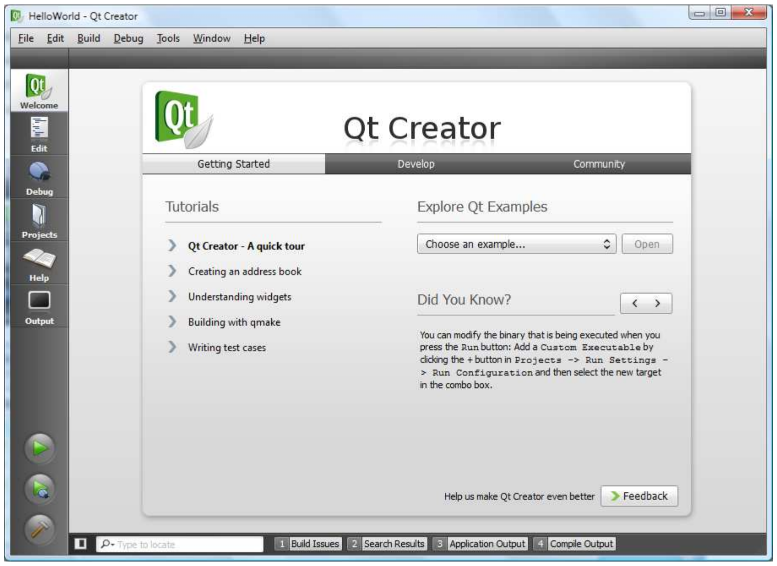
Writing Qt Creator Plugins (Beta)

Qt Creator is Nokia's cross-platform IDE. Since its release in early 2009, there has been a steady increase in the adoption of Qt Creator. In many places Qt Creator is the primary IDE for developing Qt software.



You can modify the binary that is being executed when you press the Run button: Add a `Custom Executable` by clicking the + button in `Projects -> Run Settings -> Run Configuration` and then select the new target in the combo box.

This document explains how to write plugins for Qt Creator. Most of the content of the documentation is derived from Qt Creator source code and documentation, which is ofcourse publicly available. We don't vouch for the accuracy of the content of this document. We __do__ want to tell you that the mechanisms described here worked for us. ☺. The document is a work in progress, we will be updating it from time to time as we learn more about Qt Creator plugin development.

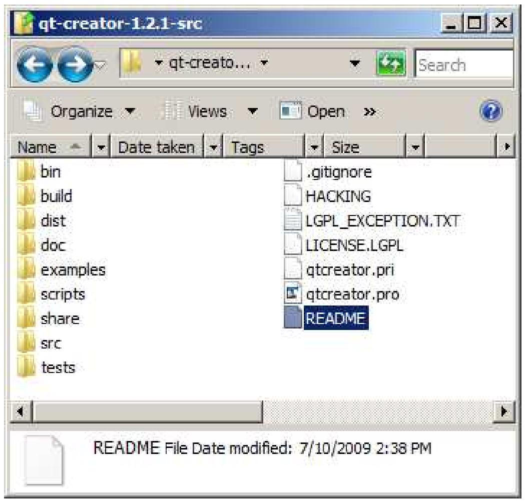
Another note before you read ahead: *The purpose set out for each plugin is really not production type. We have just chosen a simple purpose for each plugin so that the related concepts can be explained. None of the plugins described in this document may be of any real use to you or anyone else; but the concepts explained will be very useful ☺*

CONTENTS

1. Compiling Qt Creator

It recommended that you build Qt Creator in a separate directory.

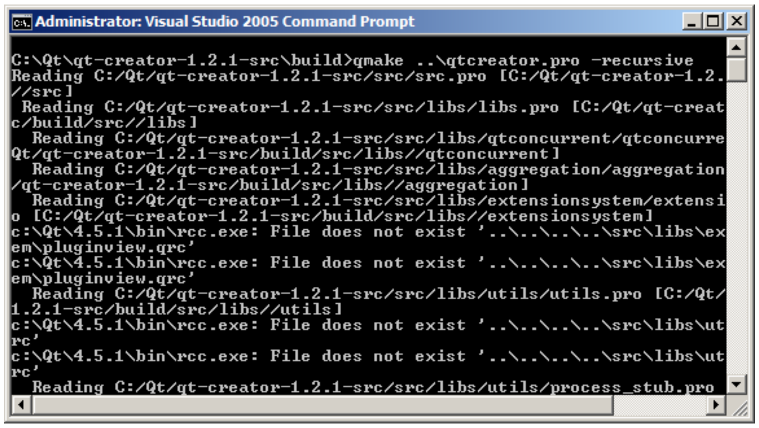
Step 1: Create "build" directory inside main source directory



```
bin          .gitignore
build        HACKING
dist         LGPL_EXCEPTION.TXT
doc          LICENSE.LGPL
examples     qtcreator.pri
scripts      qtcreator.pro
share        README
src
tests
```

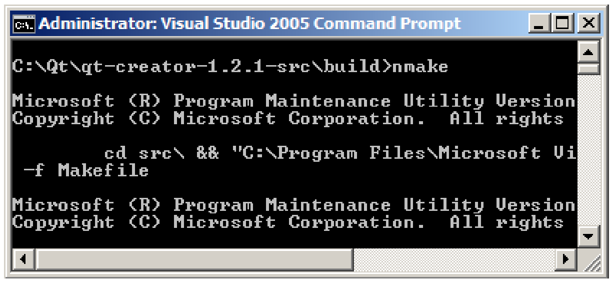
README File Date modified: 7/10/2009 2:38 PM

Step 2: Run qmake ..\qtcreator.pro –recursive command to get makefile.



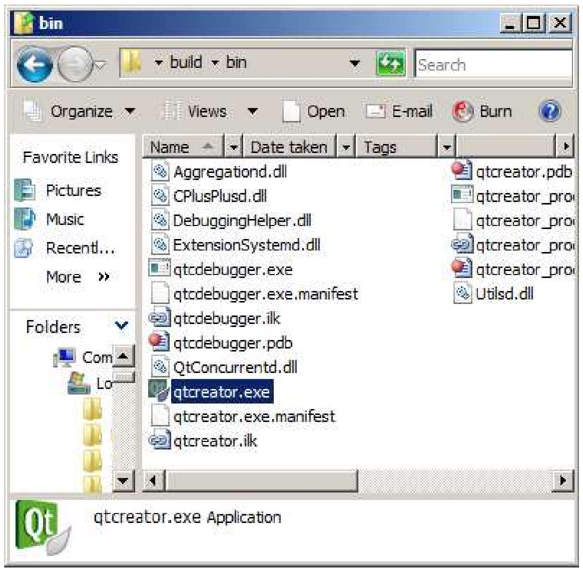
```
C:\Qt\qt-creator-1.2.1-src\build>qmake ..\qtcreator.pro -recursive
Reading C:/Qt/qt-creator-1.2.1-src/src/src.pro [C:/Qt/qt-creator-1.2.
//src]
 Reading C:/Qt/qt-creator-1.2.1-src/src/libs/libs.pro [C:/Qt/qt-creat
c/build/src//libs]
  Reading C:/Qt/qt-creator-1.2.1-src/src/libs/qtconcurrent/qtconcurre
Qt/qt-creator-1.2.1-src/build/src/libs//qtconcurrent]
  Reading C:/Qt/qt-creator-1.2.1-src/src/libs/aggregation/aggregation
/qt-creator-1.2.1-src/build/src/libs//aggregation]
  Reading C:/Qt/qt-creator-1.2.1-src/src/libs/extensionsystem/extensi
o [C:/Qt/qt-creator-1.2.1-src/build/src/libs//extensionsystem]
c:\Qt\4.5.1\bin\rcc.exe: File does not exist '..\..\..\..\src\libs\ex
em\pluginview.qrc'
c:\Qt\4.5.1\bin\rcc.exe: File does not exist '..\..\..\..\src\libs\ex
em\pluginview.qrc'
  Reading C:/Qt/qt-creator-1.2.1-src/src/libs/utils/utils.pro [C:/Qt/
1.2.1-src/build/src/libs//utils]
c:\Qt\4.5.1\bin\rcc.exe: File does not exist '..\..\..\..\src\libs\ut
rc'
c:\Qt\4.5.1\bin\rcc.exe: File does not exist '..\..\..\..\src\libs\ut
rc'
  Reading C:/Qt/qt-creator-1.2.1-src/src/libs/utils/process_stub.pro
```

Step 3: Finally run make (or mingw32-make or nmake, depending on your platform).



```
C:\Qt\qt-creator-1.2.1-src\build>nmake

Microsoft (R) Program Maintenance Utility Version
Copyright (C) Microsoft Corporation.  All rights

        cd src\ && "C:\Program Files\Microsoft Vi
 -f Makefile

Microsoft (R) Program Maintenance Utility Version
Copyright (C) Microsoft Corporation.  All rights
```

After successful compilation, you should be able to find `qtcreator.exe` inside `build\bin` directory.

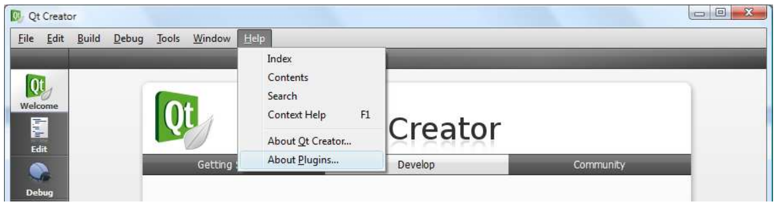


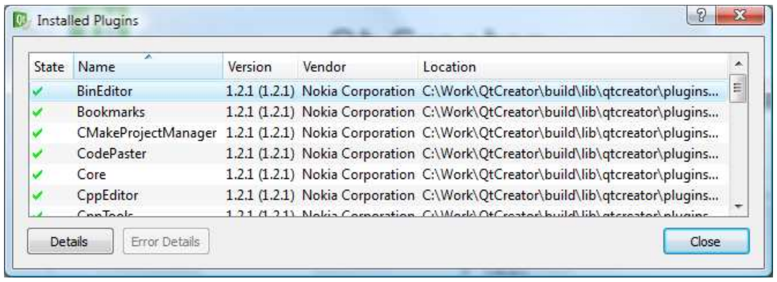
You can now launch Qt Creator.

It is important that you compile Qt Creator from its sources. Otherwise you wont be able to write and test plugins for Qt Creator.

2. First Plugin

The best way to learn about writing Qt Creator plugins is to actually start by writing the very first plugin. Let's keep our goals very simple for this one. We are going to provide a plugin for Qt Creator that does nothing. The purpose behind this "do nothing" plugin is to discover the basic classes in Qt Creator and to feel happy when our plugin shows up in the "plugin list" ☺





State	Name	Version	Vendor	Location
✓	BinEditor	1.2.1 (1.2.1)	Nokia Corporation	C:\Work\QtCreator\build\lib\qtcreator\plugins...
✓	Bookmarks	1.2.1 (1.2.1)	Nokia Corporation	C:\Work\QtCreator\build\lib\qtcreator\plugins...
✓	CMakeProjectManager	1.2.1 (1.2.1)	Nokia Corporation	C:\Work\QtCreator\build\lib\qtcreator\plugins...
✓	CodePaster	1.2.1 (1.2.1)	Nokia Corporation	C:\Work\QtCreator\build\lib\qtcreator\plugins...
✓	Core	1.2.1 (1.2.1)	Nokia Corporation	C:\Work\QtCreator\build\lib\qtcreator\plugins...
✓	CppEditor	1.2.1 (1.2.1)	Nokia Corporation	C:\Work\QtCreator\build\lib\qtcreator\plugins...

2.1 Create a plugin project in Qt Creator

Create a folder called `DoNothing` in `$$QT_CREATOR_ROOT/src/plugins` directory. The entire source code of the plugin will be put into this directory.

It may be possible to write and build Qt Creator plugins outside of its source tree, but we found it much easier to write plugins within the source tree. Refer to the last section in this chapter to understand how to build plugins from outside the source tree.

Lets first create the `DoNothing.pro` file with the following contents

```
TEMPLATE = lib
TARGET = DoNothing

include(../../qtcreatorplugin.pri)
DESTDIR = $$IDE_PLUGIN_PATH/VCreateLogic
PROVIDER = VCreateLogic
include(../../plugins/coreplugin/coreplugin.pri)

HEADERS += DoNothingPlugin.h
SOURCES += DoNothingPlugin.cpp

OTHER_FILES += DoNothing.pluginspec
```

The project file configures the following aspects of the plugin

1. Declares that DoNothing is a library. The output will be DoNothing.dll

2. Configures DoNothing to make use of settings defined in qtcreatorplugin.pri
3. Overrides the default destination directory to $$IDE_PLUGIN_PATH/VCreateLogic. By default the value will be to $$IDE_PLUGIN_PATH/Nokia
4. Configures DoNothing to make use of settings defined in coreplugin.pri
5. Provides information about the .h and .cpp files that make up the plugin

2.2 Marking the plugin for build

Edit the `$$QT_CREATOR_ROOT/src/plugins/plugins.pro` file and include the following lines at the end of the file and save the changes.

```
SUBDIRS += plugin_DoNothing
plugin_DoNothing.subdir = DoNothing
```

The above lines make sure that the next time we build Qt Creator, the DoNothing plugin is compiled along with the rest of Qt Creator plugins.

2.3 Implementing the plugin

So far we have only written the project file and marked our plugin for compilation. We now do the actual implementation of the plugin. All plugins implement the **`IPlugin`** interface. Let's take a look at how the DoNothing plugin implements the interface and understand it in stages.

In `$$QT_CREATOR_ROOT/src/plugins/DoNothing/DoNothingPluigin.h` enter the following code.

```
#ifndef DONOTHINGPLUGIN_H
#define DONOTHINGPLUGIN_H

#include <extensionsystem/iplugin.h>

class DoNothingPlugin : public ExtensionSystem::IPlugin
{
public:
    DoNothingPlugin();
    ~DoNothingPlugin();

    void extensionsInitialized();
    bool initialize(const QStringList & arguments, QString * errorString);
    void shutdown();
};

#endif // DONOTHINGPLUGIN_H
```

As you can see the DoNothingPlugin class implements the IPlugin interface and nothing else. Let's look at how the functions are implemented.

```
#include "DoNothingPlugin.h"
#include <QtPlugin>
#include <QStringList>

DoNothingPlugin::DoNothingPlugin()
{
    // Do nothing
}

DoNothingPlugin::~DoNothingPlugin()
{
    // Do notning
}
```

Apart from initializing local (non widget and non action) variables; the constructor and destructor don't do much else.

```
bool DoNothingPlugin::initialize(const QStringList& args, QString *errMsg)
{
    Q_UNUSED(args);
    Q_UNUSED(errMsg);

    return true;
}
```

The **initialize()** method is called when Qt Creator wants the plugin to initialize itself. This function is ideally used to initialize the internal state of the plugin and register actions/objects with Qt Creator. The function is called after all the dependencies of this plugin have been loaded.

Since our plugin really does nothing, we return true signifying that the initialization was successful. If the initialization was unsuccessful (for some wired reason); the **`errMsg`** string should be set to a human readable error message.

```
void DoNothingPlugin::extensionsInitialized()
{
    // Do nothing
}
```

The **`extensionsInitialized()`** method is called after this plugin has been initialized (ie. after **`initialize()`** method has been called). This method is called on plugins that depend on this plugin first.

```
void DoNothingPlugin::shutdown()
{
    // Do nothing
}
```

The **`shutdown()`** method is called when the plugin is about to be unloaded.

```
Q_EXPORT_PLUGIN(DoNothingPlugin)
```

Finally we export the plugin class by making use of the **`Q_EXPORT_PLUGIN()`** macro.

2.4 Writing the pluginspec file

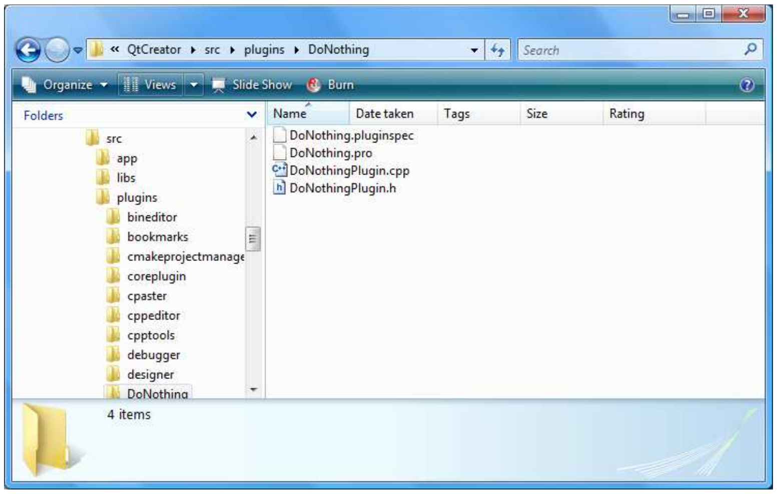
Each plugin should accompany a pluginspec file that provides some metadata about the plugin. For our plugin the pluginspec file is as follows

```
<plugin name="DoNothing" version="0.0.1" compatVersion="1.2.1">
    <vendor>VCreate Logic Pvt. Ltd.</vendor>
    <copyright>(C) 2008-2009 VCreate Logic Pvt. Ltd.</copyright>
    <license>
Do anything you want</license>
    <description>A plugin that does nothing</description>
    <url>http://www.vcreatelogic.com</url>
    <dependencyList>
        <dependency name="Core" version="1.2.1"/>
    </dependencyList>
</plugin>
```

The pluginspec file provides the following fields of information

1. Name of the plugin, which is also used as the name of the library file that provides the plugin implementation. (In our case DoNothing.dll on Windows, libDoNothing.so on Unix)
2. Version of the plugin
3. Required Qt Creator version
4. Vendor name
5. Copyright
6. License text
7. Description
8. URL of the plugin vendor
9. Dependency List – provides all the plugins that this plugin depends on. Qt Creator ensures that dependencies are loaded and initialized before this plugin.

Note: The pluginspec file should be in the same directory as the plugin's project file. Just to make things clear, the contents of the DoNothing plugin directory is as shown below

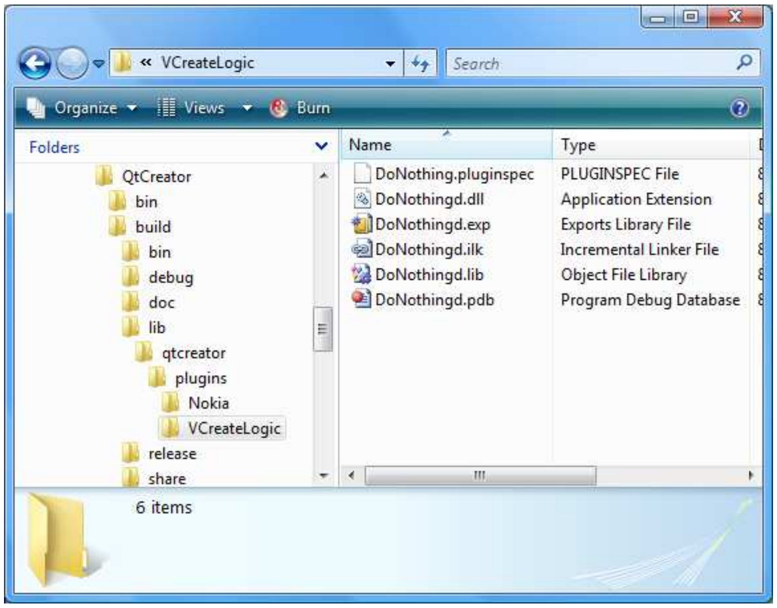


2.4 Compiling the plugin

Open a command prompt and move to the Qt Creator build directory (the same build directory you created in the previous chapter). Execute the following commands

```
qmake ..\qtcreator.pro –recursive
nmake
```

After nmake returns, you will notice a VCreateLogic folder within plugins folder whose contents are shown in the image below.

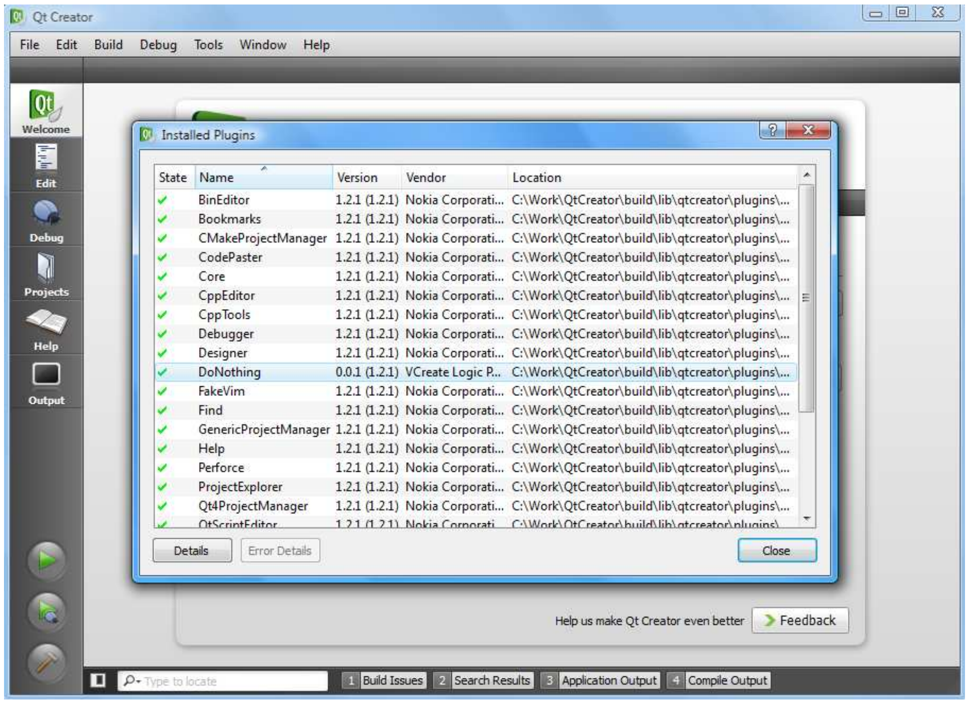


Name	Type
DoNothing.pluginspec	PLUGINSPEC File
DoNothingd.dll	Application Extension
DoNothingd.exp	Exports Library File
DoNothingd.ilk	Incremental Linker File
DoNothingd.lib	Object File Library
DoNothingd.pdb	Program Debug Database

6 items

2.5 Check out the new plugin

Launch (or relaunch) Qt Creator and notice that the "Installed Plugins" dialog box now reports that DoNothing plugin was infact loaded and initialized.



State	Name	Version	Vendor	Location
✓	BinEditor	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	Bookmarks	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	CMakeProjectManager	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	CodePaster	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	Core	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	CppEditor	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	CppTools	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	Debugger	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	Designer	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	DoNothing	0.0.1 (1.2.1)	VCreate Logic P...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	FakeVim	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	Find	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	GenericProjectManager	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	Help	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	Perforce	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	ProjectExplorer	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	Qt4ProjectManager	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\...
✓	QtScriptEditor	1.2.1 (1.2.1)	Nokia Corporati...	C:\Work\QtCreator\build\lib\qtcreator\plugins\

In the coming chapters we will learn to write more complicated plugins for Qt Creator.

2.6 Building out-of-source plugins

Thus far we have understood how to build plugins within the source tree of Qt Creator. It may not be practical for us to use the Qt Creator source tree for plugin development all the time. Suppose that you are the author of a specialized library (or application) and you want integrate your product into Qt Creator. Since you are a 3rd party developer you cannot expect to have your code in Qt Creator source tree all the time. In this section we will look at how to build plugins that are outside the Qt Creator source tree.

2.6.1 The plugin project file

The whole magic of out-of-source plugin builds lies in the project (.pro) file of your plugin. Lets the DoNothing plugin discussed in the previous section and modify (its ".pro" file) so that plugins can be built from a directory outside Qt Creator source.

