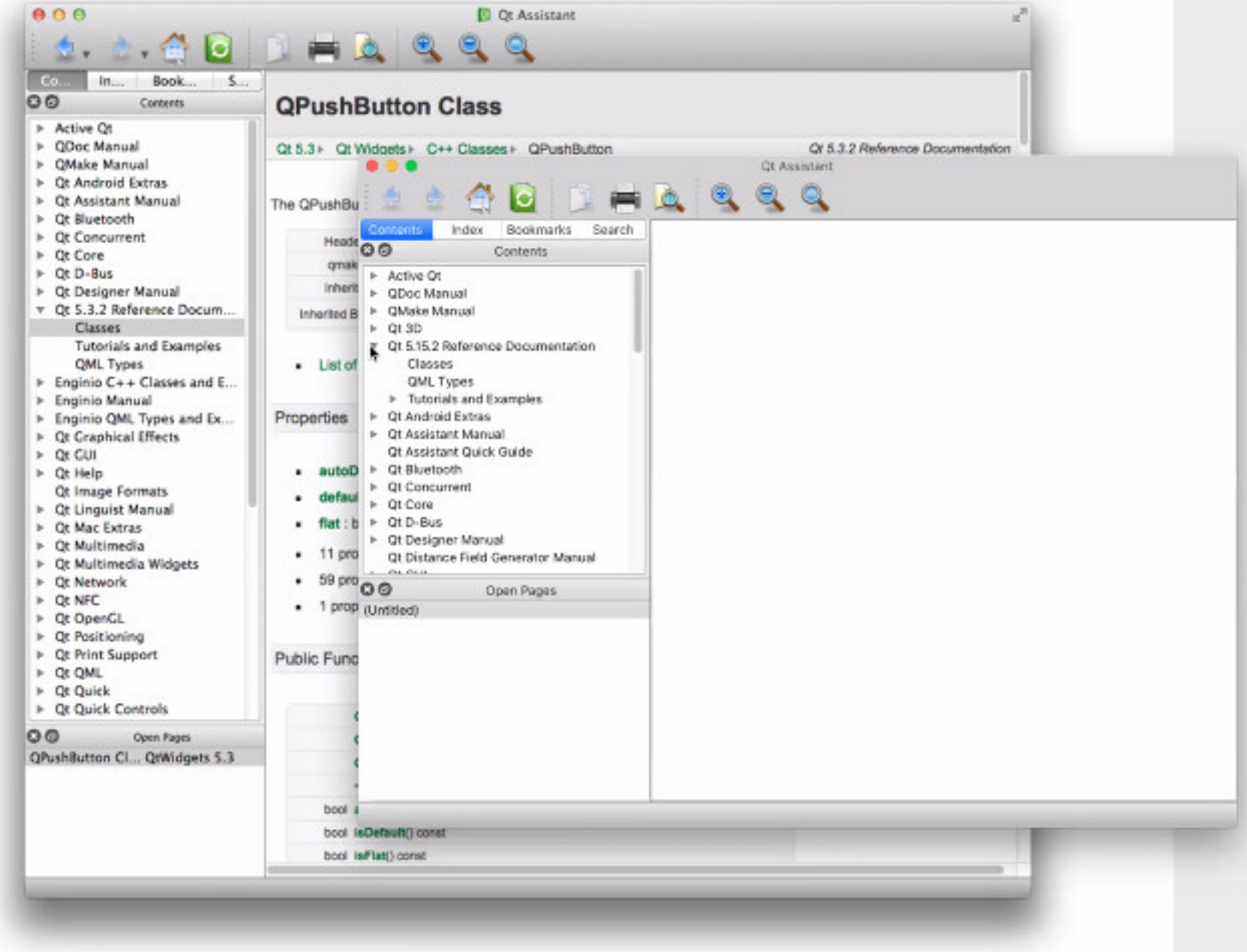
QObjects organize themselves in object trees. When you create a QObject with another object as parent, it's added to the parent's children() list, and is deleted when the parent is. It turns out that this approach fits the needs of GUI objects very well. For example, a QShortcut (keyboard shortcut) is a child of the relevant window, so when the user closes that window, the shortcut is deleted too.

- QWidget eredita da QObject
- In Qt tutte le classe GUI ereditano da QWidget. Ogni elemento visibile di una GUI è una QWidget.

The QWidget class is the base class of all user interface objects.
More...

Header:	`#include <QWidget>`
qmake:	`QT += widgets`
Inherits:	QObject and QPaintDevice.
Inherited By:	QAbstractButton, QAbstractSlider, QAbstractSpinBox, QCalendarWidget, QComboBox, QDesktopWidget, QDialog, QDialogButtonBox, QDockWidget, QFocusFrame, QFrame, QGroupBox, QKeySequenceEdit, QLineEdit, QMacCocoaViewContainer, QMacNativeWidget, QMainWindow, QMdiSubWindow, QMenu, QMenuBar, QProgressBar, QRubberBand, QSizeGrip, QSplashScreen, QSplitterHandle, QStatusBar, QTabBar, QTabWidget, QToolBar, and QWizardPage.




Qt Assistant
QPushButton Class
Qt 5.3 ▸ Qt Widgets ▸ C++ Classes ▸ QPushButton
Qt 5.3.2 Reference Documentation
Contents
Active Qt
QDoc Manual
QMake Manual
Qt Android Extras
Qt Assistant Manual
Qt Bluetooth
Qt Concurrent
Qt Core
Qt D-Bus
Qt Designer Manual
Qt 5.3.2 Reference Docum...
Classes
Tutorials and Examples
QML Types
Enginio C++ Classes and E...
Enginio Manual
Enginio QML Types and Ex...
Qt Graphical Effects
Qt GUI
Qt Help
Qt Image Formats
Qt Linguist Manual
Qt Mac Extras
Qt Multimedia
Qt Multimedia Widgets
Qt Network
Qt NFC
Qt OpenGL
Qt Positioning
Qt Print Support
Qt QML
Qt Quick
Qt Quick Controls
Open Pages
QPushButton Cl... QtWidgets 5.3
Properties
Public Func
bool isDefault() const
bool isFlat() const
Qt Assistant
Contents
Index
Bookmarks
Search
Contents
Active Qt
QDoc Manual
QMake Manual
Qt 3D
Qt 5.15.2 Reference Documentation
Classes
QML Types
Tutorials and Examples
Qt Android Extras
Qt Assistant Manual
Qt Assistant Quick Guide
Qt Bluetooth
Qt Concurrent
Qt Core
Qt D-Bus
Qt Designer Manual
Qt Distance Field Generator Manual
Open Pages
(Untitled)

