The Bridge Pattern

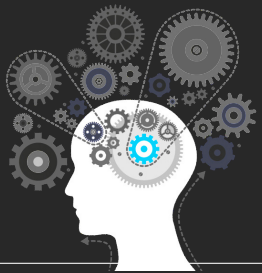
Dr. Isaac Griffith Idaho State University

Outcomes

Idaho State University | Computer Science

After today's lecture you will be able to:

- Further understanding of the design and implementation of the Model-View-Controller Pattern.
- Understanding and application of the Bridge Design Pattern.



Subsystem Design

CS 2263

Design of the Subsystems

Idaho State University | Computer Science

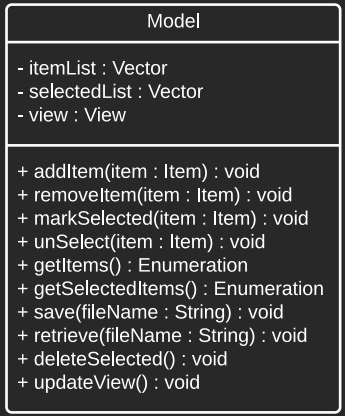
- We return ourselves to the discussion of the Drawing Program
  - Towards this end, we will consider the design of the
    - Model Subsystems
    - Item and Subclasses

Design of the Model Subsystems

Idaho State University | Computer Science

We now consider the structure of the model, which must support the following operations:

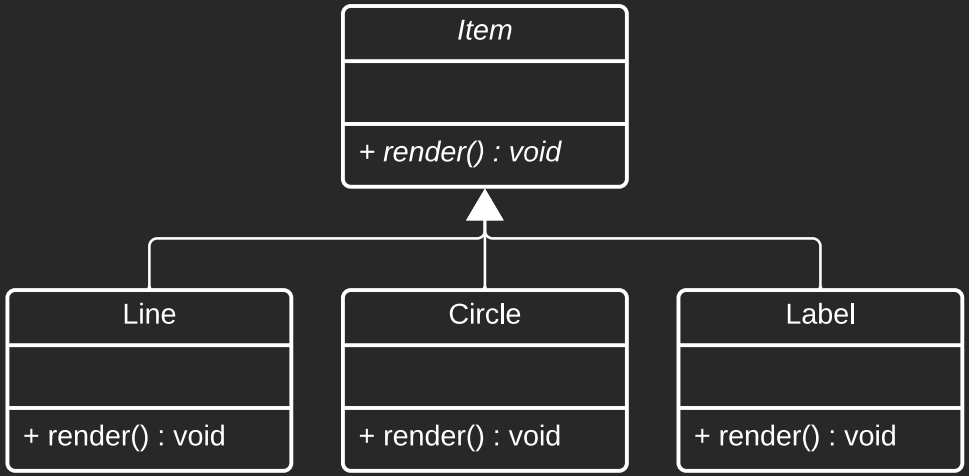
- Adding an item
- Removing an item
- Marking an item as selected
- Unselecting an item
- Getting an enumeration of selected items
- Getting an enumeration of unselected items
- Deleting selected items
- Saving the drawing
- Retrieving the drawing



Design of Item and Subclasses

- It should be evident that `Item` will have a subclass for each shape
  - Each of these will store the relevant attributes for that shape
- **Rendering the Items**
  - A tricky issue is the design of how items are to be rendered (process by which the data stored in the model is displayed in the view)
  - This depends on the following two items:
    - The technology and tools used in creating the UI (known at compile time)
    - The item that is stored (known at runtime)
  - There are two options:
    1. View is responsible for rendering
    2. `Item` is responsible for rendering and has a `render()` method
  - The first option, requires that we query each object to apply the right methods (seems to not be very object-oriented)
  - The second option, packs the methods with the data they use (in this case the `render()` method, a very OO way of doing things)

Structure of Item



Bridge Pattern

CS 2263

Catering to Multiple UI Toolkits

- Java has many UI Toolkits to choose from to perform the rendering:
  - AWT (Abstract Windowing Toolkit) - The original Java UI Toolkit
  - Swing
  - SWT and JFace
  - JavaFX
  - and many more
- For the purposes of this part of the lecture we assume that we are targeting Swing and two additional new toolkits
  - HardUI
  - EasyUI

