Implementing Model-View-Presenter in Qt

Sam Protsenko November 8, 2010 at 01:36

Designing the architecture of one project, I settled on the MVP pattern - bribed the ability to easily change ui, as well as the ease of covering tests. All the examples of MVP implementation that I found on the network were in C #. When implementing on Qt a couple of non-obvious moments arose, the solution of which was successfully found. The information collected is below.

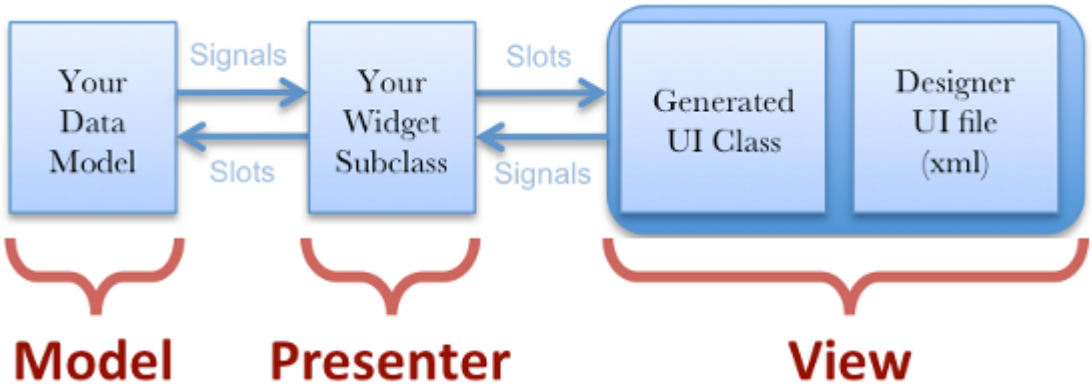
History of MVP

As follows from [1] and [2], the MVP design pattern is a modification of MVC, applied for the first time in the company IBM in the 90s of the last century when working on the object-oriented operating system Taligent. Later MVP described in detail Mike Potel.

To see what is the difference between MVP and MVC, you can see the relevant paragraphs [3]: MVC , MVP

Implementing MVP in Qt

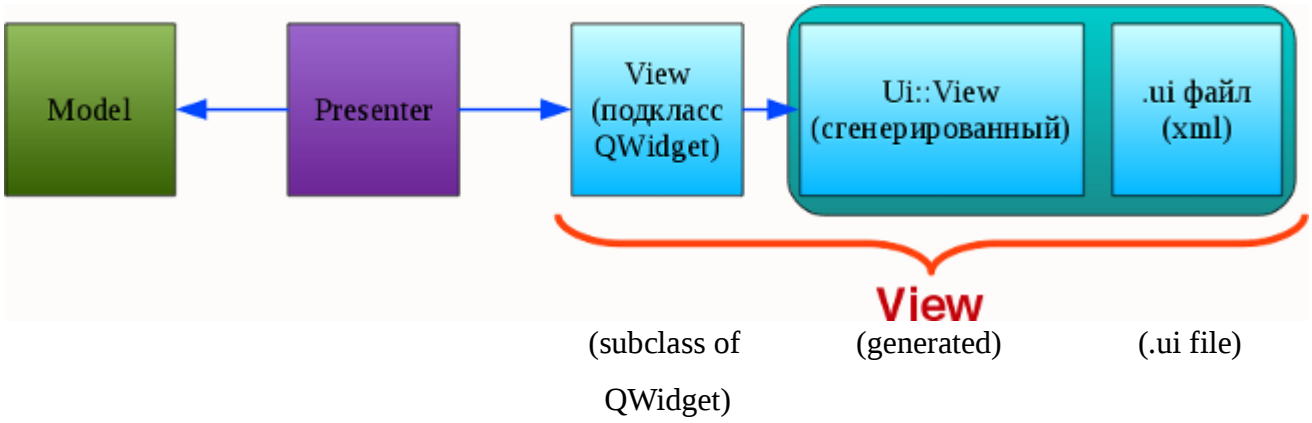
In general, MVP is already used in Qt in implicit form, as it is shown in [4]:



