PyQt with Qt Designer Demo

We will look at:

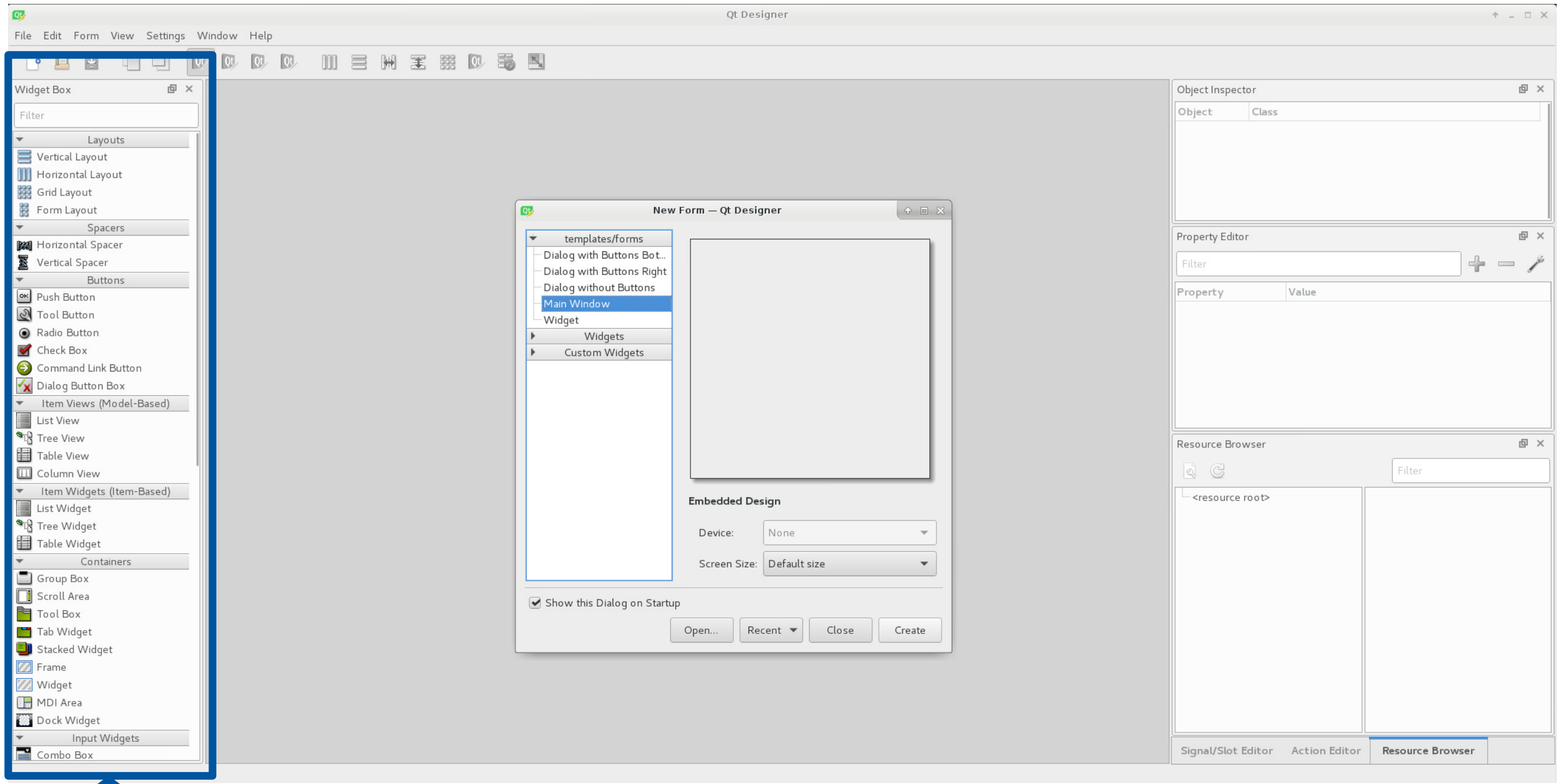
- **Using the Qt Designer**
  Approaches for using the automatically generated UI code from the designer

- **Adding Logic**
  Architectures for adding logic to our Widgets

Qt Designer

Design the look of the GUI and automatically generate the Python code.

Qt Designer can be extended by writing plugins (in C++ or Python)



A plugin is used to expose a custom widget to Designer so that it appears in Designer’s widget box just like any other widget.

