# CSE 223: Programming -2
# 11-Behavioral Design Patterns (II)

Prof. Dr. Khaled Nagi

*Department of Computer and Systems Engineering,*

*Faculty of Engineering, Alexandria University, Egypt.*

# Agenda

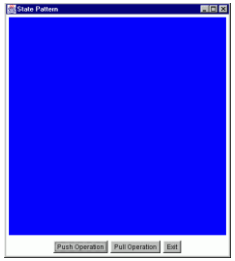
- *They are used to organize, manage, and combine behavior*.
- *They help you define the communication between objects in your system and how the flow is controlled in a complex program.*
- List
  - Chain of responsibility
  - Command
  - Little language
  - Mediator
  - Snapshot
  - Observer
  - State
  - Null object
  - Strategy
  - Template method
  - Visitor

Part II

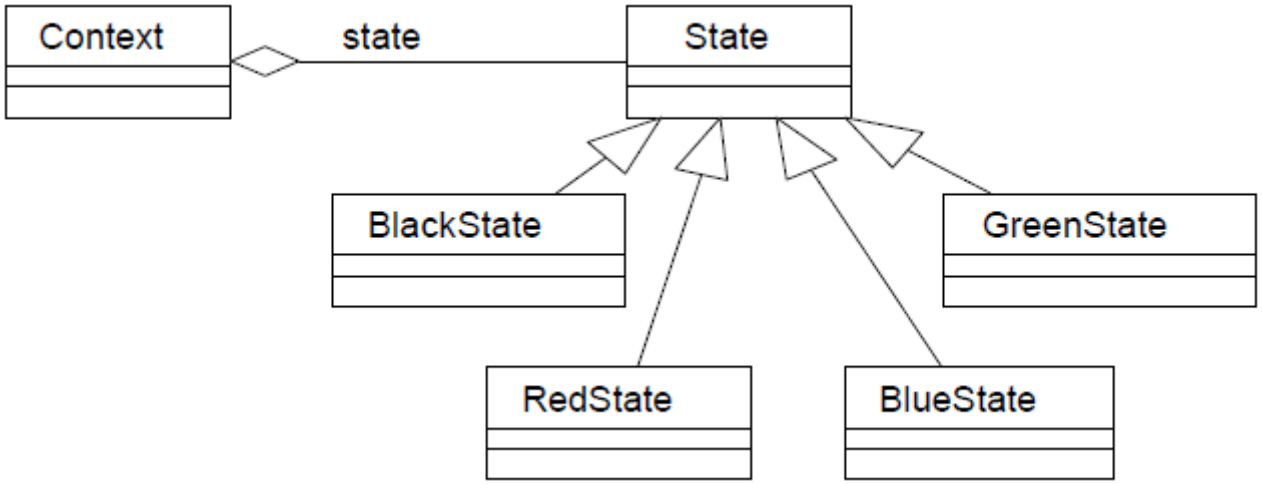

# State Pattern

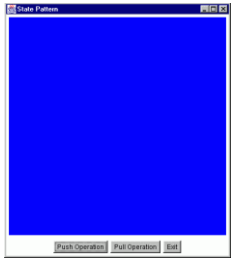
# State Pattern

- Puts all behavior associated with a state into one object
- Allows state transition logic to be incorporated into a state object rather than in a monolithic if or switch statement
- Helps avoid inconsistent states since state changes occur using just the one state object and not several objects or attributes
- Liabilities
  - Increased number of objects
- Example
  - Consider a class that has two methods, push() and pull(), whose behavior changes depending on the state of the object
  - To send the push and pull requests to the object, we'll use the following GUI with "Push" and "Pull" buttons
  - The state of the object will be indicated by the color of the canvas in the top part of the GUI
  - The states are: black, red, blue and green



# State Pattern



```
public abstract class State {
  public abstract void handlePush(Context c);
  public abstract void handlePull(Context c);
  public abstract Color getColor();
}
```

# State Pattern

```
public class BlackState extends State {
  // Next state for the Black state:
  //   On a push(), go to "red"
  //   On a pull(), go to "green"

  public void handlePush(Context c) {
    c.setState(new RedState());
  }

  public void handlePull(Context c) {
    c.setState(new GreenState());
  }

  public Color getColor() {return (Color.black);}
}
```

