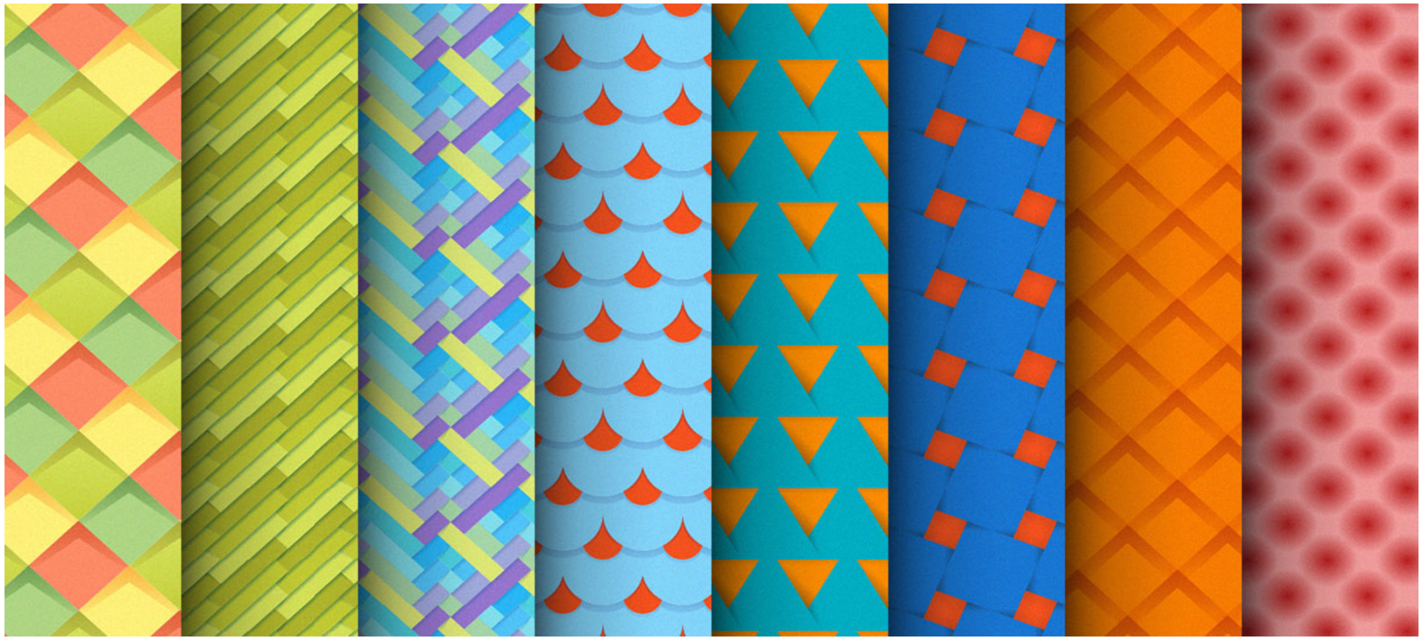


DESIGN PATTERNS

COURSE 7

PREVIOUS COURSE

- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

CURRENT COURSE

- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

CURRENT COURSE

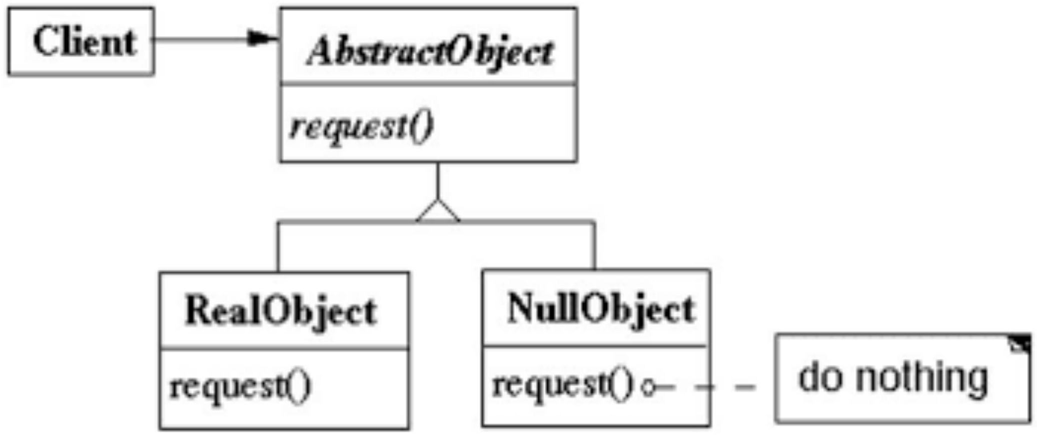
- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

NULL OBJECT

- **Intent**
  - Provide an object as a surrogate for the lack of an object of a given type.
  - The Null Object Pattern provides intelligent do nothing behavior, hiding the details from its collaborators.

- **Problem**
  - Given that an object reference may be optionally null, and that the result of a null check is to do nothing or use some default value, how can the absence of an object — the presence of a null reference — be treated transparently?

NULL OBJECT. STRUCTURE



- **AbstractObject**
  - Defines abstract primitive operations that concrete implementations have to define.
- **RealObject**
  - A real implementation of the AbstractClass performing some real actions.
- **NullObject**
  - A implementation which do nothing of the abstract class, in order to provide a non-null object to the client.
- **Client**
  - The client gets an implementation of the abstract class and uses it. It doesn't really care if the implementation is a null object or an real object since both of them are used in the same way.

NULL OBJECT. EXAMPLE

- **Returning a list of objects**

```
public List<String> returnCollection() {
  //remainder omitted
  if (/*some condition*/) {
    return null;
  } else {
    // return collection
  }
}
```

- Usage of the method
  - `if (obj.returnCollection().size() > 0) {}`
  - There exists cases in which above line could throw an exception

NULL OBJECT. EXAMPLE

- **Returning a list of objects**

```
public List<String> returnCollection() {
  //remainder omitted
  if (/*some condition*/) {
    return Collections.emptyList();
  } else {
    // return collection
  }
}
```

- Usage of the method
  - `if (obj.returnCollection().size() > 0) {}`
  - Prevents caller of the function to get NPE while trying to do things like in the code above

NULL OBJECT. EXAMPLE

- **An application that calculates discount for an order**
  - Order class

```
class Order
{
   public String productName;
   public double productCost;
   private IDiscount discount = null;
   public double CalculateDiscount(){
      return discount.CalculateDiscount(productCost);
   }
}
```

  - Discount Interface

```
interface IDiscount
{
        double calculateDiscount(double productCost);
}
```

NULL OBJECT. EXAMPLE

- An application that calculates discount for an order
    - Discount interface implementations

```
public class PremiumDiscount implememts IDiscount
{
        public double calculateDiscount(double productCost)
        {
            return (productCost*0.5);
        }
}

public class FestivalDiscount implements IDiscount
{
        public double calculateDiscount(double productCost)
        {
            return (productCost * 0.2);
        }
}
```

NULL OBJECT. EXAMPLE

- **An application that calculates discount for an order**
  - Setting a discount for an Order

```
class Order {
  // Code removed for simplification
  private IDiscount discount = null;

  public Order(IDiscount dis) {
       discount = dis;
  }
}
```

  - Creating order objects and calculating discount

```
Order PremiumOrder = new Order(new
                       PremiumDiscount());
Order FestivalOrder = new Order(new
                       FestivalDiscount())
Order NoDiscountOrder = new Order(null);
NoDiscountOrder.calculateDiscount();
```

What happes in this case?