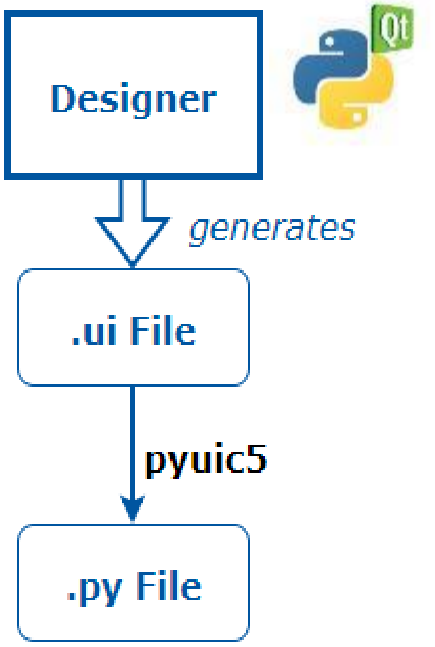
Using the Qt Designer

Designing small, reusable UI widgets

1. Use the Qt Designer to design the visual part of the widget (without logic).

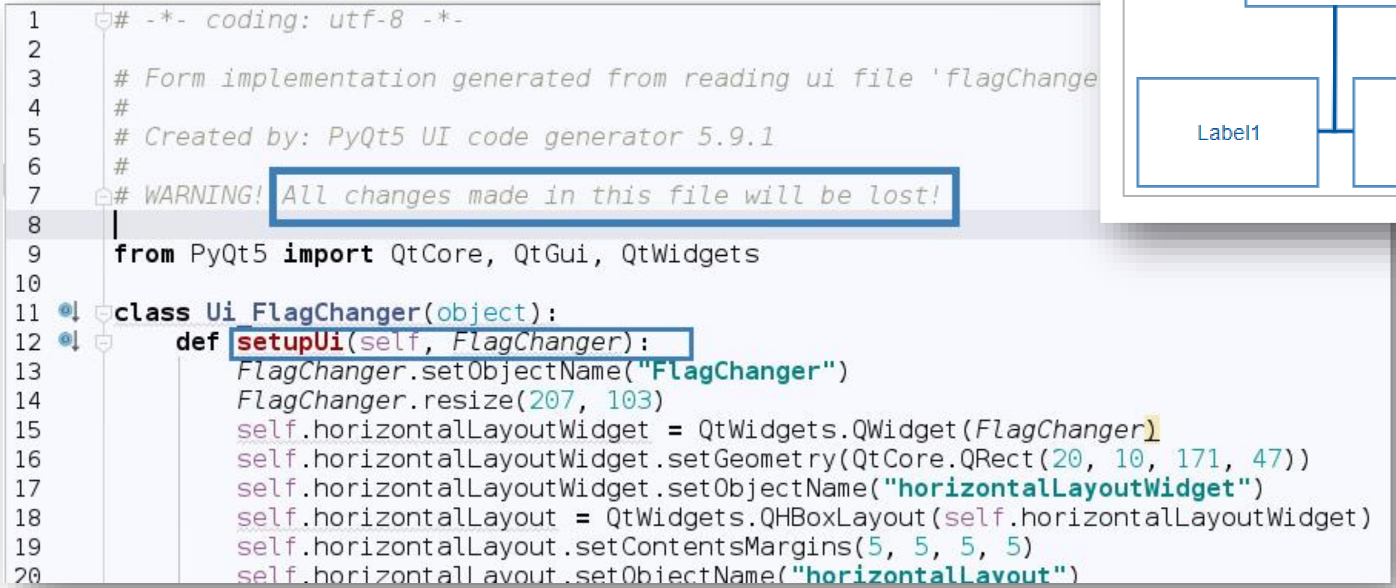
2. Using the **"pyuic5"** command transform the generated XML .ui file into python code.

3. Extend generated code to add logic to the widget using signals and slots mechanism.

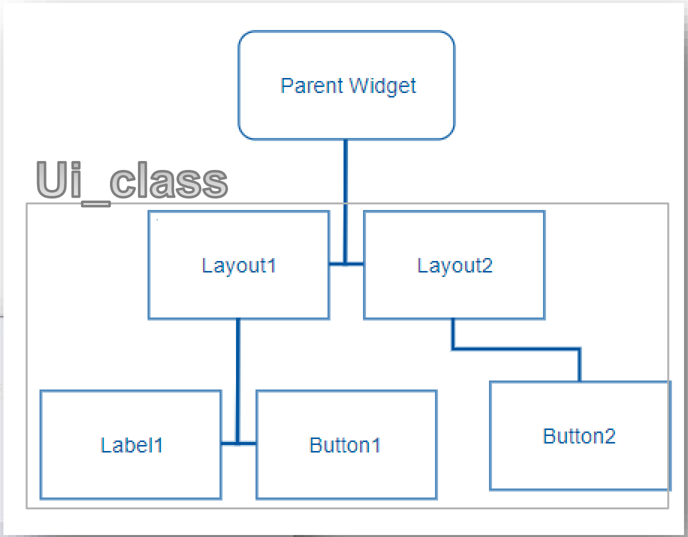


Automatically Generated Ui.py Code Structure

- The code is structured as a single class that is derived from the Python object type.
- The class contains a method called:
  *setupUi(self, **parentWidget**)*
  which builds the widget tree from the parent widget.
- This file should only be imported, never edited.

