Design Patterns VI

Ergude Bao

Beijing Jiaotong University

Content

- Behavioral patterns II

Behavioral Patterns

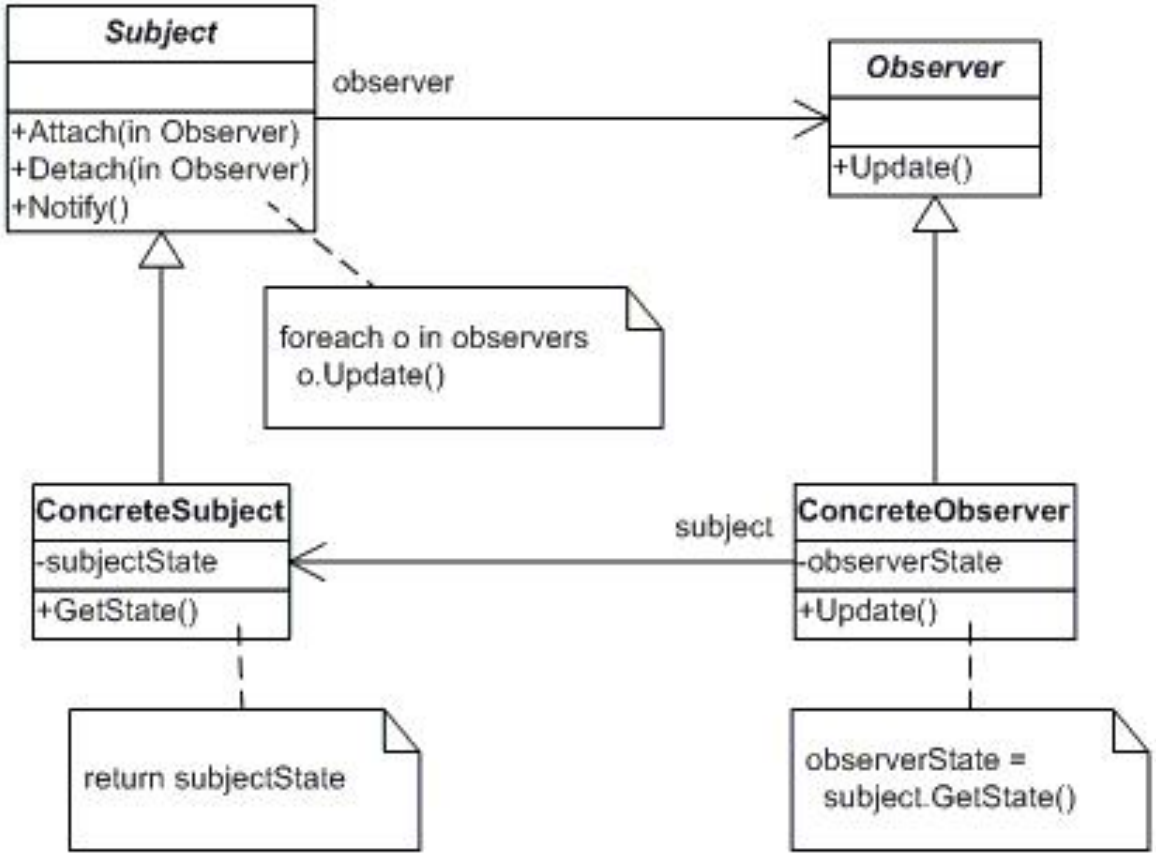
- Observer Pattern
- State Pattern
- Strategy Pattern
- Template Pattern
- Visitor Pattern

Observer Pattern

Observer Pattern

- Motivation
  - We want to reflect the change of one object in other objects
- Solution
  - Define a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically

Observer Pattern

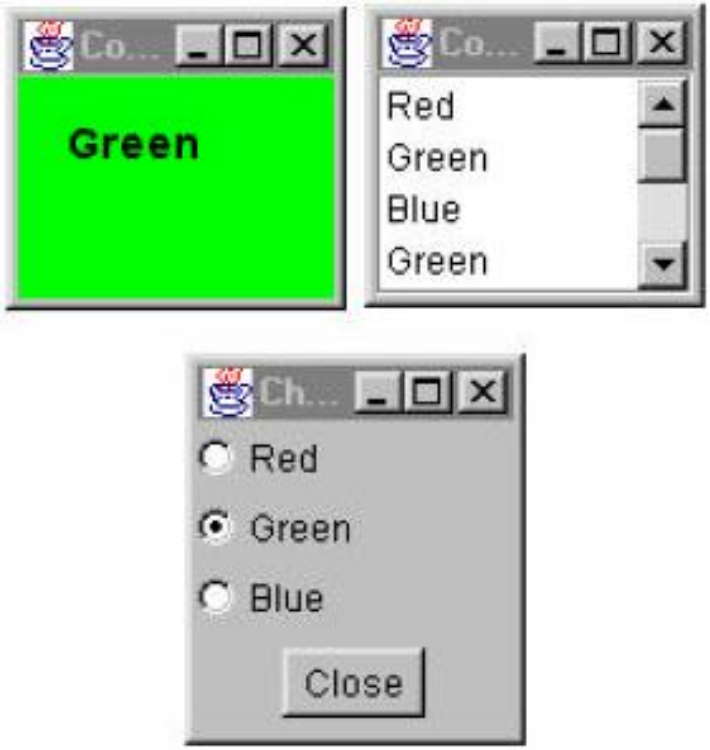


Participants

- Subject
  - Defines an interface for attaching, detaching and tracking Observer objects
- Observer
  - Defines an interface for notifications of Subject objects' update
- ConcreteSubject
  - Stores state of interest to ConcreteObserver objects
  - Sends a notification to ConcreteObserver when its state changes
- ConcreteObserver
  - Stores state that should stay consistent with the ConcreteSubject's
  - Receives the notification from ConcreteSubject to keep its state consistent with it

Example

- Two observers
  - One displays the color and the color's name
  - The other adds the current color to a list box



