A Case Study of Object Oriented Design and Implementation

- Objectives:
  - To provide a real example of object oriented development.
  - To illustrate the principles of inheritance and polymorphism.
  - To illustrate the practical use of patterns during design.
  - To show the use of standard component libraries
  - To demonstrate an iterative development process and the value of early prototyping.

Case Study--Problem Statement

**Problem**: to design and implement a simple drawing application. The application provides a drawing window, consisting of a drawing "canvas", a menu, a toolbar and a control panel. The toolbar allows the user to select tools to draw various objects on the canvas. Tools are provided for: freehand forms, simple geometric shapes (lines, ovals, rectangles), and text. Ovals and rectangles may be filled or unfilled. A tool is also provided to "erase" areas of the canvas. Tools can be selected from a menu in addition to the toolbar. The control panel allows a user to clear the drawing canvas and select various pen colors. A help menu is also provided. The application is to be developed in Java, using Swing components. It should be runnable either as an application or an applet.
MiniDraw--Iterative Development

- Proposed design iterations:
  - First Iteration: Develop a simple prototype drawing window that provides only a drawing canvas (no toolbar, control panel, or menus) and only supports freehand drawing.
  - Second Iteration: Add the control panel with clear button and pen color selection.
  - Third iteration: Add toolbar, menu, and tools for lines, geometric shapes (unfilled), and erasing.
  - Fourth iteration: Add text tool and filled geometric shapes

MiniDraw--First Iteration

- Key design issues to be addressed
  - Use of Swing components
    - establishing a drawing canvas
    - understanding basics of Java/Swing graphics
  - Basic drawing functionality
    - coupling mouse movement to drawing operations
  - Application vs. Applet issues

Brief Overview of Java Swing

- Swing is a library of components (classes) for building interactive graphical user interfaces. It is part of the Java Foundation Classes (JFC)
- Swing consists of:
  - GUI widgets--frames, buttons, labels, checkboxes, etc
  - Layout Managers--control layout of widgets in windows
  - Events and event listeners-connect GUI events to actions
  - Graphics and imaging classes--allow widgets to draw their visual appearances and display graphics and images.
Swing Overview-Continued

- Component
- Container
- JComponent
- JPanel
- JApplet
- Window
- Frame
- Dialog
- JWindow
- JFrame
- JDialog

Swing Overview: Layout Managers

- Container
  - Layout Manager
    - BorderLayout
    - GridLayout
    - FlowLayout
    - CardLayout
    - GridBagLayout
    - LayoutManager2

Note: can also do custom layouts

java.awt.event Events

- ActionEvent
- AdjustmentEvent
- ComponentEvent
- ItemEvent
- KeyEvent
- MouseEvent
- MouseEvent
- PaintEvent
- WindowEvent

java.awt.event Events

- ActionEvent
- AdjustmentEvent
- ComponentEvent
- ItemEvent
- KeyEvent
- MouseEvent
- MouseEvent
- PaintEvent
- WindowEvent
java.awt.event Listener Interfaces

Java.awt. Graphics Class
- Provides an abstraction for drawing and representing graphical images.
  Graphics g = new Graphics();
  Example methods:
  g.setColor(Color color)
  g.setFont(font)
  g.drawString(String s, int x, int y)
  g.drawLine(int x1, int y1, int x2, int y2)
  g.drawRect(int x, int y, int w, int h)
  g.drawOval(int x, int y, int w, int h)

Displaying Graphics
- Java Components have an associated graphics context (an object of class Graphics)
- The graphics context of a component can be accessed via the getGraphics() method of the component interface
- The visual display of a component can be updated by invoking the repaint() of the component interface.

MiniDraw--First Iteration
MiniDraw--First Iteration Use Case

UseCase Name:  Draw Freehand
Actor: User

Typical Course of Events:

Actor Action: System response:
1. Use case begins when user positions mouse over canvas and depresses mouse button. Begin drawing
3. User moves mouse with button depressed to sketch form on screen. Draw line on canvas that corresponds to mouse movement.