# State Pattern

```
public class Context {

  // The contained state.
  private State state = null;  // State attribute

  // Creates a new Context with the specified state.
  public Context(State state) {this.state = state;}

 // Creates a new Context with the default state.
 public Context() {this(new RedState());}

 // Returns the state.
 public State getState() {return state;}

 // Sets the state.
 public void setState(State state) {this.state = state;}

 public void push() {state.handlePush(this);}

 public void pull() {state.handlePull(this);}
```

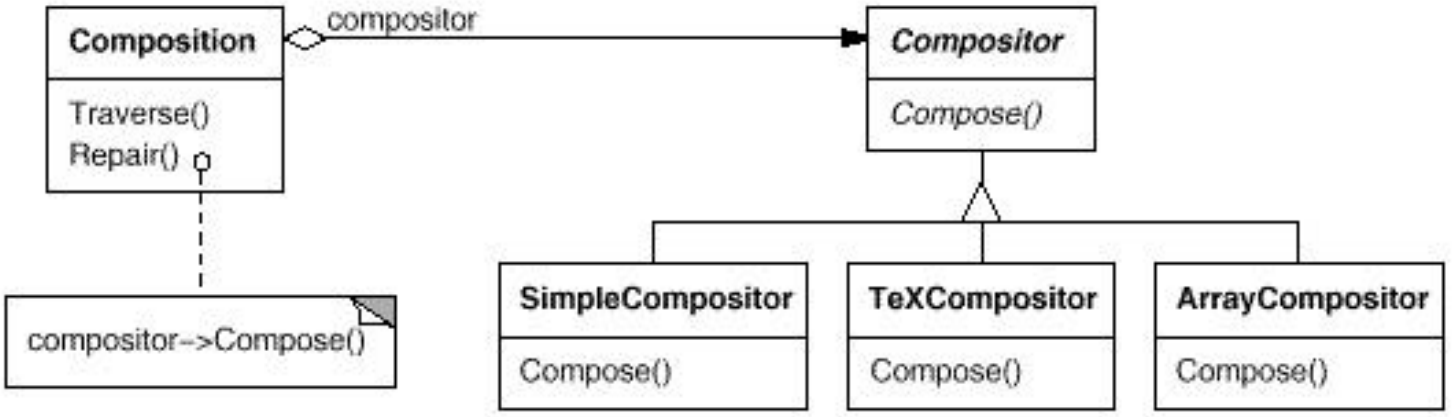
# State Pattern

- Who defines the state transitions?
  - The Context class => ok for simple situations
  - The ConcreteState classes => generally more flexible, but causes implementation dependencies between the ConcreteState classes
- When are the ConcreteState objects created?
  - Create ConcreteState objects as needed
  - Create all ConcreteState objects once and have the Context object keep references to them
- Can't we just use a state-transition table for all this?
  - Harder to understand
  - Difficult to add other actions and behavior


