Qt Quick – Hybrid models

Timo Strömmer, Jan 10, 2011

Contents

<symbio>

- QML-C++ hybrids
  - Exporting objects and properties into QML
  - Writing QML plug-ins
- Qt Mobility
  - Integration with mobile peripherals
- QML-Web hybrids
  - Web browser integration
  - Web services

<symbio>

Simple C++ / QML integration example

C++/QML HYBRIDS

Hybrid programs

<symbio>

- Hybrid programs get the benefit from both worlds
  - Ease of QML / script programming
    - See for example *hellographics* vs. *HelloGraphicsQML*
  - Power and flexibility of C++
    - Access to all services provided by the platform
    - C++ performance with larger data-sets

QML/C++ hybrid

<symbio>

- A C++ GUI application may contain *QDeclarativeView* GUI components
  - Available for example via the GUI designer
  - Each runs it's own declarative engine
    - *qmlviewer* also runs one
  - Resource-wise it's not a good idea to run many views in a single program

QML/C++ hybrid

symbio

- *QDeclarativeView* has *setSource* function, which takes the URL of the QML file as parameter
  - Thus, can load also from web
- The QML files of the application can also be bundled into a resource file

QML/C++ exercise

<symbio>

- Create a new Qt GUI Project
  - Add a *QDeclarativeView* to the GUI form
  - Add *QT += declarative* to .pro file
    - Takes declarative QT module into use
  - Add a QML file to the project
    - Implement a GUI
  - Add a new *Qt resource file* to project
    - Add a / prefix
    - Add the QML file under the prefix

QML/C++ exercise

<symbio>

- Load the QML file from resource in the MainWindow constructor
- Build and run

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    ui->declarativeView->setSource(QUrl("qrc:/Main.qml"));
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

QML/C++ interaction

<symbio>

- To access the QML core from C++ side, the *QDeclarativeView* exposes a *root context*
  - *QDeclarativeContext* class
- A property can be set with *setContextProperty* function
  - Access normally by name in QML

```
Rectangle {
    width: 300
    height: 200

    Rectangle {
        x: 25; y: 25; widt
        color: rectColor
```

```
QDeclarativeContext *context = ui->declarativeView->rootContext();
context->setContextProperty("rectColor", QColor(Qt::blue));
```

rectColor becomes property of root QML element

Exporting objects to QML

<symbio>

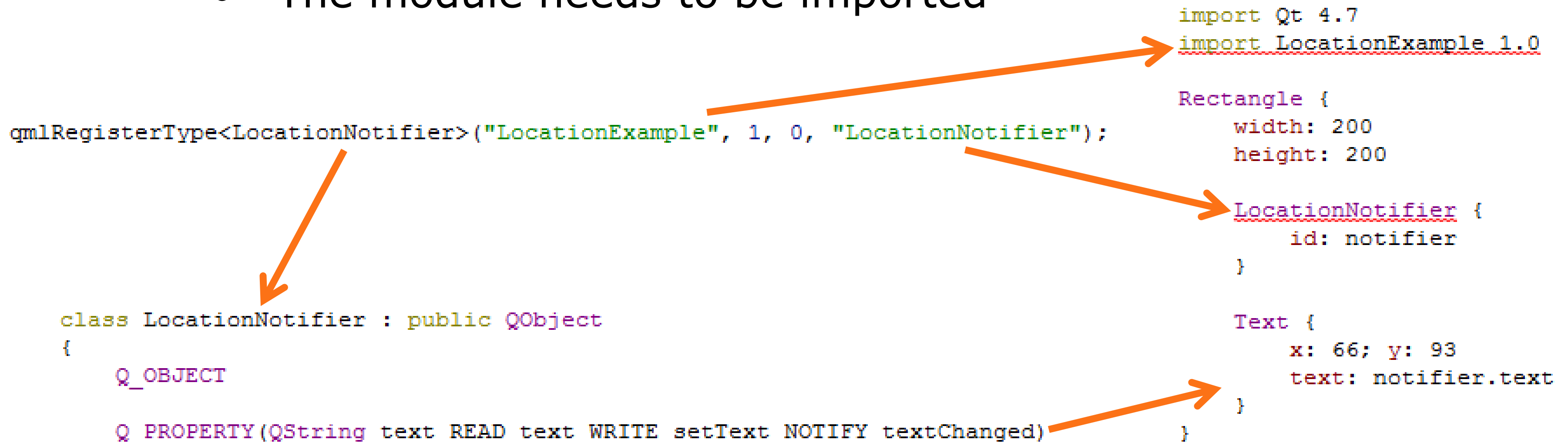
- Objects are registered with *qmlRegisterType* template function
  - Object class as template parameter
  - Function parameters:
    - *Module name*
    - Object version number (major, minor)
    - Name that is registered to QML runtime