MiniDraw--First Design Iteration

MouseListener and MouseMotionListener Interfaces

MouseListener

<<interface>>

mouseClicked(MouseEvent e)
mouseEntered(MouseEvent e)
mouseExited(MouseEvent e)
mousePressed(MouseEvent e)
mouseReleased(MouseEvent e)

MouseMotionListener

<<interface>>

mouseMoved(mouseEvent e)
mouseDragged(mouseEvent e)

Processing a mousePressed Event

mousePressed(e) DrawingCanvasListener
def: p=
getPoint():Point
1: p=getPoint():Point
def: e=MouseListener
2: startingMousePosition=p
c: MouseEvent

Processing a MouseDragged Event

mouseDragged(e) DrawingCanvasListener
1: p=getPoint():Point
def: e=MouseListener
2: drawLineSegment(startingMousePosition, p)
c: MouseEvent

: DrawingCanvas
Drawing a Line Segment on the Drawing Canvas--First Try

drawLineSegment(p1,p2)

drawingCanvas

1. drawLine(p1,p2)

2. repaint()

g:Graphics

Drawing on the Canvas--Some Complications

• The graphics context of Swing components should be accessed only within overridden update() and/or paint() methods.
• Drawing should be “double-buffered” to prevent screen flickering.
• Solution to both of these problems is to draw on a separate “image” and then copy this image to the component when component is repainted.

Managing the ImageBuffer

• The imageBuffer must be created by invoking the createImage( int, x, int y, int width, int height) method of Component class.
• ImageBuffer must be sized to match the drawing canvas.
• When the canvas is resized, the image buffer must be resized accordingly.
  – This requires instantiating a new image and copying the graphics to it.
  – The setBounds( ) method of a component is called by JRE whenever the component is resized (and also before the component is first displayed).
  – So imageBuffer initialization and resizing can be done by overriding the setBounds( ) method.
ImageBuffer Management--first attempt

Notation:
iBG = imageBufferGraphics
iB = imageBuffer

drawImage(iB)

canvas:DrawingCanvas

iBG:Graphics

setBounds(x:int, y:int, width:int, height:int)

1: newIB=createImage(width, height)

2: iBG=getGraphics():Graphics

3: [iB != null] drawImage(iB)

4: iB = newIB

iBG = imageBufferGraphics
iB = imageBuffer

ImageBuffer Management--continued

- The approach outlined on the previous slide doesn’t quite work right with respect to the following definition of canvas-resizing behavior.
  - The user should be able to draw anywhere within the displayed portion of the canvas, and nowhere outside of the displayed area.
  - If the canvas is enlarged in any dimension, it shall then be possible to draw anywhere within the enlarged visible area.
  - If the canvas is shrunk in any dimension, the area removed from visibility is preserved and will become visible again, if the canvas is subsequently enlarged.

MiniDraw Class Behavior

MiniDraw(...) :

MiniDraw

1: canvas=create()

2: listener=create(canvas)

DrawingCanvas {new}

DrawingCanvasListener {new}

3: addMouseListener(listener)

4: addMouseEventListener(listener)

Additional MiniDraw Behavior (application only)

MiniDraw

1: canvas=create()

2: listener=create(canvas)

DrawingCanvas {new}

DrawingCanvasListener {new}

5: frame=create()

6: {initialize frame}

JFrame {new}
MiniDraw Prototype--Methods and Attributes

<table>
<thead>
<tr>
<th>DrawingCanvas</th>
<th>DrawingCanvasListener</th>
</tr>
</thead>
<tbody>
<tr>
<td>imageBuffer: Image</td>
<td>canvas: DrawingCanvas</td>
</tr>
<tr>
<td>imageBufferGraphics: Graphics</td>
<td>startingMousePosition: Point</td>
</tr>
<tr>
<td>DrawingCanvas()</td>
<td>DrawingCanvasListener()</td>
</tr>
<tr>
<td>drawLineSegment(p1:Point, p2:Point)</td>
<td>canvas: DrawingCanvas mousePressed(e: mouseEvent)</td>
</tr>
<tr>
<td>update(g: Graphics)</td>
<td>update(g: Graphics)</td>
</tr>
<tr>
<td>paint(g: graphics)</td>
<td>paint(g: graphics)</td>
</tr>
<tr>
<td>setBounds(x: int, y: int, width: int, height: int)</td>
<td>setBounds(x: int, y: int, width: int, height: int)</td>
</tr>
</tbody>
</table>

