Principles of Software Construction: Objects, Design, and Concurrency
(Part 3: Design Case Studies)

Introduction to GUIs

Jonathan Aldrich  Charlie Garrod
Administrivia

• Homework 4a due tonight
• Homework 4b due Thursday, October 22
Key concepts from Tuesday
Key concepts from Tuesday

• Formal verification
• Testing
  – Coverage metrics
  – Black-box vs. white-box testing
• Static analysis
Key concepts from recitation yesterday
Key concepts from yesterday's recitation

• Discovering design patterns
• The Observer pattern
The Observer design pattern

```
<<stereotype>>
Subject
{abstract}
+Attach(observer: Observer) : void
+Detach(observer: Observer) : void
+Notify()

observers

<<stereotype>>
Observer
{interface}
+Update() {abstract}

Notify() {
    for all o on observers {
        o.Update();
    }
}

<<stereotype>>
ConcreteSubject

+subjectState: State
+GetState() : State
+SetState(state: State) : void

<<stereotype>>
ConcreteObserver

+Update()

1
subject

1
subject
```
The Observer design pattern

• Applicability
  – When an abstraction has two interdependent aspects and you want to reuse both
  – When state change to one object requires notifying others, without becoming dependent on them

• Consequences
  – Loose coupling between subject and observer, enhancing reuse
  – Support for broadcast communication
  – Notification can lead to further updates, causing a cascade effect
Learning goals for today

- Understand the design challenges and common solutions for Graphical User Interfaces (GUIs)
- Understand event-based programming
- Understand and recognize the design patterns used and how those design patterns achieve design goals.
  - Observer pattern
  - Strategy pattern
  - Template method pattern
  - Composite pattern
  - Model-view-controller
  - Other common GUI design patterns not discussed here: Decorator, Façade, Adapter, Command, ...
- Diagnose problems caused by computation in the GUI threads
Aside: Anonymous inner classes in Java

- You can implement an interface without naming the implementing class
  
  - E.g.,
    ```java
    public interface Runnable {
        public void run();
    }
    
    public static void main(String[] args) {
        Runnable greeter = new Runnable() {
            public void run() {
                System.out.println("Hi mom!");
            }
        };
        greeter.run();
    }
    ```
Scope within an anonymous inner class

- An anonymous inner class can access final (or effectively final) variables in the scope where it is defined

```java
public interface Runnable {
    public void run();
}

public static void main(String[] args) {
    final String name = "Charlie";  // final variable
    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi " + name);
        }
    };
    greeter.run();
}
```

OK
Scope within an anonymous inner class

- An anonymous inner class can access final (or effectively final) variables in the scope where it is defined

```java
public interface Runnable {
    public void run();
}

public static void main(String[] args) {
    String name = "Charlie"; // not final, but could be final
    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi " + name);
        }
    };

    greeter.run();
}
```

OK
Scope within an anonymous inner class

• An anonymous inner class can access final (or effectively final) variables in the scope where it is defined

```java
public interface Runnable {
    public void run();
}

public static void main(String[] args) {
    String name = "Charlie"; // not final; assigned later
    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi " + name);
        }
    };
    name = "Jonathan";
    greeter.run();
}
```

Compile-time error
Today: Introduction to Graphical User Interfaces (GUIs)

- Event-based programming
- Building a GUI using Java Swing
- Event-handling and decoupling with the Observer pattern
- GUI design challenges
  - Design patterns that solve them
- Practical advice: Threading architecture of a typical GUI
Event-based programming

- Style of programming where control-flow is driven by (usually external) events

```java
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(42)
}

public void performAction(ActionEvent e) {
    bigBloatedPowerPointFunction(e);
    withANameSoLongIMadeItTwoMethods(e);
    yesIKnowJavaDoesntWorkLikeThat(e);
}

public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(40)
}
```
Examples of events in GUIs
Examples of events in GUIs

• User clicks a button, presses a key
• User selects an item from a list, an item from a menu, expands a tree
• Mouse hovers over a widget, focus changes
• Scrolling, mouse wheel turned
• Resizing a window, hiding a window
• Drag and drop