The following table lists out the directory structure

Description	Directory
Qt Creator Source Code	C:\Work\QtCreator
Qt Creator Build Directory	C:\Work\QtCreator\build
DoNothing Plugin Source	C:\Work\Research\QtCreator\Plugins\DoNothing This directory currently contains • DoNothing.pluginspec • DoNothing.pro • DoNothingPlugin.cpp • DoNothingPlugin.h
Target plugin directory	C:\Work\QtCreator\build\lib\qtcreator\plugins\VCreateLogic

Let's now modify the DoNothing.pro file in C:\Work\Research\QtCreator\Plugins\DoNothing as follows.

```
QTC_SOURCE = C:/Work/QtCreator/
QTC_BUILD  = C:/Work/QtCreator/build/

TEMPLATE        = lib
TARGET          = DoNothing
IDE_SOURCE_TREE = $$QTC_SOURCE
IDE_BUILD_TREE  = $$QTC_BUILD
PROVIDER        = VCreateLogic
DESTDIR         = $$QTC_BUILD/lib/qtcreator/plugins/VCreateLogic
LIBS           += -L$$QTC_BUILD/lib/qtcreator/plugins/Nokia

include($$QTC_SOURCE/src/qtcreatorplugin.pri)
include($$QTC_SOURCE/src/plugins/coreplugin/coreplugin.pri)

HEADERS         = DoNothingPlugin.h
SOURCES         = DoNothingPlugin.cpp
OTHER_FILES     = DoNothingPlugin.pluginspec
```

The **`QTC_SOURCE`** and **`QTC_BUILD`** variables in the project file point to the source and build directories of Qt Creator. If you prefer setting these as environment variables, then use **`$$(QTC_BUILD)`** instead of **`$$QTC_BUILD`** in the project file.

The **IDE_SOURCE_TREE** and **IDE_BUILD_TREE** variables are used by qtcreatorplugin.pri to establish the include and library paths.

The **PROVIDER** and **DESTDIR** directories must be set before including qtcreatorplugin.pri. This is because the variables will be provided default values are **Nokia** and **$$IDE_BUILD_TREE/lib/qtcreator/plugins/Nokia** otherwise.

By default qtcreatorplugin.pri assumes that all the libs that a plugin may depend on are present inside the **DESTDIR**. If our **DESTDIR** is different from the default (Nokia) one, then we will need to explicitly set that. The remaining things are just the same.

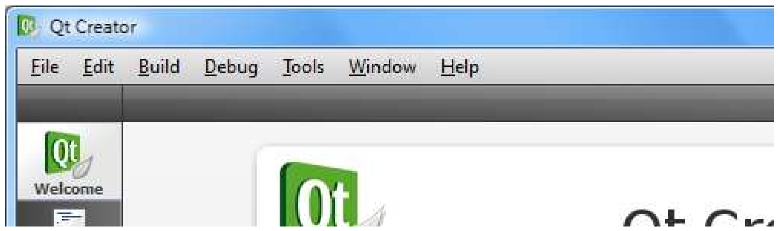
2.6.2 Compiling the plugin

Once the project file has been created, we make use of the standard qmake and make commands to compile the plugin.

```
C:\Work\Research\QtCreator\Plugins\DoNothing>qmake

C:\Work\Research\QtCreator\Plugins\DoNothing>nmake

Microsoft (R) Program Maintenance Utility Version 8.00.50727.762
Copyright (C) Microsoft Corporation.  All rights reserved.

        "C:\Program Files\Microsoft Visual Studio 8\VC\BIN\nmake.exe" -f Makefile.Debug

Microsoft (R) Program Maintenance Utility Version 8.00.50727.762
Copyright (C) Microsoft Corporation.  All rights reserved.

        copy /y "DoNothing.pluginspec"
"..\..\..\..\QtCreator\build\lib\qtcreator\plugins\VCreateLogic\DoNothing.pluginspec"
        1 file(s) copied.

........................................

        mt.exe -nologo -manifest "debug\DoNothingd.intermediate.manifest" -
outputresource:..\..\..\..\QtCreator\build\lib\qtcreator\plugins\VCreateLog
ic\DoNothingd.dll;2

C:\Work\Research\QtCreator\Plugins\DoNothing>
```

3 Adding Menu and Menu Entries

In this chapter we will understand how to add entries to existing menus in Qt Creator. We will also learn how to add new menu entries. Before moving ahead let's take a look at the menu bar in Qt Creator



The menu bar consists of the following set of default menus

- File
    - New
    - Open
    - Recent Files
- Edit
    - Advanced
- Tools
- Window
    - Panes
- Help

<u>Note: Other menu items like Build and Debug come from plugins. They are not a part of the default menu set.</u>

As Qt developers we know that the above menus are shown within a **`QMenuBar`**; and that there is a **`QMenu`** associated with each of the above menus.

3.1 Core::ActionManager

The main Qt Creator program is nothing but a plugin loader. All of the functionality provided by Qt Creator is provided by the plugins. The main plugin for Qt Creator is called "core". Without core, Qt Creator really doesn't have a personality ☺

One of the key components of the "core" is the **`ActionManager`**. **`ActionManager`** is responsible for registration of menus, menu-items and keyboard shortcuts. So if we wanted to add a new menu-item or menu, we would have to use **`ActionManager`**. The coming subsections explain this better.

To gain access to the **`ActionManager`**, the following piece of code can be used.

```
#include <coreplugin/actionmanager/actionmanager.h>
#include <coreplugin/icore.h>

Core::ActionManager* am = Core::ICore::instance()->actionManager();
```

3.2 Core::ActionContainer

`ActionContianer` represents menu or menubar in Qt Creator. Instances of **`ActionContainer`** are never created directly, instead they are accessed using **`ActionManager::createMenu()`**, **`ActionManager::createMenuBar()`** etc; but more on that later.

There is an **`ActionContainer`** associated with each of the default menus in Qt Creator. Fetching **`ActionContainer`** for a given menu can be done using the following code snippet

```
#include <coreplugin/coreconstants.h>
#include <coreplugin/actionmanager/actionmanager.h>
#include <coreplugin/icore.h>

Core::ActionManager* am = Core::ICore::instance()->actionManager();
Core::ActionContainer* ac = am->actionContainer( ID );
```

The following table lists out the ID to use for each of the menus in Qt Creator. Each of the IDs are defined as **`const char*`** static variables within the **`Core`** namespace.

Menu	ID
File	**`Core::Constants::M_FILE`**
File -> New	**`Core::Constants::M_FILE_NEW`**
File -> Open	**`Core::Constants::M_FILE_OPEN`**
File -> Recent Files	**`Core::Constants::M_FILE_RECENTFILES`**
Edit	**`Core::Constants::M_EDIT`**
Edit -> Advanced	**`Core::Constants::M_EDIT_ADVANCED`**
Tools	**`Core::Constants::M_TOOLS`**
Window	**`Core::Constants::M_WINDOW`**
Window Panes	**`Core::Constants::M_WINDOW_PANES`**
Help	**`Core::Constants::M_HELP`**

So if we want to catch hold of the “Help” menu, we can use the code snippet as follows

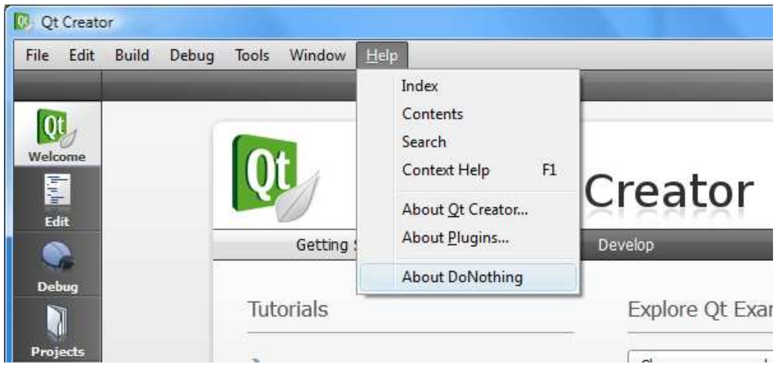
```
#include <coreplugin/coreconstants.h>
#include <coreplugin/actionmanager/actionmanager.h>
#include <coreplugin/icore.h>

Core::ActionManager* am = Core::ICore::instance()->actionManager();
Core::ActionContainer* ac = am->actionContainer( Core::Constants::M_HELP );
```

3.3 Adding menu items

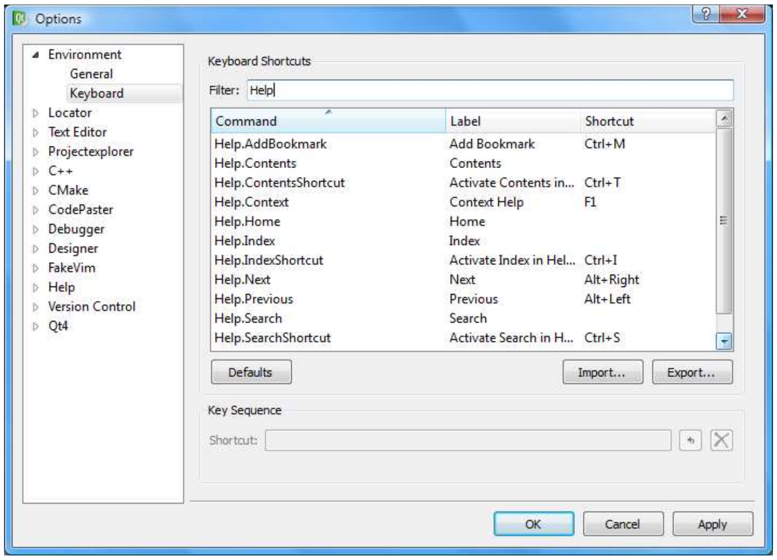
Let’s now go back to our DoNothing plugin and add a “About DoNothing” menu-item into the “Help” menu. We update the `DoNothingPlugin.cpp` file as follows

```
#include <coreplugin/coreconstants.h>
#include <coreplugin/actionmanager/actionmanager.h>
#include <coreplugin/icore.h>

bool DoNothingPlugin::initialize(const QStringList& args, QString *errMsg)
{
    Q_UNUSED(args);
    Q_UNUSED(errMsg);

    Core::ActionManager* am = Core::ICore::instance()->actionManager();
    Core::ActionContainer* ac = am->actionContainer(Core::Constants::M_HELP);

    QAction* aboutDoNothing = ac->menu()->addAction("About DoNothing");

    return true;
}
```

That's it; the "About DoNothing" action has now been added to the "Help" menu. Just compile the changes in the plugin and launch Qt Creator to notice the new menu-item.



Although adding menu-items this way works, it is not the recommended way to do so. Menu-items in Qt Creator must get listed within in the "Keyboard Shortcuts" preferences.



Filter: Help

Command	Label	Shortcut
Help.AddBookmark	Add Bookmark	Ctrl+M
Help.Contents	Contents	
Help.ContentsShortcut	Activate Contents in...	Ctrl+T
Help.Context	Context Help	F1
Help.Home	Home	
Help.Index	Index	
Help.IndexShortcut	Activate Index in Hel...	Ctrl+I
Help.Next	Next	Alt+Right
Help.Previous	Previous	Alt+Left
Help.Search	Search	
Help.SearchShortcut	Activate Search in H...	Ctrl+S

By adding menu-items the way we did here, our actions don't get listed in "Keyboard Shortcuts". For that to happen, we will have to register menu-items.

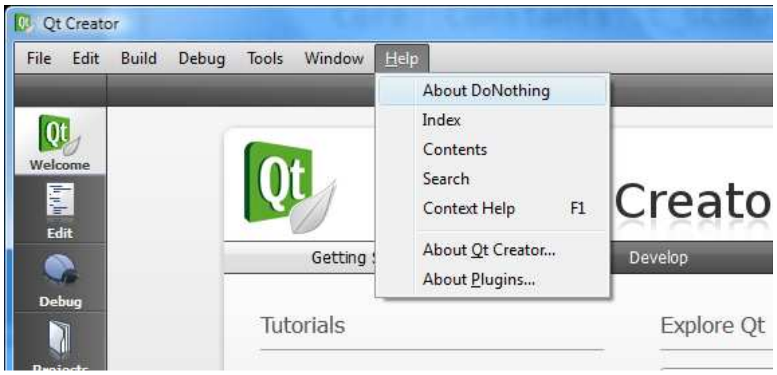
3.4 Registering menu-items.

The Core::Command class represents an action like a menu item, tool button, or shortcut. You don't create Command objects directly, instead use we use **`ActionManager::registerAction()`** to register an action and retrieve a Command. The Command object represents the user visible action and its properties.

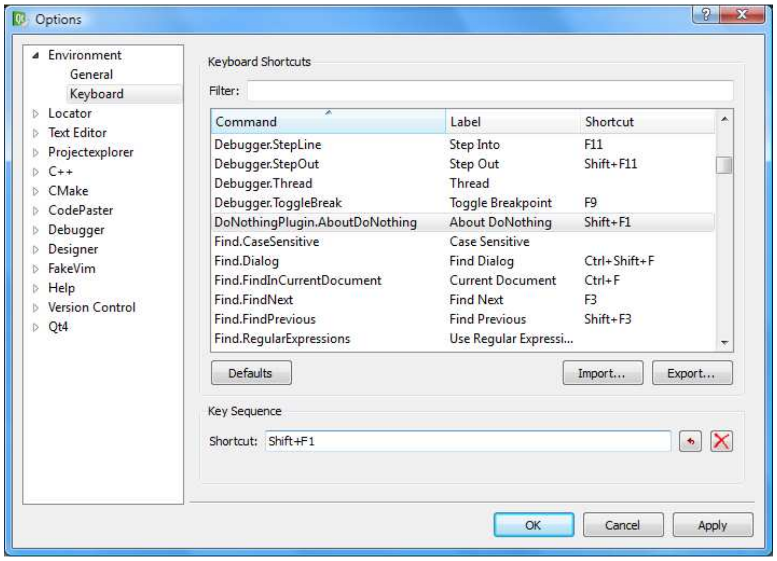
Shown below is the right way to add the “About DoNothing” menu-item from the DoNothing plugin.

```
#include <coreplugin/coreconstants.h>
#include <coreplugin/actionmanager/actionmanager.h>
#include <coreplugin/actionmanager/command.h>
#include <coreplugin/icore.h>

#include <QKeySequence>

bool DoNothingPlugin::initialize(const QStringList& args, QString *errMsg)
{
    Q_UNUSED(args);
    Q_UNUSED(errMsg);

    // Fetch the action manager
    Core::ActionManager* am = Core::ICore::instance()->actionManager();

    // Create a command for "About DoNothing".
    Core::Command* cmd = am->registerAction(new QAction(this),
                                            "DoNothingPlugin.AboutDoNothing",
                                            QList<int>() <<
                                            Core::Constants::C_GLOBAL_ID);
    cmd->action()->setText("About DoNothing");

    // Add the command to Help menu
    am->actionContainer(Core::Constants::M_HELP)->addAction(cmd);

    return true;
}
```

After compiling the changes, we can notice that the “About DoNothing” action shows up in the “Help” menu; but at the beginning.



When added this way, we will be able to find the “About DoNothing” action in the “Keyboard Shortcuts” dialog box and also associate a keyboard shortcut with it.



Command	Label	Shortcut
Debugger.StepLine	Step Into	F11
Debugger.StepOut	Step Out	Shift+F11
Debugger.Thread	Thread	
Debugger.ToggleBreak	Toggle Breakpoint	F9
DoNothingPlugin.AboutDoNothing	About DoNothing	Shift+F1
Find.CaseSensitive	Case Sensitive	
Find.Dialog	Find Dialog	Ctrl+Shift+F
Find.FindInCurrentDocument	Current Document	Ctrl+F
Find.FindNext	Find Next	F3
Find.FindPrevious	Find Previous	Shift+F3
Find.RegularExpressions	Use Regular Expressi...	

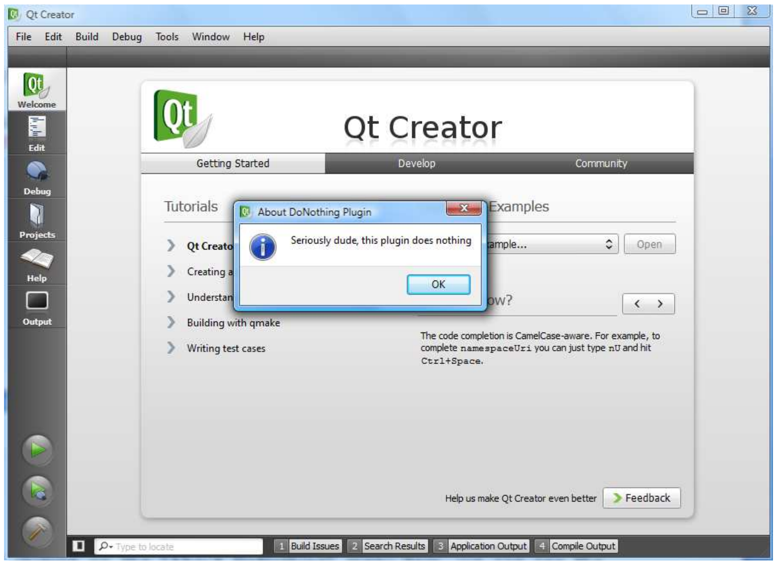
Shortcut: Shift+F1

3.5 Responding to menu-items

Since menu-items are **QAction**s, we can connect to their **triggered(bool)** or **toggled(bool)** signal and respond to trigger/toggled events. The code below shows how to do this

```
class DoNothingPlugin : public ExtensionSystem::IPlugin
{
    Q_OBJECT

private slots:
    void about();
};

bool DoNothingPlugin::initialize(const QStringList& args, QString *errMsg)
{
    ......
    Core::Command* cmd = am->registerAction(new QAction(this),
                                           "DoNothingPlugin.AboutDoNothing",
                                           QList<int>() << 0);
    ......
    connect(cmd->action(), SIGNAL(triggered(bool)), this, SLOT(about()));

    return true;
}

void DoNothingPlugin::about()
{
    QMessageBox::information(0, "About DoNothing Plugin",
        "Seriously dude, this plugin does nothing");
}
```

After compiling the changes and clicking on the “About DoNothing” menu-item, we can see the information dialog box as shown below.



Seriously dude, this plugin does nothing

The code completion is CamelCase-aware. For example, to complete `namespaceUri` you can just type `nU` and hit `Ctrl+Space`.

If you wanted the message box to have the Qt Creator main window as its parent, then you can use the following code for the about() slot.

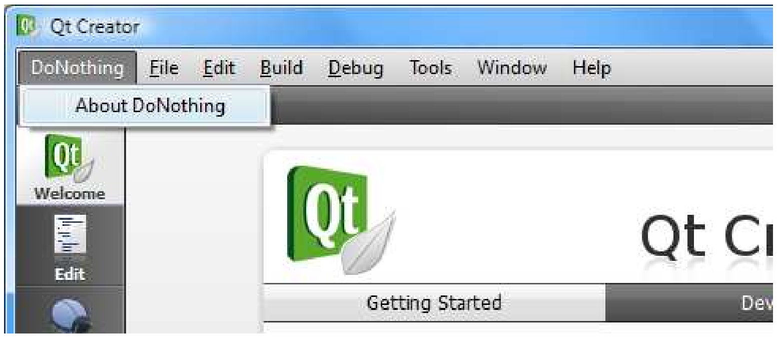
```
void DoNothingPlugin::about()
{
    QMessageBox::information(
            Core::ICore::instance()->mainWindow(),
            "About DoNothing Plugin",
            "Seriously dude, this plugin does nothing"
        );
}
```

3.6 Adding menus

The procedure for adding menus is the same. Instead of creating a **Core::Command**, we create a **Core::ActionContainer** and add it to the **MENU_BAR**. The following code snippet highlights the changes from our previous version.

```
bool DoNothingPlugin::initialize(const QStringList& args, QString *errMsg)
{
    Q_UNUSED(args);
    Q_UNUSED(errMsg);

    // Fetch the action manager
    Core::ActionManager* am = Core::ICore::instance()->actionManager();

    // Create a DoNothing menu
    Core::ActionContainer* ac
           = am->createMenu("DoNothingPlugin.DoNothingMenu");
    ac->menu()->setTitle("DoNothing");

    // Create a command for "About DoNothing".
    Core::Command* cmd = am->registerAction(new QAction(this),
                                        "DoNothingPlugin.AboutDoNothing",
                                        QList<int>() << 0);
    cmd->action()->setText("About DoNothing");

    // Add DoNothing menu to the menubar
    am->actionContainer(Core::Constants::MENU_BAR)->addMenu(ac);

    // Add the "About DoNothing" action to the DoNothing menu
    ac->addAction(cmd);

    // Connect the action
    connect(cmd->action(), SIGNAL(triggered(bool)), this, SLOT(about()));

    return true;
}
```

After recompiling the changes, you will be able to notice the DoNothing menu as shown in the screenshot below.



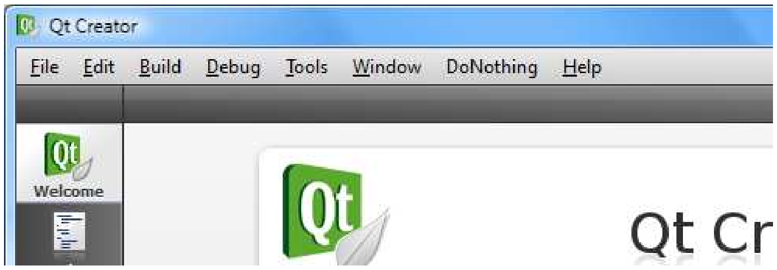
3.7 Placing menus and menu-items

It is possible to insert menus and menu-items anywhere you want. Shown below is a code snippet that inserts the "DoNothing" menu before the "Help" menu.

```
bool DoNothingPlugin::initialize(const QStringList& args, QString *errMsg)
{
    Q_UNUSED(args);
    Q_UNUSED(errMsg);

    // Fetch the action manager
    Core::ActionManager* am = Core::ICore::instance()->actionManager();

    // Create a DoNothing menu
    Core::ActionContainer* ac
            = am->createMenu("DoNothingPlugin.DoNothingMenu");
    ac->menu()->setTitle("DoNothing");

    // Create a command for "About DoNothing".
    Core::Command* cmd = am->registerAction(new QAction(this),
                                            "DoNothingPlugin.AboutDoNothing",
                                            QList<int>() << 0);
    cmd->action()->setText("About DoNothing");

    // Insert the "DoNothing" menu between "Window" and "Help".
    QMenu* windowMenu
           = am->actionContainer(Core::Constants::M_HELP)->menu();
    QMenuBar* menuBar
```

```
            = am->actionContainer(Core::Constants::MENU_BAR)->menuBar();
    menuBar->insertMenu(windowMenu->menuAction(), ac->menu());

    // Add the "About DoNothing" action to the DoNothing menu
    ac->addAction(cmd);

    // Connect the action
    connect(cmd->action(), SIGNAL(triggered(bool)), this, SLOT(about()));

    return true;
}
```

After compiling the changes, we can now notice that change in position of the “DoNothing” menu.



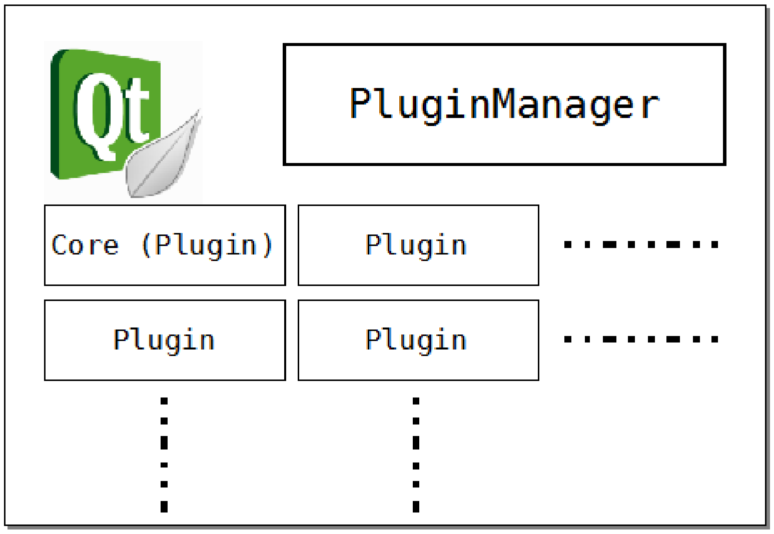
You can use a similar technique for customizing the position of menu-items.