MiniDraw Prototype--Methods and Attributes--Continued

<table>
<thead>
<tr>
<th>MiniDraw</th>
</tr>
</thead>
<tbody>
<tr>
<td>canvas: DrawingCanvas</td>
</tr>
<tr>
<td>listener: EventListener</td>
</tr>
<tr>
<td>main(ARGS: String) {static}</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
</tbody>
</table>

MiniDraw--Application vs. Applet

- **Application:**
  - Invoked via static main() method in class MiniDraw.
  - Application must instantiate a frame/window and attach itself to it, if needed
- **Applet:**
  - No main method
  - Applet is instantiated by browser/viewer
  - Three methods automatically invoked: init(), start(), and paint().
  - Applet is automatically attached to the browser/viewer window.

An Idiom (implementation pattern) for a dual Applet/Application (Swing)

```java
public class DualAppletApp extends JApplet {
    protected boolean isApplet;
    public DualAppletApp(boolean isApplet) {
        this.isApplet = isApplet;
        //...general constructor stuff goes here
        if (!isApplet) {
            init();
            start();
        }
    }
    public DualAppletApp() {
        // invoked as an applet
        this(true);
    }
}
```
public static void main(String[] args) {
    // invoked as application
    JFrame frame = new JFrame;
    frame.setTitle(title);
    frame.getContentPane().setLayout(new BorderLayout( ));
    frame.getContentPane().add(new DualAppletApp(false), BorderLayout.CENTER);
    frame.addWindowListener(new FrameListener( ));
    frame.pack( );
    frame.setSize(width, height);
    frame.show( );
}

MiniDraw--Second Design Iteration

- Objective: Add a control panel to the bottom of the canvas
- Control panel has:
  - a “clear” button to clear the drawing canvas.
  - A menu to select drawing pen colors: black, blue, green, red.

MiniDraw Second Design Iteration--Java/Swing Issues

- Control Panel will be a JPanel.
- Clear Button will be a JButton.
  - Clicking on a JButton generates an ActionEvent.
  - Listener must implement the ActionListener interface.
- Pen Color Selection Menu will be a JComboBox.
  - Selecting an item from the drop down menu will generate an ItemEvent with state ItemEvent.SELECTED.
  - Listener must implement the EventListener interface.

ActionListener and ItemEventListener Interfaces

- ActionListener
  - actionPerformed(e:ActionEvent)
- ItemListener
  - itemStateChanged(e:ItemEvent)

Possible return values:
- ItemEvent.DESELECTED
- ItemEvent.SELECTED

Returns Item affected by event
MiniDraw--Second Iteration

JComponent
JApplet

DrawingCanvas

MiniDraw

EventListener

MouseListener

MouseMotionListener

DrawingCanvasListener

MouseListener

MouseMotionListener

ActionListener

ItemListener

ControlPanel

New classes
Modified classes

ControlPanel--Additional Details

JPanel

ControlPanel

JButton

JComboBox

Collaboration Diagram--Handling an ActionEvent from Clear Button

actionPerformed(e) : ControlPanelListener

"original" pen color
penColor:Color

1: clearCanvas()
canvasWidth:int
canvasHeight:int

1.1: setColor(Color.white)
1.2: fillRect(0,0,canvasWidth, canvasHeight)
1.3 setColor(penColor)

1.4 repaint()

imageBufferGraphics:Graphics

Collaboration Diagram--Handling an ItemEvent

itemStateChanged(e) : ControlPanelListener

1: type=getStateChange():int
2: [type=SELECTED] item=getItem():Object
3: [type=SELECTED] color=itemToColor():Color

4: [type=SELECTED] setPenColor(color)

e: ItemEvent

:DrawingCanvas
ItemEvent Handling Scenario--Continued

```
4: setPenColor(color)
penColor: Color
4.1: penColor = color
4.2: setColor(penColor)
imageBufferGraphics: Graphics
```

ControlPanel--Registering a Listener

```
addControlPanelListener(listener)
1: addActionListener(listener)
2: addItemListener(listener)
```

Second Design Iteration--A Practical Issue

- For this iteration we must modify the `DrawingCanvas` class.
- Probably want to give the modified class a new name--e.g. `DrawingCanvas2`--to differentiate it from the previous iteration.
- But, then we will have to find and change all references to the class `DrawingCanvas`.
- For instance, we will have to modify the class `DrawingCanvasListener`, even though nothing has changed in it other than the name of the `DrawingCanvas` class.

