# ITEC324 Principle of CS III

Chapter 5 (Horstmann's Book)

Patterns and GUI Programming

*Modified from slides by Dr. Hwajung Lee.*

# Pattern

- A *pattern* is a description of a problem and its solution that you can apply to many programming situation.
  - Standardized patterns
  - User defined patterns

- Pattern presents proven advice in a standard format.

# The Pattern Concept

- History: Architectural Patterns by 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*

# e.g. a pattern to build a hallway.

- Name: short passages
- Context: "… long, sterile corridors set the scene for everything bad about modern architecture".
- Problem:
  - // This contains a lengthy description of passages with a dismal picture.
- Solution:
  - Keep passages short. Make them as much like rooms as possible, with carpets or wooden floors, furniture, bookshelves, beautiful windows.

# Goal is to build an invoice.

- A GUI that allows a user to select one or more products and the invoice keeps computing the total cost along with any discounts.
  - We will use this to study some patterns in Java.

# COMPOSITE Pattern

## Containers and Components

- Issue:
  - Line items in the inventory can be some primitive items such as hammer, tongs, horse-shoes etc..
  - Or,
    - A bundle: e.g., blacksmith special: get a hammer and tong and horse-shoe for 10% less!
- How here is the problem: sometime a primitive type (e.g., the bundle) can itself be a collection of other primitive types (e.g., the hammer and tongs).
- Solution: composite pattern.

# Bundles

- Bundle = set of related items with description+price

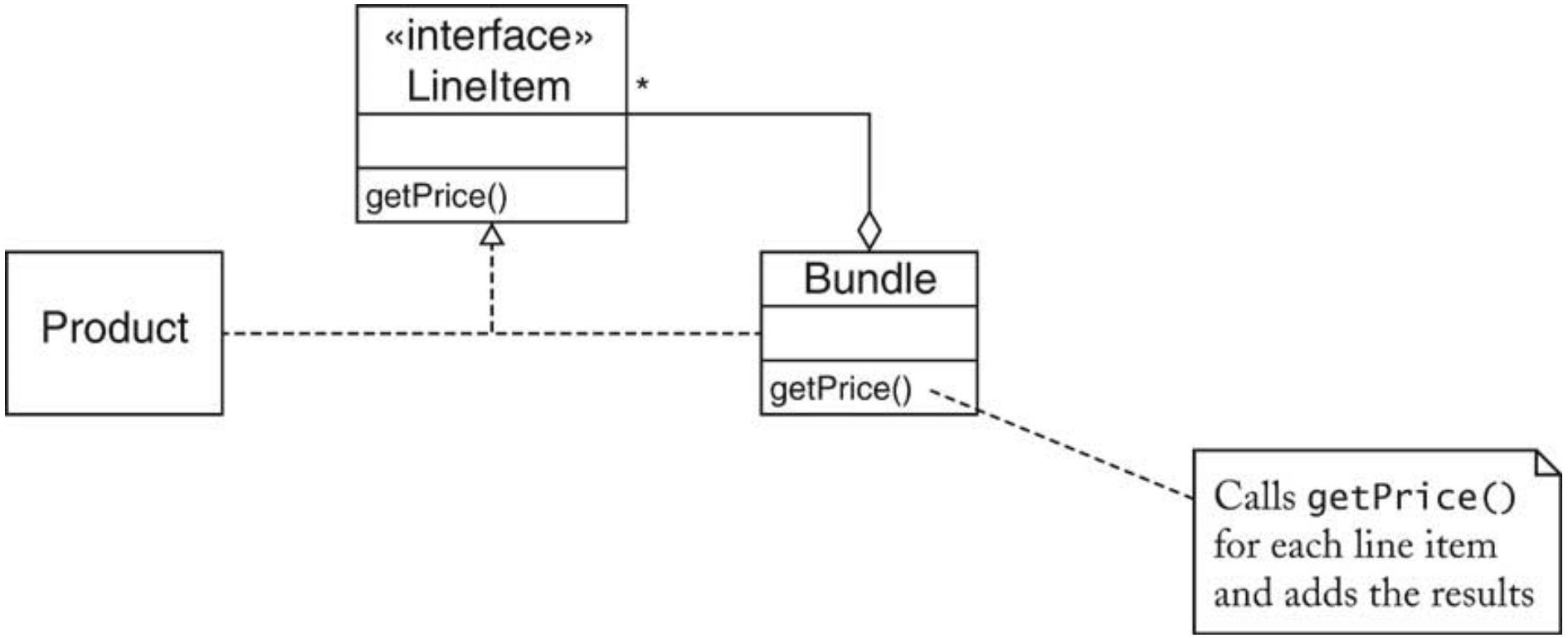
  (ex) stereo system with tuner, amplifier, CD player + speakers

  - A bundle has line items
  - A bundle is a line item

➔ Therefore, COMPOSITE pattern

Ch5/invoice/Bundle.java (look at getPrice)

# Bundles


«interface»
LineItem
*
getPrice()
Product
Bundle
getPrice()
Calls getPrice()
for each line item
and adds the results

# COMPOSITE Pattern

## Containers and Components

- Containers collect GUI components
- Sometimes, want to add a container to another container
- Container should *be* a component
- Composite design pattern
  - Composite method typically invoke component methods
  - E.g. Container.getPreferredSize invokes getPreferredSize of components

# COMPOSITE Pattern

## 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
## Containers and Components

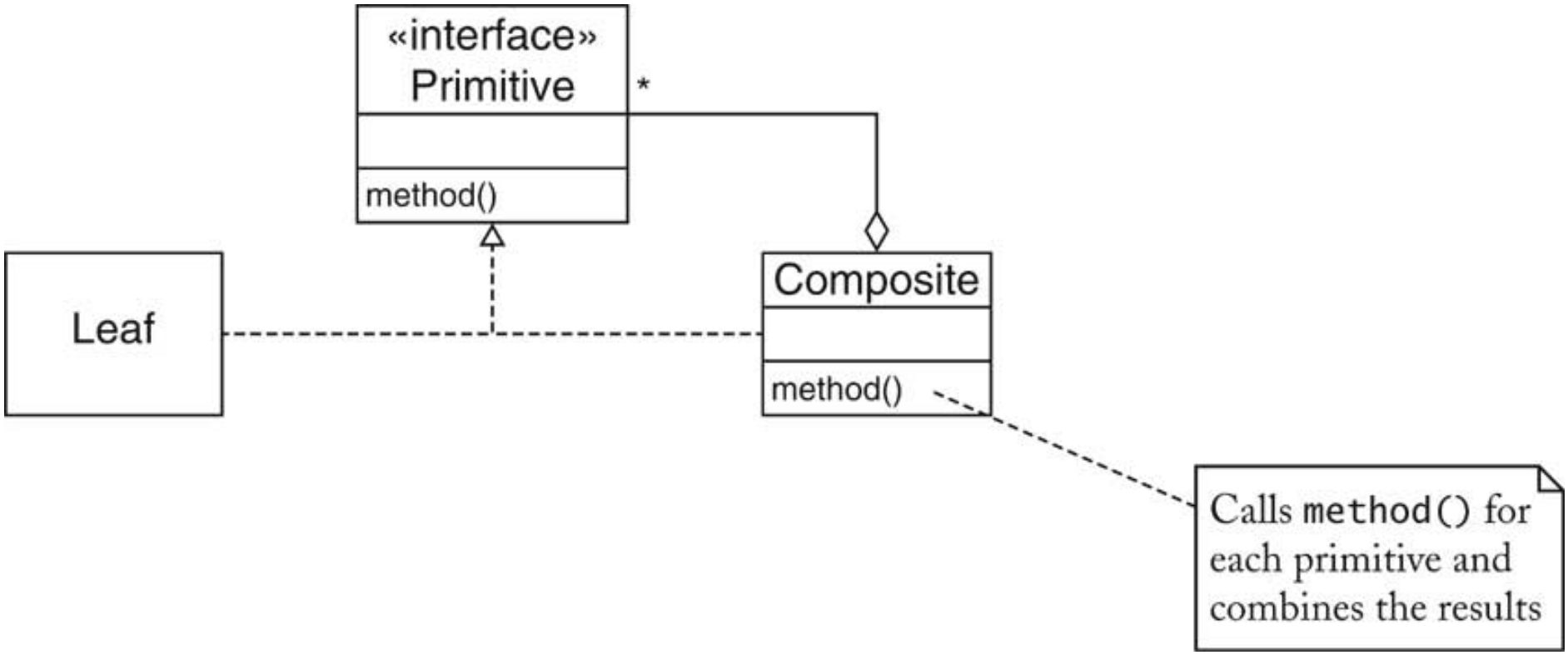
- The COMPOSITE pattern teaches how to combine several objects into an object that has the same behavior as its parts.

