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
The Iterator Pattern

• List Iterators:

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

• Why iterators?
The Iterator Pattern

• 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
The Iterator Pattern

• High level view of data structures
  • Queue

• Methods:
  • void add(E x)
  • E peek()
  • E remove()
  • int size()
The Iterator Pattern

- High level view of data structures
  - Array with random access

  ![Diagram of array with random access]

- Methods:
  - E get(int i)
  - void set(int i, E x)
  - void add(E x)
  - int size()
The Iterator Pattern

- High level view of data structures
  - Linked list with a cursor (pointer or reference to a node)
  - \texttt{E getCurrent()} //get element at cursor
  - \texttt{void set (E x)} //set element at cursor to x
  - \texttt{E remove()} //remove element at cursor
  - \texttt{void insert(E x)} //insert x before cursor
  - \texttt{void next()} //advance the cursor
  - \texttt{Boolean hasNext()} //check for end of list
The Iterator Pattern

- **Linked list with cursor**
  - We need to be able to add and delete elements in the middle of the list.
  - The state of a list consists of a sequence of nodes and a cursor pointing to one element.
  - Get, set, insert, and remove methods are relative to the cursor position.
  - Traversing this list:
    ```java
    for (list.reset(); list.hasnext(); list.next())
        //do something with list.getCurrent();
    ```
- **Pros:**
  - At least this implementation hides the pointers.
  - No separate iterator class is required.
- **Cons:**
  - Since there is only one cursor, it’s hard to compare elements
  - Printing the list has the side effect of moving the cursor.

- **The iterator is a superior concept.**
  - Can have multiple iterators as needed
  - Iterator concept is useful in other areas such as Scanner, and Streams
  - This makes the iterator into a pattern
The Pattern Concept

• Christopher Alexander – Patterns in architectural design
• Alexander et al., A Pattern Language: Towns, Buildings, Construction, Oxford Univ. Press, 1977
• Also: Alexander, The Timeless Way of Building, Oxford Univ. Press, 1979

• A pattern consists of:
  • A short name
  • A brief description of the problem
  • A prescription for a solution.

• A pattern must be applicable to multiple similar situations

• Examples:
  • In designing room layout: People feel more comfortable in rooms with windows in at least two walls
  • Hallways: Keep them as short as possible, make them look like rooms with carpet, shelves, windows, seating alcoves.
Design Patterns

• The pattern concept was introduced to the software community by the ‘gang of four’, Gamma, Helm, Johnson, and Vlissides, Design Patterns; Elements of Reusable Object-Oriented Software, Addison Wesley, 1994.

• A pattern language is a collection of patterns that can guide you through an entire design and implementation process.

• Antipatterns – things to avoid
  • The Blob class – A class that has many disparate responsibilities.
  • The Poltergeist – A spurious class whose objects are short-lived and carry no significant responsibilities
The 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

• Short Passages Pattern

• 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.
The Iterator Pattern

• Context

• An aggregate object contains element objects
• Clients need access to the element objects
• The aggregate object should not expose its internal structure
• Multiple clients may want independent access
The Iterator Pattern

- **Solution**
  - Define an iterator that fetches one element at a time
  - Each iterator object keeps track of the position of the next element
  - If there are several aggregate/iterator variations, it is best if the aggregate and iterator classes realize common interface types.
The 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>ConcretelIterator</td>
<td>anonymous class implementing ListIterator</td>
</tr>
<tr>
<td>createlIterator()</td>
<td>listIterator()</td>
</tr>
<tr>
<td>next()</td>
<td>next()</td>
</tr>
<tr>
<td>isDone()</td>
<td>opposite of hasNext()</td>
</tr>
<tr>
<td>currentItem()</td>
<td>return value of hasNext()</td>
</tr>
</tbody>
</table>
Some programs have multiple editable views
Example: HTML Editor
WYSIWYG view
structure view
source view
Editing one view updates the other
Updates seem instantaneous
Amaya is a Web client that acts both as a browser and a WYSIWYG editor. It has been designed by W3C with the primary purpose of demonstrating new technologies in a What You See Is What You Get (WYSIWYG) environment. The current version implements the Extensible Hypertext Markup Language (MathML), Scalable Vector Graphics (SVG), and Synchronized Multimedia Integration Language (SMIL).
Model/View/Controller

