COMP 303 - Lecture Notes for Week 5 - Patterns and GUI Programming

- Slides edited from, Object-Oriented Design Patterns, by Cay S. Horstmann
- Original slides available from: http://www.horstmann.com/design_and_patterns.html
- Modifications made by Laurie Hendren, McGill University
- Topics this week:
  o Patterns and GUI Programming, Chapter 5
  o Profiling Java code

Chapter Topics
- Iterators
- The Pattern Concept
- The OBSERVER Pattern
- Layout Managers and the STRATEGY Pattern
- Components, Containers, and the COMPOSITE Pattern
- Scroll Bars and the DECORATOR Pattern
- How to Recognize Patterns
- Putting Patterns to Work

**List Iterators**

LinkedList list = . . .;
ListIterator iterator = list.listIterator();
while (iterator.hasNext())
{
   Object current = iterator.next();
   . . .
}

- Why iterators?

**Classical List Data Structure**

Traverse links directly
Link currentLink = list.head;
while (currentLink != null)
{
   Object current = currentLink.data;
   currentLink = currentLink.next;
}

- Exposes implementation
- Error-prone
High-Level View of Data Structures

- Queue
  - Insert in back
  - Remove from front
- Array with random access
  - Get/set access all positions
- List

List with Cursor

```java
for (list.reset(); list.hasNext(); list.next())
{
   Object x = list.get();
   . . .
}
```

- Disadvantage: Only one cursor per list
- Iterator is superior concept

The Pattern Concept

- History: Architectural Patterns
- Christopher Alexander
- Each pattern has
  - A short name
  - A brief description of the context
  - A lengthy description of the problem
  - A prescription for the solution

Short Passages Pattern
Short Passages Pattern

Context
"...Long, sterile corridors set the scene for everything bad about modern architecture..."

Problem
a lengthy description of the problem, including
- a depressing picture
- issues of light and furniture
- research about patient anxiety in hospitals
- research that suggests that corridors over 50 ft are considered uncomfortable

Solution
Keep passages short. Make them as much like rooms as possible, with carpets or wood on the floor, furniture, bookshelves, beautiful windows. Make them generous in shape and always give them plenty of light; the best corridors and passages of all are those which have windows along an entire wall.

Iterator Pattern

Context
1. An aggregate object contains element objects
2. Clients need access to the element objects
3. The aggregate object should not expose its internal structure
4. Multiple clients may want independent access

Solution
1. Define an iterator that fetches one element at a time
2. Each iterator object keeps track of the position of the next element
3. If there are several aggregate/iterator variations, it is best if the aggregate and iterator classes realize common interface types.
**Iterator Pattern**

- Names in pattern are examples
- Names differ in each occurrence of pattern

<table>
<thead>
<tr>
<th>Name in Design Pattern</th>
<th>Actual Name (linked lists)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate</td>
<td>List</td>
</tr>
<tr>
<td>ConcreteAggregate</td>
<td>LinkedList</td>
</tr>
<tr>
<td>Iterator</td>
<td>ListIterator</td>
</tr>
<tr>
<td>ConcreteIterator</td>
<td>anonymous class implementing ListIterator</td>
</tr>
</tbody>
</table>

- `createIterator()`: listIterator()
- `next()`
- `isDone()` opposite of `hasNext()`
- `currentItem()` return value of `hasNext()`

---

**Model/View/Controller**

- Some programs have multiple editable views
- Example: HTML Editor
  - WYSIWYG view
  - structure view
  - source view
- Editing one view updates the other
- Updates seem instantaneous

---

**Model/View/Controller**

- What they are:
  - Model: data structure, no visual representation
  - Views: visual representations
  - Controllers: user interaction
- What they do:
  - Views/controllers update model
  - Model tells views that data has changed
  - Views redraw themselves

---

**Model/View/Controller**

What they are:

- Model: data structure, no visual representation
- Views: visual representations
- Controllers: user interaction

What they do:

- Views/controllers update model
- Model tells views that data has changed
- Views redraw themselves
Observer Pattern

Context

1. An object, called the subject, is source of events
2. One or more observer objects want to be notified when such an event occurs.