# Strategy Pattern

# Strategy Pattern

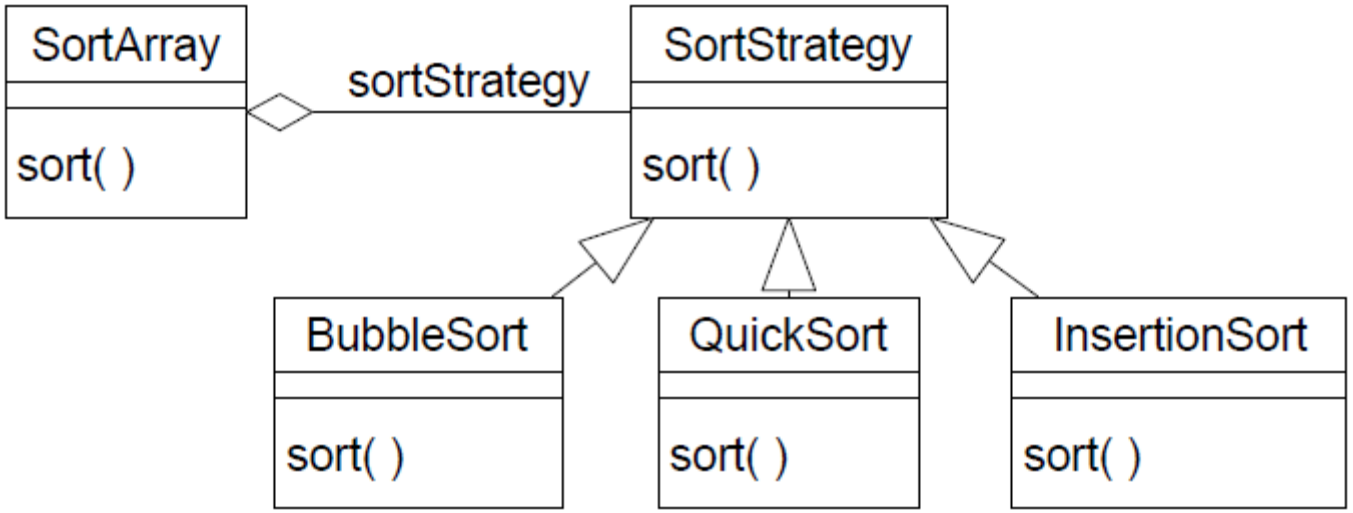
- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Note the similarities between the State and Strategy patterns!
- The difference is one of intent.
  - A *State* object encapsulates a state-dependent behavior (and possibly state transitions)
  - A *Strategy* object encapsulates an algorithm
- They are both examples of **Composition** with **Delegation**!

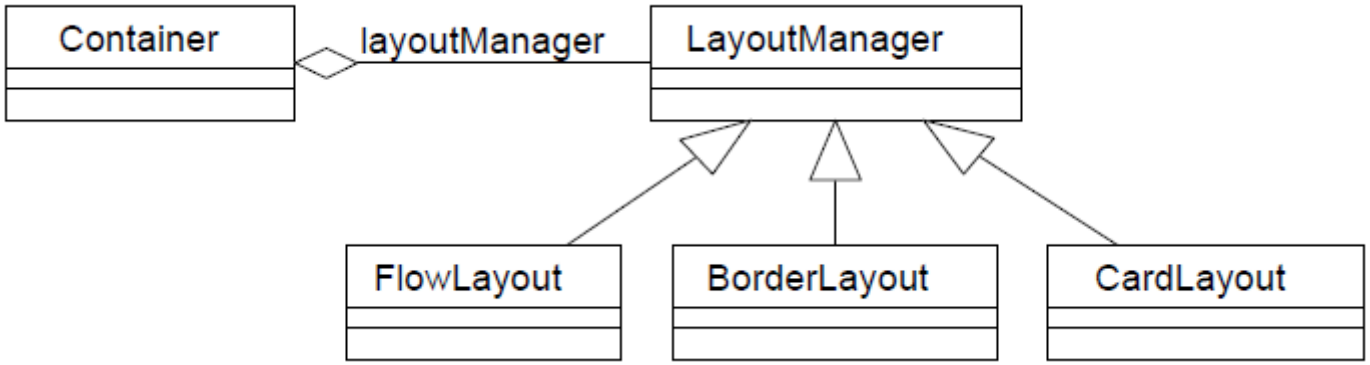


# Strategy Pattern

- Use the Strategy pattern whenever:
  - Many related classes differ only in their behavior
  - You need different variants of an algorithm
  - An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.
- Benefits
  - Provides an alternative to subclassing the Context class to get a variety of algorithms or behaviors
  - Eliminates large conditional statements
  - Provides a choice of implementations for the same behavior
- Liabilities
  - Increases the number of objects
  - All algorithms must use the same Strategy interface

