MVC design with Qt Designer and PyQt / PySide

Asked 7 years, 7 months ago Modified 10 months ago Viewed 24k times

35

40

Python newbie coming from Java (+SWT/Windowbuilder) and am having difficulty working out how to properly code a large desktop app in Python/Qt4(QtDesigner)/PySide.

I would like to keep any view logic in a controller class outside the .ui file (and it's .py conversion). Firstly as then the logic is independent of GUI framework and secondly as the .ui and resultant .py file get overwritten on any changes!.

Only examples I've found add action code to a monolithic MainWindow.py (generated from ui) or a MyForm.py (also generated from .ui). I can't see any way to link a POPO controller class to actions in QtDesigner.

Can anyone point me to workflows for creating a large scale application using QtDesigner in a scalable MVC/P methodology?

python model-view-controller pyqt pyside qt-designer

Share Improve this question Follow

edited Oct 10, 2018 at 8:06

101
7,746 3 37 64

asked Nov 2, 2014 at 11:09

Don Smythe
8,076 14 58 100

1 Answer

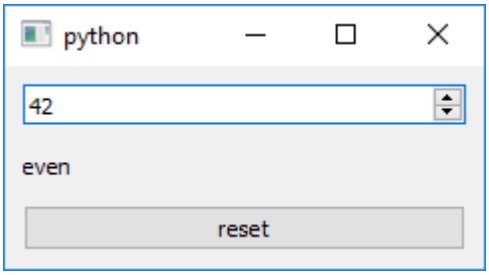
Sorted by:

Highest score (default)

82

Firstly, just be aware that Qt already uses the concept of views and models but that's not actually what you're after. In short the Qt MCV concept is a way to automatically link a widget (e.g. a QListView) to a data source (e.g. a QStringListModel) so that changes to the data in the model automagically appear in the widget and vice versa. This is a (very) useful feature but it's a different thing to an application scale MVC design pattern. The two concepts can be used together of course and that does offer some obvious shortcuts. Application scale MVC design however must be manually programmed.

Here's an example MVC application that has a single view, controller, and model. The view has 3 widgets that each independently listen for and react to changes to data in the model. The spin box and button can both manipulate data in the model via the controller.



The file structure is arranged like this:

```
project/
    mvc_app.py              # main application with App class
    mvc_app_rc.py           # auto-generated resources file (using pyrcc.exe or
equivalent)
    controllers/
        main_ctrl.py        # main controller with MainController class
        other_ctrl.py
    model/
        model.py            # model with Model class
    resources/
        mvc_app.qrc         # Qt resources file
        main_view.ui        # Qt designer files
        other_view.ui
        img/
            icon.png
    views/
        main_view.py        # main view with MainView class
        main_view_ui.py     # auto-generated ui file (using pyuic.exe or
equivalent)
        other_view.py
        other_view_ui.py
```

Application

`mvc_app.py` would be responsible for instantiating each of the view, controllers, and model(s) and passing references between them. This can be quite minimal:

```
import sys
from PyQt5.QtWidgets import QApplication
from model.model import Model
from controllers.main_ctrl import MainController
from views.main_view import MainView


class App(QApplication):
    def __init__(self, sys_argv):
        super(App, self).__init__(sys_argv)
        self.model = Model()
        self.main_controller = MainController(self.model)
        self.main_view = MainView(self.model, self.main_controller)
        self.main_view.show()


if __name__ == '__main__':
    app = App(sys.argv)
    sys.exit(app.exec_())
```

Views

Use Qt designer to create the .ui layout files to the extent that you assign variables names to widgets and adjust their basic properties. Don't bother adding signals or slots as it's generally easier just to connect them to functions from within the view class.

The .ui layout files are converted to .py layout files when processed with pyuic or pyside-uic. The .py view files can then import the relevant auto-generated classes from the .py layout files.

The view class(es) should contain the minimal code required to connect to the signals coming from the widgets in your layout. View events can call and pass basic information to a method in the view class and onto a method in a controller class, where any logic should be. It would look something like:

```
from PyQt5.QtWidgets import QMainWindow
from PyQt5.QtCore import pyqtSlot
from views.main_view_ui import Ui_MainWindow


class MainView(QMainWindow):
    def __init__(self, model, main_controller):
        super().__init__()

        self._model = model
        self._main_controller = main_controller
        self._ui = Ui_MainWindow()
        self._ui.setupUi(self)

        # connect widgets to controller

self._ui.spinBox_amount.valueChanged.connect(self._main_controller.change_amount)
        self._ui.pushButton_reset.clicked.connect(lambda:
self._main_controller.change_amount(0))

        # listen for model event signals
        self._model.amount_changed.connect(self.on_amount_changed)
        self._model.even_odd_changed.connect(self.on_even_odd_changed)
        self._model.enable_reset_changed.connect(self.on_enable_reset_changed)

        # set a default value
        self._main_controller.change_amount(42)

    @pyqtSlot(int)
    def on_amount_changed(self, value):
        self._ui.spinBox_amount.setValue(value)

    @pyqtSlot(str)
    def on_even_odd_changed(self, value):
        self._ui.label_even_odd.setText(value)

    @pyqtSlot(bool)
    def on_enable_reset_changed(self, value):
        self._ui.pushButton_reset.setEnabled(value)
```

The view doesn't do much apart from link widget events to the relevant controller function, and listen for changes in the model, which are emitted as Qt signals.

