CS 5004

LECTURE 10

MVC – CONTROLLER & VIEW

KEITH BAGLEY

SPRING 2022

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

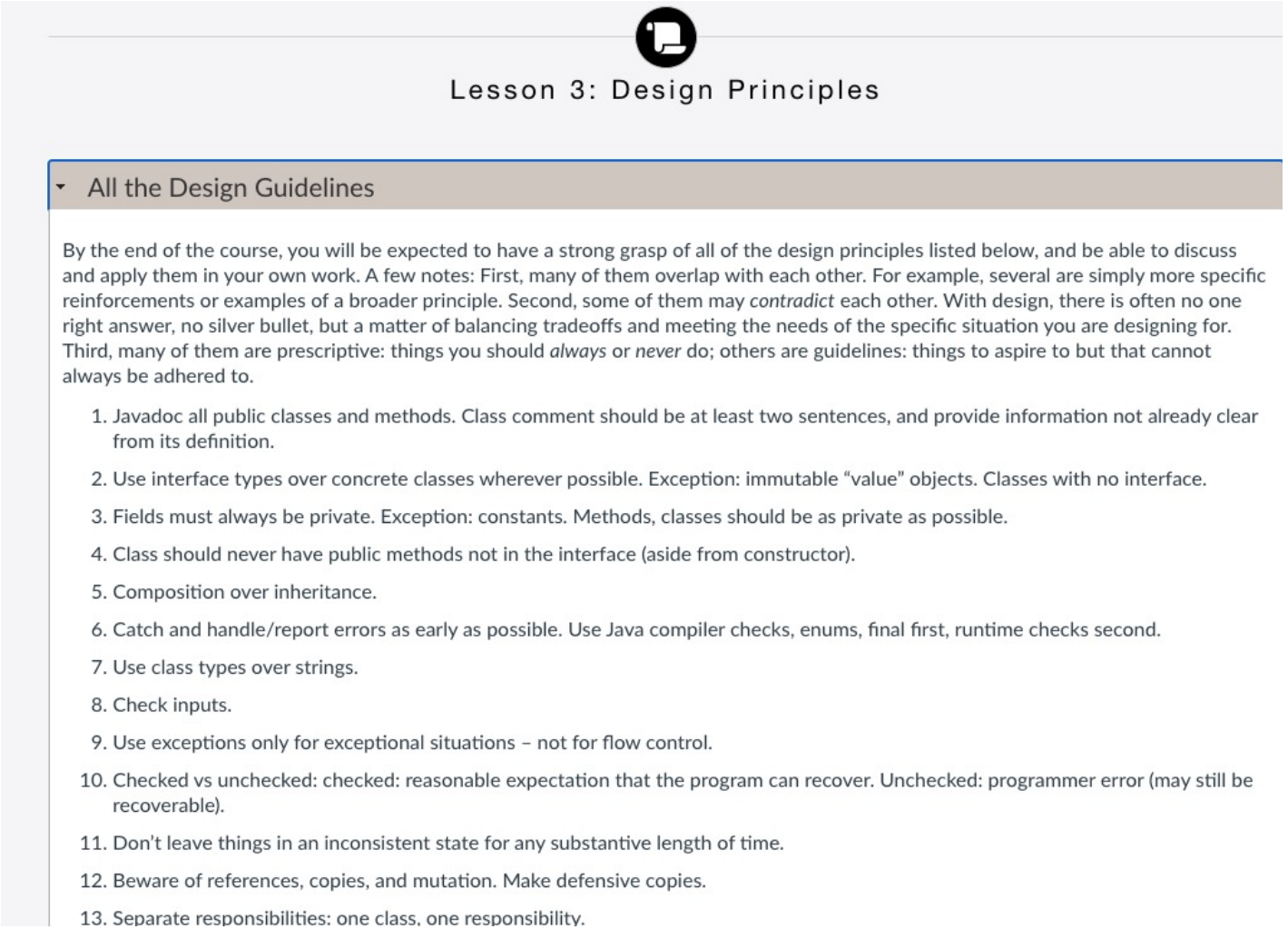
AGENDA

- Review of Design Principles (find them in Module 0 Lesson 3)
- MVC Review
- Controller
- View
- Stepwise code from blob to MVC & Text View
- GUI View with Swing
- Breakout – Enhance the Calculator
- Brief look at JavaFx
- Design Patterns Command