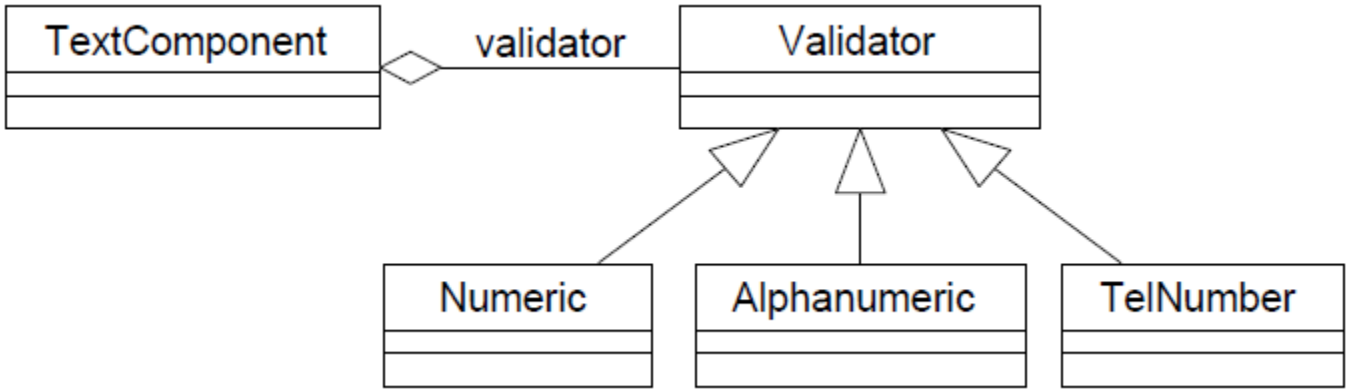
# Strategy Pattern




This is what the Java AWT does with its LayoutManagers

# Strategy Pattern




# Null object Pattern

# Null object Pattern

- Sometimes the Context may not want to use the strategy provided by its contained Strategy object. That is, the Context wants a “do-nothing” strategy.
- One way to do this is to have the Context assign a null reference to its contained Strategy object. In this case, the Context must always check for this null value:

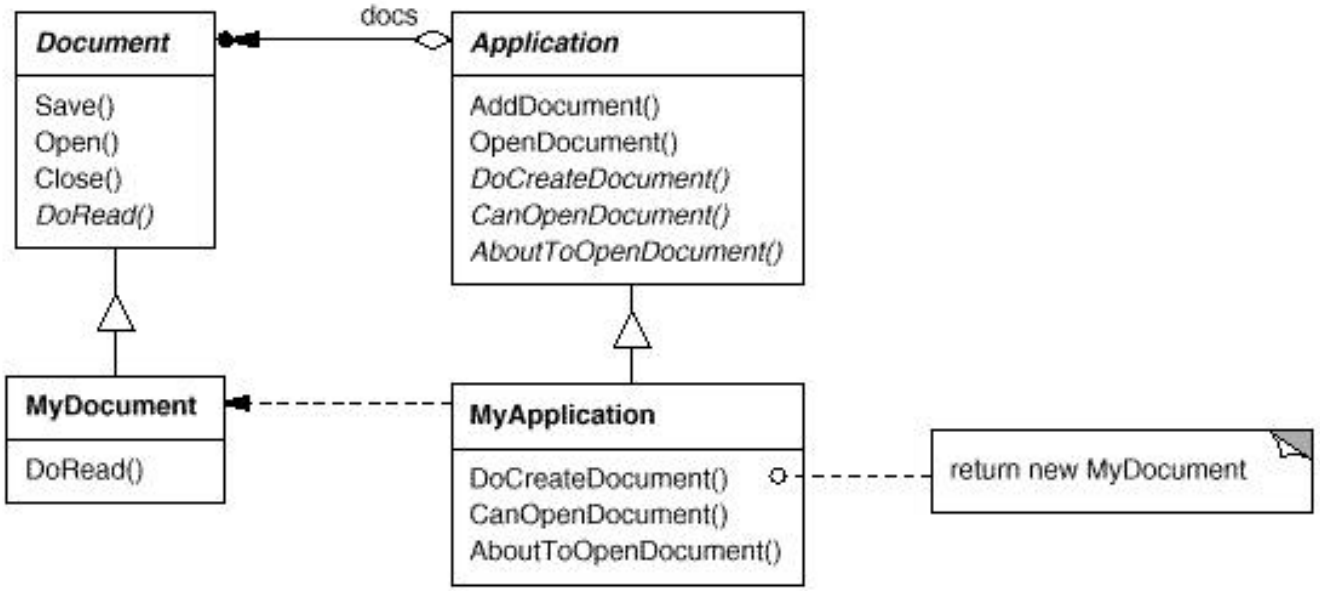
```
if (strategy != null)
    strategy.doOperation();
```

- Another way to accomplish this is to actually have a “do-nothing” strategy class which implements all the required operations of a Strategy object, but these operations do nothing. Now clients do not have to distinguish between strategy objects which actually do something useful and those that do nothing.
- Using a “do-nothing” object for this purpose is known as the Null Object Pattern


# Template method

# Template method

- Intent
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Motivation
  - Sometimes you want to specify the order of operations that a method uses, but allow subclasses to provide their own implementations of some of these operations.