Catering to Multiple UI Toolkits

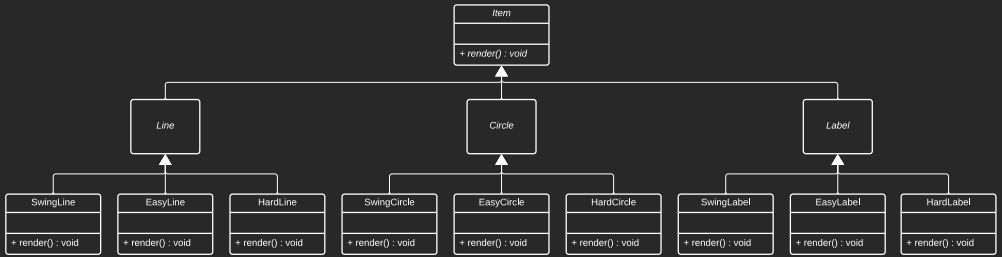
Idaho State University | Computer Science

- To target these toolkits, we need to adapt our `render` method to the specific of each UI
- This can easily be accomplished by creating an inheritance hierarchy for item, in which each shape has three versions.
  - In each case, the render method invokes the methods available in the UI while acquiring the necessary contextual information
  - An example is the `SwingCircle`:

```
public class SwingCircle extends Circle {
  // circle class for SwingUI
  public void render() {
    Graphics g = (View.getInstance()).getGraphics();
    g.drawOval(/* params */)
  }
}
```

Catering to Multiple UIs

- This implies a need for platform independent classes for each type of component (abstract classes)
  - which have a child class per UI type
- The drawback here is we need the following number of classes:
  Number of types of items $\times$ *Number of UI packages*



- Of course this is an untenable explosion in the number of classes

Bridge Pattern

- This issue at hand is that we actually are dealing with two subsystems
  - **Model:** The types of items (internal variation)
  - **View:** The types of UIs (external variation)
- Our goal is to factor out the external variations
  - The standard solution for this is the **Bridge Pattern**

Using Bridge Pattern

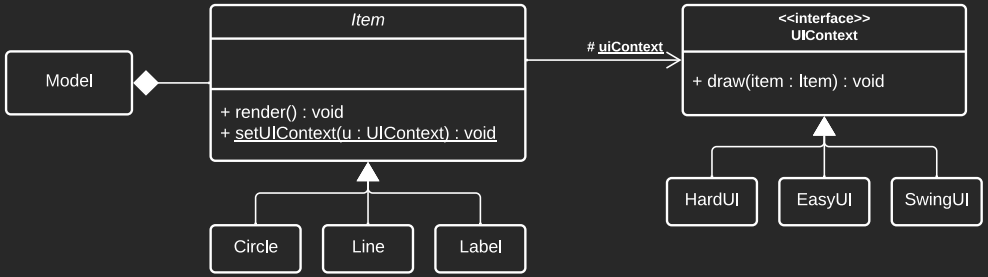
Idaho State University | Computer Science

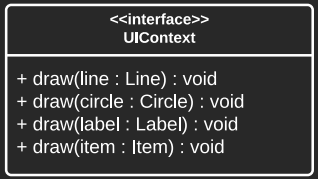
- Pattern Intent: **decouple an abstraction from its implementation so that the two can vary independently**
  - Abstraction: the class `Item`
  - Implementation: the different UIs used by the `render` method
- If we create a single inheritance hierarchy blending both the abstraction and implementation we have the following drawbacks
  - low reuse, as neither can be modified or reused independently
  - if the variations come from independent sources there will be a multiplicative effect on the number of concrete classes

Using Bridge Pattern

- The Bridge Pattern solves this by:
  - separating the abstractions and implementations into two independent inheritance hierarchies
  - provides a permanent binding between them such that changes in implementation do not affect clients
  - reducing the number of classes to *Number of types of items* + *Number of UI packages*
  - allowing multiple classes to share the same representation
- Returns us to the principle: **Favor composition over inheritance**

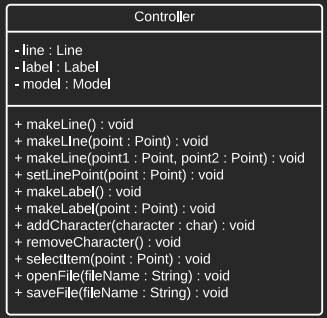
Bridge Pattern Structure





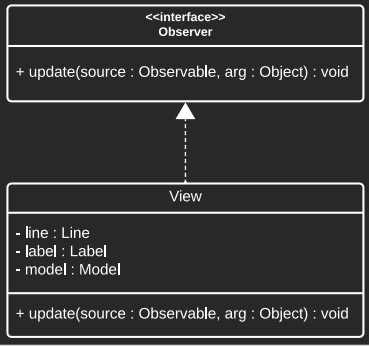
Controller Design

- The controller, unlike the view, will be designed to be unique to the drawing program and not a specific view.
  - As the controller receives input from the view to define the different items it must be capable of the following
    - Remembering the model
    - Storing the current line, label, or circle being created
  - Thus, the controller needs 4 fields
    - a `Line`
    - a `Circle`
    - a `Label`
    - a `Model`
  - Additionally we need methods which
    - Work to construct the items
    - Open or save the file



