# 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

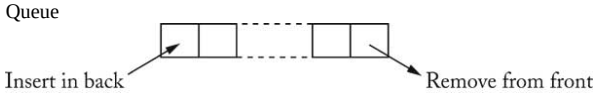
- Traverse links directly

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

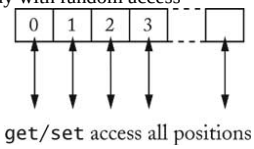
- Exposes implementation
- Error-prone

## High-Level View of Data Structures

- Queue



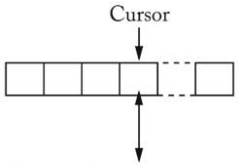

- Array with random access




- List
  ???

previous | start | next .... [Slide 5] ....

## List with Cursor




`get/set/insert/remove` access cursor position

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

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

previous | start | next .... [Slide 6] ....

## 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*

previous | start | next .... [Slide 7] ....

## Short Passages Pattern

previous | start | next .... [Slide 8] ....

## Short Passages Pattern

### Context

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

### Problem

a lengthy description of the problem, including

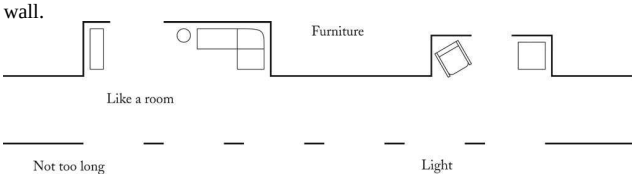
- a depressing picture
- issues of light and furniture
- research about patient anxiety in hospitals
- research that suggests that corridors over 50 ft are considered uncomfortable



## 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

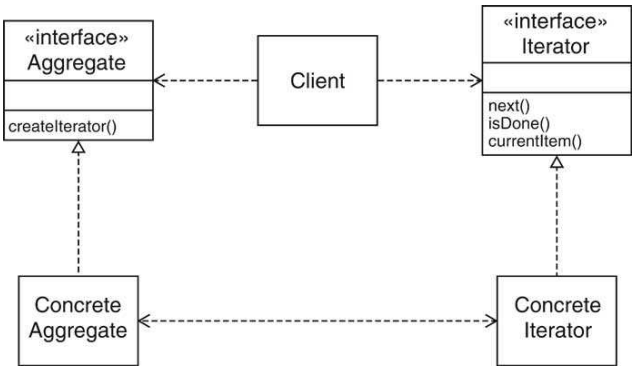
1. An aggregate object contains element objects
2. Clients need access to the element objects
3. The aggregate object should not expose its internal structure
4. Multiple clients may want independent access

