# Chapter 5: Patterns and GUI Programming

## Chapter Topics

- The Pattern Concept
- The ITERATOR Pattern
- 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

## 5.1 What is a Design Pattern ?

- Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
- A solution in terms of objects and interfaces to a problem in a context.
- Four essential elements of a pattern:
  - A short *name*
  - A brief description of the *context*
  - A lengthy description of the *problem*
  - A prescription for a *solution*

## 5.2 Design patterns

- The ITERATOR pattern
- The OBSERVER pattern
- The STRATEGY pattern
- The COMPOSITE pattern
- The DECORATOR pattern

Note: Some of the contents presented in the section 5.2 are directly quoted from "Design Patterns" by Gamma, Helm, Johnson, and Vlissides.

5.2.1 The Iterator Pattern

**Intent**

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

## Motivation

```
public class SomeCollection<E>
{   private ArrayList<E> data;
    private int count;
    public boolean hasNext() { return count < data.size(); }
    public E next()
    {  return data.get(count++); }
}
```

[Q] What would be the problem of this approach ?

The problem can be solved by having the responsibility for access and traversal out of the collection object and put it into an iterator object. The Iterator class defines an interface for accessing the elements of the collection. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

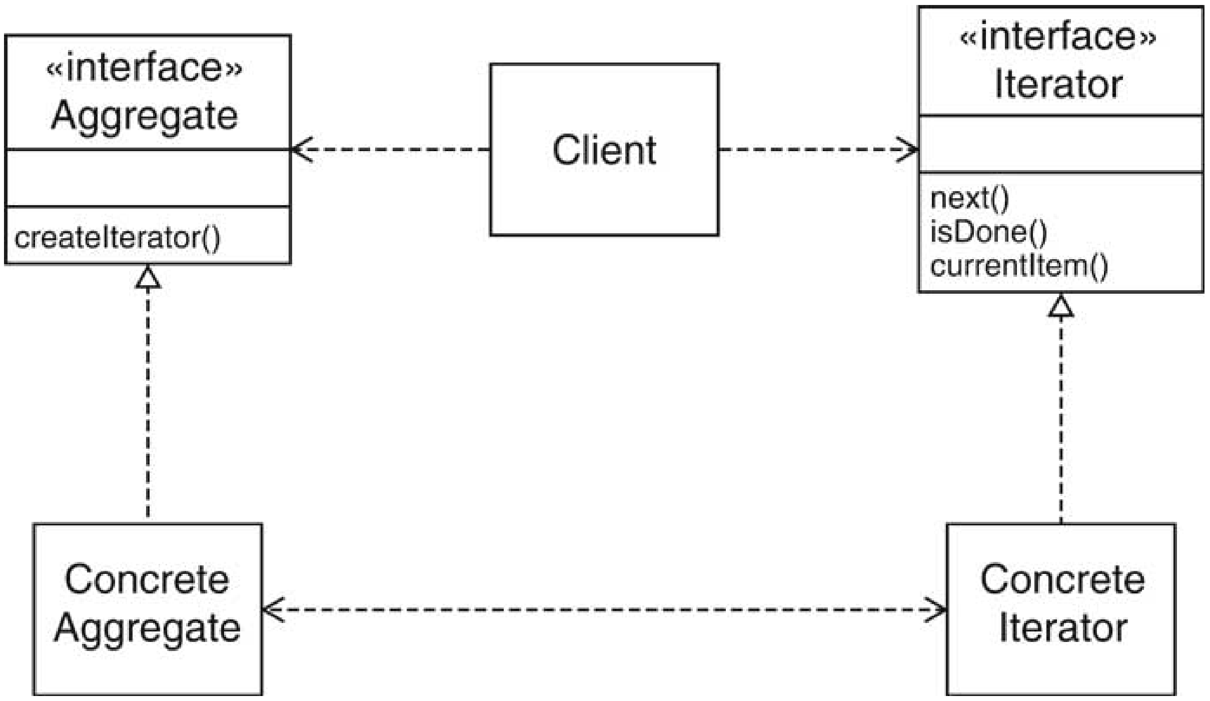
## Context

1. An object (which we will call the aggregate) contains other objects (which we'll call elements).
2. Clients (that is, methods that use the aggregate) need to access to the elements.
3. The aggregate should not expose its internal structure.
4. There may be multiple clients that need simultaneous 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 (*polymorphic iterator*).

## Structure

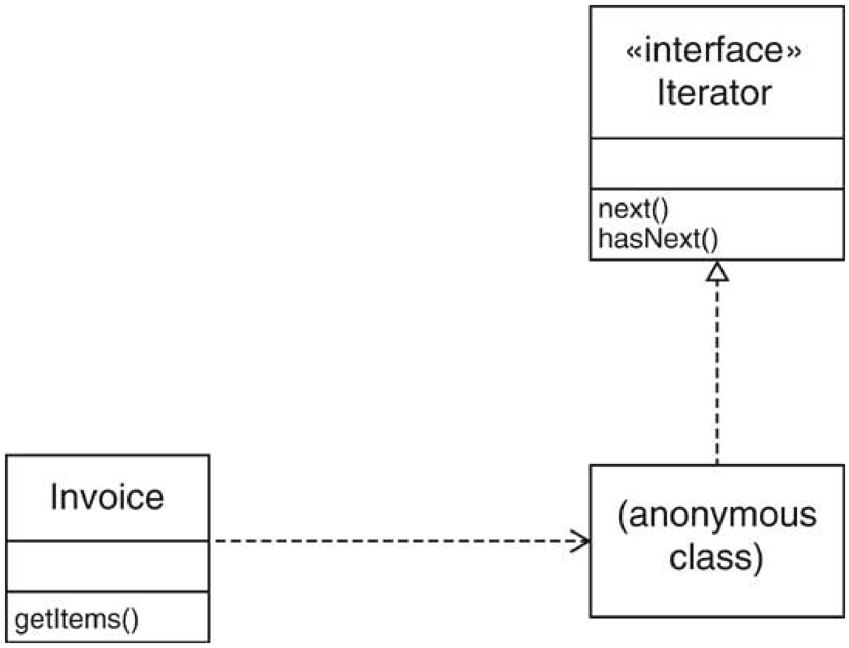


- **Iterator**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate**
  - defines an interface for creating an Iterator object.
- **ConcreteAggregate**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

