CSC 335 – MVC and Design Patterns

Dr. Jonathan Misurda
jmisurda@cs.arizona.edu

Separation of Concerns

- One of the primary goals of our OOP design is to break our program into small parts each dealing with some aspect of the total program
- Design Principle:
  - Divide our program up into small sections
  - Each section is responsible for a single task
  - Each section is separate from each other (enabling testing, reuse, and modification without worrying about other sections)
- We have called these sections **modules**, and in Java, classes are one way of implementing modularity

A Common Program Design

- Think of a typical interactive program.
- It probably, broadly, has three parts:
  1. Some way of interacting with the program
  2. Some data structure(s) that model the problem the program solves
  3. Some code that responds to the interaction and uses the data structures to do some work

Designing an Architecture

- If we are adhering to the goals of modularity via the separation of concerns, we could see each of those tasks as a separate unit of our program
- In an OOPL, that might mean each broad part becomes a separate class
- Thus, without even considering the program's behavior, we have an idea about how to break it into modules

Model/View/Controller

- This particular architecture is often called Model/View/Controller (MVC)
  - The model is the object that holds the data that represents the state of the world
  - The view shows the data to the user and allows them to interact with it
  - The controller acts as the glue between them
- Each component is kept separate
  - E.g., the view never talks to the model, except through the controller

What does MVC buy us?

- Abstraction
- If we have properly separated the concerns and made a useful interface between the model and the controller, and the controller and the view, we get something very powerful:
  - You can change the view **without** changing the controller or the model
  - You can change the model **without** changing the controller or the view

Changing the View

- What would it take to change our project from a console-based text input view to one of a graphical user interface (GUI)?
  - Would you need to change the model?
  - Would you need to change the controller?

Exceptions help keep SoC

- Note that the controller needs to enforce that it's actions are on valid input
  - This might result in some change to the view
- We do not want the controller displaying a message or asking for new input
  - That's the view's job and only the view's job
- So if we get an error, we can throw an exception, and the caller of the controller's method (the view) can handle it

In-Class Activity: MVC

- Choose an application to model in the MVC architecture
  - We've already uses Mastermind and Tic Tac Toe as examples, so choose something else
  - It doesn't have to be a game
- Use the form handed out to produce a UML diagram for each MVC class
  - Not writing any actual code

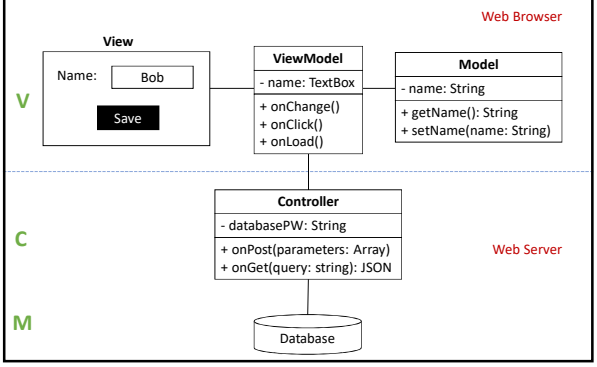
MVVM

- We hear a lot today about a similar architecture to MVC, called **Model-View-ViewModel** or MVVM
- This is a common design among a number of Single Page Application (SPA) web programming frameworks
  - Angular.js
  - Ember.js
  - Knockout.js

View Model

- A View Model is a module whose job it is to map (bind) elements from the view to the model
- For instance, we might have a textbox that represents the contents of some variable
  - When we edit the textbox, we want the variable to change
  - When we change the variable, we want the textbox to change

MVVM with MVC



Patterns

- A **pattern**
  - Describes a *problem* which occurs over and over again
  - Describes the *core* of the *solution* to that problem
- It does so "in such away that you can use this solution a million times over, without ever doing it the same way twice"
  - Christopher Alexander – A pattern Language
- A pattern is not an implementation (code). It is a design.

Architectural Patterns

- MVC and MVVM are **Architectural Patterns**
  - They described a way to design an application's architecture:
    - **Identify** the concerns to separate
    - **Map** those concerns onto classes
    - **Describe** how those classes should interact
- At no point did they ever tell you:
  - What methods to have
  - What fields to store
  - What classes beyond the top level ones (M, V, C, VM) to have
  - What code to write

Design Patterns

- A **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design.
- For instance, we may only want to have at most a single instance of an object instantiated.
- We can use a *Singleton Pattern*.

Singleton

```
public class Singleton {
	private Singleton() { }

	private static Singleton myInstance;

	public static Singleton getInstance() {
		if(myInstance == null)
			myInstance = new Singleton();
		return myInstance;
	}
}
```

java.lang.Runtime Documentation

```
public class Runtime extends Object
```

Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime method.

An application cannot create its own instance of this class.

Some `Runtime` Methods

Method	Description
`availableProcessors()`	Returns the number of processors available to the Java virtual machine.
`exec()`	Executes the specified string command in a separate process.
`exit(int status)`	Terminates the currently running Java virtual machine by initiating its shutdown sequence.
`freeMemory()`	Returns the amount of free memory in the Java Virtual Machine.
`gc()`	Runs the garbage collector.
`getRuntime()`	Returns the runtime object associated with the current Java application.

Why is `Runtime` a Singleton?

- This represents the underlying system features, it wouldn't make sense to instantiate multiple copies
- Why not just have them all be static methods, like with System?
  - ¯_(ツ)_/¯

Singletons

- Singleton is the easiest design pattern to understand
- It's also probably one of the worst
  - Captures the notion of "global" state, which can begin to break down modularity
- Throughout the course we will see more, some used in the Java Class Library and some not