Qt Assistant

Contents | Index | Bookmarks | Search

Contents

- Active Qt
- QDoc Manual
- QMake Manual
- Qt 3D
- Qt 5.15.2 Reference Documentation
  - Classes
  - QML Types
  - Tutorials and Examples
- Qt Android Extras
- Qt Assistant Manual
- Qt Assistant Quick Guide
- Qt Bluetooth
- Qt Concurrent
- Qt Core
- Qt D-Bus
- Qt Designer Manual
- Qt Distance Field Generator Manual
- Qt GUI
- Qt Gamepad
- Qt Graphical Effects
- Qt Help
- Qt Image Formats
- Qt Labs Calendar
- Qt Labs Platform
- Qt Linguist Manual
- Qt Location
- Qt Mac Extras

Open Pages

QPushButton Class | Qt Widgets 5.15.2

- Qt 5.15
- Qt Widgets
- C++ Classes
- QPushButton
- Qt 5.15.2 Reference Documentation

Contents

QPushButton Class

The QPushButton widget provides a command button. More...

Header:	#include <QPushButton>
qmake:	QT += widgets
Inherits:	QAbstractButton
Inherited By:	QCommandLinkButton

- List of all members, including inherited members

Properties

- **autoDefault** : bool
- **default** : bool
- **flat** : bool

Public Functions

QPushButton(const QIcon &*icon*, const QString &*text*, QWidget **parent* = nullptr)

QPushButton(const QString &*text*, QWidget **parent* = nullptr)

QPushButton(QWidget **parent* = nullptr)

Qt Assistant

Contents | Index | Bookmarks | Search

Contents

- Active Qt
- QDoc Manual
- QMake Manual
- Qt 3D
- Qt 5.15.2 Reference Documentation
  - Classes
  - QML Types
  - Tutorials and Examples
- Qt Android Extras
- Qt Assistant Manual
- Qt Assistant Quick Guide
- Qt Bluetooth
- Qt Concurrent
- Qt Core
- Qt D-Bus
- Qt Designer Manual
- Qt Distance Field Generator Manual
- Qt GUI
- Qt Gamepad
- Qt Graphical Effects
- Qt Help
- Qt Image Formats
- Qt Labs Calendar
- Qt Labs Platform
- Qt Linguist Manual
- Qt Location
- Qt Mac Extras

Open Pages

QPushButton Class | Qt Widgets 5.15.2

qmake: QT += widgets

Inherits: QAbstractButton

Inherited By: QCommandLinkButton

- List of all members, including inherited members

Properties

- **autoDefault** : bool
- **default** : bool
- **flat** : bool

Public Functions

	QPushButton(const QIcon &icon, const QString &text, QWidget *parent = nullptr)
	QPushButton(const QString &text, QWidget *parent = nullptr)
	QPushButton(QWidget *parent = nullptr)
virtual	**~QPushButton**()
bool	**autoDefault**() const
bool	**isDefault**() const
bool	**isFlat**() const
QMenu *	**menu**() const
void	**setAutoDefault**(bool)
void	**setDefault**(bool)
void	**setFlat**(bool)
void	**setMenu**(QMenu *menu)

Reimplemented Public Functions

virtual QSize	**minimumSizeHint**() const override
virtual QSize	**sizeHint**() const override

Qt Assistant

Contents | Index | Bookmarks | Search

Contents

- Active Qt
- QDoc Manual
- QMake Manual
- Qt 3D
- Qt 5.15.2 Reference Documentation
  - Classes
  - QML Types
  - Tutorials and Examples
- Qt Android Extras
- Qt Assistant Manual
- Qt Assistant Quick Guide
- Qt Bluetooth
- Qt Concurrent
- Qt Core
- Qt D-Bus
- Qt Designer Manual
- Qt Distance Field Generator Manual
- Qt GUI
- Qt Gamepad
- Qt Graphical Effects
- Qt Help
- Qt Image Formats
- Qt Labs Calendar
- Qt Labs Platform
- Qt Linguist Manual
- Qt Location
- Qt Mac Extras

Open Pages

QPushButton Class | Qt Widgets 5.15.2

void setAutoDefault(bool)

void setDefault(bool)

void setFlat(bool)

void setMenu(QMenu *menu)

Reimplemented Public Functions

virtual QSize minimumSizeHint() const override

virtual QSize sizeHint() const override

Public Slots

void showMenu()

Protected Functions

void initStyleOption(QStyleOptionButton *option) const

Reimplemented Protected Functions

virtual bool event(QEvent *e) override

virtual void focusInEvent(QFocusEvent *e) override

virtual void focusOutEvent(QFocusEvent *e) override

virtual bool hitButton(const QPoint &pos) const override

virtual void keyPressEvent(QKeyEvent *e) override

virtual void paintEvent(QPaintEvent *) override

Detailed Description

Cancel

Qt Assistant

Contents | Index | Bookmarks | Search

Contents

- Active Qt
- QDoc Manual
- QMake Manual
- Qt 3D
- Qt 5.15.2 Reference Documentation
  - Classes
  - QML Types
  - Tutorials and Examples
- Qt Android Extras
- Qt Assistant Manual
- Qt Assistant Quick Guide
- Qt Bluetooth
- Qt Concurrent
- Qt Core
- Qt D-Bus
- Qt Designer Manual
- Qt Distance Field Generator Manual
- Qt GUI
- Qt Gamepad
- Qt Graphical Effects
- Qt Help
- Qt Image Formats
- Qt Labs Calendar
- Qt Labs Platform
- Qt Linguist Manual
- Qt Location
- Qt Mac Extras

Open Pages

QPushButton Class | Qt Widgets 5.15.2

Detailed Description

The push button, or command button, is perhaps the most commonly used widget in any graphical user interface. Push (click) a button to command the computer to perform some action, or to answer a question. Typical buttons are OK, Apply, Cancel, Close, Yes, No and Help.

