CS 5044
Object-Oriented Programming with Java

Q&A Session

GUI development lineage of Java

- Java v1.0 (1995) Abstract Window Toolkit (AWT)
  - The first major in-language attempt at cross-platform desktop GUI support
  - Looked different on each platform, but one code base behaved the same everywhere
- Java v1.2 (1998) Swing
  - Introduced a "pluggable" look and feel; can look native or platform-independent
  - Still dependent upon AWT, Swing has remained the primary GUI for Java applications
- Java v7 (2008) JavaFX
  - Introduced as the successor to Swing, but is still far less popular
  - JavaFX v8 is now a fundamental component bundled within Java v8 (2014)
    - Much more sophisticated event model and property handling
      - APIs are fully consistent with functional programming (more on this later!)
    - Includes support for touch, sensors, native packaging, and 3D (and a lot more)
    - Open source port (mostly) runs (most) JavaFX applications on both iOS and Android
  - This is likely to be the future, but Swing will probably stick around for quite a long time
    - Even AWT is still alive and well, with no signs of being deprecated
      - Most Swing components essentially "extend" AWT classes and use AWT events
    - JavaFX can easily incorporate Swing components (and vice versa)
      - New applications should probably use only JavaFX at this point

Graphical user interfaces in general

- Most commonly associated with a Model-View-Controller (MVC) architecture
  - *Model* - Not a GUI; the underlying system which is being represented by the interface
    - Accesses or mutates the state of the objects within the system
    - Not generally developed as part of the interface; may not even be aware of the GUI
  - *View* - the facade as presented by all the visible user interface components
    - Represents the layout of all the components on the screen
  - *Controller* - elements that can be manipulated by the user (facade or otherwise)
    - Handles the various events (mouse, action, and so forth) that can be triggered
    - Updates the view whenever event processing cause state changes
- Consider some of our projects as simple examples
  - Project 3 (Tetris Brain)
    - Model: the Board object and all related objects/states (shapes, rotations, etc.)
    - View: the board, the blocks, scores at the top, and a start message
    - Controller: the keyboard and timer handlers that update the display and mode
  - Project 6 (DAB GUI)
    - Model: the DABGame from Project 4
    - View: the button, labels, combo boxes, and DABGrid component
    - Controller: The draw button and menu item handlers (plus others within DABGrid)

MVC View (Swing)

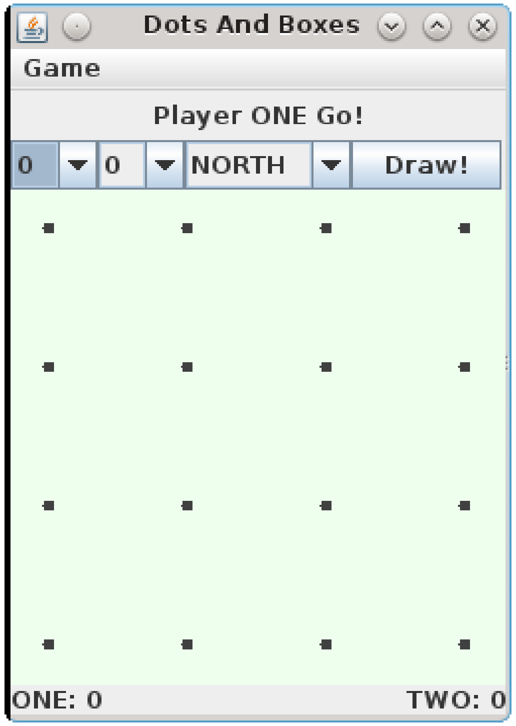
- Visual layout of components; relies upon layout managers for each container
  - Built-in layout managers Includes: Flow, Border, Box, Grid, GridBag, and Group
    - Often complex layouts are designed with help from GUI builders
      - Helps with layout and connecting to methods/fields in your code
      - Usually you still need to do at least some work by hand
        - You can always design the whole thing by hand
  - Generally we only need to combine built-in widget types (such as JButton, JLabel, etc.)
    - You can also define custom-drawn components, to render other parts of the interface
      - This is usually only necessary for games or other specialized systems
      - Tetris GUI (JavaFX) and the Project 6 DABGrid (Swing) both take this approach
- The Swing toolkit is fairly robust
  - Containers contain components or other containers, forming a tree
    - Components cover all the familiar user interface widgets and menu systems
  - You can choose native or one of several platform-independent look and feel choices
  - Swing has been around, in the same essential form, for nearly 20 years now
    - Many tutorials, best practices, and lessons learned are readily available