Solution

1. Define an observer interface type. All concrete observers implement it.
2. The subject maintains a collection of observers.
3. The subject supplies methods for attaching and detaching observers.
4. Whenever an event occurs, the subject notifies all observers.

Observer Pattern
Names in Observer Pattern

<table>
<thead>
<tr>
<th>Name in Design Pattern</th>
<th>Actual Name (Swing buttons)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>JButton</td>
</tr>
<tr>
<td>Observer</td>
<td>ActionListener</td>
</tr>
<tr>
<td>ConcreteObserver</td>
<td>the class that implements the ActionListener interface type</td>
</tr>
<tr>
<td>attach()</td>
<td>addActionListener()</td>
</tr>
<tr>
<td>notify()</td>
<td>actionPerformed()</td>
</tr>
</tbody>
</table>

Layout Managers

- User interfaces made up of components
- Components placed in containers
- Container needs to arrange components
- Swing doesn’t use hard-coded pixel coordinates
- Advantages:
  - Can switch ”look and feel”
  - Can internationalize strings
- Layout manager controls arrangement

Layout Managers

- FlowLayout: left to right, start new row when full
- BoxLayout: left to right or top to bottom
- BorderLayout: 5 areas, Center, North, South, East, West
- GridLayout: grid, all components have same size
- GridBagLayout: complex, like HTML table

Layout Managers
## Layout Managers

- Set layout manager
  ```java
  JPanel keyPanel = new JPanel();
  keyPanel.setLayout(new GridLayout(4, 3));
  ```
- Add components
  ```java
  for (int i = 0; i < 12; i++)
  keyPanel.add(button[i]);
  ```

## Voice Mail System GUI

- Same backend as text-based system
- Only `Telephone` class changes
- Buttons for keypad
- Text areas for microphone, speaker
Voice Mail System GUI

- Arrange keys in panel with GridLayout:

```java
JPanel keyPanel = new JPanel();
kPannel.setLayout(new GridLayout(4, 3));
for (int i = 0; i < 12; i++)
    JButton keyButton = new JButton(...);
    keyPanel.add(keyButton);
    keyButton.addActionListener(...);
```

Voice Mail System GUI

- Panel with BorderLayout for speaker

```java
JPanel speakerPanel = new JPanel();
speakerPanel.setLayout(new BorderLayout());
speakerPanel.add(new JLabel("Speaker:"), BorderLayout.NORTH);
speakerPanel.add(speakerField, BorderLayout.CENTER);
```

Voice Mail System GUI

- Put speaker, keypads, and microphone panel into content pane

```java
Ch5/mailgui/Telephone.java
```

Voice Mail System GUI

- Content pane already has BorderLayout

```java
Speaker:
You have reached mailbox 12.
Please leave a message now.
```

Voice Mail System GUI

```
Microphone:
Hello, File! This is Aramis. Are we still on for lunch today? Please call me back. Thanks!
```
public class Telephone {
    /** Constructs a telephone with a speaker, keypad, and microphone. */
    public Telephone() {
        JPanel speakerPanel = new JPanel();
        speakerPanel.setLayout(new BorderLayout());
        JPanel speakerField = new JTextArea(10, 25);
        JButton speechButton = new JButton("Send speech");
        speechButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent event) {
                String output = speechButton.getText();
                microphoneField.setText(output);
            }
        });
        JPanel microphonePanel = new JPanel();
        JLabel microphoneLabel = new JLabel("Microphone:");
        JTextField microphoneField = new JTextField(10, 20);
        JButton hangupButton = new JButton("Hangup");
        hangupButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent event) {
                connect.hangup();
            }
        });
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(speechButton);
        buttonPanel.add(hangupButton);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.show();
    }
    /**
     * Give instructions to the mail system user.
     */
    public void speak(String output) {
        speakerField.setText(output);
    }
    public void run(Connection c) {
        connect = c;
    }
    private JTextArea speakerField;
    private Connection connect;
}