A command button is rectangular and typically displays a text label describing its action. A shortcut key can be specified by preceding the preferred character with an ampersand in the text. For example:

```
QPushButton *button = new QPushButton("&Download", this);
```

In this example the shortcut is *Alt+D*. See the QShortcut documentation for details (to display an actual ampersand, use '&&').

Push buttons display a textual label, and optionally a small icon. These can be set using the constructors and changed later using setText() and setIcon(). If the button is disabled, the appearance of the text and icon will be manipulated with respect to the GUI style to make the button look "disabled".

A push button emits the signal clicked() when it is activated by the mouse, the Spacebar or by a keyboard shortcut. Connect to this signal to perform the button's action. Push buttons also provide less commonly used signals, for example pressed() and released().

Command buttons in dialogs are by default auto-default buttons, i.e., they become the default push button automatically when they receive the keyboard input focus. A default button is a push button that is activated when the user presses the Enter or Return key in a dialog. You can change this with setAutoDefault(). Note that auto-default buttons reserve a little extra space which is necessary to draw a default-button indicator. If you do not want this space around your buttons, call setAutoDefault(false).

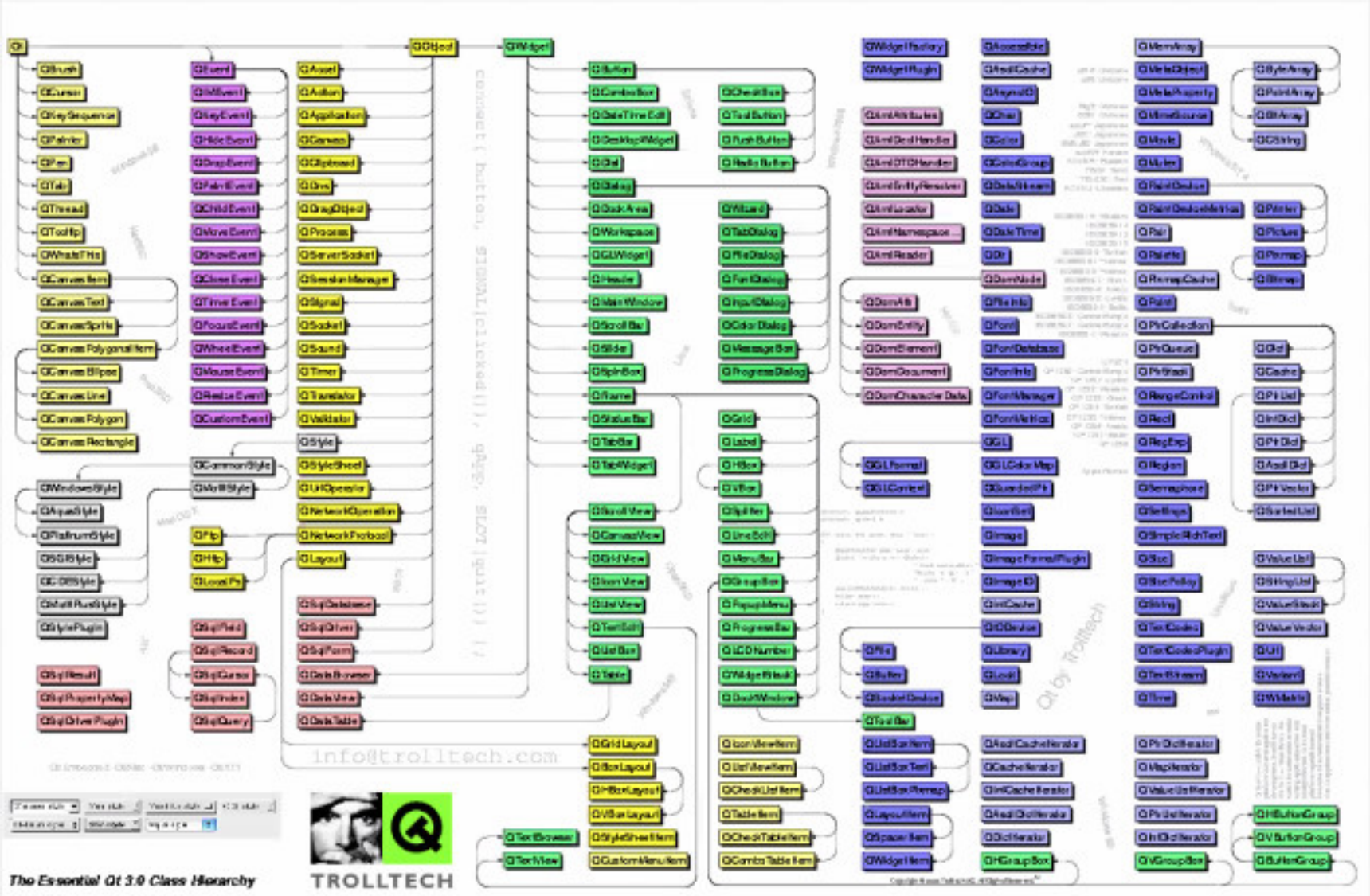
Being so central, the button widget has grown to accommodate a great many variations in the past decade. The Microsoft style guide now shows about ten different states of Windows push buttons and the text implies that there are dozens more when all the combinations of features are taken into consideration.

The most important modes or states are:

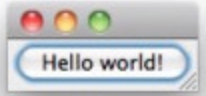
- Available or not (grayed out, disabled).
- Standard push button, toggling push button or menu button.
- On or off (only for toggling push buttons).
- Default or normal. The default button in a dialog can generally be "clicked" using the Enter or Return key.
- Auto-repeat or not.
- Pressed down or not.

As a general rule, use a push button when the application or dialog window performs an action when the user clicks on it (such as Apply, Cancel, Close and Help) and when the widget is supposed to have a wide, rectangular shape with a text label. Small, typically square buttons that change the state of the window rather than performing an action (such as the buttons in the top-right corner of the QFileDialog) are not command buttons, but tool buttons. Qt provides a special class (QToolButton) for these buttons.