4 Architecture of Qt Creator

Every large system has a well defined "system architecture"; which if understood well makes it easy for us to find our way in it. Qt Creator is no different. In this chapter we will understand the basic architecture of Qt Creator so that we can continue our understanding of writing plugins.

4.1 Nuts and Bolts of Qt Creator

The core of Qt Creator is basically only a "plugin loader". All functionality is implemented in plugins.



The core "personality" of Qt Creator is implemented in the **Core Plugin** (`Core::ICore`). We have already had a brush with the core plugin in the previous chapter. In the rest of this document we will refer to "core plugin" as Core.

The plugin manager (`ExtensionSystem::PluginManager`) provides simple means for plugin cooperation that allow plugins to provide hooks for other plugin's extensions.

4.2 What exactly is a plugin?

At the most fundamental level plugin is a shared library (DLL file on Windows, SO file on Linux, DYLIB file on Mac). From a developer's point of view plugin is a module that

1. Implements the `ExtensionSystem::IPlugin` interface in a class. This class will be referred to as "Plugin Class" in the rest of the document.
2. Exports the Plugin Class using the `Q_EXPORT_PLUGIN` macro
3. Provides a pluginspec file that provides some meta information about the plugin
4. Exposes one or more objects that might be of some interest to other plugins
5. Searches for the availability of one or more objects exposed by other plugins.

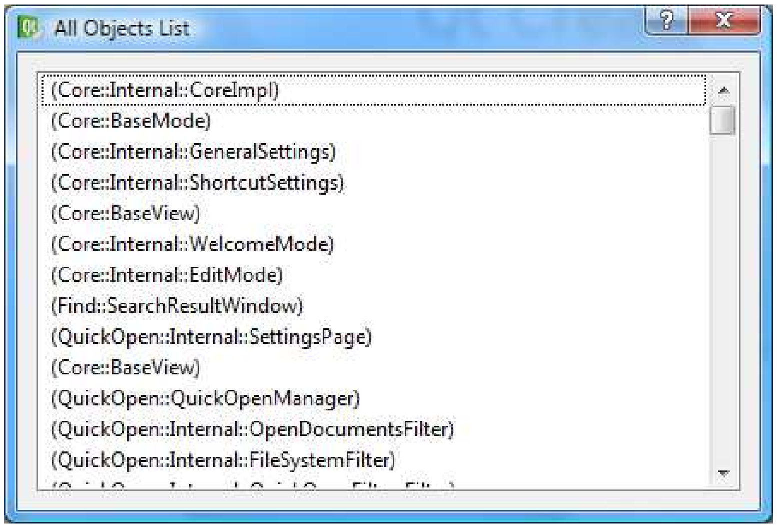
We have already had some experience with the first three aspects listed above, but we have not touched upon the last two.

4.2.1 What are exposed objects?

Exposed objects are those that land up in the **PluginManager**'s object pool. The **allObjects()** method in **PluginManager** returns the object pool as a list of **QObject** pointers. Shown below is the code that we can use to list all objects in the object-pool in a **QListWidget**.

```
#include <extensionsystem/pluginmanager.h>

ExtensionSystem::PluginManager* pm
         = ExtensionSystem::PluginManager::instance();

QList<QObject*> objects = pm->allObjects();
QListWidget* listWidget = new QListWidget;

Q_FOREACH(QObject* obj, objects)
{
    QString objInfo = QString("%1 (%2)")
                        .arg(obj->objectName())
                        .arg(obj->metaObject()->className());
    listWidget->addItem(objInfo);
}
```

When such a list widget is constructed and shown; you will see a window as shown below.



```
(Core::Internal::CoreImpl)
(Core::BaseMode)
(Core::Internal::GeneralSettings)
(Core::Internal::ShortcutSettings)
(Core::BaseView)
(Core::Internal::WelcomeMode)
(Core::Internal::EditMode)
(Find::SearchResultWindow)
(QuickOpen::Internal::SettingsPage)
(Core::BaseView)
(QuickOpen::QuickOpenManager)
(QuickOpen::Internal::OpenDocumentsFilter)
(QuickOpen::Internal::FileSystemFilter)
```

From the class names it is easy to picture the fact that each of those objects came from different plugins. I guess now we can define the term "exposed object" even better.

An exposed object is an instance of QObject (or one of its subclasses) exposed by a plugin and is available in the object-pool for other plugins to make use of.

4.2.2 How to "expose" an object form a plugin?

There are three ways to expose an object from a plugin

- **IPlugin::addAutoReleasedObject(QObject*)**
- **IPlugin::addObject(QObject*)**

- **`PluginManager::addObject(QObject*)`**

The **`IPlugin::addObject()`** and **`IPlugin::addAutoReleasedObject()`** essentially call the **`PluginManager::addObject()`** method. The **`IPlugin`** methods are only provided for convenience. It is recommended that plugins make use of the **`IPlugin`** methods for adding objects.

The only difference between **`addAutoReleasedObject()`** and **`addObject()`** is that objects added using the former method are automatically removed and deleted in the reverse order of registration when the plugin is destroyed.

At anytime plugins can make use of the **`IPlugin::removeObject(QObject*)`** method to remove its object from the object pool.

4.2.3 What objects to expose?

Plugins can expose just about any object. Normally objects that provide some sort of functionality used by other plugin(s) are exposed. Functionalities in Qt Creator are defined by means of interfaces. Listed below are some interfaces

- **`Core:: INavigationWidgetFactory`**
- **`Core::IEditor`**
- **`Core::IOptionsPage`**
- **`Core::IOutputPane`**
- **`Core::IWizard`**

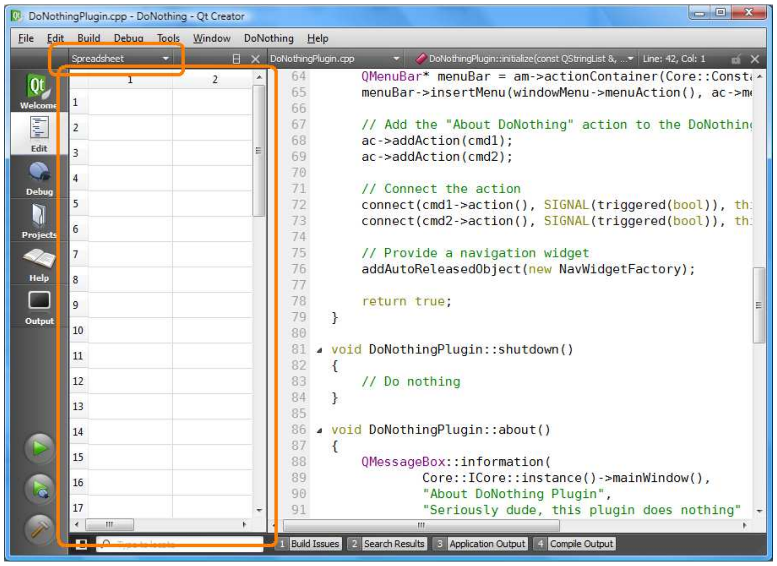
C++ developers normally assume interfaces to be classes with all its functions are public pure virtual functions. In Qt Creator interfaces are subclasses of QObject that offer one or more methods are pure virtual.

If a plugin has objects that implement an interface, then such an object has to be exposed. For example if a plugin implements the **INavigationWidgetFactory** interface in an object and exposed it, the Core will automatically use that object to show the widget provided by it as navigation widget. Take a look at the code snippet below. We provide a simple **`QTableWidget`** as navigation widget via an implementation of **`Core::INavigationWidgetFactory`**.

```
#include <coreplugin/inavigationwidgetfactory.h>

class NavWidgetFactory : public Core::INavigationWidgetFactory
{
public:
    NavWidgetFactory();
    ~NavWidgetFactory();

    Core::NavigationView createWidget();
    QString displayName();
};


#include <QTableWidget>

NavWidgetFactory::NavWidgetFactory() { }
NavWidgetFactory::~NavWidgetFactory() { }
```

```
Core::NavigationView NavWidgetFactory::createWidget()
{
    Core::NavigationView view;
    view.widget = new QTableWidget(50, 3);
    return view;
}

QString NavWidgetFactory::displayName()
{
    return "Spreadsheet";
}

bool MyPlugin::initialize(const QStringList& args, QString *errMsg)
{
    Q_UNUSED(args);
    Q_UNUSED(errMsg);

    // Provide a navigation widget factory.
    // Qt Creator's navigation widget will automatically
    // hook to our INavigationWidgetFactory implementation, which
    // is the NavWidgetFactory class, and show the QTableWidget
    // created by it in the navigation panel.
    addAutoReleasedObject(new NavWidgetFactory);

    return true;
}
```

The effect of the above code is



```
        QMenuBar* menuBar = am->actionContainer(Core::Consta
        menuBar->insertMenu(windowMenu->menuAction(), ac->me

        // Add the "About DoNothing" action to the DoNothin
        ac->addAction(cmd1);
        ac->addAction(cmd2);

        // Connect the action
        connect(cmd1->action(), SIGNAL(triggered(bool)), th
        connect(cmd2->action(), SIGNAL(triggered(bool)), th

        // Provide a navigation widget
        addAutoReleasedObject(new NavWidgetFactory);

        return true;
    }

    void DoNothingPlugin::shutdown()
    {
        // Do nothing
    }

    void DoNothingPlugin::about()
    {
        QMessageBox::information(
                Core::ICore::instance()->mainWindow(),
                "About DoNothing Plugin",
                "Seriously dude, this plugin does nothing"
```

4.2.4 Becoming aware of exposed objects

Whenever the **`PluginManager::addObject()`** is used to add an object, it (**`PluginManager`**) emits the **`objectAdded(QObject*)`** signal. This signal can be used within our applications to figure out the objects that got added.

Obviously a plugin will begin receiving the signal only after it makes a connection to it. That happens only after the plugin is initialized; which also means that the plugin will receive the **objectAdded()** signal only for objects added after the plugin was initialized.

Usually the slot that is connected to the **objectAdded()** signal will look for one or more known interfaces. Suppose that your plugin is looking for the **INavigationWidgetFactory** interface, the slot connected to **objectAdded()** will be like the one shown below.

```
void Plugin::slotObjectAdded(QObject * obj)
{
    INavigationWidgetFactory *factory
        = Aggregation::query<INavigationWidgetFactory>(obj);

    if(factory)
    {
        // use it here...
    }
}
```

4.2.4 Searching for objects

Sometimes a plugin might want to search for an object in the application that offers some functionality. We already know by now that

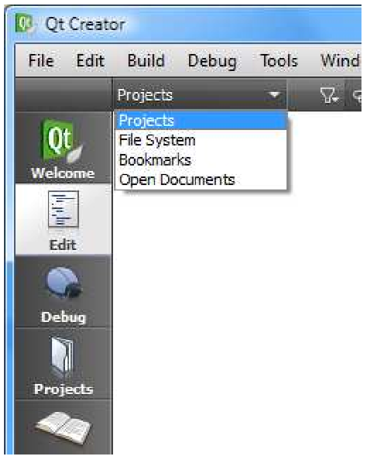
- **PluginManager::allObjects()** returns the object pool as a **QList<QObject*>**
- Connecting to **PluginManager::objectAdded()** signal helps in known objects as they get exposed

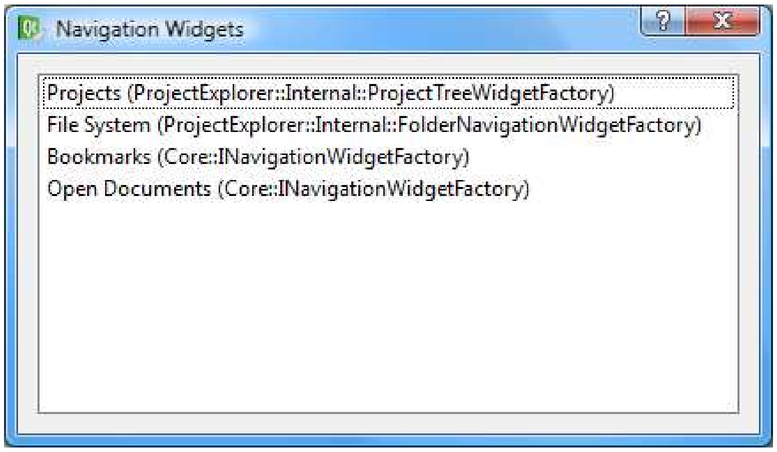
Using both of the above mentioned methods you can look for objects. Let's now understand yet another way to find objects.

Suppose that you wanted to look for objects that implement the **INavigationWidgetFactory** interface and show it in a **QListWidget**. You can make use of the **PluginManager::getObjects<T>()** method for this purpose. The following code snippet explains this

```
ExtensionSystem::PluginManager* pm
      = ExtensionSystem::PluginManager::instance();

QList<Core::INavigationWidgetFactory*> objects
    = pm->getObjects<Core::INavigationWidgetFactory>();

QListWidget* listWidget = new QListWidget();
Q_FOREACH(Core::INavigationWidgetFactory* obj, objects)
{
    QString objInfo = QString("%1 (%2)")
                        .arg(obj->displayName())
                        .arg(obj->metaObject()->className());
    listWidget->addItem(objInfo);
}
```

When the list widget is shown you will notice that the navigation widgets are shown in the same order as they are shown in the navigation combo box. Take a look at the screenshot below.





Projects (ProjectExplorer::Internal::ProjectTreeWidgetFactory)
File System (ProjectExplorer::Internal::FolderNavigationWidgetFactory)
Bookmarks (Core::INavigationWidgetFactory)
Open Documents (Core::INavigationWidgetFactory)

4.3 Aggregations

Aggregations are provided by the **Aggregation** namespace. It adds functionality for "gluing" **QObject**s of different types together, so you can "cast" between them. Using the classes and methods in this namespace you can bundle related objects into a single entity. Objects that are bundled into an aggregate can be “cast” from the aggregate into the object class type.

4.3.1 Aggregations - the old fashioned way

Suppose that you wanted an object that provided implementations of two interfaces. Normally we would go about coding the object like this

```
class Interface1
{
    ....
};
Q_DECLARE_INTERFACE("Interface1", "Interface1");

class Interface2
{
    ....
};
Q_DECLARE_INTERFACE("Interface2", "Interface2");

class Bundle : public QObject,
               public Interface1,
               public Interface2
{
    Q_OBJECT(Interface1 Interface2)

    ....
};

Bundle bundle;
```

Now we can think of **bundle** as an object that provides **Interface1** and **Interface2** implementations. We can make use of casting operators on the **bundle** object to extract **Interface1** and **Interface2**.

```
Interface1* iface1Ptr = qobject_cast<Interface1*>(&bundle);
Interface2* iface2Ptr = qobject_cast<Interface2*>(&bundle);
```

4.3.2 Aggregations - the Qt Creator way

Qt Creator's Aggregation library offers a cleaner way to define interfaces and bundle them into a single object. Instances of Aggregation::Aggregate can be created and objects can be added to it. Each of the objects added to the aggregation can implement an interface. The following code snippet shows how to create an aggregation.

```
#include <aggregation/aggregate.h>

class Interface1 : public QObject
{
    Q_OBJECT

public:
    Interface1() { }
    ~Interface1() { }
};

class Interface2 : public QObject
{
    Q_OBJECT

public:
    Interface2() { }
    ~Interface2() { }
};

Aggregation::Aggregate bundle;
bundle.add(new Interface1);
bundle.add(new Interface2);
```

The aggregation instance 'bundle' now conceptually contains implementations of two interfaces. To extract the interfaces we can make use of the following code

```
Interface1* iface1Ptr = Aggregation::query<Interface1>( &bundle );
Interface2* iface2Ptr = Aggregation::query<Interface2>( &bundle );
```

With aggregation you can also several objects of the same interface into a single bundle. For example

```
Aggregation::Aggregate bundle;
bundle.add(new Interface1);
bundle.add(new Interface2);
bundle.add(new Interface1);
bundle.add(new Interface1);

QList<Interface1*> iface1Ptrs
    = Aggregation::query_all<Interface1>( &bundle );
```

Another key advantage of Aggregation is that, you can delete any one of the objects in the bundle to delete the whole bundle. Example

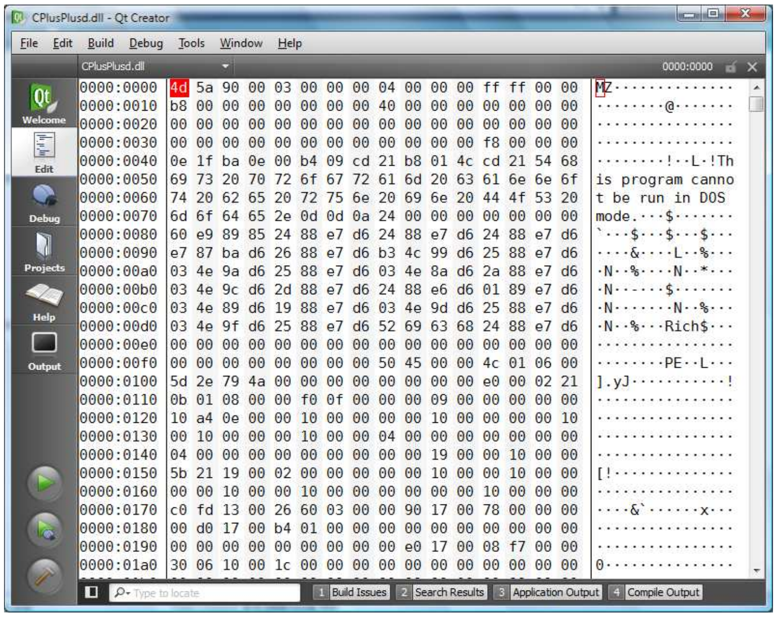
```
Aggregation::Aggregate* bundle = new Aggregation::Aggregate;
bundle->add(new Interface1);
bundle->add(new Interface2);

Interface1* iface1Ptr = Aggregation::query<Interface1>(bundle);
```

```
delete iface1Ptr;
// deletes the bundle and all objects in it
// same as delete bundle
```

The use of aggregation will become clearer when we deal with real plugin examples in the coming chapters.

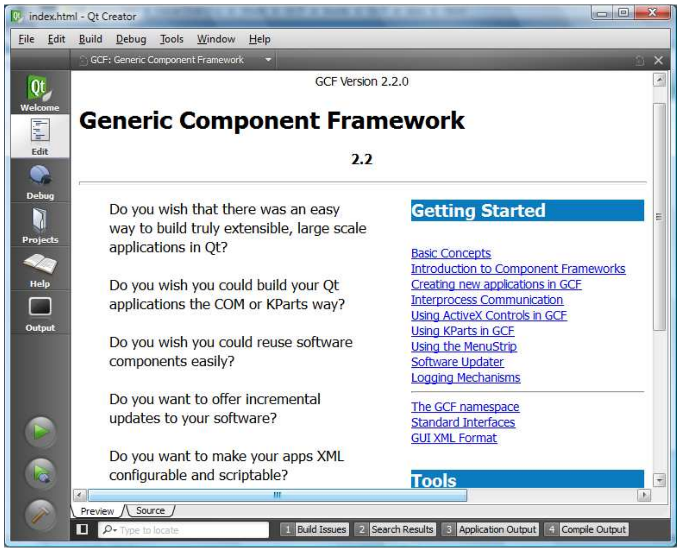
5 Adding a new Editor

At the very basic level Qt Creator is a text editor. On top of supporting editing of text files, Qt Creator also allows users to edit UI (Qt Designer) files, QRC (Resource) files, PRO/PRI (Project) files and EXE/DLL/SO (Binary) files.



```
0000:0000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00  MZ··············
0000:0010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ········@·······
0000:0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ················
0000:0030 00 00 00 00 00 00 00 00 00 00 00 00 f8 00 00 00  ················
0000:0040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68  ········!··L·!Th
0000:0050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f  is program canno
0000:0060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20  t be run in DOS 
0000:0070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00  mode.···$·······
0000:0080 60 e9 89 85 24 88 e7 d6 24 88 e7 d6 24 88 e7 d6  `···$···$···$···
0000:0090 e7 87 ba d6 26 88 e7 d6 b3 4c 99 d6 25 88 e7 d6  ····&····L··%···
0000:00a0 03 4e 9a d6 25 88 e7 d6 03 4e 8a d6 2a 88 e7 d6  ·N··%····N··*···
0000:00b0 03 4e 9c d6 2d 88 e7 d6 24 88 e6 d6 01 89 e7 d6  ·N··-···$·······
0000:00c0 03 4e 89 d6 19 88 e7 d6 03 4e 9d d6 25 88 e7 d6  ·N·······N··%···
0000:00d0 03 4e 9f d6 25 88 e7 d6 52 69 63 68 24 88 e7 d6  ·N··%···Rich$···
0000:00e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ················
0000:00f0 00 00 00 00 00 00 00 00 50 45 00 00 4c 01 06 00  ········PE··L···
0000:0100 5d 2e 79 4a 00 00 00 00 00 00 00 00 e0 00 02 21  ].yJ···········!
0000:0110 0b 01 08 00 00 f0 0f 00 00 00 09 00 00 00 00 00  ················
0000:0120 10 a4 0e 00 00 10 00 00 00 10 00 00 00 00 00 10  ················
0000:0130 00 10 00 00 00 10 00 00 04 00 00 00 00 00 00 00  ················
0000:0140 04 00 00 00 00 00 00 00 00 00 19 00 00 10 00 00  ················
0000:0150 5b 21 19 00 02 00 00 00 00 00 10 00 00 10 00 00  [!··············
0000:0160 00 00 10 00 00 10 00 00 00 00 00 10 00 00 00 00  ················
0000:0170 c0 fd 13 00 26 60 03 00 00 90 17 00 78 00 00 00  ····&`······x···
0000:0180 00 d0 17 00 b4 01 00 00 00 00 00 00 00 00 00 00  ················
0000:0190 00 00 00 00 00 00 00 00 00 e0 17 00 08 f7 00 00  ················
0000:01a0 30 06 10 00 1c 00 00 00 00 00 00 00 00 00 00 00  0···············
```

In this chapter we will understand how to provide editors for custom file formats, specifically the HTML file format. When we are done, we will be able to load HTML files from the local file system and view/edit them.



GCF Version 2.2.0

Generic Component Framework

2.2

Do you wish that there was an easy way to build truly extensible, large scale applications in Qt?

Do you wish you could build your Qt applications the COM or KParts way?

Do you wish you could reuse software components easily?

Do you want to offer incremental updates to your software?

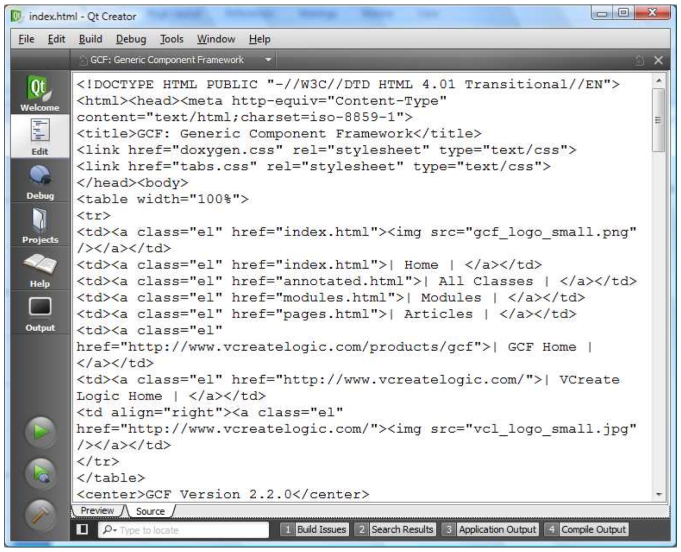
Do you want to make your apps XML configurable and scriptable?