Custom Layout Manager

- Form layout
- Odd-numbered components right aligned
- Even-numbered components left aligned
- Implement LayoutManager interface type
The **LayoutManager Interface Type**

```java
public interface LayoutManager {
    void layoutContainer(Container parent);
    Dimension minimumLayoutSize(Container parent);
    Dimension preferredLayoutSize(Container parent);
    void addLayoutComponent(String name, Component comp);
    void removeLayoutComponent(Component comp);
}
```

Form Layout

- Ch5/layout/FormLayout.java
- Ch5/layout/FormLayoutTest.java
- Note: Can use GridBagLayout to achieve the same effect
import java.awt.*;
import javax.swing.*;

public class FormLayoutTest {
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        Container contentPane = frame.getContentPane();
        contentPane.setLayout(new FormLayout());
        contentPane.add(new JLabel("Name"));
        contentPane.add(new JTextField(15));
        contentPane.add(new JLabel("Address"));
        contentPane.add(new JTextField(20));
        contentPane.add(new JLabel("City"));
        contentPane.add(new JTextField(10));
        contentPane.add(new JLabel("State"));
        contentPane.add(new JTextField(2));
        contentPane.add(new JLabel("ZIP"));
        contentPane.add(new JTextField(5));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.show();
    }
}

---

Strategy Pattern

Context

1. A class can benefit from different variants for an algorithm
2. Clients sometimes want to replace standard algorithms with custom versions

Solution

1. Define an interface type that is an abstraction for the algorithm
2. Actual strategy classes realize this interface type.
3. Clients can supply strategy objects
4. Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object
**Strategy Pattern: Layout Management**

<table>
<thead>
<tr>
<th>Name in Design Pattern</th>
<th>Actual Name (layout management)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>Container</td>
</tr>
<tr>
<td>Strategy</td>
<td>LayoutManager</td>
</tr>
<tr>
<td>ConcreteStrategy</td>
<td>a layout manager such as BorderLayout</td>
</tr>
<tr>
<td>doWork()</td>
<td>a method such as layoutContainer</td>
</tr>
</tbody>
</table>

**Strategy Pattern: Sorting**

<table>
<thead>
<tr>
<th>Name in Design Pattern</th>
<th>Actual Name (sorting)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>Collections</td>
</tr>
<tr>
<td>Strategy</td>
<td>Comparator</td>
</tr>
<tr>
<td>ConcreteStrategy</td>
<td>a class that implements Comparator</td>
</tr>
<tr>
<td>doWork()</td>
<td>compare</td>
</tr>
</tbody>
</table>

**Containers and Components**

- Containers collect GUI components
- Sometimes, want to add a container to another container
- Container should be a component
- Composite design pattern
- Composite method typically invoke component methods
- E.g. Container.getPreferredSize invokes getPreferredSize of components

**Composite Pattern**

**Context**

1. Primitive objects can be combined to composite objects
2. Clients treat a composite object as a primitive object

**Solution**

1. Define an interface type that is an abstraction for the primitive objects
2. Composite object collects primitive objects
3. Composite and primitive classes implement same interface type.
4. When implementing a method from the interface type, the composite class applies the method to its primitive objects and combines the results
Composite Pattern

- **Primitive** Component
- **Composite Container**
- **Leaf** a component without children (e.g. JButton)
- method() a method of Component (e.g. getPreferredSize)

---

Scroll Bars

- Scroll bars can be attached to components
- Approach #1: Component class can turn on scroll bars
- Approach #2: Scroll bars can surround component
  ```java
  JScrollPane pane = new JScrollPane(component);
  
  Swing uses approach #2
  JScrollPane is again a component
  ```
Decorator Pattern

Context
1. Component objects can be decorated (visually or behaviorally enhanced)
2. The decorated object can be used in the same way as the undecorated object
3. The component class does not want to take on the responsibility of the decoration
4. There may be an open-ended set of possible decorations

Decorator Pattern

Solution
1. Define an interface type that is an abstraction for the component
2. Concrete component classes realize this interface type.
3. Decorator classes also realize this interface type.
4. A decorator object manages the component object that it decorates
5. When implementing a method from the component interface type, the decorator class applies the method to the decorated component and combines the result with the effect of the decoration.

