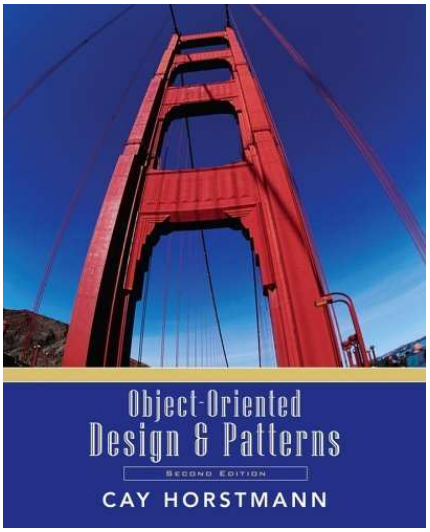
Object-Oriented Design & Patterns

Cay S. Horstmann

Chapter 5

Patterns and GUI Programming



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

- Why iterators?

Classical List Data Structure

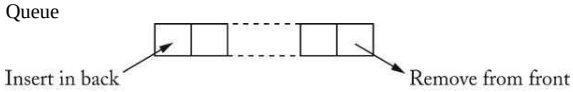
- Traverse links directly

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

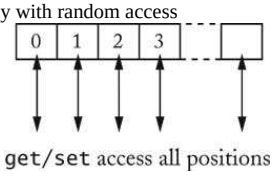
- Exposes implementation
- Error-prone

High-Level View of Data Structures

- Queue

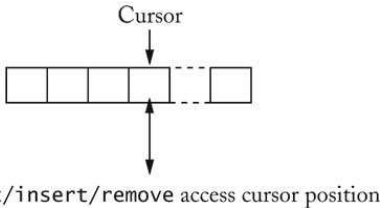


- Array with random access



- 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

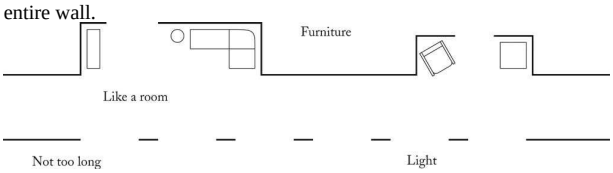
a lengthy description of the problem, including

- 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

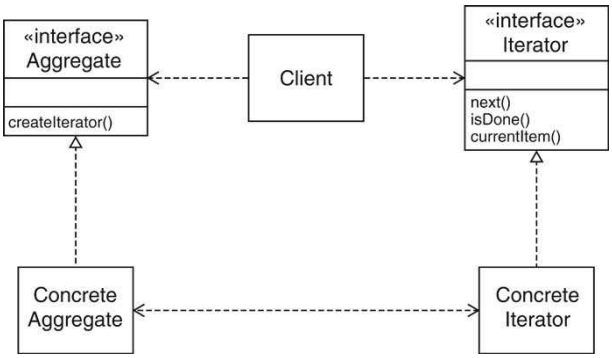
1. An aggregate object contains element objects
2. Clients need access to the element objects
3. The aggregate object should not expose its internal structure
4. Multiple clients may want independent access

Iterator Pattern

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



Iterator Pattern

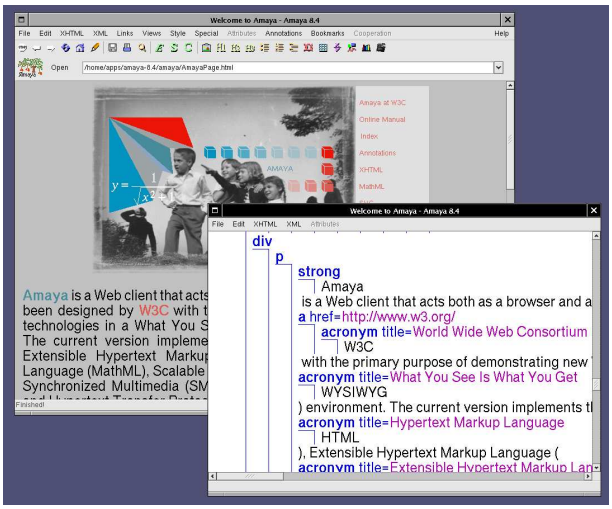
- Names in pattern are *examples*
- Names differ in each occurrence of pattern