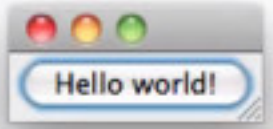
If you need toggle behavior (see setCheckable()) or a button that auto-repeats the activation signal when being pushed down like the arrows in a scroll bar (see setAutoRepeat()), a command button is probably not what you want. When in doubt, use a tool



The Essential Qt 3.0 Class Hierarchy




```
// file "main.cpp"

// inclusione di header files
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[]) {
  // una QApplication in ogni Qt GUI
  // argc e argv sono passati al costruttore di QApplication
  QApplication app(argc, argv);
```




```
// file "main.cpp"

// inclusione di header files
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[]) {
  // una QApplication in ogni Qt GUI
  // argc e argv sono passati al costruttore di QApplication
  QApplication app(argc, argv);
  // QPushButton è un widget
  // costruzione del QPushButton hello
  // hello non ha parent window, lui stesso è una window con frame e title bar
  QPushButton hello("Hello world!");
  // show() è uno slot, rende visibile un widget, che altrimenti non è visibile
  hello.show();
```

Contents Index Bookmarks Search

Contents

- Active Qt
- QDoc Manual
- QMake Manual
- Qt 3D
- Qt 5.15.2 Reference Documentation
  - Classes
  - QML Types
  - Tutorials and Examples
- Qt Android Extras
- Qt Assistant Manual
- Qt Assistant Quick Guide
- Qt Bluetooth
- Qt Concurrent
- Qt Core
- Qt D-Bus
- Qt Designer Manual
- Qt Distance Field Generator Manual
- Qt GUI
- Qt Gamepad
- Qt Graphical Effects
- Qt Help
- Qt Image Formats
- Qt Labs Calendar
- Qt Labs Platform
- Qt Linguist Manual
- Qt Location
- Qt Mac Extras
- Qt Multimedia

Open Pages

QPushButton Class | Qt Widgets 5.15.2

QPushButton Class

The QPushButton widget provides a command button. More...

Header: #include <QPushButton>

qmake: QT += widgets

Inherits: QAbstractButton

Inherited By: QCommandLinkButton

- List of all members, including inherited members

Properties

- **autoDefault** : bool
- **default** : bool
- **flat** : bool

Public Functions

	QPushButton(const QIcon &*icon*, const QString &*text*, QWidget **parent* = nullptr)
	QPushButton(const QString &*text*, QWidget **parent* = nullptr)
	QPushButton(QWidget **parent* = nullptr)
virtual	**~QPushButton**()
bool	**autoDefault**() const
bool	**isDefault**() const
bool	**isFlat**() const
QMenu *	**menu**() const
void	**setAutoDefault**(*bool*)
void	**setDefault**(*bool*)

qthelp://org.qt-project.qtwidgets.5152/qtwidgets/qpushbutton.html#QPushButton-1

```
// file

// incl
#includ
#includ

int mai
  // un
  // ar
  QAppl
  // QP
  // cc
  // he
  QPush
  // sh
  hello
```

Qt Assistant

Contents | Index | Bookmarks | Search

Contents

- Active Qt
- QDoc Manual
- QMake Manual
- Qt 3D
- Qt 5.15.2 Reference Documentation
  - Classes
  - QML Types
  - Tutorials and Examples
- Qt Android Extras
- Qt Assistant Manual
- Qt Assistant Quick Guide
- Qt Bluetooth
- Qt Concurrent
- Qt Core
- Qt D-Bus
- Qt Designer Manual
- Qt Distance Field Generator Manual
- Qt GUI
- Qt Gamepad
- Qt Graphical Effects
- Qt Help
- Qt Image Formats
- Qt Labs Calendar
- Qt Labs Platform
- Qt Linguist Manual
- Qt Location
- Qt Mac Extras
- Qt Multimedia

Open Pages

QString Class | Qt Core 5.15.2

typedef pointer

typedef reference

typedef reverse_iterator

typedef size_type

typedef value_type

Public Functions

QString(const QByteArray &*ba*)

QString(const char **str*)

QString(QString &&*other*)

QString(const QString &*other*)

QString(QLatin1String *str*)

QString(int *size*, QChar *ch*)

QString(QChar *ch*)

QString(const QChar **unicode*, int *size* = -1)

QString()

QString & operator=(const QByteArray &*ba*)

QString & operator=(QString &&*other*)

QString & operator=(const QString &*other*)

~QString()

QString & append(const QString &*str*)

QString & append(QChar *ch*)

QString & append(const QChar **str*, int *len*)