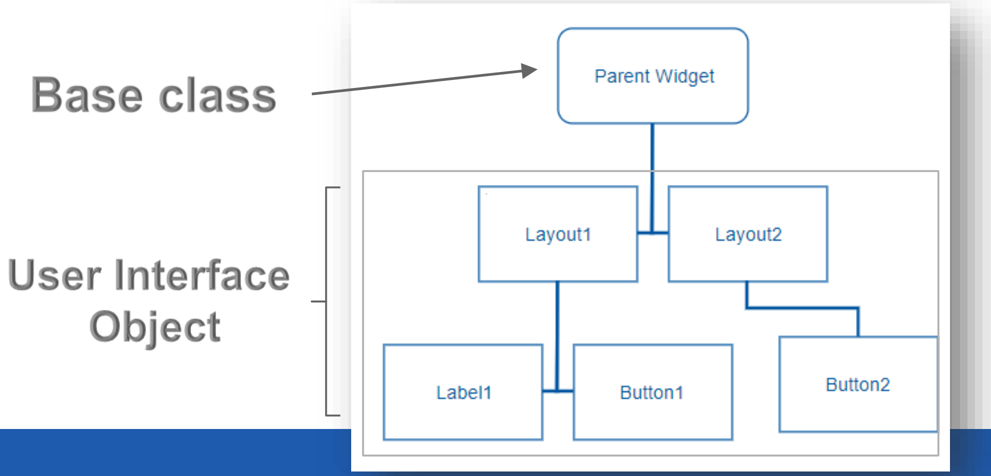


```
# -*- coding: utf-8 -*-

# Form implementation generated from reading ui file 'flagChange

#
# Created by: PyQt5 UI code generator 5.9.1
#
# WARNING! All changes made in this file will be lost!

from PyQt5 import QtCore, QtGui, QtWidgets

class Ui_FlagChanger(object):
    def setupUi(self, FlagChanger):
        FlagChanger.setObjectName("FlagChanger")
        FlagChanger.resize(207, 103)
        self.horizontalLayoutWidget = QtWidgets.QWidget(FlagChanger)
        self.horizontalLayoutWidget.setGeometry(QtCore.QRect(20, 10, 171, 47))
        self.horizontalLayoutWidget.setObjectName("horizontalLayoutWidget")
        self.horizontalLayout = QtWidgets.QHBoxLayout(self.horizontalLayoutWidget)
        self.horizontalLayout.setContentsMargins(5, 5, 5, 5)
        self.horizontalLayout.setObjectName("horizontalLayout")
```



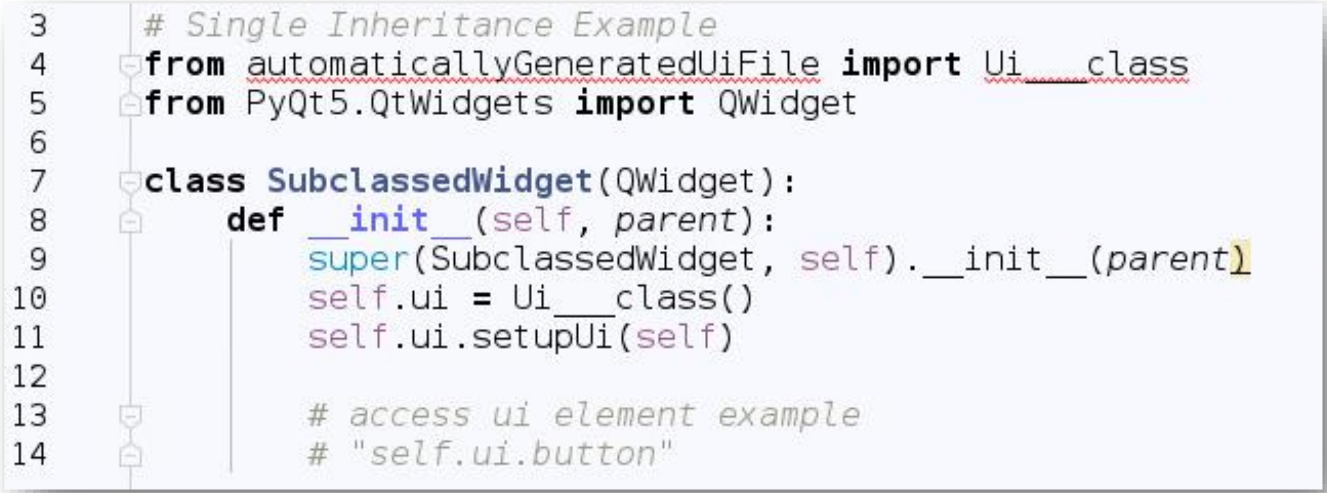
3 Approaches for the Ui Form Class

1. Direct Approach
   Construct a widget to use as a placeholder for the component, and set up the user interface inside it
2. Single Inheritance Approach
   Subclass the form's base class (e.g. Qwidget, QDialog) and include a private instance of the form's interface object.
3. Multiple Inheritance Approach
   subclass both the form's base class and the form's user interface object.