Name in Design Pattern	Actual Name (linked lists)
Aggregate	List
ConcreteAggregate	LinkedList
Iterator	ListIterator
ConcreteIterator	anonymous class implementing `ListIterator`
createIterator()	listIterator()
next()	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



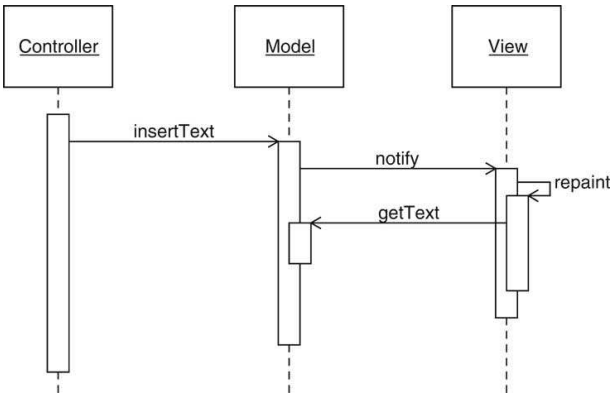
Model/View/Controller

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

Model/View/Controller

- 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

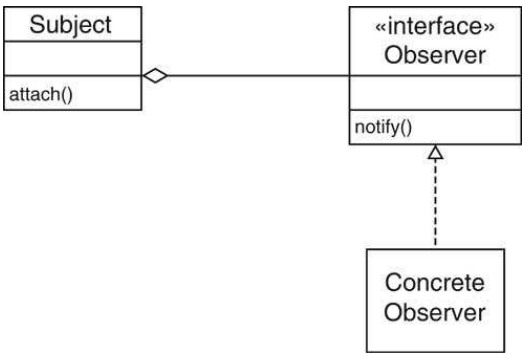
Context

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.

Observer Pattern



Names in Observer Pattern

Name in Design Pattern	Actual Name (Swing buttons)
Subject	`JButton`
Observer	`ActionListener`
ConcreteObserver	the class that implements the `ActionListener` interface type
attach()	`addActionListener()`
notify()	`actionPerformed()`

