

Human Computer Interaction

The MVC (and PAC) Design Patterns

Prof. Andrew D. Bagdanov

Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Firenze
`andrew.bagdanov AT unifi.it`

November 8, 2016

- 1 Overview
- 2 The MVC Model
- 3 An Illustrative Abstraction
- 4 A Hands-on Example
- 5 What about PAC?
- 6 Summary
- 7 Homework

Overview

- In this lecture we will finally see a formal-ish introduction to the Model-View-Controller (MVC) paradigm.
- I will try to motivate each key concept with hands-on examples using our running example widgets.
- MVC is a technique you can apply to **isolate** key components of your GUI applications.
- It helps render these components **reusable** and **future-proof**.
- **Note:** MVC is only a paradigm (sometimes it's called an **Architectural Pattern** – it does **not** give a complete recipe for any and all applications.

The MVC Model

- The Model-View Controller (MVC) design pattern that lends itself quite well to the design of graphical user interfaces in general.
- Since I come from the generation that invented design patterns, I feel the need to sell them whenever I can.
- A design pattern describes a proven solution to a recurring design problem, placing particular emphasis on the context and forces surrounding the problem, and the consequences and impact of the solution.
- There are many good reasons to use design patterns:
 - ① They are **proven**: You tap the experience, knowledge and insights of developers who have used these patterns successfully in their own work.
 - ② They are **reusable**: When a problem recurs, you don't have to invent a new **solution**: you follow the pattern and adapt it as necessary.
 - ③ They are expressive: Design patterns provide a common vocabulary of solutions, which you can use to express larger solutions succinctly.

- It is important to remember that design patterns **do not guarantee success**, and they should **never be blindly applied**.
- The programming language Smalltalk first defined the MVC concept in the 1970's.
- Since then, the MVC design idiom has become commonplace, especially in object-oriented systems.
- It has become **especially** commonplace in the world of **web application design**.
- In fact, it is illustrative to take a look at the **massive number** of **web frameworks that are based on (or support) MVC**.

- The MVC architectural pattern divides an interactive application into three components:
 - The **Model** contains the core data and related functionality.
 - **Views** display information to the user.
 - **Controllers** handle user input and mediate communication between views and model.
- Views and controllers together **comprise the user interface**.
- A **change-propagation mechanism** implemented by the Controller usually ensures consistency between the user interface and the Model.
- The most important separation is between **presentation** and **application logic**. The need for **View/Controller split** is less evident.
- MVC encompasses more of the **architecture** of an application than is typical for a design pattern, and the term **architectural pattern** for it may be more accurate.

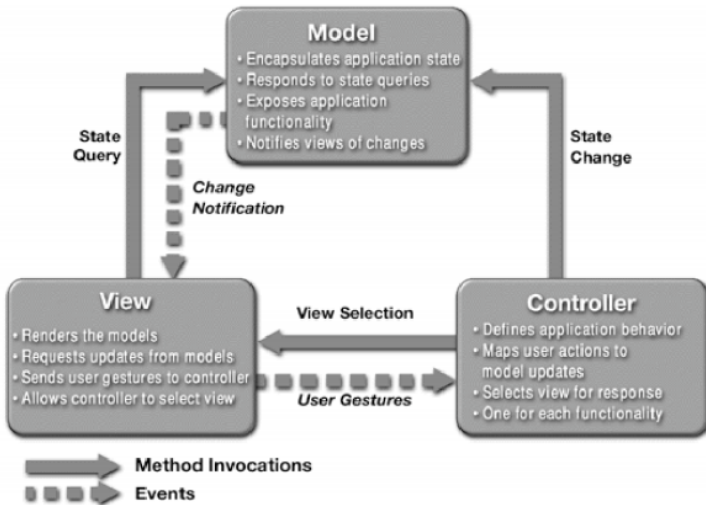
- The **Model** is a domain-specific representation of the information on which the application operates.
- The model is another name for the **application logic layer** (sometimes also called the **domain layer**).
- **Application (or domain) logic** adds meaning to raw data (e.g. calculating if today is the user's birthday, or the totals, taxes and shipping charges for shopping cart items).
- Many applications use a **persistent storage mechanism** (such as a database) to store data.
- MVC does not specifically mention the resource management layer because it is understood to be **encapsulated** by the **Model**.
- The **Model** should contain all **data and presentation-independent functionality** that needs to operate on that data.