# 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 contains a collection of primitive objects
  3. Both primitive classes and composite classes implement that interface type (from (1)).
  4. When implementing a method from the interface type, the composite class applies the method to its primitive objects and combines the results. (e.g., let us see how Bundle class does this).

# COMPOSITE Pattern



# Using your knowledge of Java Swing…

- Can you think of any composite patterns you encountered?

# COMPOSITE Pattern: examples in Java Swing.

| Name in Design Pattern | Actual Name |
|---|---|
| Primitive | Component |
| Composite | Container or a subclass such as JPanel |
| Leaf | A component that has no children such as JButton or JTextArea |
| method() | A method of the Component interface such as getPreferredSize |

# Adding decorations: DECORATOR PATTERN

- Economy is bad…
- The invoice system now wants to add a new feature:
  - For a bundle it will give an additional discount.
- How can we design our classes to provide this discount?
- Solution; DECORATOR pattern.

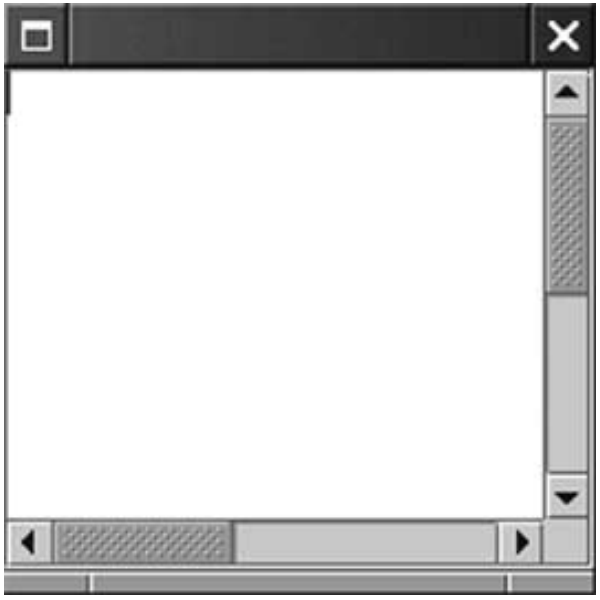
# Another example: DECORATOR Pattern

## Scroll Bars

- Scroll bars can surround component

```
JTextArea area = new JTextArea(10, 25);
JScrollPane pane = new JScrollPane(area);
```

- JScrollPane is again a component



# DECORATOR Pattern
## Scroll Bars

- The DECORATOR pattern teaches how to form a class that adds functionality to another class while keeping its interface.

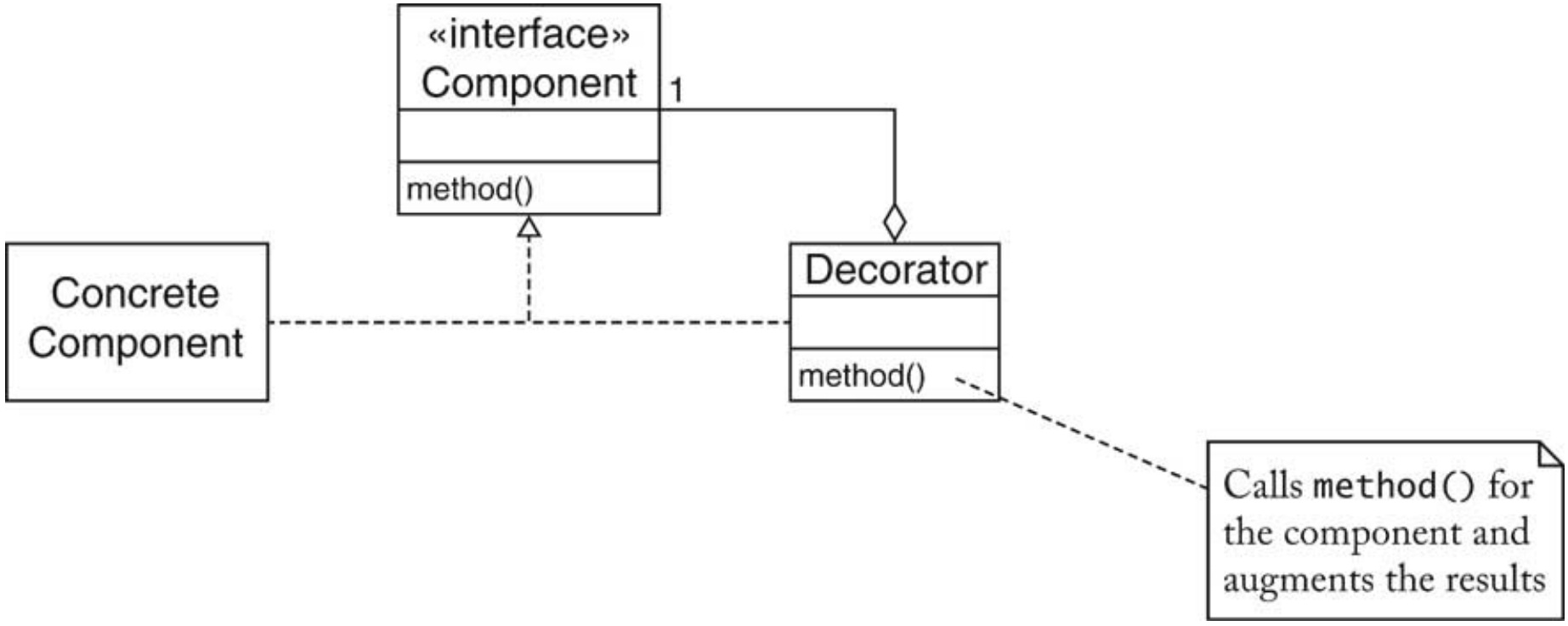
# DECORATOR Pattern

- **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

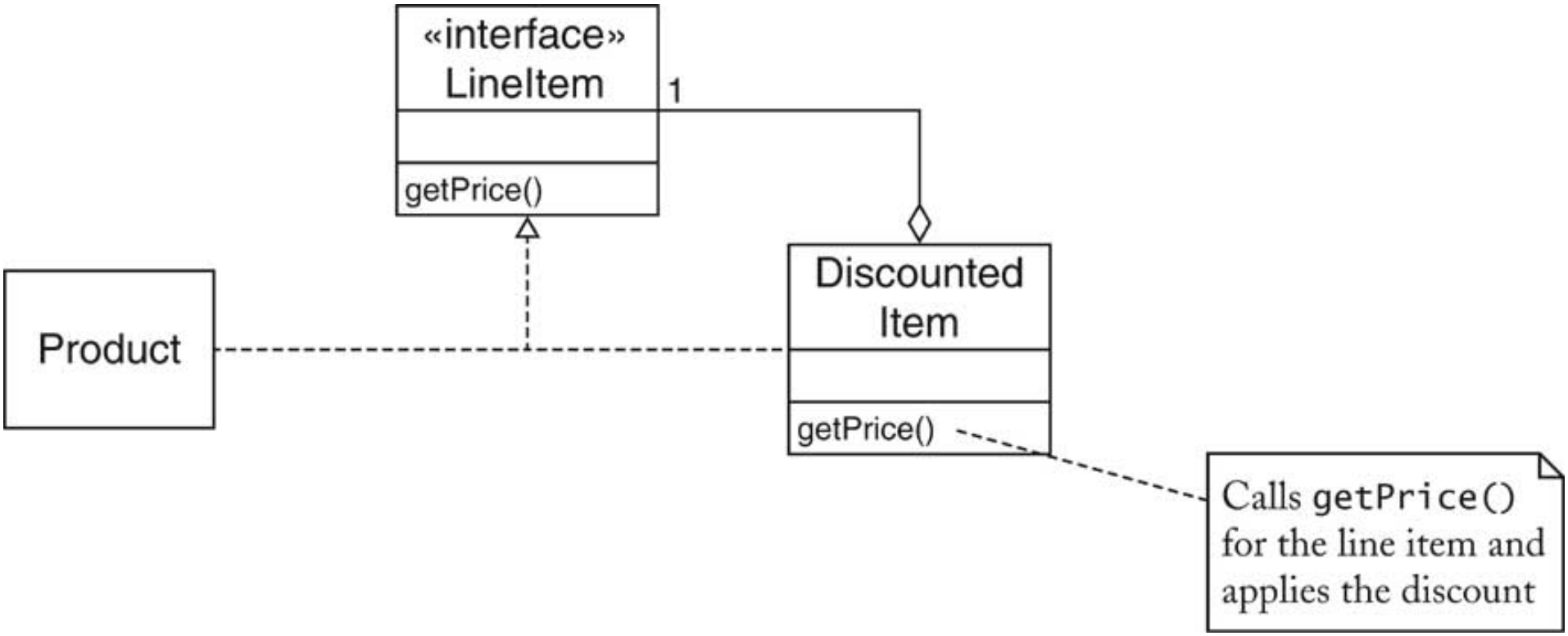
# DECORATOR Pattern

- **Solution**
  1. Define an interface type that is an abstraction for the component
  2. Concrete component classes implement this interface type.
  3. Decorator classes also implement 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



# Discounted Items



# DECORATOR Pattern

| Name in Design Pattern | Actual Name |
|---|---|
| Component | Component |
| ConcreteComponent | JTextArea |
| Decorator | JScrollPane |
| method() | A method of the Component interface. For example, the paint method paints a part of the decorated component and the scroll bars. |

# Next issue: displaying the invoice.

→ OBSERVER PATTERN

# OBSERVER Pattern

- Class exercise:
  - Part 1:  Create a GUI that contains two frames.
    - Frame 1 contains a textbox.
    - Frame 2 contains a Jlabel.
    - When any text on Frame 1 is changed, the changed text must display on Frame 2 .
  - Part 2: Now add another frame, Frame 3, which will display if the number on Frame1 is even or odd.