Decorator Pattern: Scroll Bars

<table>
<thead>
<tr>
<th>Name in Design Pattern</th>
<th>Actual Name (scroll bars)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Component</td>
</tr>
<tr>
<td>ConcreteComponent</td>
<td>JTextArea</td>
</tr>
<tr>
<td>Decorator</td>
<td>JScrollPane</td>
</tr>
<tr>
<td>method()</td>
<td>a method of Component (e.g. paint)</td>
</tr>
</tbody>
</table>
Streams
InputStreamReader reader = new InputStreamReader(System.in);
BufferedReader console = new BufferedReader(reader);

- BufferedReader takes a Reader and adds buffering
- Result is another Reader: Decorator pattern
- Many other decorators in stream library, e.g. PrintWriter

Decorator Pattern: Input Streams

<table>
<thead>
<tr>
<th>Name in Design Pattern</th>
<th>Actual Name (input streams)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Reader</td>
</tr>
<tr>
<td>ConcreteComponent</td>
<td>InputStreamReader</td>
</tr>
<tr>
<td>Decorator</td>
<td>BufferedReader</td>
</tr>
<tr>
<td>method()</td>
<td>read</td>
</tr>
</tbody>
</table>

How to Recognize Patterns
- Look at the intent of the pattern
- E.g. COMPOSITE has different intent than DECORATOR
- Remember common uses (e.g. STRATEGY for layout managers)
- Not everything that is strategic is an example of STRATEGY pattern
- Use context and solution as "litmus test"

Litmus Test
- Can add border to Swing component
  ```java
  Border b = new EtchedBorder();
  component.setBorder(b);
  ```
- Undeniably decorative
- Is it an example of DECORATOR?
**Litmus Test**

1. Component objects can be decorated (visually or behaviorally enhanced)
   **PASS**
2. The decorated object can be used in the same way as the undecorated object
   **PASS**
3. The component class does not want to take on the responsibility of the decoration
   **FAIL**--the component class has `setBorder` method
4. There may be an open-ended set of possible decorations

---

**Putting Patterns to Work**

- Invoice contains *line items*
- Line item has description, price
- Interface type `LineItem`
  - `Ch5/invoice/LineItem.java`
- Product is a concrete class that implements this interface:
  - `Ch5/invoice/Product.java`

```java
/**
 * A line item in an invoice.
 */
public interface LineItem {
    /**
     * Gets the price of this line item.
     * @return the price
     */
    double getPrice();

    /**
     * Gets the description of this line item.
     * @return the description
     */
    String toString();
}

/**
 * A product with a price and description.
 */
public class Product implements LineItem {
    /**
     * Constructs a product.
     * @param description the description
     * @param price the price
     */
    public Product(String description, double price) {
        this.description = description;
        this.price = price;
    }

    public double getPrice() { return price; }

    public String toString() { return description; }

    private String description;
    private double price;
}
```
Bundles

- Bundle = set of related items with description+price
- E.g. stereo system with tuner, amplifier, CD player + speakers
- A bundle has line items
- A bundle is a line item
- COMPOSITE pattern
- Ch5/invoice/Bundle.java (look at getPrice)

```java
import java.util.*;

public class Bundle implements LineItem {

    private ArrayList items;

    public Bundle() { items = new ArrayList(); }

    public void add(LineItem item) { items.add(item); }

    public double getPrice() {
        double price = 0;
        for (int i = 0; i < items.size(); i++) {
            LineItem item = (LineItem) items.get(i);
            price += item.getPrice();
        }
        return price;
    }

    public String toString() {
        String description = "Bundle: ";
        for (int i = 0; i < items.size(); i++) {
            if (i > 0) description += ", ";
            LineItem item = (LineItem) items.get(i);
            description += item.toString();
        }
        return description;
    }
}
```