Getting Started

Basic Concepts
Introduction to Component Frameworks
Creating new applications in GCF
Interprocess Communication
Using ActiveX Controls in GCF
Using KParts in GCF
Using the MenuStrip
Software Updater
Logging Mechanisms

The GCF namespace
Standard Interfaces
GUI XML Format

Tools



```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head><meta http-equiv="Content-Type"
content="text/html;charset=iso-8859-1">
<title>GCF: Generic Component Framework</title>
<link href="doxygen.css" rel="stylesheet" type="text/css">
<link href="tabs.css" rel="stylesheet" type="text/css">
</head><body>
<table width="100%">
<tr>
<td><a class="el" href="index.html"><img src="gcf_logo_small.png"
/></a></td>
<td><a class="el" href="index.html">| Home | </a></td>
<td><a class="el" href="annotated.html">| All Classes | </a></td>
<td><a class="el" href="modules.html">| Modules | </a></td>
<td><a class="el" href="pages.html">| Articles | </a></td>
<td><a class="el"
href="http://www.vcreatelogic.com/products/gcf">| GCF Home |
</a></td>
<td><a class="el" href="http://www.vcreatelogic.com/">| VCreate
Logic Home | </a></td>
<td align="right"><a class="el"
href="http://www.vcreatelogic.com/"><img src="vcl_logo_small.jpg"
/></a></td>
</tr>
</table>
<center>GCF Version 2.2.0</center>
```

5.1 Core Classes and Interfaces

To support a new editor type, we need to

- Implement a plugin (**`Core::IPlugin`** implementation) class that exposes an "editor factory". Chapter 2 in this document provides a detailed description on creating plugins by implementing the **`Core::IPlugin`** interface.

- Implement the editor factory, **`Core::IEditorFactory`**, interface. This interface implementation provides methods to help create instances of "editor" object for a specified mime-type.
- Implement the editor, **`Core::IEditor`**, interface. This interface implementation provides a widget that helps edit a file type (for example: HTML, ODF etc). Editors must provide access to the "file" that it is currently being shown or edited.
- Implement the file, **`Core::IFile`**, interface to help customize the loading and saving of data into disk files.

In the following subsections, we will take a look at each of the above mentioned core interfaces.

5.1.1 The Core::IFile interface

This interface abstracts file operations from the user-interface point of view. It provides virtual methods to load and save files, given a file name. It also helps in understanding the mime-type of file and value of certain flags like "modified" and "read-only". The **`Core::IFile`** interface is declared as follows in src/plugins/coreplugin/ifile.h

```
namespace Core {

class IFile : public QObject
{
    Q_OBJECT

public:
    enum ReloadBehavior { AskForReload, ReloadAll, ReloadPermissions, ReloadNone };

    IFile(QObject *parent = 0) : QObject(parent) {}
    virtual ~IFile() {}

    virtual bool save(const QString &fileName = QString()) = 0;
    virtual QString fileName() const = 0;

    virtual QString defaultPath() const = 0;
    virtual QString suggestedFileName() const = 0;
    virtual QString mimeType() const = 0;

    virtual bool isModified() const = 0;
    virtual bool isReadOnly() const = 0;
    virtual bool isSaveAsAllowed() const = 0;

    virtual void modified(ReloadBehavior *behavior) = 0;

    virtual void checkPermissions() {}

signals:
    void changed();
};

} // namespace Core
```

You may be wondering: "Why go for another interface called **`IFile`** when we already have a class called **`QFile`** that provides the exact same functionality?" Good point. The responses to this question are listed below

- **`IFile`** has to take care of loading contents of a filename into an editor (**`Core::IEditor`** interface discussed next). **`QFile`** on the other hand simply loads contents into a **`QByteArray`**.

- **IFile** has to emit the **modified()** signal when the user edits the contents of the file in the editor, but the actual disk-file contents have not been modified. **QFile** emits the **bytesWritten()** signal only when the disk-file contents have been modified.
- **IFile** has to handle how a modified file, on the disk, is reloaded. **QFile** on the other-hand doesn't need to handle this.

We will learn more about implementing the **Core::IFile** interface in a future section.

5.1.2 The Core::IEditor interface

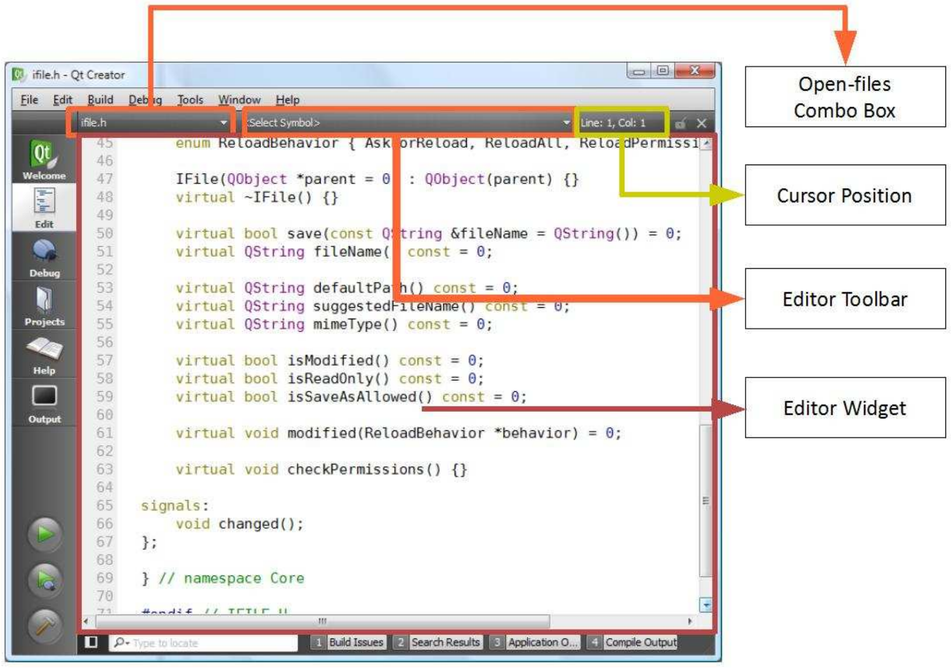
Implementations of the **Core::IEditor** interface provide editors for different types of files. It is declared as follows in src/plugins/coreplugin/editormanager/ieditor.h.

```
namespace Core {

class IContext : public QObject
{
    Q_OBJECT

public:
    IContext(QObject *parent = 0) : QObject(parent) {}
    virtual ~IContext() {}

    virtual QList<int> context() const = 0;
    virtual QWidget *widget() = 0;
    virtual QString contextHelpId() const { return QString(); }
};

class IEditor : public IContext
{
    Q_OBJECT

public:
    IEditor(QObject *parent = 0) : IContext(parent) {}
    virtual ~IEditor() {}

    virtual bool createNew(const QString &contents = QString()) = 0;
    virtual bool open(const QString &fileName = QString()) = 0;
    virtual IFile *file() = 0;
    virtual const char *kind() const = 0;
    virtual QString displayName() const = 0;
    virtual void setDisplayName(const QString &title) = 0;

    virtual bool duplicateSupported() const = 0;
    virtual IEditor *duplicate(QWidget *parent) = 0;

    virtual QByteArray saveState() const = 0;
    virtual bool restoreState(const QByteArray &state) = 0;

    virtual int currentLine() const { return 0; }
    virtual int currentColumn() const { return 0; }

    virtual bool isTemporary() const = 0;
    virtual QToolBar *toolBar() = 0;

signals:
    void changed();
};

} // namespace Core
```

The **`Core::IEditor`** interface primary provides access to

- An ***editor widget*** (**`Core::IEditor::widget()`** method) that Qt Creator can use to display contents of the file being edited.
- The file (**`Core::IEditor::file()`** method), which is a **`Core::IFile`** implementation, that Qt Creator can use to trigger the loading and saving of data from disk-files.
- An optional custom ***toolbar*** that Qt Creator can show whenever the editor becomes active.
- The ***current position*** of the edit-cursor within the file (**`Core::IEditor::currentLine()`** and **`Core::IEditor::currentColumn()`**)
- The name that needs to be displayed in the ***open-files combo box***.

Take a look at the following screenshot to get a better understanding.



```
    enum ReloadBehavior { AskForReload, ReloadAll, ReloadPermissi

    IFile(QObject *parent = 0) : QObject(parent) {}
    virtual ~IFile() {}

    virtual bool save(const QString &fileName = QString()) = 0;
    virtual QString fileName() const = 0;

    virtual QString defaultPath() const = 0;
    virtual QString suggestedFileName() const = 0;
    virtual QString mimeType() const = 0;

    virtual bool isModified() const = 0;
    virtual bool isReadOnly() const = 0;
    virtual bool isSaveAsAllowed() const = 0;

    virtual void modified(ReloadBehavior *behavior) = 0;

    virtual void checkPermissions() {}

signals:
    void changed();
};

} // namespace Core
```

We will understand more about implementing the **`Core::IEditor`** interface in a future section.

5.1.3 The Core::IEditorFactory interface

Implementations of **`Core::IEditorFactory`** interface provide methods to create instances of **`Core::IEditor`** for a supported mime-type. It is declared as follows in src/plugins/coreplugin/editormanager/ieditorfactory.h.

```
namespace Core {
```

```
class IFileFactory : public QObject
{
    Q_OBJECT
public:
    IFileFactory(QObject *parent = 0) : QObject(parent) {}
    virtual ~IFileFactory() {}

    virtual QStringList mimeTypes() const = 0;

    virtual QString kind() const = 0;
    virtual Core::IFile *open(const QString &fileName) = 0;
};

class IEditorFactory : public Core::IFileFactory
{
    Q_OBJECT
public:
    IEditorFactory(QObject *parent = 0) : IFileFactory(parent) {}
    virtual ~IEditorFactory() {}

    virtual IEditor *createEditor(QWidget *parent) = 0;
};

} // namespace Core
```

The **`IEditorFactory::mimeType()`** method should be implemented to return the mime-type supported by the editor for which the factory is implemented. The **`IEditorFactory::createEditor()`** method should be implemented to actually create a concrete editor and return the same.

5.1.4 The Core::MimeDatabase class

The **`Core::MimeDatabase`** class keeps track of all the mime-types supported by Qt Creator. It also helps figure out the mime-type of a given file. Take the following code for example:

```
#include <coreplugin/mimedatabase.h>

Core::ICore* core = Core::ICore::instance();
Core::MimeDatabase* mdb = core->mimeDatabase();

Core::MimeType type1 = mdb->findByFile( QFileInfo("C:/Temp/sample.html") );
qDebug("File Type for sample.html = %s", qPrintable(type1.type()));

Core::MimeType type2 = mdb->findByFile( QFileInfo("C:/Temp/TextEdit/Main.cpp") );
qDebug("File Type for Main.cpp = %s", qPrintable(type2.type()));

Core::MimeType type3 = mdb->findByFile( QFileInfo("C:/Temp/TextEdit/TextEdit.pro") );
qDebug("File Type for TextEdit.pro = %s", qPrintable(type3.type()));
```

When the above code is compiled and executed, we get the following as output.

```
File Type for sample.html = text/plain
File Type for Main.cpp = text/x-c++src
File Type for TextEdit.pro = text/plain
```

The **`Core::MimeDatabase`** uses filename suffix, glob patterns and "magic" matchers to figure out the mime-type of a given filename. At this point however, lets not dig into how the mime-database manages to figure out the mime-type from a filename; it is enough if we know that mime-type discovery is possible.

The **`Core::IEditorFactory`** interface, as described in the previous section, provides an editor (**`Core::IEditor`** implementation) for a specific mime-type. The following points help us understand how Qt Creator manages to pick the appropriate **`Core::IEditorFactory`** for a given filename.

1. User selects File -> Open and chooses a file to open
2. Qt Creator uses **`Core::MimeDatabase`** to figure out the mime-type of the selected file
3. Qt Creator runs through all **`Core::IEditorFactory`** implementations and picks the editor-factory that supports the mime-type evaluated in step 2
4. Qt Creator asks the selected editor factory to create an editor (Core::IEditor implementation)
5. The widget returned by **`Core::IEditor::widget()`** is then shown in the workspace area of the main-window
6. The **`Core::IEditor::open()`** method is then called to open the file selected in step 1.

5.1.5 Adding a new mime-type

If we wanted to support a new editor type, then we need to register the mime-type supported by the new editor with the **`Core::MimeDatabase`**. Several mechanisms can be used to register a new mime-type. In this chapter we will learn the simplest way to register a new mime-type from an XML file.

Suppose that we wanted to register the **text/html** mime-type and associate it with ***.html** filenames. We create an XML file as and save it as **text-html-mimetype.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<mime-info xmlns='http://www.freedesktop.org/standards/shared-mime-info'>
    <mime-type type="text/html">
        <sub-class-of type="text/plain"/>
        <comment>HTML File</comment>
        <glob pattern="*.html"/>
    </mime-type>
</mime-info>
```

We then register the mime-type described in the XML file (above) using the **`Core::MimeDatabase::addMimeTypes()`** method. Take a look at the code snippet below

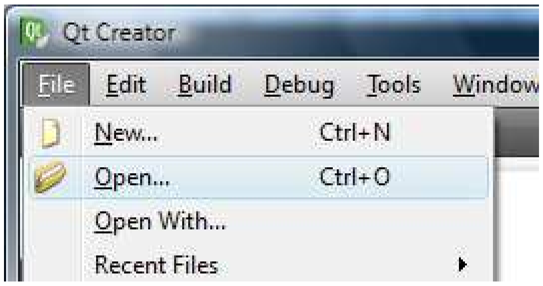
```
Core::ICore* core = Core::ICore::instance();
Core::MimeDatabase* mdb = core->mimeDatabase();

QString errMsg;
bool success = mdb->addMimeTypes("text-html-mimetype.xml", errMsg);
```

Once registered, Qt Creator will begin to map all ***.html** filenames to **text/plain** mime-type.

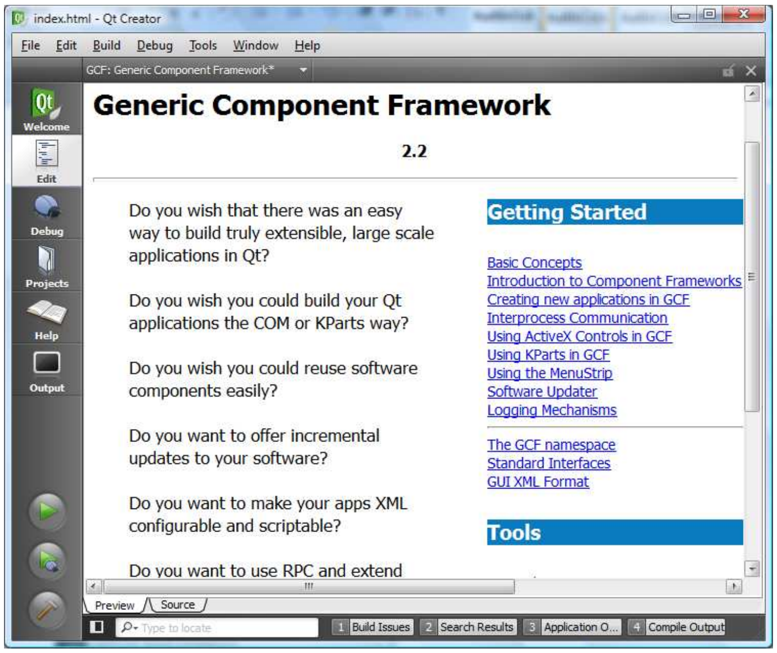
5.2 Providing a HTML Editor in Qt Creator

Let's implement a plugin for Qt Creator that provides support for viewing and editing HTML files. The following screenshots show the expected results.

The user select a HTML file to open using the standard File -> Open menuitem.



Upon selecting a HTML file, Qt Creator will show a custom editor as shown below.



Generic Component Framework

2.2

Do you wish that there was an easy way to build truly extensible, large scale applications in Qt?

Do you wish you could build your Qt applications the COM or KParts way?

Do you wish you could reuse software components easily?

Do you want to offer incremental updates to your software?

Do you want to make your apps XML configurable and scriptable?

Do you want to use RPC and extend

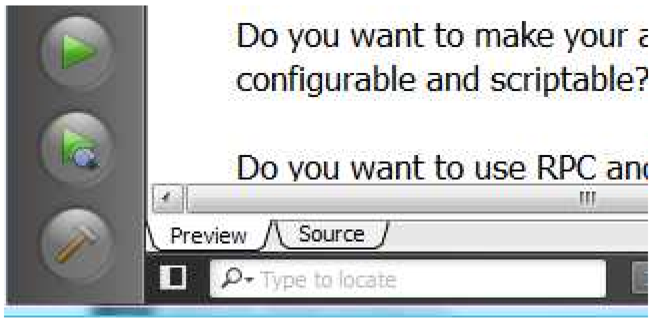
Getting Started

Basic Concepts
Introduction to Component Frameworks
Creating new applications in GCF
Interprocess Communication
Using ActiveX Controls in GCF
Using KParts in GCF
Using the MenuStrip
Software Updater
Logging Mechanisms

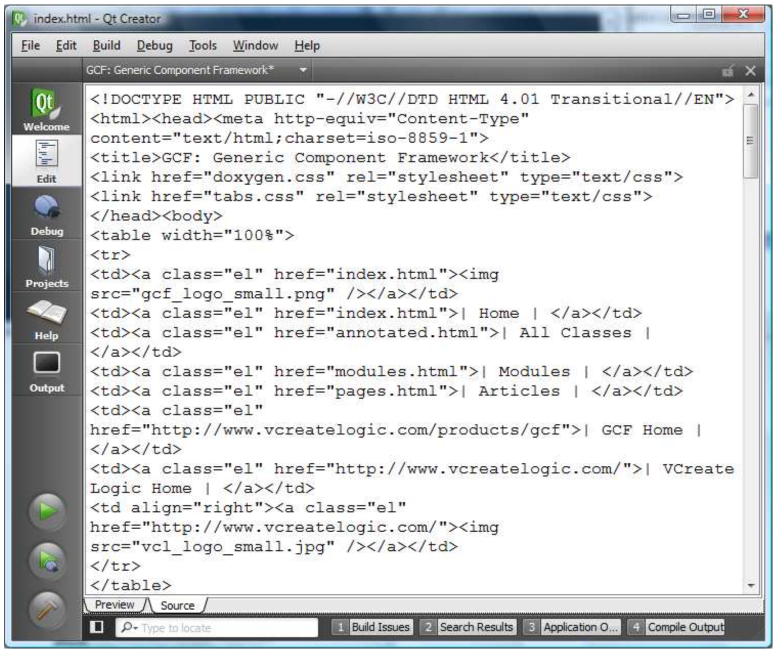
The GCF namespace
Standard Interfaces
GUI XML Format

Tools

Notice that the editor has two tabs called “Preview” and “Source” at the bottom edge.


Do you want to make your a
configurable and scriptable?
Do you want to use RPC and
Preview
Source
Type to locate

Clicking on the “Source” tab shows the HTML code in a **`QPlainTextEdit`** widget.



```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head><meta http-equiv="Content-Type"
content="text/html;charset=iso-8859-1">
<title>GCF: Generic Component Framework</title>
<link href="doxygen.css" rel="stylesheet" type="text/css">
<link href="tabs.css" rel="stylesheet" type="text/css">
</head><body>
<table width="100%">
<tr>
<td><a class="el" href="index.html"><img
src="gcf_logo_small.png" /></a></td>
<td><a class="el" href="index.html">| Home | </a></td>
<td><a class="el" href="annotated.html">| All Classes |
</a></td>
<td><a class="el" href="modules.html">| Modules | </a></td>
<td><a class="el" href="pages.html">| Articles | </a></td>
<td><a class="el"
href="http://www.vcreatelogic.com/products/gcf">| GCF Home |
</a></td>
<td><a class="el" href="http://www.vcreatelogic.com/">| VCreate
Logic Home | </a></td>
<td align="right"><a class="el"
href="http://www.vcreatelogic.com/"><img
src="vcl_logo_small.jpg" /></a></td>
</tr>
</table>
```

Users can edit the HTML in this tab, and when he moves back to the "preview" tab the changes are reflected. Users can continue to make use of File -> Save menu item to save changes to the HTML file.

For achieving the above we implement the following classes

Class	Interface / Baseclass	Description
`HtmlEditorWidget`	**`QTabWidget`**	Provides a tab widget with two tabs, one for showing the HTML preview and other for showing the HTML code.
`HtmlFile`	**`Core::IFile`**	Implements the **`IFile`** interface for the file shown in **`HtmlEditorWidget`**.
`HtmlEditor`	**`Core::IEditor`**	Implements the **`IEditor`** interface for managing the **`HtmlEditorWidget`** and hooking up an **`IFile`** with it.
`HtmlEditorFactory`	**`Core::IEditorFactory`**	Implements the **`IEditorFactory`** interface for creating instances of **`IEditor`** for the "text/html" mime-type.
`HtmlEditorPlugin`	**`Core::IPlugin`**	Implements the **`IPlugin`** interface for hooking all of the above into Qt Creator.

For every html file there is an instance each of **`HtmlEditorWidget`**, **`HtmlFile`** and **`HtmlEditor`** handling the loading, editing and saving functionality. In our implementation we provide mechanisms to help establish association between instances of the above classes that handle the same file.

5.2.1 Implementing the HTML Editor Widget

By default, Qt Creator uses a plain-text editor widget to display the contents of the HTML file being edited. We would like to offer a tabbed widget as editor. One of the tabs shows the HTML preview, the other shows the HTML code. The class **`HtmlEditorWidget`** is our editor; and it is declared as follows.

```
struct HtmlEditorWidgetData;
class HtmlEditorWidget : public QTabWidget
{
    Q_OBJECT

public:
    HtmlEditorWidget(QWidget* parent = 0);
    ~HtmlEditorWidget();

    void setContent(const QByteArray& ba, const QString& path=QString());
    QByteArray content() const;

    QString title() const;

protected slots:
    void slotCurrentTabChanged(int tab);
    void slotContentModified();

signals:
    void contentModified();
    void titleChanged(const QString&);

private:
    HtmlEditorWidgetData* d;
};
```

The constructor basically creates two different views from **`QWebView`** and **`QPlainTextEdit`** and adds them as tabs at the bottom position of the tab widget. We will learn about the signal-slot connections later.

```
HtmlEditorWidget::HtmlEditorWidget(QWidget* parent)
                 :QTabWidget(parent)
{
    d = new HtmlEditorWidgetData;

    d->webView = new QWebView;
    d->textEdit = new QPlainTextEdit;

    addTab(d->webView, "Preview");
    addTab(d->textEdit, "Source");
    setTabPosition(QTabWidget::South);
    setTabShape(QTabWidget::Triangular);

    d->textEdit->setFont( QFont("Courier", 12) );

    connect(this, SIGNAL(currentChanged(int)),
            this, SLOT(slotCurrentTabChanged(int)));
    connect(d->textEdit, SIGNAL(textChanged()),
            this, SLOT(slotContentModified()));
    connect(d->webView, SIGNAL(titleChanged(QString)),
            this, SIGNAL(titleChanged(QString)));

    d->modified = false;
}
```

The destructor does nothing other than deleting the private ‘**d**’ object.

```
HtmlEditorWidget::~HtmlEditorWidget()
{
    delete d;
}
```

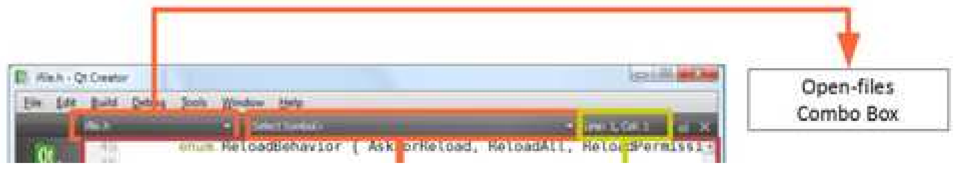
The **`setContent`** method simply sets the contents of html file to **`webView`** and **`textEdit`**.

```
void HtmlEditorWidget::setContent(const QByteArray& ba, const QString& path)
{
    if(path.isEmpty())
        d->webView->setHtml(ba);
    else
        d->webView->setHtml(ba, "file:///" + path);
    d->textEdit->setPlainText(ba);
    d->modified = false;
    d->path = path;
}
```

The **`content`** method returns the contents of the **`textEdit`**.

