

# Architectural Patterns

## Model-View-Controller Pattern

School of Computing  
Clemson University  
Clemson, SC 29634 USA

CPSC 2150: Monday, April 3, 2023

## Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

- 1 A Motivating Example
- 2 Observer Pattern
  - ▶ UML Diagram
  - ▶ Graphical User Interfaces (GUI)
  - ▶ Illustrative Example

# Homework Assignment

- 1 Programming assignment 4 is posted on your lab Canvas page.

# Exam #2

## 1 Exam #2 - April 10, 2023

- ▶ Please complete the Lockdown Browser + Monitor Practice Quiz as part of your participation grade when it opens up. More information will be announced on Canvas.
- ▶ Will post review slides soon!

# What Are Design Patterns?

“a general reusable solution to a commonly occurring problem within a given context in software design”

- ▶ Design patterns represent the collective experience of decades of object-oriented software development experience.

# Design Patterns vs. Architectural Patterns

- ▶ Design patterns specify a solution for a particular problem within a larger program.
- ▶ Architectural patterns specify a solution for the entirety of a program (or at least a significant portion of it).

# Model-View-Controller (MVC) Pattern

- ▶ One of the most common **architectural patterns** is the Model-View-Controller (MVC) design pattern
- ▶ We use this to separate our user interface from our code.
  - ▶ **Note:** distinction between user interface and Java `interface`
- ▶ This becomes increasingly important with Graphical User Interfaces (GUIs)

# Model-View-Controller (MVC) Pattern

## Model

- ▶ The **Model** in our code is the lower level model that would work with any user interface
  - ▶ Contains our **Entity Objects**
    - ▶ Entity object represent real world ideas
    - ▶ Such as... `Car`, `Employee`, `GameBoard`, `BoardPosition`, etc.
- ▶ By separating the model out it's reusable
- ▶ Our **Model** code cannot refer to the user interface at all
  - ▶ `GameBoard` has `toString()`, instead of printing to the screen
- ▶ Our **Model** is a layer of our architecture, not an individual class

# Model-View-Controller (MVC) Pattern

## View

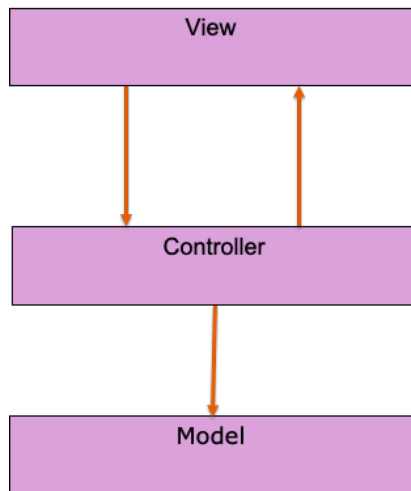
- ▶ The **View** is our user interface code
- ▶ How the user views and interacts with our system
  - ▶ Our **Boundary Objects**
- ▶ Our views have always been command line prompts, and we haven't thought about them much
- ▶ If we have a GUI, our view would become much more complex
  - ▶ All of these code files would go into our view layer

# Model-View-Controller (MVC) Pattern

## Controller

- ▶ The **Controller** is the code that connects our **View** (user interface) and our **Model**
  - ▶ Our controller observes the UI
  - ▶ Our **Control Objects**
- ▶ It passes information back and forth

# Model-View-Controller (MVC) Pattern



# Changing an user interface

- ▶ The user interface of a program changes more often than any other part.
  - ▶ **Ex 1:** Port the program to another OS
  - ▶ **Ex 2:** Make a mobile app
  - ▶ **Ex 3:** Update to the new version of the OS
- ▶ Thanks to MVC, our changes are limited
  - ▶ **View** will change to have the new UI
  - ▶ **Controller** may require minor changes to work with the UI, but all of the logic is the same<sup>1</sup>
    - ▶ Linear vs. Event-Driven Control
  - ▶ **Model** is **unchanged**

---

<sup>1</sup>Controller may or may not change, because of wildly different UI can change the method of control