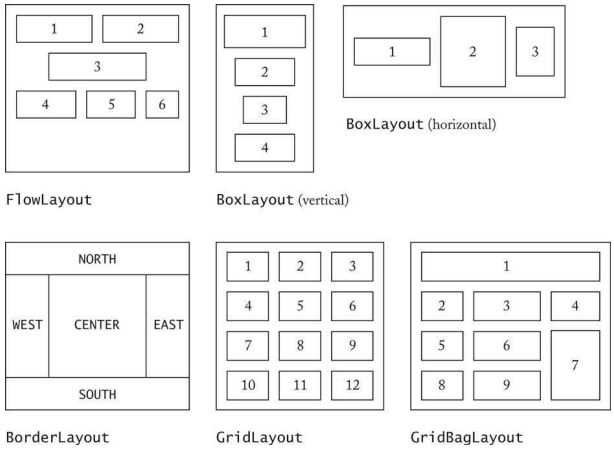
Layout Managers

- 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

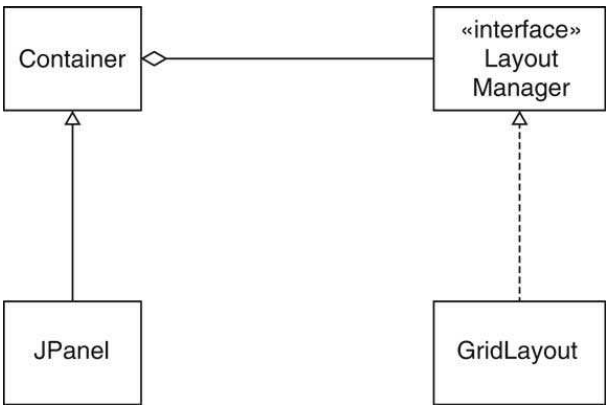
Layout Managers



Layout Managers

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

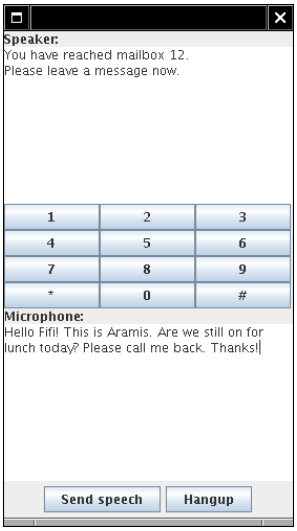

Layout Managers



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



previous | start | next

previous | start | next

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(...);}
```

previous | start | next

previous | start | next

Voice Mail System GUI

- Panel with `BorderLayout` for speaker

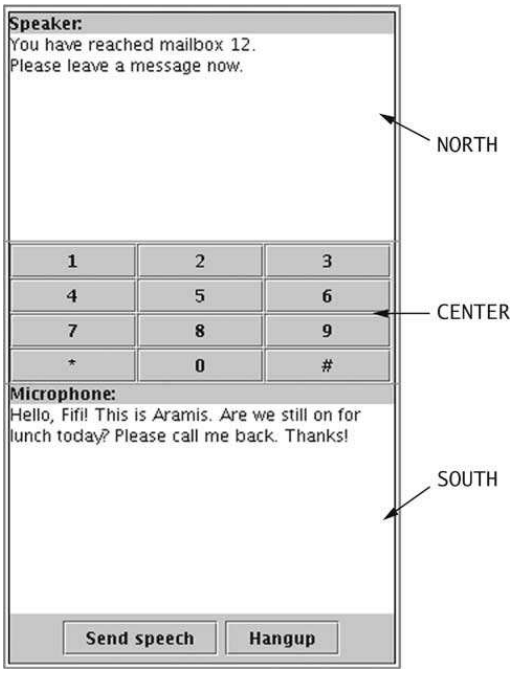
```
JPanel speakerPanel = new JPanel(); speakerPanel.setLayout(new BorderLayout()); speakerPanel.add(new JLabel("Speaker:"), BorderLayout.NORTH); speakerField = new JTextArea(10, 20); speakerPanel.add(speakerField, BorderLayout.CENTER);
```

previous | start | next

previous | start | next

Voice Mail System GUI

- Add speaker, keypads, and microphone panel to frame
- Frame already has `BorderLayout`
- Ch5/mailgui/Telephone.java



```
001: import java.awt.*;
002: import java.awt.event.*;
003: import javax.swing.*;
004:
005: /**
006:    Presents a phone GUI for the voice mail system.
007: */
008: public class Telephone
009: {
010:    /**
011:       Constructs a telephone with a speaker, keypad,
012:       and microphone.
013:    */
014:    public Telephone()
015:    {
016:       JPanel speakerPanel = new JPanel();
017:       speakerPanel.setLayout(new BorderLayout());
018:       speakerPanel.add(new JLabel("Speaker:"),
019:             BorderLayout.NORTH);
020:       speakerField = new JTextArea(10, 25);
021:       speakerPanel.add(speakerField,
022:             BorderLayout.CENTER);
023:
024:       String keyLabels = "123456789*0#";
025:       JPanel keyPanel = new JPanel();
026:       keyPanel.setLayout(new GridLayout(4, 3));
027:       for (int i = 0; i < keyLabels.length(); i++)
028:       {
029:          final String label = keyLabels.substring(i, i + 1);
030:          JButton keyButton = new JButton(label);
031:          keyPanel.add(keyButton);
032:          keyButton.addActionListener(new
033:             ActionListener()
034:             {
035:                public void actionPerformed(ActionEvent event)
036:                {
037:                   connect.dial(label);
038:                }
039:             });
040:       }
041:
042:       final JTextArea microphoneField = new JTextArea(10,25);
043:
044:       JButton speechButton = new JButton("Send speech");
```