CHECK-IN

- Almost There!
  - 2 more lectures after today!
    - Finishing Design Patterns & Course Review
  - Final “assigned lab” this week; next week is open working Q/A session for homework
  - Final homework assigned this week
- Will be making some “accommodations” for grading

DESIGN PRINCIPLES



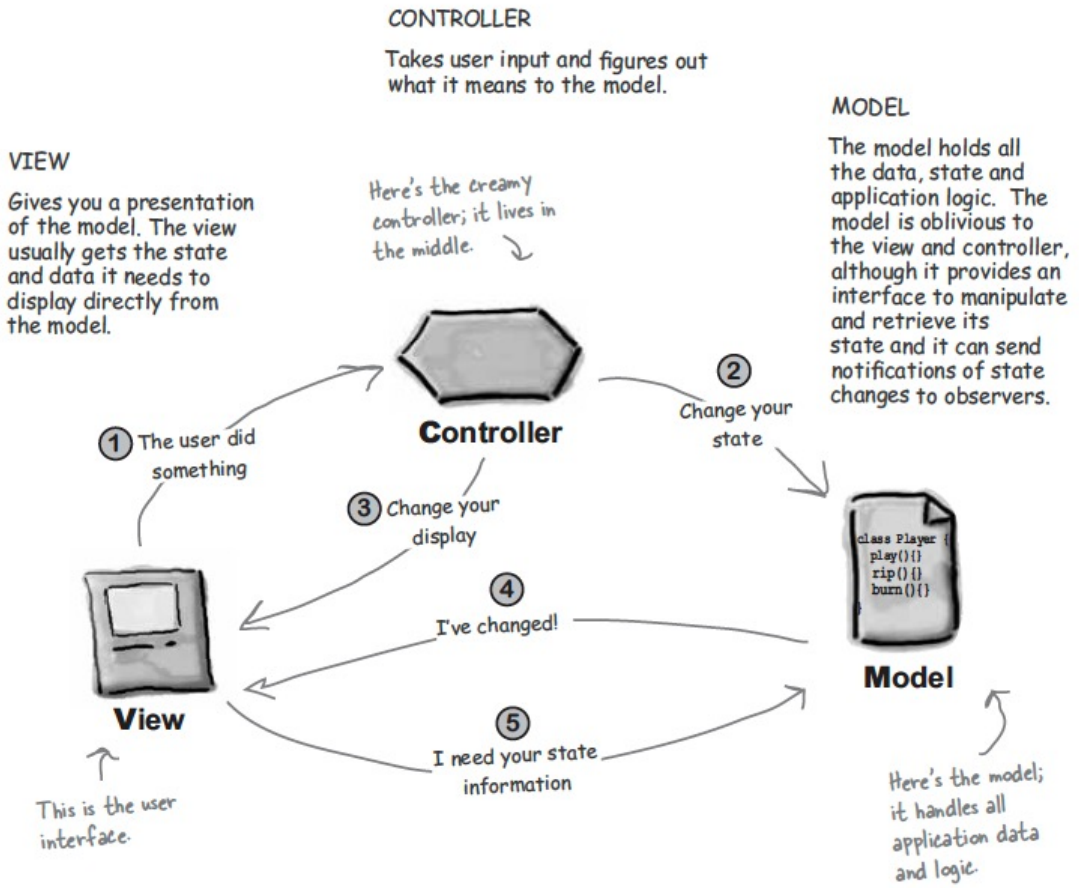
Lesson 3: Design Principles

All the Design Guidelines

By the end of the course, you will be expected to have a strong grasp of all of the design principles listed below, and be able to discuss and apply them in your own work. A few notes: First, many of them overlap with each other. For example, several are simply more specific reinforcements or examples of a broader principle. Second, some of them may *contradict* each other. With design, there is often no one right answer, no silver bullet, but a matter of balancing tradeoffs and meeting the needs of the specific situation you are designing for. Third, many of them are prescriptive: things you should *always* or *never* do; others are guidelines: things to aspire to but that cannot always be adhered to.