# Design Patterns and GUIs

- ▶ **Model-View-Controller** helps us separate our GUI from the bulk of our code
  - ▶ Helps adapt to changes in the GUI
- ▶ **Observer Pattern** helps us handle complex GUIs
  - ▶ Can't control what the user does next
  - ▶ Handle different types of events
- ▶ You can use multiple design patterns in one program

# Using Patterns in GUIs

- ▶ Remember, *everything* in Java is a `class` (or an `interface`)
- ▶ This means that our GUIs have to be a `class` as well!
- ▶ We'll create a class for the screen we want to display to the user
  - ▶ A program with multiple screens for the user to interact with will have multiple classes – one for each screen
  - ▶ Will have demo when we talk about Java GUIs!

# Using Patterns in GUIs

- ▶ Our classes for our different screens will all be a part of our **View**
- ▶ Their responsibilities are simple:
  - ▶ Display the screen correctly to the user
  - ▶ Detect any events
  - ▶ Inform the **Controller** of any events
  - ▶ Allow for the **Controller** to display output to the screen
- ▶ How does our code do this?

# Displaying the Screen

- ▶ The different types of “widgets” are all different classes as well, with defined properties and methods
- ▶ Your screen class is extending the `JFrame` class (in Java Swing)<sup>2</sup>, which creates a basic, blank-slate window
- ▶ You then add instances of the different “widget” classes to your `Screen` class by declaring a variable and calling the constructor
  - ▶ Buttons, textboxes, etc
  - ▶ This will add them inside the `JFrame`, and display them to the user

---

<sup>2</sup>More on in the Java GUI lectures

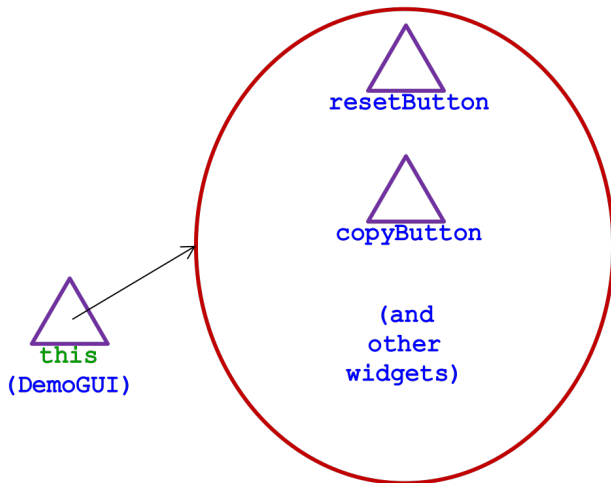
# Detecting Any Events

- ▶ We need our screen to know when a user interacts with a widget
  - ▶ Each widget is an instance of a `class`
  - ▶ How do we know when this happens?
- ▶ **Observer Pattern!**
  - ▶ Each widget is a *subject*
  - ▶ The `Screen` class is our *observer*
- ▶ After adding an instance of the widget as a variable to the `Screen` class, the `Screen` class registers as an *observer* for that widget

# Detecting An Event

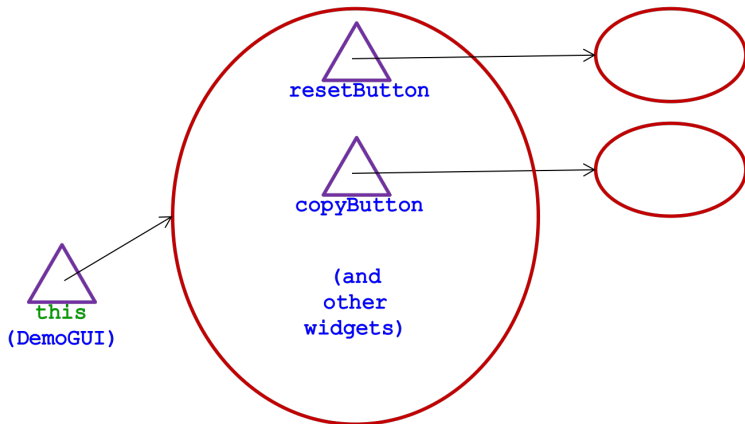
- ▶ Our **Screen** class also defines a callback method
  - ▶ Takes in a *subject* and an *event*
    - ▶ Subject – which widget had the event
    - ▶ Event – what happened to the widget (Click, hover over, etc.)
  - ▶ Checks to see what happened to what widget and responds to that event
    - ▶ Usually by calling a method in the **Controller**