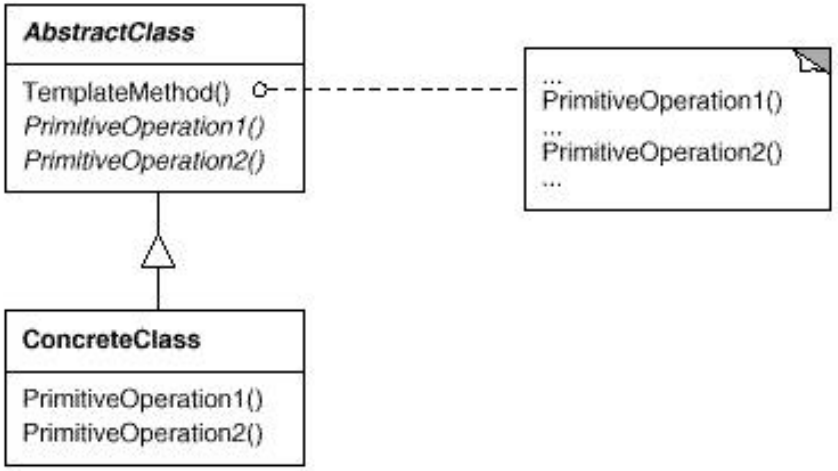
# Template method

- Example

```
public class PlainTextDocument {

  ...

  public void printPage (Page page) {

    printPlainTextHeader();    // Unique to PlainTextDocument
    System.out.println(page.body());
    printPlainTextFooter();    // Unique to PlainTextDocument

  }

  ...

}
```

# Template method

- Use the Template Method pattern:
  - To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary
  - To localize common behavior among subclasses and place it in a common class (in this case, a superclass) to avoid code duplication. This is a classic example of ”code refactoring.”
  - To control how subclasses extend superclass operations. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.
- ***The Template Method is a fundamental technique for code reuse.***



# Template method

- Operations which must be overridden by a subclass should be made abstract
- If the template method itself should not be overridden by a subclass, it should be made final
- To allow a subclass to insert code at a specific spot in the operation of the algorithm, insert “hook” operations into the template method. These hook operations may do nothing by default.
- Try to minimize the number of operations that a subclass must override, otherwise using the template method becomes tedious for the developer
- In a template method, the parent class calls the operations of a subclass and not the other way around. This is an inverted control structure that’s sometimes referred to as "the Hollywood principle," as in, "Don't call us, we'll call you".

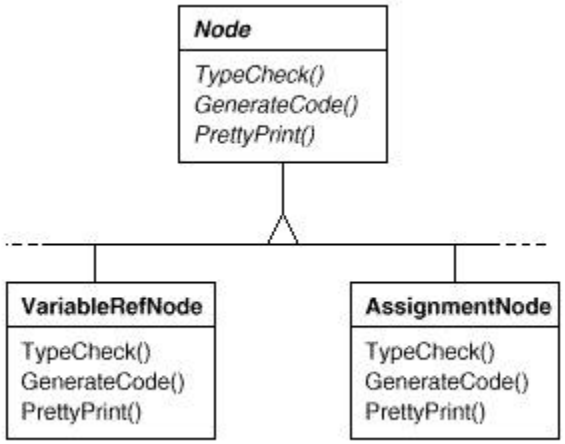

# Visitor method

# Visitor method

- Intent
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Motivation
  - Consider a compiler that parses a program and represents the parsed program as an abstract syntax tree (AST). The AST has many different kinds of nodes, such as Assignment , Variable Reference, and Arithmetic Expression nodes.
  - Operations that one would like to perform on the AST include:
    - Checking that all variables are defined
    - Checking for variables being assigned before they are used
    - Type checking
    - Code generation
    - Pretty printing/formatting