**<u>Example</u>**

| **Name in Design Pattern** | **Actual Name** |
|---|---|
| `Aggregate` | `List` |
| `ConcreteAggregate` | `LinkedList` |
| `Iterator` | `ListIterator` |
| `ConcreteIterator` | `anonymous class implementing ListIterator` |
| `createIterator()` | `listIterator()` |
| `next()` | `next()` |
| `isDone()` | `opposite of hasNext()` |
| `currentItem()` | `return value of next()` |

**Example (Case study from the textbook)**



```
public class Invoice
{   private ArrayList<LineItem> items;

    public Invoice()
   {   items = new ArrayList<LineItem>();   }

    public Iterator<LineItem> getItems()
    {   return new
                Iterator<LineItem>()
                {   private int current = 0;
                    public boolean hasNext() {   return current < items.size(); }
                    public LineItem next() {   return items.get(current++); }
                    public void remove(){ throw new UnsupportedOperationException(); }
                 };
    } //getItems
}// Invoice
```

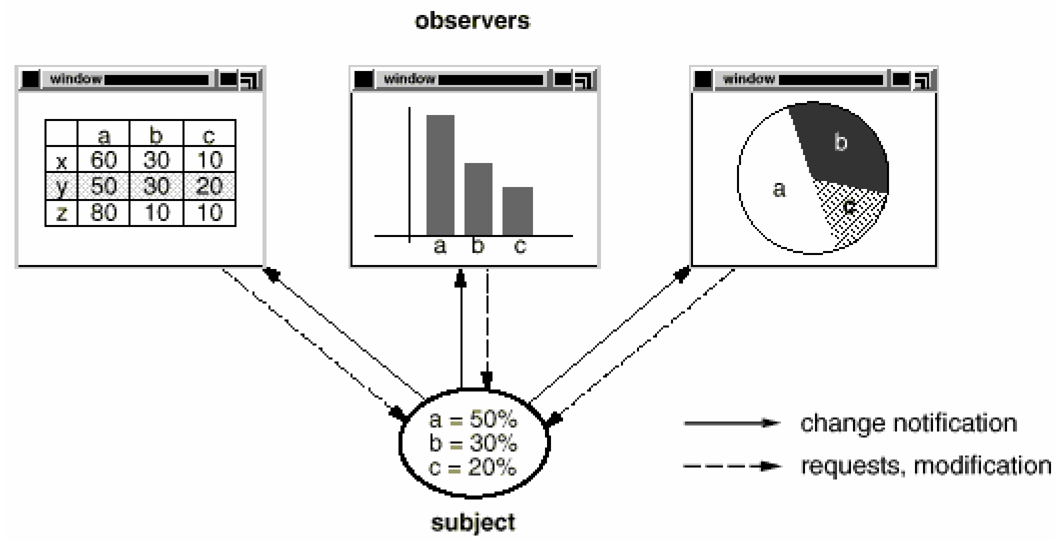
## 5.2 The Observer Pattern

**Intent**

Define one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

## Motivation

Classes defining application data and presentations can be reused independently.



For example, in the figure above, the spreadsheet, bar chart, and pie chart objects can depict the same application data. They don't know about each other, thereby letting you reuse only the one you need. When the user changes the data, all the presentations reflect the change immediately.

The key objects in this pattern are **subject** (the data object) and **observer** (the spreadsheet and bar chart). A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

