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:
  - Patterns and GUI Programming, Chapter 5
  - 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

  ```java
  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

  ![](image)

  Insert in back

  Remove from front

- Array with random access

  ![](image)

  get/set access all positions

- List

  ???
List with Cursor

```
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
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.
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>
<tr>
<td>createIterator()</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>

previous | start | next    .... [Slide 13] ....
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

Amaya is a Web client that acts both as a browser and as an authoring tool. Its primary purpose is to demonstrate new Web technologies in a WYSIWYG environment. It supports HTML, XHTML, MathML, SVG, CSS, and HTTP.

Main Features

With Amaya, you can manipulate rich Web pages containing forms, tables, and text. You can create and edit complex mathematical expressions and simple SVG graphics, style your documents using Cascading Style Sheets, and publish HTML and other documents through HTTP.

Browsing and authoring are integrated seamlessly. You can browse and edit Web pages at the same time. To make editing easier, a simple click in Amaya just moves the caret and allows text input; to follow a link, you have to double click. This behavior can be changed as well as many other features using the Special/Preferences menu.
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
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
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.
# 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>

previous | start | next .... [Slide 21] ....
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

previous | start | next .... [Slide 24] ....
Layout Managers

- Set layout manager
  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

Container

«interface»
Layout Manager

JPanel

GridLayout

previous | start | next .... [Slide 26] ....
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

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!

(previous | start | next .... [Slide 28] ....)
Voice Mail System GUI

- Arrange keys in panel with GridLayout:

  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(...);
  }
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);
speakerField = new JTextArea(10, 25);
speakerPanel.add(speakerField, BorderLayout.CENTER);
```

```
Microphone:
Hello, Fif! This is Aramis. Are we still on for lunch today? Please call me back. Thanks!
```

previous | start | next .... [Slide 30] ....
Voice Mail System GUI

- Put speaker, keypads, and microphone panel into content pane
- Content pane already has BorderLayout
- Ch5/mailgui/Telephone.java

```
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!
```

NORTH

CENTER

SOUTH
```java
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);

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();
        }
    });

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);
Container contentPane = frame.getContentPane();
contentPane.add(speakerPanel,
    BorderLayout.NORTH);
    contentPane.add(keyPanel,
        BorderLayout.CENTER);
    contentPane.add(microphonePanel,
        BorderLayout.SOUTH);
    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.*;
/**
 * A layout manager that lays out components along a central axis
 */
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);
    }
    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;
```java
int y = insets.top;

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;
}

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

public void removeLayoutComponent(Component comp) {
}

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

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

```java
Comparator comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```
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

```
context

interface Strategy
  doWork()

Concrete Strategy
```
## 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>

previous | start | next .... [Slide 38] ....
## 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>

previous | start | next .... [Slide 39] ....
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

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

<table>
<thead>
<tr>
<th>Name in Design 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>

[previous | start | next .... [Slide 43] ....]
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

```
previous | start | next .... [Slide 45] ....
```
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

Calls \texttt{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>

previous | start | next .... [Slide 49] ....
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`
/**
 * 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.

```java
public class Product implements LineItem {
    /** Constructs a product. */
    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)
import java.util.*;

/**
 * A bundle of items that is again an item.
 */
public class Bundle implements LineItem {
  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 (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;
private ArrayList items;
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
A decorator for an item that applies a discount.

```java
/**
   * Constructs a discounted item.
   * @param item the item to be discounted
   * @param discount the discount percentage
   */
public DiscountedItem(LineItem item, double discount) {
    this.item = item;
    this.discount = discount;
}

public double getPrice() {
    return item.getPrice() * (1 - discount / 100);
}

public String toString() {
    return item.toString() + " (Discount " + discount + ")";
}

private LineItem item;
private double discount;
```
Discounted Items

Calls `getPrice()` for the line item and applies the discount
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 (int i = 0; i < listeners.size(); i++)
{
    ChangeListener listener = (ChangeListener)listeners.get(i);
    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

```java
Invoice

addChangeListener()

«interface»
Change
Listener

stateChanged()

(anonymous
class)
```
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
  ```java
  public interface Iterator {
      boolean hasNext();
      Object next();
      void remove();
  }
  ```
- `remove` is "optional operation" (see ch. 8)
- `implement to throw UnsupportedOperationException`
- `implement hasNext/next manually to show inner workings`
- `Ch5/invoice/Invoice.java`
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();
        listeners = new ArrayList();
    }

    /**
     * Adds an item to the invoice.
     * @param item the item to add
     */
    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);
        }
    }

    /**
     * Adds a change listener to the invoice.
     * @param listener the change listener to add
     */
    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()
    {
        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 int current = 0;
    };

    public String format(InvoiceFormatter formatter)
    {
        String r = formatter.formatHeader();
        Iterator iter = getItems();
        while (iter.hasNext())
        {
            LineItem item = (LineItem) iter.next();
            r += formatter.formatLineItem(item);
        }

        return r + formatter.formatFooter();
    }
private ArrayList items;
private ArrayList 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
Formatting Invoices

- ch5/invoice/InvoiceFormatter.java
- ch5/invoice/SimpleFormatter.java
- ch5/invoice/InvoiceTest.java
/**
 * 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.
 */

class SimpleFormatter implements InvoiceFormatter {
    public String formatHeader()
    {
        total = 0;
        return "     I N V O I C E\n\n";
    }

    public String formatLineItem(LineItem item)
    {
        total += item.getPrice();
        return item.toString() + ": $" + item.getPrice() + "\n";
    }

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

    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 InvoiceTest {
    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
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();
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();
Formatting Invoices

INVOICE

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

TOTAL DUE: $46.86