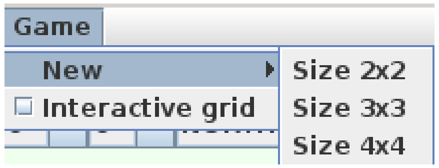
MVC Controller (Swing)

- Includes all input to the application, often as presented by the View component
- The Swing system itself is in (nearly) complete control of the application
  - Events are managed as objects in an event queue, each handled by your controller
  - Your controller code is developed as a set of call-back methods that react to events
    - Your code handles the event, then implicitly returns control back to Swing
    - We're somewhat used to this; in TDD, the test methods "control" the application
- GUI applications are necessarily event-driven in nature
  - Components trigger various events upon user interaction via mouse/keyboard
  - Timers can also be set to trigger events without user interaction
- Controller code can access (and mutate) any part of the View or the Model
  - However, note that long-running tasks can become very problematic
    - The UI will remain unresponsive while your controller is handling any event
    - The solution is asynchronous processing on one or more parallel execution threads
      - This is actually fairly easy to achieve in Java (but that's another course!)

Project 6 - DAB GUI

- We're adding a graphical user interface to our DABGame!

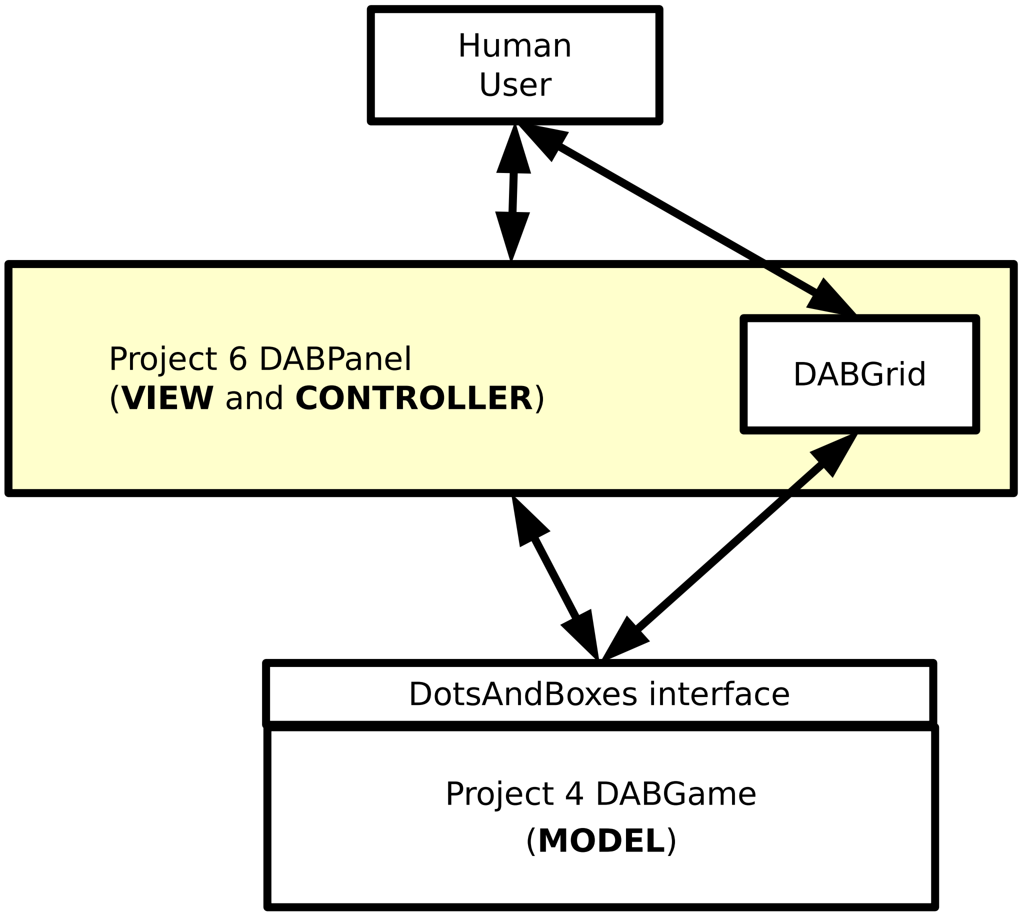




Project 6: DABGame (MVC "Model")