• A package arrives from a web service, connection drops, ...
• System shutdown, ...
Interaction with command-line interfaces

Scanner input = new Scanner(System.in);
while (questions.hasNext()) {
    Question q = question.next();
    System.out.println(q.toString());
    String answer = input.nextLine();
    q.respond(answer);
}
while (true) {
    if (isKeyDown("Alt+Q"))
        break;
    if (isKeyDown("F1"))
        openHelp();
    if (isMouseDown(10 ...)
        startMovingWindow();
    ...
}
//static public void main...
JFrame window = ...
window.setDefaultCloseOperation(
    WindowConstants.EXIT_ON_CLOSE);
window.setVisible(true);

//on add-button click:
String email =
    emailField.getText();
emaillist.add(email);
Event-based GUIs

//static public void main...
JFrame window = ...
window.setDefaultCloseOperation(
    WindowConstants.EXIT_ON_CLOSE);
window.setVisible(true);

//on add-button click:
String email = emailField.getText();
emaillist.add(email);

//on remove-button click:
int pos = emaillist.getSelectedItem();
if (pos>=0) emaillist.delete(pos);
(Blocking) Interactions with users

- Game
- Dealer
- Player

- newGame
- addCards
- addCards
- getAction
- action
- [action==hit] addCard

blocking execution
Interactions with users through events

• Do not block waiting for user response
• Instead, react to user events

e.g.:
Programming an event-based GUI

• Typically use a GUI framework
  – Register code (a.k.a. callbacks) to handle different events
  – Operating system / GUI framework detects events
    • Determines which components are registered to handle the event and calls the event handlers
    • Your code is idle until called to handle events
  – Program exits by calling some exit method
Programming an event-based GUI

• Setup phase
  – Describe how the GUI window should look
  – Use libraries for windows, widgets, and layout
  – Embed specialized code for later use
  – Register callbacks

• Execution
  – Framework gets events from OS
    • Raw events: mouse clicks, key presses, window becomes visible, etc.
  – Framework processes events
    • Click at 10,40: which widget?
    • Resize window: what to re-layout and redraw?
  – Triggers callback functions of corresponding widgets (if registered)
Example: The AlarmWindow

- \ldots\texttt{edu.cmu.cs.cs214.rec06.alarmclock.AlarmWindow}
  - Creates a JFrame with a JPanel to go in it
  - Creates a text label and a button
  - Makes the window (and its contents) visible when the alarm goes off

- When the dismiss button is clicked, its event handler hides the window
Example: The CustomerManagementUI

- ...rec06.customerlist.gui.CustomerManagementUI
  - Creates a JFrame with a JPanel to go in it
  - Makes the window (and its contents) visible

- ...rec06.customerlist.gui.CustomerManagementPanel
  - Creates numerous labels and text fields, a customerAddButton
  - Registers an event handler for the customerAddButton

- When the customerAddButton is clicked, its event handler gets the text from the text fields and adds a customer to the list
GUI frameworks in Java

- **AWT**
  - Native widgets, only basic components, dated
- **Swing**
  - Java rendering, rich components
- **SWT + JFace**
  - Mixture of native widgets and Java rendering; created for Eclipse for faster performance
- **Others**
  - Apache Pivot, SwingX, JavaFX, ...

- Different in their specific designs, but similar overall strategies and concepts
Swing

JFrame

JPanel

JButton

JTextField

...
Swing has many widgets

- JLabel
- JButton
- JCheckBox
- JChoice
- JRadioButton
- JTextField
- JTextArea
- JList
- JScrollBar
- … and more

- JFrame is the Swing Window
- JPanel (aka a pane) is the container to which you add your components (or other containers)
To create a simple Swing application