Controllers

The controller class(es) perform any logic and then sets data in the model. An example:

```
from PyQt5.QtCore import QObject, pyqtSlot


class MainController(QObject):
    def __init__(self, model):
        super().__init__()

        self._model = model

    @pyqtSlot(int)
    def change_amount(self, value):
        self._model.amount = value

        # calculate even or odd
        self._model.even_odd = 'odd' if value % 2 else 'even'

        # calculate button enabled state
        self._model.enable_reset = True if value else False
```

The `change_amount` function takes the new value from the widget, performs logic, and sets attributes on the model.

Model

The model class stores program data and state and some minimal logic for announcing changes to this data. This model shouldn't be confused with the Qt model (see http://qt-project.org/doc/qt-4.8/model-view-programming.html) as it's not really the same thing.

The model might look like:

```
from PyQt5.QtCore import QObject, pyqtSignal


class Model(QObject):
    amount_changed = pyqtSignal(int)
    even_odd_changed = pyqtSignal(str)
    enable_reset_changed = pyqtSignal(bool)

    @property
    def amount(self):
        return self._amount

    @amount.setter
    def amount(self, value):
        self._amount = value
        self.amount_changed.emit(value)
```

```
    @property
    def even_odd(self):
        return self._even_odd

    @even_odd.setter
    def even_odd(self, value):
        self._even_odd = value
        self.even_odd_changed.emit(value)

    @property
    def enable_reset(self):
        return self._enable_reset

    @enable_reset.setter
    def enable_reset(self, value):
        self._enable_reset = value
        self.enable_reset_changed.emit(value)

    def __init__(self):
        super().__init__()

        self._amount = 0
        self._even_odd = ''
        self._enable_reset = False
```

Writes to the model automatically emit signals to any listening views via code in the `setter` decorated functions. Alternatively the controller could manually trigger the signal whenever it decides.

In the case where Qt model types (e.g. QStringListModel) have been connected with a widget then the view containing that widget does not need to be updated at all; this happens automatically via the Qt framework.

UI source file

For completion, the example `main_view.ui` file is included here:

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>93</width>
    <height>86</height>
   </rect>
  </property>
  <widget class="QWidget" name="centralwidget">
   <layout class="QVBoxLayout">
    <item>
     <widget class="QSpinBox" name="spinBox_amount"/>
    </item>
    <item>
     <widget class="QLabel" name="label_even_odd"/>
    </item>
```

```
      <item>
       <widget class="QPushButton" name="pushButton_reset">
        <property name="enabled">
         <bool>false</bool>
        </property>
       </widget>
      </item>
     </layout>
    </widget>
   </widget>
   <resources/>
   <connections/>
  </ui>
```

It is converted to `main_view_ui.py` by calling:

```
pyuic5 main_view.ui -o ..\views\main_view_ui.py
```

The resource file `mvc_app.qrc` is converted to `mvc_app_rc.py` by calling:

```
pyrcc5 mvc_app.qrc -o ..\mvc_app_rc.py
```

Interesting links

[Why Qt is misusing model/view terminology?]

Share Improve this answer Follow edited Aug 4, 2021 at 23:31 answered Nov 2, 2014 at 12:09

101
7,746 3 37 64

2 OK I am beginning to see now how things fit together. The way I see it as set up - the main_view.py is the glue layer that links the model to the ui_main_view.py and signals to main_cont.py. This is what I couldn't understand how to do with qtDesigner. Thanks for the detailed explanation. – Don Smythe Nov 3, 2014 at 12:40

2 Added some code examples and a link to a code generator that can create most of the code automatically. – 101 Dec 23, 2014 at 6:52

1 Just don't create an `Ui_MainWindow()` object, and build the widgets in the view manually (as shown in many tutorials). – 101 Oct 11, 2019 at 1:14

1 I haven't use the State Machine API, but I can't see why it wouldn't be possible to use them together. – 101 Oct 11, 2019 at 1:16

1 @thinwybk that's getting a little complicated, could you please ask a new question? – 101 Oct 15, 2019 at 21:55