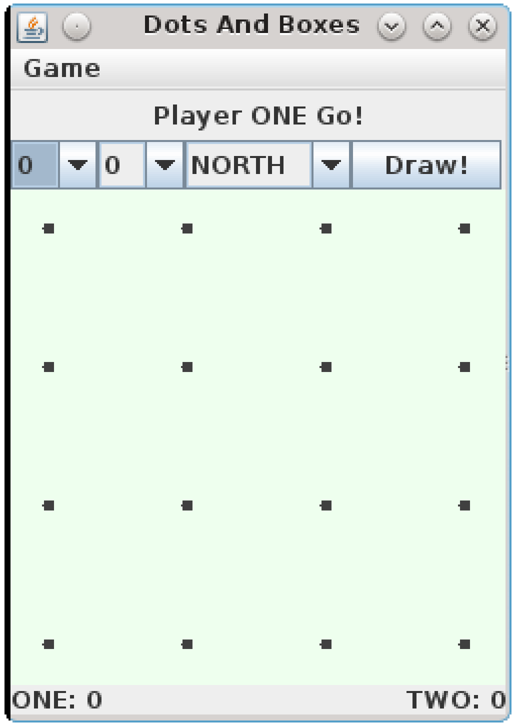
- Project 6 depends entirely upon the Project 4 DABGame as the Model
  - Please don't worry if your Project 4 is incomplete or otherwise not working properly
    - A fully operational reference implementation is available for all to download
    - The reference solution passes all the Web-CAT tests (source code not included)
    - It's easy to switch between your implementation and the reference implementation
- DABGame doesn't need any changes to act as the model
  - View and Controller both interact with the Model via the DotsAndBoxes interface

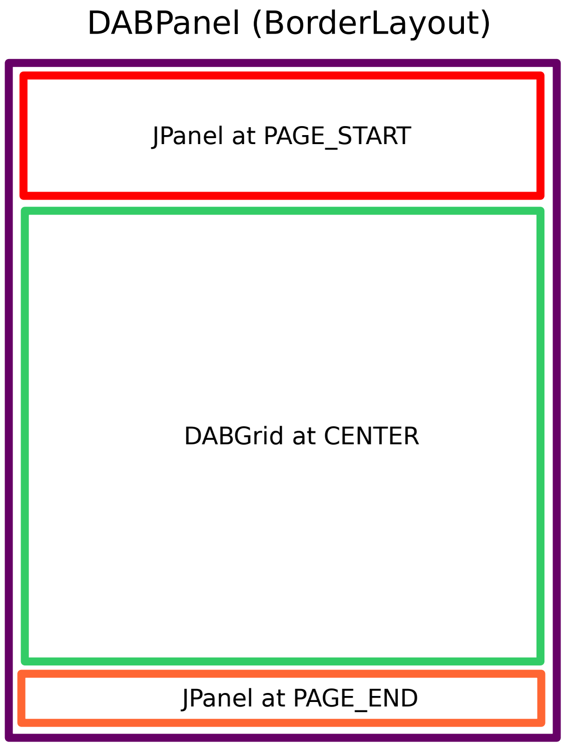
Project 6 - MVC


Human User
Project 6 DABPanel (**VIEW** and **CONTROLLER**)
DABGrid
DotsAndBoxes interface
Project 4 DABGame (**MODEL**)

Project 6: Component layout (MVC "View")