Details about modules from:
http://doc.trolltech.com/4.7/qdeclarativemodules.html

```
qmlRegisterType<LocationNotifier>("LocationExample", 1, 0, "LocationNotifier");
```

Using exported classes

- The exported classes can be used as any other QML component
  - The module needs to be imported



QML object visibility

<symbio>

- Visibility at QML side
  - QObject *properties* become element properties
    - *on<Property>Changed* hook works if the NOTIFY signal is specified at C++ side
      - Also note that C++ signal name doesn't matter
  - QObject *signals* can be hooked with *on<Signal>*
  - QObject *slots* can be called as JS functions

<symbio>

QML plug-in projects

HYBRID PROGRAMMING

QML plug-ins

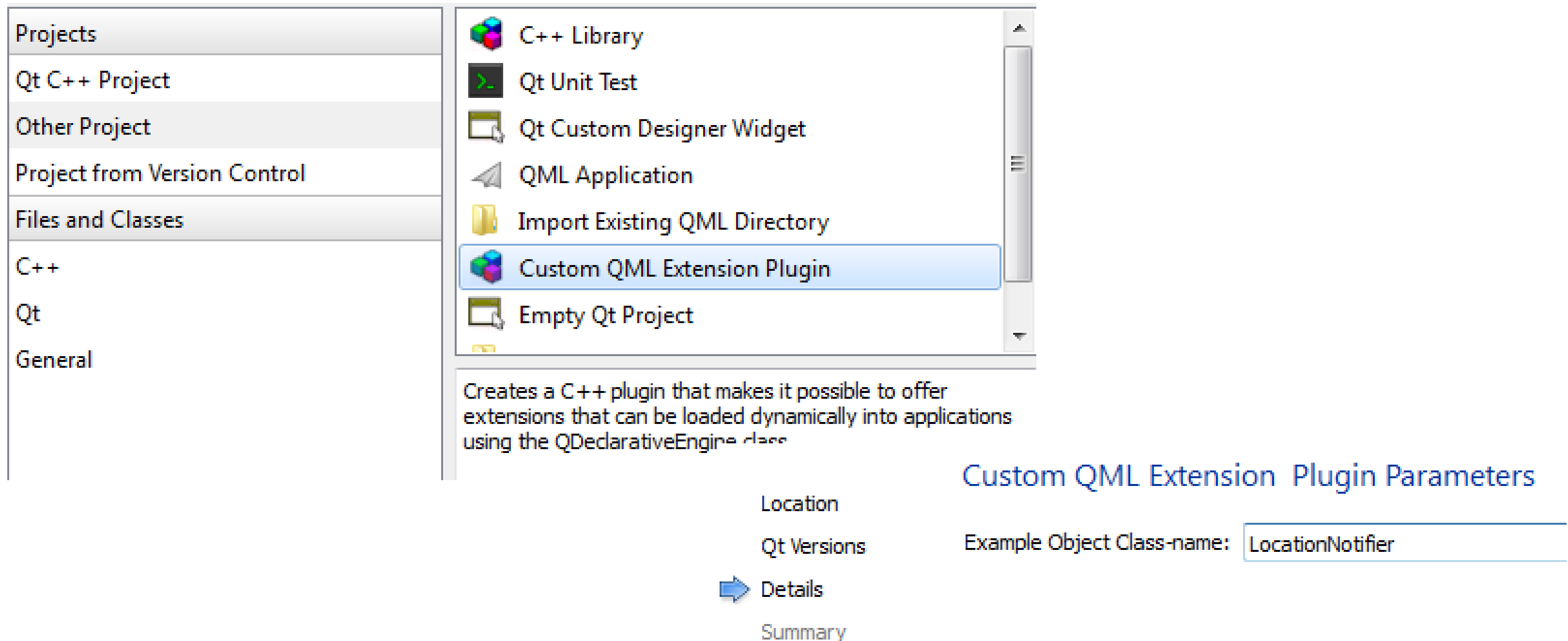
<symbio>

- A plug-in allows QML runtime to load Qt/C++ libraries
  - Thus, QML/C++ hybrid code can be run via *qmlviewer* or some other QML launcher application