Bundles

```
Calls getPrice() for each line item and adds the results
```
Discounted Items

- Store may give discount for an item
- Discounted item is again an item
- DECORATOR pattern
- Ch5/invoice/DiscountedItem.java (look at getPrice)
- Alternative design: add discount to LineItem
**Change Listeners**

- Use standard `ChangeListener` interface type

  ```java
  public interface ChangeListener
  {
    void stateChanged(ChangeEvent event);
  }
  ```

- Invoice collects `ArrayList` of change listeners
- When the invoice changes, it notifies all listeners:

  ```java
  ChangeEvent event = new ChangeEvent(this);
  for (int i = 0; i < listeners.size(); i++)
  {
    ChangeListener listener = (ChangeListener)listeners.get(i);
    listener.stateChanged(event);
  }
  ```

**Display** adds itself as a change listener to the invoice
- Display updates itself when invoice object changes state

  ```java
  final Invoice invoice = new Invoice();
  final JTextArea textArea = new JTextArea(20, 40);
  ```

  ```java
  final Invoice invoice = new Invoice();
  final JTextArea textArea = new JTextArea(20, 40);
  ```

  ```java
  ChangeListener listener = new ChangeListener()
  {
    public void stateChanged(ChangeEvent event)
    {
      textArea.setText(...);
    }
  };
  ```

**Observing the Invoice**

- Invoice collect line items
- Clients need to iterate over line items
- Don’t want to expose `ArrayList`
- May change (e.g. if storing invoices in database)
- ITERATOR pattern

**Iterating Through Invoice Items**

- Invoice collect line items
- Clients need to iterate over line items
- Don’t want to expose `ArrayList`
- May change (e.g. if storing invoices in database)
- ITERATOR pattern
Iterators

- Use standard `Iterator` interface type
- `public interface Iterator`
  ```java
  boolean hasNext();
  Object next();
  void remove();
  ```
- `remove` is "optional operation" (see ch. 8)
- Implement to throw `UnsupportedException`
- Implement `hasNext/next` manually to show inner workings
- Ch5/invoice/Invoice.java

```java
import java.util.*;
import javax.swing.event.*;

public class Invoice {
    private ArrayList items;
    private ArrayList listeners;

    public Invoice() {
        items = new ArrayList();
        listeners = new ArrayList();
    }

    public void addItem(LineItem item) {
        items.add(item);
        // notify all observers of the change to the invoice
        ChangeEvent event = new ChangeEvent(this);
        for (int i = 0; i < listeners.size(); i++) {
            ChangeListener listener = (ChangeListener) listeners.get(i);
            listener.stateChanged(event);
        }
    }

    public void addChangeListener(ChangeListener listener) {
        listeners.add(listener);
    }

    /**
     * Gets an iterator that iterates through the items.
     * @return an iterator for the items
     */
    public Iterator getItems() {
        return new Iterator() {
            private int current = 0;

            public boolean hasNext() {
                return current < items.size();
            }

            public Object next() {
                Object r = items.get(current);
                current++;
                return r;
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }

            private String format(StringBuilder builder, LineItem item) {
                return builder.append(formatter.formatLineItem(item)).toString();
            }

            @Override
            public String format(StringBuilder builder) {
                return format(builder, item);
            }
        };
    }

    public String format(InvoiceFormatter formatter) {
        StringBuilder builder = new StringBuilder();
        builder.append(formatter.formatHeader());
        Iterator iter = getItems();
        while (iter.hasNext()) {
            LineItem item = (LineItem) iter.next();
            r += formatter.formatLineItem(item);
        }
        return builder.append(formatter.formatFooter()).toString();
    }
}
```
Iterators

```
previous | start | next .... 
[Slide 66] ....
```