# Illustrative Example



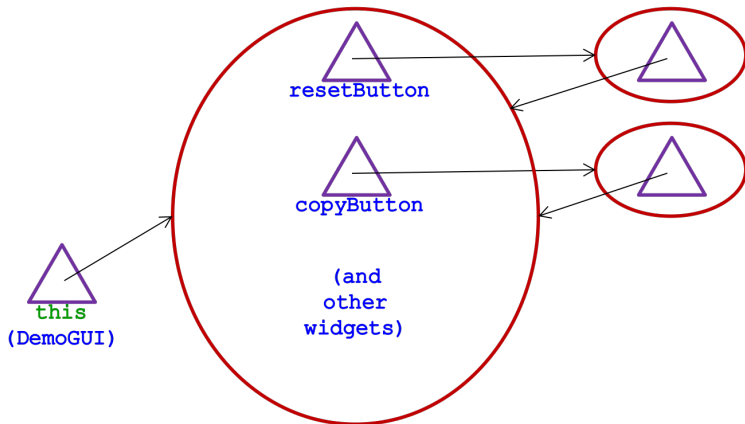
Set Up by `DemoGUI` Constructor

# Illustrative Example



Before registration of `this` as an observer of the buttons.

# Illustrative Example



After registration of `this` as an observer of the buttons.

# Allowing Controller to display output

- ▶ Our **Screen** class calls a class in the **Controller** section of the code
  - ▶ That **Controller** will call any code in the **Model** that is necessary
  - ▶ The **Controller** needs to be able to provide output to the screen.
- ▶ Our **Screen** class just needs to provide methods for displaying information to the user<sup>3</sup>
  - ▶ **Controller** calls those methods
  - ▶ Methods could be specified in a Java **interface**

---

<sup>3</sup>Keep the methods general: **ShowTextToUser** and not **SetOutputTextBox**

# Design Patterns in GUIs

- ▶ Our observer pattern occurs entirely in the **View** section of our code
  - ▶ Our **Screen** observes our widgets to detect event
- ▶ Our classes in our **View** section of the code also have corresponding **Controller** objects
  - ▶ Needs to be able to call methods on the **Controller** object
  - ▶ **Controller** object needs to be able to call methods on the **Screen** class
  - ▶ They need references to each other

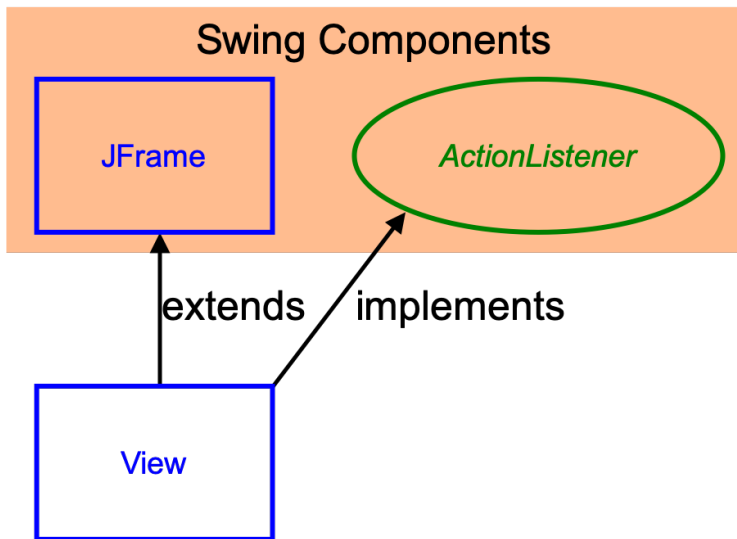
# Controller Objects

- ▶ Just like the **View** section of our code could have multiple files, our **Controller** section will have multiple files
  - ▶ They control the flow of different tasks in our system
    - ▶ Can create new screens (**Views**) and new **Controller** objects to pass control
  - ▶ Often tied to different screens that can appear
  - ▶ Each **Control Object** will have a reference to the instance of the screen class that allows to user to complete that particular task
    - ▶ That object also needs a reference back to the **Control Object**