# OBSERVER Pattern

- Some programs have multiple editable views of the same data.
  - Example: HTML Editor
    - WYSIWYG (What you see is what you get) view
    - A structural view
    - …
- When you edit on of the view, the other updates automatically and instantaneously.
  - How is this behavior implemented?
  - Solution: Model/View/Controller (MVC).

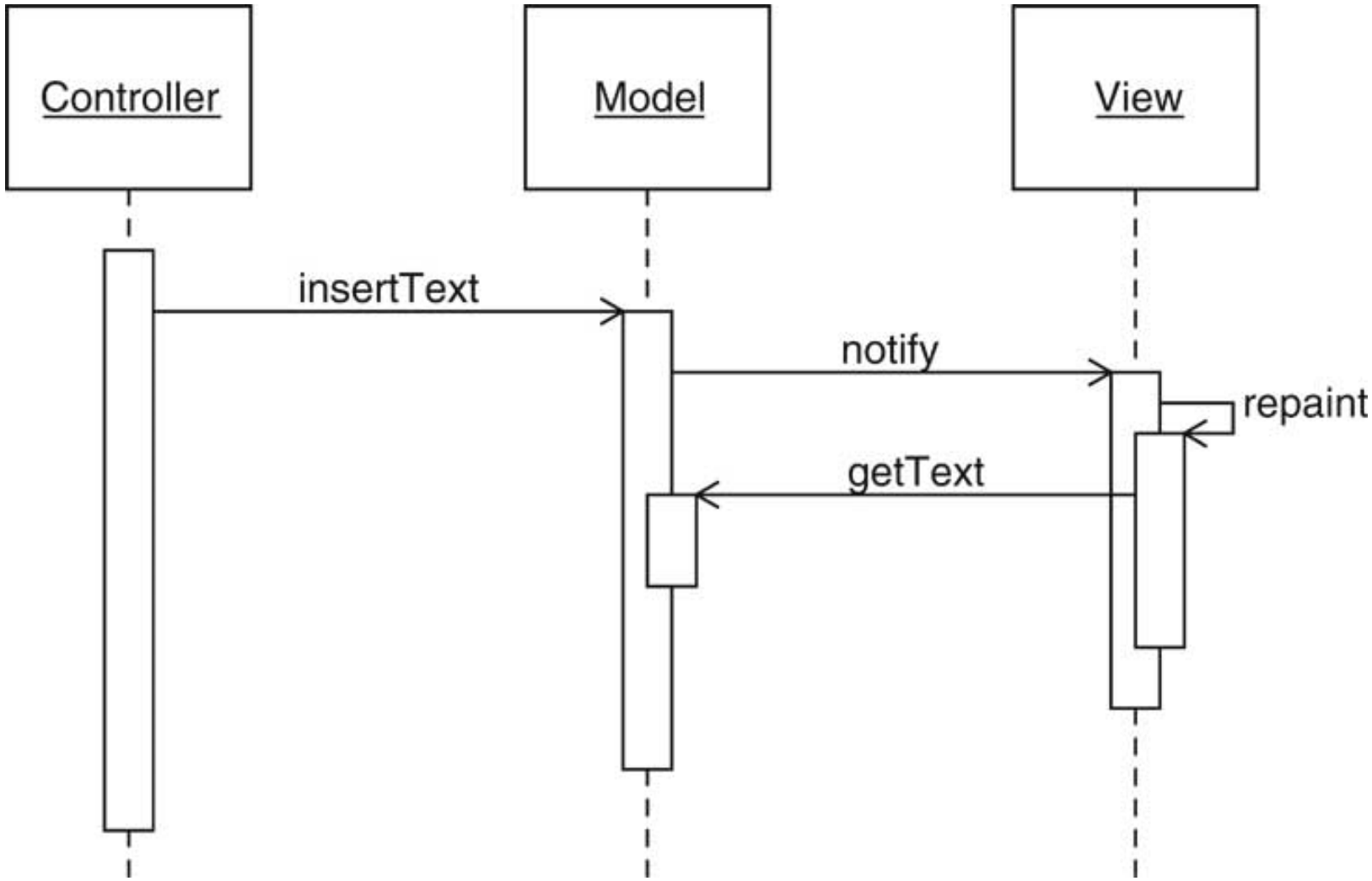
# OBSERVER Pattern

- Model/view/controller architecture
  - Model
    - The raw data
    - Data structure
    - No visual appearance
  - Views
    - Visual representations
  - Controllers
    - An object that processes user interaction (using mouse, keyboard, GUI interface, …)
    - Each view has a controller.
- Next: how a controller works.

# OBSERVER Pattern

- When a user types text into one of the windows:
  - The **controller** tells the **models** to insert the text that the user typed.
  - The **model** notifies all **views** of a change in the model.
  - All **views** repaint themselves.
  - During paint, each **view** asks the **models** for the current text.
- This architecture minimizes the coupling between the model, views, and controllers.

# OBSERVER Pattern



# OBSERVER Pattern

- The OBSERVER pattern teaches how an object can tell other objects about events.

- **Context**
  1. An object (which we'll call the *subject)* is source of events (such as "my data has changed").
  2. One or more objects (called the *observer* ) want to know when an event occurs.

# OBSERVER Pattern

- **Solution**
  1. Define an observer interface type. Observer classes must implement this interface type.
     1. Java Swing does provide a "ChangeListener" interface that can be used…

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

  2. The subject maintains a collection of observer objects.
  3. The subject class supplies methods for attaching observers.

     We can achieve this by making the main program of our code have some method that keeps adding the new "observers" as they are created. E.g., the "addChangeListener" method in the class Invoice.

# OBSERVER Pattern

- **Solution**

  4. Whenever an event occurs, the subject notifies all observers.

# 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:

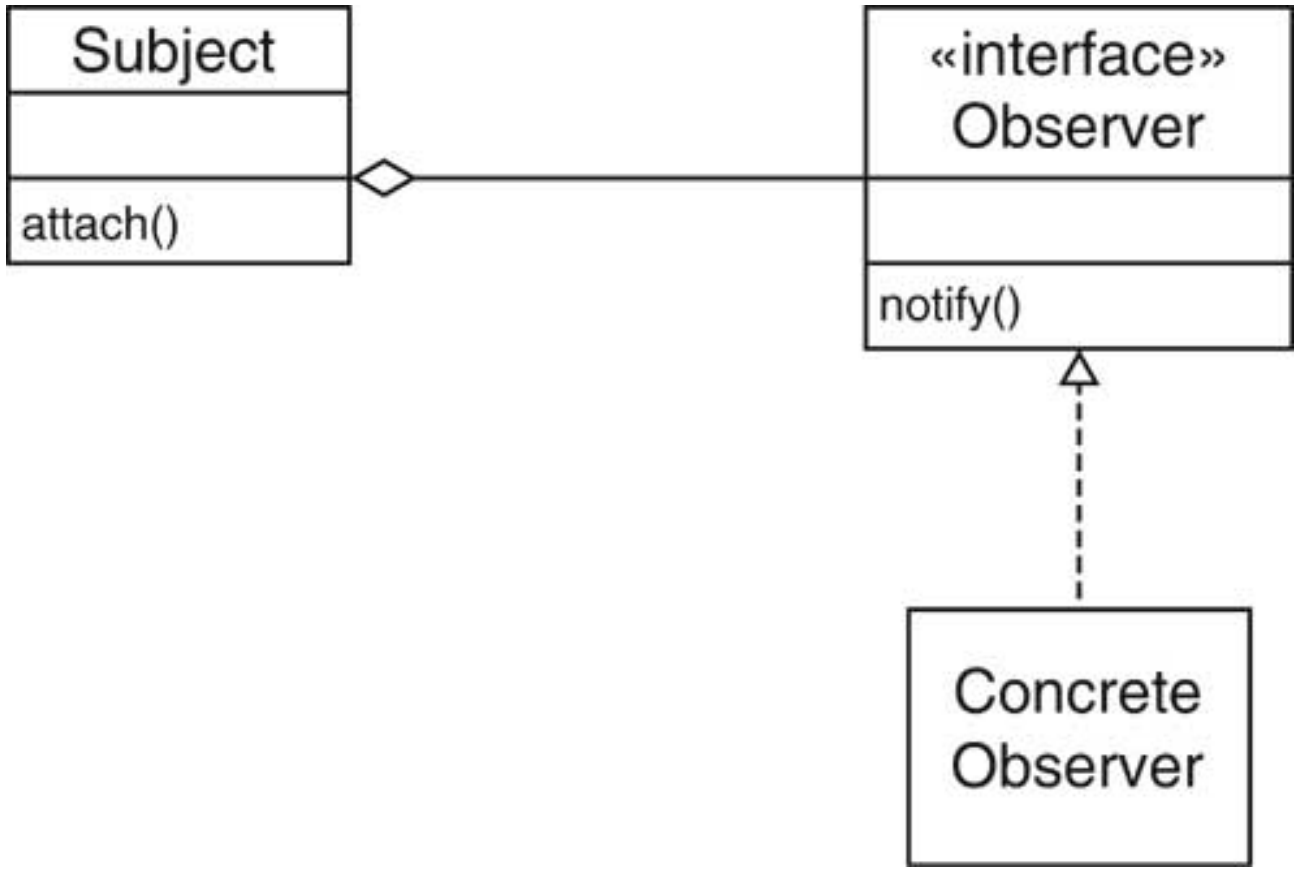
```
ChangeEvent event = new ChangeEvent(this);
for (ChangeListener listener : listeners)
    listener.stateChanged(event);
```