Observers in Java

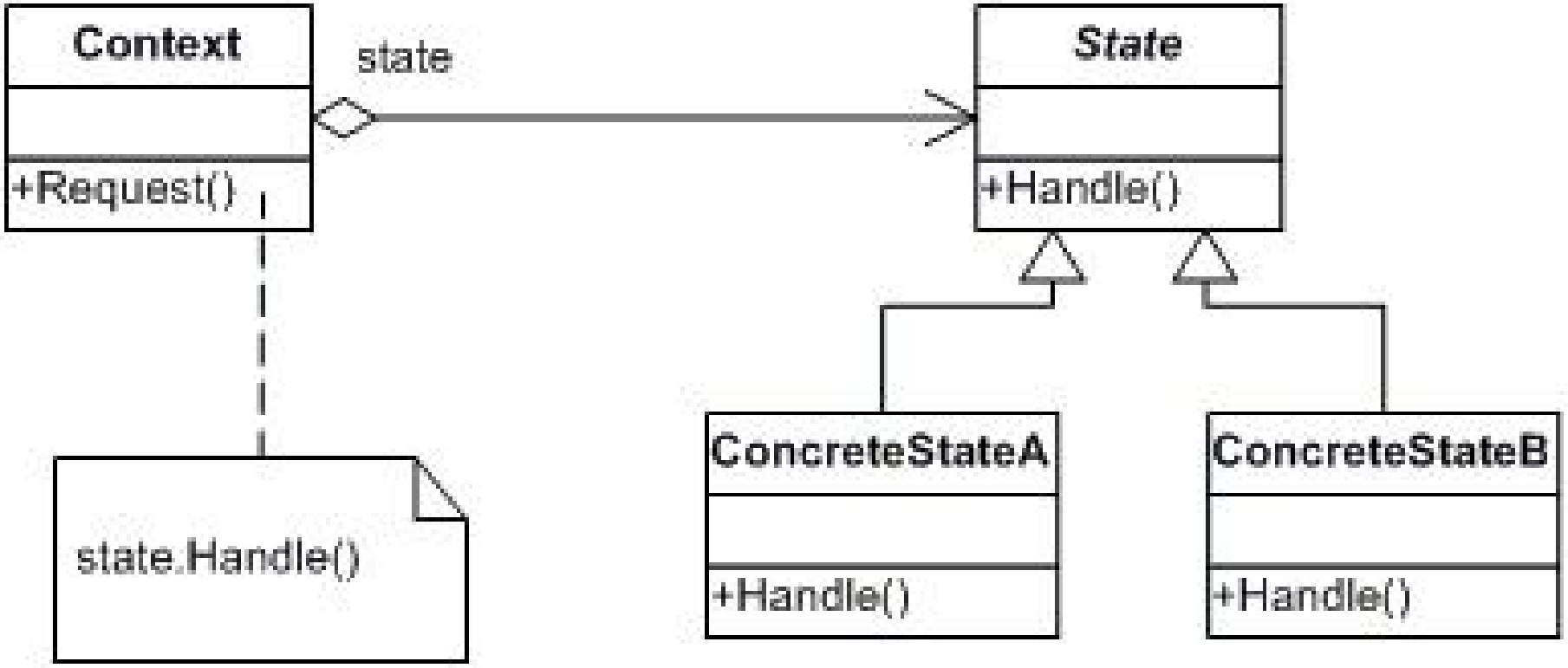
- Java provides the java.util.Observerable and java.util.Observer classes as built-in support for the observer pattern
  - The java.util.Observable class is the base Subject class
  - The java.util.Observer interface is the Observer interface
- Java's GUI event model is based on the observer pattern
  - Event source is a ConcreteSubject
  - Event listener is a ConcreteObserver

State Pattern

State Pattern

- Motivation
  - We want to perform slightly different operations in a class based on its state
  - We do not want to switch among these operations
- Solution
  - Put the corresponding operations of each state in one separate class
  - Switch these operations along with the switch of states