- A **View** Renders the Model (or some aspect of the Model) into a form suitable for interaction, typically a user interface element.
- MVC is often seen in web applications, where the View is the HTML page and the **code which gathers dynamic data (from the Model) for the page**.
- Note that there can be **multiple Views** of the same Model – that is one of the main advantages of **separation of concerns** offered by MVC.
- If the user changes the model via the Controller of one View, all other views dependent on this data should reflect the changes.
- The model therefore must somehow **notify all views** whenever its data changes, and Views must **retrieve new data from the model** and **update the displayed information**.

- Finally, a **Controller** processes and responds to events, typically user actions, and may invoke changes on the associated **Model** and **Views**.
- Often, each View has an **associated Controller** component (i.e. they are **tightly coupled**).
- It is often even debatable whether a **View's Controller** should even be a separate component.
- Controllers receive input, usually as events that encode mouse movement, activation of mouse buttons, or keyboard input.
- Events are translated to service requests for the Model or the associated View.
- **The user interacts with the system solely through controllers.**

MVC: A High-level Overview

- Here is a typical diagram of the MVC model in action:



- Though MVC comes in a million scenarios, this is what a typical application might look like.
 - 1 The **user** (remember him?) interacts with the user interface in some way (e.g., user presses a button).
 - 2 A **Controller** handles the input event from the user interface, often via a registered handler or callback.
 - 3 The **Controller** accesses the **Model**, possibly updating it in a way appropriate to the user's action (e.g., **Controller** updates user's shopping cart). Complex **Controllers** are often use the **command pattern** to encapsulate actions and simplify **extension and composability**.
 - 4 A **View** uses the **Model** to generate an appropriate user interface (e.g., View produces a screen listing the shopping cart contents).
 - 5 The **View** gets its own data from the **Model** and the **Model** has **no direct knowledge of the View**.
 - 6 Repeat.

An Illustrative Abstraction

- We will now have a look at an entertaining (and illustrative) presentation on MVC by Prof. Ron Fedkiw from Stanford University's course on mobile application design.
- [SWITCH PRESENTATION]

A Hands-on Example

- Now let's go back to a version of our **running example** and see how we can adapt to the MVC architecture.
- Recall that we had a `ClickButton` widget class, already defined in the KV language:

```
<CounterButton@Button>:
```

```
    counter: 0
    text: 'Clicks: {}'.format(self.counter)
    on_press: self.counter += 1
```

- In this simple example, we have in a single class: the **Model** (a integer counter), the **View** (a Kivy `Button` we inherit from) and **Controller** (intercepting property events to update **View**).
- With such simple state and update logic, it is hard to motivate the need for MVC.

- But, what if we begin increasing the complexity of the update logic?

Our class extends EventDispatcher so we can use properties.

```
class CounterModel(EventDispatcher):
```

```
    # The counter value
```

```
    counter = NumericProperty()
```

```
    interval = NumericProperty()
```

```
    # Constructor
```

```
    def __init__(self, initval=0, maxval=None):
```

```
        super(EventDispatcher, self).__init__()
```

```
        self.counter = initval
```

```
        self._maxval = maxval
```

```
        self._lastupdate = time.time()
```

```
        self.interval = 0.0
```

```
    # Method to bump the counter.
```

```
    def inc_counter(self):
```

```
        # This ensures the bound on counter if maxval not None.
```

```
        if not self._maxval or (self.counter < self._maxval):
```

```
            self.counter += 1
```

```
            now = time.time()
```

```
            self.interval = now - self._lastupdate
```

```
            self._lastupdate = now
```

- We can imagine more complex Models, with multiple data types (**structures**), multiple items (**lists**, **dicts**).
- More importantly, we can imagine more Models which require more complex **backends** (e.g. a **DBMS**).
- In such cases, it is essential to isolate the internal structure and logic of the Model:
 - To keep the logic of presentation in Views uncluttered with data-access and data update code.
 - To abstract access to Model data and render Views and Controllers independent of the actual storage model used by the Model.
 - To make it possible for multiple Views to share the same Model.
- This example is clearly artificial, but let's pretend.