NULL OBJECT. EXAMPLE

- **An application that calculates discount for an order**
  - Adding a null discount class

```
public class NullDiscount extends IDiscount {
  public double calculateDiscount(double productCost){
            return 0;
  }
}
```

  - Refactor previous object creation

```
Order NoDiscountOrder = new Order(new NullDiscount());
NoDiscountOrder.calculateDiscount();
```

NULL OBJECT. EXAMPLE

- **An application that calculates discount for an order**
  - Null Objects remove null checks

```
public double calculateDiscount() {
   if (discount == null) {
         return 0;
    }
    return discount.calculateDiscount(ProductCost);
}
```

NULL OBJECT

- **The Null Object class is often implemented as a Singleton.**
  - Since a null object usually does not have any state, its state can't change, so multiple instances are identical. Rather than use multiple identical instances, the system can just use a single instance repeatedly.
- **The Null Object design pattern is more likely to be used in conjunction with the Factory pattern.**
  - The reason for this is obvious: A Concrete Classes need to be instantiated and then to be served to the client. The client uses the concrete class. The concrete class can be a Real Object or a Null Object.
- **The Null Object can be used to remove old functionality by replacing it with null objects.**
  - The big advantage is that the existing code doesn't need to be touched.
- **The Null Object Pattern is used to avoid special if blocks for do nothing code, by putting the "do nothing" code in the Null Object which becomes responsible for doing nothing.**
  - The client is not aware anymore if the real object or the null object is called so the 'if' section is removed from client implementation.

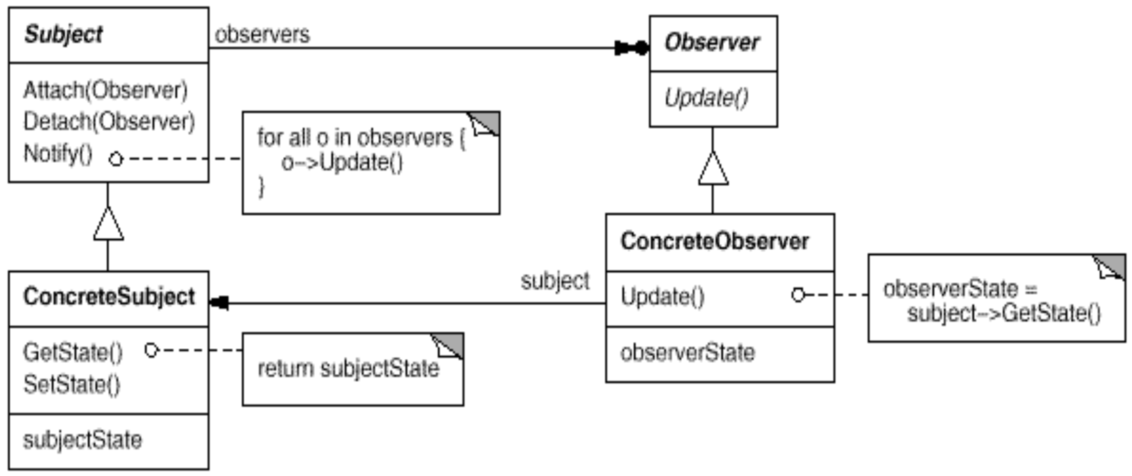
CURRENT COURSE

- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **<u>Observer</u>**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

OBSERVER

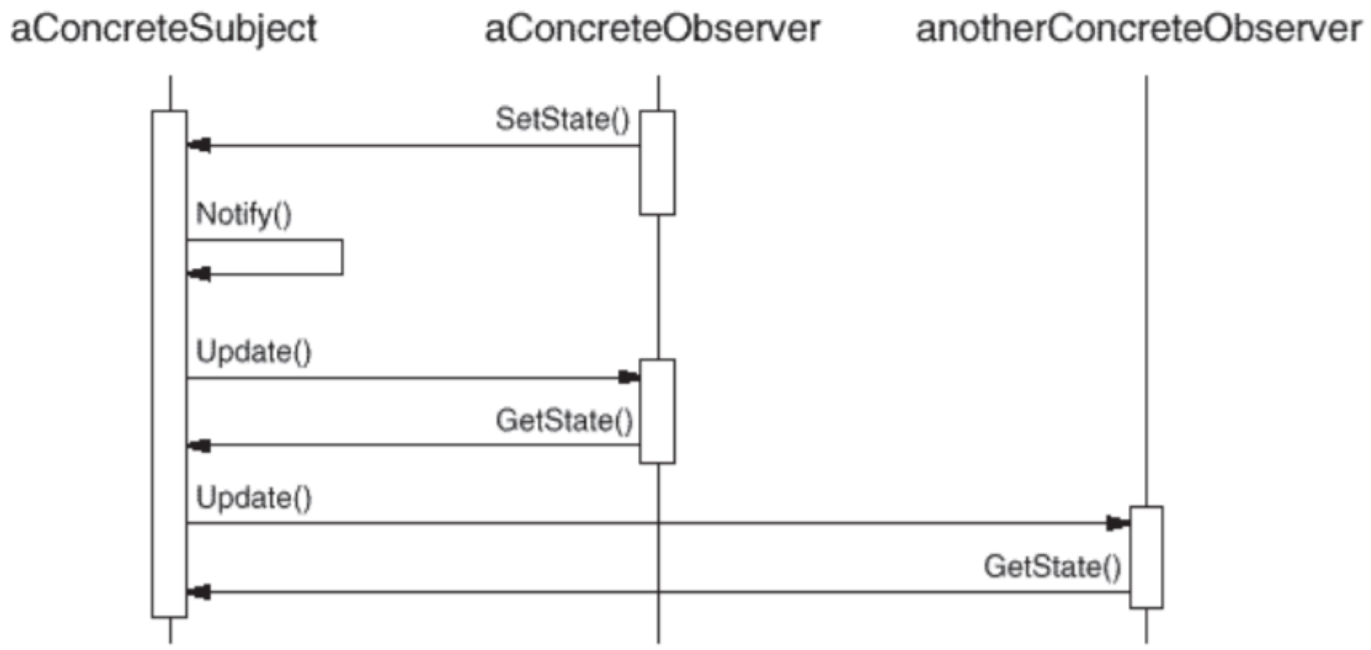
- **Intent**
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
  - The "View" part of Model-View-Controller.
- **Problem**
  - A large monolithic design does not scale well as new graphing or monitoring requirements are levied.
- **Also known**
  - Dependents, Publish-Subscribe

OBSERVER. STRUCTURE



- **Subject (Observable)**
  - interface or abstract class defining the operations for attaching and de-attaching observers to the client.
- **ConcreteSubject**
  - concrete Subject class.
  - it maintain the state of the object and when a change in the state occurs it notifies the attached Observers.
- **Observer**
  - interface or abstract class defining the operations to be used to notify this object.
- **ConcreteObserver**
  - concrete Observer implementations.