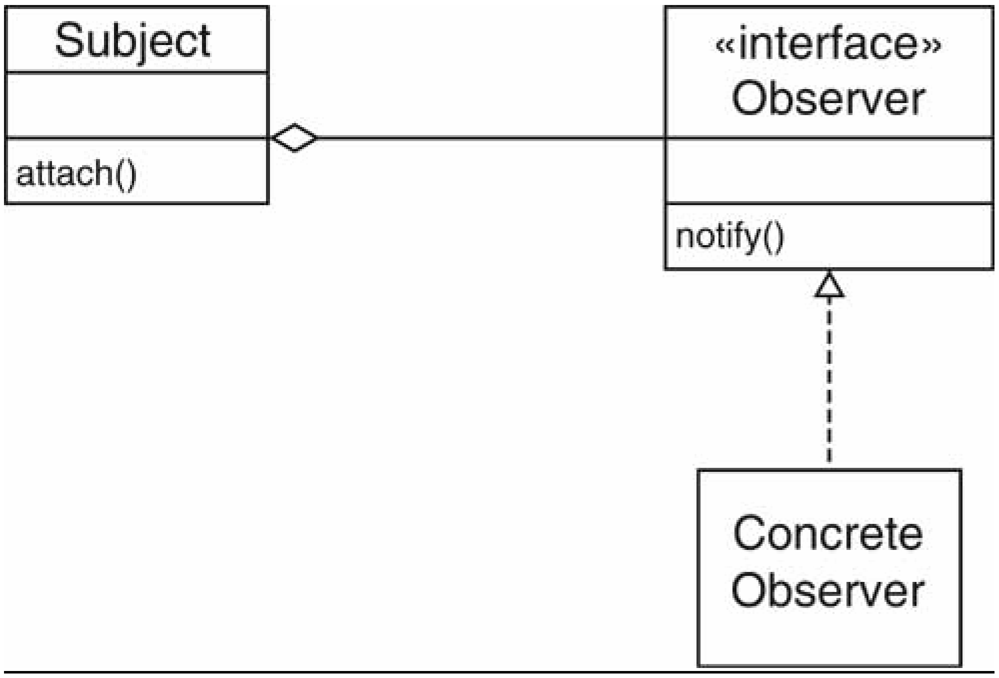
## Context

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

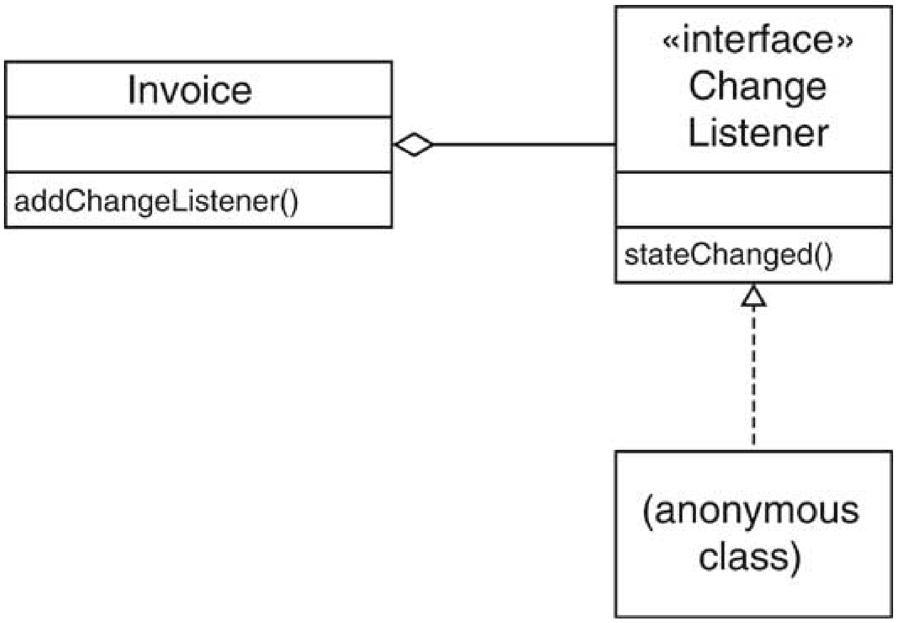
1. Define an observer interface type. All concrete observers implement it.
2. The subject maintains a collection of observers.
3. The subject supplies methods for attaching and detaching observers.
4. Whenever an event occurs, the subject notifies all observers.

## Structure



- **Subject**
    - knows its observers. Any number of Observer objects may observe a subject.
    - provides an interface for attaching and detaching Observer objects.
- **Observer**
    - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteObserver**
    - maintains a reference to a ConcreteSubject object.
    - stores state that should stay consistent with the subject's.
    - implements the Observer and updates the graphical interface to keep its state consistent with the subject's.

### Example (Case study from the textbook)



<u>The  Model/Viewer/Controller architecture</u>

- The Smalltalk Model/Viewer/Controller (MVC) architecture is the first and perhaps best-known example of the Observer pattern.
  - Model → Subject
  - Views → Observers
  - Controllers

(a) Model

- The model is the code that carries out some task. It is built with no necessary concern for how it will "look and feel" when presented to the user. It has a purely functional interface, meaning that it has a set of public functions that can be used to achieve all of its functionality.
- However, a model must be able to "register" views and it must be able to "notify" all of its registered views when any of its functions cause its state to be changed.

(b) Many Views

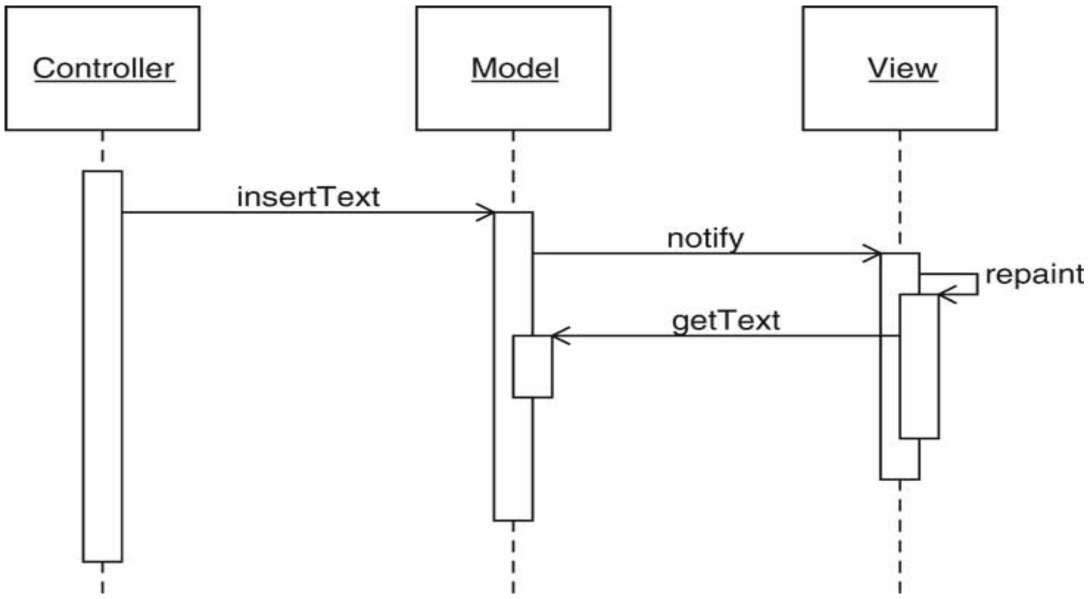
- A model in MVC can have several views. A view provides graphical user interface (GUI) components for a model. It gets the values that it displays by querying the model of which it is a view.
- In Java the views are built of AWT or SWING components.

(c) Many Controllers

- The controller can call mutator methods of the model to get it to update its state. Of course, then the model will notify ALL registered views that a change has been made and so they will update what they display to the user as appropriate.
- In Java the controllers are the listeners in the Java event structure.