1. Javadoc all public classes and methods. Class comment should be at least two sentences, and provide information not already clear from its definition.
2. Use interface types over concrete classes wherever possible. Exception: immutable "value" objects. Classes with no interface.
3. Fields must always be private. Exception: constants. Methods, classes should be as private as possible.
4. Class should never have public methods not in the interface (aside from constructor).
5. Composition over inheritance.
6. Catch and handle/report errors as early as possible. Use Java compiler checks, enums, final first, runtime checks second.
7. Use class types over strings.
8. Check inputs.
9. Use exceptions only for exceptional situations – not for flow control.
10. Checked vs unchecked: checked: reasonable expectation that the program can recover. Unchecked: programmer error (may still be recoverable).
11. Don't leave things in an inconsistent state for any substantive length of time.
12. Beware of references, copies, and mutation. Make defensive copies.
13. Separate responsibilities: one class, one responsibility.

MVC REVIEW



Source: Head First Design Patterns, 2014

1. **You're the user — you interact with the view.**
The view is your window to the model. When you do something to the view (like click the Play button) then the view tells the controller what you did. It's the controller's job to handle that.

2. **The controller asks the model to change its state.**
The controller takes your actions and interprets them. If you click on a button, it's the controller's job to figure out what that means and how the model should be manipulated based on that action.

3. **The controller may also ask the view to change.**
When the controller receives an action from the view, it may need to tell the view to change as a result. For example, the controller could enable or disable certain buttons or menu items in the interface.

4. **The model notifies the view when its state has changed.**
When something changes in the model, based either on some action you took (like clicking a button) or some other internal change (like the next song in the playlist has started), the model notifies the view that its state has changed.

5. **The view asks the model for state.**
The view gets the state it displays directly from the model. For instance, when the model notifies the view that a new song has started playing, the view requests the song name from the model and displays it. The view might also ask the model for state as the result of the controller requesting some change in the view.

MVC MECHANICS

- Setup
  - Instantiate Model
  - Instantiate View
  - Instantiate Controller. Pass reference to both M and V to C
    - Controller registers with view so view has a reference back to the controller

MVC MECHANICS

- Processing
  - View captures user action
  - View informs controller (calls a method)
  - Controller accesses model.
    - Possible model update due to this access
  - If model state/data changes, Controller asks View to update accordingly

TYPES OF CONTROLLERS

- Synchronous
  - Sequential “get input, pass input to model, get board and pass to view” operations
  - Similar to what is implemented for textual Tic-Tac-Toe game
- Asynchronous
  - Reactive, typically event-driven.
  - Controller handles events depending on user input. Most programs with graphical user interfaces behave this way

ENTRY POINT

- Typically, there is an entry-point (or main) method or class that initiates the MVC/MVP/HMVC/etc. program
- The entry point does not directly implement any functionality (the model does), does not directly interact with the user (the controller does) and does not directly show any results (the view does).
  - The entry point creates the model, view(s) and controller(s) and then gives control to the controller

ENTRY POINT

```
class ProgramRunner {
  public static void main(String[] args) {
    Model model = new Model(...); //set up before if needed
    View view = new View(...);
    //setup details of view if needed
    Controller controller = new Controller(model,view,...); //give controller the model and view
    controller.go(); //give control to the controller. Controller relinquishes only when program ends
  }
}
```

CONTROLLER HOOKED INTO ENTRY POINT

```
8
9       class HardwiredController implements ICalcController {
0           public void go(ICalculator calc) {
1               Objects.requireNonNull(calc);
2               int num1, num2;
3               System.out.println("Enter two numbers to add");
4               System.out.print(">>> ");
5               Scanner scan = new Scanner(System.in);
6               num1 = scan.nextInt();
7               num2 = scan.nextInt();
8               System.out.print(calc.add(num1, num2));
9           }
0       }
1
2       public class ControllerHardwired {
3           public static void main(String[] args) {
4               new HardwiredController().go(new CalculatorModel());
5           }
6       }
```

Applying this to our calculator

SEPARATION OF CONCERNS & DECOUPLING

- Controllers (1) take inputs from the user, and (2) delegate to the model and the view.
  - In a well-designed controller, the input from the user is handled and used in a manner that is independent of the method used to get that input from the user.
    - Example: The user enters a string - the functionality of the program should not change regardless of the data coming from the command line or from a text box on a form

ABSTRACTION & DECOUPLING CONTROLLERS

- We often abstract the input, so it can work with many data sources.
- Similarly the controller can abstract the output so that it can work with many data sinks.
- For testing purposes we can attach a data source that allows us to programmatically feed input (e.g. from a file or a string) and parse the output (e.g. output written to a file or a string) to verify the testing objectives.
  - We can call this “synthetic data”
- For the actual application we can attach the required data source (e.g. keyboard) and data sink (e.g. console)

