Object Oriented Programming and Design in Java

Session 7
Instructor: Bert Huang
Announcements

• Homework 1 due now
• Homework 2 posted on website, due Mar. 3
• For fastest email queries, email all TAs and me
  ● {bert@cs., jwg2116@, lep2128@, yh2315@}@columbia.edu
Review

- Named ActionListeners
- Timers
- Interfaces and polymorphism
- Examples: List, Comparator, Collection, Iterator
Today's Plan

• Introduction to **programming patterns**
• Patterns in GUI programming
  • Model/View/Controller, Observer, Composite, Decorator, Strategy
Programming Patterns

• Common design challenges have been solved over and over by others

• Many solutions are recorded as patterns, useable in your own design

• Higher level form of abstraction than more explicit, code-specific ideas (e.g., encapsulation)
Pattern Format

• Patterns are defined by a general **context**, the design challenge

• And a **solution**, which prescribes how to design your program in the context

• Since patterns are general, they will feature many interfaces
Iterator: Context

- An aggregate object contains element objects
- Clients need access to the elements
- The aggregate should not expose its internal structure
- There may be multiple clients that need simultaneous access
Iterator: Solution

- Define an iterator class that fetches on element at a time
- Each iterator object keeps track of the position of the next element to fetch
- If there are variations of the aggregate and iterator class, implement common interface types.
Patterns in GUI Programming

- We saw in our example GUI programs that GUI code can get messy
- Thus, there are many useful patterns people have established for GUIs
Model-View-Controller

- Context: GUI displays some data that the user can affect via GUI
- Solution: separate objects into a model, a view and a controller
  - Model - stores the data
  - View - displays the data from Model
  - Controller - maps user actions to model updates
MVC Diagram

View

Model
"Hello World"

Controller

View
### MVC Responsibilities

<table>
<thead>
<tr>
<th>Model</th>
<th>Stores text and formatting markup (fonts, sizes, colors)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Notifies View to update when Model changes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>View</th>
<th>Displays text with proper fonts and sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Displays toolbar</td>
</tr>
<tr>
<td></td>
<td>Notifies Controller when user edits text or clicks toolbar commands</td>
</tr>
</tbody>
</table>

| Controller | Notifies model to change text when user inputs |
|            | Notifies model to perform special commands when toolbar buttons are clicked |
Pattern: Observer

- A subject object is the source of events
- One or more observer objects want to know when an event occurs

Define an observer interface type
- The subject maintains collection of observer objects
- The subject provides methods for attaching observers
- Whenever an event occurs, the subject notifies all observers
Observers in MVC

- View observes Model; when Model changes, it notifies View
- Controller observes View; when user manipulates View, it notifies Controller
Pattern: Composite
JPanel

```java
JPanel panel = new JPanel();
panel.setLayout(new GridLayout(0,1));

panel.add(new JButton("JComponents added"));
panel.add(new JLabel("to this JPanel"));
panel.add(new JTextField("are laid out"));
panel.add(new JButton("by GridLayout"));

frame1.add(panel);
```

```java
frame1.add(new JButton("JComponents added"));
frame1.add(new JLabel("to this JFrame"));
frame1.add(new JTextField("are laid out"));
frame1.add(new JButton("by FlowLayout"));
```

Pattern: Composite

- Primitive objects can be combined into composite objects
- Clients treat a composite object as a primitive object
- Define an interface type that abstracts primitive objects
- Composite object contains a collection of primitive objects
- Both primitive and composite classes implement interface
- When implementing methods from the interface, composite class applies method to its primitive objects and combines the results
Pattern: Decorator
public static void main(String[] args) {
    JFrame frame = new JFrame();
    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(10,10));
    for (int i=0; i<ROWS; i++)
        for (int j=0; j<COLS; j++)
            panel.add(new JButton("Button (" + i + "," + j + ")");
    frame.add(new JScrollPane(panel), BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
}
Pattern: Decorator

- You want to enhance the behavior of a component class
- A decorated component can be used in the same way as a plain component
- The component class shouldn’t be responsible for the decoration
- There may be an open-ended set of possible decorations

- Define an interface type that abstracts the component
- Concrete component classes implement this interface
- Decorator classes also implement this interface
- Decorator objects manage the component that it decorates
Pattern: Strategy
LayoutManager

- **BoxLayout** - draws components in a row or a column
- **BorderLayout** - lets you specify where to draw component (north, south, east, west, center)
- **GridLayout** - draws components in a grid pattern
Different Layouts

```java
JFrame flowFrame = new JFrame("FlowLayout");
JFrame boxFrame = new JFrame("BoxLayout");
JFrame gridFrame = new JFrame("GridLayout");

flowFrame.setLayout(new FlowLayout());
boxFrame.setLayout(new BoxLayout(boxFrame.getContentPane(), BoxLayout.Y_AXIS));
gridFrame.setLayout(new GridLayout(2,3));

for (int i=0; i<6; i++)
{
    flowFrame.add(new JButton("Component "+i));
    boxFrame.add(new JButton("Component "+i));
    gridFrame.add(new JButton("Component "+i));
}
```
Pattern: Strategy

- A context class benefits from different variants of an algorithm
- Clients of the context class sometimes want to supply custom versions of the algorithm

- Define an interface type, called a strategy, that abstracts the algorithm
- Each concrete strategy class implements a version of the algorithm
- The client supplies a concrete strategy object to the context class
- Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object
Using Patterns

- Lots of established, useful patterns
- Make sure the context applies to situation before trying solution
- Understand why pattern solves the problem before applying solution
Reading

• Horstmann Ch. 5
  • Download and try code example(s)
• Next week, we’ll go over some off-book Java GUI material