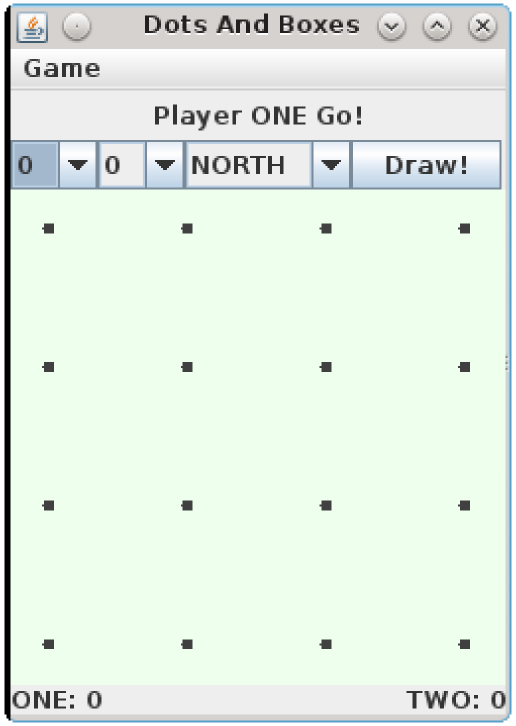
- This is just a **suggestion** (you don't need to replicate this exact layout)
  - Any reasonable arrangement of the components is perfectly acceptable

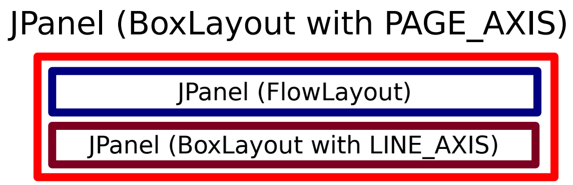




Project 6: Component layout (MVC "View")

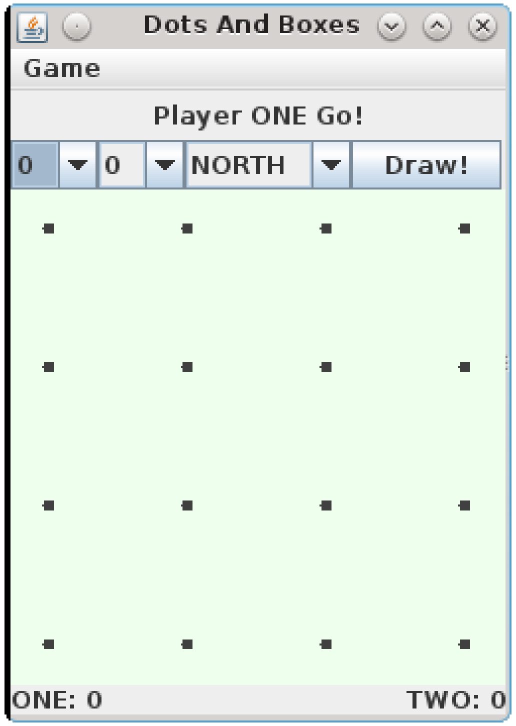
- This is just a **suggestion** (you don't need to replicate this exact layout)
  - Any reasonable arrangement of the components is perfectly acceptable




JPanel (BoxLayout with PAGE_AXIS)
JPanel (FlowLayout)
JPanel (BoxLayout with LINE_AXIS)

Project 6: Component layout (MVC "View")

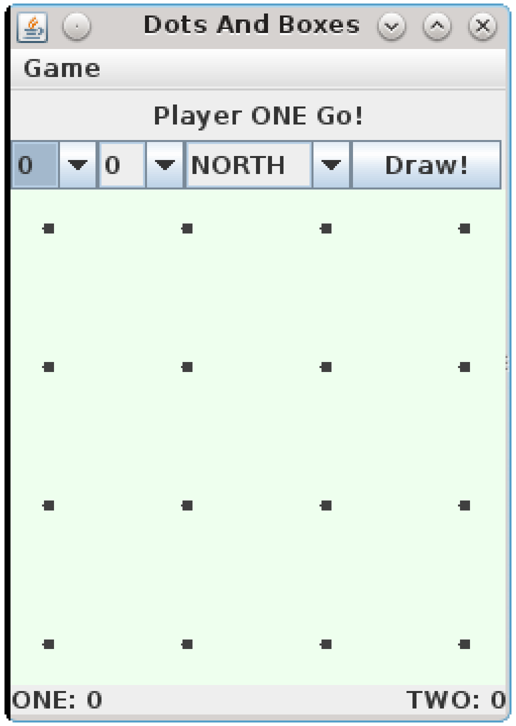
- This is just a **suggestion** (you don't need to replicate this exact layout)
  - Any reasonable arrangement of the components is perfectly acceptable



DABGrid component (no layout)