### 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



## 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()` |

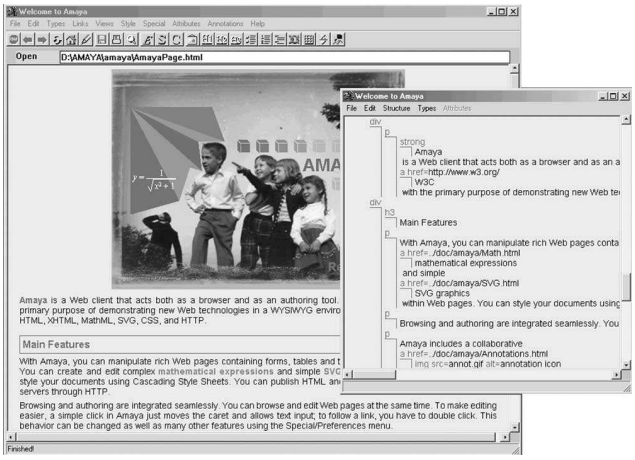


## Model/View/Controller

- 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

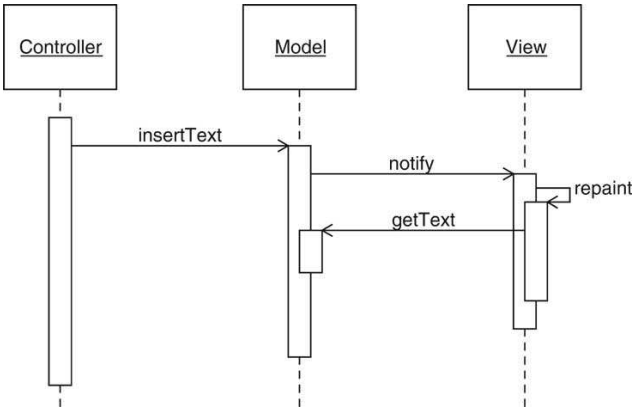





## Model/View/Controller

- 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

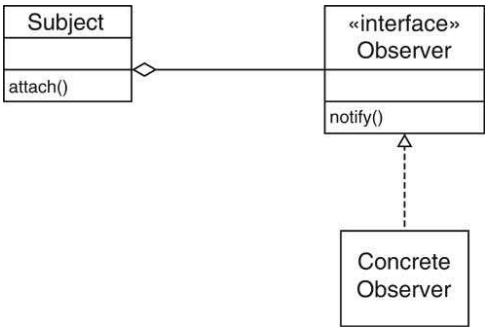
1. An object, called the subject, is source of events
2. One or more observer objects want to be notified when such an event occurs.

### 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.



## 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

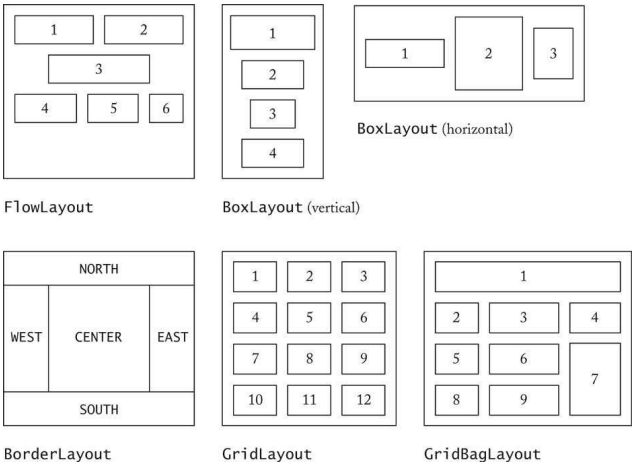


## Layout Managers

- `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

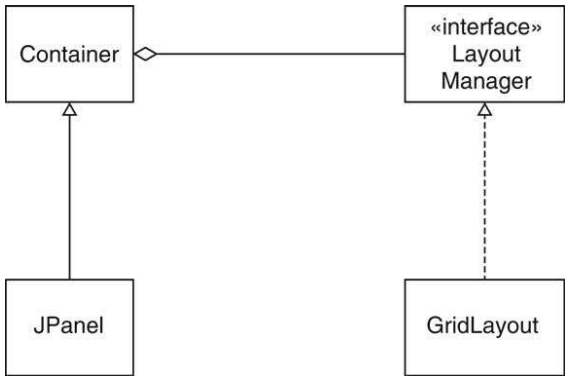


## Layout Managers

- 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

## 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

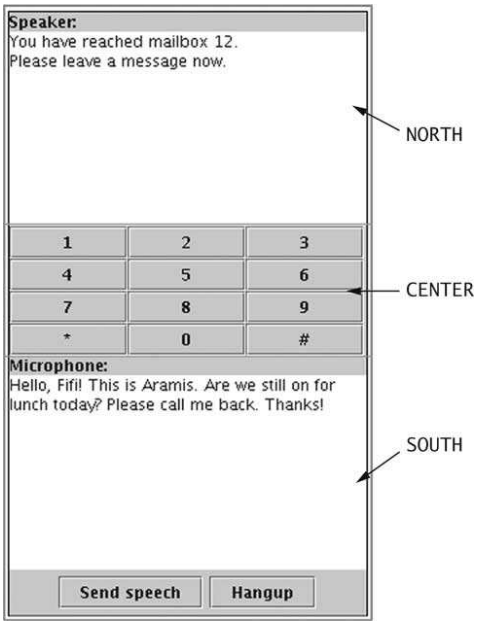
- Panel with `BorderLayout` for speaker

```
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);
```




