MVP Tutorial

Dimitar Tasev

What is MVP?

- Stands for Model-View-Presenter
- Design Pattern
- Used for development of Graphical Interfaces
- There are a lot of similar, but different patterns
  - MVC – Model-View-Controller
  - MVVM – Model-View-ViewModel (used in Windows Forms) [1]
  - Flux – used by Facebook with React [2]

[1] "Introduction to Model/View/ViewModel pattern for building WPF apps" – John Gossman, https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/
[2] Flux – In depth overview, including video talk, https://facebook.github.io/flux/docs/in-depth-overview.html

History of MVP

- Originated as Model-View-Controller
- First published description in 1987 for Smalltalk-80 v2.0 [3]
  - “The central concept behind the Smalltalk-80 user interface is the Model-View-Controller (MVC) paradigm.”
- Evolved into Model-View-Presenter mid-1990s [4]
  - “Taligent, a wholly-owned subsidiary of IBM, is developing a next generation programming model for the C++ and Java programming languages, called Model-View-Presenter or MVP, based on a generalization of the classic MVC programming model of Smalltalk”

[3] Steve Burbeck (1987, updated 1992). “Applications Programming in Smalltalk-80: How to use Model-ViewController (MVC). Available at http://www.dgp.toronto.edu/~dwigdor/teaching/csc2524/2012_F/papers/mvc.pdf
[4] Mike Potel. MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java http://www.wildcrest.com/Potel/Portfolio/mvp.pdf

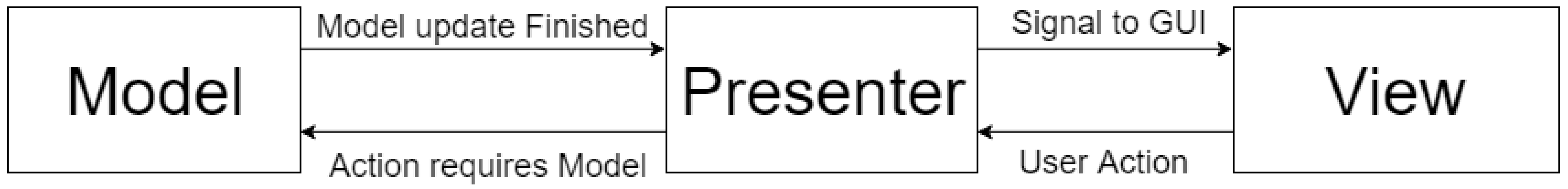
What is the goal of MVP?

“the framework exists to separate the representation of information from user interaction”[5]

[5] The DCI Architecture: A New Vision of Object-Oriented Programming –Trygve Reenskaug and James Coplien – March 20, 2009.

How does it work?

- **Three** main components [4][6]
  - **View** – User Interface: How does the user interact with my data?
    - The ‘look’ of the GUI, what the user sees and clicks
  - **Model** – Data Management: How do I manage my data?
    - Does hard sums, e.g. stores references to workspaces, runs Algorithms on them
  - **Presenter** – How to show the result of the algorithm in the View?



What do we benefit from MVP?

- Separation of components makes them:
  - Smaller code size per component
  - Easier to read
  - Easier to understand
  - Easier to test
- Testing
  - Allows testing of the logic behind the View
  - The Real View is not necessary for testing - mocking

Restrictions and Gotchas

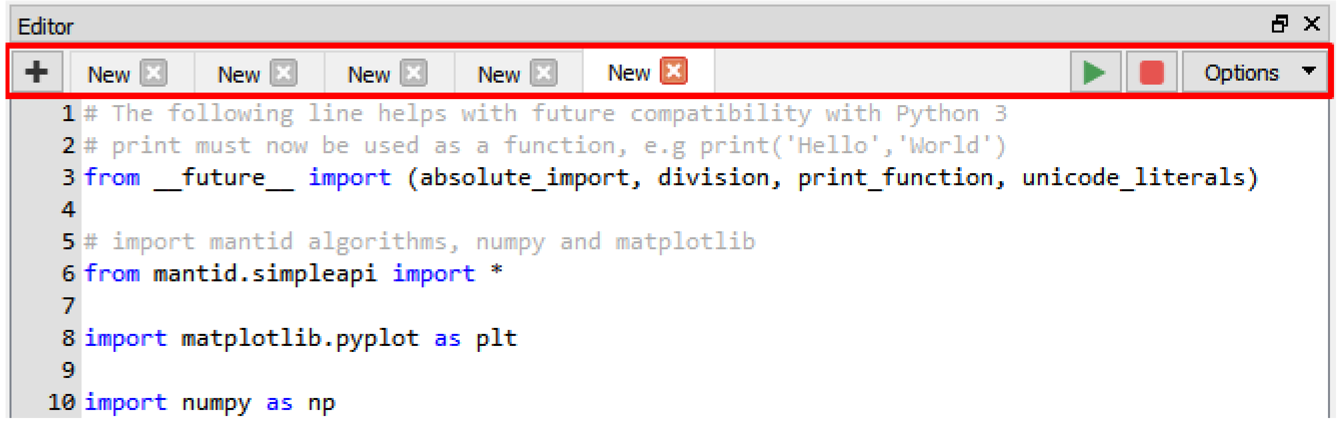
- Presenters should avoid being `QObject`s
  - This could have been done have connections with the Presenter
  - This forces testing to require a QApplication
  - Usually a problem in C++ (Qt4 only?). You DON’T NEED to do it in Python!
  - You can connect to functions
  - Watch out for thread issues if using ADS/Algorithm/etc Observers!