OBSERVER. SEQUENCE



OBSERVER

- **Aplicability**
  - The change of a state in one object must be reflected in another object without keeping the objects tight coupled.
  - The framework we are writing needs to be enhanced in future with new observers with minimal changes.

- **Examples**
  - Model View Controller Pattern
    - The observer pattern is used in the model view controller (MVC) architectural pattern. In MVC the this pattern is used to decouple the model from the view. View represents the Observer and the model is the Observable object.
  - Event management
    - This is one of the domains where the Observer patterns is extensively used. Swing and .Net are extensively using the Observer pattern for implementing the events mechanism.
  - A publisher/subscriber relationship with one source having many subscribers
    - RSS feeds. When someone wants to get updates from a particular feed, It will add it to its feed reader. Any time that the RSS feed has an update, it will appear in reader automatically

OBSERVER. IMPLEMENTATION

- **We could implement the Observer pattern "from scratch" in Java**

- **But Java provides the `Observable/Observer` classes as built-in support for the Observer pattern**

  - The `java.util.Observable` class is the base Subject class. Any class that wants to be observed extends this class.
    - Provides methods to add/delete observers
    - Provides methods to notify all observers
    - A subclass only needs to ensure that its observers are notified in the appropriate mutators
    - Uses a Vector for storing the observer references

  - The `java.util.Observer` interface is the Observer interface. It must be implemented by any observer class

OBSERVER. IMPLEMENTATION

- **`java.util.Observable Class`**
  - `public Observable()`
    - Construct an Observable with zero Observers
  - `public synchronized void addObserver(Observer o)`
    - Adds an observer to the set of observers of this object
  - `public synchronized void deleteObserver(Observer o)`
    - Deletes an observer from the set of observers of this object
  - `protected synchronized void setChanged()`
    - Indicates that this object has changed
  - `protected synchronized void clearChanged()`
    - Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change. This method is called automatically by `notifyObservers()`.

OBSERVER. IMPLEMENTATION

- **java.util.Observable Class**
  - `public synchronized boolean hasChanged()`
    - Tests if this object has changed. Returns true if `setChanged()` has been called more recently than `clearChanged()` on this object; false otherwise.
  - `public void notifyObservers(Object arg)`
    - If this object has changed, as indicated by the `hasChanged()` method, then notify all of its observers and then call the `clearChanged()` method to indicate that this object has no longer changed. Each observer has its `update()` method called with two arguments: this observable object and the arg argument. The arg argument can be used to indicate which attribute of the observable object has changed.
  - `public void notifyObservers()`
    - Same as above, but the arg argument is set to null. That is, the observer is given no indication what attribute of the observable object has changed.

OBSERVER. IMPLEMENTATION

- **java.util.Observer Interface**
  - `public abstract void update(Observable o, Object arg)`
    - This method is called whenever the observed object is changed. An application calls an observable object's `notifyObservers()` method to have all the object's observers notified of the change.
    - Parameters:
      - o - the observable object
      - arg - an argument passed to the notifyObservers method

OBSERVER. EXAMPLE

```
/**  A subject to observe! */

public class ConcreteSubject
                    extends Observable {
   private String name;
   private float price;

   public ConcreteSubject(String name,
                                float price) {
       this.name = name;
       this.price = price;
       System.out.println(
         "ConcreteSubject created: " + name
         + " at " + price);
   }

   public String getName() {return name;}
   public float getPrice() {return price;}
   public void setName(String name) {
      this.name = name;
      setChanged();
      notifyObservers(name);
   }
   public void setPrice(float price) {
      this.price = price;
      setChanged();
      notifyObservers(new Float(price));
   }
}
```

OBSERVER. EXAMPLE

```
// An observer of name changes.
public class NameObserver implements Observer {
   private String name;
   public NameObserver() {
      name = null;
      System.out.println("NameObserver created: Name is " + name);
    }
    public void update(Observable obj, Object arg) {
        if (arg instanceof String) {
                name = (String)arg;
                System.out.println("NameObserver: Name changed to ” + name);
        } else {
                System.out.println("NameObserver:” +
                                " Some other change to subject!");
        }
    }
}
```

OBSERVER. EXAMPLE

```
// An observer of price changes.
public class PriceObserver implements Observer {
     private float price;
     public PriceObserver() {
         price = 0;
          System.out.println("PriceObserver created: Price is " + price);
     }
     public void update(Observable obj, Object arg) {
         if (arg instanceof Float) {
                price = ((Float)arg).floatValue();
               System.out.println("PriceObserver: Price changed to " + price);
         } else {
                System.out.println(”PriceObserver: Some other “ +
                                                    “ change to subject!");
 }
 }}
```

OBSERVER. EXAMPLE

```
// Test program for ConcreteSubject,
// NameObserver and PriceObserver
public class TestObservers {
 public static void main(String args[]) {
 // Create the Subject and Observers.
    ConcreteSubject s =
         new ConcreteSubject("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
 // Add those Observers!
    s.addObserver(nameObs);
    s.addObserver(priceObs);
 // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Sugar Crispies");
  }
}
```

Test program output

```
ConcreteSubject created: Corn Pops at 1.29
NameObserver created: Name is null
PriceObserver created: Price is 0.0

PriceObserver: Some other change to subject!
NameObserver: Name changed to Frosted Flakes
PriceObserver: Price changed to 4.57

NameObserver: Some other change to subject!
PriceObserver: Price changed to 9.22

NameObserver: Some other change to subject!
PriceObserver: Some other change to subject!
NameObserver: Name changed to Sugar Crispies
```

OBSERVER

- **Problem**
  - Suppose the class which we want to be an observable is already part of an inheritance hierarchy: class SpecialSubject extends ParentClass
  - Since Java does not support multiple inheritance, how can we have ConcreteSubject extend both Observable and ParentClass?