Single Inheritance Approach

Subclass a Qt widget, and set up the User Interface from within the constructor.



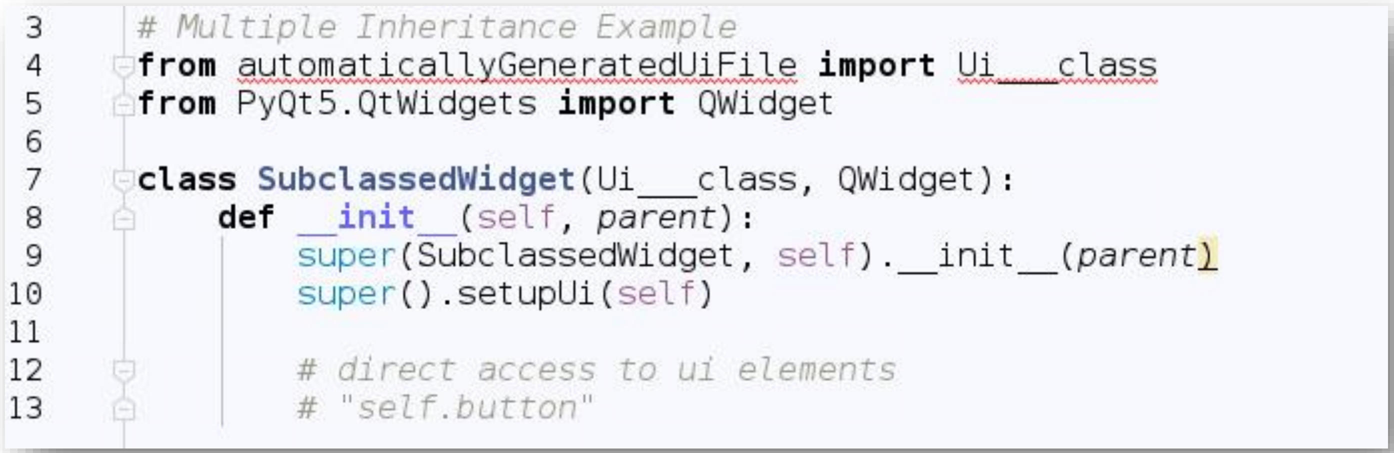
```python
# Single Inheritance Example
from automaticallyGeneratedUiFile import Ui___class
from PyQt5.QtWidgets import QWidget

class SubclassedWidget(QWidget):
    def __init__(self, parent):
        super(SubclassedWidget, self).__init__(parent)
        self.ui = Ui___class()
        self.ui.setupUi(self)

        # access ui element example
        # "self.ui.button"
```

- ✓ Expose the widgets and layouts used in the form to the Qt widget subclass, providing a **standard system for making signal and slot connections** between the user interface and other objects in your application

- ✓ Encapsulation of the user interface widget variables within the "ui" data member

Multiple Inheritance Approach

Subclass both the form’s base class and the form’s user interface object.



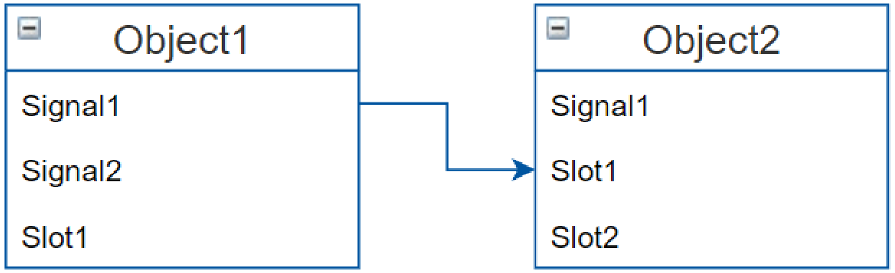
```python
# Multiple Inheritance Example
from automaticallyGeneratedUiFile import Ui___class
from PyQt5.QtWidgets import QWidget

class SubclassedWidget(Ui___class, QWidget):
    def __init__(self, parent):
        super(SubclassedWidget, self).__init__(parent)
        super().setupUi(self)

        # direct access to ui elements
        # "self.button"
```

- ✓ This allows the widgets in the form to be used directly from within the scope of the subclass
- ✓ Direct creation of Signals, Slots and Connections