DECOUPLED CONTROLLER

```
12  public class ControllerDecoupled {
13    public static void main(String[] args) {
14      try {
15        // new DecoupledController(new InputStreamReader(System.in), System.out).go(new CalculatorModel());
16        new DecoupledController(new StringReader( s: "+ 2 3 - 5 1 q"), System.out).go(new CalculatorModel());
17
18      } catch (IOException e) {
19        e.printStackTrace();
20      }
21    }
22  }
23
24  class DecoupledController implements ICalcController {
25    final Readable in;
26    final Appendable out;
27    DecoupledController(Readable in, Appendable out) {
28      this.in = in;
29      this.out = out;
30    }
31    public void go(ICalculator calc) throws IOException {
32      Objects.requireNonNull(calc);
33      int num1, num2;
34      Scanner scan = new Scanner(this.in);
35      while (true) {
36        switch (scan.next()) {
37          case "+":
38            num1 = scan.nextInt();
            num2 = scan.nextInt();
```

Abstracting source and sink for data

TESTING THE CONTROLLER

```java
import controllers.DecoupledController;
import controllers.ICalcController;
import model.CalculatorModel;

import static org.junit.Assert.assertEquals;

public class DecoupledControllerTest {
  @Test
  public void testGo() throws Exception {
    StringBuffer out = new StringBuffer();
    Reader in = new StringReader( s: "+ 3 4 + 8 9 q");
    ICalcController controller = new DecoupledController(in, out);
    controller.go(new CalculatorModel());
    assertEquals( expected: "7\n17\n", out.toString());
  }
}
```

Using a simple StringBuffer as our “sink” we can store the output passing through the controller and check it for validity

TESTING THE CONTROLLER IN ISOLATION

- The previous tests verifies two things: whether the model works correctly and that the controller receives inputs and transmits the result produced by the model.
- This is redundant, because we would have tests to separately verify the model's correctness.
- The (strategically buggy) controller shown here will still pass these tests

```
class BuggyController6 implements CalcController {
  final Readable in;
  final Appendable out;
  BuggyController6(Readable in, Appendable out) {
    this.in = in;
    this.out = out;
  }
  public void go(Calculator calc) throws IOException {
    Objects.requireNonNull(calc);
    int num1, num2;
    Scanner scan = new Scanner(this.in);
    while (true) {
      switch (scan.next()) {
        case "+":
          num1 = scan.nextInt() + 10;  //wrong input, bug
          num2 = scan.nextInt() - 10; //wrong input, but this bug nullifies above bug
          this.out.append(String.format("%d\n", calc.add(num1, num2)));
          break;
        case "q":
          return;
      }
    }
  }
}
```

TESTING THE CONTROLLER IN ISOLATION

- A better way to test the controller would be by isolating it to verify that the controller, *in isolation*, works correctly
  - It works correctly if it reads the inputs in the correct sequence, sends them to the model correctly, and transmits any outputs to the view correctly.
- In order to isolate the controller, we provide it with a “mock” model. A “mock” of an object is another object that “looks like the real object, but is simpler”.
  - We create an explicit interface for the model for the above example We then make our model object implement this interface.

TESTING THE CONTROLLER IN ISOLATION

```
11  public class MockModelTest {
12    @Test
13    public void testGo() throws Exception {
14      StringBuffer out = new StringBuffer();
15      Reader in = new StringReader( s: "+ 3 4 + 8 9 q");
16      ICalcController controller = new ControllerMock(in, out);
17      StringBuilder log = new StringBuilder(); //log for mock model
18      controller.go(new MockModel(log, uniqueCode: 1234321));
19      assertEquals( expected: "Input: 3 4\nInput: 8 9\n", log.toString()); //inputs reached the model correct
20      assertEquals( expected: "1234321\n1234321\n",out.toString()); //output of model transmitted correctly
21
22      out = new StringBuffer();
23      in = new StringReader( s: "- 5 1 q");
24      controller = new ControllerMock(in, out);
25      log = new StringBuilder(); //log for mock model
26      controller.go(new MockModel(log, uniqueCode: 111222));
27      assertEquals( expected: "Input: 5 1\n", log.toString()); //inputs reached the model correctly
28      assertEquals( expected: "111222\n",out.toString()); //output of model transmitted correctly
```

MOCK MODELS

- The mock does not actually add numbers: it merely logs the inputs provided to it, and returns a unique number provided to it at creation
- Mocks are used with “synthetic” (artificial) data to ensure the controller (or any other component) works properly.
  - Testing data transmission and valid passing of values, particularly through endpoints
  - Mocks do not test whether the controller-model combination produced the correct answer. However this test, along with those for the model, collectively verify the correctness of this combination.