# Controller Objects

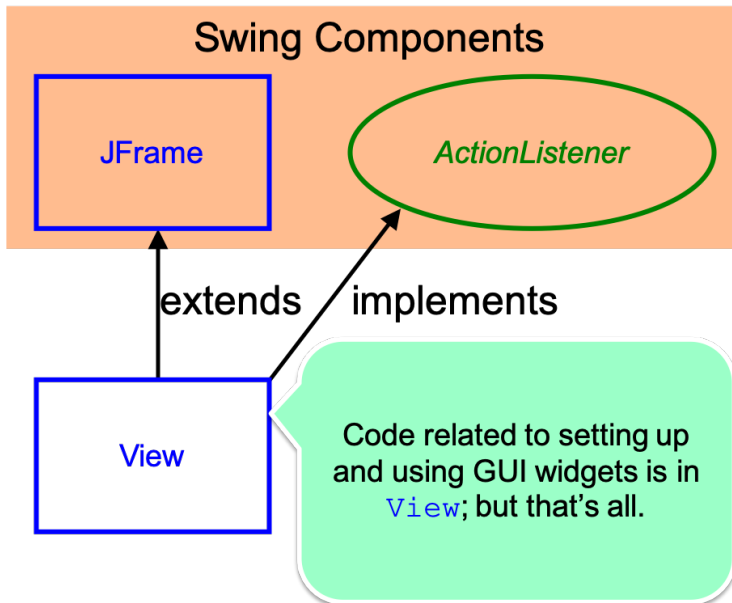
- ▶ **Control Objects** can also contain instances of **Entity Objects**
  - ▶ The classes for entity objects are part of our **Model** section of code
- ▶ The **Control Object** gathers input from the user via the **View/Screen** object
  - ▶ Uses that input to create/use entity objects in the **Model**
  - ▶ Gets output from the entity objects
  - ▶ Passes output back to the **View/Screen**

## Example: Simple MVC GUI Demo<sup>4</sup>

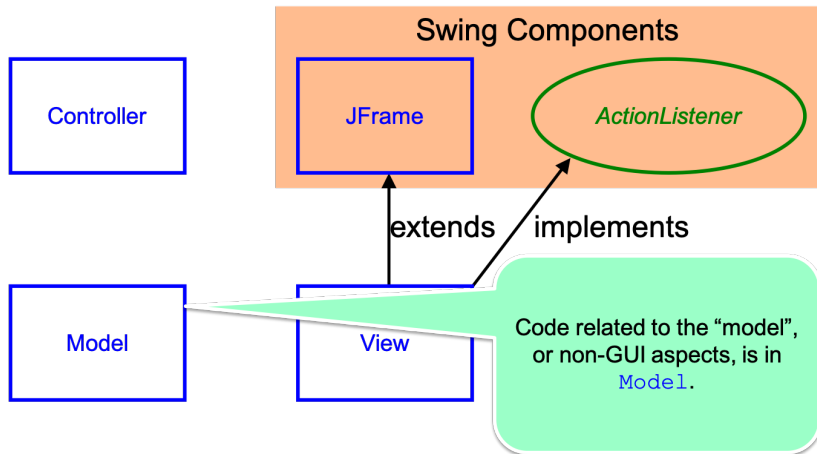


<sup>4</sup>Live Demo of This In Java Gui Lectures!