Adding the logic

Signals and Slots are used for communication between objects

- **Signal** = is emitted by a Qt Object when a particular event is fired
- **Slot** = a Python callable (function) that is called in response to a particular signal



```
object1.signal1.connect( object2.slot1 )
```

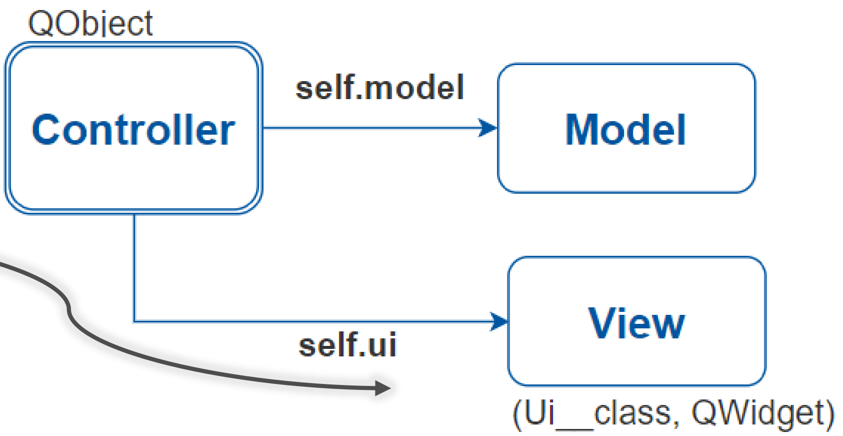
```
button.clicked.connect(self.slot_method)
```

MVC Architecture

The **Controller** is a separate QObject:

- Inside the Controller we have the Model and the View
- We are obliged to use multiple inheritance for the View Class