Restrictions and Gotchas

- Models should **NEVER** have to be QObjects
    - You should not connect to the model
    - Makes it harder to follow
    - Harder to test
- View does not have a direct reference to the Model
    - View should **NOT** directly access the Model
    - Information flow is through the Presenter

Restrictions and Gotchas

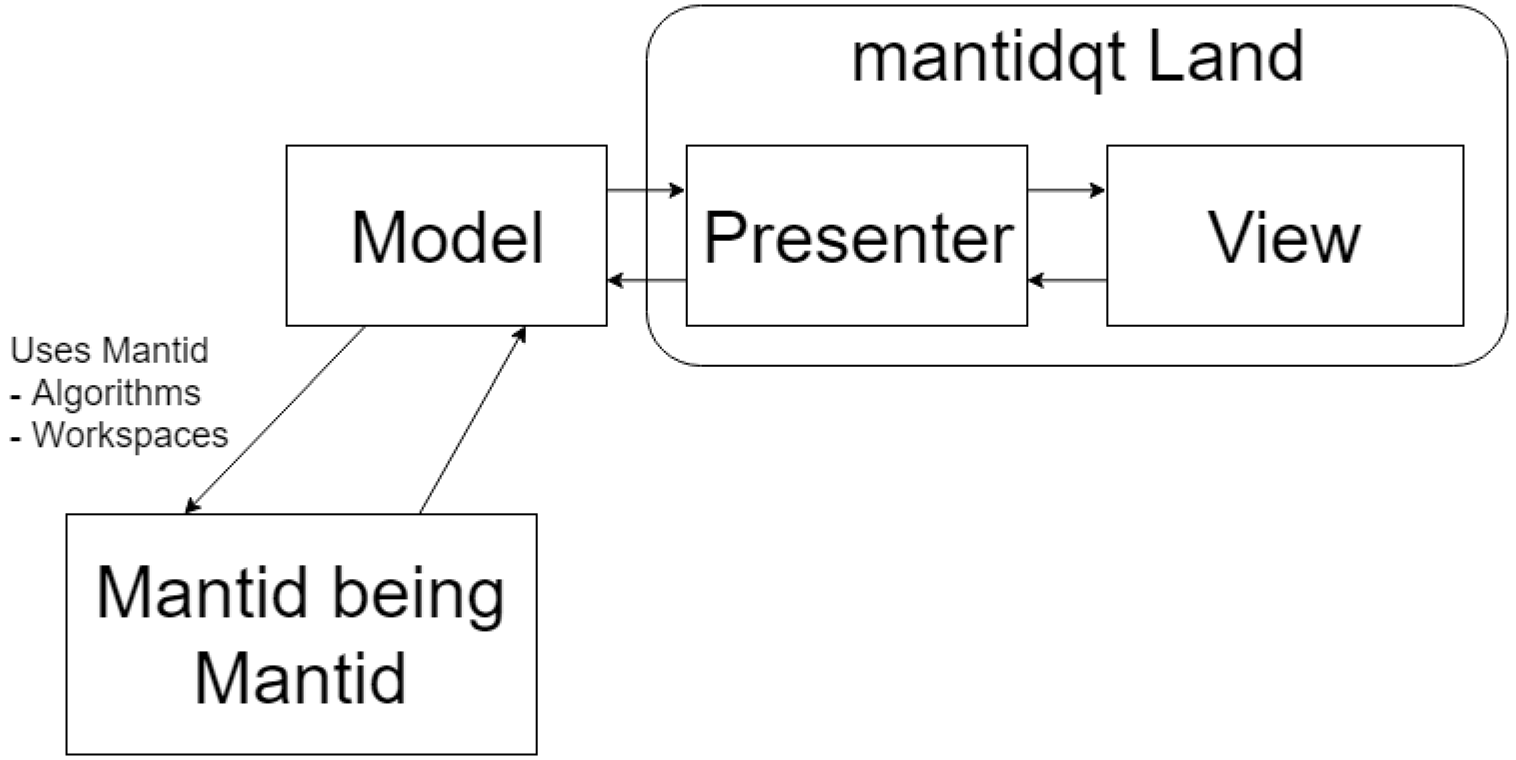
- Presenters can GROW large
  - Can be hard to judge how much should be in the Presenter versus Model
  - Maybe the View can be split into multiple MVPs
    - Example: Tabs are in a separate MVP from the rest of the code editor



```
# The following line helps with future compatibility with Python 3
# print must now be used as a function, e.g print('Hello','World')
from __future__ import (absolute_import, division, print_function, unicode_literals)

# import mantid algorithms, numpy and matplotlib
from mantid.simpleapi import *

import matplotlib.pyplot as plt

import numpy as np
```