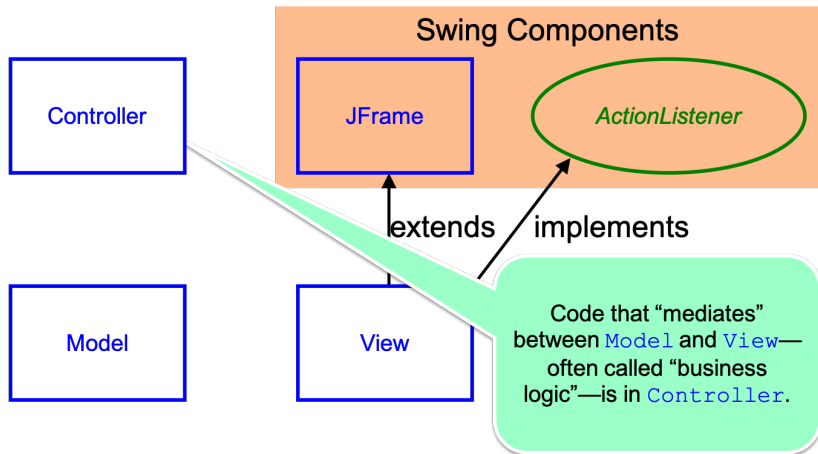
## Example: Simple MVC GUI Demo



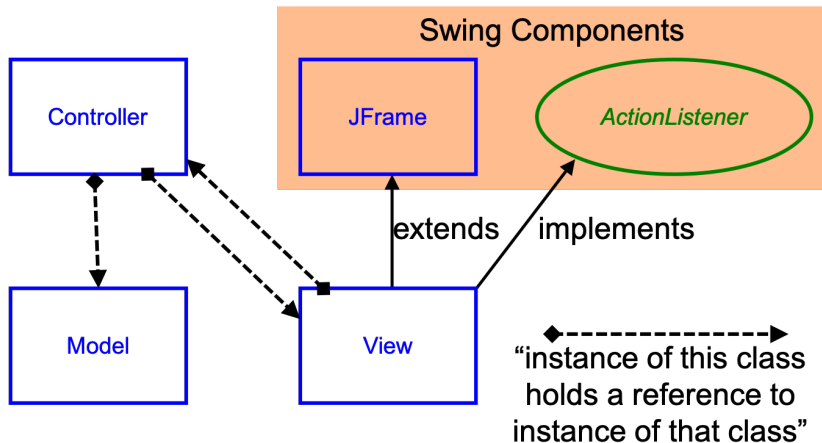
# Example: Simple MVC GUI Demo



## Example: Simple MVC GUI Demo



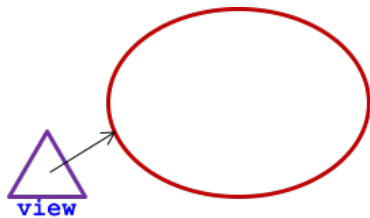
## Example: Simple MVC GUI Demo



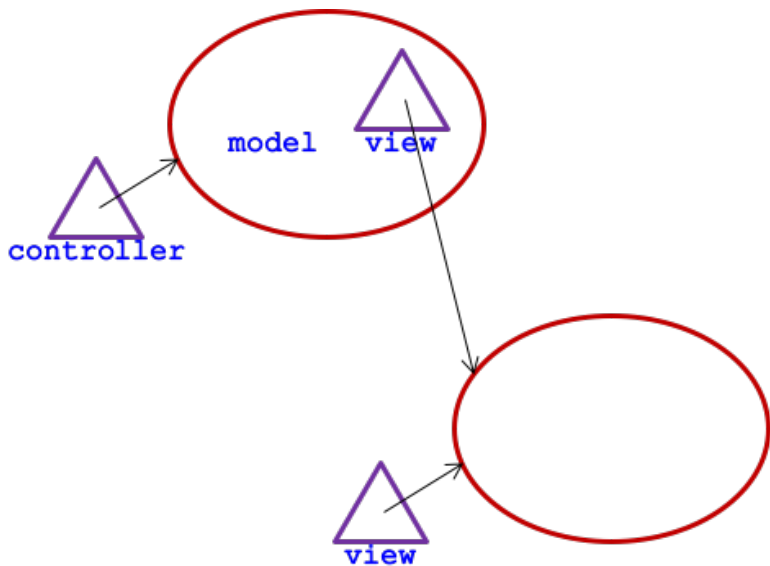
# Our main method

- ▶ So how does this actually run?
- ▶ We still need a main driver class and method
  - ▶ The `main` method is the entry point of the program
- ▶ `main` method must:
  - ▶ Create our view instance
  - ▶ Create our controller instance, passing in the view to the constructor
  - ▶ Register our controller as an observer of our view
    - ▶ View can call methods of the controller class to react to events
- ▶ `main` method might:
  - ▶ Create our model instance as well
  - ▶ Often done by the controller since it knows what model object(s) it will need