# Visitor method

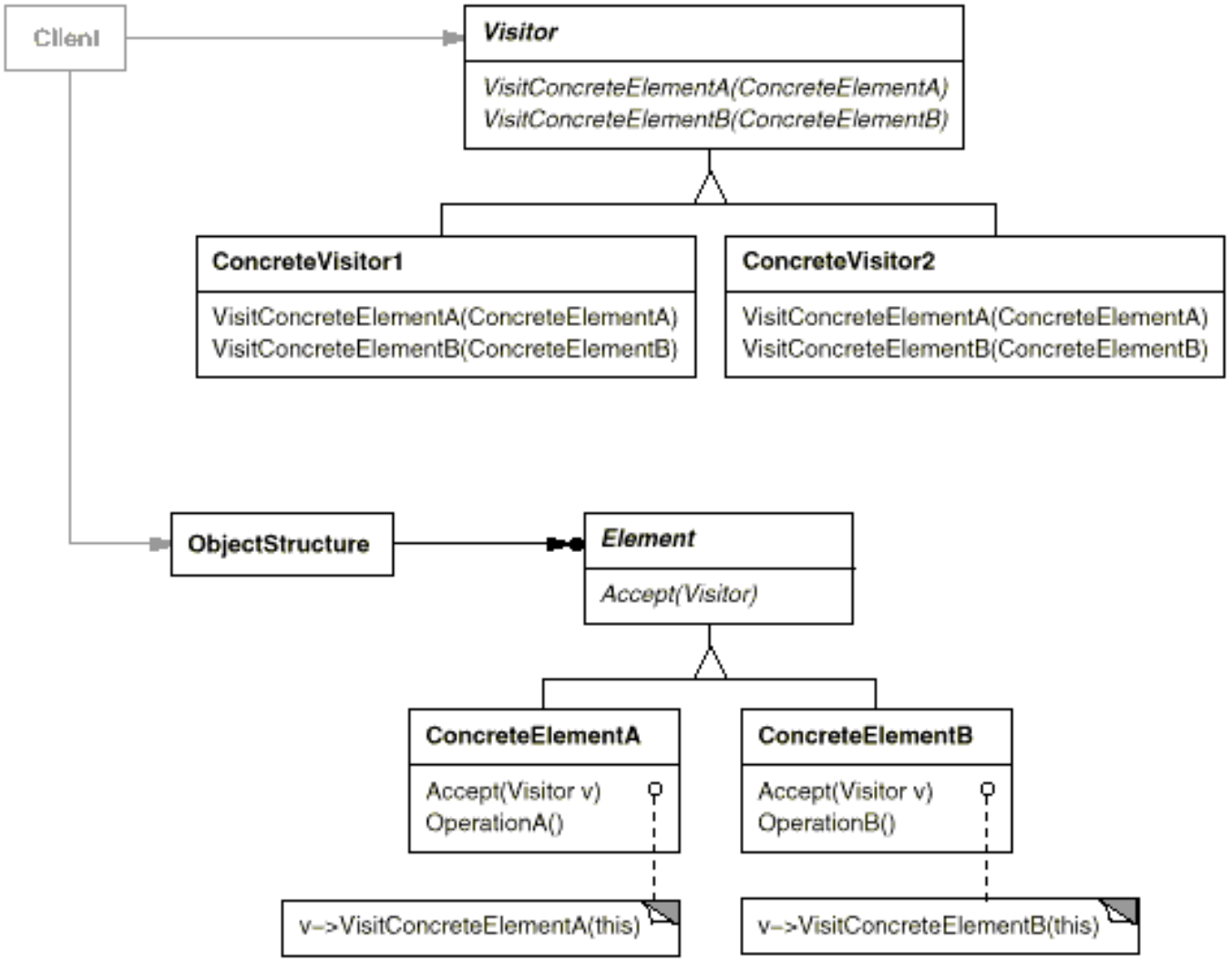
- Problems with this approach:
  - Adding new operations requires changes to all of the node classes
  - It can be confusing to have such a diverse set of operations in each node class. For example, mixing type-checking code with pretty-printing code can be hard to understand and maintain.
- Another solution is to encapsulate a desired operation in a separate object, called a visitor. The visitor object then traverses the elements of the tree. When a tree node "accepts" the visitor, it invokes a method on the visitor that includes the node type as an argument. The visitor will then execute
- the operation for that node - the operation that used to be in the node class.