- **Solution**
  - Use Delegation
  - We will have SpecialSubject contain an Observable object
  - We will delegate the observable behavior that SpecialSubject needs to this contained Observable object

OBSERVER. DELEGATION

```
// A subclass of Observable that allows delegation.
public class DelegatedObservable extends Observable
{
    public void clearChanged() {
        super.clearChanged();
    }

    public void setChanged() {
      super.setChanged();
    }
}
```

OBSERVER. DELEGATION

```
public class SpecialSubject
                  extends ParentClass {
   private String name;
   private float price;
   private DelegatedObservable obs;
   public SpecialSubject(String name,
                          float price) {
       this.name = name;
       this.price = price;
       obs = new DelegatedObservable();
   }
   public String getName() {return name;}
   public float getPrice() {
                         return price;
   }
   public Observable getObservable() {
                         return obs;
   }
```

```
public void setName(String name) {
      this.name = name;
      obs.setChanged();
      obs.notifyObservers(name);
 }

 public void setPrice(float price) {
     this.price = price;
     obs.setChanged();
     obs.notifyObservers(
                    new Float(price));
 }
}
```

OBSERVER. DELEGATION

```
// Test program for SpecialSubject with a Delegated Observable.
public class TestSpecial {
   public static void main(String args[]) {
       // Create the Subject and Observers.
        SpecialSubject s = new SpecialSubject("Corn Pops", 1.29f);
        NameObserver nameObs = new NameObserver();
        PriceObserver priceObs = new PriceObserver();
        // Add those Observers!
        s.getObservable().addObserver(nameObs);
        s.getObservable().addObserver(priceObs);
        // Make changes to the Subject.
        s.setName("Frosted Flakes");
        s.setPrice(4.57f);
        s.setPrice(9.22f);
        s.setName("Sugar Crispies");
   }
}
```

How this call can be simplified?

OBSRVER

- **Consequences**
  - Benefits
    - Minimal coupling between the Subject and the Observer
      - Can reuse subjects without reusing their observers and vice versa
      - Observers can be added without modifying the subject
      - All subject knows is its list of observers
      - Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
      - Subject and observer can belong to different abstraction layers
    - Support for event broadcasting
      - Subject sends notification to all subscribed observers
      - Observers can be added/removed at any time

OBSERVER

- **Consequences**
  - Liabilities
    - Possible cascading of notifications
    - Observers are not necessarily aware of each other and must be careful about triggering updates
    - Simple update interface requires observers to deduce changed item

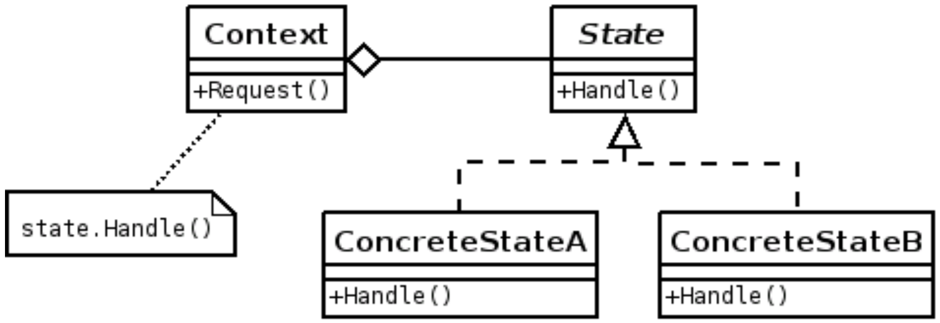
CURRENT COURSE

- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

STATE

- **Intent**
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Problem**
  - A monolithic object's behavior is a function of its state, and it must change its behavior at runtime depending on that state. Or, an application is characterized by large and numerous case statements that vector flow of control based on the state of the application.

STATE. STRUCTURE



- **Context**
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state.
- **State**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **Concrete State**
  - each subclass implements a behavior associated with a state of Context

STATE.EXAMPLE

- **Implement the changing states for a Fan**
  - A fan goes from
    - Turn on
    - Low fan
    - Medium fan
    - High fan
    - Turn off
  - What you do?
    - Identify actions: turn on, turn off, change low/medium/height fan
    - Identify the states
    - Identify transitions

STATE. EXAMPLE

- **Implement the changing states for a Fan**
  - without pattern

```
class CeilingFanPullChain {
    private int m_current_state;
    public CeilingFanPullChain() {  m_current_state = 0;   }
    public void press(boolean direction) {
      if (m_current_state == 0) {
            if (direction) {
                  m_current_state = 1; System.out.println("   low speed");
           } else {
                  System.out.println("   no down action possible from here");   }
     } else if (m_current_state == 1) {
         if (direction) {
             m_current_state = 2; System.out.println("   medium speed");
         } else {
             m_current_state = 0; System.out.println("   turning off");       }
     }
```

STATE. EXAMPLE

```
    } else if (m_current_state == 2) {
              if (direction) {
                      m_current_state = 3;System.out.println("   high speed");
               } else {
                      m_current_state = 1; System.out.println("   low speed");
              }
       } else if (m_current_state == 3) {
              if (direction) {
                      System.out.println("   no up  action possible from here ");
             } else {
                      m_current_state = 2; System.out.println("   medium speed");
             }
     }
  }
}
```

STATE. EXAMPLE

```
public class StateDemo {
    public static void main(String[] args) {
         Random r = new Random();
         CeilingFanPullChain chain = new CeilingFanPullChain();
          while (true) {
                    boolean b=r.nextBoolean();
                    System.out.print("Press " + (b?"up": "down"));
                    get_line();
                    chain.press(b);
          }}
      static String get_line() {
          BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
          String line = null;
          try {
                line = in.readLine();
          } catch (IOException ex) {  ex.printStackTrace();}
          return line;
       }}
```

STATE. EXAMPLE

- Implement the changing state for a Fan
  - using the pattern

```
class CeilingFanPullChain {
          private State m_current_state;
          private boolean direction;
          public boolean getDirection() {
                    return this.direction;
          }
          public CeilingFanPullChain() {
                    m_current_state = new Off();
          }
          public void set_state(State s) {
                    m_current_state = s;
          }
          public void change(boolean direction) {
                    m_current_state.change(this);
          }
}
```

STATE. EXAMPLE

```
interface State {  void change(CeilingFanPullChain wrapper); }
class Off implements State {
     public void change(CeilingFanPullChain wrapper) {
             if (wrapper.getDirection()) {
                   wrapper.set_state(new Low()); System.out.println("   low speed");
             } else {
                   System.out.println("   no down action possible from here");
              }
      }
}
class Low implements State {
      public void change(CeilingFanPullChain wrapper) {
            if (wrapper.getDirection()) {
                wrapper.set_state(new Medium());  System.out.println("   medium 
speed");
            } else {
                   wrapper.set_state(new Off());  System.out.println("   turning off");
            }
    }
}
```