## Voice Mail System GUI

- Put speaker, keypads, and microphone panel into content pane
- Content pane 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:
017:       JPanel speakerPanel = new JPanel();
018:       speakerPanel.setLayout(new BorderLayout());
019:       speakerPanel.add(new JLabel("Speaker:"),
020:          BorderLayout.NORTH);
021:       speakerField = new JTextArea(10, 25);
022:       speakerPanel.add(speakerField,
023:          BorderLayout.CENTER);
024:
025:       String keyLabels = "123456789*0#";
026:       JPanel keyPanel = new JPanel();
027:       keyPanel.setLayout(new GridLayout(4, 3));
028:       for (int i = 0; i < keyLabels.length(); i++)
029:       {
030:          final String label = keyLabels.substring(i, i + 1);
031:          JButton keyButton = new JButton(label);
032:          keyPanel.add(keyButton);
033:          keyButton.addActionListener(new
034:             ActionListener()
035:             {
036:                public void actionPerformed(ActionEvent event)
037:                {
038:                   connect.dial(label);
039:                }
040:             });
041:       }
```

```
042:
043:       final JTextArea microphoneField = new JTextArea(10, 25);
044:
045:       JButton speechButton = new JButton("Send speech");
046:       speechButton.addActionListener(new
047:          ActionListener()
048:          {
049:             public void actionPerformed(ActionEvent event)
050:             {
051:                connect.record(microphoneField.getText());
052:                microphoneField.setText("");
053:             }
054:          });
055:
056:       JButton hangupButton = new JButton("Hangup");
057:       hangupButton.addActionListener(new
058:          ActionListener()
059:          {
060:             public void actionPerformed(ActionEvent event)
061:             {
062:                connect.hangup();
063:             }
064:          });
065:
066:       JPanel buttonPanel = new JPanel();
067:       buttonPanel.add(speechButton);
068:       buttonPanel.add(hangupButton);
069:
070:       JPanel microphonePanel = new JPanel();
071:       microphonePanel.setLayout(new BorderLayout());
072:       microphonePanel.add(new JLabel("Microphone:"),
073:          BorderLayout.NORTH);
074:       microphonePanel.add(microphoneField,
075:          BorderLayout.CENTER);
076:       microphonePanel.add(buttonPanel,
077:          BorderLayout.SOUTH);
078:
079:       JFrame frame = new JFrame();
080:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
081:       Container contentPane = frame.getContentPane();
082:       contentPane.add(speakerPanel,
```

```
083:          BorderLayout.NORTH);
084:       contentPane.add(keyPanel,
085:          BorderLayout.CENTER);
086:       contentPane.add(microphonePanel,
087:          BorderLayout.SOUTH);
088:
089:       frame.pack();
090:       frame.show();
091:    }
092:
093:    /**
094:       Give instructions to the mail system user.
095:    */
096:    public void speak(String output)
097:    {
098:       speakerField.setText(output);
099:    }
100:
101:    public void run(Connection c)
102:    {
103:       connect = c;
104:    }
105:
106:    private JTextArea speakerField;
107:    private Connection connect;
108: }
```

## 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/FormLayoutTest.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: 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,
53:             dright.height);
54:
55:          cleft.setBounds(xcenter - dleft.width, y + (height -
56:             dleft.height) / 2, dleft.width, dleft.height);
57:
58:          cright.setBounds(xcenter + GAP, y + (height
59:             - dright.height) / 2, dright.width, dright.height);
60:          y += height;
61:       }
62:    }
63:
64:    public void addLayoutComponent(String name,
65:       Component comp)
66:    {}
67:
68:    public void removeLayoutComponent(Component comp)
69:    {}
70:
71:    private int left;
72:    private int right;
73:    private int height;
74:    private static final int GAP = 6;
75: }
```

```
01: import java.awt.*;
02: import javax.swing.*;
03:
04: public class FormLayoutTest
05: {
06:    public static void main(String[] args)
07:    {
08:       JFrame frame = new JFrame();
09:       Container contentPane = frame.getContentPane();
10:       contentPane.setLayout(new FormLayout());
11:       contentPane.add(new JLabel("Name"));
12:       contentPane.add(new JTextField(15));
13:       contentPane.add(new JLabel("Address"));
14:       contentPane.add(new JTextField(20));
15:       contentPane.add(new JLabel("City"));
16:       contentPane.add(new JTextField(10));
17:       contentPane.add(new JLabel("State"));
18:       contentPane.add(new JTextField(2));
19:       contentPane.add(new JLabel("ZIP"));
20:       contentPane.add(new JTextField(5));
21:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22:       frame.pack();
23:       frame.show();
24:    }
25: }
26:
27:
28:
```