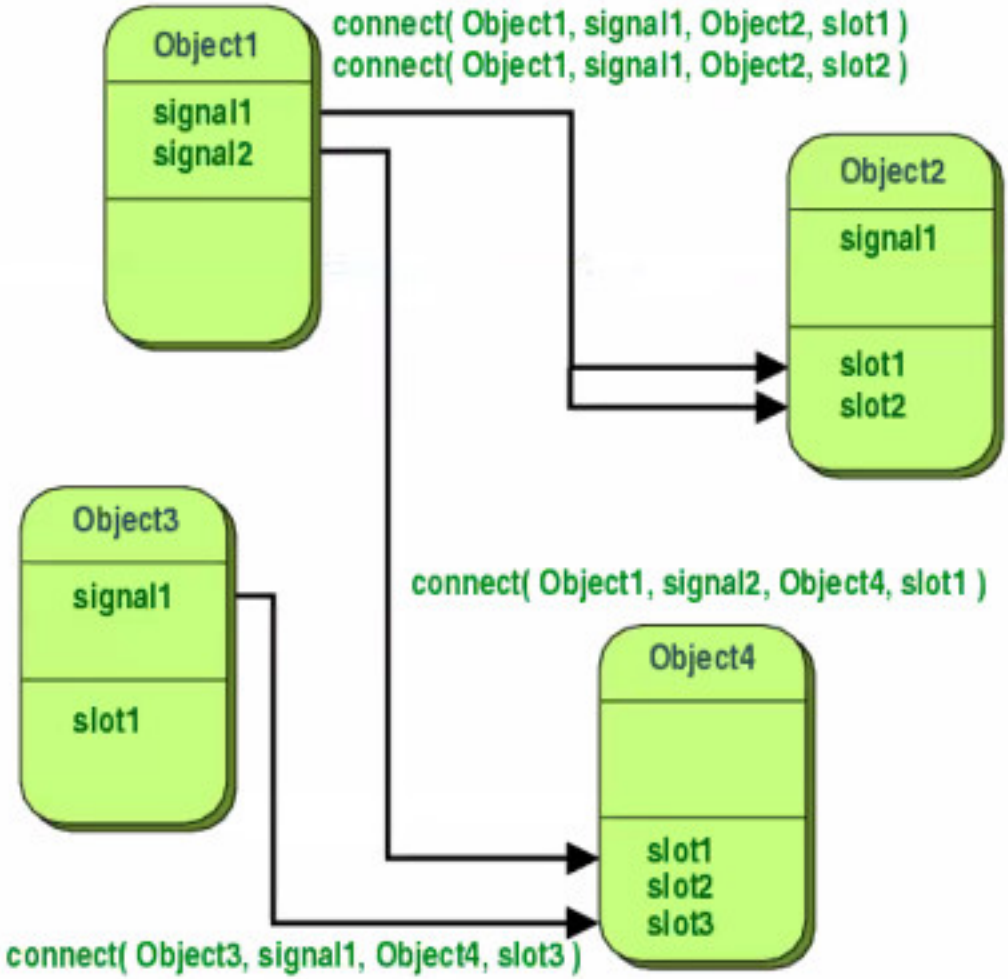
QString & append(const QStringRef &*reference*)

Compilazione manuale da shell

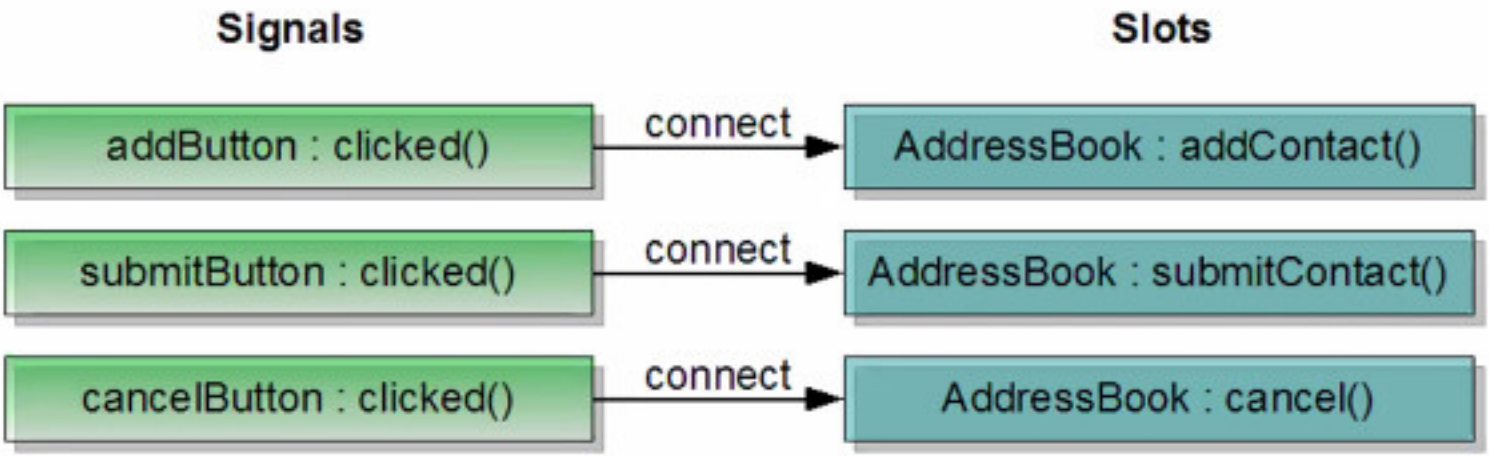
ex01 — bash — 89×44

```
ranzato-timecapsule0:ex01 francescoranzato$ qmake -project
ranzato-timecapsule0:ex01 francescoranzato$ ls
.DS_Store  ex01.pro   main.cpp
ranzato-timecapsule0:ex01 francescoranzato$ qmake
Info: creating stash file /Users/francescoranzato/Dropbox/MacBookPro/dida/15-16/pogg/slid
es/Qt/ex01/.qmake.stash
ranzato-timecapsule0:ex01 francescoranzato$ ls
.DS_Store     .qmake.stash  Makefile      ex01.pro      main.cpp
ranzato-timecapsule0:ex01 francescoranzato$ make
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cl
ang++ -c -pipe -O2 -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.
platform/Developer/SDKs/MacOSX10.9.sdk -mmacosx-version-min=10.6 -Wall -W -fPIE -DQT_NO_D
EBUG -DQT_WIDGETS_LIB -DQT_GUI_LIB -DQT_CORE_LIB -I/Users/francescoranzato/Qt5.3.2/5.3/cl
ang_64/mkspecs/macx-clang -I. -I. -I/Users/francescoranzato/Qt5.3.2/5.3/clang_64/lib/QtWi
dgets.framework/Versions/5/Headers -I/Users/francescoranzato/Qt5.3.2/5.3/clang_64/lib/QtG
ui.framework/Versions/5/Headers -I/Users/francescoranzato/Qt5.3.2/5.3/clang_64/lib/QtCore
.framework/Versions/5/Headers -I. -I/Applications/Xcode.app/Contents/Developer/Platforms/
MacOSX.platform/Developer/SDKs/MacOSX10.9.sdk/System/Library/Frameworks/OpenGL.framework/
Versions/A/Headers -I/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform
/Developer/SDKs/MacOSX10.9.sdk/System/Library/Frameworks/AGL.framework/Headers -F/Users/f
rancescoranzato/Qt5.3.2/5.3/clang_64/lib -o main.o main.cpp
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cl
ang++ -headerpad_max_install_names -Wl,-syslibroot,/Applications/Xcode.app/Contents/Devel
oper/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.9.sdk -mmacosx-version-min=10.6 -o
 ex01.app/Contents/MacOS/ex01 main.o   -F/Users/francescoranzato/Qt5.3.2/5.3/clang_64/lib
 -framework QtWidgets -framework QtGui -framework QtCore -framework OpenGL -framework AGL

ranzato-timecapsule0:ex01 francescoranzato$ ls
.DS_Store     Makefile      ex01.pro      main.o
.qmake.stash  ex01.app/     main.cpp
ranzato-timecapsule0:ex01 francescoranzato$
```