```
QByteArray HtmlEditorWidget::content() const
{
    QString htmlText = d->textEdit->toPlainText();
    return htmlText.toAscii();
}
```

The **`title()`** method returns the title from the webView. The string returned from this method will be used in the open-file combo box.



```
QString HtmlEditorWidget::title() const
{
    return d->webView->title();
}
```

The following connection made in the constructor of **`HtmlEditorWidget`** makes sure that when the user moves from “source” to “preview” tab, the HTML content viewed in the preview tab is updated.

```
connect(this, SIGNAL(currentChanged(int)),
            this, SLOT(slotCurrentTabChanged(int)));
```

```
void HtmlEditorWidget::slotCurrentTabChanged(int tab)
{
    if(tab == 0 && d->modified)
        setContent( content(), d->path );
}
```

Following connection makes sure that **`setContentModified()`** slot is called whenever user edits the html source. The slot **`setContentModified()`** simply sets modified to true and emits the signal "**`contentModified`**". We will know the usability of this signal later in the section while understanding **`HtmlFile`** class.

```
connect(d->textEdit, SIGNAL(textChanged()),
            this, SLOT(slotContentModified()));

void HtmlEditorWidget::slotContentModified()
{
    d->modified = true;
    emit contentModified();
}
```

Following connection simply emits **`titleChanged()`** signal on title change of **`webView`**. We will know more about this signal later.

```
connect(d->webView, SIGNAL(titleChanged(QString)),
            this, SIGNAL(titleChanged(QString)));
```

5.2.2 Implementing the Core::IFile interface

We implement the **`Core::IFile`** interface in the **`HtmlFile`** class. Apart from implementing the pure virtual functions from **`IFile`** (introduced in section 5.1.1); the **`HtmlFile`** class has methods to get/set the modified flag which indicates the modification status of the file contents.

```
struct HtmlFileData;
class HtmlFile : public Core::IFile
{
    Q_OBJECT

public:
    HtmlFile(HtmlEditor* editor, HtmlEditorWidget* editorWidget);
    ~HtmlFile();

    void setModified(bool val=true);

    // Declare all the virtual functions from IFile here..

protected slots:
    void modified() { setModified(true); }

private:
    HtmlFileData* d;
};

struct HtmlFileData
{
    HtmlFileData()
```

```
        : mimeType(HtmlEditorConstants::C_HTMLEDITOR_MIMETYPE),
          editorWidget(0), editor(0), modified(false) { }

    const QString mimeType;
    HtmlEditorWidget* editorWidget;
    HtmlEditor* editor;
    QString fileName;
    bool modified;
};
```

In the constructor implementation is simple. It establishes the association between an instance of **`HtmlFile`** with the corresponding **`HtmlEditor`** and the **`HtmlEditorWidget`** instances.

```
HtmlFile::HtmlFile(HtmlEditor* editor, HtmlEditorWidget* editorWidget)
         : Core::IFile(editor)
{
    d = new HtmlFileData;
    d->editor = editor;
    d->editorWidget = editorWidget;
}
```

The destructor does nothing other than deleting the private '**d**' object.

```
HtmlFile::~HtmlFile()
{
    delete d;
}
```

The **`setModified()`** function stores the modification flag and emits the **`changed()`** signal.

```
void HtmlFile::setModified(bool val)
{
    if(d->modified == val)
        return;

    d->modified = val;
    emit changed();
}

bool HtmlFile::isModified() const
{
    return d->modified;
}
```

Returns the mime-type handled by this class.

```
QString HtmlFile::mimeType() const
{
    return d->mimeType;
}
```

The save method is called when file->save action (Ctrl+s) is triggered. This saves the contents of **`HtmlEditorWidget`** (the contents shown by plain text editor) in the file as shown below. The modified flag is set to false after the contents are saved into the file.

```
bool HtmlFile::save(const QString &fileName)
{
    QFile file(fileName);

    if(file.open(QFile::WriteOnly))
    {
        d->fileName = fileName;

        QByteArray content = d->editorWidget->content();
        file.write(content);

        setModified(false);

        return true;
    }

    return false;
}
```

The **`open`** method is called when file->open action is triggered. This opens the file and calls the **`setContent()`** method of **`HtmlEditorWidget`**. The display name is set to the title of the html file.

```
bool HtmlFile::open(const QString &fileName)
{
    QFile file(fileName);

    if(file.open(QFile::ReadOnly))
    {
        d->fileName = fileName;

        QString path = QFileInfo(fileName).absolutePath();
        d->editorWidget->setContent(file.readAll(), path);

        d->editor->setDisplayName(d->editorWidget->title());
        return true;
    }

    return false;
}
```

The following methods implement the “filename” property.

```
void HtmlFile::setFilename(const QString& filename)
{
    d->fileName = filename;
}

QString HtmlFile::fileName() const
{
    return d->fileName;
}
```

We implement the **`defaultPath()`**, **`suggestedFileName()`**, **`fileFilter()`** and **`fileExtension()`** methods to do nothing at the moment.

```
QString HtmlFile::defaultPath() const
{
    return QString();
}

QString HtmlFile::suggestedFileName() const
{
    return QString();
}

QString HtmlFile::fileFilter() const
{
    return QString();
}

QString HtmlFile::fileExtension() const
{
    return QString();
}
```

Since we want to edit the file, we return false in **`isReadOnly()`** method and true from **`isSaveAsAllowed()`** method.

```
bool HtmlFile::isReadOnly() const
{
    return false;
}

bool HtmlFile::isSaveAsAllowed() const
{
    return true;
}
```

The **`modified()`** function has to be implemented to customize the way in which the Html editor should handle reloading of a modified file. This function is called if a html-file was modified outside of Qt Creator, while it is being edited. For now we do nothing.

```
void HtmlFile::modified(ReloadBehavior* behavior)
{
    Q_UNUSED(behavior);
}
```

5.2.3 Implementing the Core::IEditor interface

The **`HtmlEditor`** class implements **`IEditor`** interface to provide an editor widget for html (*.html) files and associate a **`HtmlFile`** instance with it.

```
#include <coreplugin/editormanager/ieditor.h>

struct HtmlEditorData;
class HtmlEditor : public Core::IEditor
{
    Q_OBJECT

public:
    HtmlEditor(HtmlEditorWidget* editorWidget);
    ~HtmlEditor();

    bool createNew(const QString& /*contents*/ = QString());
    QString displayName() const;
    IEditor* duplicate(QWidget* /*parent*/);
    bool duplicateSupported() const;
    Core::IFile* file();
    bool isTemporary() const;
    const char* kind() const;
    bool open(const QString& fileName = QString()) ;
    bool restoreState(const QByteArray& /*state*/);
    QByteArray saveState() const;
    void setDisplayName(const QString &title);
    QToolBar* toolBar();

    // From Core::IContext
    QWidget* widget();
    QList<int> context() const;

protected slots:
    void slotTitleChanged(const QString& title)
    { setDisplayName(title); }

private:
    HtmlEditorData* d;
};
```

HtmlEditorData holds object of **HtmlEditorWidget** and **HtmlFile**. **displayName** is used as visual description of the document.

```
struct HtmlEditorData
{
    HtmlEditorData() : editorWidget(0), file(0) { }

    HtmlEditorWidget* editorWidget;
    QString displayName;
    HtmlFile* file;
    QList<int> context;
};
```

The constructor creates initializes itself on an **HtmlEditorWidget**. It creates an **HtmlFile** instance so that its association with **HtmlEditor** and widget is set.

```
HtmlEditor::HtmlEditor(HtmlEditorWidget* editorWidget)
           : Core::IEditor(editorWidget)
{
    d = new HtmlEditorData;
    d->editorWidget = editorWidget;
    d->file = new HtmlFile(this, editorWidget);

    Core::UniqueIDManager* uidm = Core::UniqueIDManager::instance();
    d->context << uidm->uniqueIdentifier(HtmlEditorConstants::C_HTMLEDITOR);

    connect(d->editorWidget, SIGNAL(contentModified()),
            d->file, SLOT(modified()));
    connect(d->editorWidget, SIGNAL(titleChanged(QString)),
            this, SLOT(slotTitleChanged(QString)));
    connect(d->editorWidget, SIGNAL(contentModified()),
            this, SIGNAL(changed()));
}
```

The destructor does nothing other than deleting the private '**d**' object.

```
HtmlEditor::~HtmlEditor()
{
    delete d;
}
```

The following functions are self explanatory.

```
QWidget* HtmlEditor::widget()
{
    return d->editorWidget;
}

QList<int> HtmlEditor::context() const
{
    return d->context;
}

Core::IFile* HtmlEditor::file()
{
    return d->file;
}
```

The **`createNew()`** method is implemented to reset the contents of the **`HtmlEditorWidget`** and **`HtmlFile`** objects. For now we ignore the contents parameter.

```
bool HtmlEditor::createNew(const QString& contents)
{
    Q_UNUSED(contents);

    d->editorWidget->setContent(QByteArray());
    d->file->setFilename(QString());

    return true;
}
```

The **`open()`** method causes the **`HtmlFile`** to open a given filename. It is assumed that the filename is a HTML filename.

```
bool HtmlEditor::open(const QString &fileName)
{
    return d->file->open(fileName);
}
```

The following method returns the '**`kind`**' of the editor.

```
namespace HtmlEditorConstants
```

```
{
    const char* const C_HTMLEDITOR_MIMETYPE = "text/html";
    const char* const C_HTMLEDITOR          = "HTML Editor";
}

const char* HtmlEditor::kind() const
{
    return HtmlEditorConstants::C_HTMLEDITOR;
}
```

The string returned by **`displayName`** is used in the open-file combobox. The following methods help set and fetch the display name.

```
QString HtmlEditor::displayName() const
{
    return d->displayName;
}

void HtmlEditor::setDisplayName(const QString& title)
{
    if(d->displayName == title)
        return;

    d->displayName = title;

    emit changed();
}
```

We implement the following methods to do nothing in this example.

```
bool HtmlEditor::duplicateSupported() const
{
    return false;
}

Core::IEditor* HtmlEditor::duplicate(QWidget* parent)
{
    Q_UNUSED(parent);

    return 0;
}

QByteArray HtmlEditor::saveState() const
{
    return QByteArray();
}

bool HtmlEditor::restoreState(const QByteArray& state)
{
    Q_UNUSED(state);

    return false;
}

QToolBar* HtmlEditor::toolBar()
```

```
{
    return 0;
}

bool HtmlEditor::isTemporary() const
{
    return false;
}
```

5.2.4 Implementing the Core::IEditorFactory interface

The **HtmlEditorFactory** class implements the **Core::IEditorFactory** interface; and is declared as follows.

```
#include <coreplugin/editormanager/ieditorfactory.h>

struct HtmlEditorFactoryData;
class HtmlEditorFactory : public Core::IEditorFactory
{
    Q_OBJECT

public:
    HtmlEditorFactory(HtmlEditorPlugin* owner);
    ~HtmlEditorFactory();

    QStringList mimeTypes() const;
    QString kind() const;

    Core::IEditor* createEditor(QWidget* parent);
    Core::IFile* open(const QString &fileName);

private:
    HtmlEditorFactoryData* d;
};
```

HtmlEditorFactoryData structure holds the private data of the **HtmlEditorFactory** class. Notice that the constructor initializes the mime-types to **HtmlEditorConstants::C_HTMLEDITOR_MYMETYPE.** It also initializes the **kind** of the editor. This kind should be same as **kind** of **HtmlEditor**.

```
namespace HtmlEditorConstants
{
    const char* const C_HTMLEDITOR_MIMETYPE = "text/html";
    const char* const C_HTMLEDITOR          = "HTML Editor";
}

struct HtmlEditorFactoryData
{
    HtmlEditorFactoryData()
        : kind(HtmlEditorConstants::C_HTMLEDITOR)
    {
        mimeTypes << QString(HtmlEditorConstants::C_HTMLEDITOR_MIMETYPE);
    }

    QString kind;
    QStringList mimeTypes;
};
```

The following methods are self-explanatory.

```
HtmlEditorFactory::HtmlEditorFactory(HtmlEditorPlugin* owner)
                  :Core::IEditorFactory(owner)
{
    d = new HtmlEditorFactoryData;
}

HtmlEditorFactory::~HtmlEditorFactory()
{
    delete d;
}

QStringList HtmlEditorFactory::mimeTypes() const
{
    return d->mimeTypes;
}

QString HtmlEditorFactory::kind() const
{
    return d->kind;
}
```

The **`open()`** method should be implemented to return the **`IFile`** that is currently handling the given filename. If there are none, then a new editor for the file is created and it's **`IFile`** is returned. To fully understand this process take a look at the implementation of **`Core::EditorManager::openEditor()`** method.

```
Core::IFile* HtmlEditorFactory::open(const QString& fileName)
{
    Core::EditorManager* em = Core::EditorManager::instance();
    Core::IEditor* iface = em->openEditor(fileName, d->kind);
    return iface ? iface->file() : 0;
}
```

This method creates and returns an instance of the **`HtmlEditor`** class.

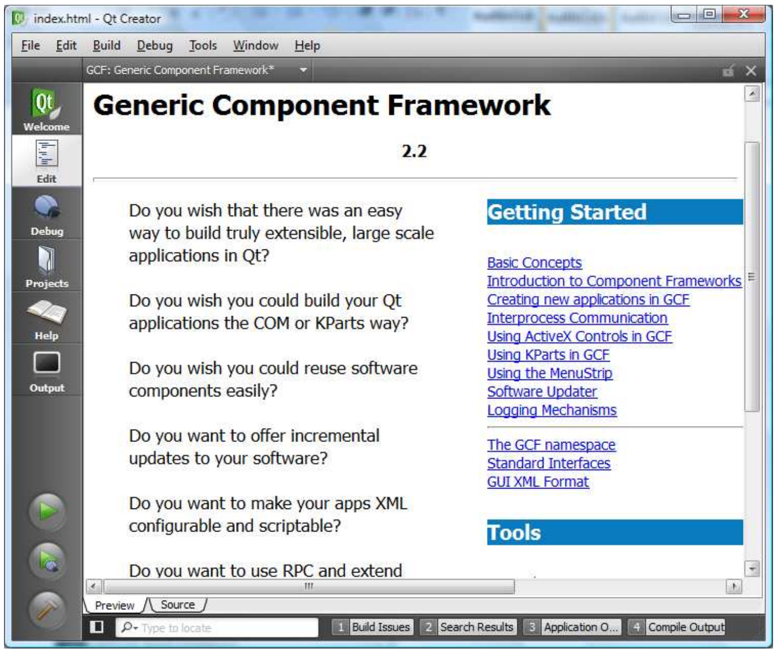
```
Core::IEditor* HtmlEditorFactory::createEditor(QWidget* parent)
{
    HtmlEditorWidget* editorWidget = new HtmlEditorWidget(parent);
    return new HtmlEditor(editorWidget);
}
```

5.2.5 Implementing the plugin

We implement the **`HtmlEditorPlugin`** plugin class using the same means described in Chapter 2. The only change is the **`initialize()`** method implementation.

```
bool HtmlEditorPlugin::initialize(const QStringList &arguments, QString* errMsg)
{
    Q_UNUSED(arguments);

    Core::ICore* core = Core::ICore::instance();
    Core::MimeDatabase* mdb = core->mimeDatabase();
    if(!mdb->addMimeTypes("text-html-mimetype.xml", errMsg))
        return false;

    addAutoReleasedObject(new HtmlEditorFactory(this));

    return true;
}
```

When the plugin is compiled and Qt Creator is (re)started; we will be able to load HTML files using the newly implemented editor plugin.



Generic Component Framework

2.2

Do you wish that there was an easy way to build truly extensible, large scale applications in Qt?

Do you wish you could build your Qt applications the COM or KParts way?

Do you wish you could reuse software components easily?

Do you want to offer incremental updates to your software?

Do you want to make your apps XML configurable and scriptable?

Do you want to use RPC and extend

Getting Started

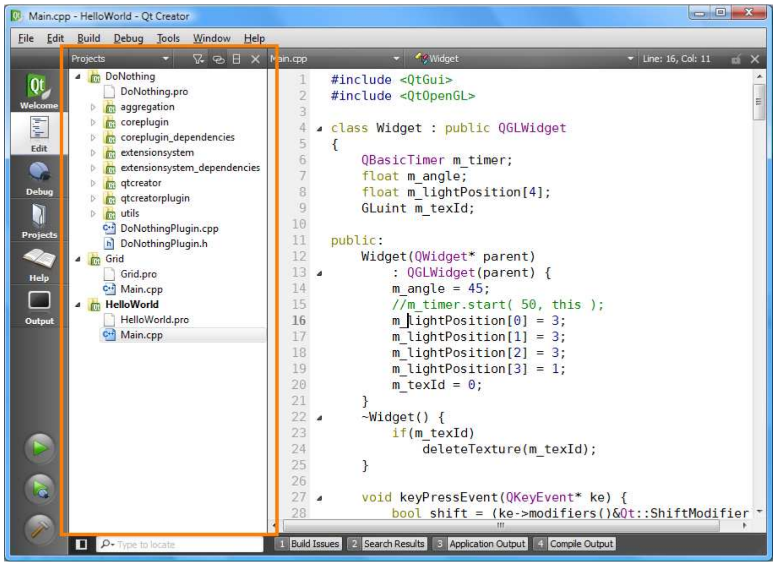
Basic Concepts
Introduction to Component Frameworks
Creating new applications in GCF
Interprocess Communication
Using ActiveX Controls in GCF
Using KParts in GCF
Using the MenuStrip
Software Updater
Logging Mechanisms

The GCF namespace
Standard Interfaces
GUI XML Format

Tools

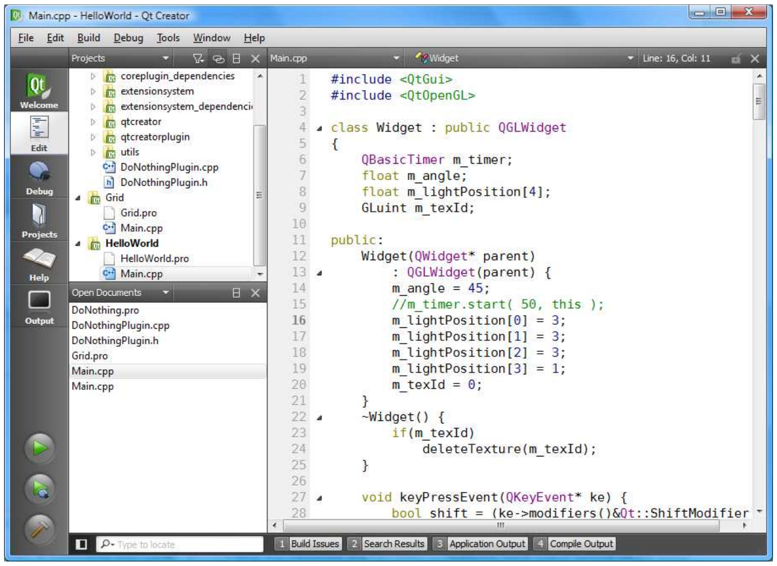
6 Adding a Navigation Sidebar

Navigation panel in Qt Creator is the area where Project, File System, Bookmark and Open Documents sidebars are shown. Sidebar is one of the widgets in the "Navigation Panel". Take a look at the marked area in the screenshot below.



```
#include <QtGui>
#include <QtOpenGL>

class Widget : public QGLWidget
{
    QBasicTimer m_timer;
    float m_angle;
    float m_lightPosition[4];
    GLuint m_texId;

public:
    Widget(QWidget* parent)
        : QGLWidget(parent) {
        m_angle = 45;
        //m_timer.start( 50, this );
        m_lightPosition[0] = 3;
        m_lightPosition[1] = 3;
        m_lightPosition[2] = 3;
        m_lightPosition[3] = 1;
        m_texId = 0;
    }
    ~Widget() {
        if(m_texId)
            deleteTexture(m_texId);
    }

    void keyPressEvent(QKeyEvent* ke) {
        bool shift = (ke->modifiers()&Qt::ShiftModifier
```

Qt Creator makes it possible for us to divide the navigation panel into windows and view more than one side bar at the same time. Take a look at the screenshot below.



```
#include <QtGui>
#include <QtOpenGL>

class Widget : public QGLWidget
{
    QBasicTimer m_timer;
    float m_angle;
    float m_lightPosition[4];
    GLuint m_texId;

public:
    Widget(QWidget* parent)
        : QGLWidget(parent) {
        m_angle = 45;
        //m_timer.start( 50, this );
        m_lightPosition[0] = 3;
        m_lightPosition[1] = 3;
        m_lightPosition[2] = 3;
        m_lightPosition[3] = 1;
        m_texId = 0;
    }
    ~Widget() {
        if(m_texId)
            deleteTexture(m_texId);
    }

    void keyPressEvent(QKeyEvent* ke) {
        bool shift = (ke->modifiers()&Qt::ShiftModifier
```

In this chapter we will understand how to add a new side bar to Qt Creator.

6.1 Core::INavigationWidgetFactory interface

The Core of Qt Creator exposes an interface called **`Core::INavigationWidgetFactory`**. The interface is defined as follows in plugins/corelib/inavigationwidgetfactory.h

```
class CORE_EXPORT INavigationWidgetFactory : public QObject
```

```
{
    Q_OBJECT

public:
    INavigationWidgetFactory();
    virtual ~INavigationWidgetFactory();

    virtual QString displayName() = 0;
    virtual QKeySequence activationSequence();
    virtual NavigationView createWidget() = 0;
    virtual void saveSettings(int position, QWidget *widget);
    virtual void restoreSettings(int position, QWidget *widget);
};
```

And NavigationView (the return type of createWidget()) is

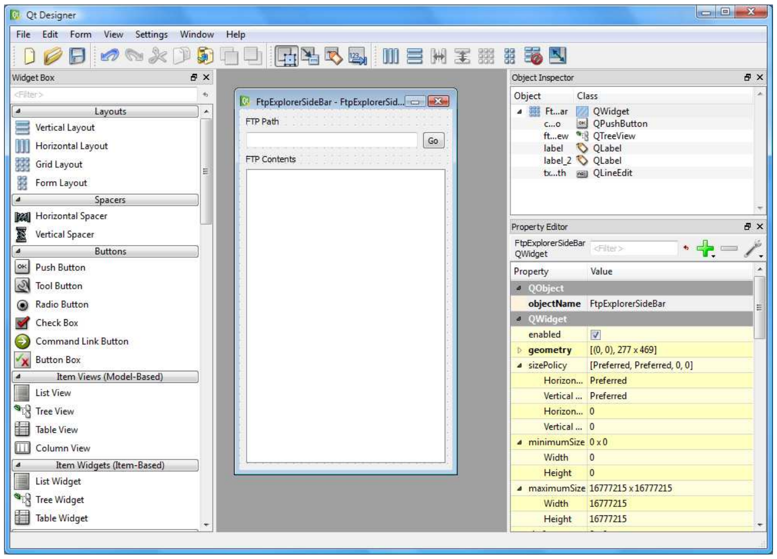
```
struct NavigationView
{
    QWidget *widget;
    QList<QToolButton *> doockToolBarWidgets;
};
```

Plugins that provide a navigation siderbar (or widget) must implement this interface. In addition to implementing the interface, the plugin has to use "expose" an instance of that interface using methods described in section 4.2.2.

6.2 Preparing a navigation sidebar (widget)

Suppose that we wanted to provide a FTP browser as a side bar widget from our plugin.

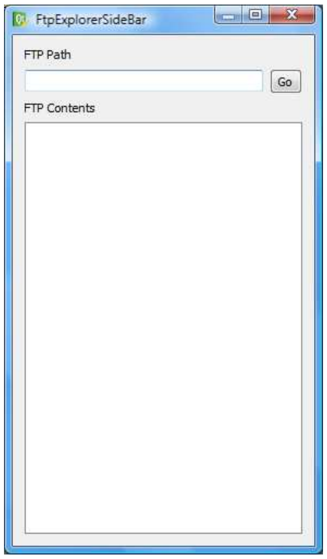
Step 1: Let's design a widget to that effect first in Qt Designer.



Object	Class
Ft...ar	QWidget
c...o	QPushButton
ft...ew	QTreeView
label	QLabel
label_2	QLabel
tx...th	QLineEdit

FtpExplorerSideBar
QWidget

Property	Value
QObject	
objectName	FtpExplorerSideBar
QWidget	
enabled	✓
geometry	[(0, 0), 277 x 469]
sizePolicy	[Preferred, Preferred, 0, 0]
Horizon...	Preferred
Vertical ...	Preferred
Horizon...	0
Vertical ...	0
minimumSize	0 x 0
Width	0
Height	0
maximumSize	16777215 x 16777215
Width	16777215
Height	16777215

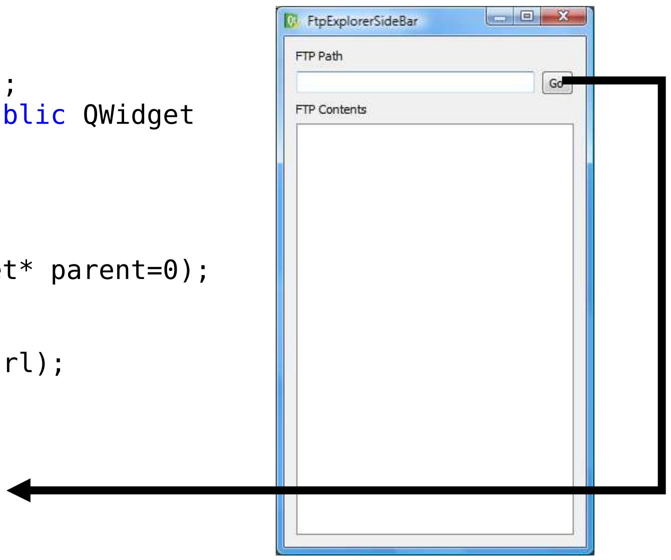


When the user enters the URL of a FTP directory in the "FTP Path" line edit and clicks "Go", the contents of the FTP directory are shown in the QTreeView below. For the sake of this example we call the widget as **`FtpExplorerSideBar`**.

Step 2: Implementing the FtpExplorerSideBar widget

The **`FtpExplorerSideBar`** class definition is as follows

```
#include <QWidget>

struct FtpExplorerSideBarData;
class FtpExplorerSideBar : public QWidget
{
    Q_OBJECT

public:
    FtpExplorerSideBar(QWidget* parent=0);
    ~FtpExplorerSideBar();

    void setUrl(const QUrl& url);
    QUrl url() const;

protected slots:
    void on_cmdGo_clicked();

private:
    FtpExplorerSideBarData* d;
};
```



The UI design is done in Qt Designer, so we only leave open a slot for responding to the **`clicked()`** signal from the **`cmdGo`** push button.

The implementation of **`FtpExplorerSideBar`** is as follows

```
#include "FtpExplorerSideBar.h"
#include "ui_FtpExplorerSideBar.h"

#include "FtpDirModel.h"

struct FtpExplorerSideBarData
{
    FtpDirModel* model;
    Ui::FtpExplorerSideBar ui;
};
```

We make use of the FtpDirModel class from VCL for serving contents of a FTP directory via an QAbstractItemModel interface. Visit http://www.vcreatelogic.com/p/2009/07/ftpdirmodel-a-qabstractitemmodel-implementation-for-browsing-ftp-directories/ to download a copy of the FtpDirModel code for your use.

The constructor basically initializes the UI and creates the model-view association.