## Strategy Pattern

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

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



## Strategy Pattern

### Context

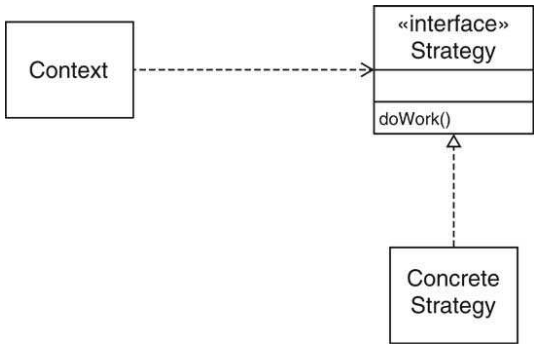
1. A class can benefit from different variants for an algorithm
2. Clients sometimes want to replace standard algorithms with custom versions

### 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



## 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

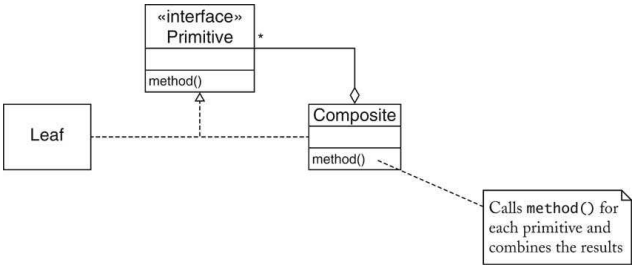
### Context

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`) |

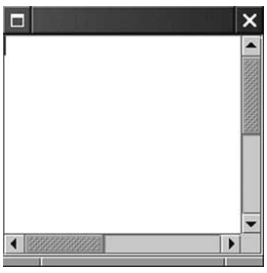


## Scroll Bars

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

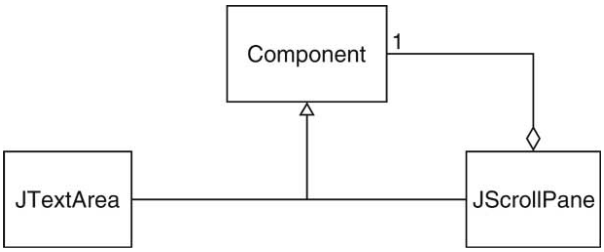
```
JScrollPane pane = new JScrollPane(component);
```