## Formatting Invoices

- Simple format: dump into text area
- May not be good enough.
- E.g. HTML tags for display in browser
- Want to allow for multiple formatting algorithms
- STRATEGY pattern

```
previous | start | next .... 
[Slide 67] ....
```

```
Formatting Invoices

ch5/invoice/InvoiceFormatter.java
ch5/invoice/SimpleFormatter.java
ch5/invoice/InvoiceTest.java
```

```
01: /**
02: This interface describes the tasks that an invoice 
03: formatter needs to carry out.
04: */
05: ...
06: */
07: /**
08: Formats the header of the invoice.
09: @return the invoice header
10: */
11: String formatHeader();
12: /**
13: Formats a line item of the invoice.
14: @return the formatted line item
15: */
16: String formatLineItem(LineItem item);
17: /**
18: Formats the footer of the invoice.
19: @return the invoice footer
20: */
21: String formatFooter();
22: }
```

```
Formatting Invoices

ch5/invoice/InvoiceFormatter.java
ch5/invoice/SimpleFormatter.java
ch5/invoice/InvoiceTest.java
```

```
01: /**
02: This interface describes the tasks that an invoice 
03: formatter needs to carry out.
04: */
05: ...
06: */
07: /**
08: Formats the header of the invoice.
09: @return the invoice header
10: */
11: String formatHeader();
12: /**
13: Formats a line item of the invoice.
14: @return the formatted line item
15: */
16: String formatLineItem(LineItem item);
17: /**
18: Formats the footer of the invoice.
19: @return the invoice footer
20: */
21: String formatFooter();
22: }
```
public class SimpleFormatter implements InvoiceFormatter
{
    String formatHeader()
    {
        return "| I N V O I C E |
    }

    String formatFooter()
    {
        return "\n\nTOTAL DUE: $" + total + "\n";
    }

    private double total;
}

public class InvoiceFormatter
{
    LineItem addItem(LineItem item)
    {
        total += item.getPrice();
        return item.toString();
    }

    String formatHeader()
    {
        return "I N V O I C E \n\n";
    }

    String formatFooter()
    {
        return "TOTAL DUE: $" + total + "\n";
    }
}

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class InvoiceTest
{
    public static void main(String[] args)
    {
        final Invoice invoice = new Invoice();
        final InvoiceFormatter formatter = new SimpleFormatter();
        final JTextArea textArea = new JTextArea(20, 40);
        ChangeListener listener = new ChangeListener()
        {
            public void stateChanged(ChangeEvent event)
            {
                textArea.setText(invoice.format(formatter));
            }
        }
        invoice.addChangeListener(listener);
        final JComboBox combo = new JComboBox();
        final Product hammer = new Product("Hammer", 19.95);
        final Product nails = new Product("Assorted nails", 9.95);
        combo.addItem(hammer);
        combo.addItem(nails);
        combo.add(new Discounter(bundle, 10));
        // make a button for adding the currently selected
        // item to the invoice
        JButton addButton = new JButton("Add");
        addButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                LineItem item = (LineItem) combo.getSelectedItem();
                invoice.addItem(item);
            }
        });
        // add box and the add button into a panel
        JPanel panel = new JPanel();
        panel.add(combo);
        panel.add(addButton);
        // add the text area and panel to the content pane
        JFrame frame = new JFrame();
        Container contentPane = frame.getContentPane();
        contentPane.add(new JScrollPane(textArea), BorderLayout.CENTER);
        contentPane.add(panel, BorderLayout.SOUTH);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.show();
    }

    public class Invoice
    {
        // data members
        
        // methods
    }

    public class LineItem
    {
        // data members
        
        // methods
    }

    public class Product
    {
        // data members
        
        // methods
    }

    public class Discounter
    {
        Bundle bundle;
        int discount;
        
        // methods
    }

    public class Bundle
    {
        // data members
        
        // methods
    }
}

Formatting Invoices

| I N V O I C E |
\n\nHammer: $19.95
Bundle: Hammer, Assorted nails (Discount 10.00%) : $26.91
\n\nTOTAL DUE: $46.86
Formatting Invoices

```
Invoice

«interface»
Invoice
Formatter

formatHeader()
formatLineItem()
formatFooter()

Simple
Formatter
```