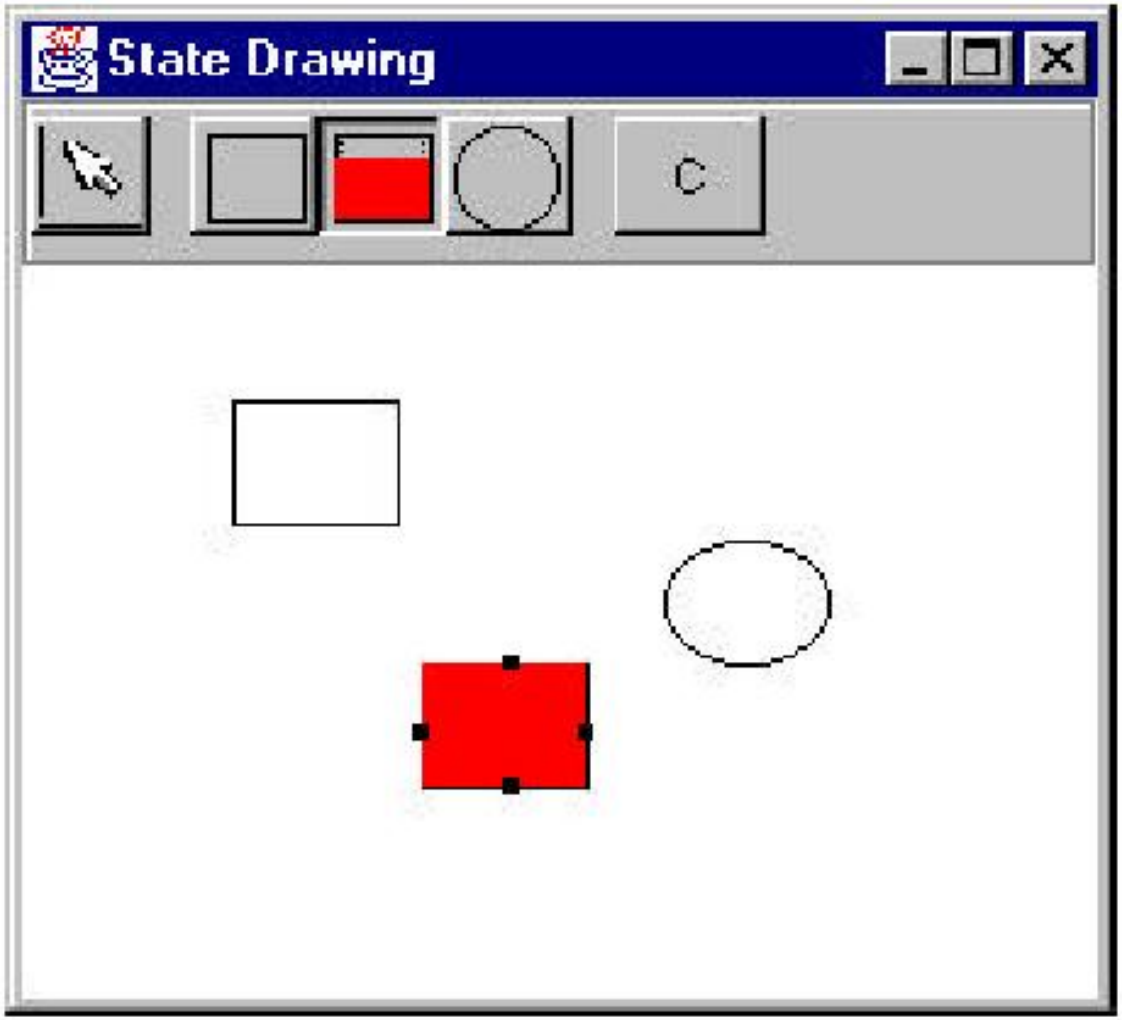
State Pattern



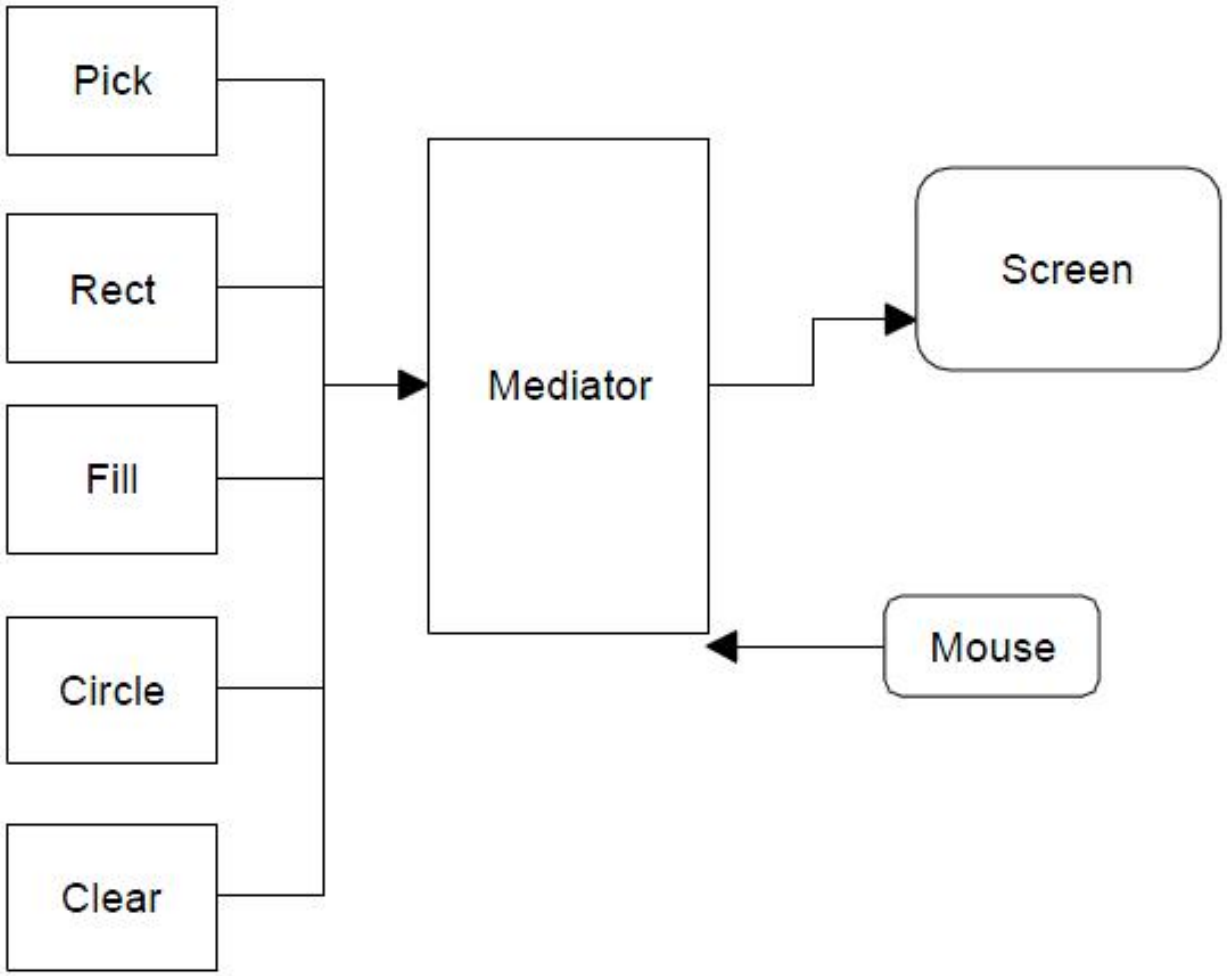
Participants

- Context
  - Maintains an object of ConcreteState corresponding to the current state
- State
  - Defines an interface for the ConcreteState
- ConcreteState
  - Implements the State interface with operations corresponding to the state of Context