- 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



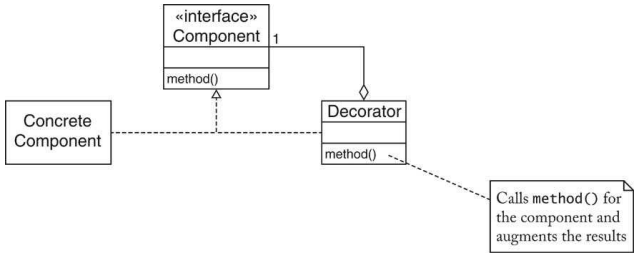
# Decorator Pattern

## 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"

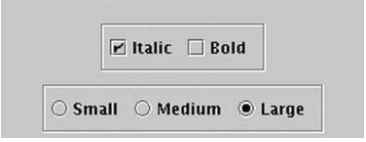


## Litmus Test

- 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



```
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: }
```

```
01: /**
02:    A product with a price and description.
03: */
04: public class Product implements LineItem
05: {
06:    /**
07:       Constructs a product.
08:       @param description the description
09:       @param price the price
10:    */
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: }
```

## Bundles

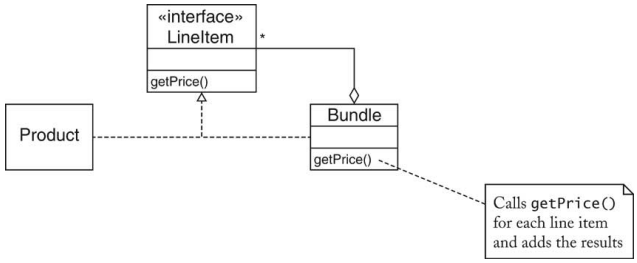
- Bundle = set of related items with description+price
- E.g. stereo system with tuner, amplifier, CD player + speakers
- A bundle has line items
- A bundle is a line item
- COMPOSITE pattern
- Ch5/invoice/Bundle.java (look at `getPrice`)



```
01: import java.util.*;
02:
03: /**
04:    A bundle of items that is again an item.
05: */
06: public class Bundle implements LineItem
07: {
08:    /**
09:       Constructs a bundle with no items.
10:    */
11:    public Bundle() { items = new ArrayList(); }
12:
13:    /**
14:       Adds an item to the bundle.
15:       @param item the item to add
16:    */
17:    public void add(LineItem item) { items.add(item); }
18:
19:    public double getPrice()
20:    {
21:       double price = 0;
22:       for (int i = 0; i < items.size(); i++)
23:       {
24:          LineItem item = (LineItem) items.get(i);
25:          price += item.getPrice();
26:       }
27:       return price;
28:    }
29:
30:    public String toString()
31:    {
32:       String description = "Bundle: ";
33:       for (int i = 0; i < items.size(); i++)
34:       {
35:          if (i > 0) description += ", ";
36:          LineItem item = (LineItem) items.get(i);
37:          description += item.toString();
38:       }
39:       return description;
```

```
40:    }
41:
42:    private ArrayList items;
43: }
```

## Bundles



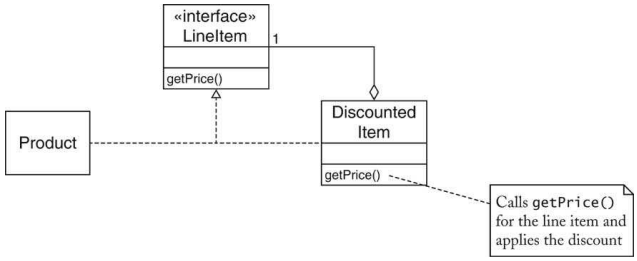
## Discounted Items

- 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`



```
01: /**
02:    A decorator for an item that applies a discount.
03: */
04: public class DiscountedItem implements LineItem
05: {
06:    /**
07:       Constructs a discounted item.
08:       @param item the item to be discounted
09:       @param discount the discount percentage
10:    */
11:    public DiscountedItem(LineItem item, double discount)
12:    {
13:       this.item = item;
14:       this.discount = discount;
15:    }
16:
17:    public double getPrice()
18:    {
19:       return item.getPrice() * (1 - discount / 100);
20:    }
21:
22:    public String toString()
23:    {
24:       return item.toString() + " (Discount " + discount
25:          + "%)";
26:    }
27:
28:    private LineItem item;
29:    private double discount;
30: }
```