Second Iteration--A “Better” Approach

```
DrawingCanvas
DrawingCanvas2
MiniDraw
MiniDraw2
```

```
MouseListener
MouseMotionListener
```

```
Component
JApplet
```

```
EventListener
ControlPanel
```

```
New classes
Modified classes
```

```
EventListener
ControlPanelListener
```

```
ComboBox: JComboBox
```
Second Iteration--A “Better” Approach (Continued)

- One More Detail: Need to modify original MiniDraw class so that associated components--DrawingCanvas, DrawingCanvasListener--are instantiated by a factory method.
- This will permit future design iterations to instantiate the proper types of components.
- E.g. MiniDraw2 can override the createDrawingCanvas() factory method of MiniDraw to instantiate a DrawingCanvas2 class instead of a DrawingCanvas class.

Factory Methods in MiniDraw Class

<table>
<thead>
<tr>
<th>MiniDraw</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>createDrawingCanvas():DrawingCanvas</td>
<td></td>
</tr>
<tr>
<td>createDrawingCanvasListener(canvas:DrawingCanvas):EventListener</td>
<td></td>
</tr>
</tbody>
</table>

Overrides superclass' factory method to instantiate a DrawingCanvas2

<table>
<thead>
<tr>
<th>MiniDraw2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>createDrawingCanvas():DrawingCanvas</td>
<td></td>
</tr>
<tr>
<td>createControlPanel():ControlPanel</td>
<td></td>
</tr>
<tr>
<td>createControlPanelListener(DrawingCanvas2:canvas):EventListener</td>
<td></td>
</tr>
</tbody>
</table>

Second Iteration--New Classes

- DrawingCanvas

Second Iteration--New Classes (continued)

- DrawingCanvas2
  - canvasWidth: int
  - canvasHeight: int
  - penColor: Color

- ControlPanel
  - clearButton: JButton
  - comboBox: JComboBox
  
Create clearButton, create and initialize comboBox

See earlier collaboration diagram
Second Iteration--New Classes
(continued)

<table>
<thead>
<tr>
<th>ControlPanelListener</th>
</tr>
</thead>
<tbody>
<tr>
<td>canvas:DrawingCanvas2</td>
</tr>
</tbody>
</table>
| ControlPanelListener(canvas:DrawingCanvas2) | actionPerformed(e:ActionEvent) 
| itemStateChanged(e:ItemEvent) | -itemToColor(item:Item) |

- See earlier collaboration diagrams

MiniDraw--Third Design Iteration

- Objective: To add toolbar, menu, and tools for unfilled geometric shapes and erasing.
- Swing Issues:
  - Will use Box class for ToolBar.
  - Will use JButtons for ToolBar buttons.
  - Will use JMenuBar and JMenu for menus

Design Class Model--First Attempt
(Control Panel Components not shown)
Problems with this Design Concept

- Which components need explicit knowledge of tools?
  - ToolBar
  - MenuBar
  - ToolBarListener
  - ToolBoxListener
  - DrawingCanvasListener
  - MiniDraw3

- High coupling
- To add a new tool, all of these classes must be modified

Problems with Design Concept--Continued

- ToolBarListener and MenuBarListener perform nearly the identical function.
- A better design concept would:
  - allow the configuration of tools to be specified in a single place.
  - Combine ToolBar and MenuBar Listener functionality.

Moving Toward an Improved Design

- The Swing Action interface and AbstractAction class provide a neat mechanism for combining menu and toolbar event handling.
- An AbstractAction Object implements the ActionListener interface and also provides:
  - a name that can be used as a text label of a component
  - a set of values that can store various attributes such an icon to be painted on the component, a “tool tip” string, and a description of the action.