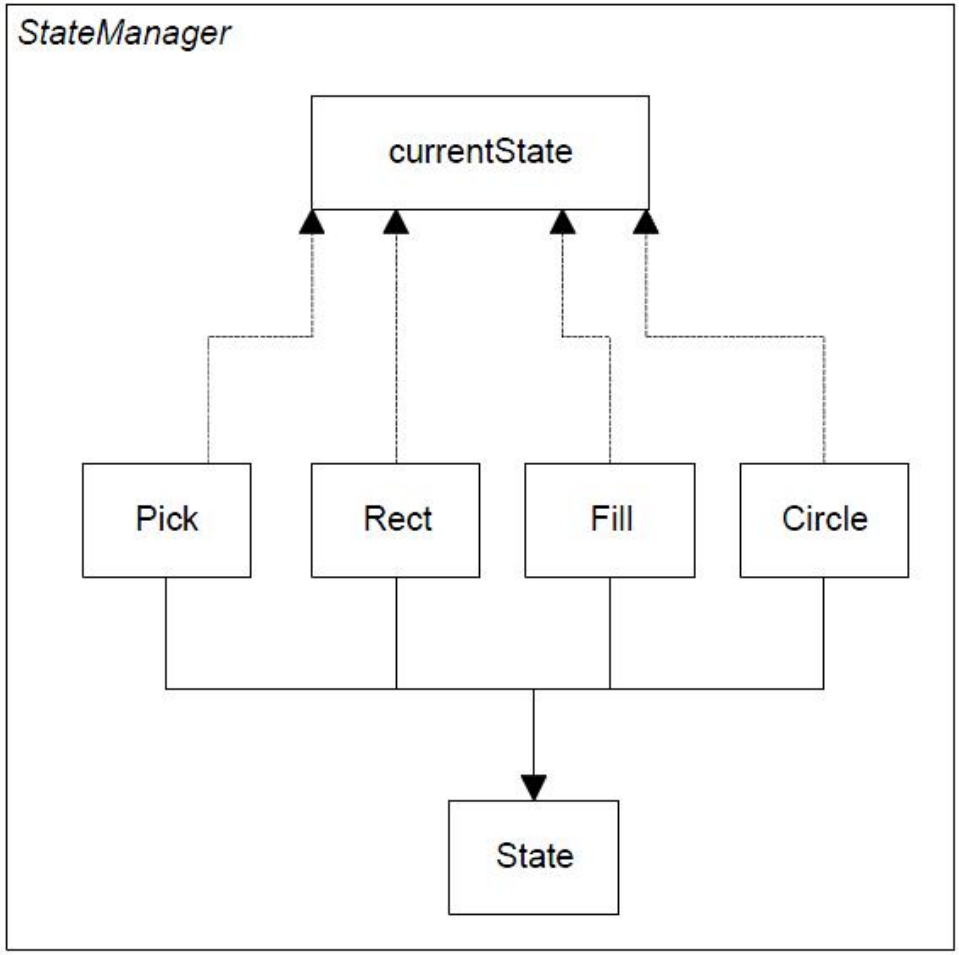
Example



Example



Example

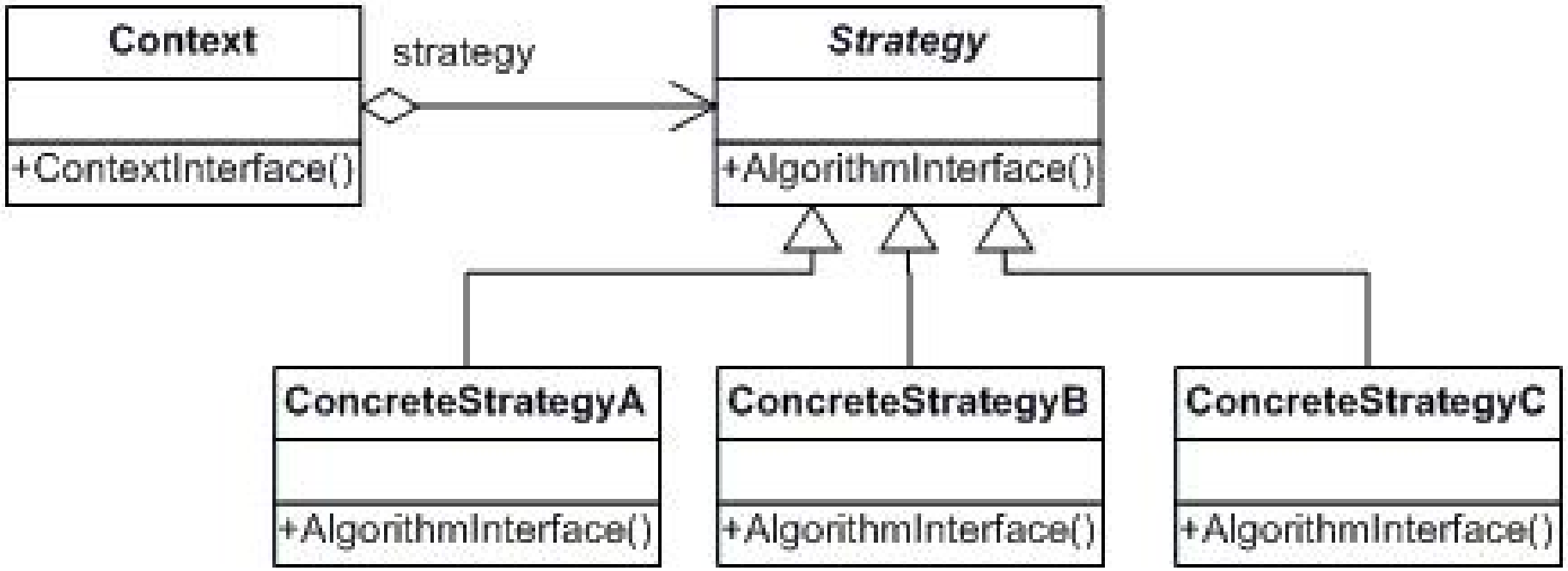


Strategy pattern

Strategy Pattern

- Motivation
  - We want to process input data to a class with different algorithms depending on the data
- Solution
  - Encapsulate the algorithms in separate classes and select one of these classes for each kind of data

Strategy Pattern



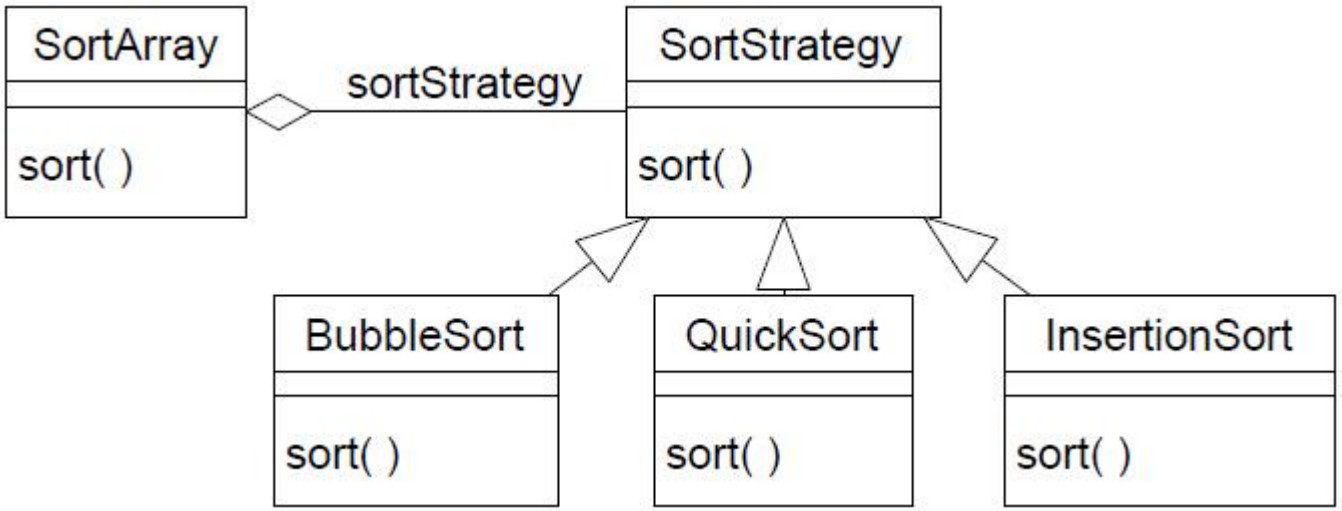
Participants

- Strategy
  - Defines an interface common to all supported algorithms
- ConcreteStrategy
  - Implements an algorithm using the Strategy interface
- Context
  - Maintains a reference to a ConcreteStrategy object
  - May define an interface for the ConcreteStrategy object to access its data

Example I

- Situation: A class wants to decide at run-time what algorithm it should use to sort an array. Many different sort algorithms are already available
- Solution: Encapsulate the different sort algorithms using the Strategy pattern

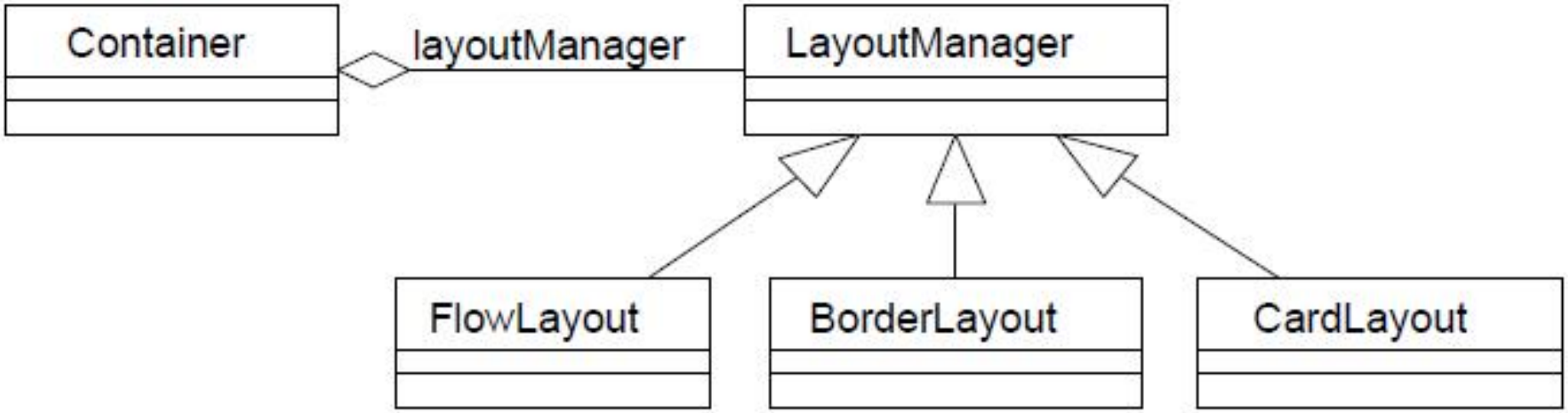
Example I


SortArray
sort( )
sortStrategy
SortStrategy
sort( )
BubbleSort
sort( )
QuickSort
sort( )
InsertionSort
sort( )

Example II

- Situation: A GUI container object wants to decide at run-time what strategy it should use to layout the GUI components it contains. Many different layout strategies are already available
- Solution: Encapsulate the different layout strategies using the Strategy pattern. This is what the Java AWT does with its LayoutManagers!