STATE. EXAMPLE

```
class Medium implements State {
        public void change(CeilingFanPullChain wrapper) {
               if (wrapper.getDirection()) {
                         wrapper.set_state(new High());
                         System.out.println("   high speed");
               } else {
                        wrapper.set_state(new Low());
                        System.out.println("   low speed");
              }
      }
}
class High implements State {
         public void change(CeilingFanPullChain wrapper) {
               if (wrapper.getDirection()) {
                         System.out.println("   no up action possible from here");
                } else {
                          wrapper.set_state(new Medium());
                          System.out.println("   medium speed");
              }
        }
}
```

STATE. EXAMPLE

```
public class StateDemo {
            public static void main(String[] args) {
                        Random r = new Random();
                        CeilingFanPullChain chain = new CeilingFanPullChain();
                        while (true) {
                                    boolean b=r.nextBoolean();
                                    System.out.print("Press " + (b?"up": "down"));
                                    get_line();
                                    chain.change(r.nextBoolean());
                        }
            }

            static String get_line() {
                        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
                        String line = null;
                        try {line = in.readLine();} catch (IOException ex) {ex.printStackTrace();}
                        return line;
            }
}
```

STATE

- **Consequences**
  - Benefits
    - Puts all behavior associated with a state into one object
    - Allows state transition logic to be 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

STATE

- **Kill off if/then statements.**
  - This is one of the primary goals of many of the original Gang of Four patterns, and it’s a worthy goal. If/then branching can breed bugs.

- **Reduces duplication by eliminating repeated if/then or switch statements, same as its close cousin the Strategy pattern.**

- **Increased cohesion.**
  - By aggregating state specific behavior into State classes with intention revealing names it’s easier to find that specific logic, and it’s all in one place.

- **Potential extensibility.**

- **Better testability.**

CURRENT COURSE

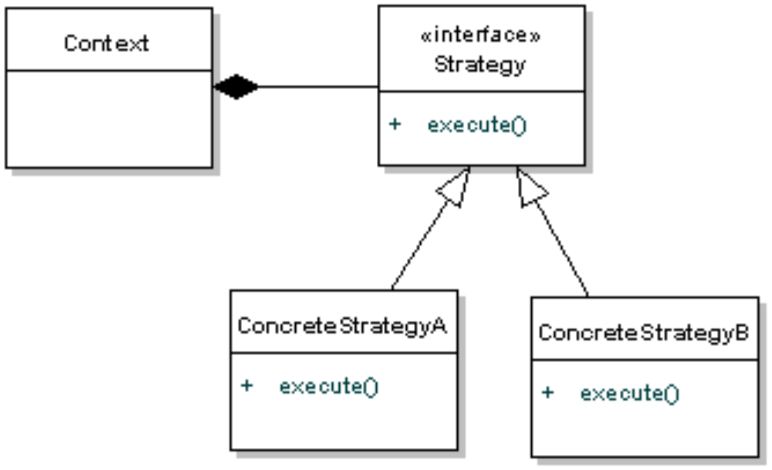
- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

STRATEGY

- **Intent**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
  - Capture the abstraction in an interface, bury implementation details in derived classes.

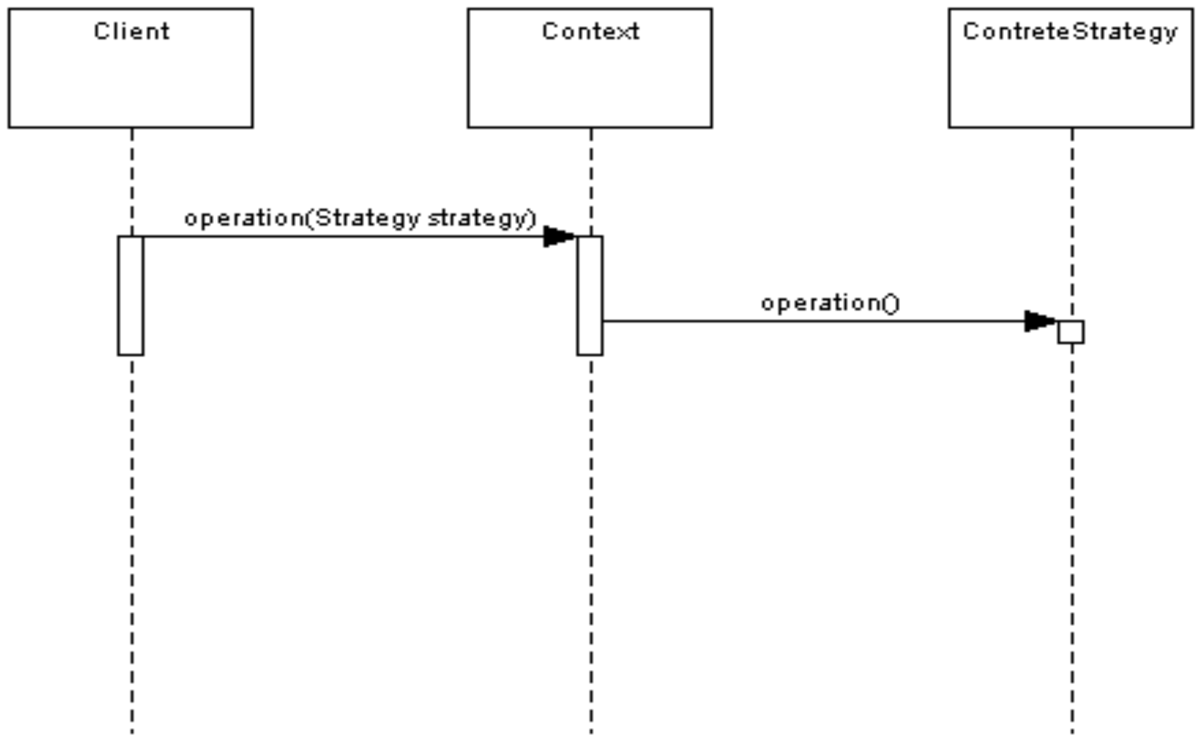
- **Problem**
  - One of the dominant strategies of object-oriented design is the "open-closed principle".

STRATEGY. STRUCTURE



- **Strategy**
  - defines an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy**
  - each concrete strategy implements an algorithm.
- **Context**
  - contains a reference to a strategy object.
  - may define an interface that lets strategy accessing its data.