Swing Action Objects--Continued

- When a Swing Action Object is added to a Swing container, the container:
  - creates a component that is appropriate for the container
  - retrieves appropriate attributes from the Action object to customize the component.
  - Sets the initial enabled or disabled state of the action object
  - renders the component
Still More about the Action Interface

- Action interface provides the following methods:
  - putValue(key, value)  //set an attribute
  - getValue(key)  //retrievec an attribute
  - setEnabled(b)  //Set enabled state
  - isEnabled()  //test enabled state

- Key values:
  - NAME  //e.g. text label
  - DEFAULT  //e.g. icon
  - SHORT_DESCRIPTION  //e.g. tool tip
  - LONG_DESCRIPTION
  - SMALL_ICON

Using Action Objects as Tool Listeners

ToolBar
Box

JMenu
JMenuBar

Action
<<interface>>

AbstractAction

ToolListener

Note: Action objects can be directly added to a JMenu as menu items.

Behavior of a ToolListener

actionPerformed(e)

tool:Tool

1:setCurrentTool(tool)

currentTool:Tool

????????

1.1 currentTool = tool

Design Question: What class should be responsible for keeping track of the currently selected tool?

Behavior of the DrawingCanvasListener

DrawingCanvasListener
canvas: DrawingCanvas
startingMousePosition:Point

DrawingCanvasListener(
canvas:DrawingCanvas)
mousePressed(c.mouseEvent)
mouseDragged(c.mouseEvent)
mouseReleased(c.mouseEvent)

Behavior depends on currently selected tool.

Some tools may want to use additional event notifications.
Behavior of the DrawingCanvasListener--continued

DrawingCanvasListener needs to “delegate” behavior to the appropriate tool.

Solution: Use a state pattern:

```
Context
Request( )

State
Handle( )

ConcreteStateA
Handle( )

ConcreteStateB
Handle( )
```

Use of the State Pattern by DrawingCanvasListener

```
DrawingCanvasListener

get CurrentTool( ): Tool
mousePressed(e)
mouseDragged(e)

...

currentState->Handle( )
```

Designing the Tools

- Tools must implement (delegated) MouseListener and MouseMotionListener methods and draw appropriate graphics on the ImageBuffer.
- Note that in our first design iteration, we placed the method drawLineSegment(...) in the class DrawingCanvas. In retrospect, this was not a good choice. Why?

Rethinking the Allocation of Responsibility for Drawing Operations

- Move responsibility for drawing on the ImageBuffer to the appropriate tool.
- Add a public method `getImageBufferGraphics()` to DrawingCanvas to allow tools to access the graphics context of the ImageBuffer.
Use Case--Drawing a Line

**UseCase Name:** Draw Line  
**Actor:** User  
**Typical Course of Events:***

<table>
<thead>
<tr>
<th>Actor Action</th>
<th>System response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Use case begins when user selects “line” from the toolbar or menu.</td>
<td>Sets the Line tool as the current tool.</td>
</tr>
<tr>
<td>2. User positions mouse over canvas and depresses mouse button.</td>
<td>Establish starting point of line and color for setXORmode.</td>
</tr>
<tr>
<td>3. User moves mouse with button depressed.</td>
<td>Display &quot;temporary&quot; line on canvas between starting point and current mouse position. Latest displayed line is displayed “permanently”.</td>
</tr>
<tr>
<td>4. User releases mouse.</td>
<td></td>
</tr>
</tbody>
</table>

The Line Tool--Design Issues

- How do we display a “temporary” line?
- Solution: Graphics context provides an “exclusive or” (XOR) mode.
  - Color of a drawn figure depends on both current pen color and color of underlying pixels.
  - Drawing a figure once will make it visible.
  - Drawing the same figure (in the same position) makes it disappear.
  - XOR mode is selected by the setXORMode(Color) method of Graphics class.