# 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)
      {
               String formattedInvoice = …;
               textArea.setText(formattedInvoice);
      }
   };
invoice.addChangeListener(listener);
```

# OBSERVER Pattern



# OBSERVER Pattern

| Name in Design Pattern | Actual Name |
|---|---|
| Subject | JButton |
| Observer | ActionListener |
| ConcreteObserver | The class that implements the ActionListener interface type |
| attach() | addActionListener |
| notify() | actionPerformed |

# Iterating Through Invoice Items

- Accessing 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

# List Iterators

- Example

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

- Why does the Java library use an iterator to traverse a linked list?

# Classical List Data Structure

- Programmer manipulates the links directly

```
Link currentLink = countries.head;
while (currentLink != null)
{
      do something with currentLink.data;
      currentLink = currentLink.next;
}
```

  - Thus, this exposes implementation.
    - Break Encapsulation unnecessarily
    - Easy to mess up and corrupt the link structure of a linked list

# ITERATOR Pattern

- The ITERATOR pattern teaches how to access the elements of an aggregate object.

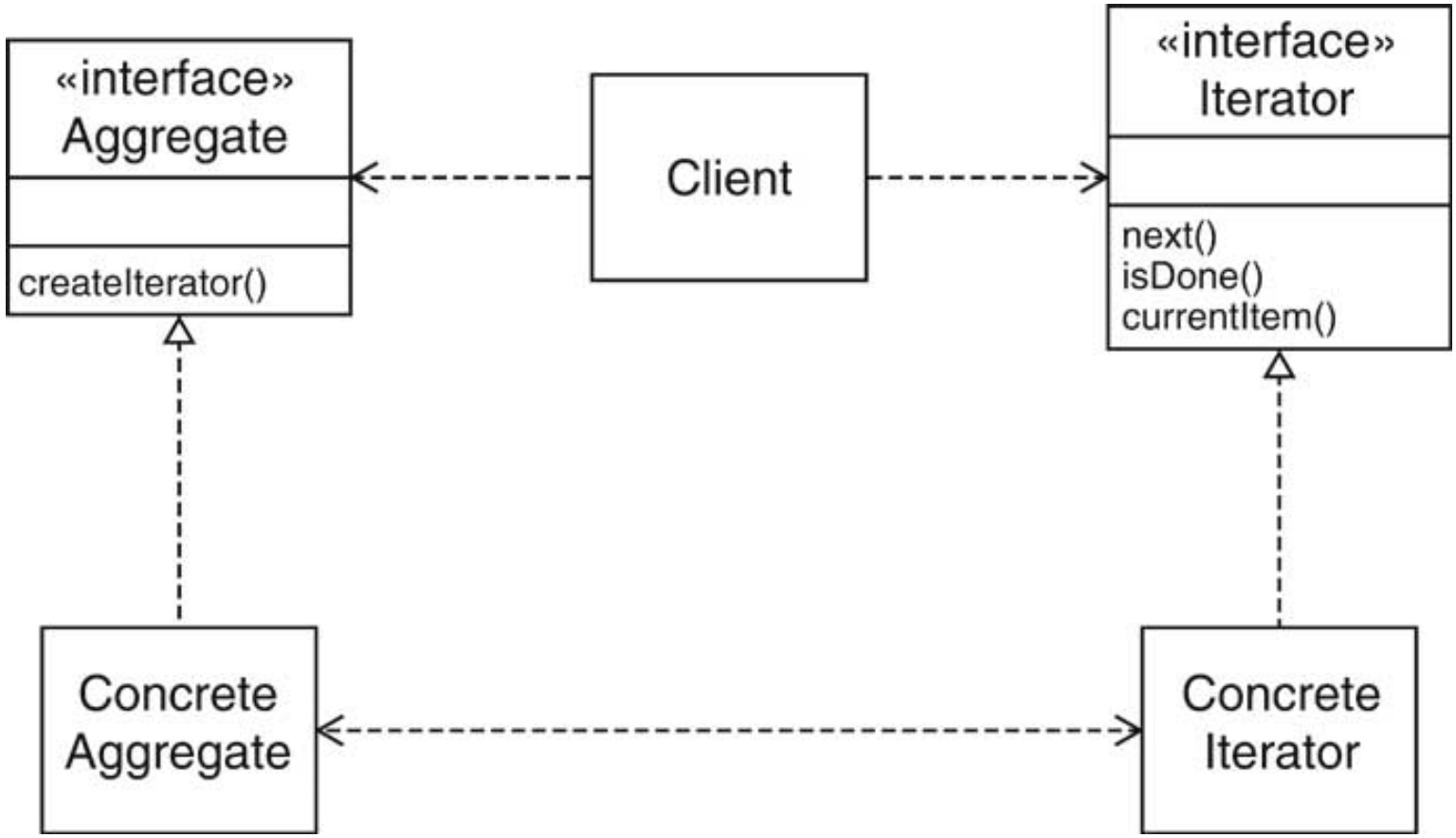
# ITERATOR Pattern

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

# ITERATOR Pattern

- **Solution**
  1. Define an iterator that fetches one element at a time.
  2. Each iterator object needs to keep track of the position of the next element to fetch.
  3. If there are several variations of the aggregate and iterator classes, it is best if they implement common interface type. Then the client only needs to know the interface types, not the concrete classes.

# ITERATOR Pattern



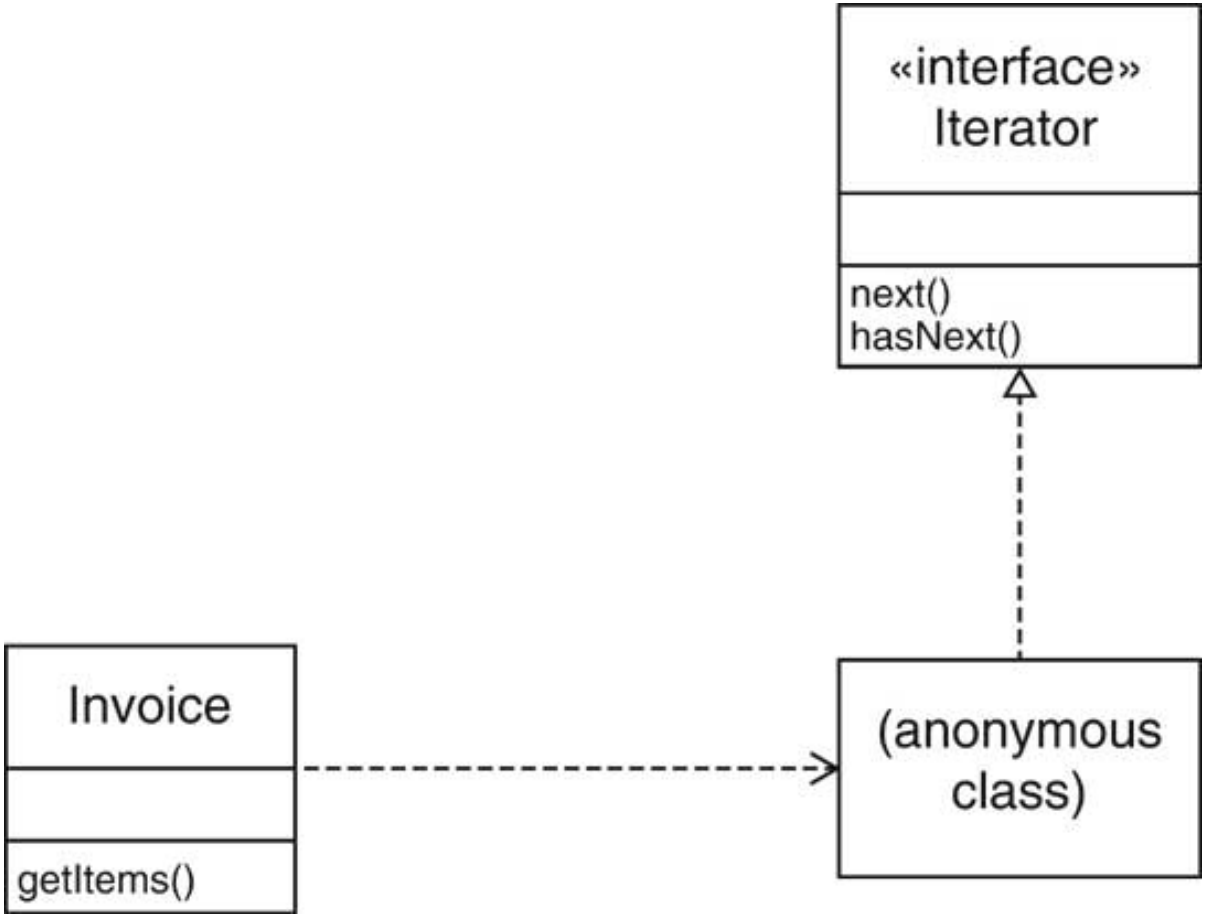
# ITERATOR Pattern

- Names in pattern are *examples* and may differ in each occurrence of pattern.

# ITERATOR Pattern

| Name in Design Pattern | Actual Name |
|---|---|
| Aggregate | List |
| ConcreteAggregate | LinkedList |
| Iterator | ListIterator |
| ConcreteIternator | An anonymous class that implements the ListIterator interface type |
| createIterator() | listIterator() |
| next() | next() |
| isDone() | Opposite of hasNext() |
| currentItem() | Return value of next() |

# Iterators


«interface»
Iterator
next()
hasNext()
Invoice
getItems()
(anonymous
class)

# Iterators

- Use standard Iterator interface type