```
045:       speechButton.addActionListener(new
046:          ActionListener()
047:          {
048:             public void actionPerformed(ActionEvent event)
049:             {
050:                connect.record(microphoneField.getText());
051:                microphoneField.setText("");
052:             }
053:          });
054:
055:       JButton hangupButton = new JButton("Hangup");
056:       hangupButton.addActionListener(new
057:          ActionListener()
058:          {
059:             public void actionPerformed(ActionEvent event)
060:             {
061:                connect.hangup();
062:             }
063:          });
064:
065:       JPanel buttonPanel = new JPanel();
066:       buttonPanel.add(speechButton);
067:       buttonPanel.add(hangupButton);
068:
069:       JPanel microphonePanel = new JPanel();
070:       microphonePanel.setLayout(new BorderLayout());
071:       microphonePanel.add(new JLabel("Microphone:"),
072:             BorderLayout.NORTH);
073:       microphonePanel.add(microphoneField, BorderLayout.CENTER);
074:       microphonePanel.add(buttonPanel, BorderLayout.SOUTH);
075:
076:       JFrame frame = new JFrame();
077:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
078:       frame.add(speakerPanel, BorderLayout.NORTH);
079:       frame.add(keyPanel, BorderLayout.CENTER);
080:       frame.add(microphonePanel, BorderLayout.SOUTH);
081:
082:       frame.pack();
083:       frame.setVisible(true);
084:    }
085:
086:    /**
087:       Give instructions to the mail system user.
088:    */
```

```
089:    public void speak(String output)
090:    {
091:       speakerField.setText(output);
092:    }
093:
094:    public void run(Connection c)
095:    {
096:       connect = c;
097:    }
098:
099:    private JTextArea speakerField;
100:    private Connection connect;
101: }
```

Custom Layout Manager

- Form layout
- Odd-numbered components right aligned
- Even-numbered components left aligned
- Implement `LayoutManager` interface type

The `LayoutManager` Interface Type

```
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/FormLayoutTester.java
- Note: Can use `GridBagLayout` to achieve the same effect


```
01: import java.awt.*;
02:
03: /**
04:    A layout manager that lays out components along a central axis
05: */
06: public class FormLayout implements LayoutManager
07: {
08:    public Dimension preferredLayoutSize(Container parent)
09:    {
10:       Component[] components = parent.getComponents();
11:       left = 0;
12:       right = 0;
13:       height = 0;
14:       for (int i = 0; i < components.length; i += 2)
15:       {
16:          Component cleft = components[i];
17:          Component cright = components[i + 1];
18:
19:          Dimension dleft = cleft.getPreferredSize();
20:          Dimension dright = cright.getPreferredSize();
21:          left = Math.max(left, dleft.width);
22:          right = Math.max(right, dright.width);
23:          height = height + Math.max(dleft.height,
24:                dright.height);
25:       }
26:       return new Dimension(left + GAP + right, height);
27:    }
28:
29:    public Dimension minimumLayoutSize(Container parent)
30:    {
31:       return preferredLayoutSize(parent);
32:    }
33:
34:    public void layoutContainer(Container parent)
35:    {
36:       preferredLayoutSize(parent); // Sets left, right
37:
38:       Component[] components = parent.getComponents();
39:
40:       Insets insets = parent.getInsets();
41:       int xcenter = insets.left + left;
42:       int y = insets.top;
43:
44:       for (int i = 0; i < components.length; i += 2)
```

```
45:      {
46:         Component cleft = components[i];
47:         Component cright = components[i + 1];
48:
49:         Dimension dleft = cleft.getPreferredSize();
50:         Dimension dright = cright.getPreferredSize();
51:
52:         int height = Math.max(dleft.height, dright.height);
53:
54:         cleft.setBounds(xcenter - dleft.width, y + (height
55:               - dleft.height) / 2, dleft.width, dleft.height);
56:
57:         cright.setBounds(xcenter + GAP, y + (height
58:               - dright.height) / 2, dright.width, dright.height);
59:         y += height;
60:      }
61:   }
62:
63:   public void addLayoutComponent(String name, Component comp)
64:   {}
65:
66:   public void removeLayoutComponent(Component comp)
67:   {}
68:
69:   private int left;
70:   private int right;
71:   private int height;
72:   private static final int GAP = 6;
73: }
```

```
01: import java.awt.*;
02: import javax.swing.*;
03:
04: public class FormLayoutTester
05: {
06:    public static void main(String[] args)
07:    {
08:       JFrame frame = new JFrame();
09:       frame.setLayout(new FormLayout());
10:       frame.add(new JLabel("Name"));
11:       frame.add(new JTextField(15));
12:       frame.add(new JLabel("Address"));
13:       frame.add(new JTextField(20));
14:       frame.add(new JLabel("City"));
15:       frame.add(new JTextField(10));
16:       frame.add(new JLabel("State"));
17:       frame.add(new JTextField(2));
18:       frame.add(new JLabel("ZIP"));
19:       frame.add(new JTextField(5));
20:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21:       frame.pack();
22:       frame.setVisible(true);
23:    }
24: }
25:
26:
27:
```


