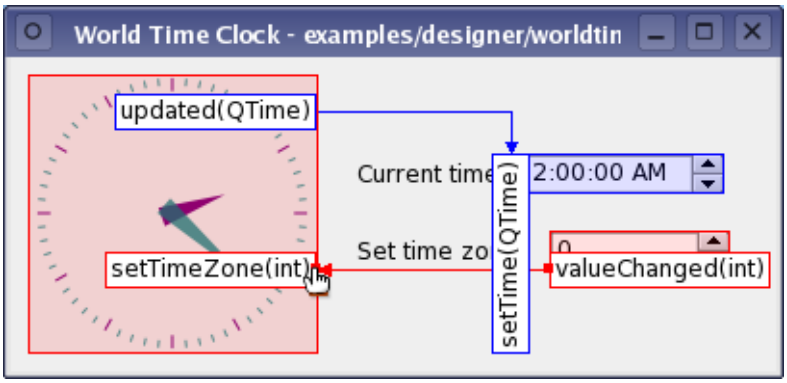
为Qt设计器创建自定义小部件

*Qt Designer*基于插件的架构允许用户定义的和第三方自定义小部件进行编辑，就像使用标准Qt小部件一样。所有自定义小部件的功能都可供*Qt Designer*使用，包括小部件属性，信号和插槽。由于*Qt Designer*在表单设计过程中使用真实的小部件，因此自定义小部件的显示方式将与预览时相同。



Qt设计者模块为您提供了在*Qt设计器*中创建自定义小部件的功能。

开始

要将自定义小部件与 *Qt Designer* 集成，您需要为该小部件提供合适的描述和适当的文件。`.pro`

提供接口描述

要通知 *Qt 设计器*您要提供的小组件类型，请创建 QDesignerCustomWidget接口的子类，用于描述小组件公开的各种属性。其中大多数是由基类中纯虚拟的函数提供的，因为只有插件的作者才能提供此信息。

功能	返回值的说明
`name()`	提供小组件的类的名称。
`group()`	*Qt 设计器*的小部件框中小部件所属的组。
`toolTip()`	帮助用户在*Qt设计器*中识别小部件的简短描述。
`whatsThis()`	为*Qt设计器*的用户提供的对小部件的较长描述。
`includeFile()`	必须包含在使用此小组件的应用程序中的头文件。此信息存储在 UI 文件中，并将用于在为包含自定义构件的窗体生成的代码中创建合适的语句。`uic#includes`
`icon()`	一个图标，可用于在*Qt设计器*的小部件框中表示小部件。
`isContainer()`	如果小部件将用于保存子小部件，则为 true;否则为假。
`createWidget()`	指向自定义小部件实例的 QWidget 指针，该实例是使用提供的父项构造的。

功能 返回值的说明

domXml()	小组件属性的描述，例如其对象名称、大小提示和其他标准 QWidget 属性。
codeTemplate()	此函数保留供 *Qt 设计器*将来使用。

还可以重新实现另外两个虚函数：

initialize()	为自定义小部件设置扩展程序和其他功能。应在此功能中设置自定义容器扩展（请参阅 QDesigner 容器扩展）和任务菜单扩展（请参阅 QDesigner 任务菜单扩展）。
isInitialized()	如果小部件已初始化，则返回 true;否则返回 false。重新实现通常会检查函数是否已被调用并返回此测试的结果。initialize()

关于函数的说明domXml()

该函数返回一个 UI 文件代码段，*Qt Designer* 的小组件工厂使用该代码段来创建自定义小组件及其适用属性。domXml()

从Qt 4.4开始，*Qt设计器*的小部件框允许一个完整的UI文件来描述**一个**自定义小部件。可以使用 标记加载 UI 文件。指定 <ui> 标记允许添加包含自定义小组件附加信息的 <自定义widget> 元素。如果不需要其他信息，则标记就足够了<ui><widget>

如果自定义小部件未提供合理的大小提示，则需要在子类中的函数返回的字符串中指定默认几何图形。例如，自定义小部件插件示例提供的示例按以下方式定义默认小部件几何：domXml()AnalogClockPlugin

```
    ...
R"(
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>100</width>
        <height>100</height>
      </rect>
    </property>
")
    ...
```

An additional feature of the function is that, if it returns an empty string, the widget will not be installed in *Qt Designer*'s widget box. However, it can still be used by other widgets in the form. This feature is used to hide widgets that should not be explicitly created by the user, but are required by other widgets.domXml()

A complete custom widget specification looks like:

```
<ui language="c++"> displayname="MyWidget">
    <widget class="widgets::MyWidget" name="mywidget"/>
    <customwidgets>
        <customwidget>
            <class>widgets::MyWidget</class>
            <addpagemethod>addPage</addpagemethod>
            <propertyspecifications>
                <stringpropertyspecification name="fileName" notr="true" type="singleline"/>
```

```
        </customwidget>
    </customwidgets>
</ui>
```

Attributes of the tag:`<ui>`

Attribute	Presence	Values	Comment
`language`	optional	"c++", "jambi"	This attribute specifies the language the custom widget is intended for. It is mainly there to prevent C++-plugins from appearing in Qt Jambi.
`displayname`	optional	Class name	The value of the attribute appears in the Widget box and can be used to strip away namespaces.