Project 6: Component layout (MVC "View")

- This is just a **suggestion** (you don't need to replicate this exact layout)
  - Any reasonable arrangement of the components is perfectly acceptable

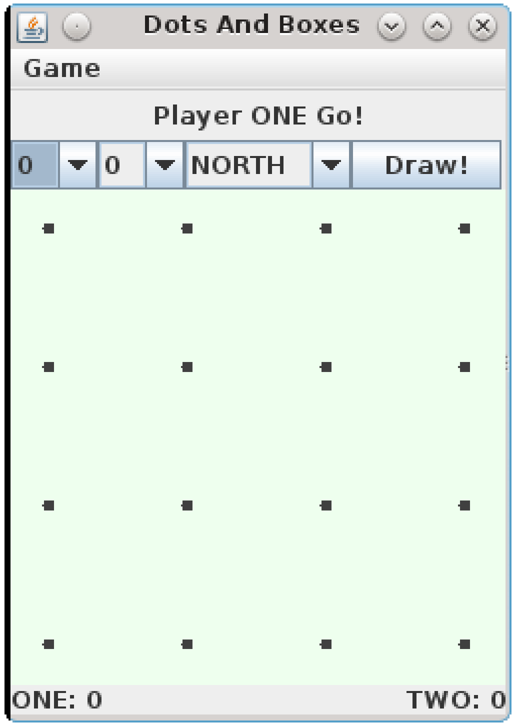


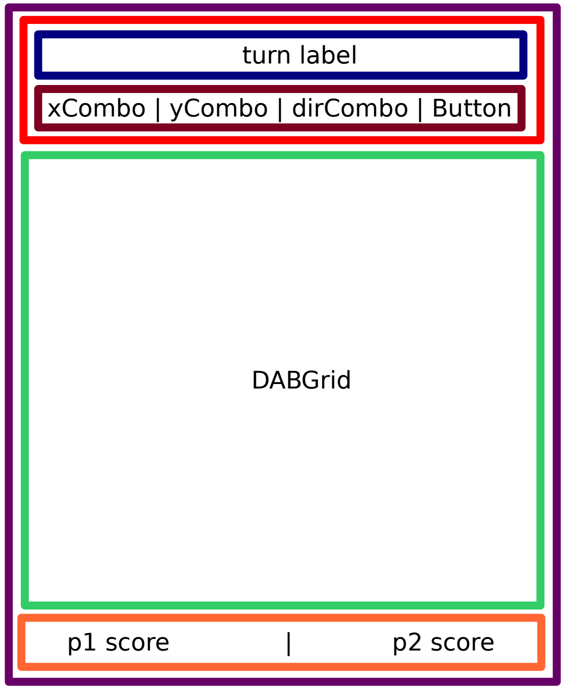
JPanel (BorderLayout)

label at LINE_START label at LINE_END

Project 6: Component layout (MVC "View")

- This is just a **suggestion** (you don't need to replicate this exact layout)
  - Any reasonable arrangement of the components is perfectly acceptable





Project 6: Event handling (MVC "Controller")

- Interactive components:
  - Menu items
    - New Game (three possible sizes)
    - Interactive Mode (toggles activate/deactivate)
  - Draw Button
- Each of these components posts an ActionEvent which we must handle
  - New Game items invoke our startGame() helper, each with an appropriate size parameter
  - Interactive Mode is slightly more complex
    - Proceed based on the isSelected() state of the checkbox item
      - Selected?
        - » Call setCallback() on the DABGrid to notify us when an edge is drawn
        - » The callback implementation will just call our updateStatus() helper
      - Unselected?
        - » Call setCallback(null) on the DABGrid to disable interactivity
  - Draw Button just invokes our handleDrawButton() method
    - This method reads the X, Y, and Direction combo boxes, then calls game.drawEdge()
      - If the return from drawEdge() is true, call our updateStatus() helper

Adding event listeners

- We'll cover the new functional programming aspects of Java 8 in depth next week
  - These aspects introduce an entirely new syntax into the Java language
- For now, there are two features that will be particularly convenient for Project 6
  - Lambda Expressions
    - Can take the place of certain methods and classes
      - Particularly for anonymous inner classes, which are very common in GUI code
  - Method References
    - Can take the place of certain Lambda Expressions, when parameters can be inferred
      - Also very common in GUI code
- Don't worry about the details at this point
  - All the code you need is in the next few slides

Menu action listeners (classic technique)

- Three "new game" menu items, each of which needs an `ActionListener` assigned

- This works exactly as expected, but it's extremely verbose, fragile, and awkward
  - Each item needs an action command String, which is then detected in the listener
  - Don't use this approach!