Drawing a “Temporary Line to Track Mouse Movement

**State of the display between MouseEvent occurrences**

- Established by MousePressed event  
  - startPos  
- currentPos  
- Position at most recent MouseEvent  
- Line drawn in color specified for setXORmode

Drawing a “Temporary Line to Track Mouse Movement(continued)

**Actions to update displayed line at a MouseDragged Event**

- Established by MousePressed event  
  - startPos  
- currentPos  
- Mouse position for this MouseDragged event.  
  - 1: Erase this line by redrawing it.  
  - 2: Draw new line in XORMode color
**LineTool: Collaboration Diagrams**

- `mousePressed(e)`
  - :LineTool
  - `startPos`: Point
  - `currentPos`: Point
  - `saveColor`: Color
  - `iBG`: Graphics
  - `DrawingCanvas`

**LineTool MouseDragged Method**

- `mouseDragged(e)`
  - :LineTool
  - `startPos`: Point
  - `currentPos`: Point
  - `saveColor`: Color
  - `iBG`: Graphics

**LineTool: MouseReleased Method**

- `mouseReleased(e)`
  - :LineTool
  - `startPos`: Point
  - `currentPos`: Point
  - `saveColor`: Color
  - `iBG`: Graphics

**Use Case -- Drawing a Rectangle**

**UseCase Name:** Draw Unfilled Rectangle

**Actor:** User

**Typical Course of Events:**

**Actor Action:**
1. Use case begins when user selects “rectangle” from the toolbar or menu.
2. User positions mouse over canvas and depresses mouse button.
3. User moves mouse with button depressed.
4. User releases mouse.

**System response:**
- Sets the rectangle tool as the current tool.
- Establish starting corner of rectangle.
- Display “temporary” rectangle on canvas with starting corner and current mouse position as opposite diagonal corners.
- Latest displayed rectangle is displayed “permanently”.

**Remarks:**
- Latest displayed rectangle is displayed “permanently”.

Use Cases: Drawing an Oval

• Similar to Draw Rectangle use case.

Use Case--Erasing

UseCase Name: Erase
Actor: User

Typical Course of Events:

Actor Action: System response:
1. Use case begins when user selects “erase” from the toolbar or menu. Sets the erase tool as the current tool.
2. User positions mouse over canvas and depresses mouse button. Establish starting position for erasure.
3. User moves mouse with button depressed. A 5x5 pixel square centered at current mouse position is painted white.

ToolBar Class: Collaboration
Diagram for Constructor

MiniDraw--Fourth Design Iteration

• Objective: Add additional tools for drawing filled ovals and rectangles, and a tool for typing text.

• Note that in the third design iteration the tool classes for all of the geometric objects were very similar. All of them involve:
  – establishing an initial “end” of the object when mouse button is pressed.
  – drawing a “temporary” figure when the mouse is dragged.
  – drawing a “permanent” figure when the mouse button is released, with the cursor position at mouse release giving the second “end point” for the figure.

• The tools differ only in the specific figure drawn.
Using Subclasses to Coalesce Common Behavior of Geometric Tools

TwoEndShapeTool

LineTool | OvalTool | RectangleTool | FilledOvalTool | FilledRectangleTool

Problem: How can we move the common behavior to the superclass TwoEndShapeTool while delegating responsibility for the specific drawing operations to the appropriate subclass?
Solution: Use the GoF Strategy Pattern.

The Strategy Pattern

Context

AlgorithmInterface( )

ConcreteStrategyA

ConcreteStrategyB

ConcreteStrategyC

Tool

TwoEndShapeTool

TwoEndShapeTool

(canvas:DrawingCanvas, shape:TwoEndShape)

MousePressed( )

MouseDragged( )

MouseReleased( )

... Strategy

LineShape

draw( ... )

drawOutline( ... )

OvalShape

draw( ... )

drawOutline( ... )

RectangleShape

draw( ... )

drawOutline( ... )

Use of The Strategy Pattern In MiniDraw

Adding A Text Tool to MiniDraw

- The interface for keyboard events is KeyListener

KeyListener

keyPressed(e:KeyEvent)

keyReleased(e:KeyEvent)

keyTyped(e:KeyEvent)

KeyEvent

getKeyChar( ):char

...
Keyboard Input--The Focus Issue

- Only one component can receive keyboard input at any given time.
- To receive keyboard input, a component must have the keyboard focus.
- Focus is determined by a “focus manager”.
- A component can gain keyboard focus via a requestFocus() method.
- In MiniDraw, clicking the mouse on the drawing canvas (after selecting the Text Tool) should give focus to the canvas and establish the starting point for the displayed text.