VIEWS

- Displays results to user
- Doesn’t care how the results are produced
- Doesn’t care when it shows the results (it’s told to update accordingly)

THERE MAY BE ONE OR MANY VIEWS

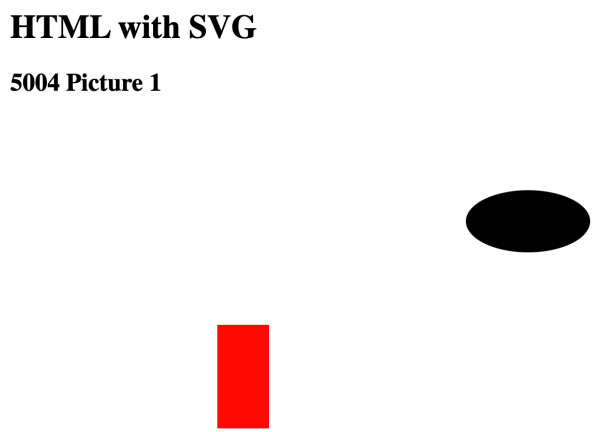
- Different views on the same model on the same platform
  - Graphical view of shapes
  - Spreadsheet view of shape data
- Different views of the same model on different platforms
  - Console view
  - Native GUI view
  - Web view
- Different views for logical models in different language implementations
  - E.g. Java, Python, Ruby, etc.
- etc

DESCRIPTIVE VIEWS

- HTML and SVG
  - HTML is a markup language that defines the structure of web content (mainly static)
  - SVG (Scalable Vector Graphics) is used to define graphics for the Web
  - The HTML <svg> element is a container for SVG graphics.
  - SVG has several methods for drawing paths, boxes, circles, text, and graphic images
    - See https://www.w3schools.com/html/html5_svg.asp for documentation and examples

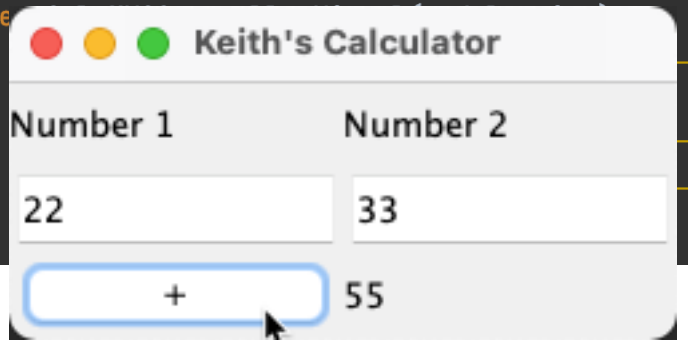
A VIEW FOR THE WEB

```
<!DOCTYPE html>
<html>
<body>
<h1>HTML with SVG</h1>
<div>
    <h2>5004 Picture 1</h2>
    <svg width="1000" height="1000">
        <rect id="R" x="200.0" y="200.0" width="50.0" height="100.0" fill="rgb(255,10,0)">
        </rect>
        <ellipse id="O" cx="500.0" cy="100.0" rx="60.0" ry="30.0" fill="rgb(0,0,1)">
        </ellipse>
    </svg>
</div>
<div>
```



A “NATIVE” SWING VIEW

```
Calculator.java  CalculatorWithModel.java  VisualCalculator.java  CalcTextMVC.java
 2  import java.awt.event.ActionEvent;
 3  import java.awt.event.ActionListener;
 4  import java.io.IOException;
 5  import java.io.InputStreamReader;
 6
 7  import javax.swing.*;
 8
 9  public class VisualCalculator {
10    public static void main(String[] args) throws IOException {
11      CalculatorModel model = new CalculatorModel();
12      SwingView view = new SwingView( caption: "Keith's Calculator");
13      CalcMVCControllerVisual controller = ne
14      controller.go();
15    }
16  }
17
18  class SwingView extends JFrame {
19    private JButton btnAdd;
```