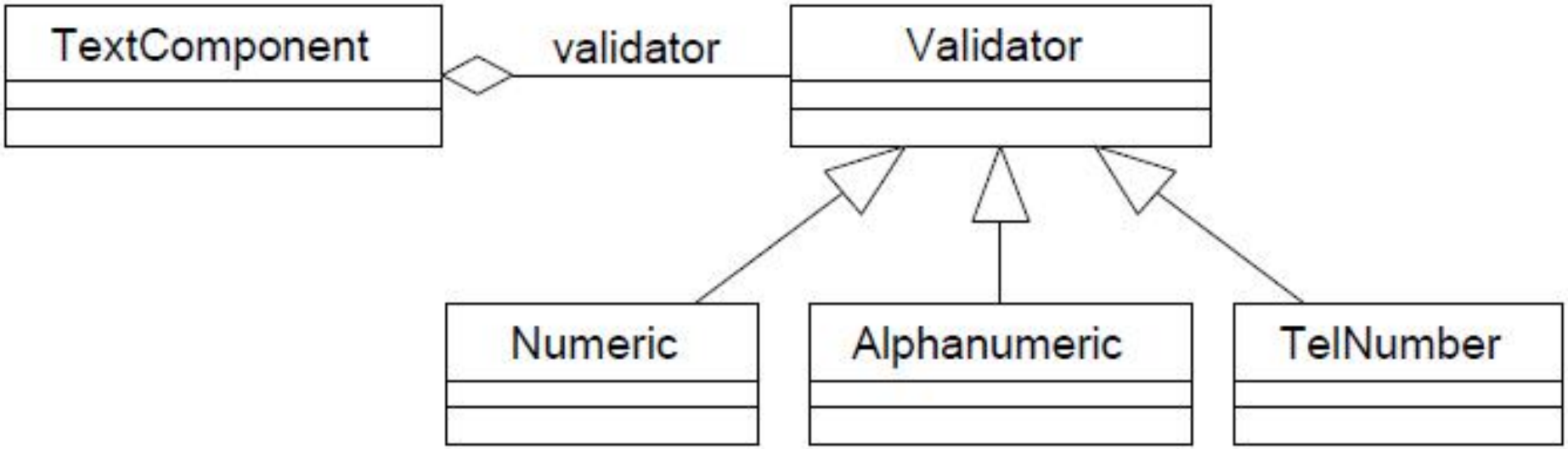
Example II


Container
layoutManager
LayoutManager
FlowLayout
BorderLayout
CardLayout

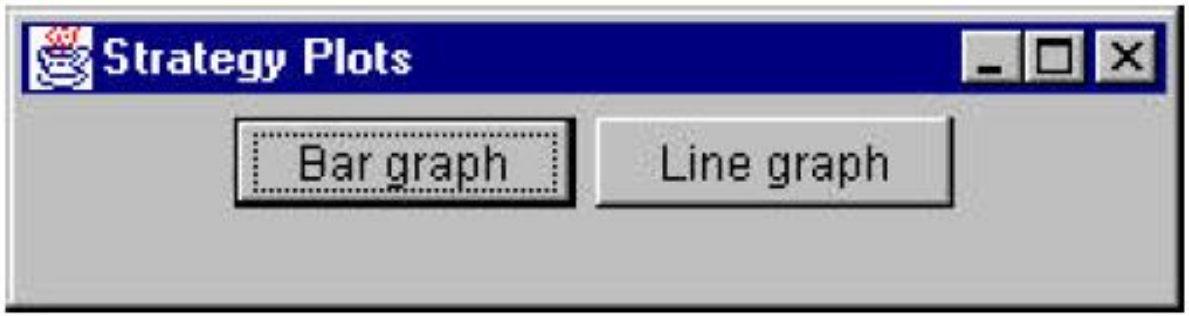
Example III

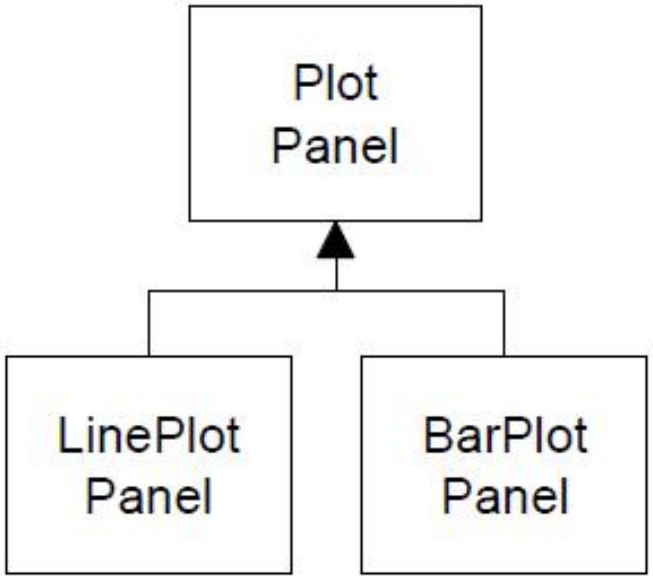
- Situation: A GUI text component object wants to decide at runtime what strategy it should use to validate user input. Many different validation strategies are possible for numeric fields, alphanumeric fields, telephone-number fields, etc.
- Solution: Encapsulate the different input validation strategies using the Strategy pattern