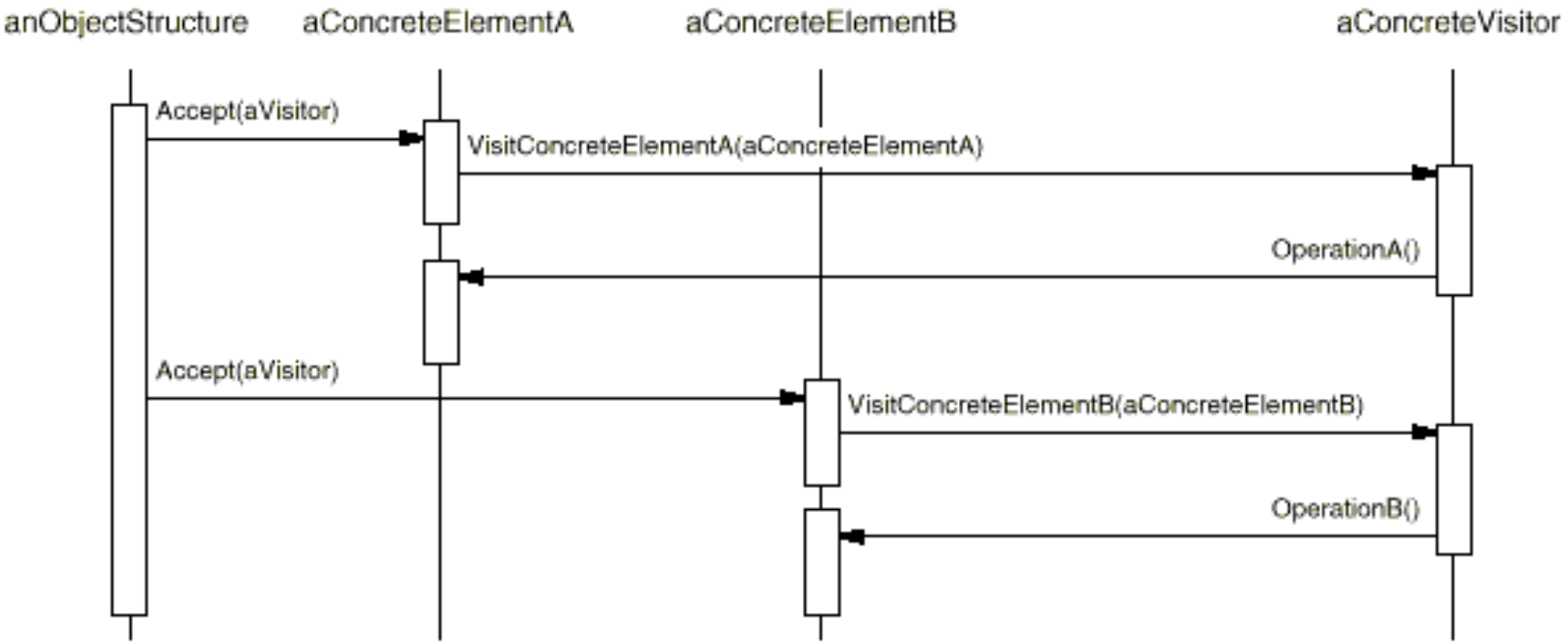
# Visitor method

- Use the Visitor pattern in any of the following situations:
  - When many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations
  - When the classes defining the object structure rarely change, but you often want to define new operations over the structure. (If the object structure classes change often, then it's probably better to define the operations in those classes.)
- When an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes

# Visitor method



# Visitor method



# Visitor method

- Benefits
  - Adding new operations is easy
  - Related behavior isn't spread over the classes defining the object structure; it’s localized in a visitor. Unrelated sets of behavior are partitioned in their own visitor subclasses.
  - Visitors can accumulate state as they visit each element in the object structure. Without a visitor, this state would have to be passed as extra arguments to the operations that perform the traversal.
- Liabilities
  - Adding new ConcreteElement classes is hard. Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class.
  - The ConcreteElement interface must be powerful enough to let visitors do their job. You may be forced to provide public operations that access an element’s internal state, which may compromise its encapsulation.