To add the widget to a larger application we need to access the view class

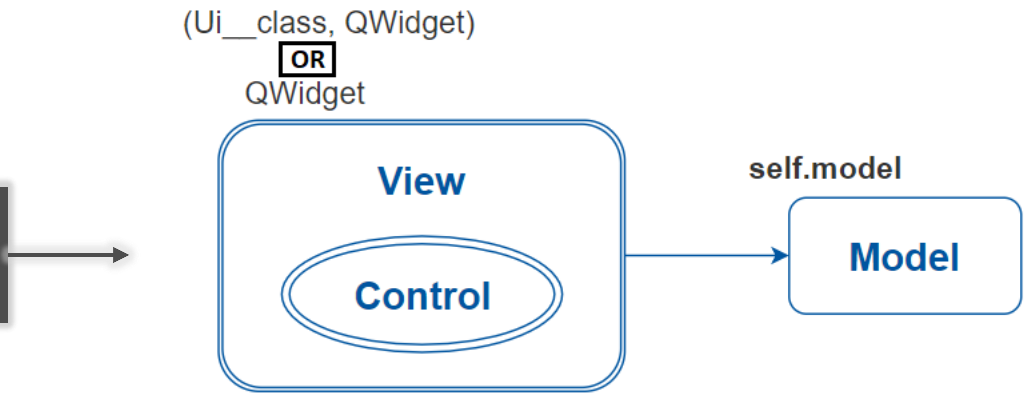


MV Architecture – The Qt Way

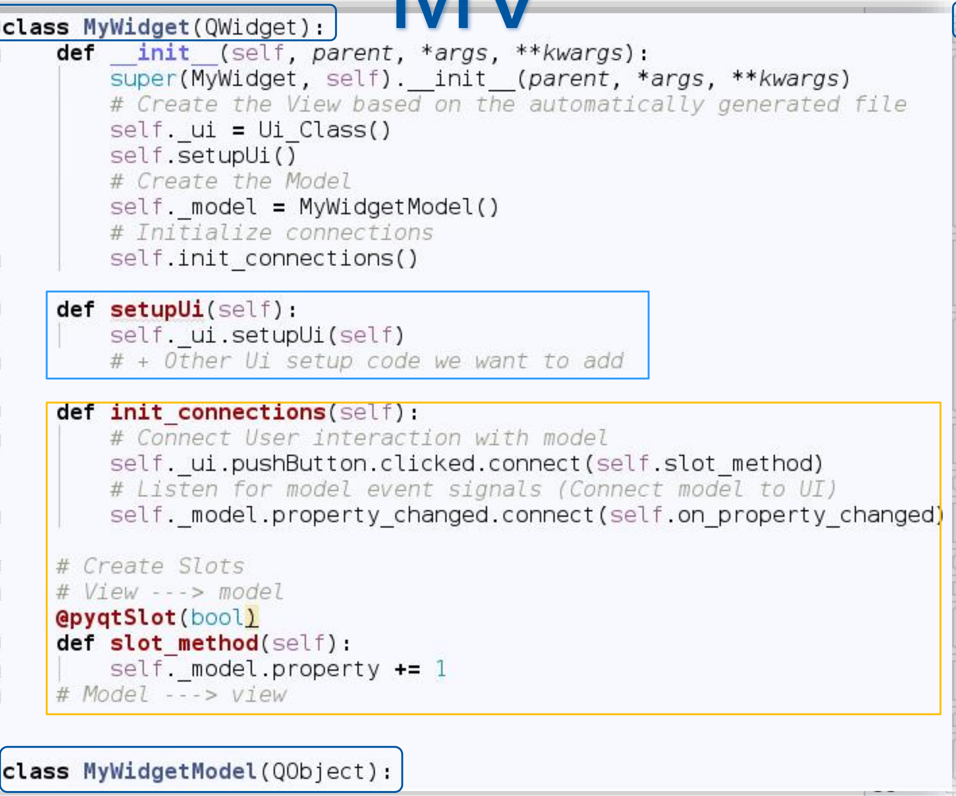
The logic is part of the View

- Connections of signals to slots happen inside the View Class
- Can be used with single or multiple inheritance

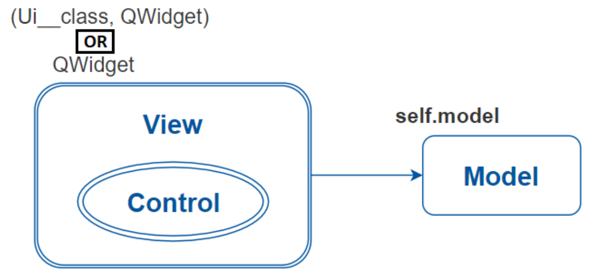
Can be used directly to compose larger UIs