Example III

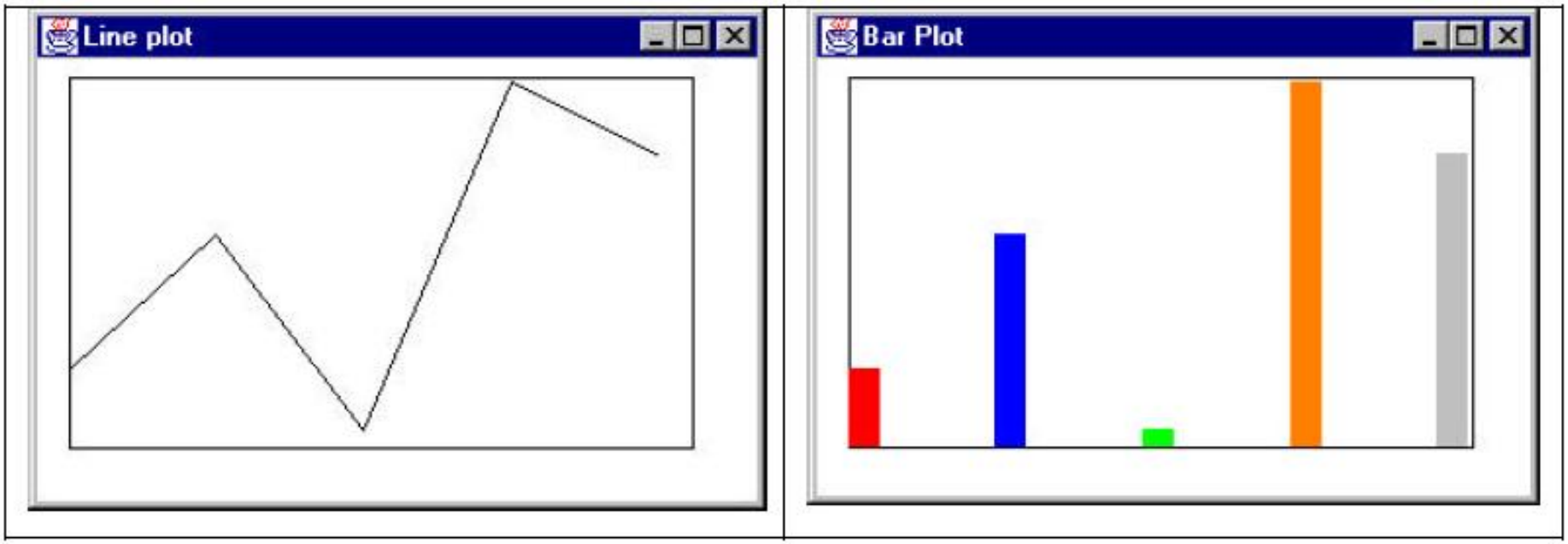

TextComponent
validator
Validator
Numeric
Alphanumeric
TelNumber

Example IV


Strategy Plots
Bar graph
Line graph


Plot Panel
LinePlot Panel
BarPlot Panel

Example IV



Strategy Pattern vs. State Pattern

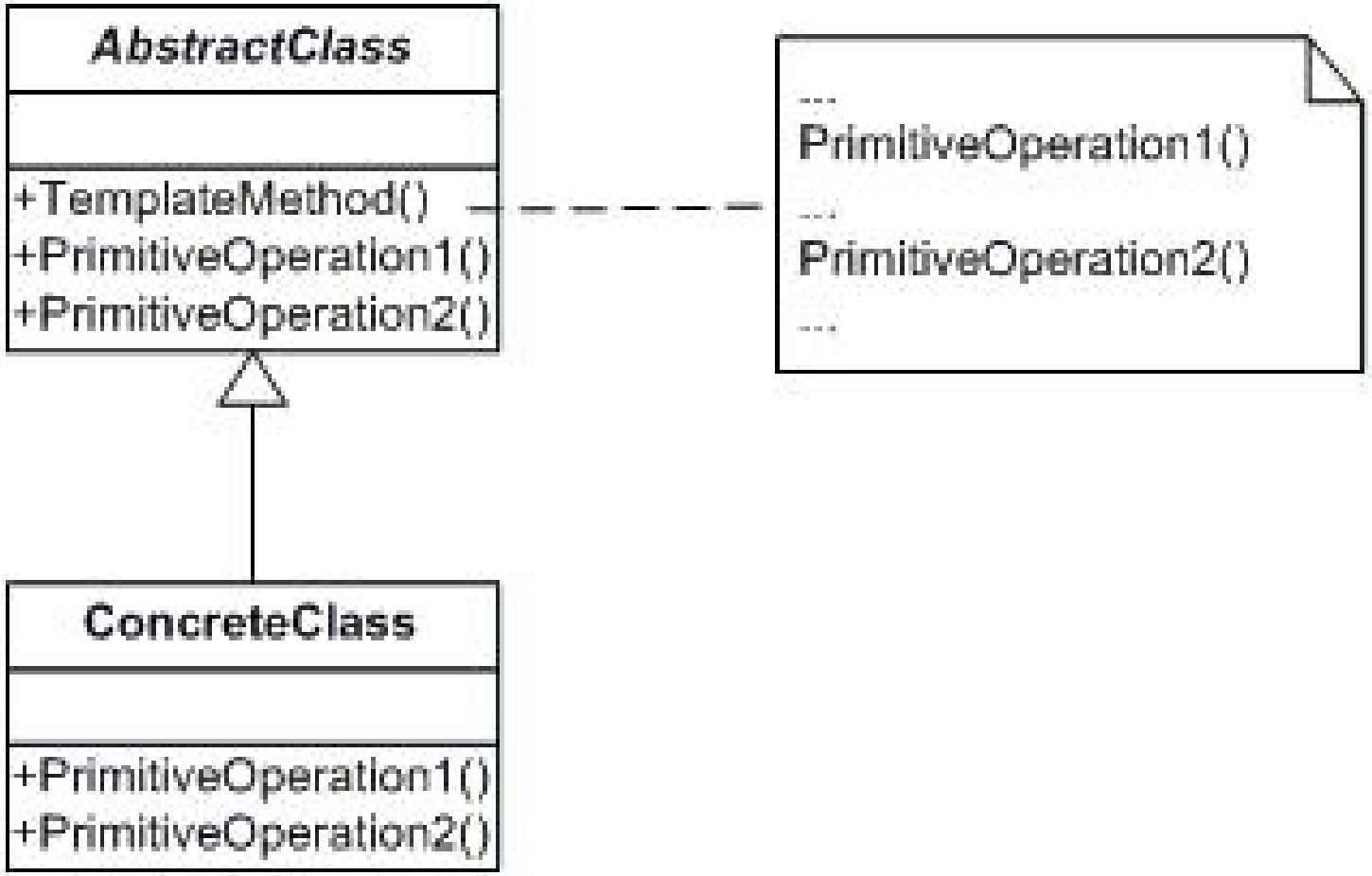
- Similarity: both put operation sets in separate classes to avoid a large number of switches among the sets
- Difference: strategy put alternative operation sets for the same purpose in the classes, while state put state dependent operation sets in the classes

Template Pattern

Template Pattern

- Motivation
  - We want to fix the order of operations for a method, and allow flexible implementations of some operations
- Solution
  - Define the skeleton of an algorithm in a base class, and defer some steps to subclasses

Template Pattern



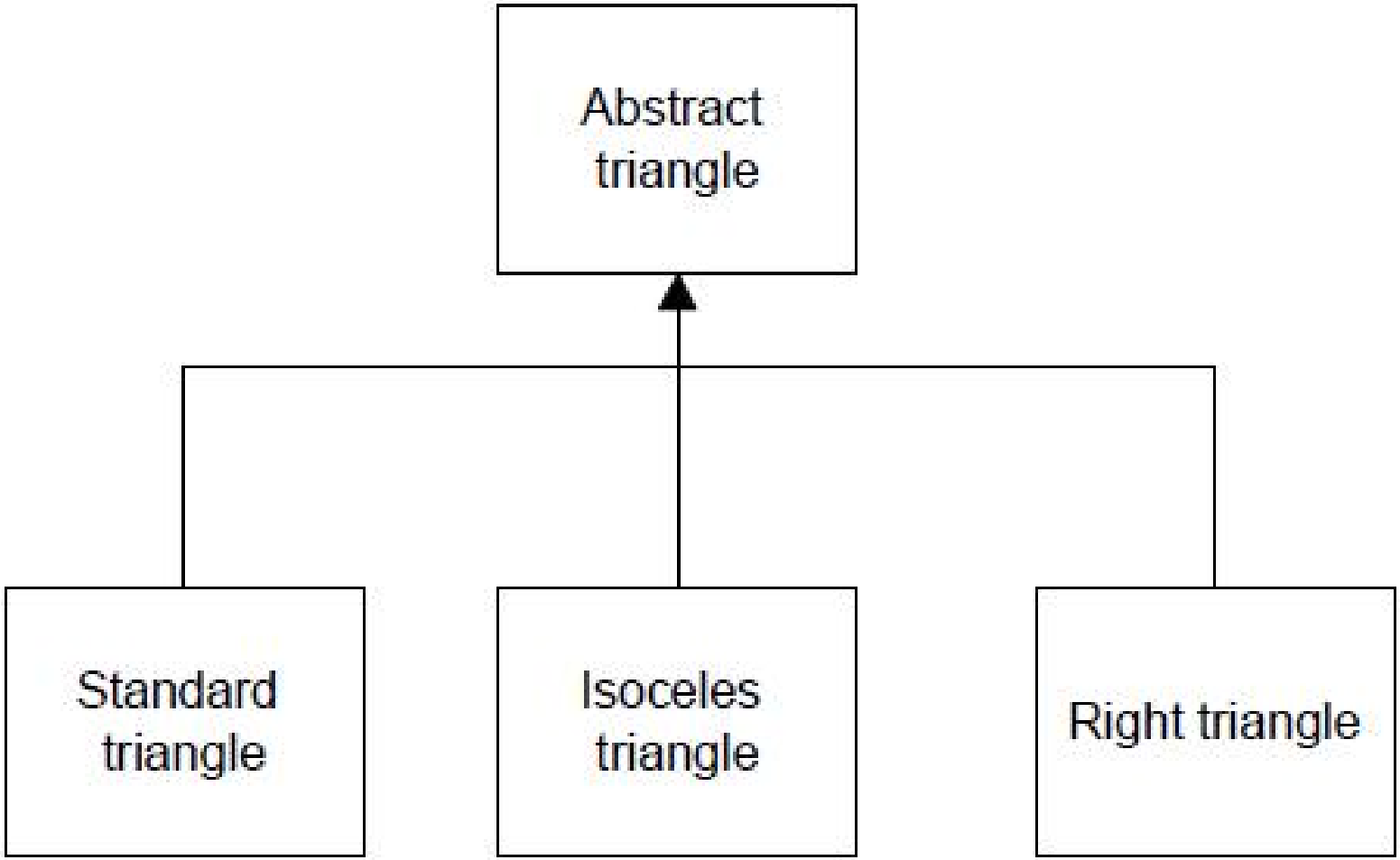
Participants

- AbstractClass
  - Implements a template method defining the skeleton of the algorithm with primitive operations and other operations
  - Defines abstract primitive operations for ConcreteClass to implement
- ConcreteClass
  - Implements the primitive operations to carry out subclass specific steps of the algorithm