```
public interface Iterator<E>
{
   boolean hasNext();
   E next();
   void remove();
}
```

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

# Formatting Invoices

- Simple format: dump into text area
  - May not be good enough

OR

- Invoice on a Web page?
  - E.g. HTML tags for display in browser

➔Want to allow for multiple formatting algorithms

➔STRATEGY pattern

# STRATEGY Pattern

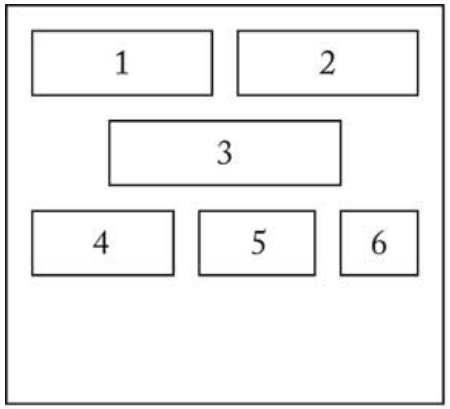
## Layout Managers

- What if we need to specify pixel position of components when
  - User interfaces are made up of use interface *components*
  - Components are placed in *containers*
- Swing doesn't use hard-coded pixel coordinates for each component.
  - Advantages:
    - Can switch between various "look and feel"
    - Can internationalize strings
- Layout manager arranges the components in a container.

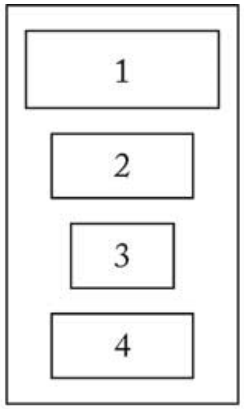
# STRATEGY Pattern
## Layout Managers

- 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: the rows & columns can have different sizes and components can span multiple rows and columns

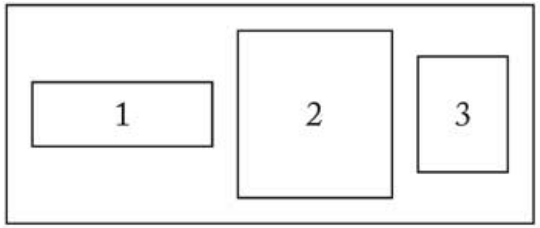
# STRATEGY Pattern
## Layout Managers



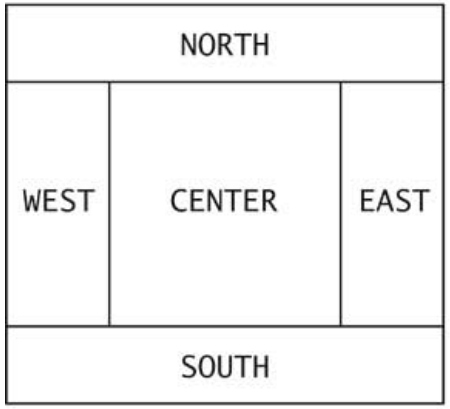

FlowLayout



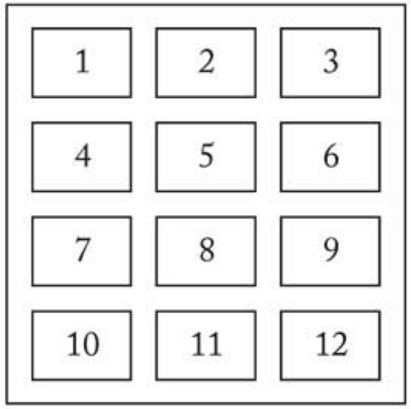

BoxLayout (vertical)



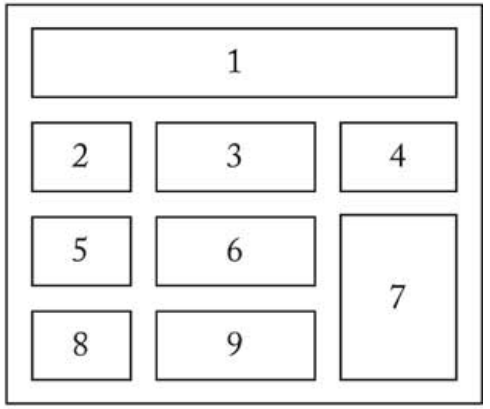

BoxLayout (horizontal)




BorderLayout




GridLayout




GridBagLayout

# STRATEGY Pattern
## Layout Managers

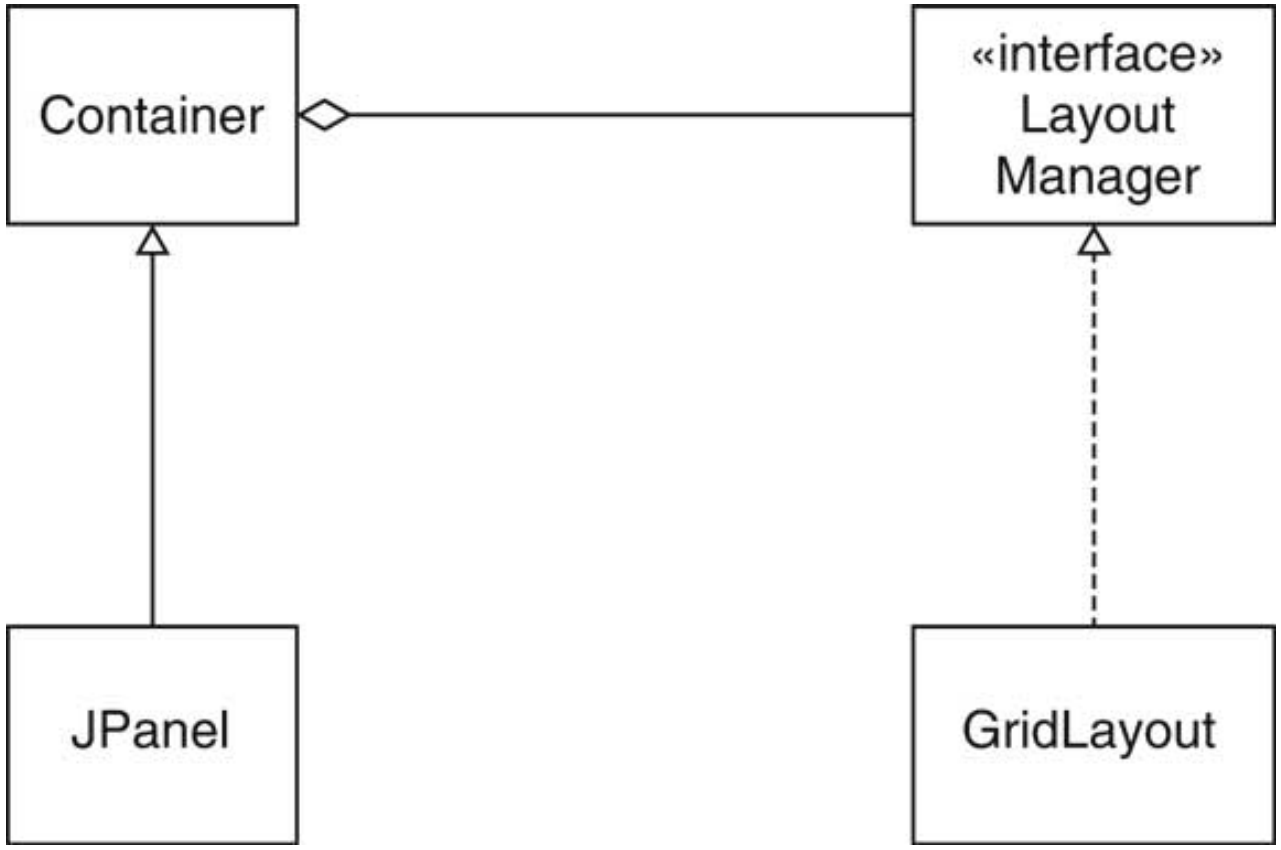
- Panel
  - Set layout manager