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

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

- Model notifies views when something interesting happens
- Button notifies action listeners when something interesting happens
- Views attach themselves to model in order to be notified
- Action listeners attach themselves to button in order to be notified
- Generalize: Observers attach themselves to subject
The Observer Pattern

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

• Solution
  • Define an observer interface type. All concrete observers implement it.
  • The subject maintains a collection of observers.
  • The subject supplies methods for attaching and detaching observers.
  • Whenever an event occurs, the subject notifies all observers.
The Observer Pattern
The 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

- **FlowLayout**
- **BoxLayout (horizontal)**
- **BoxLayout (vertical)**
- **BorderLayout**
- **GridLayout**
- **GridBagLayout**
Layout Managers

• Set layout manager:

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

[Diagram showing relationships between Container, "interface" Layout Manager, JPanel, and GridLayout]
Voice Mail System GUI

- Same backend as text-based system
- Only Telephone class changes
- Buttons for keypad
- Text areas for microphone, speaker
Layout Managers

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

Microphone:
Hello Fifi! This is Aramis. Are we still on for lunch today? Please call me back. Thanks!
Voice Mail System GUI

• Arrange keys in panel with GridLayout:

```java
JPanel keyPanel = new JPanel();
keyPanel.setLayout(new GridLayout(4, 3));
for (int i = 0; i < 12; i++)
{
    JButton keyButton = new JButton(...);
    keyPanel.add(keyButton);
    keyButton.addActionListener(...);
}
```
Panel with BorderLayout for speaker

```java
JPanel speakerPanel = new JPanel();
speakerPanel.setLayout(new BorderLayout());
speakerPanel.add(new JLabel("Speaker:"), BorderLayout.NORTH);
speakerField = new JTextArea(10, 25);
speakerPanel.add(speakerField, BorderLayout.CENTER);
```
Layout Managers

- Microphone Panel:
Voice Mail System GUI

• Add speaker, keypads, and microphone panel to frame
• Frame already has BorderLayout
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Presents a phone GUI for the voice mail system.
 */
public class Telephone {
    /** Constructs a telephone with a speaker, keypad, and microphone. */
    public Telephone() {
        JPanel speakerPanel = new JPanel();
        speakerPanel.setLayout(new BorderLayout());
        speakerPanel.add(new JLabel("Speaker:"), BorderLayout.NORTH);
        speakerField = new JTextArea(10, 25);
        speakerPanel.add(speakerField, BorderLayout.CENTER);

        String keyLabels = "123456789*0#";
        JPanel keyPanel = new JPanel();
        keyPanel.setLayout(new GridLayout(4, 3));
for (int i = 0; i < keyLabels.length(); i++)
{
    final String label = keyLabels.substring(i, i + 1);
    JButton keyButton = new JButton(label);
    keyPanel.add(keyButton);
    keyButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            connect.dial(label);
        }
    });
}

final JTextArea microphoneField = new JTextArea(10, 25);
Voice Mail System GUI

JButton speechButton = new JButton("Send speech");
speechButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         connect.record(microphoneField.getText());
         microphoneField.setText("");
      }
   });

JButton hangupButton = new JButton("Hangup");
hangupButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         connect.hangup();
      }
   });
Voice Mail System GUI

```java
JPanel buttonPanel = new JPanel();
    buttonPanel.add(speechButton);
    buttonPanel.add(hangupButton);

JPanel microphonePanel = new JPanel();
    microphonePanel.setLayout(new BorderLayout());
    microphonePanel.add(new JLabel("Microphone:"), BorderLayout.NORTH);
    microphonePanel.add(microphoneField, BorderLayout.CENTER);
    microphonePanel.add(buttonPanel, BorderLayout.SOUTH);

JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.add(speakerPanel, BorderLayout.NORTH);
    frame.add(keyPanel, BorderLayout.CENTER);
    frame.add(microphonePanel, BorderLayout.SOUTH);

    frame.pack();
    frame.setVisible(true);
```
Voice Mail System GUI