• Make a Window (a JFrame)
• Make a container (a JPanel)
  – Put it in the window
• Add components (Buttons, Boxes, etc.) to the container
  – Use layouts to control positioning
  – Set up observers (a.k.a. listeners) to respond to events
  – Optionally, write custom widgets with application-specific display logic
• Set up the window to display the container
• Then wait for events to arrive...
Reacting to events
Creating a button

```java
//static public void main...
JFrame window = ...

JPanel panel = new JPanel();
window.setContentPane(panel);

JButton button = new JButton("Click me");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked");
    }
});
panel.add(button);

window.setVisible(true);
```
Creating a button

```java
//static public void main...
JFrame window = ...

JPanel panel = new JPanel();
window.setContentPane(panel);

JButton button = new JButton("Click me");
button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("Button clicked");
    }
});
panel.add(button);
window.setVisible(true);
```
ActionListeners

- Listeners are objects with callback functions
- Listeners can be registered to handle events on widgets
- All registered widgets are called if event occurs

```java
interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

```java
class ActionEvent {
    int when;
    String actionCommand;
    int modifiers;
    Object source();
    int id;
    ...
}
```
ActionListeners

- Listeners are objects with callback functions
- Listeners can be registered to handle events on widgets
- All registered widgets are called if event occurs

```java
interface ActionListener {
    void actionPerformed(ActionEvent e);
}

class ActionEvent {
    int when;
    String actionCommand;
    int modifiers;
    Object source();
    int id;
    ...
}

class AbstractButton extends JComponent {
    private List<ActionListener> listeners;
    public void addActionListener(ActionListener l) {
        listeners.add(l);
    }
    protected void fireActionPerformed(ActionEvent e) {
        for (ActionListener l: listeners)
            l.actionPerformed(e);
    }
}
```
ActionListeners

- Listeners are objects with callback functions
- Listeners can be registered to handle events on widgets
- All registered widgets are called if event occurs

```java
interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

```java
class ActionEvent {
    int when;
    String actionCommand;
    int modifiers;
    Object source();
    int id;
    ...
}
```

```java
class AbstractButton extends JComponent {
    private List<ActionListener> listeners;
    public void addActionListener(ActionListener l) {
        listeners.add(l);
    }
    protected void fireActionPerformed(ActionEvent e) {
        for (ActionListener l: listeners)
            l.actionPerformed(e);
    }
}
```
Recall the observer design pattern
Design discussion

• Button implementation should be reusable but customizable
  – Different button label, different event-handling
• Must decouple button's action from the button itself
• Listeners are separate independent objects
  – A single button can have multiple listeners
  – Multiple buttons can share the same listener
Swing has many event listener interfaces:

- ActionListener
- AdjustmentListener
- FocusListener
- ItemListener
- KeyListener
- MouseListener
- TreeExpansionListener
- TextListener
- WindowListener
- …and on and on…

```java
interface ActionListener {
    void actionPerformed(ActionEvent e);
}

class ActionEvent {
    int when;
    String actionCommand;
    int modifiers;
    Object source();
}
Decoupling from a GUI
A GUI design challenge

• Consider the transit simulator, implemented by a `World` class:
  – Simulator `World` stores all entities
  – GUI shows entities, triggers new events in simulator
  – When should the GUI update the screen?
A GUI design challenge, part 2

• What if we add a warning to alert if a bus has moved?
A GUI design challenge, part 3

- What if the simulator world changes for reasons not caused by the GUI?
A GUI design challenge, part 3: one possible design

- Let the World tell the GUI that something happened
A GUI design challenge, part 3: one possible design

• Let the World tell the GUI that something happened

Problem: This couples the World to the GUI implementation.
Core implementation vs. GUI

• Core implementation: Application logic
  – Computing some result, updating data

• GUI
  – Graphical representation of data
  – Source of user interactions

• Design guideline: *Avoid coupling the GUI with core application*
  – Multiple UIs with single core implementation
  – Test core without UI
  – *Design for change, design for reuse, design for division of labor; low coupling, high cohesion*
A GUI design challenge, part 3: one possible design

• Let the World tell the GUI that something happened

Problem: This couples the World to the GUI implementation.
Decoupling with the Observer pattern

- Let the world tell *all* interested components about updates
An architectural pattern: Model-View-Controller (MVC)

- **Model**: Manage data related to the application domain.
- **View**: Manage display of information on the screen.
- **Controller**: Manage inputs from user: mouse, keyboard, menu, etc.
Model-View-Controller (MVC)

Passive model

Active model

Separating application core and GUI, a summary

- Reduce coupling: do not allow core to depend on UI
- Create and test the core without a GUI
  - Use the Observer pattern to communicate information from the core (Model) to the GUI (View)
More GUI design challenges
Swing layouts

The simplest, and default, layout. Wraps around when out of space.

Like FlowLayout, but no wrapping.

More sophisticated layout managers

see http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html
A naïve hard-coded implementation