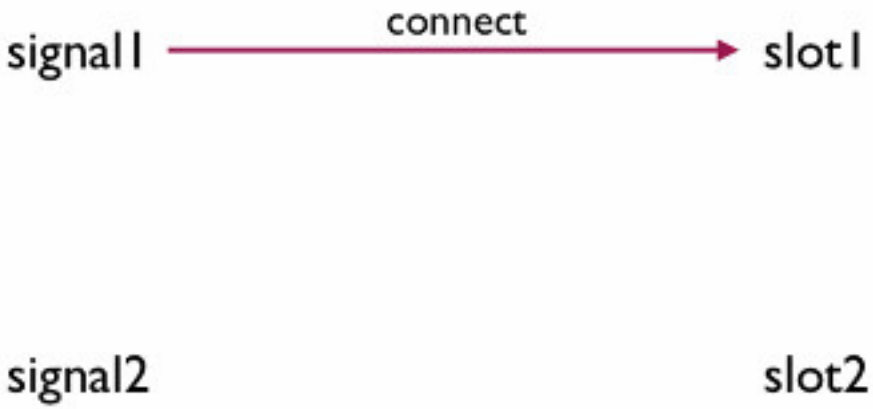
Signal e slot



Signal e slot



Signal e slot



- Ogni QObject definisce i segnali che può emettere
- Ogni QObject può connettere i propri slots a diversi segnali (tipicamente di altri oggetti)
- Quando un QObject emette un segnale **sig**, tutti gli slot connessi a **sig** vengono invocati
- Un segnale può invocare diversi slots
- Uno slot può essere connesso a diversi segnali

QMetaObject::Connection QObject::connect(const QObject * *sender*, const char * *signal*, const QObject * *receiver*, const char * *method*, Qt::ConnectionType *type* = Qt::AutoConnection) [static]

Creates a connection of the given *type* from the *signal* in the *sender* object to the *method* in the *receiver* object. Returns a handle to the connection that can be used to disconnect it later.

You must use the SIGNAL() and SLOT() macros when specifying the *signal* and the *method*, for example:

```
QLabel *label = new QLabel;
QScrollBar *scrollBar = new QScrollBar;
QObject::connect(scrollBar, SIGNAL(valueChanged(int)),
                 label,  SLOT(setNum(int)));
```

This example ensures that the label always displays the current scroll bar value. Note that the signal and slots parameters must not contain any variable names, only the type. E.g. the following would no work and return false:

```
// WRONG
QObject::connect(scrollBar, SIGNAL(valueChanged(int value)),
                 label, SLOT(setNum(int value)));
```

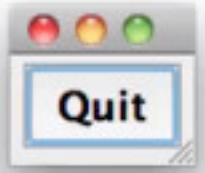
A signal can also be connected to another signal:

```
class MyWidget : public QWidget
{
    Q_OBJECT // moc: macro Q_OBJECT in ogni classe con signal/slot

public:
    MyWidget();

signals:
    void buttonClicked();

private:
    QPushButton *myButton;
};

MyWidget::MyWidget()
{
    myButton = new QPushButton(this);
    connect(myButton, SIGNAL(clicked()),
            this, SIGNAL(buttonClicked()));
}
```

In this example, the `MyWidget` constructor relays a signal from a private member variable, and makes it available under a name that relates to `MyWidget`.

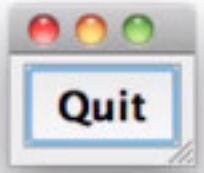
A signal can be connected to many slots and signals. Many signals can be connected to one slot.

If a signal is connected to several slots, the slots are activated in the same order in which the connections were made, when the signal is emitted.



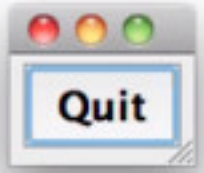
QObject::connect

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

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  // costruisce il pulsante con una label
  QPushButton quit("Quit");
  // ridimensiona il pulsante quit
  quit.resize(75, 30);
  // setta la font del pulsante quit
  quit.setFont(QFont("Times", 18, QFont::Bold));
```



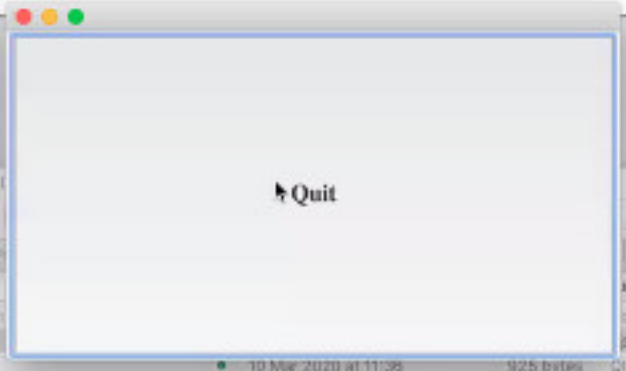
QObject::connect

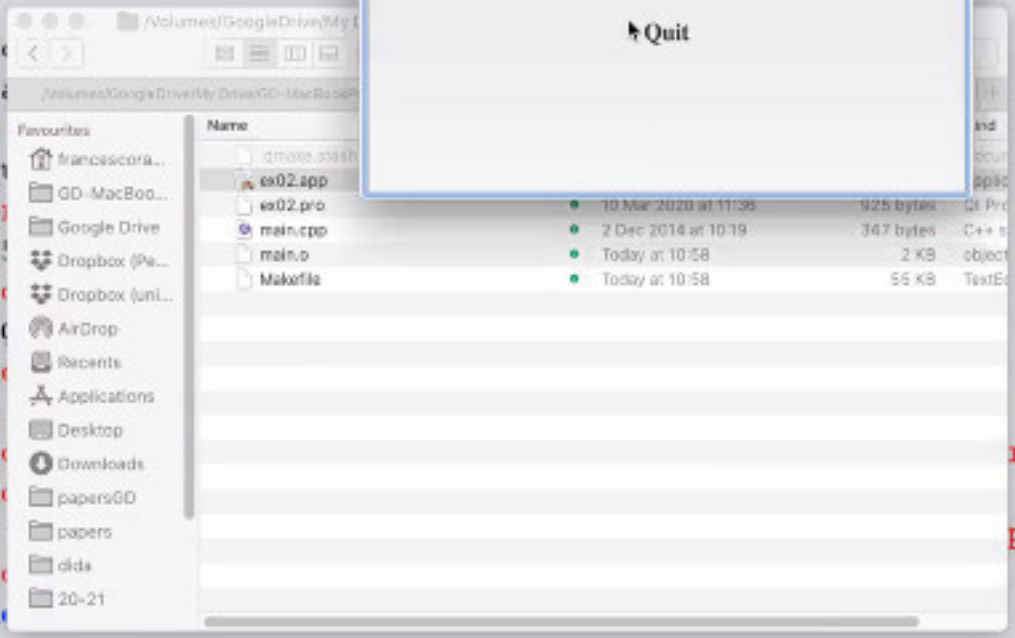
```
#include <QApplication>
#include <QFont>
#include <QPushButton>