/**
   * 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
Custom Layout Manager

• 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);
}
```
import java.awt.*;

/** A layout manager that lays out components along a central axis */

public class FormLayout implements LayoutManager{

    public Dimension preferredLayoutSize(Container parent)
    {
        Component[] components = parent.getComponents();
        left = 0;
        right = 0;
        height = 0;
        for (int i = 0; i < components.length; i += 2)
        {
            Component cleft = components[i];
            Component cright = components[i + 1];
            Dimension dleft = cleft.getPreferredSize();
            Dimension dright = cright.getPreferredSize();
            left = Math.max(left, dleft.width);
            right = Math.max(right, dright.width);
            height = height + Math.max(dleft.height, dright.height);
        }
        return new Dimension(left + GAP + right, height);
    }
}
Form Layout

public Dimension minimumLayoutSize(Container parent)
{
    return preferredLayoutSize(parent);
}

public void layoutContainer(Container parent)
{
    preferredLayoutSize(parent); // Sets left, right

    Component[] components = parent.getComponents();

    Insets insets = parent.getInsets();
    int xcenter = insets.left + left;
    int y = insets.top;

public class FormLayout {

    private int left;
    private int right;
    private int height;
    private static final int GAP = 6;

    public void addLayoutComponent(String name, Component comp) {
    }

    public void removeLayoutComponent(Component comp) {
    }

    // Other methods...

    for (int i = 0; i < components.length; i += 2) {
        Component cleft = components[i];
        Component cright = components[i + 1];
        Dimension dleft = cleft.getPreferredSize();
        Dimension dright = cright.getPreferredSize();
        int height = Math.max(dleft.height, dright.height);
        cleft.setBounds(xcenter - dleft.width, y + (height - dleft.height) / 2, dleft.width, dleft.height);
        cright.setBounds(xcenter + GAP, y + (height - dright.height) / 2, dright.width, dright.height);
        y += height;
    }
}
import java.awt.*;
import javax.swing.*;

public class FormLayoutTester
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        frame.setLayout(new FormLayout());
        frame.add(new JLabel("Name"));
        frame.add(new JTextField(15));
        frame.add(new JLabel("Address"));
        frame.add(new JTextField(20));
        frame.add(new JLabel("City"));
        frame.add(new JTextField(10));
        frame.add(new JLabel("State"));
        frame.add(new JTextField(2));
        frame.add(new JLabel("ZIP"));
        frame.add(new JTextField(5));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
Strategy Pattern

- Pluggable strategy for layout management
- Layout manager object responsible for executing concrete strategy
- Generalizes to Strategy Design Pattern
- Other manifestation: Comparators

```java
Comparator<Country> comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```
Strategy Pattern

**Context**

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

**Solution**

- Define an interface type that is an abstraction for the algorithm
- Actual strategy classes realize this interface type.
- Clients can supply strategy objects
- Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object
Strategy Pattern

```
Context

«interface» Strategy
    doWork()

Concrete Strategy
```
**Strategy Pattern**

- **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>
<tr>
<td>Name in Design Pattern</td>
<td>Actual Name (sorting)</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<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>
Composite Pattern

• 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
  
  • Primitive objects can be combined to composite objects
  • Clients treat a composite object as a primitive object

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

Calls `method()` for each primitive and combines the results.
## Composite Pattern

<table>
<thead>
<tr>
<th>Name in Pattern</th>
<th>Actual Name (AWT components)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>Component</td>
</tr>
<tr>
<td>Composite</td>
<td>Container</td>
</tr>
<tr>
<td>Leaf</td>
<td>a component without children (e.g. JButton)</td>
</tr>
<tr>
<td>method()</td>
<td>a method of Component (e.g. getPreferredSize)</td>
</tr>
</tbody>
</table>
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
  • JScrollPane pane = new JScrollPane(component);
• Swing uses approach #2
• JScrollPane is again a component
Scroll Bars
Decorator Pattern