Strategy Pattern

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

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


Strategy Pattern

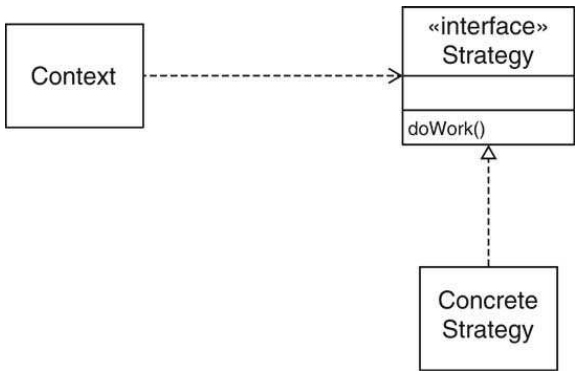
Context

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



Strategy Pattern: Layout Management

Name in Design Pattern	Actual Name (layout management)
Context	Container
Strategy	LayoutManager
ConcreteStrategy	a layout manager such as `BorderLayout`
doWork()	a method such as `layoutContainer`

Strategy Pattern: Sorting

Name in Design Pattern	Actual Name (sorting)
Context	Collections
Strategy	Comparator
ConcreteStrategy	a class that implements `Comparator`
doWork()	`compare`

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

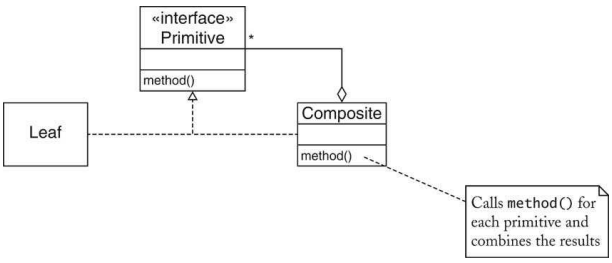
Context

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



Composite Pattern

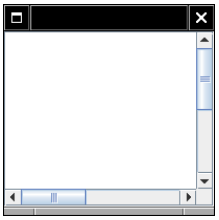
Name in Design Pattern	Actual Name (AWT components)
Primitive	Component
Composite	Container
Leaf	a component without children (e.g. `JButton`)
method()	a method of `Component` (e.g. `getPreferredSize`)

Scroll Bars

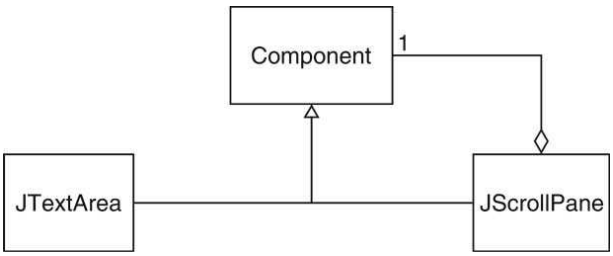
- Scroll bars can be attached to components
- Approach #1: `Component` class can turn on scroll bars
- Approach #2: Scroll bars can surround component

```
JScrollPane pane = new JScrollPane(component);
```

- Swing uses approach #2
- `JScrollPane` is again a component



Scroll Bars



Decorator Pattern

Context

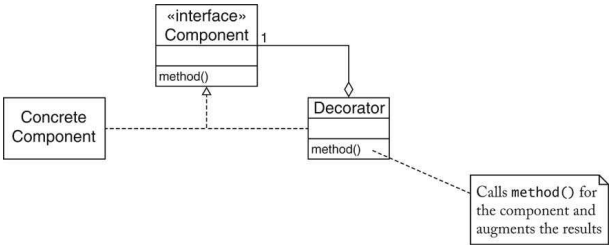
1. Component objects can be decorated (visually or behaviorally enhanced)
2. The decorated object can be used in the same way as the undecorated object
3. The component class does not want to take on the responsibility of the decoration
4. There may be an open-ended set of possible decorations

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



Decorator Pattern: Scroll Bars

Name in Design Pattern	Actual Name (scroll bars)
Component	Component
ConcreteComponent	JTextArea
Decorator	JScrollPane
method()	a method of `Component` (e.g. `paint`)