STRATEGY BEHAVIOR

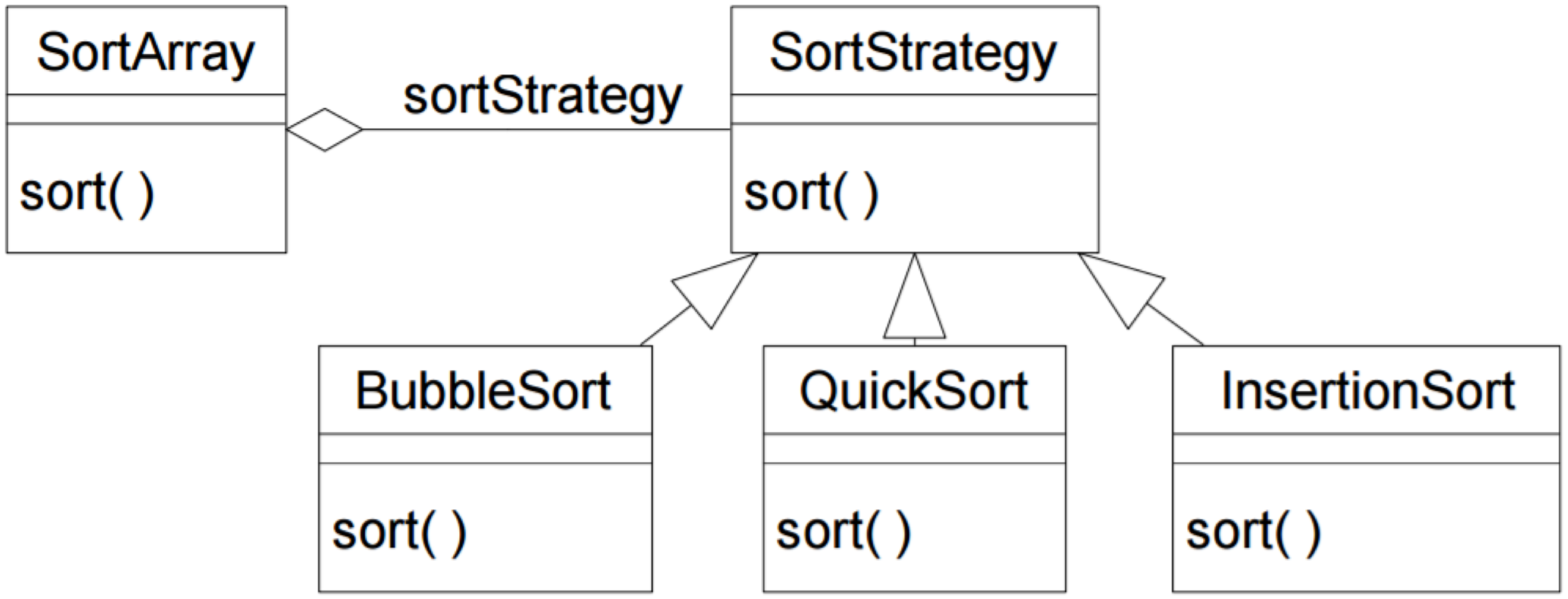


STRATEGY. EXAMPLE

- **A Strategy defines a set of algorithms that can be used interchangeably**
  - Modes of transportation to an airport is
    - Several options exist such as
      - driving one's own car
      - taking a taxi
      - an airport shuttle
      - a city bus
      - a limousine service
    - For some airports, subways and helicopters are also available as a mode of transportation to the airport.
    - Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably.
    - The traveler must chose the Strategy based on trade-offs between cost, convenience, and time.

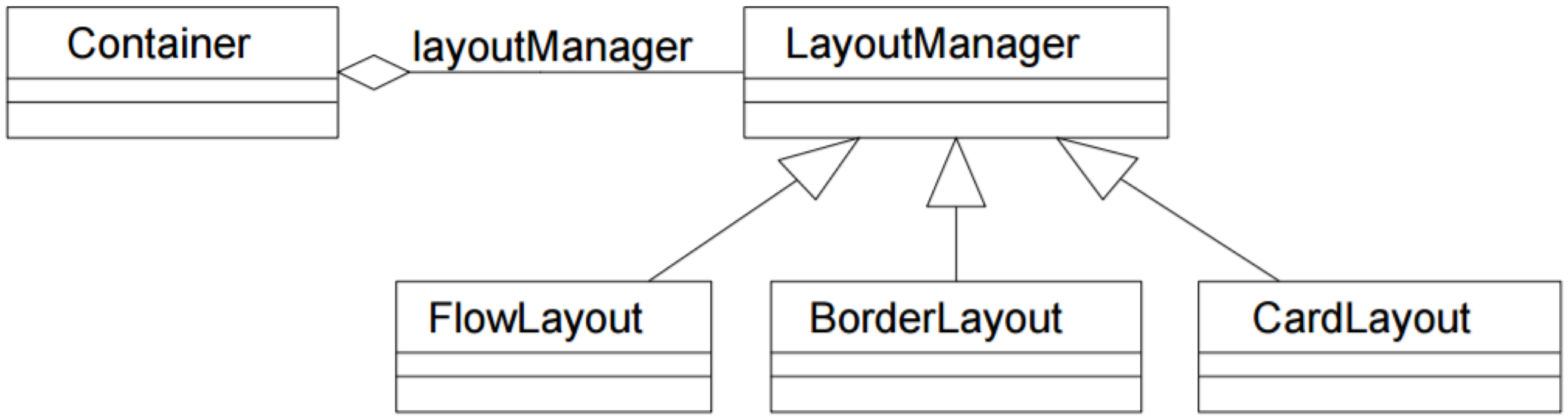
STRATEGY. EXAMPLE

- 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!



STRATEGY. EXAMPLE

- 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



STRATEGY. EXAMPLE

- **Example**
  - Different operation between two numbers

STRATEGY. EXAMPLE

```
public interface Strategy {
    public int doOperation(int num1, int num2);
}

public class OperationAdd implements Strategy{
    @Override
     public int doOperation(int num1, int num2) {
             return num1 + num2;
     }
}

public class OperationSubstract implements Strategy{
      @Override
      public int doOperation(int num1, int num2) {
             return num1 - num2;
      }
}
```

STRATEGY. EXAMPLE

```
public class OperationMultiply implements Strategy{
      @Override
      public int doOperation(int num1, int num2) {
             return num1 * num2;
      }
}

public class Context {
      private Strategy strategy;
      public Context(Strategy strategy){
          this.strategy = strategy;
      }
      public int executeStrategy(int num1, int num2){
           return strategy.doOperation(num1, num2);
      }
}
```

STRATEGY. EXAMPLE

```
public class StrategyPatternDemo {

      public static void main(String[] args) {
           Context context = new Context(new OperationAdd());
           System.out.println("10 + 5 = " +
                                                context.executeStrategy(10, 5));

           context = new Context(new OperationSubstract());
           System.out.println("10 - 5 = " +
                                                context.executeStrategy(10, 5));

           context = new Context(new OperationMultiply());
           System.out.println("10 * 5 = " +
                                                context.executeStrategy(10, 5));
      }
}
```

STRATEGY

- **Applicability**
  - 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.