(d) Interaction among Model, View and Controller

- Controllers update model
- Model tells views that data has changed
- Views redraw themselves



```
public class Invoice
{
      private ArrayList<LineItem> items;
      private ArrayList<ChangeListener> listeners;
      public Invoice()
      {
           items = new ArrayList<LineItem>();
           listeners = new ArrayList<ChangeListener>();
      }
   /**
      Adds a change listener to the invoice.
      @param listener the change listener to add
   */
   public void addChangeListener(ChangeListener listener)
   {    listeners.add(listener); }

   /* Accessor that returns the invoice in a formatted string.
   public String format(InvoiceFormatter formatter)
   { // will be covered in the strategy pattern part. String r = 
formatter.formatHeader();
      Iterator<LineItem>iter = getItems();
      while (iter.hasNext())
         r += formatter.formatLineItem(iter.next());
      return r + formatter.formatFooter();

   }

   /**
      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);
   }

}
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

      // 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
     // Add the text area and panel to the frame
   }
}
```

## 5.3 The Strategy Pattern

### Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

### Motivation

Many algorithms exist for the layout management of GUI components. It is not desirable to hard-wire all such algorithms into the classes that require them  for several reasons:

- Clients that need layout management get more complex if they include the layout management code. That makes clients bigger and harder to maintain, especially if they support multiple layout management algorithms.
- Different algorithms will be appropriate at different times. We don't want to support multiple layout management algorithms if we don't use them all.
- It's difficult to add new algorithms and vary existing ones when layout management is an integral part of a client.

We can avoid these problems by defining classes that encapsulate different layout management algorithms. An algorithm that's encapsulated in this way is called a **strategy**.

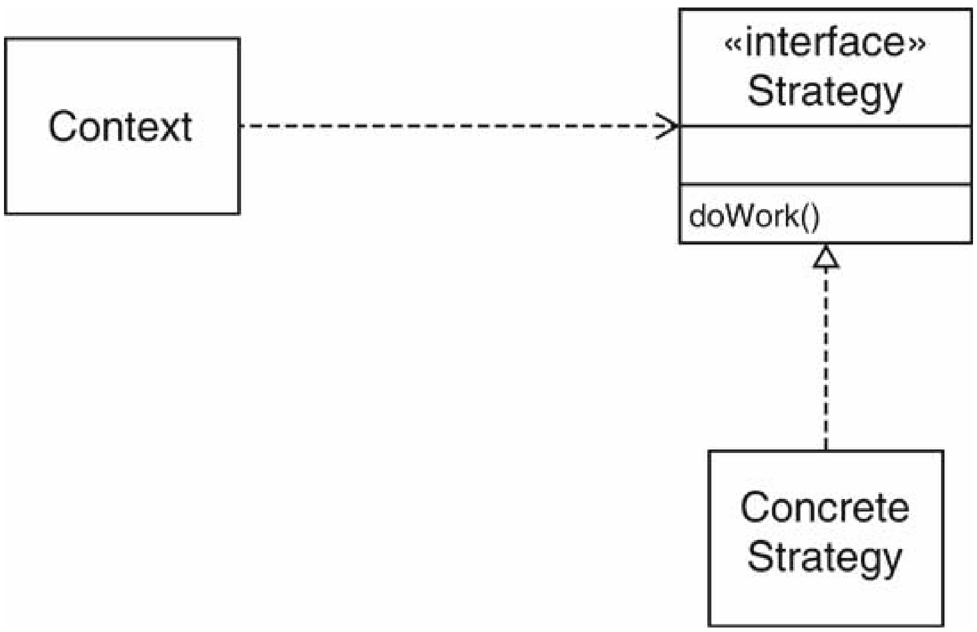
### Context

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

## Solution

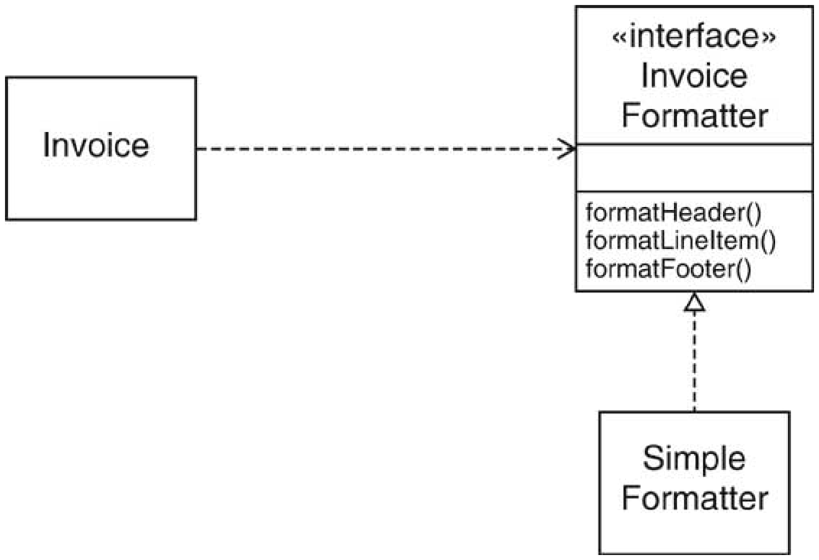
1. Define an interface type that is an abstraction for the algorithm. We will call this interface type the *strategy*.
2. Concrete strategy classes realize this interface type.
3. The client supplies a concrete strategy object to the context class.
4. Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object

## Structure



- **Strategy**
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy**
  - implements the algorithm using the Strategy interface.
- **Context**
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

## Example (Case study from the textbook)



<u>Strategy</u>

```
public interface InvoiceFormatter
{
  String formatHeader();
  String formatLineItem(LineItem item);
  String formatFooter();
}
```

<u>Concrete Strategy</u>

```
public class SimpleFormatter implements InvoiceFormatter
{
  private double total;

  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)); }
}
```

<u>Context</u>

```
public class Invoice
```

```
{  …
  public String format(InvoiceFormatter formatter)
   {
      String r = formatter.formatHeader();
      Iterator<LineItem>iter = getItems(); // returns an iterator of this invoice.
      while (iter.hasNext())
         r += formatter.formatLineItem(iter.next());
      return r + formatter.formatFooter();
   }
…
}
```

<u>Client</u>

```
public class InvoiceTester
{
   public static void main(String[] args)
   {
      final Invoice invoice = new Invoice();
      final InvoiceFormatter formatter = new SimpleFormatter(); // instance of concrete strategy

      // 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)); // the client provides the context with the
concrete strategy object.
            }
         };
      invoice.addChangeListener(listener);
. . .

}
```

## Example: Layout Management

1. Terminologies

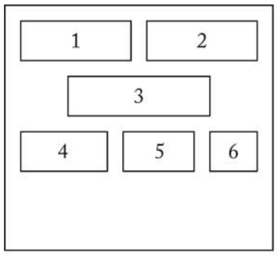
- User interfaces made up of *components* such as Buttons, TextArea, etc.

