DESIGN PATTERNS

Course 7

PREVIOUS COURSE CONTENT

- Behavioral patterns
  - Template Method
  - Command
  - Interpreter
  - Mediator
  - Memento

CURRENT CURSE CONTENT

- Behavioral patterns
  - Chaim of responsibility
  - Strategy
- Partitioning patterns
  - Filter
  - Composite object
  - Read-only Interface

CHAIM OF RESPONSIBILITY

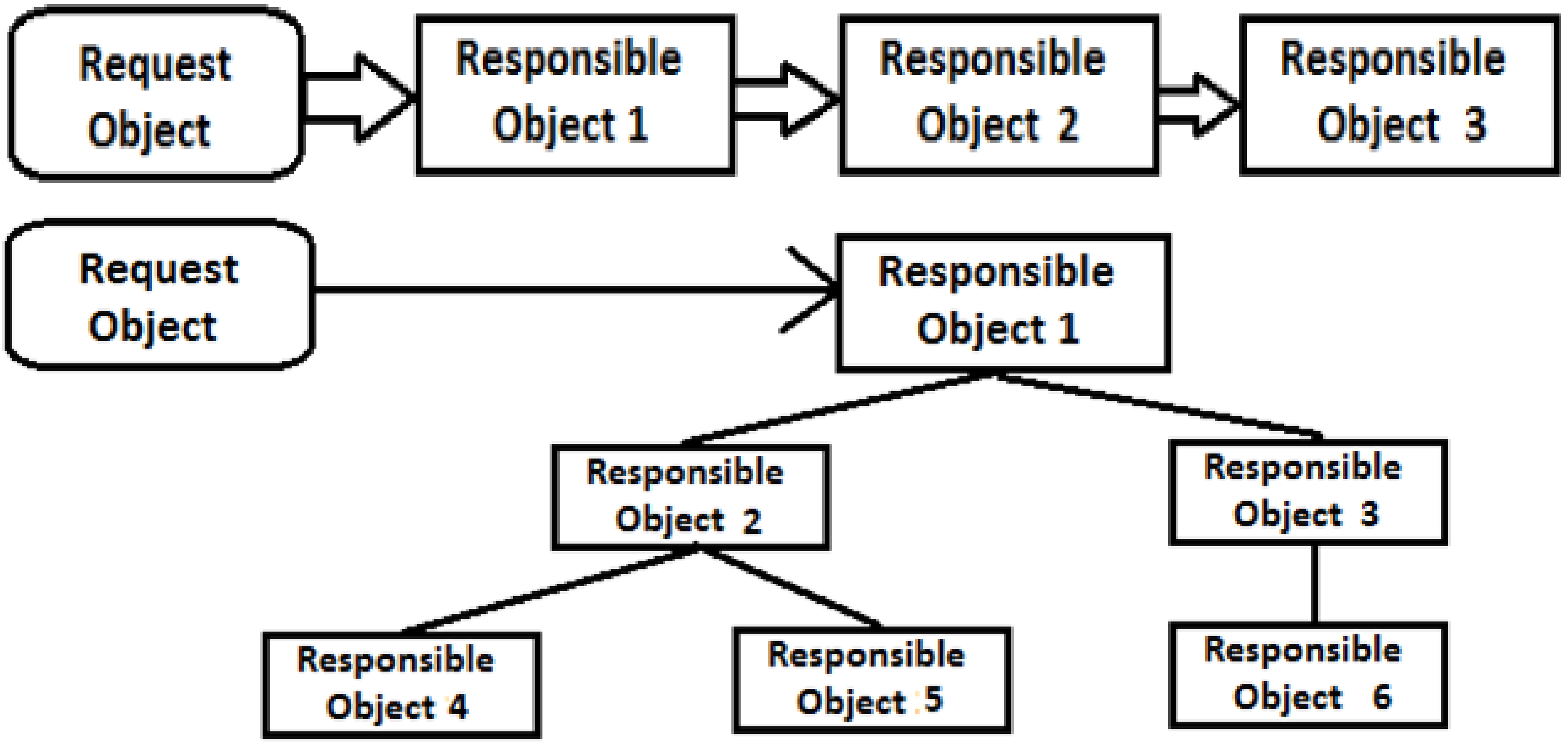
Intent

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
- An object-oriented linked list with recursive traversal.