```
JPanel keyPanel = new JPanel();
keyPanel.setLayout(new GridLayout(4, 3));
```

  - Add components

```
for (int i = 0; i < 12; i++)
{
    keyPanel.add(button[i]);
} //end for
```

# STRATEGY Pattern
## Layout Managers

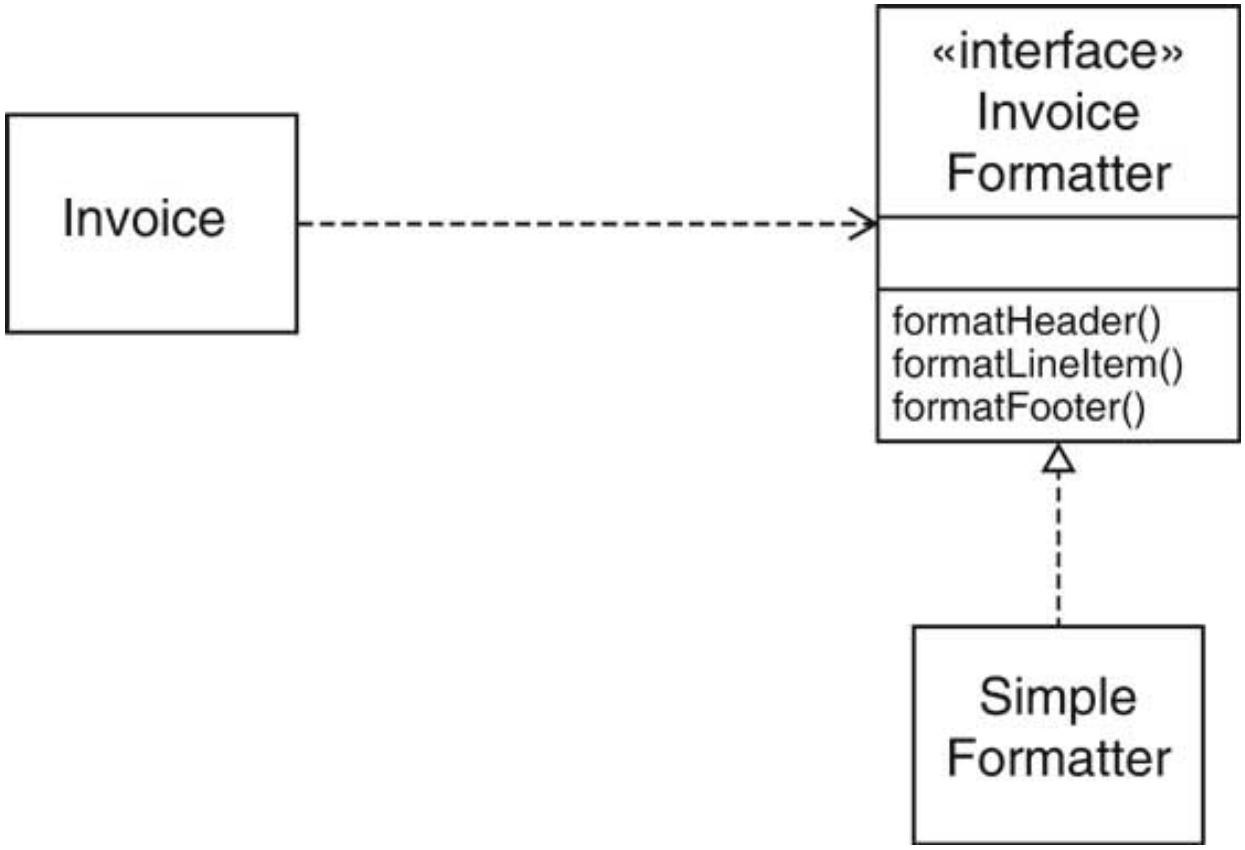


# STRATEGY Pattern

## (Ex) Voice Mail System GUI

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

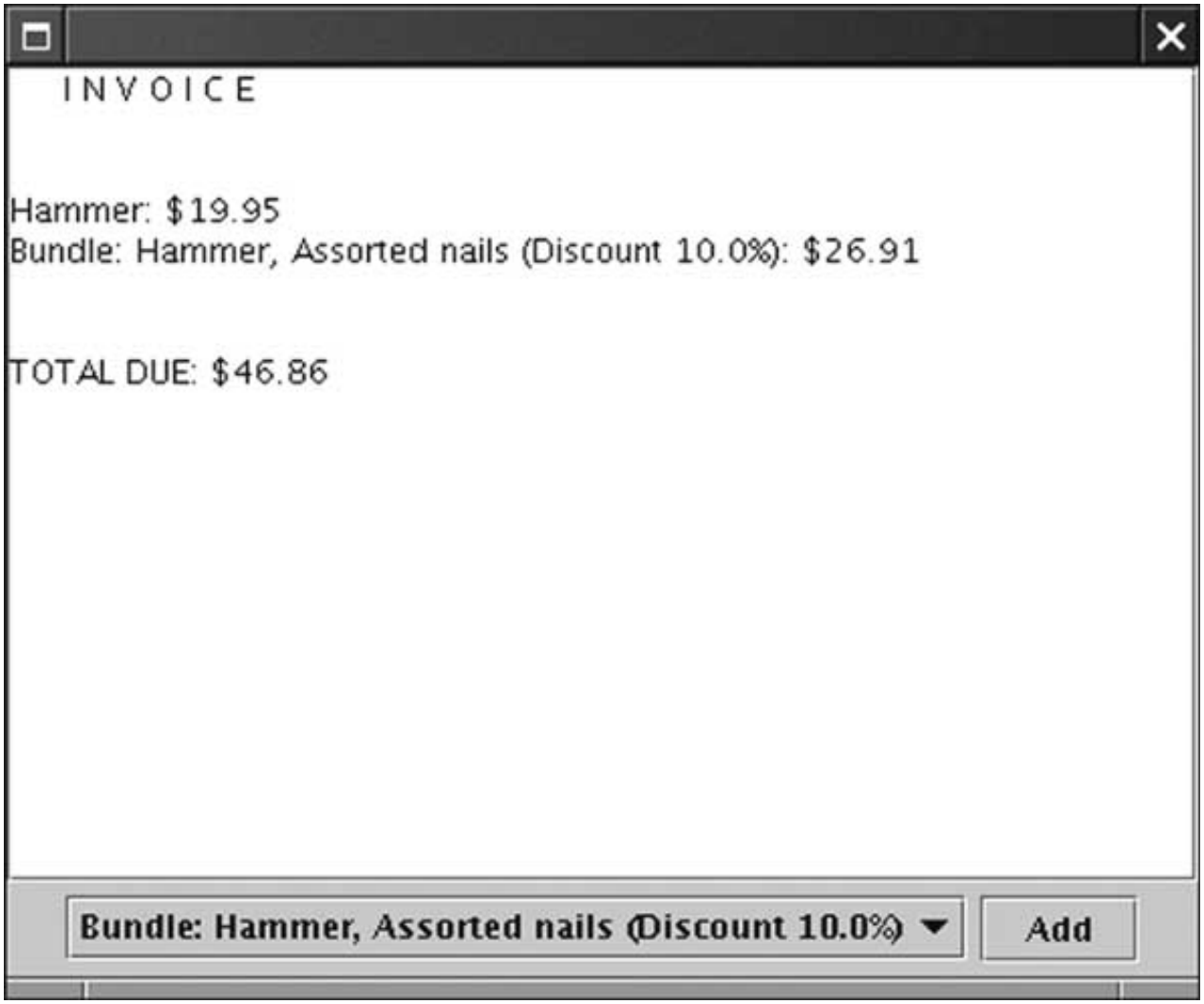
# Formatting Invoices



# Formatting Invoices

- ch5/invoice/InvoiceFormatter.java
- ch5/invoice/SimpleFormatter.java
- ch5/invoice/Invoice.java
- ch5/invoice/InvoiceTester.java

# Formatting Invoices



# How to Recognize Patterns

- Look at the *intent* of the pattern

  (ex1) COMPOSITE pattern: to group component into a whole

  (ex2) DECORATOR pattern: to decorate a component

  (ex3) STRATEGYpattern: to wrap an algorithm into a class.
- 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

- We can add border to Swing component.
  Border b = new EtchedBorder()
  component.setBorder(b);

- Is it an example of DECORATOR?

# Litmus Test

- Component objects can be decorated (visually or behaviorally enhanced)

  **PASS**

- The decorated object can be used in the same way as the undecorated object

  **PASS**

- The component class does not want to take on the responsibility of the decoration

  **FAIL--the component class has setBorder method**

- There may be an open-ended set of possible decorations