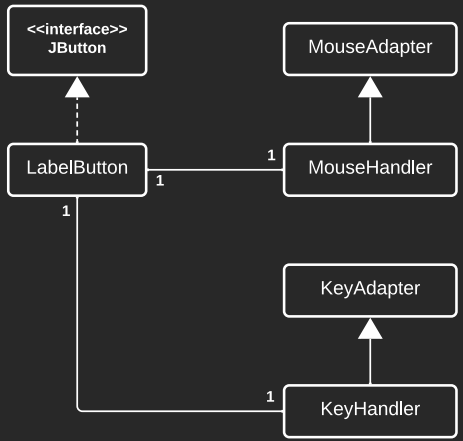
View Design

- Although, the MVC pattern allows the subsystems to be largely independent
- The view is affected by the controller and model in two important ways
  1. The view provides the mechanism to refresh the display when the model changes
  2. The view employs specific technology for constructing the UI
- The first issue is handled using the observer pattern
- The second requires more consideration
  - The trick is how and when does the view set the UIContext of the model
  - We would normally do this immediately before refreshing the view
  - The only issue is when there are multiple views updating simultaneously



Accepting Input

- The last piece of design is defining how we will handle input.
- We know that the user issues commands by clicking on buttons
- We also know that the items are created using a combination of mouse and key events on the drawing panel
- To accomodate this, we construct a specialization of JButton for each of our items
  - Within this class we create both a `MouseHandler` and `KeyHandler` to process the events as they occur
  - Additionally, we override the `addMouseListener` and `addKeyListener` methods of the drawing panel to ensure there is only ever one handler attached



Implementation

CS 2263

Item


```
import java.io.*;
import java.awt.*;

public abstract class Item implements Serializable {
  protected static UIContext uiContext;

  public static void setUIContext(UIContext ctx) {
    Item.uiContext = ctx;
  }

  public abstract boolean includes(Point point);

  protected double distance(Point point1, Point point2) {
    double xDiff = point1.getX() - point2.getX();
    double yDiff = point1.getY() - point2.getY();
    return ((double) (Math.sqrt(xDiff * xDiff +
                      yDiff * yDiff)));
  }

  public void render() {
    uiContext.draw(this);
  }
}
```

```
public class Line extends Item {
  private Point point1;
  private Point point2;

  public Line(Point point1, Point point2) {
    this.point1 = point1;
    this.point2 = point2;
  }

  public Line(Point point1) {
    this.point1 = point1;
  }

  public Line() {}

  public boolean includes(Point point) {
    return ((distance(point, point1) < 10.0) ||
            (distance(point, point2) < 10.0));
  }

  public void render() {
    uiContext.draw(this);
  }
```

SwingUI


```
public class SwingUI implements UIContext {
  private Graphics g;
  // fields for context variables

  public void setGraphics(Graphics grahpics) {
    g = graphics;
  }

  // other methods to set context variables

  public void draw(Circle circle) {
    g.drawOval(/* params */);
  }

  public void draw(Line line) {
    g.drawLine(/* params */);
  }

  public void draw(Label label) {
    g.drawString(/* params */);
  }

  public void draw(Item item) {
    // error
  }
```

Model Class

Idaho State University | Computer Science

```
public class Model extends Observable {
  private Vector itemList;
  private Vector selectedList;
  private static UIContext uiContext;

  public Model() {
    itemList = new Vector();
    selectedList = new Vector();
  }

  public static void setUIContext(
              UIContext uiContext) {
    Model.uiContext = uiContext;
    Item.setUiContext(uiContext);
  }

  public void updateView() {
    setChanged();
    notifyObservers(null);
  }

  public void addItem(Item item) {
    itemList.add(item);
    updateView();
  }

  public void markSelected(Item item) {
    if (itemList.contains(item)) {
      itemList.remove(item);
      selectedList.add(item);
      updateView();
    }
  }

  public void deleteSelectedItems() {
    selectedList.removeAllElements();
    updateView();
  }

  public Enemeration getItems() {
    return itemList.elements();
  }
}
```