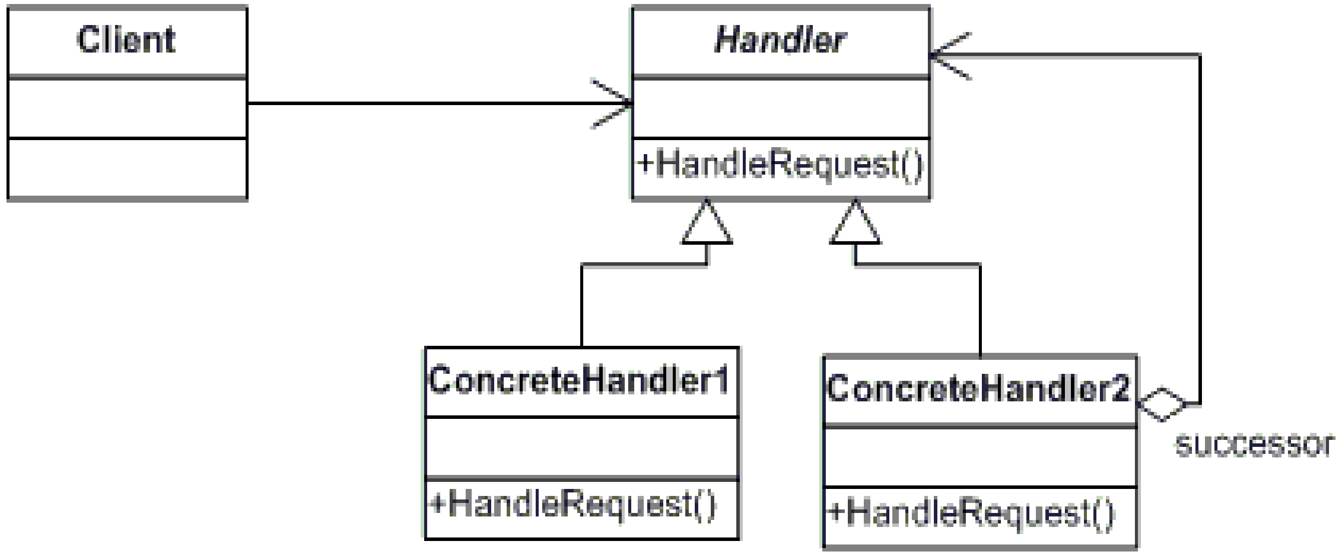
Problem

- There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

CHAIM OF RESPONSIBILITY



CHAIM OF RESPONSIBILITY



Handler

- defines an interface for handling the requests
- (optional) implements the successor link