LET'S EXAMINE SOME VIEW CODE

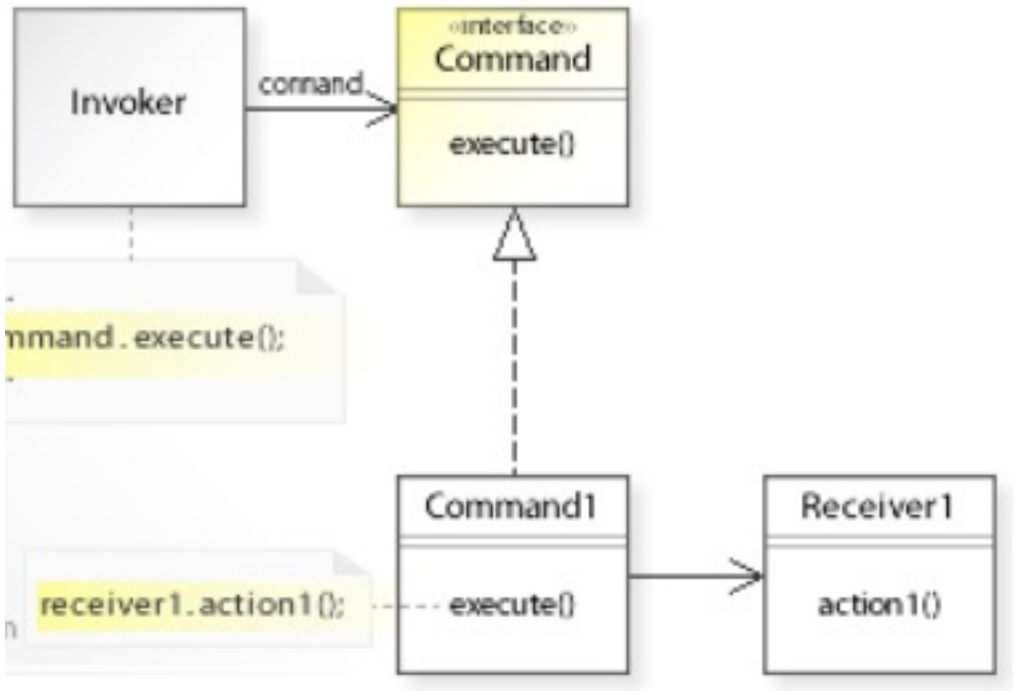
JAVAFX – THAT OTHER LIBRARY

JAVAFX – THAT OTHER LIBRARY

- All of the assets for the project are Swing-focused
- Swing is easier to get started and using
- Swing is no longer being developed, and is effectively on oxygen
- JavaFx did NOT achieve high rates of adoption
- JavaFx approach is more rationally designed and “cleaner”
- The latest version of JavaFx is much harder to get started with because Oracle had a collective “brain cramp” and made things more difficult than easier
- Neither Swing nor JavaFx is the clear “winner” for implementing “pure Java” user interfaces

COMMAND PATTERN

- Command Pattern encapsulates a request as an object. This allows us to log requests, undo actions, and track simple commands for more complex classes



COMMAND PATTERN

- Discussion: Consider your chess piece program.
  - If you were building a complete chess game, how might the command pattern help?

```
// psuedocode
public class Move {
  Position oldLocation;
  Position newLocation;
  Chesspiece piece;
  public Move(Chesspiece piece, Position newLocation) {
    // create a move command, initialize instance variables
  }
  public boolean execute() {
    if(piece.isValidMove(this.newLocation)) {
      this.piece.move(this.newLocation);
      return true;
    }
    return false;
  }

  public boolean undo() { // take back Move
    // probably need to verify the move executed previously,
    // but you get the point
    this.piece.move(this.oldLocation);
    // etc.
  }
}
```

COMMAND PATTERN

- When simple command tracking, logging or “undoing” is required, the command pattern is useful

```
public class ChessGame {

  // yada yada
  List<Move> moves = new List<>();

  // while playing the ChessGame
  Move currentMove = new Move(currentPiece, newLocation);
  if(currentMove.execute()) {
    moves.add(currentMove);
  }

  // later...we want to replay the game or take back moves...
  Collections.reverse(moves);
  moves.forEach((each)->each.undo());
}
```

Q & A

THANKS!

- Stay safe, be encouraged, & see you next week!