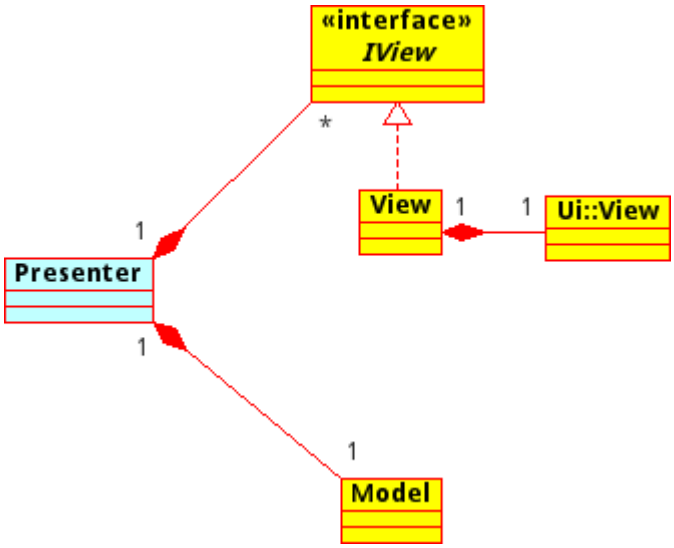
But such an MVP implementation has a number of drawbacks:

- It is impossible to create multiple views for one representative (Presenter)
- The *Inversion of Control* pattern described in [3] (here) can not be used, because In this schema, the View is generated automatically and can not be inherited from the interface

The full MVP implementation will look like this:



Hence the class diagram:



Consider each class:

- **IView** is an interface class that defines the methods and signals that a particular View must implement (see *Inversion of Control* in [3])
- **Ui :: View** - the class generated by the .ui file of the Qt designer
- **View** is a concrete representation inherited from QWidget (or its descendant) and from IView. It contains GUI logic, i.e. the behavior of objects (for example, their animation)
- **Model** - model of domain data (Domain Model); contains variables, flags, table models, etc.
- **Presenter** - representative; realizes the interaction between the model and the view. It also interacts with the outside world through it. with the main application, with the server (for example, through the application services layer), etc.

At the stage of "drawing squares" everything is clear. I had problems when trying to implement.

1. When writing IView, you need to declare signals. But for this, IView must be inherited from QObject. Then, when trying to inherit View simultaneously from QWidget and IView, an error occurred. It turned out that the class can not be inherited simultaneously from two QObject-objects (even virtual inheritance did not help). How to get around this problem is shown in [5]: reference . Thus, IView does not inherit from QObject and simply declares signals as fully virtual (abstract) methods in the public section. This is quite logical, because the signal is also a function, on the call of which the observers are notified through their slots (see the Observer pattern).
2. Another problem occurred when binding IView signals in the Presenter class. The fact is that Presenter contains a link to IView (a particular View is added to Presenter at runtime, but stored as IView - polymorphism is used). But to connect signals and slots, the static `QObject::connect()` method is used, which accepts only the descendants of QObject as objects, and IView is not. But we know that any of our View will be there. So the problem is solved by dynamic type reduction, as shown in [6]:

```
// where m_view is IView
QObject *view_obj = dynamic_cast<QObject *>(m_view);
QObject::connect(view_obj, SIGNAL(okActionTriggered()),
                 this, SLOT(processOkAction()));
```

It's also worth noting that when implementing IView, only the header (.h) file is needed. If you created a. Cpp file for IView, you must delete it, otherwise problems may arise during compilation.

In principle, this information is enough to independently implement MVP in Qt, but I'll give a simple example (only the implementation of MVP, without considering the interaction in the context of the real application).

A simple example of using MVP in Qt

In the archive 3 examples, this is the same application, but with additions in each example.

1. Implementing MVP
2. Added the ability to create multiple views
3. Full synchronization between views when changing data

All code is commented in Doxygen-style, i.e. you can easily generate documentation using Doxywizard.

Download examples here

Out of scope

Here is a list of interesting questions on the topic that are not included in the article:

- testing of modules of the received system
- the implementation of the View as libraries (using QtPlugin) in the MVP context
- View implementation with QtDeclarative (QML) in MVP context
- transfer submissions to Presenter through a factory class (so you can easily change the styles)

Comments are welcome on any information on these issues.

Sources of information

1. http://www.rsdn.ru/article/patterns/generic-mvc2.xml
2. http://en.wikipedia.org/wiki/Model-view-presenter
3. http://www.rsdn.ru/article/patterns/ModelViewPresenter.xml
4. http://thesmithfam.org/blog/2009/09/27/model-view-presenter-and-qt/
5. http://doc.trolltech.com/qq/qq15-academic.html
6. http://developer.qt.nokia.com/forums/viewthread/284