# Visitor method – Composite

```
public abstract class Component {
  protected String name;

  public Component(String name) {this.name = name;}

  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  public abstract double getPrice();

  public abstract void accept(ComponentVisitor v);
}
```

```
public class Widget extends Component {
  protected double price;

  public Widget(String name, double price) {
    super(name);
    this.price = price;
  }

  public void setPrice(double price) { this.price = price; }
  public double getPrice() { return price; }

  public void accept (ComponentVisitor v) { v.visit(this); }
}
```

# Visitor method – Composite

```
public class WidgetAssembly extends Component
  protected Vector components;

  public WidgetAssembly (String name) {
    super(name);
    components = new Vector();
  }

  public void addComponent (Component c) {
    components.addElement(c);
  public void removeComponent (Component c) {
    components.removeElement(c);
  }

  public double getPrice() {

    double totalPrice = 0.0;

    Enumeration e = components.elements();
    while (e.hasMoreElements()) {
      totalPrice += ((Component) e.nextElement()).getPrice();
    }
    return totalPrice;
  }

  public void accept (ComponentVisitor v) { v.visit(this); }
}
```

# Visitor method – Composite

```
public abstract class ComponentVisitor {

  public abstract void visit(Widget w);
  public abstract void visit(WidgetAssembly wa);

}
```

```
public class SimpleVisitor extends ComponentVisitor {

  public SimpleVisitor() {}

  public void visit (Widget w) {
    System.out.println("Visiting a Widget");
  }

  public void visit (WidgetAssembly wa) {
    System.out.println("Visiting a WidgetAssembly");
  }

}
```

# Visitor method – Composite

```
public class PriceVisitor extends ComponentVisitor {
  private double maxPrice;

  public PriceVisitor(double maxPrice) {
    this.maxPrice = maxPrice; }

  public void visit (Widget w) {
    double price = w.getPrice();
    if (price > maxPrice)
      System.out.println("Don't Buy! Widget price of " +
        price + " exceeds maximum price (" + maxPrice + ").");
    else
      System.out.println("Buy! Widget price of " + price +
        " is less than maximum price (" + maxPrice + ").");
  }
  public void visit (WidgetAssembly wa) {
    double price = wa.getPrice();
    if (price > maxPrice)
      System.out.println("Don't Buy! WidgetAssembly price of " +
        price + " exceeds maximum price (" + maxPrice + ").");
    else
      System.out.println("Buy! WidgetAssembly price of " +
        price + " is less than maximum price (" +
        maxPrice + ").");
  }

}
```

# Visitor method – Composite

```
public class VisitorTest {

  public static void main (String[] args) {

    // Create some widgets.
    Widget w1 = new Widget("Widget1", 10.00);
    Widget w2 = new Widget("Widget2", 20.00);
    Widget w3 = new Widget("Widget3", 30.00);

    // Add then to a widget assembly.
    WidgetAssembly wa = new WidgetAssembly("Chassis");
    wa.addComponent(w1);
    wa.addComponent(w2);
    wa.addComponent(w3);

    // Visit some nodes with a SimpleVisitor.
    SimpleVisitor sv = new SimpleVisitor();
    w1.accept(sv);
    w2.accept(sv);
    w3.accept(sv);
    wa.accept(sv);

    // Visit some nodes with a PriceVisitor.
    PriceVisitor pv = new PriceVisitor(25.00);
    w1.accept(pv);
    w2.accept(pv);
    w3.accept(pv);
    wa.accept(pv);

  }

}
```

```
Visiting a Widget
Visiting a Widget
Visiting a Widget
Visiting a WidgetAssembly
Buy! Widget price of 10.0 is less than maximum price (25.0).
Buy! Widget price of 20.0 is less than maximum price (25.0).
Don't Buy! Widget price of 30.0 exceeds maximum price (25.0).
Don't Buy! WidgetAssembly price of 60.0 exceeds maximum price
```