- Components placed in *containers* such as Frames, Panels, etc.

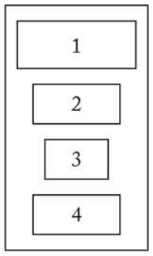
2. Why Layout Managers ?
- Absolute positioning, with which the programmer specifies pixel positions for each GUI component, is not a good idea because the component size can change from the original design.
- The component size can change from the original design, usually for the following reasons:
  - o The Swing library users can select a different "look and feel" such as Window look and feel or Metal look and feel, etc. The same component, such as button, will have a different size in a different look and feel.
  - o The program gets translated into a different language.

3. Built-in Layout Managers

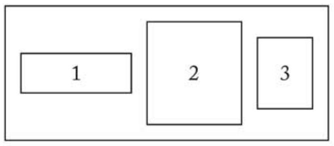
- FlowLayout: left to right, start new row when full
- BoxLayout: left to right or top to bottom
- BorderLayout: 5 areas, Center, North, South, East, West
- GridLayout: grid, all components have same size
- GridBagLayout: complex, like HTML table



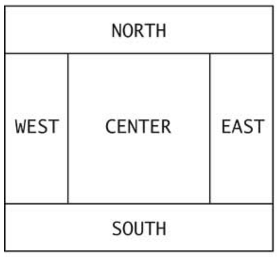
FlowLayout



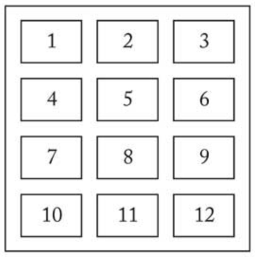
BoxLayout (vertical)



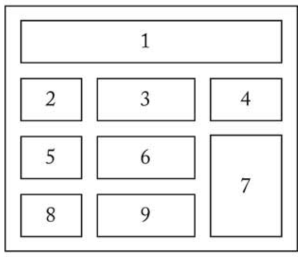
BoxLayout (horizontal)



BorderLayout



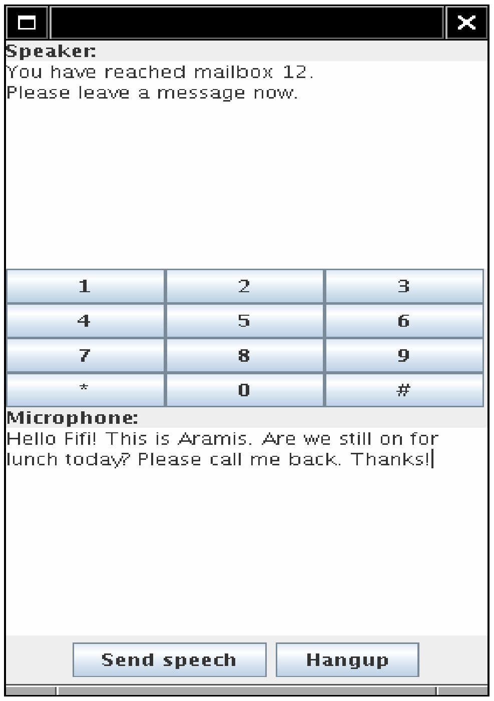
GridLayout



GridBagLayout

4. The Use of Layout Managers in Voice Mail System GUI

- Ch5/mailgui/Telephone.java
- A Voice Mail System consists of a Speaker, a Keypad, and a Microphone.


Speaker:
You have reached mailbox 12.
Please leave a message now.
1
2
3
4
5
6
7
8
9
*
0
#
Microphone:
Hello Fifi! This is Aramis. Are we still on for lunch today? Please call me back. Thanks!
Send speech
Hangup

- Each part of the VMS consists of various GUI components.
  - Speaker: Label and TextArea
  - KeyPad: Buttons
- Microphone: Lable, TextArea, and two ButtonsThe related GUI components will be stored in a Panel and a desired layout manager, which controls the component arrangement, is registered with the Panel.

**Speaker**

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

**Keypad**

```
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);
   ...
}
```

**Microphone**

```
final JTextArea microphoneField = new JTextArea(10,25);

JPanel buttonPanel = new JPanel();// flowlayout is default
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);
```

Note that each areas of a border layout can only hold a single component. Therefore, two buttons are placed inside another panel and that panel is added to the SOUTH area of the microphone panel.

**Putting them together**

Now, three panels, speakerPanel, keyPanel, microphonePanel, are ready to be placed in a frame. The default layout manager for a frame is a border layout manager.

```
JFrame frame = new JFrame(); // border layout is default
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.add(speakerPanel, BorderLayout.NORTH);
frame.add(keyPanel, BorderLayout.CENTER);
frame.add(microphonePanel, BorderLayout.SOUTH);
```