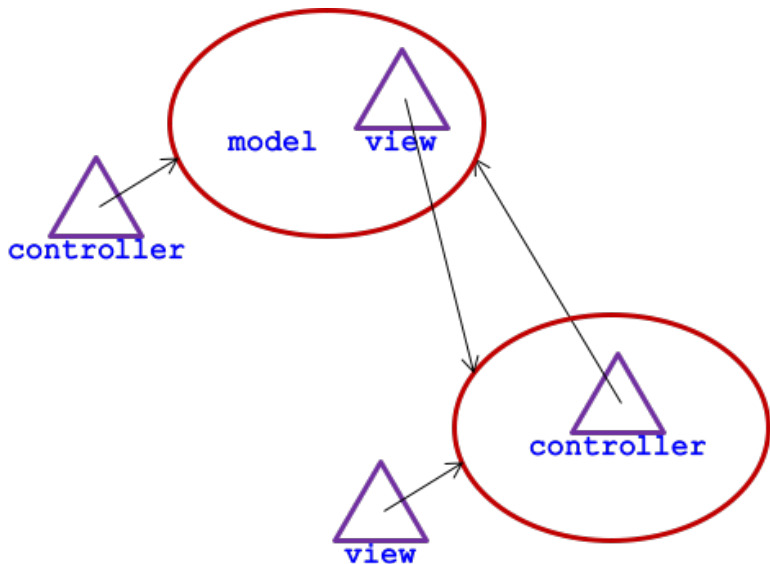
## Set-up by `main`: Create view



## Set-up by `main`: Create controller



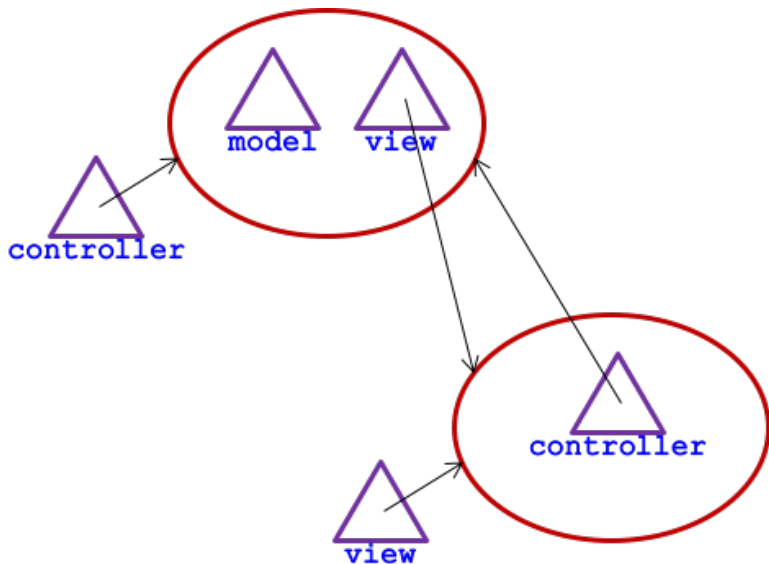
## After `view.registerObserver`



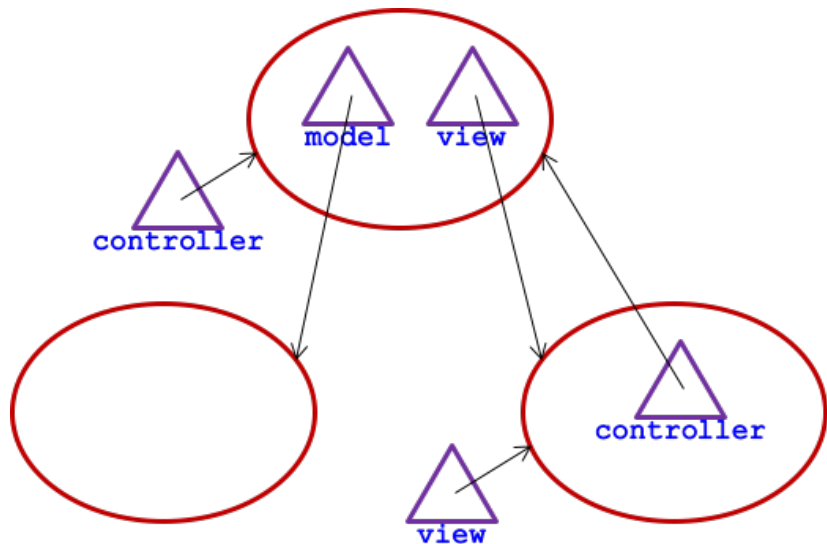
# Now, Who's In Charge?

- ▶ **Note:** when `main` is executed:
  - ▶ Calls all constructors
  - ▶ Registers the controller as an observer of view
  - ▶ Then returns to `main`, and there is no more code
- ▶ After that, what code is executing?
  - ▶ Depends on the program
  - ▶ A GUI based program will then wait for user interactions (**View** handles with the **Observer Pattern**)
  - ▶ Other code may pass control to the controller object

## After Controller declares Model object



## After calling Model constructor

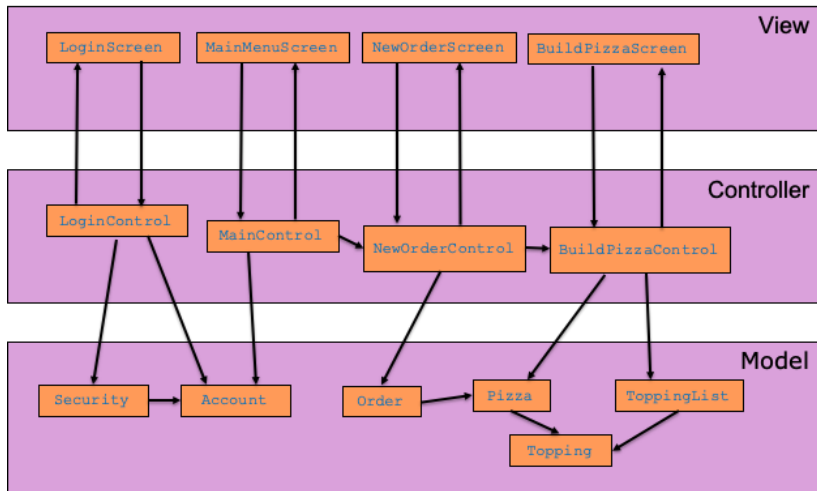


# A More Complicated Example

- ▶ Let's consider a Point of Sale (POS) system for a Pizzeria
- ▶ Used by the employees to add and track orders

# POS System for a Pizzeria

## Model-View-Controller



# MVC and GUIs

- ▶ After exam 2, we will discuss graphical user interfaces
- ▶ We will see how Java uses the **Model-View-Controller** design pattern to create and work with GUIs using the Java [Swing](#) Library

# Homework Assignment

- 1 Programming assignment 4 is posted on your lab Canvas page.

# Exam #2

## 1 Exam #2 - April 10, 2023

- ▶ Please complete the Lockdown Browser + Monitor Practice Quiz as part of your participation grade when it opens up. More information will be announced on Canvas.
- ▶ Will post review slides soon!

# Summary

- 1 Model-View-Controller (MVC) Pattern
- 2 Changing an user interface
- 3 Design Patterns and GUIs
  - ▶ Illustrative Example
- 4 Allowing Controller to display output
- 5 Simple MVC GUI Demo
- 6 Our `main` method
- 7 POS System for a Pizzeria