Model-View-Controller

Motivation
The MVC pattern
Using the Observer pattern in Java
Multiple views, loosely coupled to the underlying data model.
Many applications have multiples views of one “document”
Observations

• When one view changes, other(s) often need to change.
  – Ideally, we want a single representation of the underlying data, and multiple views of that data.

• The user interface code probably changes more and faster than the underlying application
  – Many recent changes in MS Office were to UI code
  – Excel’s underlying functions and data structures are probably very similar to Visicalc, the original spreadsheet

• How do we design software to support these observations?
Possible Design: Tight Coupling

- Issues with bundling everything together:
  - What if we want to display data from a different type of source (e.g., a database)?
  - What if we want to add new ways to view the data?
- Primary problem with this approach:
  - Data and presentation are tightly coupled
Solution: Model-View-Controller (MVC)

User

Controller

Model

View

User perceives feedback from the system.

Controller translates input to the model.

Model notifies the view about changes.

View presents output to the user.

User expresses their intention to the controller.

Controller translates the user's intention into an input for the model.

Model changes based on the input from the controller.

View updates the user interface with the new output from the model.
MVC History

Developed for Smalltalk-80 in 1979 by Trygve Reenskaug, while visiting Xerox PARC.

Now a standard design pattern for graphical user interfaces that is used at many levels, including the overall application design and individual visual components.

Variations

- Model-View-Presenter
- Model-View-Adapter
- Hierarchical Model-View-Controller

We use “standard” MVC in this course.
Interface architecture decomposed into three parts (classes):
- **Model**: manages the data and its manipulation
- **View**: manages the presentation of the data
- **Controller**: manages user interaction
These classes are loosely coupled:

- **View and Controller both know about the model (through a public interface that the model defines).**
  - Controller is able to update the model based on user input.
  - View needs to be able to display data from the model.
- **Model only knows about the View through it’s interface.**
  - Notifies the view(s) when the model’s internal state changes.

```java
interface IView {
    public void updateView();
}
```
HelloMVC1
1 view

HelloMVC2
2 (or more) views

HelloMVC3
Includes anonymous inner classes, inner classes, etc.

Credit: Joseph Mack for original code
http://www.austintek.com/mvc/
• **MVC in Theory**
  – View and Controller both refer to Model directly
  – Model uses the observer design pattern to inform view of changes

• **MVC in Practice**
  – Model is very loosely coupled with UI using the observer pattern
  – The View and Controller are tightly coupled – why?

• If the View and Controller are tightly coupled, do we still need an iView interface?
  • Why not just have the controller just tell the view to update?
NOTE: MyView does not need to implement IView.
- It could provide an anonymous inner class to MyModel instead.

```java
class MyView ...
{
    model.addView(new IView() {
        void updateView() {
            ...
        }
    });
}
```
Observer Design Pattern
MVC is an instance of the Observer design pattern

- Provides a well-defined mechanism that allows objects to communicate without knowing each others’ specific types
  - Promotes loose coupling
- Related to
  - “publish-subscribe” pattern
  - “listeners”
  - delegates in C#
Observer Design Pattern

**Subject**
- `Vector<Observer> observers`
- `attach(Observer o)`
- `detach(Observer o)`
- `notify()`

**ConcreteSubject**
- `subjectState`
- `getState()`
- `setState()`

**ConcreteObserver**
- `observerState`
- `update()`

- `notify(): for all o in observers { o->update() }`

**Observer**
- `update()`

(subject) --> (observer)
MVC as Observer Pattern

```plaintext
MVC

subject

observer

MyModel
- ArrayList<lView> views
  other instance variables omitted
+ MyModel()
+ void addView(lView view)
+ void removeView(lView view)
- void updateAllViews()
  model-specific methods omitted

MyView
- MyModel model
  view-specific variables omitted
+ MyView(MyModel model)
+ void updateView()
- void layoutView()
- class MyController ...

IView
+ void updateView()
```
java.util provides an Observer interface and Observable class

- **Observer** is like **Iview**
  - i.e. the View implements **Observer**
- **Observable** is the “Subject” being observed
  - i.e. the Model extends **Observable**
- base class maintains a list of Observers and methods to notify them
• Series of demo programs that use MVC
• Program requirements:
  – **vary** base and height of right triangle, display hypotenuse
• TriangleModel
  – stores base and height, calculates hypotenuse
  – constrains base and height values to acceptable range
Issues with SimpleTextView

- Precision of Hypotenuse varies; sometimes wider than the textbox.
- Hypotenuse can be edited but that doesn’t change the model.
- Tabbing or clicking out of base or height doesn’t do anything; must hit ‘Enter’.
Multiple Views

Triangle

Base: 50
Height: 50
Hypotenuse: 70.711

Base: 50
Height: 50
Hypotenuse: 70.7068

Base: 50
Height: 50
Hypotenuse: 70.711
Graphical View
Graphical View

Base: 100
Height: 92.473
Hypotenuse: 136.203
Practical Details
• Set up the infrastructure
  – Create three or more empty classes:
    • the model
    • one or more view/controller classes (extends JComponent or JPanel)
    • a class containing the main method
  – In the main method:
    • create an instance of the model
    • create instances of the views/controllers, passing them a reference to the model
    • display the view(s) in a frame
public class Main{

    public static void main(String[] args){
        JFrame frame = new JFrame("HelloMVC1");

        // create Model and initialize it
        Model model = new Model();
        // create Controller, tell it about model
        Controller controller = new Controller(model);
        // create View, tell it about model and controller
        View view = new View(model, controller);
        // tell Model about View.
        model.setView(view);
    }

    class View extends JPanel implements IView {

        // the view's main user interface
        private JButton button;

        // the model that this view is showing
    }
}
• Build and test the model
  – Design, implement, and test the model
    • add commands used by controllers to change the model
    • add queries used by the view to update the display
  – Call `updateAllViews()` just before exiting any public method that changes the model’s data
• Build the Views and Controllers
  – Design the UI as one or more views. For each view:
    • Construct widgets
    • Lay the widgets out in the view
    • Write and register appropriate controllers for each widget
    • Write `updateView()` to get and display info from the model
    • Register view (with `updateView()` method) with the model
Summary
• Separation of concerns enables alternative forms of interaction with the same underlying data.
  – Data and how it is manipulated (the model) will remain fairly constant over time.
  – How we present and manipulate that data (view and controller) via the user interface will likely change more often than the underlying model.
  – E.g. transitioning an application from desktop to smartphone to watch versions.
• Separation of concerns enables multiple, simultaneous views of the data.

• Given the same set of data, we may want to render it in multiple ways:
  – a table of numbers
  – a pie chart
  – a line graph
  – an audio stream
  – ...

• A separate model makes it easier for different UI components to use the same data
  – Each view is unencumbered by the details of the other views
  – Reduces dependencies on the GUI that could change
• Separation of concerns enables one to more easily develop and test data-specific manipulations that are independent of the user interface
  – Build tests that exercise the model independent of the interface
  – Makes automated tested of user interfaces practical