# Recap of Standardized Patterns

# ITERATOR Pattern

- The ITERATOR pattern teaches how to access the elements of an aggregate object.

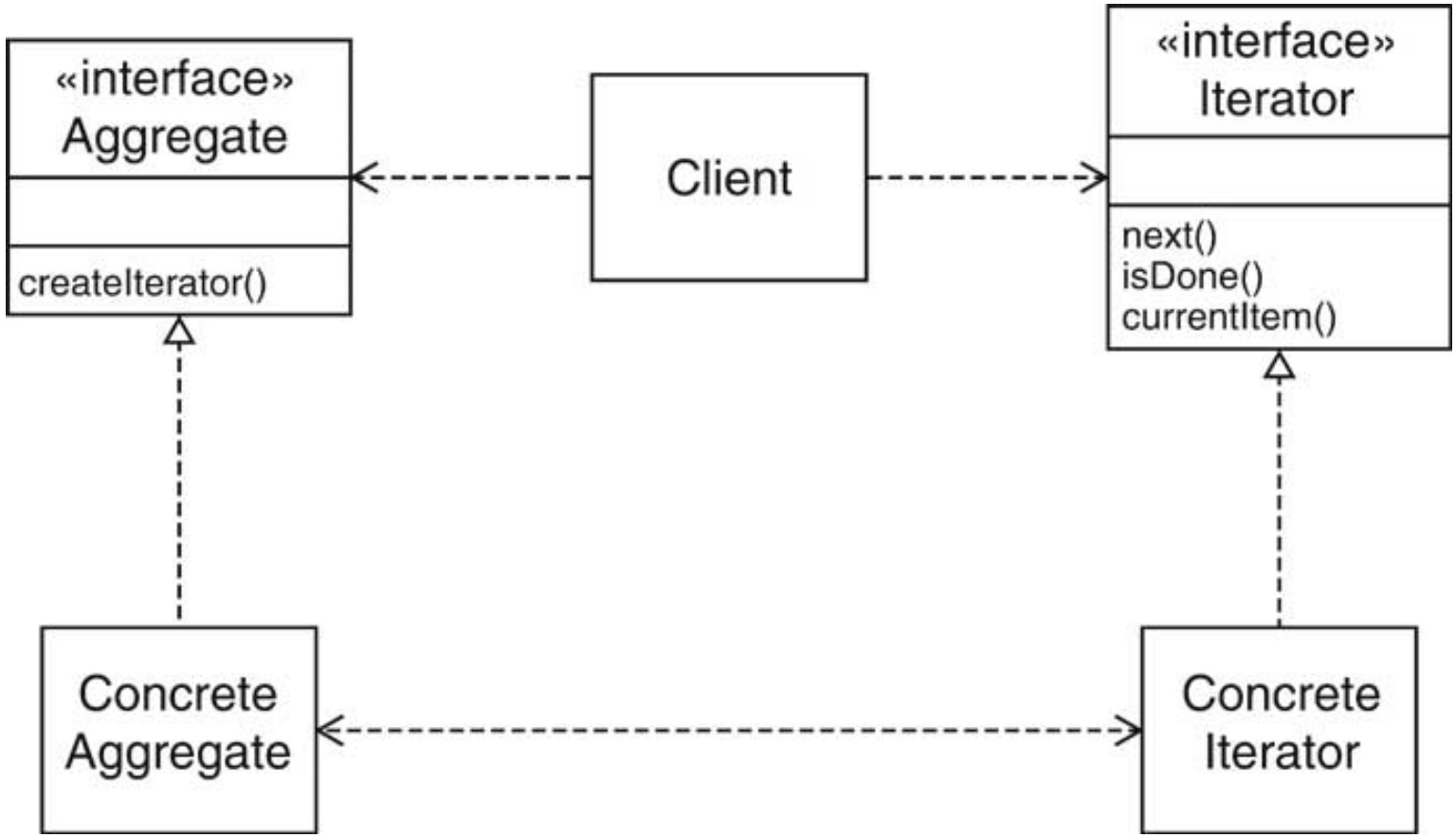
# ITERATOR Pattern

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

# ITERATOR Pattern

- **Solution**
  1. Define an iterator that fetches one element at a time.
  2. Each iterator object needs to keep track of the position of the next element to fetch.
  3. If there are several variations of the aggregate and iterator classes, it is best if they implement common interface type. Then the client only needs to know the interface types, not the concrete classes.

# ITERATOR Pattern



# OBSERVER Pattern

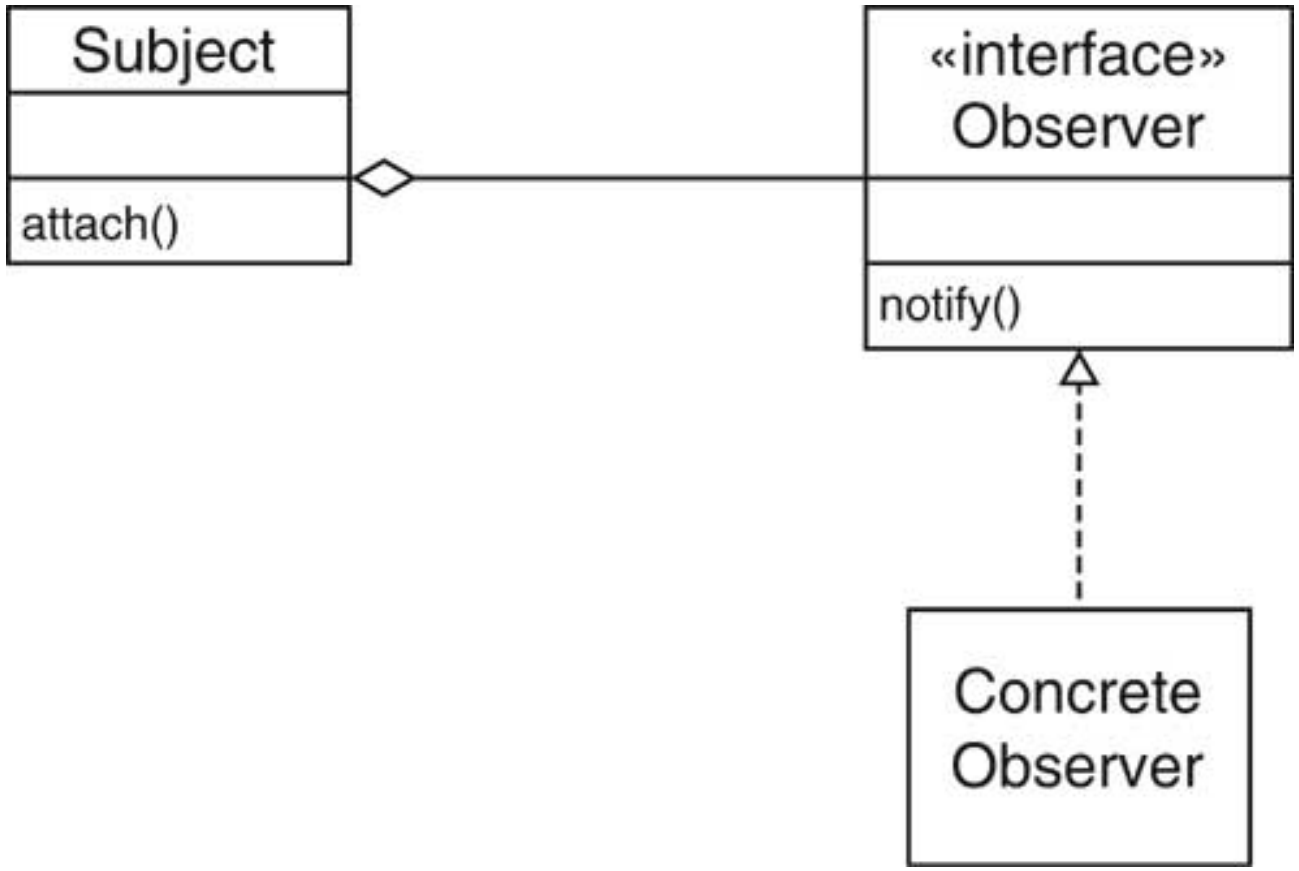
- The OBSERVER pattern teaches how an object can tell other objects about events.

- **Context**
  1. An object (which we'll call the *subject)* is source of events (such as "my data has changed").
  2. One or more objects (called the *observer* ) want to know when an event occurs.

# OBSERVER Pattern

- **Solution**
  1. Define an observer interface type. Observer classes must implement this interface type.
  2. The subject maintains a collection of observer objects.
  3. The subject class supplies methods for attaching observers.
  4. Whenever an event occurs, the subject notifies all observers.

# OBSERVER Pattern



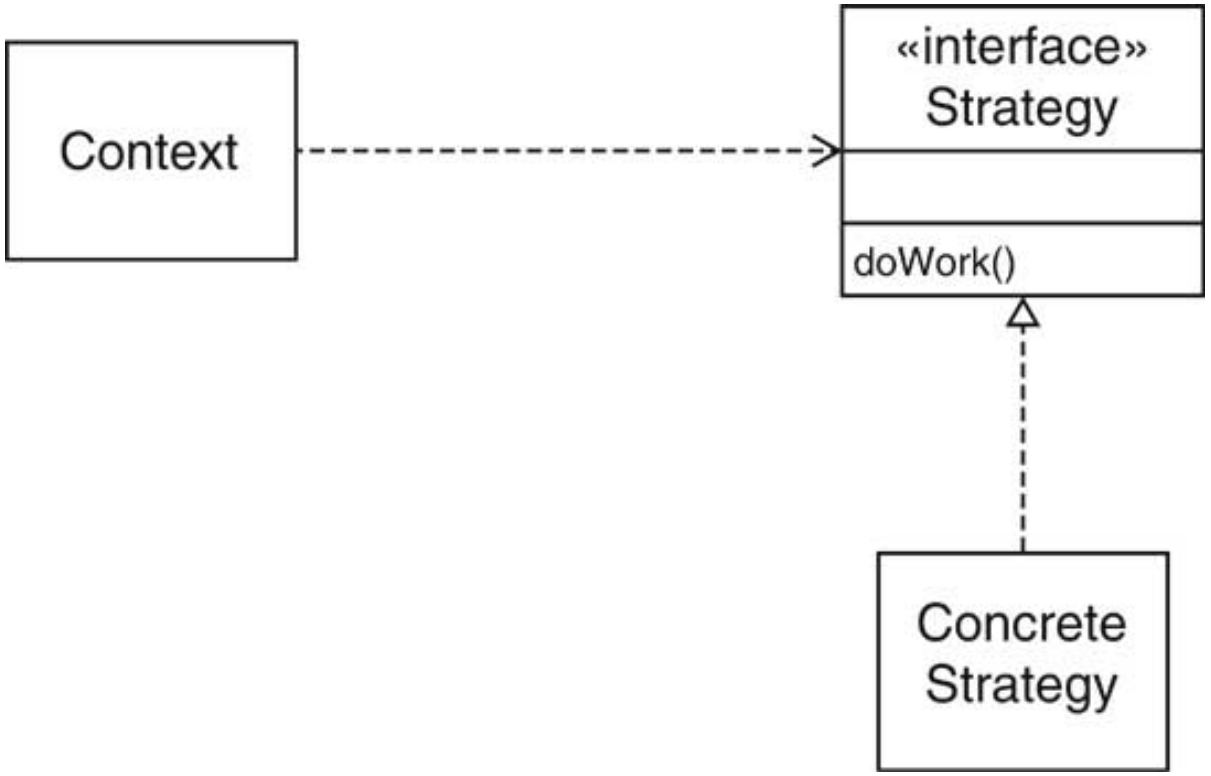