Using MVP in Mantid in Practice

Your widget in mantidqt/widgets/your_widget



Code example – Presenter is the owner

```python
class TofConverterPresenter(object):
    def __init__(self, view=None, model=None):
        self.view = view if view else TofConverterView(parent=None, presenter=self)
        self.model = model if model else TofConverterModel(presenter=self)
```

How to initialise?

- Instantiate Presenter
- It makes the View and the Model

```python
class TofConverterModel(object):
    def __init__(self, presenter=None):
        self.presenter = presenter
```

```python
class TofConverterView(base, form):
    def __init__(self, parent=None, presenter=None):
        super(TofConverterView, self).__init__(parent)
        self.setupUi(self)

        self.presenter = presenter
```

```python
def show_find_replace_dialog(self):
    self.find_replace_dialog = EmbeddedFindReplaceDialog(self, self.editor)
    self.layout.insertWidget(0, self.find_replace_dialog.view)
```

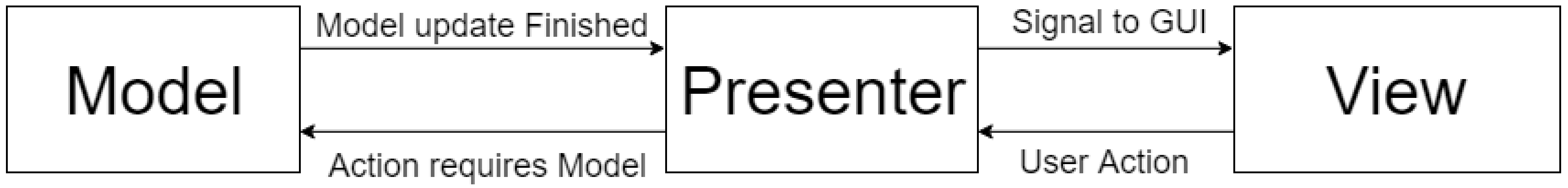
Cons:

- If embedding into another widget, view has to be retrieved separately

Existing MVPs implementations

- Table/Matrix workspace displays
  - Python MVP
  - mantidqt/widgets/workspacedisplays
- Project Recovery
  - MVP in C++ - ProjectRecoveryView.h
  - MVP in Python – projectrecoverywidgetview.py
- Workspace Presenter
  - MVP in C++ - WorkspacePresenter.h
- AlgorithmProgress (C++ Qt5 Only Widget)
  - MVP in C++ - AlgorithmProgressWidget.h