## 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 (int i = 0; i < listeners.size(); i++)
  {
    ChangeListener listener = (ChangeListener)listeners.get(i);
    listener.stateChanged(event);
  }
```



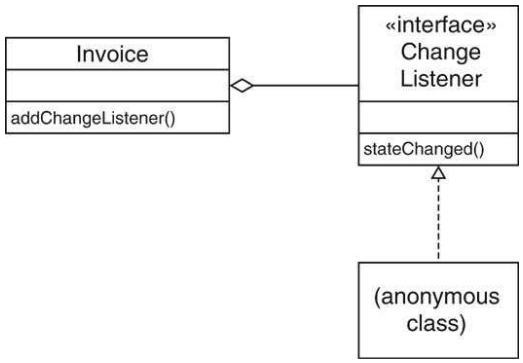
## Change Listeners

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

```
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
{
boolean hasNext();
Object next();
void remove();
}
```

- `remove` is "optional operation" (see ch. 8)
- implement to throw `UnsupportedException`
- implement `hasNext/next` manually to show inner workings
- Ch5/invoice/Invoice.java

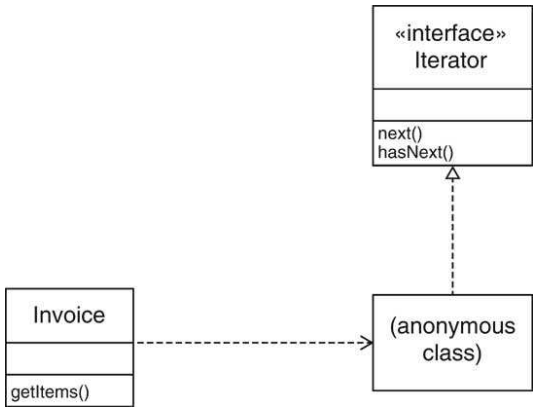


```
01: import java.util.*;
02: import javax.swing.event.*;
03:
04: /**
05:    An invoice for a sale, consisting of line items.
06: */
07: public class Invoice
08: {
09:    /**
10:       Constructs a blank invoice.
11:    */
12:    public Invoice()
13:    {
14:       items = new ArrayList();
15:       listeners = new ArrayList();
16:    }
17:
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 (int i = 0; i < listeners.size(); i++)
28:       {
29:          ChangeListener listener
30:             = (ChangeListener) listeners.get(i);
31:          listener.stateChanged(event);
32:       }
33:    }
34:
35:    /**
36:       Adds a change listener to the invoice.
37:       @param listener the change listener to add
38:    */
39:    public void addChangeListener(ChangeListener listener)
40:    {
41:       listeners.add(listener);
```

```
42:    }
43:
44:    /**
45:       Gets an iterator that iterates through the items.
46:       @return an iterator for the items
47:    */
48:    public Iterator getItems()
49:    {
50:       return new
51:          Iterator()
52:          {
53:             public boolean hasNext()
54:             {
55:                return current < items.size();
56:             }
57:             public Object next()
58:             {
59:                Object r = items.get(current);
60:                current++;
61:                return r;
62:             }
63:             public void remove()
64:             {
65:                throw new UnsupportedOperationException();
66:             }
67:             private int current = 0;
68:          };
69:    }
70:
71:    public String format(InvoiceFormatter formatter)
72:    {
73:       String r = formatter.formatHeader();
74:       Iterator iter = getItems();
75:       while (iter.hasNext())
76:       {
77:          LineItem item = (LineItem) iter.next();
78:          r += formatter.formatLineItem(item);
79:       }
80:       return r + formatter.formatFooter();
81:    }
```