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  // costruisce il pulsante con una label
  QPushButton quit("Quit");
  // ridimensiona il pulsante quit
  quit.resize(75, 30);
  // setta la font del pulsante quit
  quit.setFont(QFont("Times", 18, QFont::Bold));
  // invocazione di connect(), metodo statico di QObject
  // stabilisce una connessione tra due QObject
  // Ogni QObject (e quindi ogni QWidget) può avere signal (mandare messaggi)
  // e slot (ricevere messaggi)
  // Il segnale clicked di quit è connesso allo slot quit() di app:
  // quando si clicka il pulsante quit l'applicazione app termina
  QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));
  quit.show();
  return QApplication::exec();
}
```



QObject::connect

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

int main(int ar
  QApplication a
  // costruisce
  QPushButton qu
  // ridimensio
  quit.resize(7
  // setta la fo
  quit.setFont(
  // invocazion
  // stabilisce
  // Ogni QObje                                  re messaggi)
  // e slot (ri
  // Il segnale                                  p:
  // quando si
  QObject::conn
  quit.show();
  return QApplication::exec();
}
```





void **QAbstractButton::clicked(bool *checked* = false)** [signal]

This signal is emitted when the button is activated (i.e., pressed down then released while the mouse cursor is inside the button), when the shortcut key is typed, or when click() or animateClick() is called. Notably, this signal is *not* emitted if you call setDown(), setChecked() or toggle().

If the button is checkable, *checked* is true if the button is checked, or false if the button is unchecked.

void **QCoreApplication::quit()** [static slot]

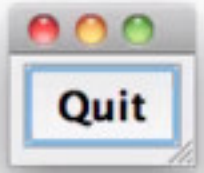
Tells the application to exit with return code 0 (success). Equivalent to calling QCoreApplication::exit(0).

It's common to connect the QApplication::lastWindowClosed() signal to quit(), and you also often connect e.g. QAbstractButton::clicked() or signals in QAction, QMenu, or QMenuBar to it.

Example:

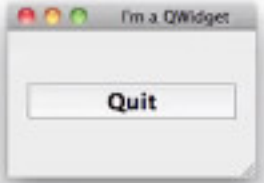
```
QPushButton *quitButton = new QPushButton("Quit");
connect(quitButton, SIGNAL(clicked()), &app, SLOT(quit()));
```

See also exit(), aboutToQuit(), and QApplication::lastWindowClosed().



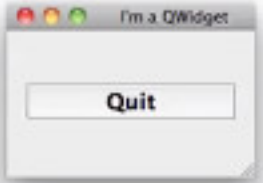
QObject::connect

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

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  // costruisce il pulsante con una label
  QPushButton quit("Quit");
  // ridimensiona il pulsante quit
  quit.resize(75, 30);
  // setta la font del pulsante quit
  quit.setFont(QFont("Times", 18, QFont::Bold));
  // invocazione di connect(), metodo statico di QObject
  // stabilisce una connessione tra due QObject
  // Ogni QObject (e quindi ogni QWidget) può avere signal (mandare messaggi)
  // e slot (ricevere messaggi)
  // Il segnale clicked di quit è connesso allo slot quit() di app:
  // quando si clicka il pulsante quit l'applicazione app termina
  QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));
  quit.show();
  return QApplication::exec();
}
```



Parent window

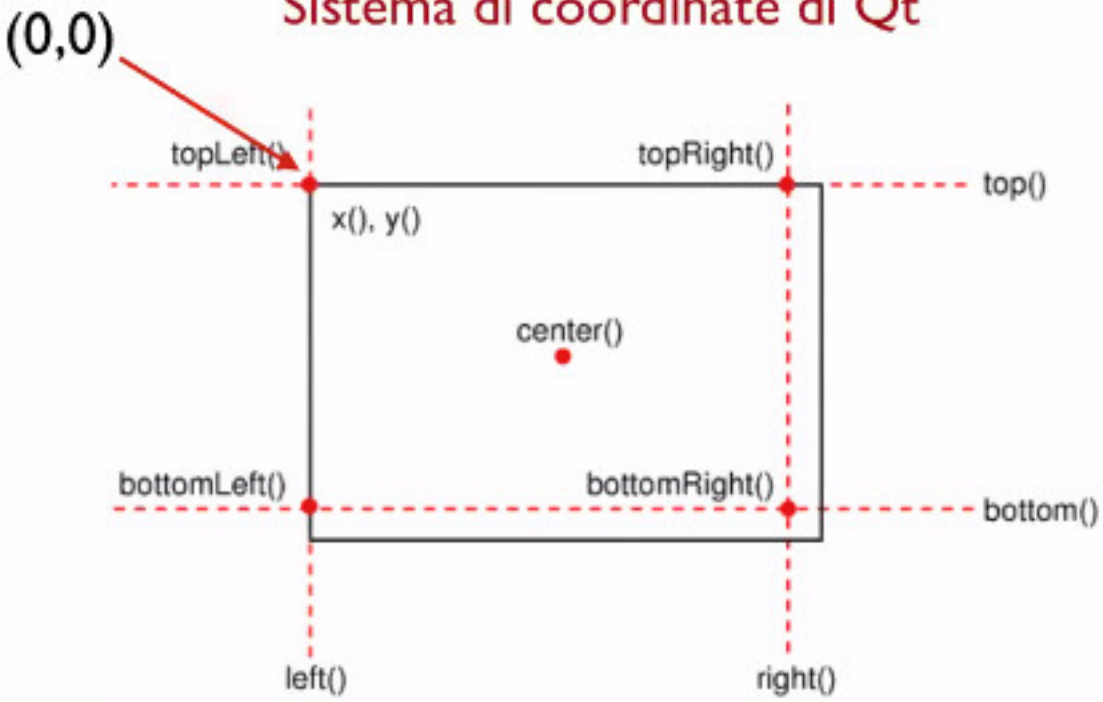
```
#include <QApplication>
#include <QFont>
#include <QPushButton>
#include <QWidget>