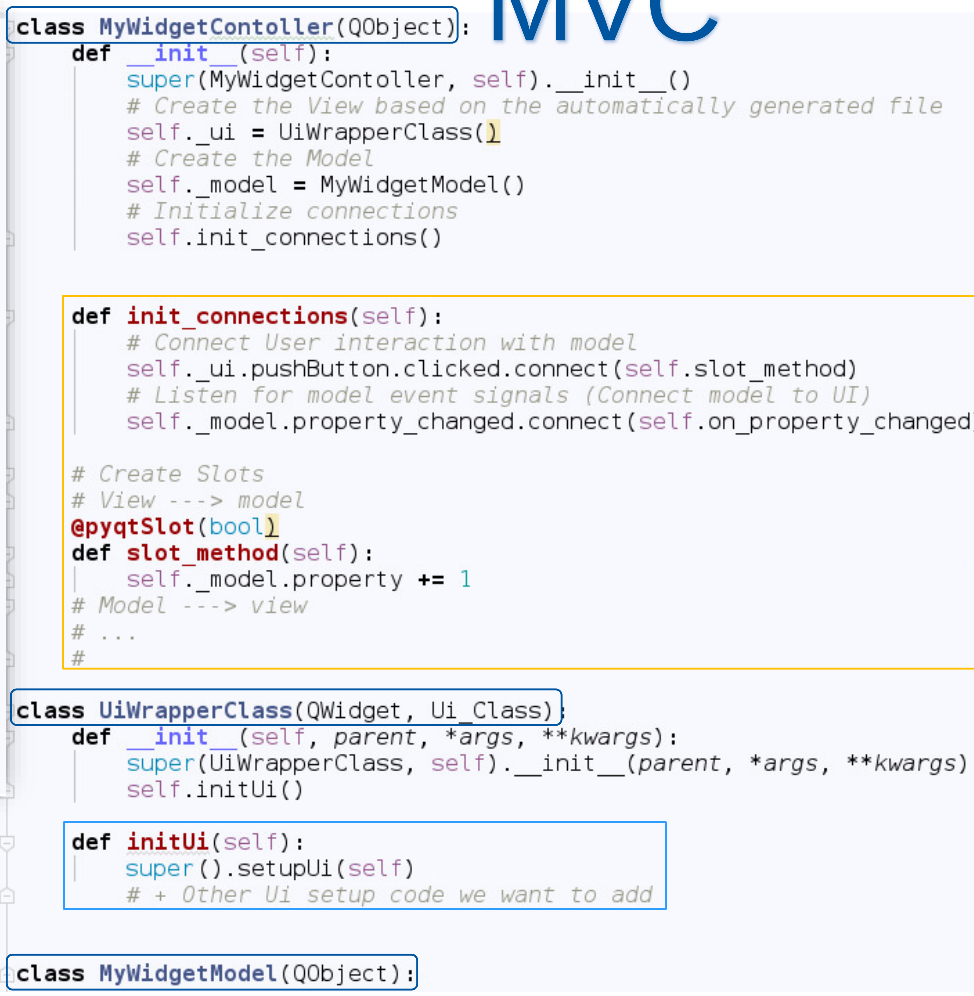
MV



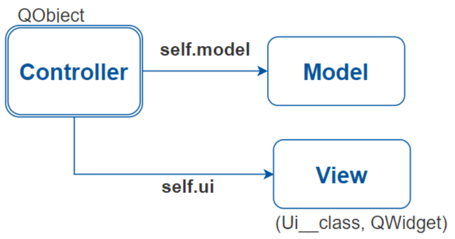
```
class MyWidget(QWidget):
    def __init__(self, parent, *args, **kwargs):
        super(MyWidget, self).__init__(parent, *args, **kwargs)
        # Create the View based on the automatically generated file
        self._ui = Ui_Class()
        self.setupUi()
        # Create the Model
        self._model = MyWidgetModel()
        # Initialize connections
        self.init_connections()

    def setupUi(self):
        self._ui.setupUi(self)
        # + Other Ui setup code we want to add

    def init_connections(self):
        # Connect User interaction with model
        self._ui.pushButton.clicked.connect(self.slot_method)
        # Listen for model event signals (Connect model to UI)
        self._model.property_changed.connect(self.on_property_changed)

    # Create Slots
    # View ---> model
    @pyqtSlot(bool)
    def slot_method(self):
        self._model.property += 1
    # Model ---> view

class MyWidgetModel(QObject):
```



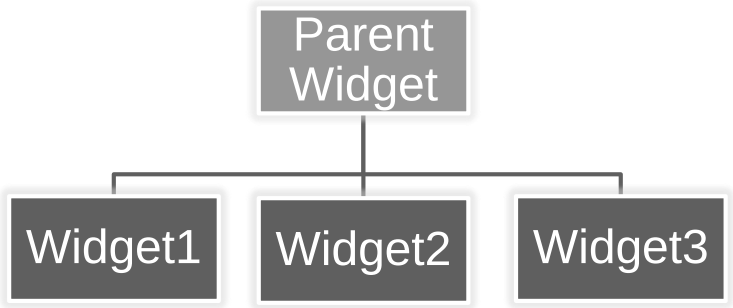
MVC

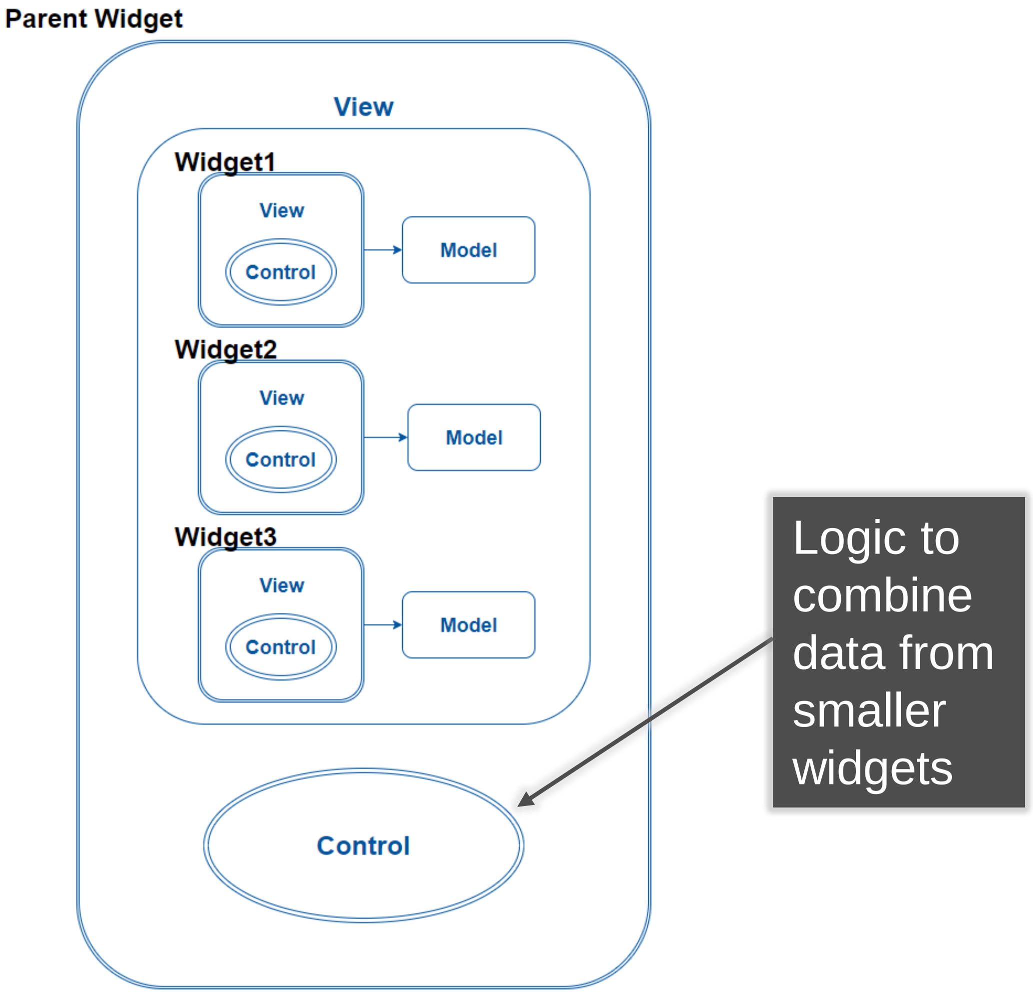


```
class MyWidgetContoller(QObject):
    def __init__(self):
        super(MyWidgetContoller, self).__init__()
        # Create the View based on the automatically generated file
        self._ui = UiWrapperClass()
        # Create the Model
        self._model = MyWidgetModel()
        # Initialize connections
        self.init_connections()


    def init_connections(self):
        # Connect User interaction with model
        self._ui.pushButton.clicked.connect(self.slot_method)
        # Listen for model event signals (Connect model to UI)
        self._model.property_changed.connect(self.on_property_changed

    # Create Slots
    # View ---> model
    @pyqtSlot(bool)
    def slot_method(self):
        self._model.property += 1
    # Model ---> view
    # ...
    #

class UiWrapperClass(QWidget, Ui_Class):
    def __init__(self, parent, *args, **kwargs):
        super(UiWrapperClass, self).__init__(parent, *args, **kwargs)
        self.initUi()

    def initUi(self):
        super().setupUi(self)
        # + Other Ui setup code we want to add


class MyWidgetModel(QObject):
```