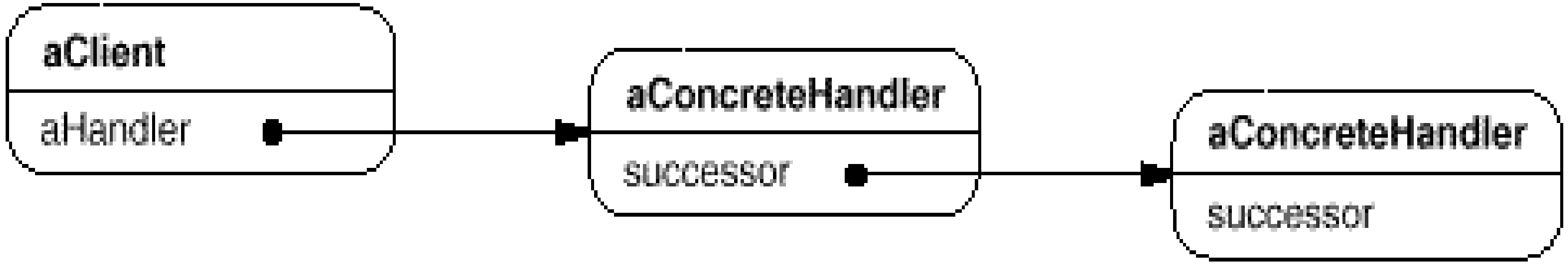
ConcreteHandler

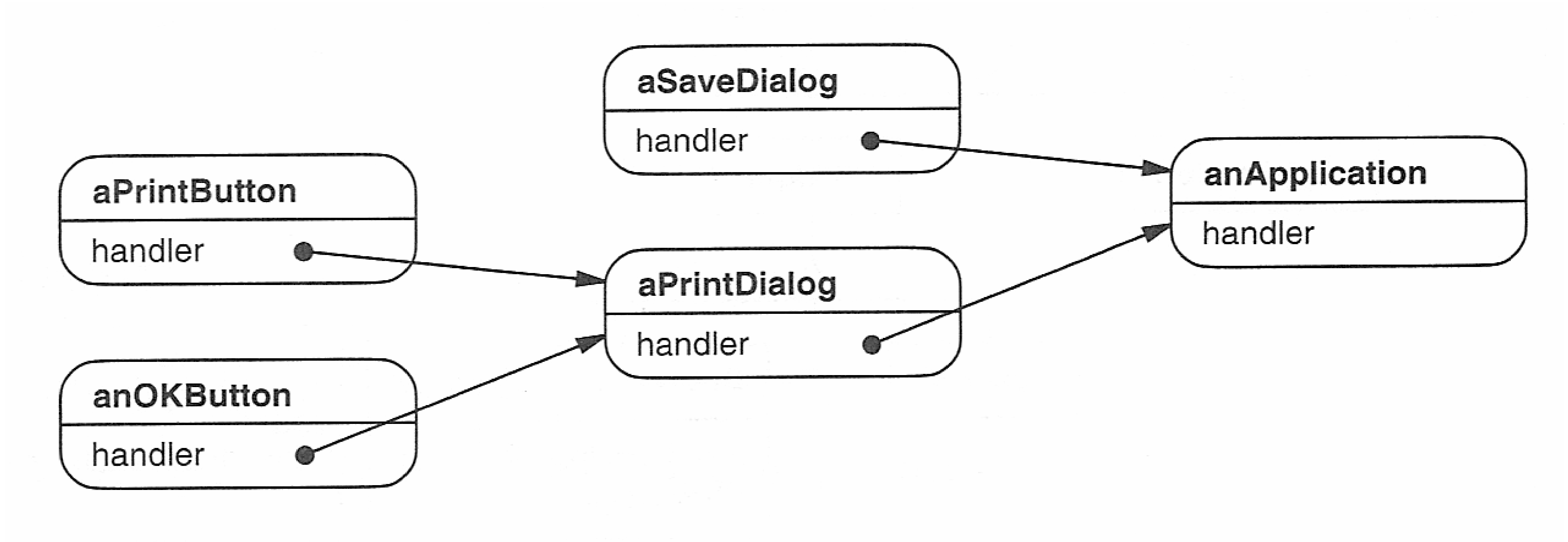
- handles requests it is responsible for
- can access its successor
- if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor

Client

- initiates the request to a ConcreteHandler object on the chain

CHAIM OF RESPONSIBILITY





CHAIM OF RESPONSIBILITY

Examples

- Designing the software that uses a set of GUI classes where it is needed to propagate GUI events from one object to another.
  - When an event, such as the pressing of a key or the click of the mouse, the event is needed to be sent to the object that has generated it and also to the object or objects that will handle it.

- Designing the software for a system that approves the purchasing requests.
  - In this case, the values of purchase are divided into categories, each having its own approval authority. The approval authority for a given value could change at any time and the system should be flexible enough to handle the situation.

- Designing a shipping system for electronic orders.
  - The steps to complete and handle the order differs form one order to another based on the customer, the size of the order, the way of shipment, destination and more other reasons. The business logic changes also as special cases appear, needing the system to be able to handle all cases.

CHAIM OF RESPONSIBILITY. EXAMPLE

```
public abstract class PlanetHandler {
        PlanetHandler successor;
        public void setSuccessor(PlanetHandler successor) {
                this.successor = successor;
        }
        public abstract void handleRequest(PlanetEnum request);
}
public enum PlanetEnum {
        MERCURY, VENUS, EARTH, MARS, JUPITER, SATURN, URANUS, NEPTUNE;
}
```

CHAIM OF RESPONSIBILITY. EXAMPLE

```
public class MercuryHandler extends PlanetHandler {
        public void handleRequest(PlanetEnum request) {
                if (request == PlanetEnum.MERCURY) {
                        System.out.println("MercuryHandler handles " + request);
                        System.out.println("Mercury is hot.\n");
                } else {
                        System.out.println("MercuryHandler doesn't handle " + request);
                        if (successor != null) {
                                successor.handleRequest(request);
                        }
                }}}
```