# STRATEGY Pattern

- The STRATEGY pattern teaches how to supply variants of an algorithm

# STRATEGY Pattern

- **Context**
  1. A class (called *context* class) can benefit from different variants for an algorithm
  2. Clients of the context class sometimes want to supply custom versions of the algorithm
- **Solution**
  1. Define an interface type that is an abstraction for the algorithm. We'll call this interface type the *strategy*.
  2. Concrete strategy classes implement the strategy interface type. Each strategy class implements a version of the algorithm.
  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.

# STRATEGY Pattern



# COMPOSITE Pattern
## Containers and Components

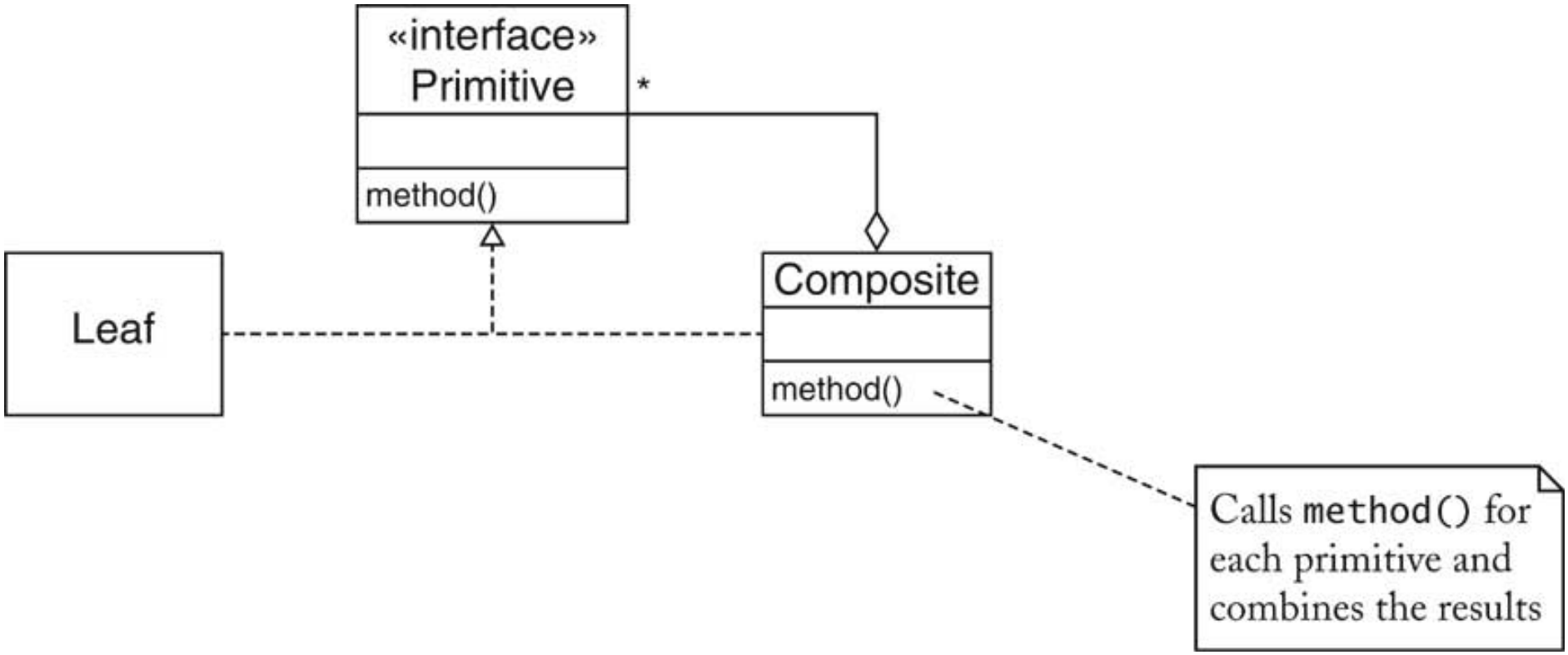
- The COMPOSITE pattern teaches how to combine several objects into an object that has the same behavior as its parts.

# 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 contains a collection of primitive objects
  3. Both primitive classes and composite classes implement that 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



# DECORATOR Pattern
## Scroll Bars

- The DECORATOR pattern teaches how to form a class that adds functionality to another class while keeping its interface.

# 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 implement this interface type.
  3. Decorator classes also implement 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