Methods in AbstractClass

- Concrete methods
  - Complete methods that carry out some basic function that all subclasses will want to use
- Hook methods
  - Methods that contain a default implementation of some operations, but may be overridden in subclasses
- Template methods
  - Methods that is not intended to be overridden, but describe an algorithm with actual implementation of its details in subclasses
- Abstract methods
  - Methods that are not filled in at all and must be implemented in subclasses

Example

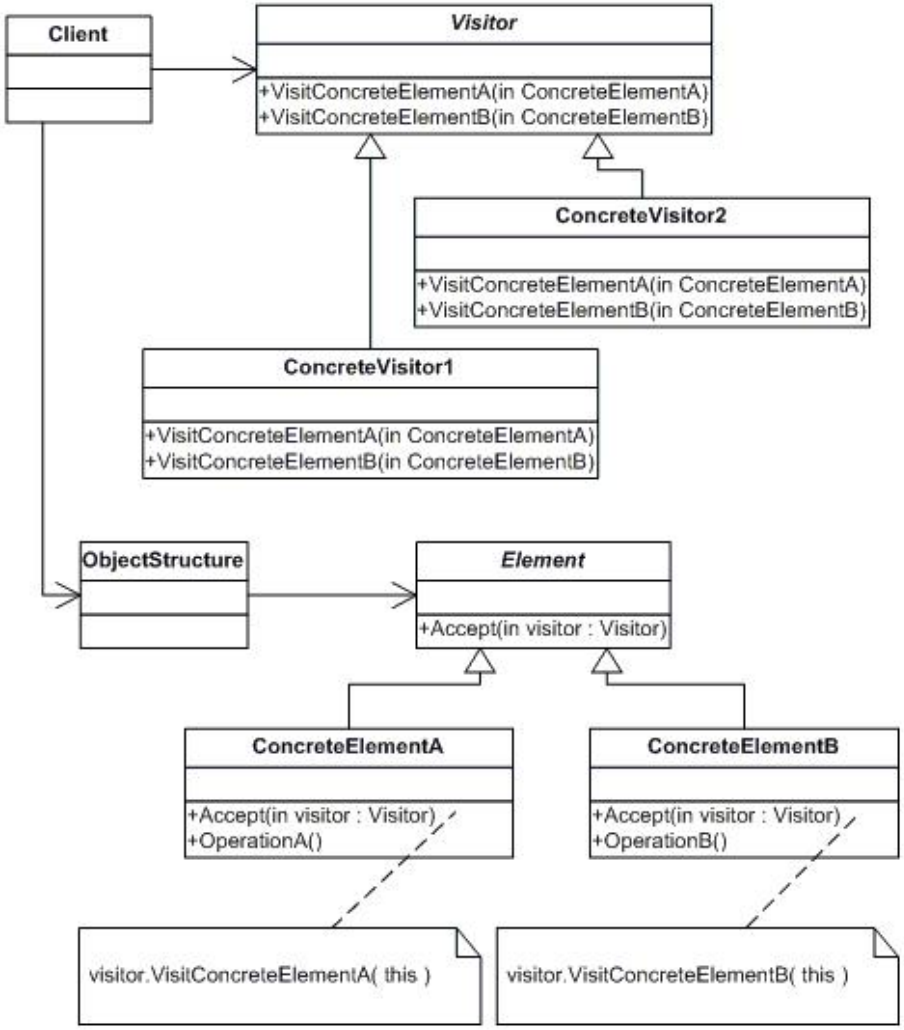

Abstract triangle
Standard triangle
Isoceles triangle
Right triangle

Visitor Pattern

Visitor Pattern

- Motivation
  - When similar operations have to be performed on several different classes, we want to separate the operations from the classes
- Solution
  - Create a separate class for each type of similar operations to visit resources in the original class and make operations