## 5. Where is a Strategy pattern being used in this example ?

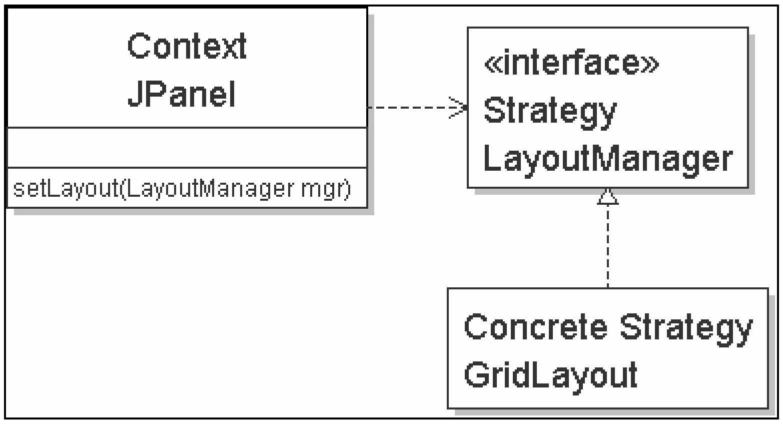
The `setLayout` method of a container is written in terms of a general `LayoutManager` interface, not in terms of a specific layout manager such as `GridLayout` or `Border Layout`. Therefore, the setLayout manager can take any specific layout manager as long as it is a LayoutManage (pluggable strategy for layout management). A specific algorithm to arrange components is described in a specific layout manager who is responsible for executing concrete strategy.

```
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);
}
```

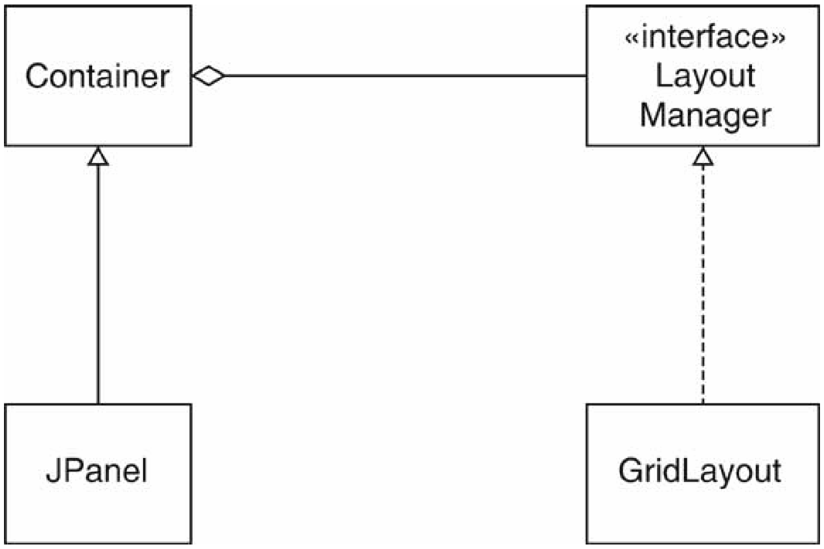
`class JPanel`

`public void setLayout(LayoutManager mgr)`

| **Name in Design Pattern** | **Actual Name (layout management)** |
|---|---|
| `Context` | `Container such as JPanel` |
| `Strategy` | `LayoutManager` |
| `ConcreteStrategy` | `a layout manager such as GridLayout` |
| `doWork()` | `method such as layoutContainer` |



The following class diagram is more implementation specific.



## 5.5 The Composite Pattern

**Intent**

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

## Motivation

- The user can group components to form larger components, which in turn can be grouped to form still larger components. For example, Text and Lines are graphical primitives and a Picture is a container for these primitives.
- Both primitives and container objects should be treated identically, otherwise, the implementation will be complicated.
- The Composite pattern describes how to use recursive composition so that clients don't have to make the distinction between primitives and container objects.
- The key idea is the use of an interface that represents *both* primitives and their containers. For example, any operation declared in this interface will be the common operation among component objects.

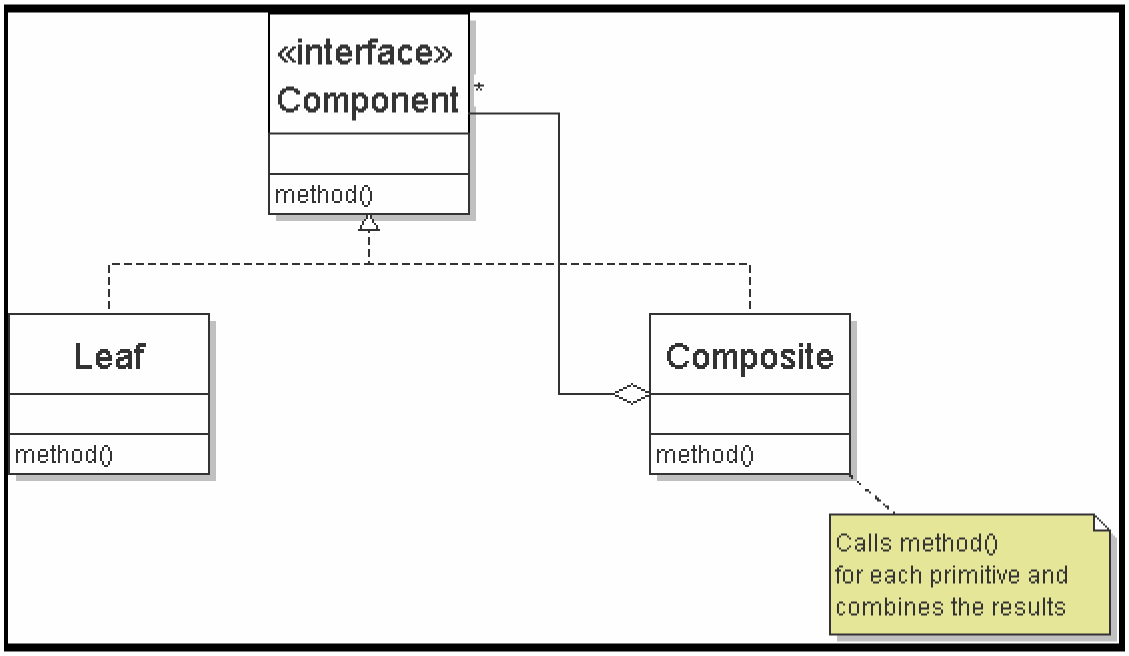
## Context

1. Primitive objects can be combined to composite objects
2. Clients treat a composite object and a primitive object in the same way

## Solution

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

## Structure



**Component**

- declares the interface for component objects in the composition.

**Leaf**

- represents leaf objects in the composition. A leaf has no children.
- defines behavior for primitive objects in the composition.