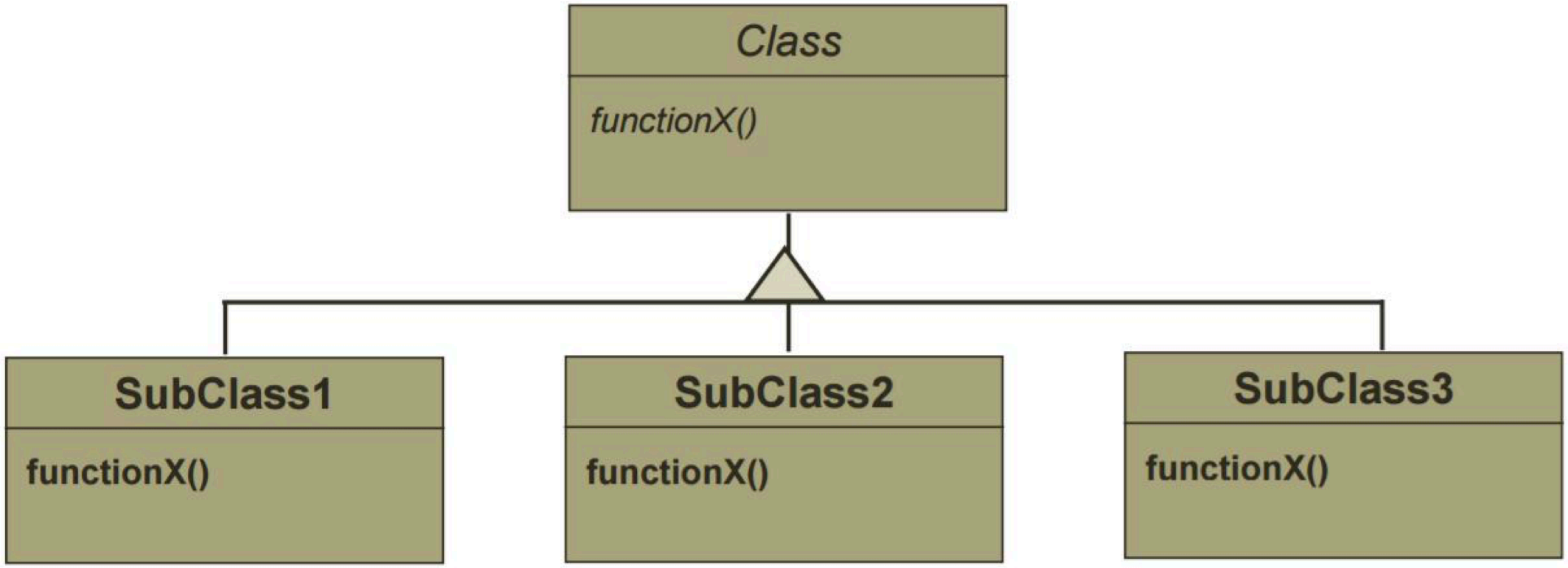
STRATEGY

- **Consequences**
  - 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 VS SUBCLASSING

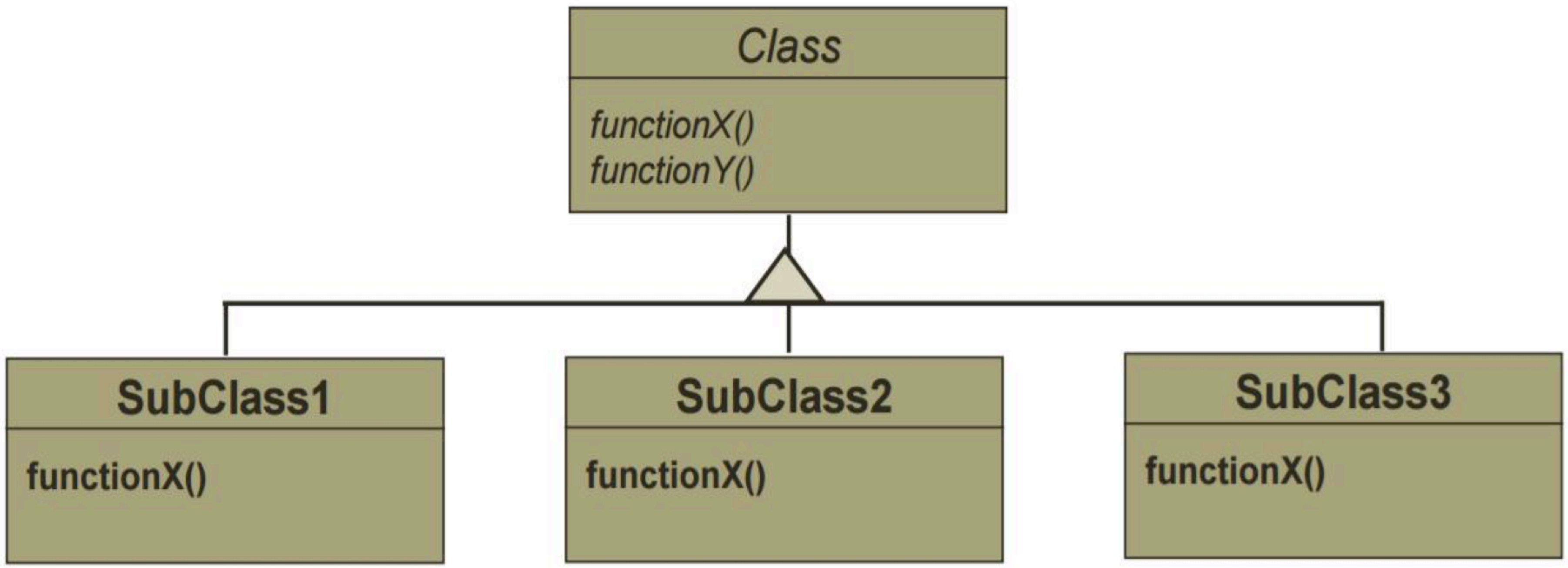
- Strategy can be used in place of subclassing
- Strategy is more dynamic
- Multiple strategies can be mixed in any combination where subclassing would be difficult