CHAIM OF RESPONSIBILITY. EXAMPLE

```
public class VenusHandler extends PlanetHandler {
        public void handleRequest(PlanetEnum request) {
                if (request == PlanetEnum.VENUS) {
                        System.out.println("VenusHandler handles " + request);
                        System.out.println("Venus is poisonous.\n");
                } else {
                        System.out.println("VenusHandler doesn't handle " + request);
                        if (successor != null) {
                                successor.handleRequest(request);
                        }
                }}}
```

CHAIM OF RESPONSIBILITY. EXAMPLE

```
public class EarthHandler extends PlanetHandler {
        public void handleRequest(PlanetEnum request) {
                if (request == PlanetEnum.EARTH) {
                        System.out.println("EarthHandler handles " + request);
                        System.out.println("Earth is comfortable.\n");
                } else {
                        System.out.println("EarthHandler doesn't handle " + request);
                        if (successor != null) {
                                successor.handleRequest(request);
                        }
                }}}
```

CHAIM OF RESPONSIBILITY. EXAMPLE. SIMPLE CHAIM

```
public class Demo {

    public static void main(String[] args) {
    PlanetHandler chain = setUpChain();
    chain.handleRequest(PlanetEnum.VENUS);

    chain.handleRequest(PlanetEnum.MERCURY);

    chain.handleRequest(PlanetEnum.EARTH);

    chain.handleRequest(PlanetEnum.JUPITER);
    }
```

```
public static PlanetHandler setUpChain() {
        PlanetHandler mercuryHandler =
                        new MercuryHandler();
        PlanetHandler venusHandler =
                        new VenusHandler();
        PlanetHandler earthHandler =
                         new EarthHandler();
        mercuryHandler.setSuccessor(venusHandler);
        venusHandler.setSuccessor(earthHandler);
        return mercuryHandler;
}}
```

CHAIM OF RESPONSIBILITY

Benefits

- ❑Decoupling of senders and receivers
- ❑Added flexibility
- ❑Sender doesn’t need to know specifically who the handlers are

Disadvantages

- ❑Client can’t explicitly specify who handles a request
- ❑No guarantee of request being handled (request falls off end of chain)

CHAIM OF RESPONSIBILITY

JDK Example

- try-catch block
- javax.servlet.Filter#doFilter()
- java.util.logging.Logger#log

CURSE CONTENT

- Behavioral patterns
  - Chaim of responsibility
  - Strategy
- Partitioning patterns
  - Filter
  - Composite object
  - Read-only Interface

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

Strategy

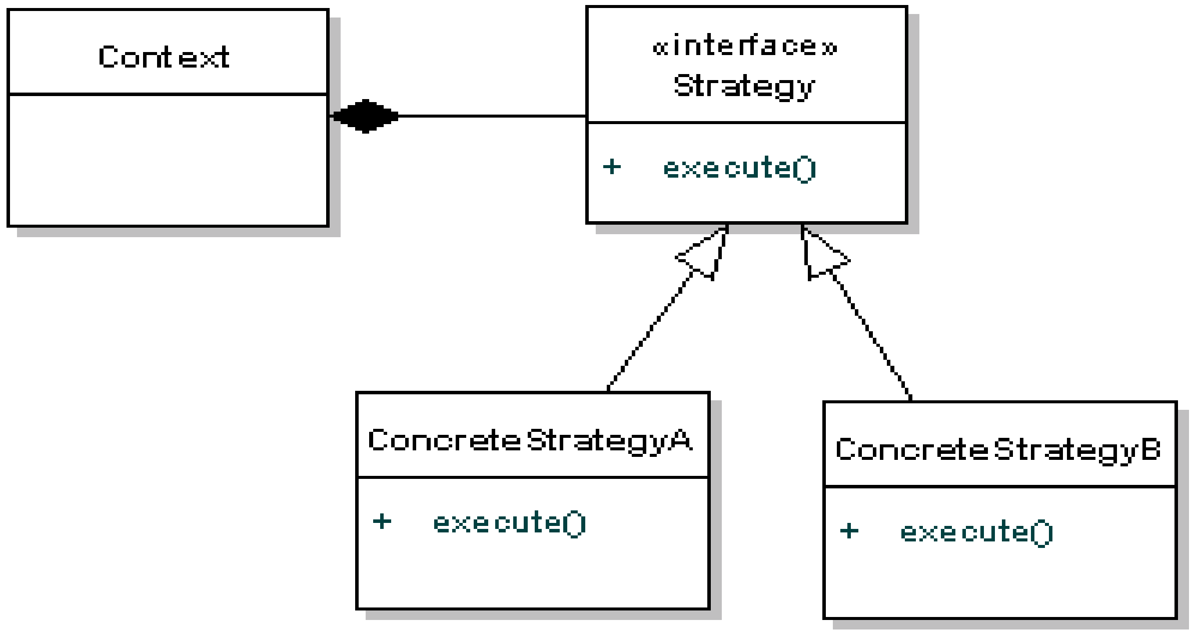
- ❑ defines an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