- Now let's define two Views in the KV design language (Kivy Buttons in our case):

```
<CounterButton@ButtonWithModel>:
```

```
text: 'Clicks: {}'.format(self.model.counter)
on_press: self.model.inc_counter()
```

```
<CounterIntervalButton@ButtonWithModel>:
```

```
text: 'Clicks: {} [ {:.2f} ]'.format(self.model.counter,
                                     self.model.interval)
on_press: self.model.inc_counter()
```

- These Views assume they have an associated model (which should probably be accessed through a **singleton**).
- They are not dependent on the **Model**, but on the **model interface**.
- We are free to **change the implementation** of counter as much as we like.

- Now we can instantiate and start using these new MVC components.
- Note, this is all technically **Controller** code:

Simple, single button app.

```
def single_button():  
    root = Factory.CounterButton(CounterModel())  
    runTouchApp(root)
```

Simple, both button types displayed.

```
def both_buttons():  
    left = Factory.CounterButton(CounterModel())  
    right = Factory.CounterIntervalButton(CounterModel())  
    top = BoxLayout(orientation='horizontal')  
    top.add_widget(left)  
    top.add_widget(right)  
    runTouchApp(top)
```

- With the new Model implementation, we can share the same Model instance across multiple views:

Both button types, shared model.

```
def shared_buttons():  
    left = Factory.CounterButton(CounterModel())  
    right = Factory.CounterIntervalButton(left.model)  
    top = BoxLayout(orientation='horizontal')  
    top.add_widget(left)  
    top.add_widget(right)  
    runTouchApp(top)
```

What about PAC?

- Our discussion today has been suspiciously silent on the topic of the Presentation-Abstraction-Control (PAC) design pattern.
- As I have previously mentioned, PAC can be thought of as **hierarchically** applied MVC.
- In PAC, the **Presentation** component corresponds to **View**, and **Abstraction** to **Model** in MVC.
- Another difference is that in PAC, there is a **strict separation** between **Presentation** and **Abstraction** components (the **Control** mediates **all communication** between components).
- This has significant efficiency advantages in applications with complex **Abstractions** or **Presentations** (or both).

Summary

- MVC (and PAC) are architectural design patterns that promote **reusability** and **extensibility** through structured **separation of concerns**.
- We saw through examples how even simple **Models** can be made more reusable through abstraction of the data backend.
- These models can support multiple **Views**, and can even be shared among multiple **Views** simultaneously.
- As with all patterns, **MVC should not be applied blindly**, but always with a critical eye for elegance, readability, and maintainability.
- Note that in **modern GUI frameworks**, many aspects traditionally handled by **Control** components is handled automatically by event processing subsystems.
- Also note that many GUI frameworks (like Kivy) directly support MVC organization through their underlying architecture (see Homework).

Homework

Exercise 12.1: extending the example

Read the Kivy documentation on **Adapters**. This is a class in Kivy designed to help mediate communication between **Models** and **Views**. Implement a new **View** that visualized our existing CounterModel **Model** from this lesson as a **ListView**. The **ListView** should display the click count, and the interval.

Exercise 12.2: extending the example (again)

Extend the view implemented in the previous exercise (and the **CounterModel**, probably) to include a **Reset** button in the visualization of the **ListView** of CounterModel.

Exercise 12.3: reset functionality.

Implement the required functionality for the reset button associated with each **ListItem** in the previous exercise. **NOTE**: This will require extensions to the model.