STRATEGY VS SUBCLASSING



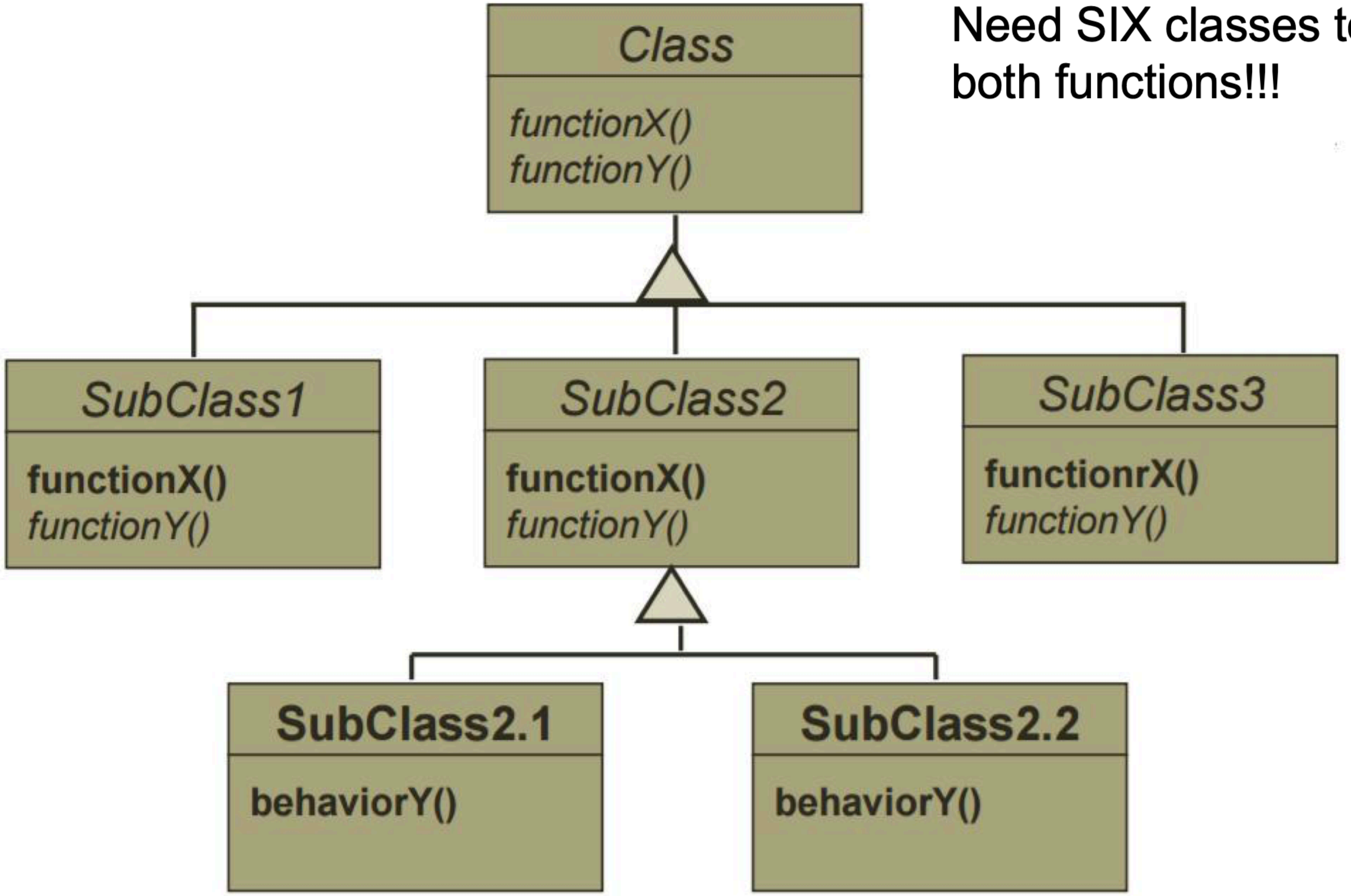
STRATEGY VS SUBCLASSING

Add another method



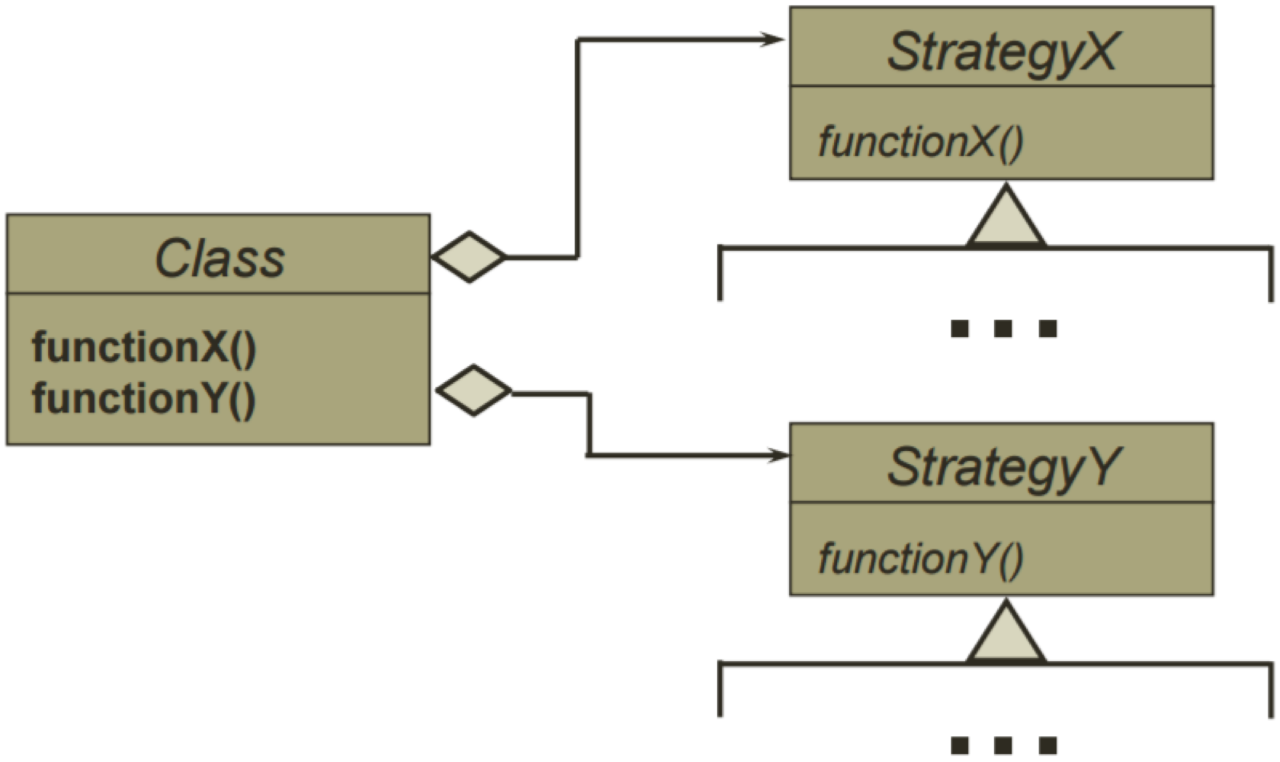
STRATEGY VS

What happens?
Need SIX classes to handle both functions!!!



STRATEGY VS SUBCLASSING

A more simple approach


StrategyX
functionX()
Class
functionX()
functionY()
StrategyY
functionY()
...
...