- As an alternative, we could create additional classes, but that makes it even worse
  - Don't use this approach!

- If only there were a better way...
  - Java 8, to the rescue!

```java
public class DABPanel extends JPanel implements ActionListener {
    private JMenuBar createMenuBar() {
        // ... other code ...
        JMenuItem menuItemNewGame2 = new JMenuItem();
        menuItemNewGame2.setActionCommand("start2");
        menuItemNewGame2.addActionListener(this);

        JMenuItem menuItemNewGame3 = new JMenuItem();
        menuItemNewGame3.setActionCommand("start3");
        menuItemNewGame3.addActionListener(this);

        JMenuItem menuItemNewGame4 = new JMenuItem();
        menuItemNewGame4.setActionCommand("start4");
        menuItemNewGame4.addActionListener(this);
        // ... other code ...
    }

    @Override
    public void actionPerformed(ActionEvent ae) {
        if ("start2".equals(ae.getActionCommand())) {
            startGame(2);
        } else if ("start3".equals(ae.getActionCommand())) {
            startGame(3);
        } else if ("start4".equals(ae.getActionCommand())) {
            startGame(4);
        }
    }
```

Menu action listeners (lambda expression)

- Java 8, to the rescue!
- Use a Lambda Expression to define an anonymous inner class for each `ActionListener`
- Advantages:
  - Significantly less code
  - No action commands
  - No separate method required
  - No interface to implement
- Disadvantages:
  - Syntax is newer and may be somewhat confusing
  - Controller and View are less clearly separated than before
- Overall a cleaner approach

```java
public class DABPanel extends JPanel {
    private JMenuBar createMenuBar() {
        // ... other code ...
        JMenuItem menuItemNewGame2 = new JMenuItem();
        menuItemNewGame2.addActionListener(e -> startGame(2));

        JMenuItem menuItemNewGame3 = new JMenuItem();
        menuItemNewGame3.addActionListener(e -> startGame(3));

        JMenuItem menuItemNewGame4 = new JMenuItem();
        menuItemNewGame4.addActionListener(e -> startGame(4));
        // ... other code ...
    }
}
```

Button action listener (classic technique)

- The drawButton needs an `ActionListener` assigned
- This works exactly as expected, but it's extremely verbose, fragile, and awkward
  - We need yet another unique action command String
  - Don't use this approach!
- As an alternative, we could create additional classes, but that makes it even worse
  - Don't use this approach!
- If only there were a better way...
  - Java 8, to the rescue!

```java
public class DABPanel extends JPanel implements ActionListener {
    public DABPanel() {
        // ... other code ...
        drawButton = new JButton();
        drawButton.setActionCommand("draw");
        drawButton.addActionListener(this);
        // ... other code ...
    }

    @Override
    public void actionPerformed(ActionEvent ae) {
        if ("draw".equals(ae.getActionCommand())) {
            handleDrawButton(ae);
        }
    }

    private void handleDrawButton(ActionEvent ae) {
        // ... event handler goes here ...
    }

}
```

Button action listener (lambda expression)

- Getting better now!
- Improvements are same as with new game menu items
- Can we do even more?
  - Yes, but only slightly...

```
public class DABPanel extends JPanel {
    public DABPanel() {
        // ... other code ...
        drawButton = new JButton();
        drawButton.addActionListener(e -> handleDrawButton(e));
        // ... other code ...
    }

    private void handleDrawButton(ActionEvent ae) {
        // ... event handler goes here ...
    }
}
```