```java
class JPanel {
    protected void doLayout() {
        switch(getLayoutType()) {
            case BOX_LAYOUT: adjustSizeBox(); break;
            case BORDER_LAYOUT: adjustSizeBorder(); break;
            ...
        }
    }
    private adjustSizeBox() { ... }
}
```

- A new layout would require changing or overriding JPanel
A better solution: delegate the layout responsibilities

• Layout classes, e.g.:
  ```java
  contentPane.setLayout(new FlowLayout());
  contentPane.setLayout(new GridLayout(4,2));
  ```

• Similarly, there are border classes to draw the borders, e.g.:
  ```java
  contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
  ```

What design pattern is this?
Recall the strategy pattern
Another GUI design challenge: nesting containers

- A JFrame contains a JPanel, which contains a JPanel (and/or other widgets), which contains a JPanel (and/or other widgets), which contains…

What design pattern best models this?
Recall the composite design pattern
Yet another GUI design challenge: partial customization

public void paint(Graphics g)

Invoked by Swing to draw components. Applications should not invoke paint directly, but should instead use the repaint method to schedule the component for redrawing.

This method actually delegates the work of painting to three protected methods: paintComponent, paintBorder, and paintChildren. They're called in the order listed to ensure that children appear on top of component itself. Generally speaking, the component and its children should not paint in the insets area allocated to the border. Subclasses can just override this method, as always. A subclass that just wants to specialize the UI (look and feel) delegate's paint method should just override paintComponent.

Overrides:

- paint in class Container

Parameters:

- g - the Graphics context in which to paint

See Also:

- paintComponent(java.awt.Graphics),
- getComponentGraphics(java.awt.Graphics), repaint(long, int, int, int)
Recall the template method pattern

- **Applicability**
  - When an algorithm consists of varying and invariant parts that must be customized
  - When common behavior in subclasses should be factored and localized to avoid code duplication
  - To control subclass extensions to specific operations

- **Consequences**
  - Code reuse
  - Inverted “Hollywood” control: don’t call us, we’ll call you
  - Ensures the invariant parts of the algorithm are not changed by subclasses
The Swing threading architecture

**main() thread**
- Create window
- Set up callbacks
- Show window (thread ends)

**GUI thread**
- Loop forever:
  - Get system event
  - Invoke callback

  e.g. callback code:
  - Run complex numeric algorithm
  - (Problem: UI is unresponsive)
  - Show result
The Swing threading architecture: worker threads

**main() thread**
- Create window
- Set up callbacks
- Show window (thread ends)

**GUI thread**
- Loop forever:
  - Get system event
  - Invoke callback

**Worker thread**
- Worker thread execution:
  - invoke doInBackground()
  - run complex numeric algorithm
  - store result in SwingWorker
  - signal to UI that we are done

Callback code:
- create SwingWorker
- start it executing

<table>
<thead>
<tr>
<th>SwingWorker</th>
</tr>
</thead>
<tbody>
<tr>
<td>result : Long</td>
</tr>
</tbody>
</table>
Summary

• GUIs are full of design patterns
  – Strategy pattern
  – Template Method pattern
  – Composite pattern
  – Observer pattern
  – Decorator pattern
  – Façade pattern
  – Adapter pattern
  – Command pattern
  – Model-View-Controller

• Swing for Java GUIs

• Separation of GUI and Core