The tag tells *Qt Designer* and uic which method should be used to add pages to a container widget. This applies to container widgets that require calling a particular method to add a child rather than adding the child by passing the parent. In particular, this is relevant for containers that are not a a subclass of the containers provided in *Qt Designer*, but are based on the notion of *Current Page*. In addition, you need to provide a container extension for them.`<addpagemethod>`

The element can contain a list of property meta information.`<propertyspecifications>`

The tag may be used to specify a tool tip to be shown in Property Editor when hovering over the property. The property name is given in the attribute and the element text is the tooltip. This functionality was added in Qt 5.6.`<tooltip>name`

For properties of type string, the tag can be used. This tag has the following attributes:
`<stringpropertyspecification>`

Attribute	Presence	Values	Comment
`name`	required	Name of the property	
`type`	required	See below table	The value of the attribute determines how the property editor will handle them.
`notr`	optional	"true", "false"	If the attribute is "true", the value is not meant to be translated.

Values of the attribute of the string property:`type`

Value	Type
`"richtext"`	Rich text.
`"multiline"`	Multi-line plain text.
`"singleline"`	Single-line plain text.
`"stylesheet"`	A CSS-style sheet.
`"objectname"`	An object name (restricted set of valid characters).
`"url"`	URL, file name.

Plugin Requirements

In order for plugins to work correctly on all platforms, you need to ensure that they export the symbols needed by *Qt Designer*.

custom widget class within a plugin, that *Qt Designer* will instantiate.

Creating Well Behaved Widgets

Some custom widgets have special user interface features that may make them behave differently to many of the standard widgets found in *Qt Designer*. Specifically, if a custom widget grabs the keyboard as a result of a call to QWidget::grabKeyboard(), the operation of *Qt Designer* will be affected.

To give custom widgets special behavior in *Qt Designer*, provide an implementation of the initialize() function to configure the widget construction process for *Qt Designer* specific behavior. This function will be called for the first time before any calls to createWidget() and could perhaps set an internal flag that can be tested later when *Qt Designer* calls the plugin's createWidget() function.

Building and Installing the Plugin

A Simple Plugin

The Custom Widget Plugin Example demonstrates a simple *Qt Designer* plugin.

The file for a plugin must specify the headers and sources for both the custom widget and the plugin interface. Typically, this file only has to specify that the plugin's project is to be built as a library, but with specific plugin support for *Qt Designer*. This is done with the following declarations:`.pro`

```
QT          += widgets uiplugin
CONFIG      += plugin
TEMPLATE    = lib
```

The variable contains the keyword . It indicates that the plugin uses the abstract interfaces QDesignerCustomWidgetInterface and QDesignerCustomWidgetCollectionInterface only and has no linkage to the *Qt Designer* libraries. When accessing other interfaces of *Qt Designer* that have linkage, should be used instead; this ensures that the plugin dynamically links to the *Qt Designer* libraries and has a run-time dependency on them.`QTuiplugindesigner`

If plugins are built in a mode that is incompatible with *Qt Designer*, they will not be loaded and installed. For more information about plugins, see the Plugins HOWTO document.

It is also necessary to ensure that the plugin is installed together with other *Qt Designer* widget plugins:

```
target.path = $$[QT_INSTALL_PLUGINS]/designer
INSTALLS += target
```

The variable is a placeholder to the location of the installed Qt plugins. You can configure *Qt Designer* to look for plugins in other locations by setting the environment variable before running the application.`$[QT_INSTALL_PLUGINS]QT_PLUGIN_PATH`

> **Note:** *Qt Designer* will look for a subdirectory in each path supplied.`designer`

See QCoreApplication::libraryPaths() for more information about customizing paths for libraries and plugins with Qt applications.

Splitting up the Plugin

In a real world scenario, you do not want to have dependencies of the application making use of the custom widgets to the *Qt Designer* headers and libraries as introduced by the simple approach explained above.

The following sections describe how to resolve this.

Linking the Widget into the Application

The source and header file of the custom widget can be shared between the application and *Qt Designer* by creating a file for inclusion: `.pri`

```
INCLUDEPATH += $$PWD
HEADERS += $$PWD/analogclock.h
SOURCES += $$PWD/analogclock.cpp
```

This file would then be included by the file of the plugin and the application: `.pro`

```
include(customwidget.pri)
```

Sharing the Widget Using a Library

Another approach is to put the widget into a library that is linked to the *Qt Designer* plugin as well as to the application. It is recommended to use static libraries to avoid problems locating the library at run-time.

For shared libraries, see Creating Shared Libraries.

Using the Plugin with QUiLoader

The preferred way of adding custom widgets to QUiLoader is to subclass it reimplementing QUiLoader::createWidget().

However, it is also possible to use *Qt Designer* custom widget plugins (see QUiLoader::pluginPaths() and related functions). To avoid having to deploy the *Qt Designer* libraries onto the target device, those plugins should have no linkage to the *Qt Designer* libraries (, see Creating Custom Widgets for Qt Designer#BuildingandInstallingthePlugin).`QT = uiplugin`

Related Examples

For more information on using custom widgets in *Qt Designer*, refer to the Custom Widget Plugin and World Time Clock Plugin examples for more information about using custom widgets in *Qt Designer*. Also, you can use the QDesignerCustomWidgetCollectionInterface class to combine several custom widgets into a single library.

‹ Using Custom Widgets with Qt Designer

Creating Custom Widget Extensions ›

Qt DOCUMENTATION

Contact Us

Company

About Us
Investors
Newsroom
Careers
Office Locations

Licensing

Terms & Conditions
Open Source
FAQ

Support

Support Services
Professional Services
Partners
Training

For Customers

Support Center
Downloads
Qt Login
Contact Us
Customer Success

Community

Contribute to Qt
Forum
Wiki
Downloads
Marketplace

Feedback Sign In