Live Qt Connection Debugging

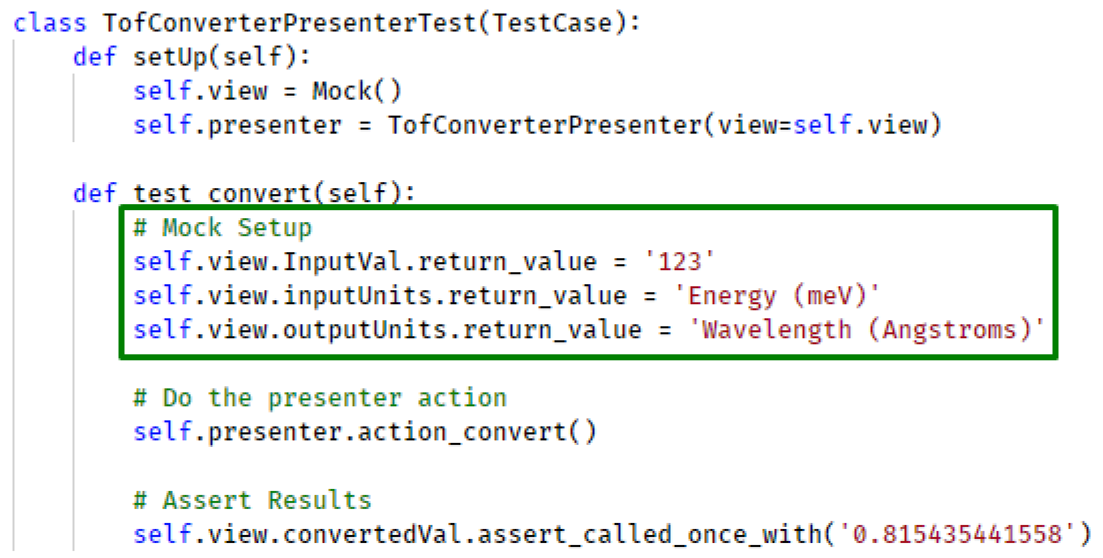


Ways to test

- Unit testing – unittest.TestCase
  - Presenter
  - Model
- Mock testing – mantid.py3compat.mock
  - Use the py3compat for easy Py2/3 compatible import
  - View
- Gui testing – GuiTest
  - Runs an event loop
  - Can simulate clicks
  - Can inspect *all* Qt objects in the application

Mocking a view

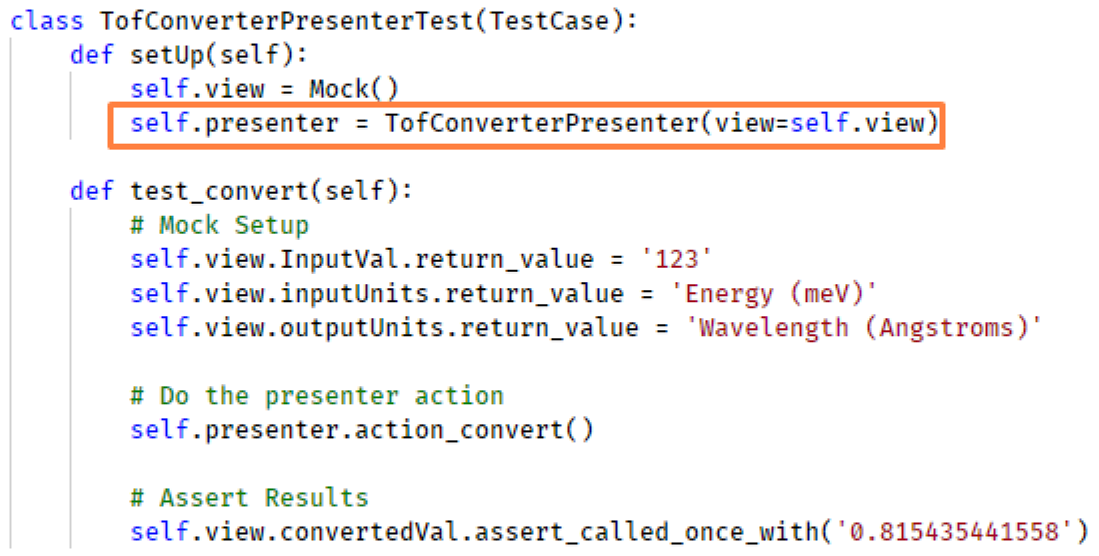
- Benefits
  - You do not need the original view
- Drawbacks
  - You need to set up the view’s expected return values



```
class TofConverterPresenterTest(TestCase):
    def setUp(self):
        self.view = Mock()
        self.presenter = TofConverterPresenter(view=self.view)

    def test_convert(self):
        # Mock Setup
        self.view.InputVal.return_value = '123'
        self.view.inputUnits.return_value = 'Energy (meV)'
        self.view.outputUnits.return_value = 'Wavelength (Angstroms)'

        # Do the presenter action
        self.presenter.action_convert()

        # Assert Results
        self.view.convertedVal.assert_called_once_with('0.815435441558')
```

Mocking a view

- The view is passed as a parameter to the presenter
    - This allows easy replacement without ever instantiating the Qt View
    - The same can be done for the model



```
class TofConverterPresenterTest(TestCase):
    def setUp(self):
        self.view = Mock()
        self.presenter = TofConverterPresenter(view=self.view)

    def test_convert(self):
        # Mock Setup
        self.view.InputVal.return_value = '123'
        self.view.inputUnits.return_value = 'Energy (meV)'
        self.view.outputUnits.return_value = 'Wavelength (Angstroms)'

        # Do the presenter action
        self.presenter.action_convert()

        # Assert Results
        self.view.convertedVal.assert_called_once_with('0.815435441558')
```