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

Name in Design Pattern	Actual Name (input streams)
Component	Reader
ConcreteComponent	InputStreamReader
Decorator	BufferedReader
method()	read

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

```
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();
}
```

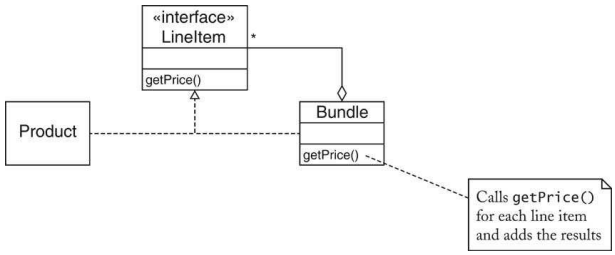
```java
/**
   A product with a price and description.
*/
public class Product implements LineItem
{
   /**
      Constructs a product.
      @param description the description
      @param price the price
   */
   public Product(String description, double price)
   {
      this.description = description;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

/**
   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<LineItem>(); }

   /**
      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;
   }

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

   private ArrayList<LineItem> items;
}
```


Bundles

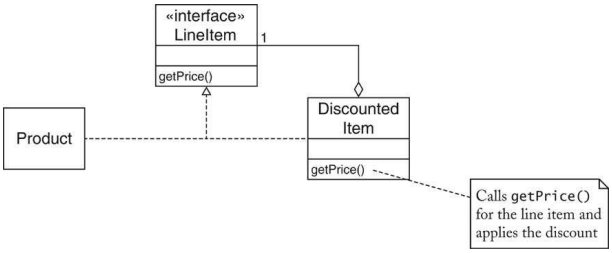


Discounted Items

- 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.
*/
public class DiscountedItem implements LineItem
{
   /**
      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



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

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

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

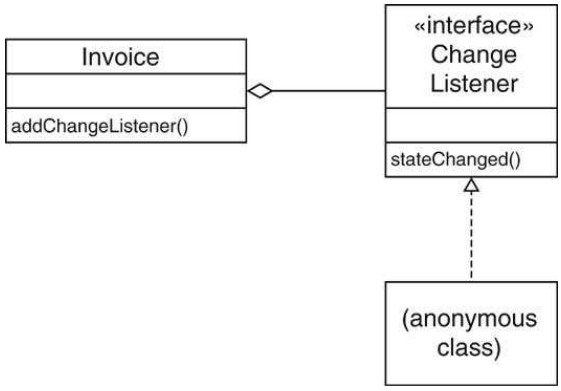
```
ChangeEvent event = new ChangeEvent(this);
for (ChangeListener listener : listeners)
   listener.stateChanged(event);
```


Change Listeners

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

```
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

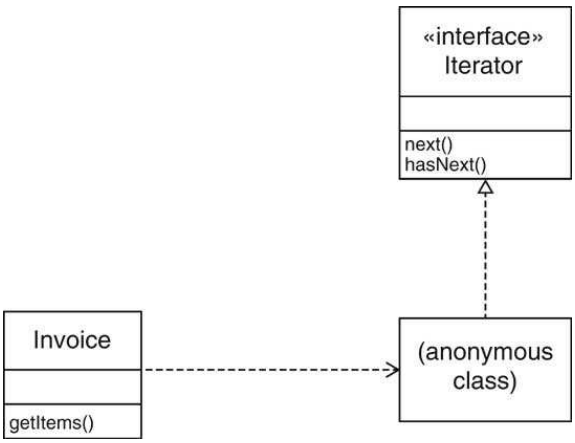
Iterators

- Use standard `Iterator` interface type
```
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
- 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<LineItem>();
      listeners = new ArrayList<ChangeListener>();
   }

  /**
      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 (ChangeListener listener : listeners)
         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<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



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/InvoiceTester.java

```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();
}
```

```java
/**
   A simple invoice formatter.
*/
public class SimpleFormatter implements InvoiceFormatter
{
   public String formatHeader()
   {
      total = 0;
      return "     I N V O I C E\n\n\n";
   }

   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;
}
```

```java
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
      JButton addButton = new JButton("Add");
      addButton.addActionListener(new
         ActionListener()
```

```java
         {
            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

Formatting Invoices