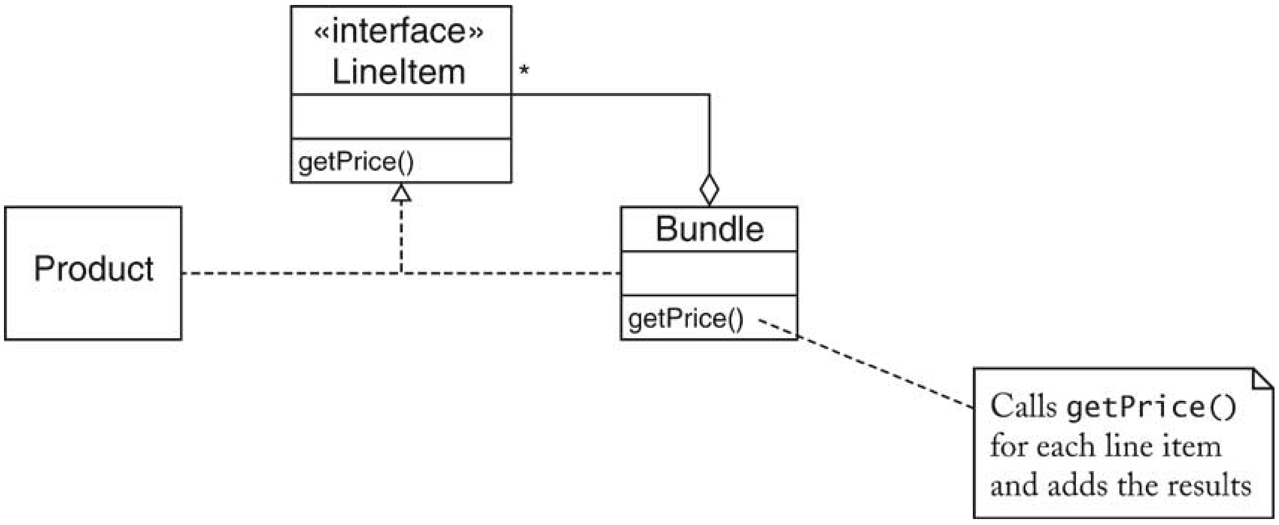
**Composite**

- defines behavior for components having children.
- stores child components.
- implements child-related operations in the Component interface.

**Example:**

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

**Example (Case study from the textbook)**



```
public interface LineItem
{
  double getPrice();
  String toString();
}

public class Product implements LineItem
{   private String description;
    private double price;

    public Product(String description, double price)
    {  this.description = description;
       this.price = price;
    }
     public double getPrice() { return price; }
     public String toString() { return description; }
}
public class Bundle implements LineItem
{    private ArrayList<LineItem> items;

     public Bundle() { items = new ArrayList<LineItem>(); }
     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;
  }
}
```

## 5.6 Decorator Pattern

### Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

### Motivation

- We want to add responsibilities to individual objects, not to an entire class. For example, you want to add properties like borders or behaviors like scrolling to any user interface component such as a text area.
- One way is to make the text area responsible for supplying scroll bars. For example, java.awt.TextArea does that. One of its constructor public TextArea (String text, int rows, int columns, int scrollbars) allows you to set the scroll bar option. However, this is not an effective way because of the following reasons:
  - It would be wasteful if every component such as text areas, panels should have to supply an option for scroll bars.
  - There can be many different sets of decorations, and the component classes cannot anticipate all of them.
- Better design approach is the use of decorators: To enclose the component in another object that adds the border. The enclosing object is called a **decorator**. The important point is that the decorator is also a component so that all of the functionality of the component class still applies to the decorator.

### Context

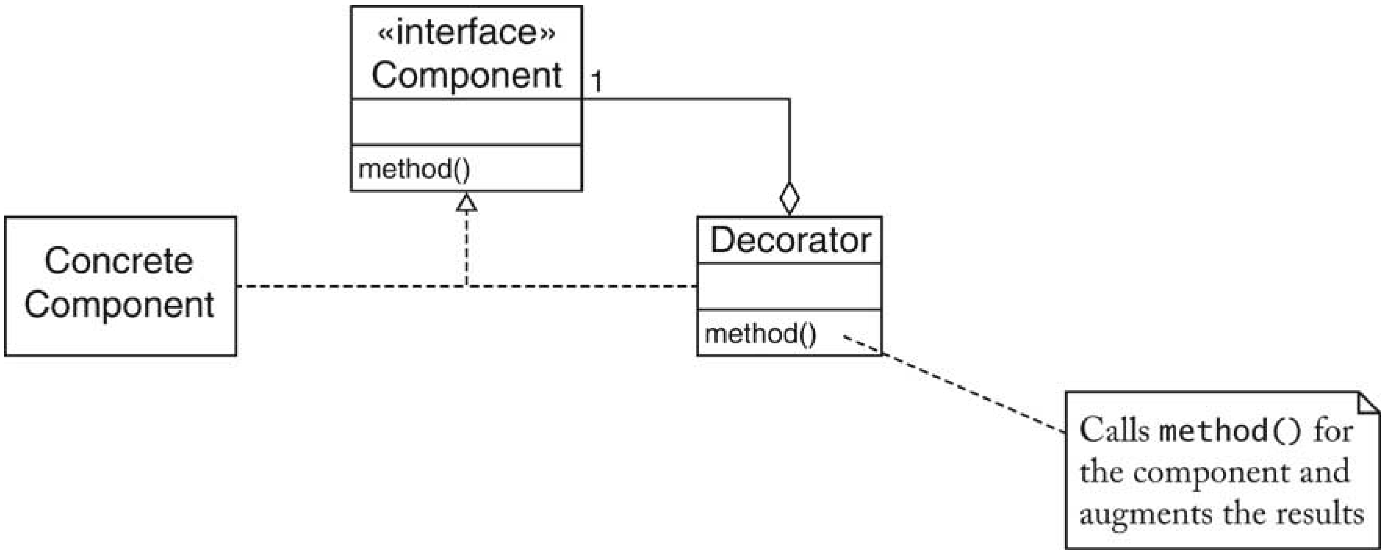
1. Component objects can be decorated (visually or behaviorally enhanced)
2. The decorated object can be used in the same way as the undecorated object
3. The component class does not want to take on the responsibility of the decoration
4. There may be an open-ended set of possible decorations