Controller Class

Idaho State University | Computer Science

```
public class Controller {
  private Line line;
  private Label label;
  private int pointCount;

  public void makeLine() {
    makeLine(null, null);
    pointCount = 0;
  }

  public void makeLine(Point point1, Point point2) {
    line = new Line(point1, point2);
    pointCount = 2;
    model.addItem(line);
  }

  public void setLinePoint(Point point) {
    if (++pointCount == 1) {
      line.setPoint1(point);
    } else if (pointCount == 2) {
      pointCount = 0;
      line.setPoint2(point);
    }
    model.updateView();
  }
  // methods for creating circles and labels

  public void selectItem(Point point) {
    Enumeration enumeration = model.getItems();
    while (enumeration.hasMoreElements()) {
      Item item = (Item) (enumeration.nextElement());
      if (item.includes(point)) {
        model.markSelected(item);
        break;
      }
    }
  }
  // methods for opening and saving a file
}
```

View Class

Idaho State University | Computer Science

```
public class View extends JFrame implements Observer {
  private JPanel drawingPanel;
  private JPanel buttonPanel;
  // JButton references for buttons such as draw line, delete, etc.

  private class DrawingPanel extends JPanel {
    private MouseListener currentMouseListener;
    // keylistener too!
    public void paintComponent(Graphics g) {
      model.setUI(NewSwingUI.getInstance());
      super.paintComponent(g);
      (NewSwingUI.getInstance()).setGraphics(g);
      g.setColor(Color.BLUE);
      Enumeration enumeration = model.getItems();
      while (enumeration.hasMoreElements()) {
        ((Item) enemeration.nextElement()).render();
      }
      g.setColor(Color.RED)
      enumeration = model.getSelectedItems();
      while (enumeration.hasMoreElements()) {
        ((Item) enumeration.nextElement()).render();
      }
    }
    public void addMouseListener(MouseListener newListener) {
      removeMouseListener(currentMouseListener);
      currentMouseListener = newListener;
      super.addMouseListener(newListener);
    }
    // similar for addKeyListener
  }

  public View() {
    // code to create the buttons and panels and put them in the JFram
  }

  public void update(Observable model, Object dummy) {
    drawingPanel.repaint();
  }
}
```

Driver Program

Idaho State University | Computer Science

```
public class DrawingProgram {
  public static void main(String[] args) {
    Model model = new Model();
    View view = new View();
    Controller controller = new Controller();
    Controller.setModel(model);
    View.setController(controller);
    View.setModel(model);
    model.addObserver(view);
    view.show();
  }
}
```

LabelButton

Idaho State University | Computer Science

```
public class LabelButton extends JButton implements ActionListener {
  protected JPanel drawingPanel;
  protected View view;
  private KeyHandler keyHandler;
  private MouseHandler mouseHandler;
  private Controller controller;

  public LabelButton(Controller controller, View jFrame, JPanel jPanel) {
    super("Label");
    this.controller = controller;
    keyHandler = new KeyHandler();
    addActionListener(this);
    view = jFrame;
    drawingPanel = jPanel;
  }

  public void paintComonent(Graphics g) {
    model.setUI(NewSwingUI.getInstance());
    // ...
  }
  public void actionPerformed(ActionEvent event) {
    drawingPanel.addMouseListener(mouseHandler = new MouseHandler());
  }
```

```
private class MouseHandler extends MouseAdapter {
  public void mouseClicked(MouseEvent event) {
    view.setCursor(new Cursor(Cursor.TEXT_CURSOR));
    Controller.instance().makeLabel(event.getPoint());
    drawingPanel.requestFocusInWindow();
    drawingPanel.addKeyListener(keyHandler);
    drawingPanel.addFocusListener(keyHandler);
  }
}

private class KeyHandler extends KeyAdapter implements FocusListener {
  public void keyTyped(KeyEvent event) {
    char character = event.getKeyChar();
    if (character >= 32 && character <= 126) {
      Controller.instance().addCharacter(event.getKeyChar());
    }
  }
```

LabelButton


```
    public void keyPressed(KeyEvent event) {
      if (event.getKeyCode() == KeyEvent.VK_ENTER) {
        view.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
        drawingPanel.removeMouseListener(mouseHandler);
        drawingPanel.removeKeyListener(keyHandler);
        drawingPanel.repaint();
      } else if (event.getKeyCode() == KeyEvent.VK_BACK_SPACE) {
        Controller.instance().removeCharacter();
      }
    }

    public void focusLost(FocusEvent event) {
      view.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
      drawingPanel.removeMouseListener(mouseHandler);
      drawingPanel.removeKeyListener(keyHandler);
      drawingPanel.repaint();
    }
  }
}
```

Design Critique

- A key drawback to the approach covered here is
  - We need to change the controller class everytime new operations are added or if we change the way things are implemented
  - The controller embodies all the implementation in a single class -> making things more complicated
- In the next lecture we will see how we can improve this, as we add in the Undo functionality.

For Next Time

- Review Chapter 11.5 - 11.6
- Review this lecture
- Read Chapter 11.7 - 11.10
- Watch Lecture 32

Are there any questions?