```
FtpExplorerSideBar::FtpExplorerSideBar(QWidget* parent)
:QWidget(parent)
{
    d = new FtpExplorerSideBarData;
    d->ui.setupUi(this);

    d->model = new FtpDirModel(this);
    d->ui.ftpView->setModel(d->model);
}
```

The destructor does nothing other than deleting the private '**d**' object.

```
FtpExplorerSideBar::~FtpExplorerSideBar()
{
    delete d;
}
```

The setter and getter methods for URL basically transfer the URL to the model.

```
void FtpExplorerSideBar::setUrl(const QUrl& url)
{
    d->model->setUrl(url);
}

QUrl FtpExplorerSideBar::url() const
{
    return d->model->url();
}
```

When the "Go" button is clicked, the **on_cmdGo_clicked()** slot is called. We basically set the URL for the **FtpDirModel** to fetch and have the view update itself.

```
void FtpExplorerSideBar::on_cmdGo_clicked()
{
    QUrl url(d->ui.txtFtpPath->text());
    d->model->setUrl(url);
}
```

With this the **FtpExplorerSideBar** widget is ready.

Step 3: Implementing the INavigationWidgetFactory interface

We implement the **INavigationWidgetFactory** interface in a class whose definition is as follows

```
#include <coreplugin/inavigationwidgetfactory.h>

class FtpViewNavigationWidgetFactory
      : public Core::INavigationWidgetFactory
{
public:
    FtpViewNavigationWidgetFactory() { }
    ~FtpViewNavigationWidgetFactory() { }

    Core::NavigationView createWidget();
    QString displayName();
};
```

The **createWidget()** method is implemented to return an instance of the **FtpExplorerSideBar** widget that was explained in the previous step.

```
Core::NavigationView FtpViewNavigationWidgetFactory::createWidget()
{
    Core::NavigationView view;
    view.widget = new FtpExplorerSideBar;

    return view;
}
```

The **`displayName()`** method is implemented to return a descriptive name that Qt Creator should use for showing the side-bar.

```
QString FtpViewNavigationWidgetFactory::displayName()
{
    return "Ftp View";
}
```

With this the **`INavigationWidgetFactory`** implementation is ready.

Step 4: Implementing the ftp-view plugin

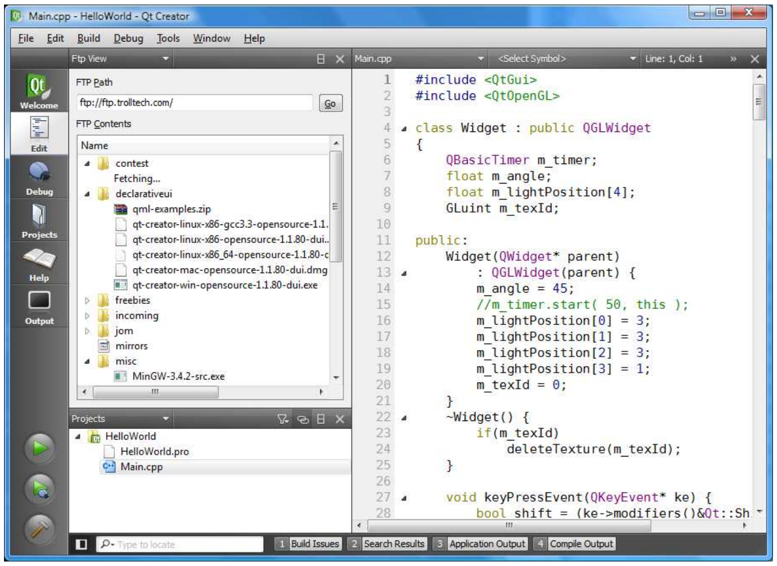
We implement the ftp-view plugin class similar to the **`DoNothingPlugin`** class described in Chapter 2. Hence, we only describe the implementation of the initialize method of the **`FtpViewPlugin`** class here.

```
bool FtpViewPlugin::initialize(const QStringList& args, QString *errMsg)
{
    Q_UNUSED(args);
    Q_UNUSED(errMsg);

    addAutoReleasedObject(new FtpViewNavigationWidgetFactory);

    return true;
}
```

In the **`initialize()`** method, an instance of the **`INavigationWidgetFactory`** implementation is created and added to the object pool. Once the object is added to the pool, **`ExtensionSystem::PluginManager`** emits the **`objectAdded()`** signal, which is then trapped by the Core of Qt Creator. The Core then makes use of our implementation of **`INavigationWidgetFactory`** interface and places an instance of **`FtpExplorerSideBar`** in the navigation panel.

Step 5: Testing the plugin

Upon compiling the plugin and restarting Qt Creator, we can notice the “Ftp View” side bar as shown below.



```
#include <QtGui>
#include <QtOpenGL>

class Widget : public QGLWidget
{
    QBasicTimer m_timer;
    float m_angle;
    float m_lightPosition[4];
    GLuint m_texId;

public:
    Widget(QWidget* parent)
        : QGLWidget(parent) {
        m_angle = 45;
        //m_timer.start( 50, this );
        m_lightPosition[0] = 3;
        m_lightPosition[1] = 3;
        m_lightPosition[2] = 3;
        m_lightPosition[3] = 1;
        m_texId = 0;
    }
    ~Widget() {
        if(m_texId)
            deleteTexture(m_texId);
    }

    void keyPressEvent(QKeyEvent* ke) {
        bool shift = (ke->modifiers()&Qt::Sh
```

6.3 Saving and restoring sidebar states

Sometimes it may be necessary to save the state of the sidebar when Qt Creator is shutdown, so that the state can be restored when Qt Creator is restarted next.

The **INavigationWidgetFactory** provides two virtual methods

```
class CORE_EXPORT INavigationWidgetFactory : public QObject
{
    Q_OBJECT

public:
    INavigationWidgetFactory();
    virtual ~INavigationWidgetFactory();

    virtual QString displayName() = 0;
    virtual QKeySequence activationSequence();
    virtual NavigationView createWidget() = 0;
    virtual void saveSettings(int position, QWidget *widget);
    virtual void restoreSettings(int position, QWidget *widget);
};
```

- The **saveSettings()** method can be re-implemented to save the settings of the widget (whose pointer is provided as second parameter) into the **QSettings** returned by **Core::ICore:: settings()**.
- The **restoreSettings()** method can be re-implemented to restore the settings of the widget.

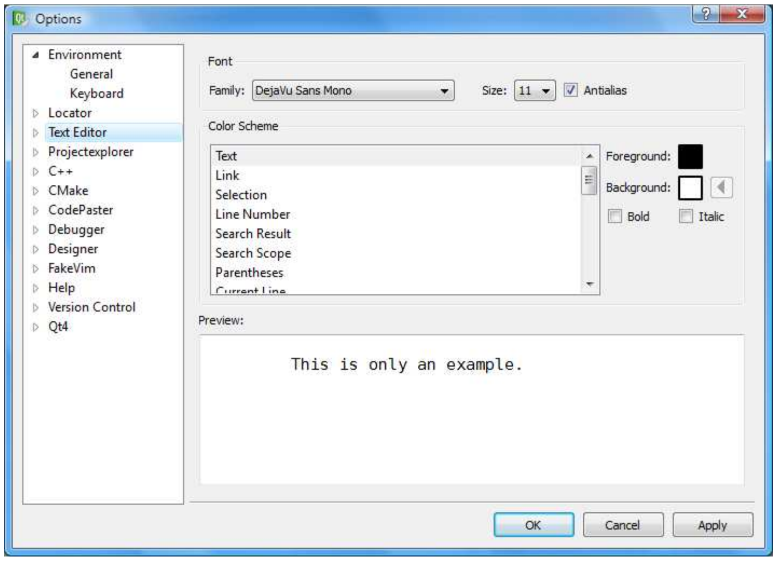
Shown below is a sample implementation of the above methods for the **FtpViewNavigationWidgetFactory** class.

```
void FtpViewNavigationWidgetFactory::saveSettings
     (int position, QWidget *widget);
{
    FtpExplorerSideBar* ftpExp = qobject_cast<FtpExplorerSideBar*>(widget);
    if(!ftpExp)
        return;
    QSettings *settings = Core::ICore::instance()->settings();
    settings->setValue("FtpView.URL", ftpExp->url().toString());
}

void FtpViewNavigationWidgetFactory::restoreSettings
     (int position, QWidget *widget)
{
    FtpExplorerSideBar* ftpExp = qobject_cast<FtpExplorerSideBar*>(widget);
    if(!ftpExp)
        return;

    QSettings *settings = Core::ICore::instance()->settings();
    QString urlStr = settings->value("FtpView.URL").toString();

    ftpExp->setUrl( QUrl(urlStr) );
}
```

By implementing the **`saveSettings()`** and **`restoreSettings()`** method, we can ensure that the FTP path browsed in the last session is reopened in the new session.

7 Adding Page to Preferences Dialog

Preferences dialog in Qt Creator is used to configure the Qt Creator settings. Since Qt Creator is just a plugin loader that loads all the relevant plugins, the preferences dialog shows pages that configure plugins. You can get to it by clicking Tools->Options.



Family: DejaVu Sans Mono Size: 11 [✓ Antialias]

```
This is only an example.
```

Each plugin provides one or more options pages that get shown in the preferences dialog. In the following sub-sections we will learn how to add our own pages to the dialog.

7. 1 Core::IOptionsPage interface

The Core of Qt Creator exposes an interface called **`Core::IOptionsPage`**. The interface is defined in plugins/coreplugin/dialogs/ioptionspage.h.

```
class CORE_EXPORT IOptionsPage : public QObject
{
     Q_OBJECT
public:
     IOptionsPage( *parent = 0) : QObject(parent) {}
     virtual ~IOptionsPage() {}

     virtual QString id() const = 0;
     virtual QString trName() const = 0;
     virtual QString category() const = 0;
     virtual QString trCategory() const = 0;

     virtual QWidget *createPage(QWidget *parent) = 0;
     virtual void apply() = 0;
     virtual void finish() = 0;
};
```

By implementing the above interface and exposing an instance of it, we will be able to register new pages with the preferences dialog.

7.2 Preparing the options-page

Let's implement a plugin that shows an options page that lists out all the open and modified files.

Step 1: Implementing the "modified file" list widget

The modified file list widget is simply a **`QListWidget`** that shows all the modified files from the project manager. The class declaration is as follows

```
#include <QListWidget>

class ModifiedFileListWidget: public QListWidget
{
    Q_OBJECT

public:
    ModifiedFileListWidget(QWidget* parent=0);
    ~ModifiedFileListWidget();
};
```

Within the constructor we populate the list widget with names of the modified pages

```
#include <coreplugin/filemanager.h>
#include <coreplugin/icore.h>
#include <coreplugin/ifile.h>

ModifiedFileListWidget::ModifiedFileListWidget(QWidget* parent)
:QListWidget(parent)
{
    // Show the list of modified pages
    Core::FileManager* fm = Core::ICore::instance()->fileManager();
    QList<Core::IFile*> files = fm->modifiedFiles();

    for(int i=0; i<files.count();i++)
        this->addItem(files.at(i)->fileName());
}
```

The destructor does nothing.

```
ModifiedFileListerPage::~ModifiedFileListerPage()
{

}
```

Step 2: Implementing the Core::IOptionsPage interface

We implement the **`Core::IOptionsPage`** interface in a class called **`ModifiedFileLister`**. The class declaration is as follows

```
#include <coreplugin/dialogs/ioptionspage.h>

class ModifiedFileLister : public Core::IOptionsPage
{
    Q_OBJECT

public:
    ModifiedFileLister(QObject *parent = 0);
    ~ModifiedFileLister();

    // IOptionsPage implementation
    QString id() const;
    QString trName() const;
    QString category() const;
    QString trCategory() const;
    QWidget *createPage(QWidget *parent);
    void apply();
    void finish();
};
```

The constructor and destructor are straightforward and easy to understand.

```
ModifiedFileLister::ModifiedFileLister(QObject *parent)
     : IOptionsPage(parent)
{

}

ModifiedFileLister::~ModifiedFileLister()
{

}
```

The **`id()`** method should be implemented to return a unique identifier for the options page provided by this class. The string will be used internally to ***id***entify the page.

```
QString ModifiedFileLister::id() const
{
    return "ModifiedFiles";
}
```

The **`trName()`** method should be implemented to return a translated string name that will be shown in the options dialog.

```
QString ModifiedFileLister::trName() const
{
    return tr("Modified Files");
}
```

The **`category()`** and **`trCategory()`** methods should be implemented to return the group under which we want to show the page. The latter returns the translated version of the string returned by the former.

```
QString ModifiedFileLister::category() const
{
    return "Help";
}

QString ModifiedFileLister::trCategory() const
{
    return tr("Help");
}
```

The **createPage()** method should be implemented to return a new instance of the page implemented in step 1.

```
QWidget *ModifiedFileLister::createPage(QWidget *parent)
{
    return new ModifiedFileListWidget(parent);
}
```

The methods **apply()** and **finish()** can be implemented to accept the changes made by the user made on the page. In our case we don't have any changes to accept, so we leave the methods empty.

```
void ModifiedFileLister::apply()
{
    // Do nothing
}

void ModifiedFileLister::finish()
{
    // Do nothing
}
```

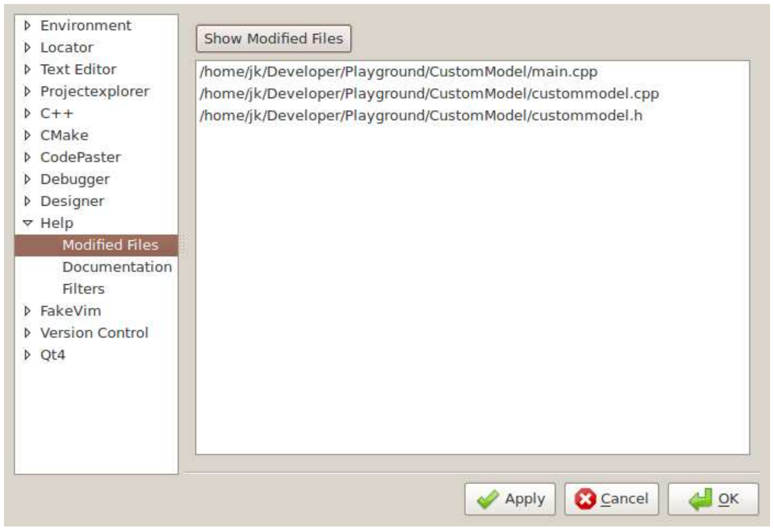
Step 3: Implementing the modified-file-lister plugin

We implement the plugin class similar to the **DoNothingPlugin** class described in Chapter 2. Hence, we only describe the implementation of the initialize method of the **ModifiedFileListerPlugin** class here.

```
bool ModifiedFileListerPlugin::initialize(const QStringList& args, QString *errMsg)
{
    Q_UNUSED(args);
    Q_UNUSED(errMsg);

    addAutoReleasedObject(new ModifiedFileLister);

    return true;
}
```

Step 4: Testing the plugin

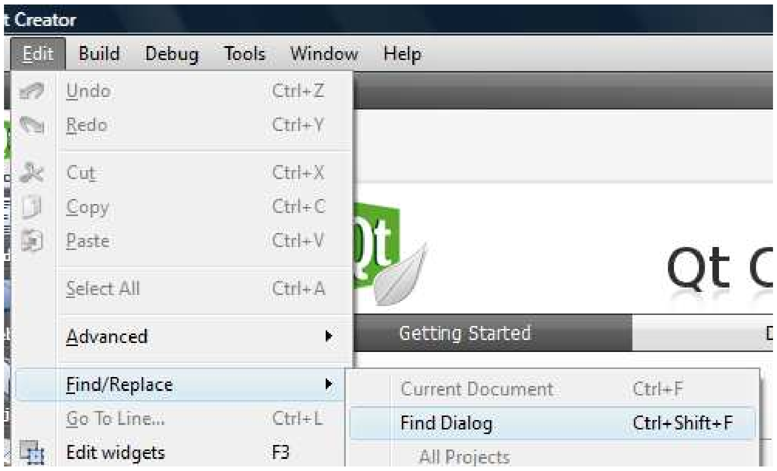
Upon compiling the plugin and restarting Qt Creator, we can notice in the options dialog the newly added "Modified Files" page.

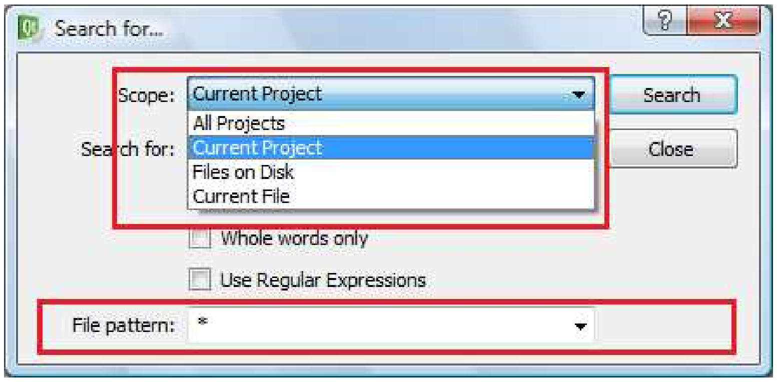


```
/home/jk/Developer/Playground/CustomModel/main.cpp
/home/jk/Developer/Playground/CustomModel/custommodel.cpp
/home/jk/Developer/Playground/CustomModel/custommodel.h
```

8 ADDING FILTER TO FIND DIALOG BOX

Qt Creator's find dialog box allows users to search for a text or regular expression in opened projects and unloaded disk files. Clicking on "Edit -> Find/Replace -> Find Dialog" shows the find dialog box.





In the find dialog box the "scope" and "configuration widget" are extensible. It is possible to add more items to the scope combo box and against every item it is possible to provide a configuration widget that needs to be shown below. Each item in the "scope" combo box is called "find filter" in Qt Creator lingo.

8.1 Find::IFindFilter interface

The *find* plugin in Qt Creator exposes an interface called **`Find::IFindFilter`**. The interface is declared as follows in the src/plugins/find/ifindfilter.h header.

```
class FIND_EXPORT IFindFilter : public QObject
{
    Q_OBJECT
public:

    virtual ~IFindFilter() {}

    virtual QString id() const = 0;
    virtual QString name() const = 0;
    virtual bool isEnabled() const = 0;
    virtual QKeySequence defaultShortcut() const = 0;
    virtual void findAll(const QString &txt, QTextDocument::FindFlags findFlags) = 0;

    virtual QWidget *createConfigWidget() { return 0; }
    virtual void writeSettings(QSettings *settings) { Q_UNUSED(settings); }
    virtual void readSettings(QSettings *settings) { Q_UNUSED(settings); }

signals:
    void changed();
};
```

By implementing the **IFindFilter** interface and adding an instance of it to the object pool, we will be able to add a new find-filter; which means another entry in the "scope" combo box of the find dialog box.

8.2 Providing a custom filter

Suppose that we wanted to provide a custom filter that will allow us to look for files in the loaded projects that include a given header. In the following steps we will understand how to write a find filter for this.

Step 1: Declaring the HeaderFilter class

We first begin by declaring a class called HeaderFilter that implements the **Find::IFindFilter** interface. The class definition is as follows.

```
#include <find/ifindfilter.h>

class HeaderFilter : public Find::IFindFilter
{
    Q_OBJECT

public:
    HeaderFilter();
    ~HeaderFilter();

    QString id() const;
    QString name() const;
    bool isEnabled() const;
    QKeySequence defaultShortcut() const;
    void findAll(const QString &txt,
                 QTextDocument::FindFlags findFlags);
    QWidget *createConfigWidget();

private:
    HeaderFilterData *d;
};
```

Step 2: Implementing the HeaderFilter class

The constructor and destructors are currently empty. We will fill in more code as we progress with our understanding of the **IFindFilter** interface.

```
struct HeaderFilterData
{

};

HeaderFilter:: HeaderFilter()
```

```
{
    d = new HeaderFilterData;
}

HeaderFilter::~ HeaderFilter()
{
    delete d;
}
```

The **id()** method should be implemented to return a unique identifier for the find filter.

```
QString HeaderFilter::id() const
{
    return "HeaderFilter";
}
```

The **name()** method should be implemented to return the string that gets shown in the “scope” combo box of the find dialog box.