### 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.

## Structure



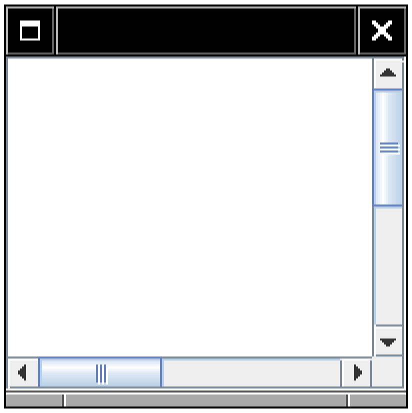
- **Component**
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent** (JTextArea)
  - defines an object to which additional responsibilities can be attached.
- **Decorator** (JScrollPane)
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.

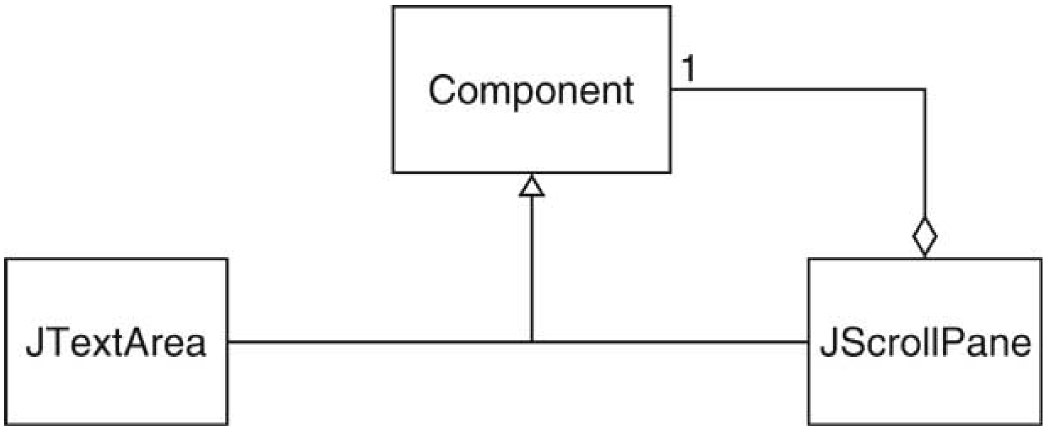
## Example: Scroll Bars

- Scroll bars can surround component

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

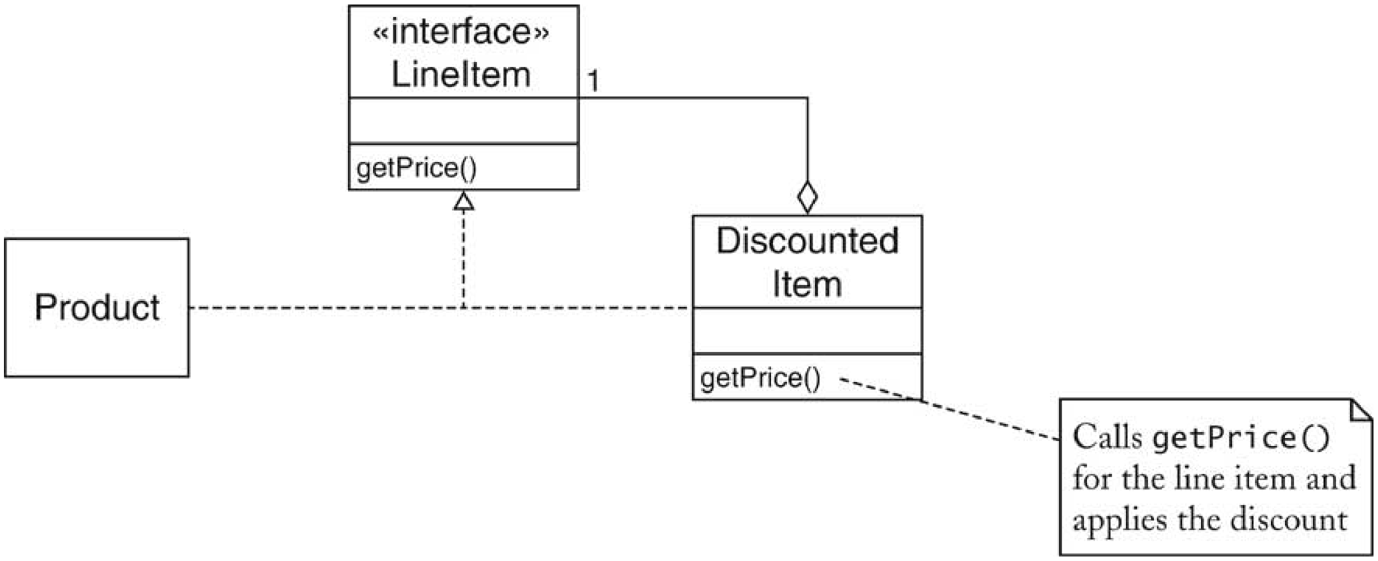
- `JScrollPane` is again a component





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

**Example (Case study from the textbook)**



```
public class DiscountedItem implements LineItem
{   private LineItem item;
    private double discount;

  /**
    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 + "%)"; }
}
```

Note: Composite Pattern vs. Decorator Pattern

- A decorator enhances the behavior of a *single* component, whereas a composite collects *multiple* components.
- A decorator aims to enhance, whereas a composite merely collects.

## 5.3 How to Recognize Patterns

- Use context and solution as "litmus test"
- Example:

```
Border b = new EtchedBorder()
component.setBorder(b);
```

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

## 5.4 Case Study: Putting Patterns to Work

We went over each part of this case study as we learned each design pattern.