```
82:
83:    private ArrayList items;
84:    private ArrayList listeners;
85: }
```

## 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/InvoiceTest.java



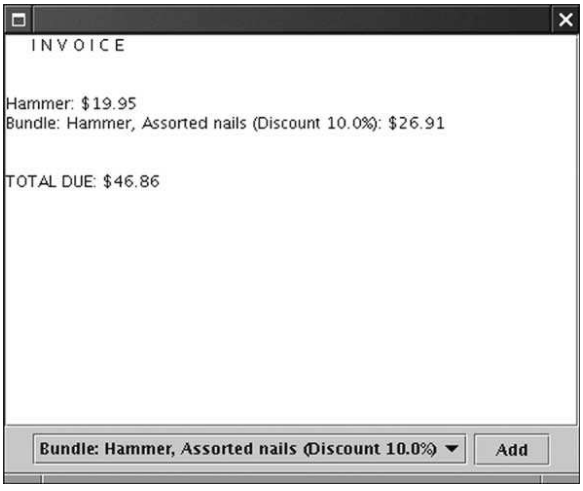
```
01: /**
02:    This interface describes the tasks that an invoice
03:    formatter needs to carry out.
04: */
05: public interface InvoiceFormatter
06: {
07:    /**
08:       Formats the header of the invoice.
09:       @return the invoice header
10:    */
11:    String formatHeader();
12:    /**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

```
01: /**
02:    A simple invoice formatter.
03: */
04: public class SimpleFormatter implements InvoiceFormatter
05: {
06:    public String formatHeader()
07:    {
08:       total = 0;
09:       return "     I N V O I C E\n\n\n";
10:    }
11:
12:    public String formatLineItem(LineItem item)
13:    {
14:       total += item.getPrice();
15:       return item.toString() + ": $"
16:          + item.getPrice() + "\n";
17:    }
18:
19:    public String formatFooter()
20:    {
21:       return "\n\nTOTAL DUE: $" + total + "\n";
22:    }
23:
24:    private double total;
25: }
26:
```

```
01: import java.awt.*;
02: import java.awt.event.*;
03: import javax.swing.*;
04: import javax.swing.event.*;
05:
06: /**
07:    A program that tests the invoice classes.
08: */
09: public class InvoiceTest
10: {
11:    public static void main(String[] args)
12:    {
13:       final Invoice invoice = new Invoice();
14:       final InvoiceFormatter formatter = new SimpleFormatter();
15:
16:       // this text area will contain the formatted invoice
17:       final JTextArea textArea = new JTextArea(20, 40);
18:
19:       // when the invoice changes, update the text area
20:       ChangeListener listener = new
21:          ChangeListener()
22:          {
23:             public void stateChanged(ChangeEvent event)
24:             {
25:                textArea.setText(invoice.format(formatter));
26:             }
27:          };
28:       invoice.addChangeListener(listener);
29:
30:       // add line items to a combo box
31:       final JComboBox combo = new JComboBox();
32:       Product hammer = new Product("Hammer", 19.95);
33:       Product nails = new Product("Assorted nails", 9.95);
34:       combo.addItem(hammer);
35:       Bundle bundle = new Bundle();
36:       bundle.add(hammer);
37:       bundle.add(nails);
38:       combo.addItem(new DiscountedItem(bundle, 10));
39:
40:       // make a button for adding the currently selected
41:       // item to the invoice
```

```
42:       JButton addButton = new JButton("Add");
43:       addButton.addActionListener(new
44:          ActionListener()
45:          {
46:             public void actionPerformed(ActionEvent event)
47:             {
48:                LineItem item = (LineItem) combo.getSelectedItem();
49:                invoice.addItem(item);
50:             }
51:          });
52:
53:       // put the combo box and the add button into a panel
54:       JPanel panel = new JPanel();
55:       panel.add(combo);
56:       panel.add(addButton);
57:
58:       // add the text area and panel to the content pane
59:       JFrame frame = new JFrame();
60:       Container contentPane = frame.getContentPane();
61:       contentPane.add(new JScrollPane(textArea),
62:          BorderLayout.CENTER);
63:       contentPane.add(panel, BorderLayout.SOUTH);
64:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
65:       frame.pack();
66:       frame.show();
67:    }
68: }
```

## Formatting Invoices



## Formatting Invoices