```
QString HeaderFilter::name() const
{
    return tr("Header Filter");
}
```

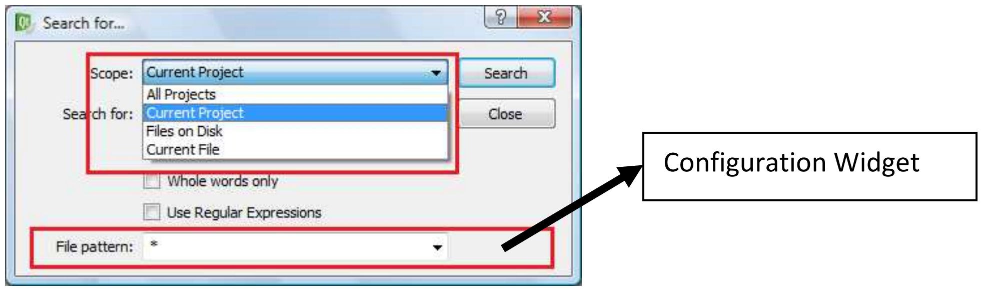
The **`isEnabled()`** method should be implemented to return whether the find filter is enabled or not. In our case we would like to show the filter enabled if projects are loaded in Qt Creator, false otherwise. To fully understand the implementation of the function, we must first study the **`ProjectExplorer`** namespace. For now let’s just return true and revisit the function after learning about the **`ProjectExplorer`** namespace.

```
bool HeaderFilter::isEnabled() const
{
    return true;
}
```

The **`defaultShortcut()`** method should be implemented to return a key-sequence that the user can use to launch the find dialog box with the “header filter” selected in “scope”. In our implementation we return an invalid key-sequence.

```
QKeySequence HeaderFilter::defaultShortcut() const
{
    return QKeySequence();
}
```

The **`createConfigWidget()`** method should be implemented to return a configuration widget that gets shown at the bottom edge of the find dialog box.



For our header-filter; let's return a simple **`QLabel`** because we don't want to provide any configuration options.

```
QWidget *HeaderFilter::createConfigWidget()
{
    return (new QLabel("This is a header filter"));
}
```

The **`findAll()`** method should be implemented to perform the actual "find" or "search" operation. We need to understand few key classes in the **`ProjectExplorer`**, **`TextEditor`**, **`Find`** and **`Core::Utils`** namespace before attempting to implement the filter. For now implement the method to do nothing.

```
void HeaderFilter::findAll(
        const QString &text,
        QTextDocument::FindFlags findFlags
    )
{
    // Do nothing
}
```

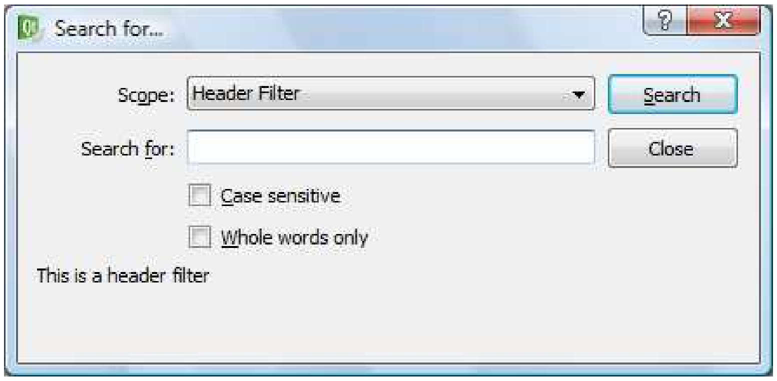
Step 3: Implementing the "header-filter" plugin.

We implement the header-filter plugin very similar to the **`DoNothingPlugin`** class described in Chapter 2. Here we only look at the implementation if the **`initialize()`** method.

```
bool HeaderFilterPlugin::initialize(
        const QStringList& args,
        QString *errMsg)
{
    Q_UNUSED(args);
    Q_UNUSED(errMsg);

    addAutoReleasedObject(new HeaderFilter);

    return true;
}
```

Step 4: Testing the plugin

Upon compiling the plugin and restarting Qt Creator, we can notice the "Header Filter" item in the scope combo box of the find dialog box.



Scope: Header Filter
Search for:
Case sensitive
Whole words only
This is a header filter

Currently no "finding" or "searching" is done by our plugin because we have not yet implemented the **`HeaderFilter::findAll()`** method.

8.3 The ProjectExplorer namespace

The **`ProjectExplorer`** namespace comprises of classes and interfaces that make up the project management system in Qt Creator. This namespace is provided by the ***projectexplorer*** plugin. Support for project types are provided by plugins. For example

- ***cmakeprojectmanager*** plugin provides implementations of interfaces in **`ProjectExplorer`** namespace for supporting CMake projects
- ***qt4projectmanager*** plugin provides support for Qt 4 projects
- ***qmlprojectmanager*** plugin provides support for QML projects

Let's take a look at few key classes and interfaces in the **`ProjectManager`** namespace

Class/Interface	Description
`ProjectExplorer::IProjectManager`	This interface must be implemented to provide support for a kind of project. Implementations of this interface help load projects into Qt Creator.
`ProjectExplorer::Project`	This interface describes a project in terms of • A file (**`Core::IFile`**) that describes the project. • A Boolean flag describing whether the project builds an application or library • Build steps (**`ProjectExplorer:: BuildStep`**) that need to be performed in order to build and clean the project • Run configurations that need to be used for running the project • The environment within which the project needs to be run • The root node in the project explorer panel • Include paths and macros to be used while building the project
`ProjectManager::`	This class is the implementation of the Core::IPlugin interface for the

ProjectExplorerPlugin	project explorer plugin. Through this class we can • gain access to all the open projects • gain access the current project • gain access to the currently selected node (file/folder) in the project explorer panel • gain access to the build manager (**ProjectManager::BuildManager**)

8.3.1 Getting a list of open-projects

Using the **ProjectManager::ProjectExplorerPlugin** class we can catch hold of all the open-projects in Qt Creator. The following code snippet shows how to do that

```
#include <extensionsystem/pluginmanager.h>
#include <projectexplorer/projectexplorer.h>

// Catch hold of the plugin-manager
ExtensionSystem::PluginManager* pm
        = ExtensionSystem::PluginManager::instance();

// Look for the ProjectExplorerPlugin object
ProjectExplorer::ProjectExplorerPlugin* projectExplorerPlugin
    = pm->getObject<ProjectExplorer::ProjectExplorerPlugin>();

// Fetch a list of all open projects
QList<ProjectExplorer::Project*> projects
    = d->projectPlugin->session()->projects();
```

From the **projects** list we can gain access to the project file and all the other (source, header, resource etc) files in the project. To gain access to the project file(s) we can

```
Q_FOREACH(ProjectExplorer::Project* project, projects)
{
    QString name = project->name();
    Core::IFile* projectFile = project->file();

    // Do something with the above. For example:
    qDebug("Project %s has project file as %s",
           qPrintable(name),
           qPrintable(projectFile->fileName()));
}
```

While the above code snippet helps with fetching the project file (CMakeLists.txt, .pro etc..), it doesn't help us fetch all the files associated with the project.

8.3.2 Getting a list of files

From the **projects** list we can get a string-list of all file names associated with the project using the following code snippet.

```
// Make a list of files in each project
QStringList files;
```

```
Q_FOREACH(ProjectManager::Project* project, projects)
    files += project->files(Project::AllFiles);
```

8.3.3 Enabling the HeaderFilter conditionally

Ideally the header-filter should be enabled only if there is atleast one open project. To make this happen, we upgrade the HeaderFilter implementation as follows

```
struct HeaderFilterData
{
    ProjectExplorer::ProjectExplorerPlugin* projectExplorer() {
        if(m_projectPlugin)
            return m_projectPlugin;

        ExtensionSystem::PluginManager* pm
                = ExtensionSystem::PluginManager::instance();
        m_projectPlugin
            = pm->getObject<ProjectExplorer::ProjectExplorerPlugin>();

        return m_projectPlugin;
    }

private:
    ProjectExplorer::ProjectExplorerPlugin* m_projectPlugin;
};

bool HeaderFilter::isEnabled() const
{
    QList<ProjectManager::Project*> projects
        = d->projectExplorer()->openProjects();

    if(projects.count())
        return true;

    return false;
}
```

8.4 Searching in files

In the previous section we understood how to gain access to the file names of all files associated with open projects. We are now in a position to search within files. Let's begin the implementation of the **HeaderFilter::findAll()** method, and understand more concepts as we progress.

```
void HeaderFilter::findAll(
        const QString &text,
        QTextDocument::FindFlags findFlags
    )
{
    // Fetch a list of all open projects
    QList<Project*> projects
        = d->projectPlugin->session()->projects();

    // Make a list of files in each project
    QStringList files;
    Q_FOREACH(Project* project, projects)
        files += project->files(Project::AllFiles);
    // Remove duplicates
    files.removeDuplicates();

    // Search for text in 'files'
    // ...
}
```

The number of files that need to be searched can be varying. It may be as little as 1 and as high as 1000 or more! Hence searching for something in files within the **`findAll()`** method is a bad idea. If the **`findAll()`** method takes too long then it may cause Qt Creator to appear frozen until searching is finished.

The solution to this is

- We make use of **`QtConcurrent`** and spawn multiple threads to perform the actual searching
- We initialize a **`QFutureWatcher`** on the **`QFuture`** returned by **`QtConcurrent`** to emit signals as and when search results are generated
- We catch the signals generated by **`QFutureWatcher`** and list search results as they come

Qt Creator's core utils library provides a readymade function called **`findInFiles()`** that looks for a string within a list of files and returns a **`QFuture`** to monitor search results. The function is declared as follows in src/libs/utils/filesearch.h

```
namespace Core {
namespace Utils {

class FileSearchResult
{
public:
    QString fileName;
    int lineNumber;
    QString matchingLine;
    int matchStart;
    int matchLength;
};

QFuture<FileSearchResult> findInFiles(
                                const QString &searchTerm,
                                const QStringList &files,
                                QTextDocument::FindFlags flags
                            );

QFuture<FileSearchResult> findInFilesRegExp(
                                const QString &searchTerm,
                                const QStringList &files,
                                QTextDocument::FindFlags flags
                            );
} // namespace Utils
} // namespace Core
```

Lets now continue with the implementation of **`HeaderFilter::findAll()`** by making use of the **`findInFiles()`** method.

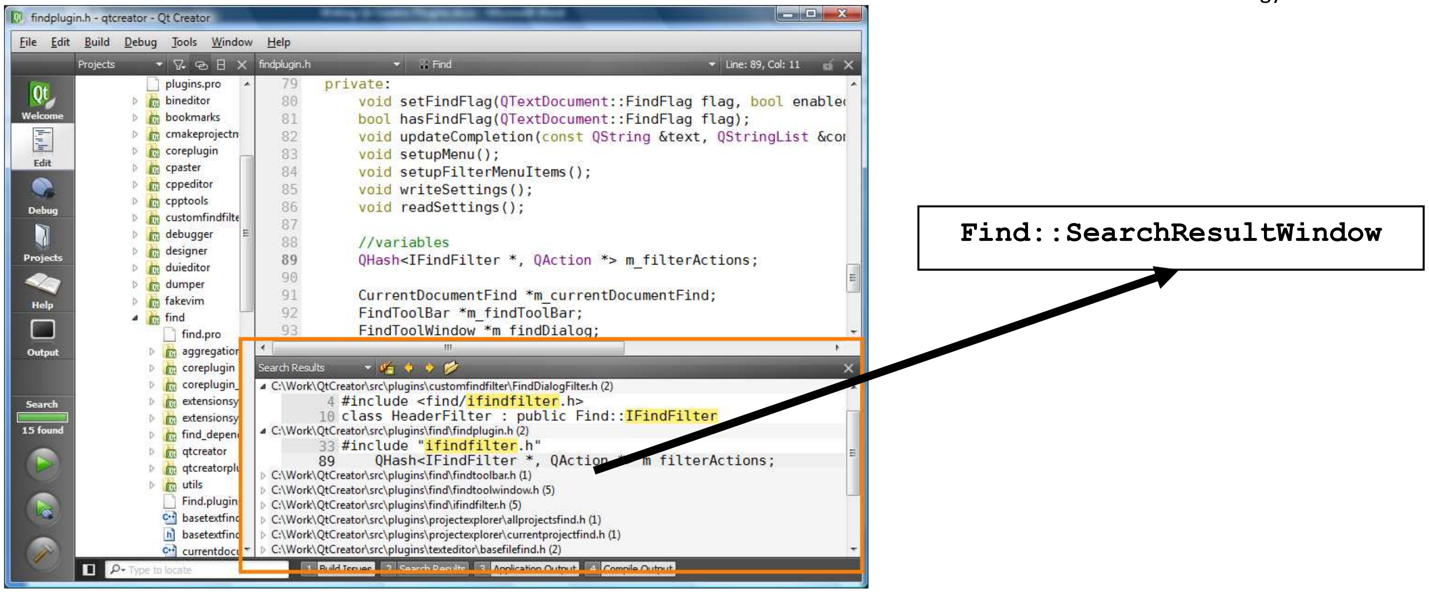
```
struct HeaderFilterData
```

```
{
    QFutureWatcher<FileSearchResult> watcher;

      ProjectExplorer::ProjectExplorerPlugin* projectExplorer() {
          ...
    }
};

HeaderFilter::HeaderFilter()
{
    d->watcher.setPendingResultsLimit(1);
    connect(&d->watcher, SIGNAL(resultReadyAt(int)),
            this, SLOT(displayResult(int)));
}

void HeaderFilter::findAll(
        const QString &text,
        QTextDocument::FindFlags findFlags
    )
{
    ....

    // Remove duplicates
    files.removeDuplicates();

    // Variable to hold search results that will
    // come as the search task progresses
    QFuture<FileSearchResult> searchResults;

    // Begin searching
    QString includeline = "#include <" + text + ">";
    searchResults = Core::Utils::findInFiles(includeline, files, findFlags);

    // Let the watcher monitor the search results
    d->watcher.setFuture(searchResults);
}

void HeaderFilter::displayResult(int index)
{
    // TODO
}
```

In the revised **`findAll()`** implementation we make use of the **`findInFiles()`** method to spawn multiple background threads to do all the finding. As search results are generated, the **`displayResult(int)`** slot is called. In this slot we can now show search results to the user.

8.5 Showing search results

The ‘find’ plugin provides an object of class **`Find::SearchResultWindow`**. This class provides access to the widget that displays search results.



```
private:
    void setFindFlag(QTextDocument::FindFlag flag, bool enable
    bool hasFindFlag(QTextDocument::FindFlag flag);
    void updateCompletion(const QString &text, QStringList &co
    void setupMenu();
    void setupFilterMenuItems();
    void writeSettings();
    void readSettings();

    //variables
    QHash<IFindFilter *, QAction *> m_filterActions;

    CurrentDocumentFind *m_currentDocumentFind;
    FindToolBar *m_findToolBar;
    FindToolWindow *m_findDialog;
```

```
C:\Work\QtCreator\src\plugins\customfindfilter\FindDialogFilter.h (2)
     4 #include <find/ifindfilter.h>
    10 class HeaderFilter : public Find::IFindFilter
C:\Work\QtCreator\src\plugins\find\findplugin.h (2)
    33 #include "ifindfilter.h"
    89     QHash<IFindFilter *, QAction *> m_filterActions;
C:\Work\QtCreator\src\plugins\find\findtoolbar.h (1)
C:\Work\QtCreator\src\plugins\find\findtoolwindow.h (5)
C:\Work\QtCreator\src\plugins\find\ifindfilter.h (5)
C:\Work\QtCreator\src\plugins\projectexplorer\allprojectsfind.h (1)
C:\Work\QtCreator\src\plugins\projectexplorer\currentprojectfind.h (1)
C:\Work\QtCreator\src\plugins\texteditor\basefilefind.h (2)
```

We would like to show our search results in the "search result window" as well. To do so, we modify the **HeaderFilter** code as follows

```
#include <find/searchresultwindow.h>

struct HeaderFilterData
{
    // Method to search and return the search window
    Find::SearchResultWindow* searchResultWindow() {
        if(m_searchResultWindow)
            return m_searchResultWindow;

        ExtensionSystem::PluginManager* pm
                = ExtensionSystem::PluginManager::instance();
        m_searchResultWindow
            = pm->getObject<Find::SearchResultWindow>();

        return m_searchResultWindow;
    }

private:
    Find::SearchResultWindow *m_searchResultWindow;
};

HeaderFilter::HeaderFilter()
{
    // displayResult(int) is called when every a new
    // search result is generated
    connect(&d->watcher, SIGNAL(resultReadyAt(int)),
            this, SLOT(displayResult(int)));
}

void HeaderFilter::findAll(
        const QString &text,
        QTextDocument::FindFlags findFlags
    )
```

```
{
    ...

    // Clear the current search results
    d->searchResultWindow()->clearContents();

    // Begin searching
    QString includeline = "#include <" + text + ">";
    searchResults = findInFiles(text, files, findFlags);

    ...
}

void HeaderFilter::displayResult(int index)
{
    FileSearchResult result = d->watcher.future().resultAt(index);
    ResultWindowItem *item = d->searchResultWindow()->addResult(result.fileName,
                              result.lineNumber,
                              result.matchingLine,
                              result.matchStart,
                              result.matchLength);
    if (item)
        connect(item, SIGNAL(activated(const QString&,int,int)),
                this, SLOT(openEditor(const QString&,int,int)));
}

void HeaderFilter::openEditor(const QString &fileName, int line, int column)
{
    // TODO
}
```

Whenever the user double clicks on the search results, the **`openEditor()`** method is called. In that method we should have Qt Creator open the corresponding file and mark the searched text.

8.6 Opening the searched files

Without going into too many details about the texteditor plugin, lets just take a look at the function that loads a named file for us and moves the cursor to a specified location. The **`BaseTextEditor`** class in the **`TextEditor`** namespace provides a static method called **`openEditorAt()`** that serves our purpose.

```
namespace TextEditor {

class BaseTextEditor : public QPlainTextEdit
{
public:
    ....

    static ITextEditor *openEditorAt(
                const QString &fileName,
                int line, int column = 0,
                const QString &editorKind
                );
    ....
};
} // TextEditor namespace
```

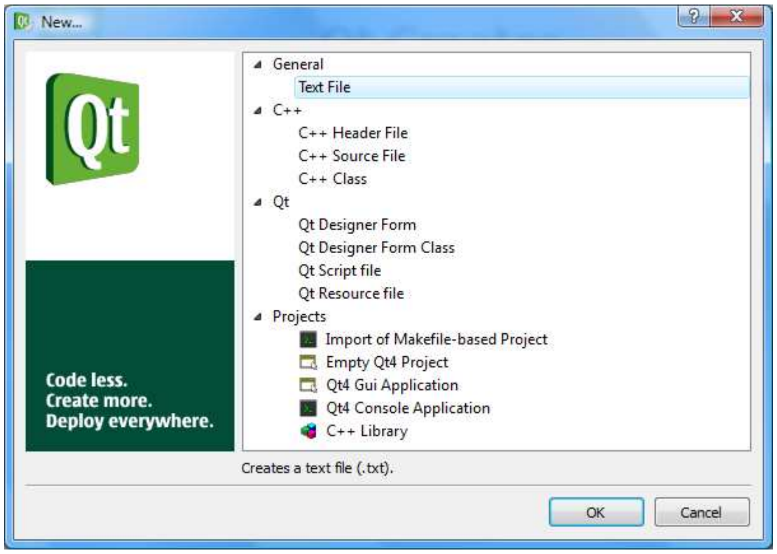
We now update the **`HeaderFilter::openEditor()`** slot as follows

```
#include <texteditor/basetexteditor.h>

void HeaderFilter::openEditor(const QString &fileName, int line, int column)
{
    TextEditor::BaseTextEditor::openEditorAt(fileName, line, column);
}
```

With this the **`HeaderFilter`** is now complete.

9 ADD A NEW PROJECT TYPE

New projects in Qt Creator can be created by clicking on the "File -> New" menu item and selecting the required project type. Shown below is the new project dialog box.



Creates a text file (.txt).

In this chapter we will learn how to add new project types into the dialog box above.

9.1 Core::IWizard interface

Qt Creator provides a Core::IWizard interface that can be implemented to support new project types. The interface is defined as follows in **`src/plugins/coreplugin/dialogs/iwizard.h`**.

```
class CORE_EXPORT IWizard : public QObject
{
    Q_OBJECT
public:
    enum Kind {
        FileWizard,
        ClassWizard,
        ProjectWizard
    };

    virtual Kind kind() const = 0;
    virtual QIcon icon() const = 0;
    virtual QString description() const = 0;
    virtual QString name() const = 0;
    virtual QString category() const = 0;
    virtual QString trCategory() const = 0;
    virtual QStringList runWizard(const QString &path, QWidget *parent) = 0;
};
```

Qt Creator supports the following types of new entities

- File
- Class

- Project

Core::IWizard has to be implemented to support any of the above project types.

9.1.1 Sample implementation of Core::IWizard

Let’s implement the **IWizard** interface to support a new project type called “Custom Project”. The idea is to see the new project type listed in the new project wizard that shows up on clicking “File -> New”.

Step 1: Implementing the Core::IWizard interface

Lets create a class called **CustomProjectWizard** and subclass it from **Core::IWizard**.

```
class CustomProjectWizard : public Core::IWizard
{
public:
    CustomProjectWizard() { }
    ~CustomProjectWizard() { }

    Core::IWizard::Kind kind() const;
    QIcon icon() const;
    QString description() const;
    QString name() const;

    QString category() const;
    QString trCategory() const;

    QStringList runWizard(const QString &path, QWidget *parent);
};
```

Below we will discuss the implementation of each of the functions.

The **kind()** function should be implemented to return the type of “new” project we support in our implementation of **IWizard**. Valid values are **FileWizard**, **ClassWizard** and **ProjectWizard**. In our implementation we return **ProjectWizard**.

```
Core::IWizard::Kind CustomProjectWizard::kind() const
{
    return IWizard::ProjectWizard;
}
```

The **icon()** implementation must return an icon to use against the project type in the New project dialog box. In our implementation we return the Qt Creator icon itself.

```
QIcon CustomProjectWizard::icon() const
{
    return qApp->windowIcon();
}
```

The **description()**, **name()** and **category()** methods must return some meta data of the new project/file/class type we are providing in the **IWizard** implementation.

```
QString CustomProjectWizard::description() const
{
    return "A custom project";
}

QString CustomProjectWizard::name() const
{
    return "CustomProject";
}

QString CustomProjectWizard::category() const
{
    return "VCreate Logic";
}
```

The **`trCategory()`** method should be implemented to return a translated category string. This is the name that is shown on the "New…" dialog box.

```
QString CustomProjectWizard::trCategory() const
{
    return tr("VCreate Logic");
}
```

If the user selects the "CustomProject" category supported by our implementation of **`IWizard`** and selects Ok in the "New…" dialog box; then the **`runWizard()`** method is called. This method must be implemented to show a dialog box or **`QWizard`**, ask questions from the user about the new project/file/class being created and return a list of newly created files. In our implementation of the **`IWizard`** we will return an empty string list.

```
QStringList CustomProjectWizard::runWizard(const QString &path, QWidget *parent)
{
    Q_UNUSED(path);
    Q_UNUSED(parent);

    QMessageBox::information(parent, "Custom Wizard Dialog", "Hi there!");

    return QStringList();
}
```

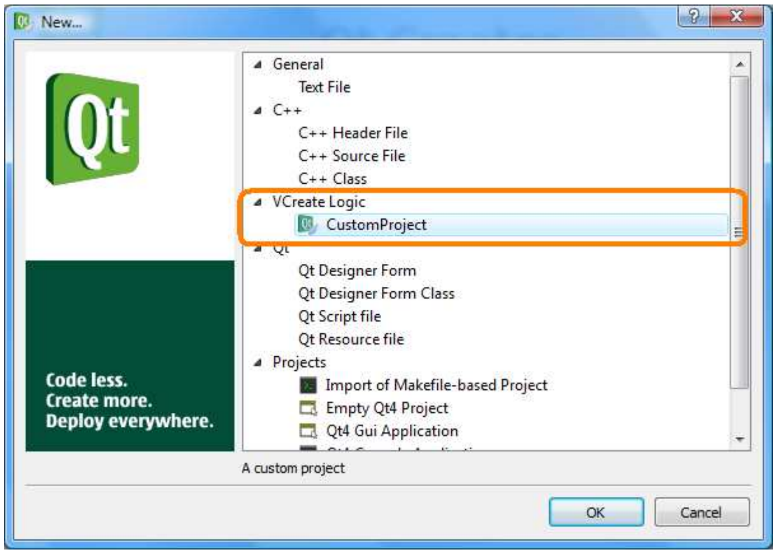
Step 2: Providing the wizard from a plugin

We implement a custom-project plugin using the same means as described in Chapter 2. The only change is in the **`initialize()`** method implementation of the plugin.