Better mocking of a view

- Do not mock out the whole view with
  - view = Mock()
- Mock out the interface of the view

```python
class MockCodeEditorTabView(MockQWidget):
    """
    Represents the QTabView used to contain all tabs
    """

    def __init__(self):
        super(MockCodeEditorTabView, self).__init__()
        self.last_tab_clicked = StrictPropertyMock()
        self.mock_code_editor_tab = MockCodeEditorTab()

        self.widget.return_value = self.mock_code_editor_tab
```

```python
class MockQWidget(object):
    def __init__(self):
        self.addWidget = StrictMock()
        self.replaceWidget = StrictMock()
        self.widget = StrictMock()
        self.hide = StrictMock()
        self.show = StrictMock()
        self.close = StrictMock()
        self.exec_ = StrictMock()
```

Use StrictMock for mocking functions

- StrictMock, StrictPropertyMock are Mantid implementations
  - Not available in Python's Mock package
  - They wrap Python's Mock class
- StrictMock does NOT allow you to call anything that has not been explicitly declared

```
class MockQWidget(object):
    def __init__(self):
        self.addWidget = StrictMock()
        self.replaceWidget = StrictMock()
        self.widget = StrictMock()
        self.hide = StrictMock()
        self.show = StrictMock()
        self.close = StrictMock()
        self.exec_ = StrictMock()
```

```
mockwidget = MockQWidget()
# OK
mockwidget.addWidget(None)
# Error, test fails
mockwidget.unexpected_function()
```

Using **qtpy**

- Connecting things
  - Much easier than C++ with Qt4
  - Somewhat easier than C++ with Qt5
- self.button.clicked.connect(*recieving_function*)
- To see what you get on the *recieving_function*, you read the Qt docs!

Instructions to start off

- All OSs
  - git clone https://github.com/DTasev/mvp
- Windows
  - Go to a build
  - Start command-prompt.bat
  - Navigate to where you cloned the repo
  - Type `powershell` if you don’t like `cmd`, the environment will be kept
- Linux
  - Go

Instructions to start off

- Start with `python tof_converter`
- Entry point is `__main__.py`
  - Run with `python __main__.py` or `python .` Inside `mvp/exercise/tof_converter`
- It creates the presenter
- Which creates the view
- Which shows itself

Exercise 1

- Make the `Convert` button work using a MVP approach
- Use the provided functions from the ‘model.py’ file
- Hints:
  - Add function to presenter
  - Connect to it
  - Import the function from the model

Exercise 2

- Add the Model class.
- Make `Convert` work for all input/output units
- Hints:
  - The class should wrap code already in `model.py`
  - The presenter should instantiate the model and use it

Exercise 3

- Add unit test for the presenter `Convert` action
- Mock the View objects that are read by the Presenter
- File is `test/test_tof_convert_presenter.py`
- Hints:
  - Refactor the model's possible inputs/outputs into a list/enum

Exercise 4

- Comment the following lines in view.py
  - `history.setVisible`
  - `historyLabel.setVisible`
- If you start the TofConverter a new widget will show up
- It stores the previous conversions. Happens on `Convert` click.
- How will you implement the widget?
  - Extend existing presenter and model
  - versus
  - Add new M~~V~~P (no view for it)?

Exercise 5

- Allow the user to double click an entry in the history to load that value back into the view.
- Allow deletion of items with a `-` (minus) button

Exercise 6

- Unit test / mock the History widget
- Scattering angle and Flight Path should be disabled by default - setDisabled(True)
  - If Momentum or d-spacing are selected as either input or output enable \`scattering angle\` field
  - If Time of Flight is selected, enable \`Total flight path\` field

References

- [1] “Introduction to Model/View/ViewModel pattern for building WPF apps” – John Gossman, https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/
- [2] Flux – In depth overview, including video talk, https://facebook.github.io/flux/docs/in-depth-overview.html
- [3] Steve Burbeck (1987, updated 1992). “Applications Programming in Smalltalk-80: How to use Model-ViewController (MVC). Available at http://www.dgp.toronto.edu/~dwigdor/teaching/csc2524/2012_F/papers/mvc.pdf
- [4] MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java Mike Potel http://www.wildcrest.com/Potel/Portfolio/mvp.pdf
- [5] The DCI Architecture: A New Vision of Object-Oriented Programming –Trygve Reenskaug and James Coplien – March 20, 2009.
- [6] MVP Introduction, http://developer.mantidproject.org/MVPTutorial/Introduction.html