• Context:

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

• Solution:

• Define an interface type that is an abstraction for the component
• Concrete component classes realize this interface type.
• Decorator classes also realize this interface type.
• A decorator object manages the component object that it decorates.
• 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

- Concrete Component
- Decorator
  - Calls `method()` for the component and augments the results
## 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>
Decorator Pattern

• 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 a Pattern

• 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"
Can add border to Swing component
Border b = new EtchedBorder()
component.setBorder(b);
Undeniably decorative
Is it an example of DECORATOR?

Litmus Test:
Component objects can be decorated (visually or behaviorally enhanced)
PASS
The decorated object can be used in the same way as the undecorated object
PASS
The component class does not want to take on the responsibility of the decoration
FAIL--the component class has setBorder method
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:

```java
01: /**
02:  A line item in an invoice.
03: */
04: public interface LineItem
05: {
06:  /**
07:   Gets the price of this line item.
08:   @return the price
09: */
10:   double getPrice();
11:  /**
12:   Gets the description of this line item.
13:   @return the description
14: */
15:   String toString();
16: }
```
Putting Patterns to Work

• Product is a concrete class that implements the LineItem interface:

  • 01: /** A product with a price and description.
  • 03: */
  • 04: public class Product implements LineItem
  • 05: {
  • 11: public Product(String description, double price)
  • 12: {
  • 13:     this.description = description;
  • 14:     this.price = price;
  • 15: }  
  • 16: public double getPrice() { return price; }  
  • 17: public String toString() { return description; }  
  • 18: private String description;
  • 19: private double price;
  • 20: }
Putting Patterns to Work

• 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
• Bundles

```java
import java.util.*;
/**
    A bundle of line items that is again a line item.
 */
public class Bundle implements LineItem {

    /**
       Constructs a bundle with no items.
    */
    public Bundle() { items = new ArrayList<>(); }

    /**
       Adds an item to the bundle.
       @param item the item to add
    */
    public void add(LineItem item) { items.add(item); }

    public double getPrice() {
        double price = 0;
        for (LineItem item : items) price += item.getPrice();
        return price;
    }

```
Putting Patterns to Work

• Bundles

28: public String toString()
29: {
30:     String description = "Bundle: ";
31:     for (int i = 0; i < items.size(); i++)
32:         {
33:             if (i > 0) description += ", ";
34:             description += items.get(i).toString();
35:         }
36:     return description;
37: }
38: 
39: private ArrayList<LineItem> items;
40: }
Putting Patterns to Work

• Discounted Items
• Store may give discount for an item
• Discounted item is again an item
• DECORATOR pattern
• Alternative design: add discount to LineItem
Model View Separation

- GUI has commands to add items to invoice
- GUI displays invoice
- Decouple input from display
- Display wants to know when invoice is modified
- Display doesn't care which command modified invoice
- OBSERVER pattern
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 (ChangeListener listener : listeners)
    listener.stateChanged(event);
  ```
Change Listeners

• 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);
ChangeListener listener = new ChangeListener()
{
    public void stateChanged(ChangeEvent event)
    {
        textArea.setText(...);
    }
};
```
Observing the Invoice
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
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
- Use standard Iterator interface type
  ```java
  public interface Iterator<LineItem>
  {
    boolean hasNext();
    LineItem next();
    void remove();
  }
  ```
  - remove is "optional operation" (see ch. 8)
  - implement to throw UnsupportedException
  - implement hasNext/next manually to show inner workings
import java.util.*;
import javax.swing.event.*;

/**
 * An invoice for a sale, consisting of line items.
 */
public class Invoice
{
    /**
     * Constructs a blank invoice.
     */
    public Invoice()
    {
        items = new ArrayList<LineItem>();
        listeners = new ArrayList<ChangeListener>();
    }
}
Iterating Through Invoice Items