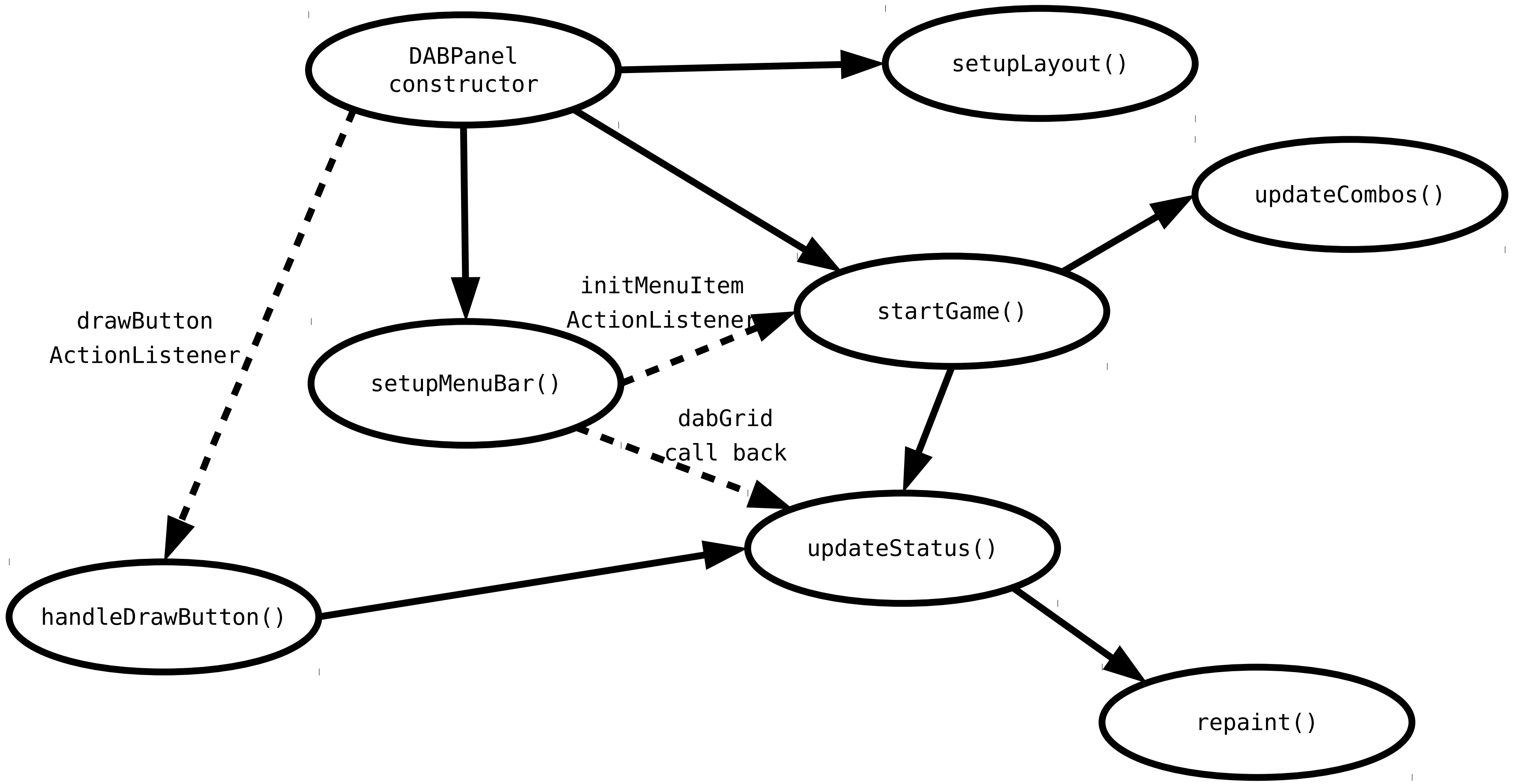
Button action listener (method reference)

- A Method Reference can infer the parameters
- Works exactly like the lambda expression in this case
- Just a cleaner syntax for the same solution, but sometimes even little improvements can make a big difference!

```
public class DABPanel extends JPanel {
    public DABPanel() {
        // ... other code ...
        drawButton = new JButton();
        drawButton.addActionListener(this::handleDrawButton);
        // ... other code ...
    }

    private void handleDrawButton(ActionEvent ae) {
        // ... event handler goes here ...
    }
}
```

Project 6 method interaction overview


DABPanel constructor
setupLayout()
updateCombos()
startGame()
initMenuItem ActionListener
drawButton ActionListener
setupMenuBar()
dabGrid call back
updateStatus()
handleDrawButton()
repaint()