Quick start

<symbio>

- Create a QML extension plug-in project
  - Wizard generates one *QObject*-based class



QML plug-in basics

<symbio>

- A QML plug-in library must have a class which extends *QDeclarativeExtensionPlugin*
  - Wizard generates

```
class mobilityplugin : public QDeclarativeExtensionPlugin
{
    Q_OBJECT

public:
    void registerTypes(const char *uri);
};
```

  - The plugin has *registerTypes* function, which is used to define components that are exported to the QML runtime

QML plug-in basics

<symbio>

- The API used by QML runtime to load the plug-in is created via preprocessor *macro*
  - *Q_EXPORT_PLUGIN* if plug-in project and class names are the same (wizard does that)
  - *Q_EXPORT_PLUGIN2* if names are different
    - *See qmlpluginexample* directory

```
Q_EXPORT_PLUGIN(mobilityplugin);
```

```
TEMPLATE = lib
TARGET   = mobilityplugin
QT += declarative
CONFIG += qt plugin
```

```
class mobilityplugin : public QDeclarativeExtensionPlugin
```

QML plug-in basics

<symbio>

- The plug-in must define a *qmldir* file
  - Describes the name of the plug-in to load
    - *libmobilityplugin.so* on Linux
    - *mobilityplugin.dll* on Windows
    - Optionally may specify a sub-directory

qmldir

```
1 plugin mobilityplugin
2
```

Quick start continued

<symbio>

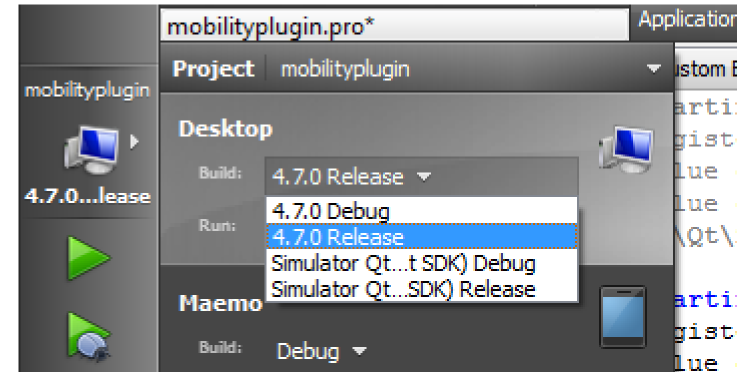
- Create a QML application project
  - Copy the *qmldir* file from the C++ plug-in into the application directory
- Edit the C++ plug-in *.pro* file
  - Add *DESTDIR* statement to point to the QML application directory

```
DESTDIR = ../MobilityExample/plugin
```

Quick start continued

<symbio>

- Switch to *Release* build
  - Fails with *Debug* libraries



- Build
  - The *.so* or *.dll* should be in the QML application directory where *qmldir* file also sits

Example plug-in

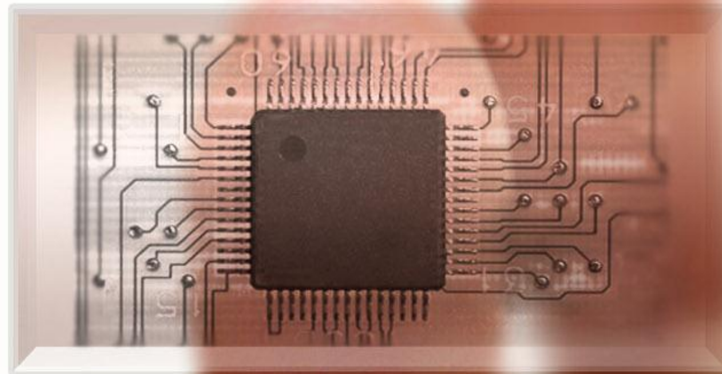
<symbio>

- See *PluginExample* and *qmlpluginexample* directories
    - Stores *TextInput* content into a file while user is typing
    - Uses *QSettings* API from Qt core
        - File in ~/.config/Symbio/QmlPluginExample.conf

First name: Foo

Last name: Bar

<symbio>



SERIOUS ABOUT SOFTWARE