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  // QWidget è la classe base di tutti i widget
  // Un QWidget è un atomo di una GUI: riceve eventi dal sistema (mouse,
  // keyboard, etc), e rappresenta sè stessa sullo schermo.
  // Una QWidget è detenuta dal suo parent
  // Una QWidget senza parent è detta una independent window (con frame e
  // taskbar). La posizione iniziale è controllata dal sistema
  QWidget window;
  // setta il titolo
  window.setWindowTitle("I'm a QWidget");
  // ridimensionamento di window
  window.resize(200, 120);
  // quit ha come parent window, ovvero quit è figlio di window
  // Un figlio è sempre mostrato nell'area del suo parent, per default
  // al top-left corner alla posizione (0,0)
  QPushButton quit("Quit", &window);
```

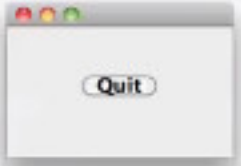


Parent window

```
#include <QApplication>
#include <QFont>
#include <QPushButton>
#include <QWidget>

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  // QWidget è la classe base di tutti i widget
  // Un QWidget è un atomo di una GUI: riceve eventi dal sistema (mouse,
  // keyboard, etc), e rappresenta sè stessa sullo schermo.
  // Una QWidget è detenuta dal suo parent
  // Una QWidget senza parent è detta una independent window (con frame e
  // taskbar). La posizione iniziale è controllata dal sistema
  QWidget window;
  // setta il titolo
  window.setWindowTitle("I'm a QWidget");
  // ridimensionamento di window
  window.resize(200, 120);
  // quit ha come parent window, ovvero quit è figlio di window
  // Un figlio è sempre mostrato nell'area del suo parent, per default
  // al top-left corner alla posizione (0,0)
  QPushButton quit("Quit", &window);
  quit.setFont(QFont("Times", 18, QFont::Bold));
  // QWidget::setGeometry(x,y,w,h):
  // (x,y) coordinate del top-left corner in pixel
  // (w,h) base ed altezza in pixel
  quit.setGeometry(10, 40, 180, 40);
  QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));
  // la chiamata di show() su window chiama show() anche su tutti i figli
  window.show();
  return app.exec();
}
```

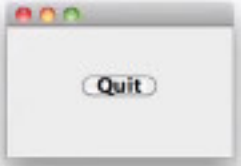
Sistema di coordinate di Qt





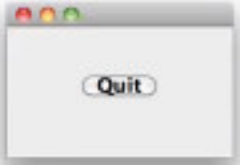
Widget class

```
// eredito da QWidget, posso quindi essere un top-level widget
class MyWidget : public QWidget {
public:
  // costruttore con argomento il QWidget parent,
  // dove il default 0 significa top-level
  MyWidget(QWidget *parent = 0) : QWidget(parent) {
```



Widget class

```
// eredito da QWidget, posso quindi essere un top-level widget
class MyWidget : public QWidget {
public:
  // costruttore con argomento il QWidget parent,
  // dove il default 0 significa top-level
  MyWidget(QWidget *parent = 0) : QWidget(parent) {
    // dimensiona fissa
    setFixedSize(200, 120);
    // MyWidget ha un QPushButton come figlio
    // tr("Quit") marca la stringa "Quit" per possibili traduzioni run-time
    QPushButton* quit = new QPushButton(tr("Quit"), this);
    quit->setGeometry(62, 40, 75, 30);
    quit->setFont(QFont("Times", 18, QFont::Bold));
    // qApp è una variabile globale dichiarata in <QApplication>
    // che punta all'unica istanza di QApplication del programma
    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    // il puntatore quit è variabile locale, e non campo dati
    // Qt automaticamente distrugge il QPushButton quando MyWidget è distrutta
    // Quindi MyWidget non necessita di distruttore
    }
};
```



Widget class

```
// eredito da QWidget, posso quindi essere un top-level widget
class MyWidget : public QWidget {
public:
  // costruttore con argomento il QWidget parent,
  // dove il default 0 significa top-level
  MyWidget(QWidget *parent = 0) : QWidget(parent) {
    // dimensiona fissa
    setFixedSize(200, 120);
    // MyWidget ha un QPushButton come figlio
    // tr("Quit") marca la stringa "Quit" per possibili traduzioni run-time
    QPushButton* quit = new QPushButton(tr("Quit"), this);
    quit->setGeometry(62, 40, 75, 30);
    quit->setFont(QFont("Times", 18, QFont::Bold));
    // qApp è una variabile globale dichiarata in <QApplication>
    // che punta all'unica istanza di QApplication del programma
    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    // il puntatore quit è variabile locale, e non campo dati
    // Qt automaticamente distrugge il QPushButton quando MyWidget è distrutta
    // Quindi MyWidget non necessita di distruttore
    }
};

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