18: /**
19:   Adds an item to the invoice.
20:   @param item the item to add
21: */
22: public void addItem(LineItem item)
23: {  
24:     items.add(item);
25:     // Notify all observers of the change to the invoice
26:     ChangeEvent event = new ChangeEvent(this);
27:     for (ChangeListener listener : listeners)
28:         listener.stateChanged(event);
29: }
30: 
31: /**
32:   Adds a change listener to the invoice.
33:   @param listener the change listener to add
34: */
35: public void addChangeListener(ChangeListener listener)
36: {  
37:     listeners.add(listener);
38: }
/**
 * Gets an iterator that iterates through the items.
 * @return an iterator for the items
 */

public Iterator<LineItem> getItems()
{
    return new Iterator<LineItem>()
    {
        public boolean hasNext()
        {
            return current < items.size();
        }

        public LineItem next()
        {
            return items.get(current++);
        }

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

        private int current = 0;
    };
}
public String format(InvoiceFormatter formatter) {
    String r = formatter.formatHeader();
    Iterator<LineItem> iter = getItems();
    while (iter.hasNext())
        r += formatter.formatLineItem(iter.next());
    return r + formatter.formatFooter();
}

private ArrayList<LineItem> items;
private ArrayList<ChangeListener> listeners;
Iterators

```
Invoice
getItems()

«interface» Iterator
next()
hasNext()

(anonymous class)
```
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
/**
 * This interface describes the tasks that an invoice formatter needs to carry out.
 */

public interface InvoiceFormatter {

    /**
     * Formats the header of the invoice.
     * @return the invoice header
     */
    String formatHeader();

    /**
     * Formats a line item of the invoice.
     * @return the formatted line item
     */
    String formatLineItem(LineItem item);

    /**
     * Formats the footer of the invoice.
     * @return the invoice footer
     */
    String formatFooter();
}
/**
 * A simple invoice formatter.
 */
public class SimpleFormatter implements InvoiceFormatter {
    public String formatHeader() {
        total = 0;
        return "     I N V O I C E
            
            
            
        ";
    }

    public String formatLineItem(LineItem item) {
        total += item.getPrice();
        return String.format("%s: $%.2f\n", item.toString(), item.getPrice());
    }

    public String formatFooter() {
        return String.format("\n\nTOTAL DUE: $%.2f\n", total);
    }

    private double total;
}
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * A program that tests the invoice classes.
 */
public class InvoiceTester {
    public static void main(String[] args) {
        final Invoice invoice = new Invoice();
        final InvoiceFormatter formatter = new SimpleFormatter();

        // This text area will contain the formatted invoice
        final JTextArea textArea = new JTextArea(20, 40);

        // When the invoice changes, update the text area
        ChangeListener listener = new ChangeListener()
        {
            public void stateChanged(ChangeEvent event)
            {
                textArea.setText(invoice.format(formatter));
            }
        };
        invoice.addChangeListener(listener);
// Add line items to a combo box
final JComboBox combo = new JComboBox();
Product hammer = new Product("Hammer", 19.95);
Product nails = new Product("Assorted nails", 9.95);
combo.addItem(hammer);
Bundle bundle = new Bundle();
bundle.add(hammer);
bundle.add(nails);
combo.addItem(new DiscountedItem(bundle, 10));

// Make a button for adding the currently selected item to the invoice
// 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);
    }
});
// Put the combo 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();
frame.add(new JScrollPane(textArea), BorderLayout.CENTER);
frame.add(panel, BorderLayout.SOUTH);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.pack();
frame.setVisible(true);
Formatting Invoices

INVOICE

Hammer: $19.95
Bundle: Hammer, Assorted nails (Discount 10.0%): $26.91

TOTAL DUE: $46.86
Formatting Invoices

Invoice

```
 «interface»
 Invoice
 Formatter

 formatHeader()
 formatLineItem()
 formatFooter()

 Simple
 Formatter
```
End