MiniDraw Fourth Iteration--Conceptual Model Extensions

- Use a separate KeyboardListener to handle KeyEvents
  - can be implemented to delegate to KeyboardTool class only.
  - This way DrawingCanvasListener doesn’t need to check the currentTool type to avoid delegating to a tool that doesn’t implement KeyListener interface.
- Have KeyboardTool establish focus in its MousePressed or MouseClicked method.
  - This way, focus is established only when currentTool is a KeyboardTool.
  - KeyboardTool has knowledge of DrawingCanvas anyway, so no new couplings are introduced.
MiniDraw Fourth Iteration--New Classes

Tool

TwoEndShapeTool

TwoEndShapeTool(canvas: DrawingCanvas, shape: TwoEndShape)
mousePressed(e: MouseEvent)
mousedownDragged(e: MouseEvent)
mouseReleased(e: MouseEvent)

MiniDraw Fourth Iteration Classes--Continued

TwoEndShape

Draw(g: Graphics, x0: int, y0: int, x1: int, y1: int)
drawOutline(g: Graphics, x0: int, y0: int, x1: int, y1: int)

LineShape

OvalShape

FilledRectangleShape

FilledOvalShape

KeyListener

KeyboardListener

KeyListener(canvas: DrawingCanvas)
keyPressed(e: KeyEvent)
keyReleased(e: KeyEvent)
keyTyped(e: KeyEvent)

Collaboration Diagram for KeyPressed method

1: tool := getCurrentTool()
2:[tool != null]
keyPressed(e)

canvas: DrawingCanvas
tool: KeyboardTool
MiniDraw Fourth Iteration Classes--Continued

Tool

KeyboardTool

keyPressed(keyEvent e)
keyReleased(keyEvent e)
keyTyped(keyEvent e)

TextTool

TextTool(canvas: DrawingCanvas)
mousePressed(e: MouseEvent)
keyPressed(e: KeyEvent)
[remaining methods of interface are null]

Collaboration Diagram for mousePressed method of TextTool

mousePressed(e)

1: requestFocus()
3: iGraphics = getImageBufferGraphics()

canvas: DrawingCanvas
startingPos: Point

text: StringBuffer

2: startingPos = getPoint()

4: setFont(font)

iGraphics: Graphics
text: StringBuffer [new]

Collaboration Diagram for keyPressed method of TextTool

keyPressed(e)

1: nextChar = getKeyChar()

canvas: DrawingCanvas
startingPos: Point
text: StringBuffer

3: iGraphics = getImageBufferGraphics()

2: append(nextChar)

iGraphics: Graphics
text: StringBuffer
Reaping the Benefits of Extensible Design: Adding a “Triangle Tool”

• Can draw triangle as “two end” figure:

![Diagram of triangle drawing as two end figure]

Adding a Triangle Tool--Continued

• All we need to do is create a TriangleShape class that can draw a triangle as shown on the previous page.
• Then simply add a new TwoEndShapeTool, with a TraingleShape parameter, to the ToolList.
• And we’re done.
• Total time it took me to implement it: <10 minutes.

The TriangleShape Class

```java
import java.awt.*;
public class TriangleShape implements TwoEndShape {
    public void drawOutline(Graphics g, int x0, int y0, int x1, int y1) {
        g.drawLine(x0, y0, x1, y1);
        int midX = x0 + (x1-x0)/2;
        g.drawLine(x0, y0, midX, y1);
        g.drawLine(x1, y0, midX, y1);
    }
    public void draw(Graphics g, int x0, int y0, int x1, int y1) {
        drawOutline(g, x0, y0, x1, y1);
    }
}
```

Adding the Triangle to the ToolList

```java
protected ToolList createToolList() {
    ToolList actions = new ToolList();
    actions.add(new ToolListener("Triangle", getImageIcon("triangle.jpg"),
                                "Triangle drawing tool", (DrawingCanvas3)canvas,
                                new TwoEndShapeTool ((DrawingCanvas3)canvas,
                                new TriangleShape())));
}
```