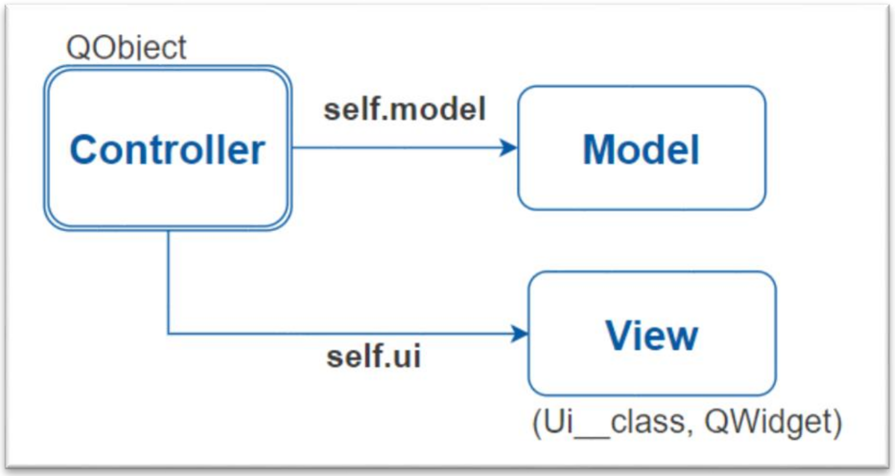
Using our custom Widgets

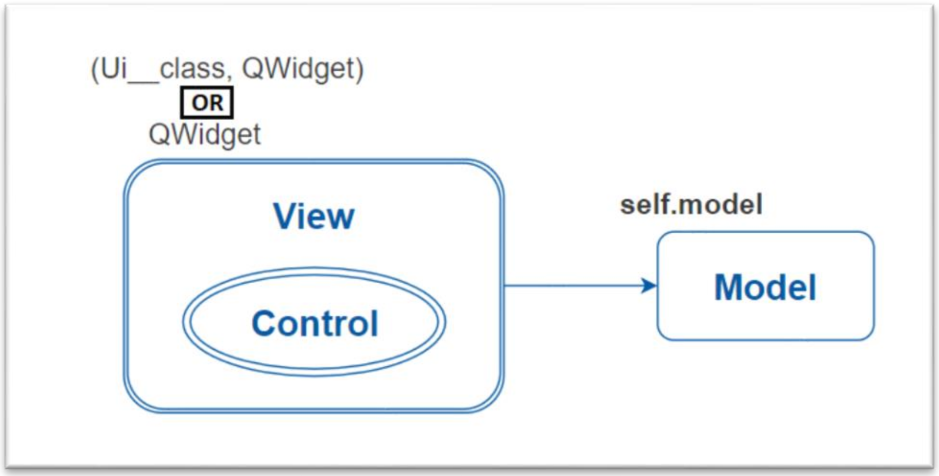




Summing Up

- MVC vs MV?
- Single vs Multiple Inheritance?





Thanks!