```
bool CustomProjectPlugin::initialize(const QStringList& args, QString *errMsg)
{
    Q_UNUSED(args);
    Q_UNUSED(errMsg);

    addAutoReleasedObject(new CustomProjectWizard);

    return true;
}
```

Step 3: Testing the plugin

Upon compiling the plugin and restarting Qt Creator, we can notice the new project type in the "New.." dialog box. Take a look at the screenshot below.



9.2 Predefined IWizard implementation – Core::BaseFileWizard

Qt Creator's core provides a default implementation of the **`IWizard`** interface in the form of the **`Core::BaseFileWizard`** class. This class implements provides default implementation of all the methods in the **`IWizard`** interface and adds some virtual methods of its own. To make use of the class, we need to subclass from it and implement one or more methods.

9.2.1 Core::GeneratedFile and Core::GeneratedFiles

Normally a new wizard (**`IWizard`** implementation) is implemented to allow the user to provide some hints and have one or more files automatically generated. The **`Core::GeneratedFile`** helps abstract each of the files that need generation. We will learn later on that within subclasses of **`Core::BaseFileWizard`**, we create an instance of **`Core::GeneratedFile`** for each file that is automatically generated.

The **`Core::GeneratedFile`** class is defined as follows in **`coreplugin/basefilewizard.h`**

```
class GeneratedFile
{
public:
    GeneratedFile();
    explicit GeneratedFile(const QString &path);
    GeneratedFile(const GeneratedFile &);
    GeneratedFile &operator=(const GeneratedFile &);
    ~GeneratedFile();

    QString path() const;
    void setPath(const QString &p);

    QString contents() const;
    void setContents(const QString &c);

    QString editorKind() const;
    void setEditorKind(const QString &k);

    bool write(QString *errorMessage) const;
private:
    QSharedDataPointer<GeneratedFilePrivate> m_d;
};

typedef QList<GeneratedFile> GeneratedFiles;
```

Files that need to be generated by subclasses of **`Core::BaseFileWizard`** are represented by the **`Core::GeneratedFile`** class. The class contains three key properties of a file that needs generation

1. Name of the file (with its absolute path).
2. The kind of editor needed for editing the file. Some valid values for editor kind are
   a. **`CppEditor::Constants::CPPEDITOR_KIND`**
   b. **`GenericProjectManager::Constants::PROJECT_KIND`**
   c. **`Git::Constants:: GIT_COMMAND_LOG_EDITOR_KIND`**
   d. **`Git::Constants:: C_GIT_COMMAND_LOG_EDITOR`**
3. Contents of the file.

Suppose that we wanted to generate a C++ source file with the following contents

```
#include <iostream>

int main()
{
    cout << "Hello World\n";

    return 0;
}
```

We would use **`Core::GeneratedFile`** for generating the above contents as follows

```
#include <coreplugin/basefilewizard.h>
#include <cppeditor/cppeditorconstants.h>

Core::GeneratedFile genFile("C:/Path/To/Source.cpp");
genFile.setEditorKind(CppEditor::Constants::CPPEDITOR_KIND);
genFile.setContents(
                        "#include <iostream>\n"
                        "\n"
                        "int main()\n"
                        "{\n"
                        "    cout << \"Hello World\n\";\n"
                        "    \n"
                        "    return 0;\n"
                        "}
                    );
genFile.write();
```

9.2.2 The "Item Model" class wizard

Suppose that we wanted to provide a new class wizard that helps automatically generate the skeleton of an item model based on few hints like

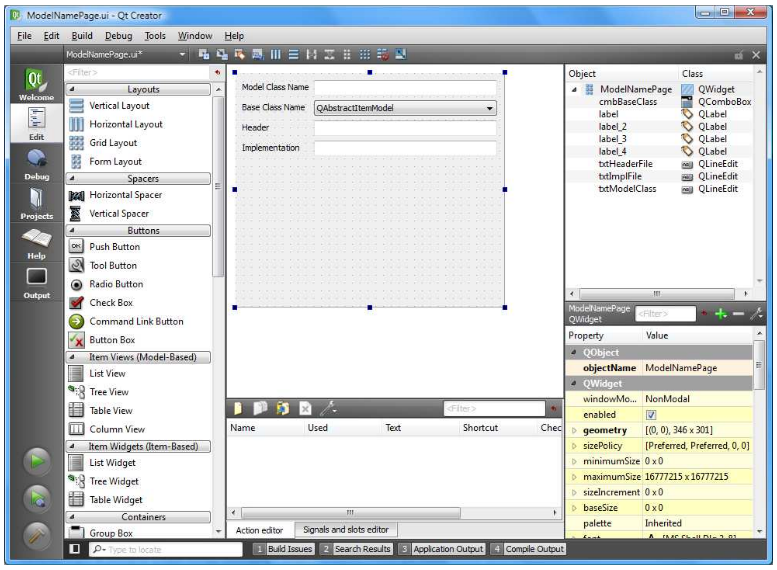
- Model Class Name

- Base Class Name (can be **`QAbstractItemModel`**, **`QAbstractListModel`** and **`QAbstractTableModel`**)
- Header file name and
- Source file name

Lets implement a plugin that will provide the new "Item Model" class wizard in Qt Creator.

Step 1: Design the class wizard page

Lets design a simple page in Qt Designer that accepts hints as described above.



Object	Class
ModelNamePage	QWidget
cmbBaseClass	QComboBox
label	QLabel
label_2	QLabel
label_3	QLabel
label_4	QLabel
txtHeaderFile	QLineEdit
txtImplFile	QLineEdit
txtModelClass	QLineEdit

ModelNamePage
QWidget

Property	Value
QObject	
objectName	ModelNamePage
QWidget	
windowMo...	NonModal
enabled	✓
geometry	[(0, 0), 346 x 301]
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited

The design is saved as ModelNamePage.ui.

Step 2: Implement the class wizard page

Lets import the UI in a Qt/C++ and provide easy to use methods to help fetch information from the page. First we design a structure that captures all the "item model" class hints.

```
struct ModelClassParameters
{
    QString className;
    QString headerFile;
    QString sourceFile;
    QString baseClass;
    QString path;
};
```

Next we declare a wizard page class that imports the UI designed in the previous step and provides methods to access the hints provided by the user in the page.

```
#include <QWizardPage>
#include "ui_ModelNamePage.h"

class ModelNamePage : public QWizardPage
{
    Q_OBJECT
public:
    ModelNamePage(QWidget* parent=0);
    ~ModelNamePage();

    void setPath(const QString& path);
    ModelClassParameters parameters() const;

private slots:
    void on_txtModelClass_textEdited(const QString& txt);

private:
    Ui::ModelNamePage ui;
    QString path;
};
```

The constructor and destructor are straight forward and easy to understand.

```
ModelNamePage::ModelNamePage(QWidget* parent)
:QWizardPage(parent)
{
    setTitle("Enter model class information");
    setSubTitle("The header and source file names will be derived from the class name");

    ui.setupUi(this);
}

ModelNamePage::~ModelNamePage()
{
}
```

The **setPath()** method basically stores the path in the private variable.

```
void ModelNamePage::setPath(const QString& path)
{
    this->path = path;
}
```

The **on_txtModelClass_textEdited()** slot computes the header and source file names based on the classname.

```
void ModelNamePage::on_txtModelClass_textEdited(const QString& txt)
{
    d->ui.txtHeaderFile->setText(txt + ".h");
    d->ui.txtImplFile->setText(txt + ".cpp");
}
```

Finally the **parameters()** method returns all the hints entered by the user in a **ModelClassParameters** instance.

```
ModelClassParameters ModelNamePage::parameters() const
{
    ModelClassParameters params;
    params.className = ui.txtModelClass->text();
    params.headerFile = ui.txtHeaderFile->text();
    params.sourceFile = ui.txtImplFile->text();
    params.baseClass = ui.cmbBaseClass->currentText();
    params.path = path;

    return params;
}
```

Step 3: Subclass Core::BaseFileWizard

The **`Core::BaseFileWizard`** class is defined as follows in **`coreplugin/basefilewizard.h`**

```
class CORE_EXPORT BaseFileWizard : public IWizard
{
    virtual ~BaseFileWizard();

    // IWizard
    virtual Kind kind() const;
    virtual QIcon icon() const;
    virtual QString description() const;
    virtual QString name() const;
    virtual QString category() const;
    virtual QString trCategory() const;
    virtual QStringList runWizard(const QString &path, QWidget *parent);

protected:
    typedef QList<QWizardPage *> WizardPageList;
    explicit BaseFileWizard(const BaseFileWizardParameters &parameters,
                            QObject *parent = 0);

    virtual QWizard *createWizardDialog(QWidget *parent,
                             const QString &defaultPath,
                             const WizardPageList &extensionPages) const = 0;

    virtual GeneratedFiles generateFiles(const QWizard *w,
                             QString *errorMessage) const = 0;

    virtual bool postGenerateFiles(const GeneratedFiles &l,
                             QString *errorMessage);
};
```

<u>Note: Some methods from the actual BaseFileWizard class are not shown here.</u>

The **`BaseFileWizard`** class implements the **`IWizard`** interface and offers three new functions

- **`createWizardDialog`** – This function can be over-ridden by subclasses to provide a wizard that the **`runWizard()`** method is supposed to show.
    - The **parent** parameter should be used as the parent widget of the returned QWizard
    - The **defaultPath** parameter should be the default location for generated files
    - The **extensionPages** parameter lists out all the pages that should be shown in the wizard by default.
- **`generateFiles`** – This method is called after the user is done with the wizard. Implementations of this method must create the required files as instances of **`Core::GeneratedFile`** class.
- **`postGenerateFiles`** – This method is called after **`generateFiles()`** returns. The default implementation opens the newly generated files; however subclasses can choose to do anything they want.

We subclass the BaseFileWizard as follows for our “item model” wizard

```
#include <coreplugin/basefilewizard.h>

class ModelClassWizard : public Core::BaseFileWizard
{
    Q_OBJECT

public:
    ModelClassWizard(const Core::BaseFileWizardParameters &parameters,
                     QObject *parent = 0);
    ~ModelClassWizard();

    QWizard *createWizardDialog(QWidget *parent,
                const QString &defaultPath,
                const WizardPageList &extensionPages) const;

    Core::GeneratedFiles generateFiles(const QWizard *w,
                QString *errorMessage) const;

private:
    QString readFile(const QString& fileName,
                const QMap<QString,QString>& replacementMap) const;
};
```

The constructor and destructor methods are straight forward and easy to understand.

```
ModelClassWizard::ModelClassWizard(
            const Core::BaseFileWizardParameters &parameters,
            QObject *parent)
            : Core::BaseFileWizard(parameters, parent)
{

}

ModelClassWizard::~ModelClassWizard()
{

}
```

The **`createWizardDialog()`** method is implemented to create a **`QWizard`** with its first page as the **`ModelNamePage`** class implemented step 2. Other default pages are added as usual.

```
QWizard* ModelClassWizard::createWizardDialog(
            QWidget *parent,
            const QString &defaultPath,
            const WizardPageList &extensionPages) const
{
    // Create a wizard
    QWizard* wizard = new QWizard(parent);
    wizard->setWindowTitle("Model Class Wizard");

    // Make our page as first page
    ModelNamePage* page = new ModelNamePage(wizard);
    int pageId = wizard->addPage(page);
    wizard->setProperty("_PageId_", pageId);
    page->setPath(defaultPath);

    // Now add the remaining pages
    foreach (QWizardPage *p, extensionPages)
        wizard->addPage(p);

    return wizard;
}
```

The **`readFile()`** method is implemented to read a file and return its contents as a string. Before returning the file's contents as string, the function uses the replacement table passed as second parameter to fix the string.

```
QString ModelClassWizard::readFile(const QString& fileName, const QMap<QString,QString>&
replacementMap) const
{
    QFile file(fileName);
    file.open(QFile::ReadOnly);

    QString retStr = file.readAll();

    QMap<QString,QString>::const_iterator it = replacementMap.begin();
    QMap<QString,QString>::const_iterator end = replacementMap.end();
    while(it != end)
    {
        retStr.replace(it.key(), it.value());
        ++it;
    }

    return retStr;
}
```

Suppose we have a file (**`sample.txt`**) whose contents are as follows

```
#ifndef {{UPPER_CLASS_NAME}}_H
#define {{UPPER_CLASS_NAME}}_H

#include <{{BASE_CLASS_NAME}}>

struct {{CLASS_NAME}}Data;
class {{CLASS_NAME}} : public {{BASE_CLASS_NAME}}
{
    Q_OBJECT

public:
    {{CLASS_NAME}}(QObject* parent=0);
    ~{{CLASS_NAME}}();

    int rowCount(const QModelIndex& parent) const;
    QVariant data(const QModelIndex& index, int role) const;

private:
    {{CLASS_NAME}}Data* d;
};
```

```
#endif // {{UPPER_CLASS_NAME}}_H
```

Lets say we wanted to replace the hints in {{xyz}} with something more appropriate, we could use the following code snippet.

```
QMap<QString,QString> replacementMap;
replacementMap["{{UPPER_CLASS_NAME}}"] = "LIST_MODEL";
replacementMap["{{BASE_CLASS_NAME}}"] = "QAbstractListModel";
replacementMap["{{CLASS_NAME}}"] = "ListModel";

QString contents = readFile("Sample.txt", replacementTable);
```

When the above code is executed, the ***contents*** string will contain

```
#ifndef LIST_MODEL_H
#define LIST_MODEL_H

#include <QAbstractListModel>

struct ListModelData;
class ListModel : public QAbstractListModel
{
    Q_OBJECT

public:
    ListModel(QObject* parent=0);
    ~ListModel();

    int rowCount(const QModelIndex& parent) const;
    QVariant data(const QModelIndex& index, int role) const;

private:
    ListModelData* d;
};

#endif // LIST_MODEL_H
```

Seems like magic isnt it? ☺. We create similar “template” header and source files for item, list and table model classes and create a resource for use in our project.

Now, lets look at the implementation of the **`generateFiles()`** method. This method basically creates two **`Core::GeneratedFile`** instances and populates them with appropriate data before returning them in a list.

```
Core::GeneratedFiles ModelClassWizard::generateFiles(
           const QWizard *w,
           QString *errorMessage) const
{
    Q_UNUSED(errorMessage);

    Core::GeneratedFiles ret;

    int pageId = w->property("_PageId_").toInt();
    ModelNamePage* page = qobject_cast<ModelNamePage*>(w->page(pageId));
    if(!page)
        return ret;

    ModelClassParameters params = page->parameters();

    QMap<QString,QString> replacementMap;
    replacementMap["{{UPPER_CLASS_NAME}}"] = params.className.toUpper();
    replacementMap["{{BASE_CLASS_NAME}}"] = params.baseClass;
    replacementMap["{{CLASS_NAME}}"] = params.className;
    replacementMap["{{CLASS_HEADER}}"] = QFileInfo(params.headerFile).fileName();

    Core::GeneratedFile headerFile(params.path + "/" + params.headerFile);
    headerFile.setEditorKind(CppEditor::Constants::CPPEDITOR_KIND);

    Core::GeneratedFile sourceFile(params.path + "/" + params.sourceFile);
    sourceFile.setEditorKind(CppEditor::Constants::CPPEDITOR_KIND);

    if(params.baseClass == "QAbstractItemModel")
    {
        headerFile.setContents(
              readFile(":/CustomProject/ItemModelHeader", replacementMap) );
        sourceFile.setContents(
              readFile(":/CustomProject/ItemModelSource", replacementMap) );
    }
    else if(params.baseClass == "QAbstractTableModel")
    {
        headerFile.setContents(
              readFile(":/CustomProject/TableModelHeader", replacementMap) );
        sourceFile.setContents(
              readFile(":/CustomProject/TableModelSource", replacementMap) );
    }
    else if(params.baseClass == "QAbstractListModel")
    {
        headerFile.setContents(
              readFile(":/CustomProject/ListModelHeader", replacementMap) );
        sourceFile.setContents(
              readFile(":/CustomProject/ListModelSource", replacementMap) );
    }

    ret << headerFile << sourceFile;

    return ret;
}
```

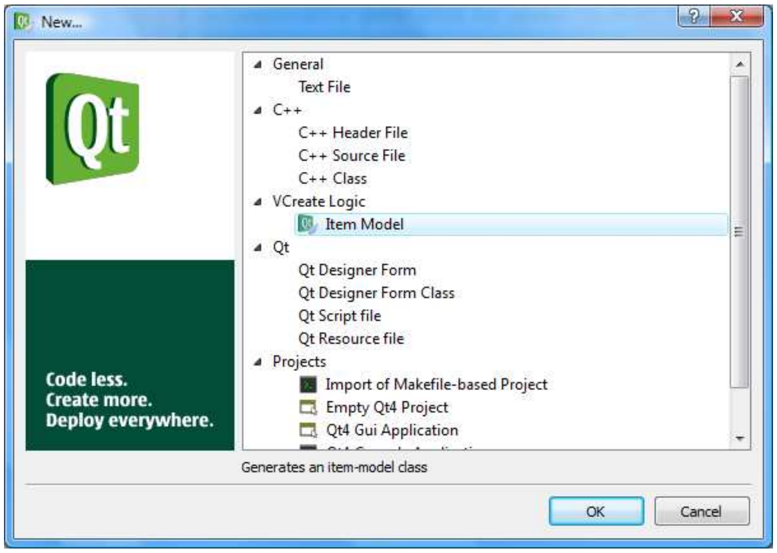
Step 4: Implementing the plugin

We implement the item model wizard plugin using the same means as described in Chapter 2. The only change is in the **`initialize()`** method implementation of the plugin.

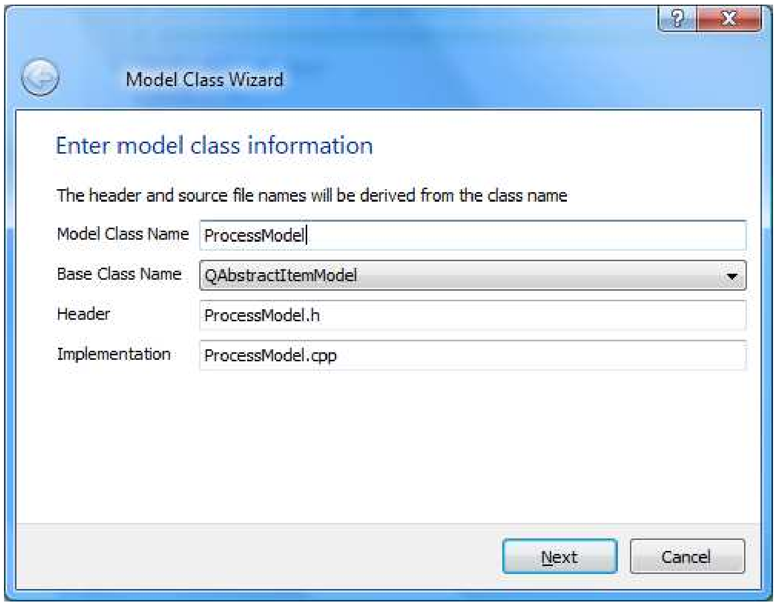
```
bool ItemModelWizard::initialize(const QStringList& args, QString *errMsg)
{
    Q_UNUSED(args);
    Q_UNUSED(errMsg);

    Core::BaseFileWizardParameters params;
    params.setKind(Core::IWizard::ClassWizard);
    params.setIcon(qApp->windowIcon());
    params.setDescription("Generates an item-model class");
    params.setName("Item Model");
    params.setCategory("VCreate Logic");
    params.setTrCategory(tr("VCreate Logic"));

    addAutoReleasedObject(new ModelClassWizard(params, this));

    return true;
}
```

Step 5: Testing the plugin

Upon compiling the plugin and restarting Qt Creator, we can notice the new project type in the "New.." dialog box. The following screenshots showcase the wizard that was just implemented.



Upon selecting the "Item Model" class wizard, we can see the ModelNamePage in a custom wizard.

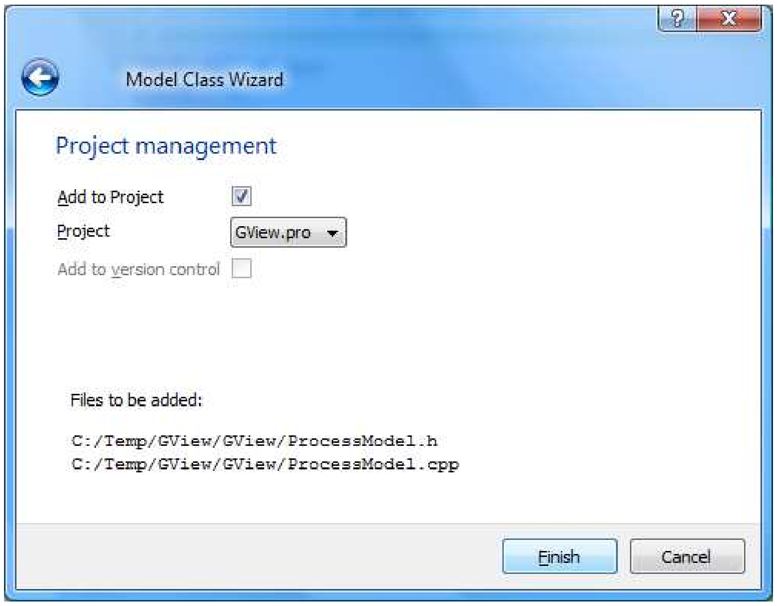


Enter model class information

The header and source file names will be derived from the class name

Model Class Name	ProcessModel
Base Class Name	QAbstractItemModel
Header	ProcessModel.h
Implementation	ProcessModel.cpp

We enter the appropriate details and click "Next". Qt Creator then shows us a built-in page to allow addition of the newly generated files into the current project.

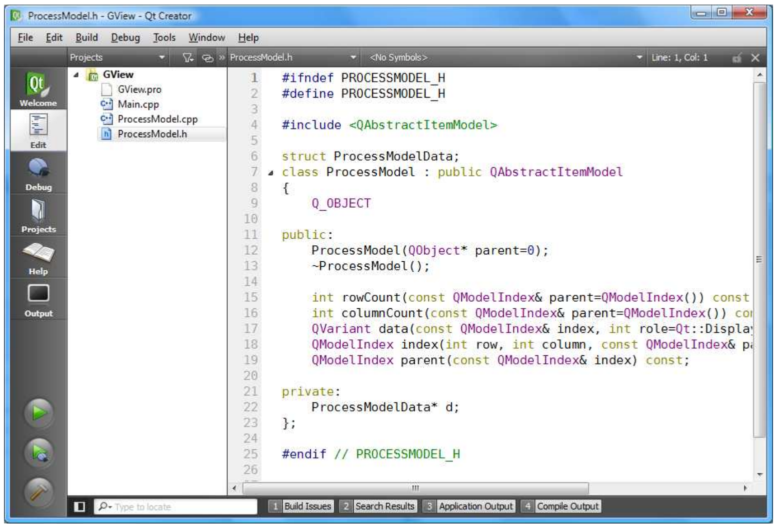


Add to Project ✓
Project: GView.pro
Add to version control

Files to be added:

```
C:/Temp/GView/GView/ProcessModel.h
C:/Temp/GView/GView/ProcessModel.cpp
```

Upon clicking "Finish", we can notice the newly generated files in the editor.



```
#ifndef PROCESSMODEL_H
#define PROCESSMODEL_H

#include <QAbstractItemModel>

struct ProcessModelData;
class ProcessModel : public QAbstractItemModel
{
    Q_OBJECT

public:
    ProcessModel(QObject* parent=0);
    ~ProcessModel();

    int rowCount(const QModelIndex& parent=QModelIndex()) const
    int columnCount(const QModelIndex& parent=QModelIndex()) co
    QVariant data(const QModelIndex& index, int role=Qt::Displa
    QModelIndex index(int row, int column, const QModelIndex& p
    QModelIndex parent(const QModelIndex& index) const;

private:
    ProcessModelData* d;
};

#endif // PROCESSMODEL_H
```