Visitor Pattern

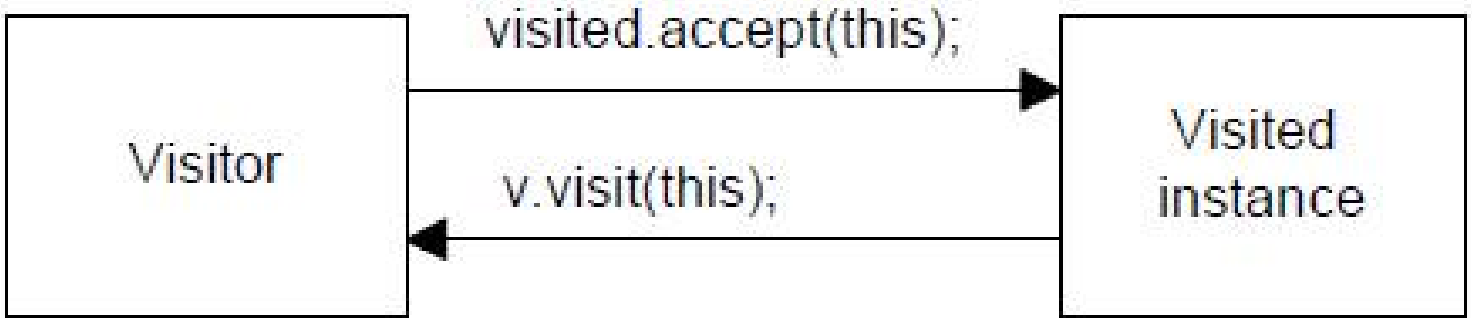


Participants

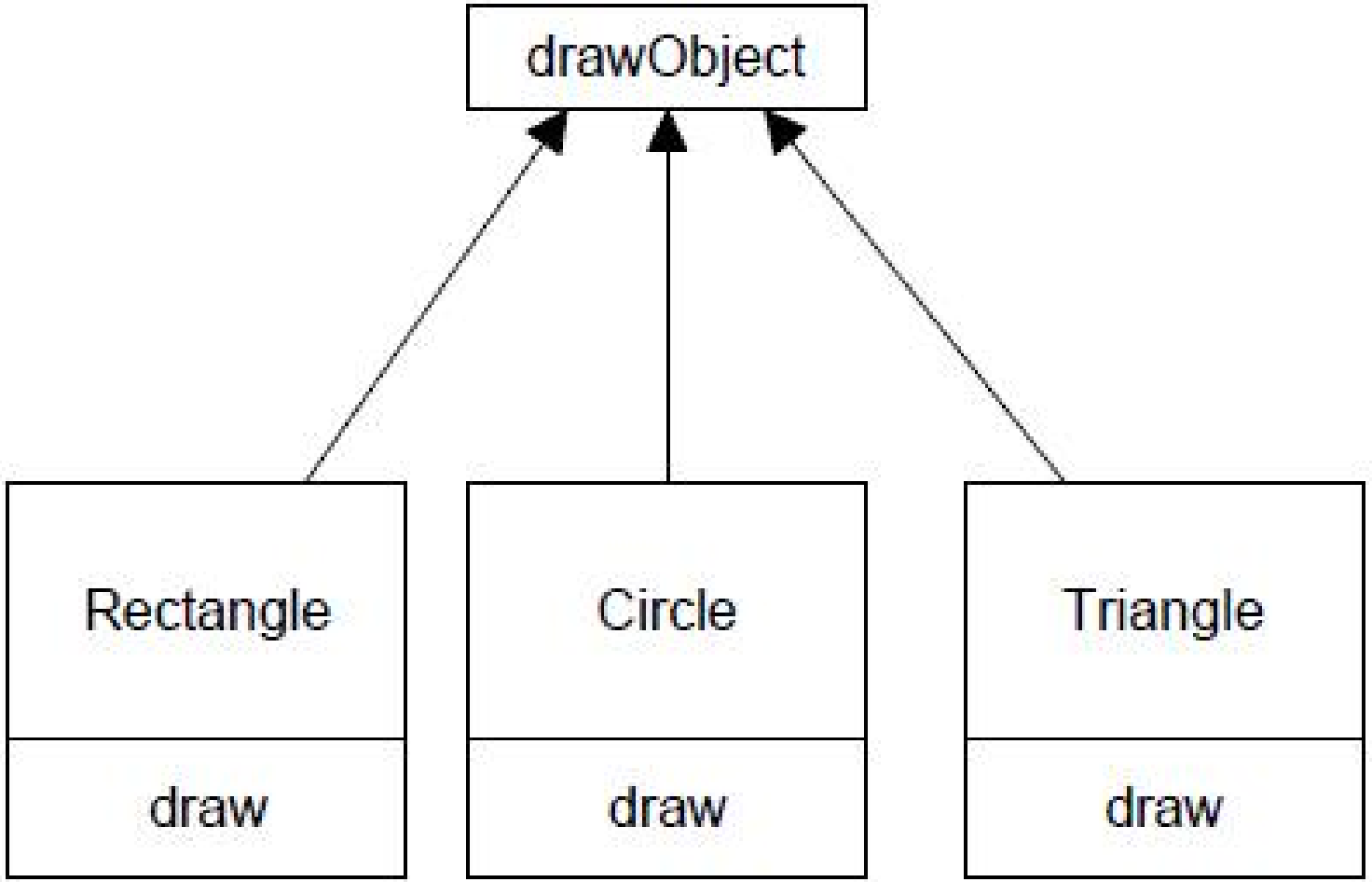
- Visitor
  - Declares a visit operation for each class of ConcreteElement in ObjectStructure
- ConcreteVisitor
  - Implements each operation declared by visitor
- Element
  - Defines an Accept operation that takes a visitor as an argument
- ConcreteElement
  - Implements an Accept operation that takes a visitor as an argument
- ObjectStructure
  - Provides a high-level interface to allow the visitor to visit its elements

Double Dispatching

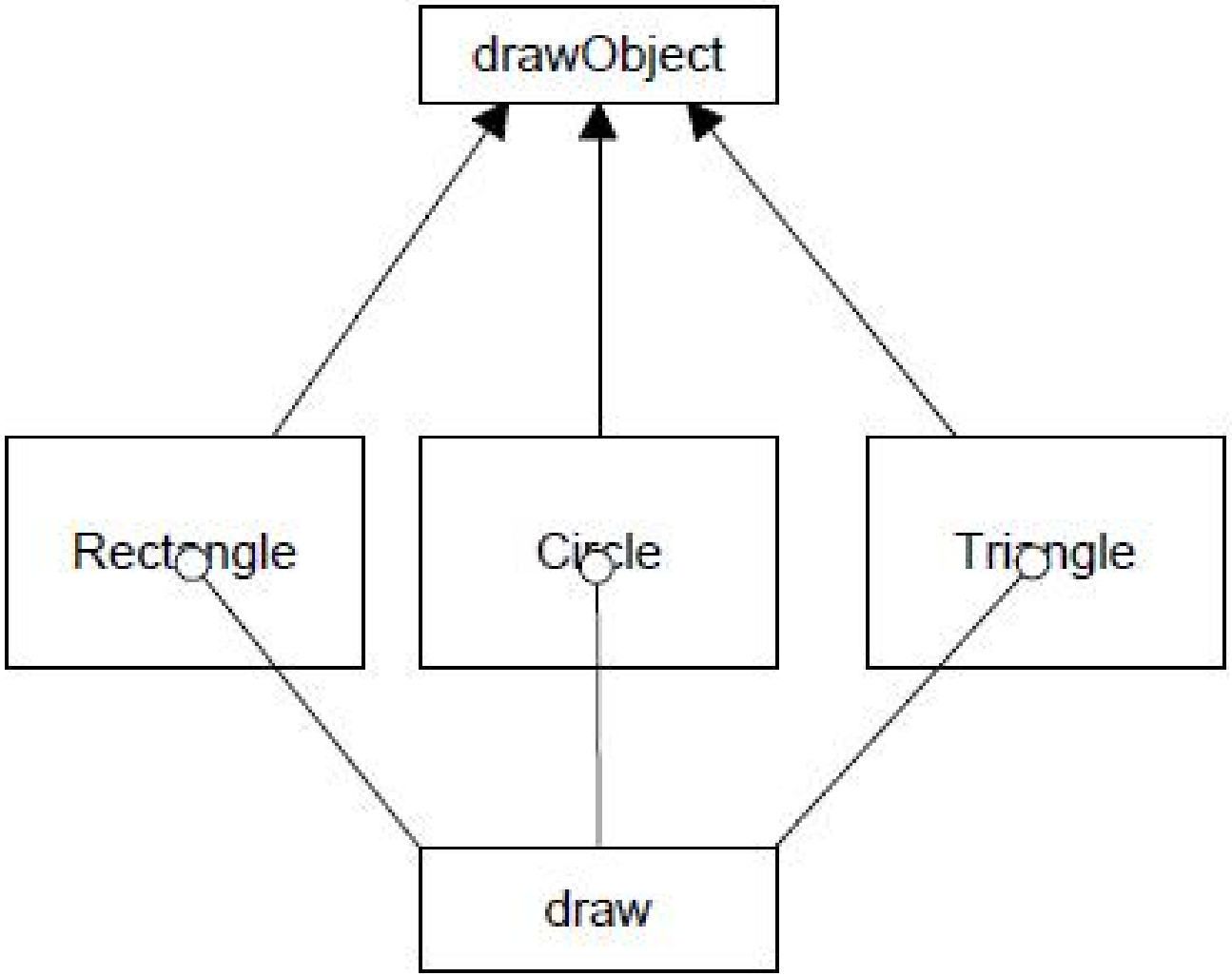
- Dispatching a method twice for the Visitor to work
  - The element object accepts an visitor object, by passing the visitor object to the element object
  - The visitor object visits the element object, by passing the element object to the visitor object



Example



Example



Advantages

- Related behavior is not spread over the classes defining the object structure; it is localized in a visitor
- It is very easy to add new visitors as long as the structure remains unchanged
- 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

Disadvantages

- 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
  - Public operations may be needed to access an element's internal state, which may compromise encapsulation

Visitor vs. Iterator

- Similarity: extract common things of several classes into one class
- Difference: visitor can do various operations but needs careful binding with the original classes, while iterator traverses aggregate objects only but can bind easily