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

Example

- **Families of related algorithms.**
  - The strategies can be defined as a hierarchy of classes offering the ability to extend and customize the existing algorithms from an application. At this point the composite design pattern can be used with a special care.

- **Passing data to/from Strategy object**
  - Usually each strategy need data from the context have to return some processed data to the context. This can be achieved in 2 ways.
    - creating some additional classes to encapsulate the specific data.
    - passing the context object itself to the strategy objects. The strategy object can set returning data directly in the context.

-

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;
   }
}
```

STRATEGY. EXAMPLE

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

STRATEGY. EXAMPLE

```
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));
   }
}
```

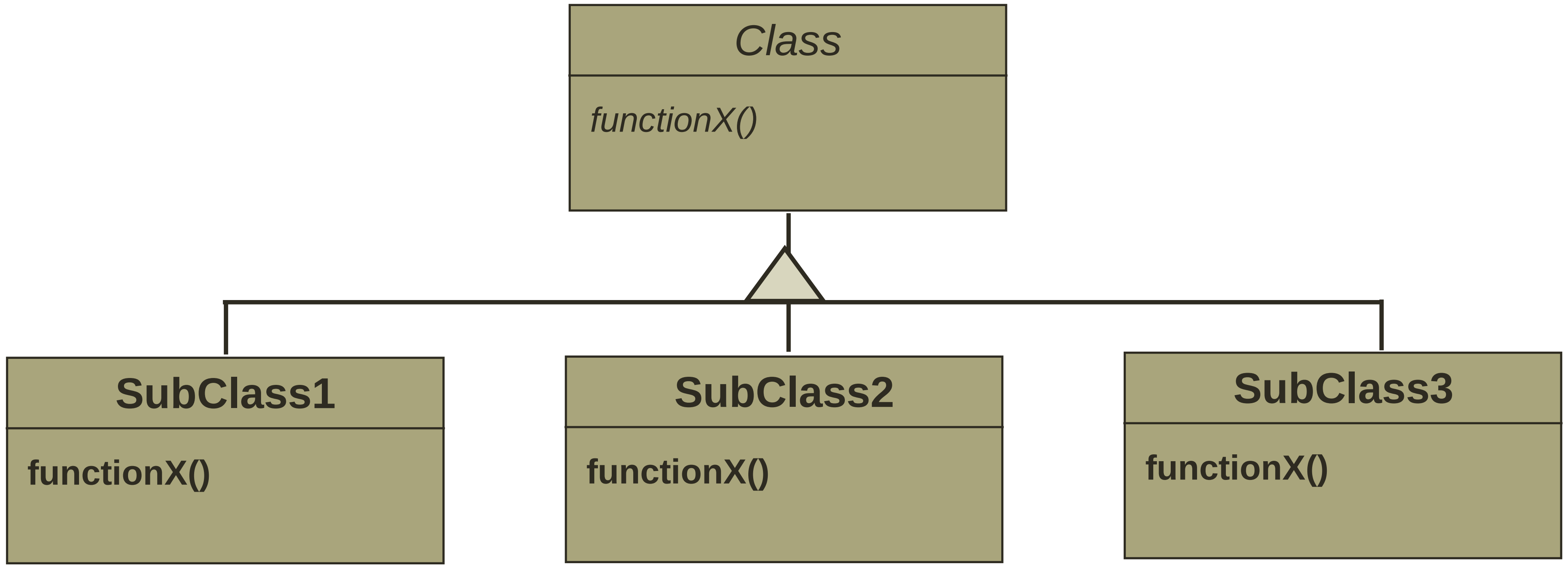
STRATEGY VS SUBCLASSING

- Strategy can be used in place of subclassing

- Strategy is more dynamic

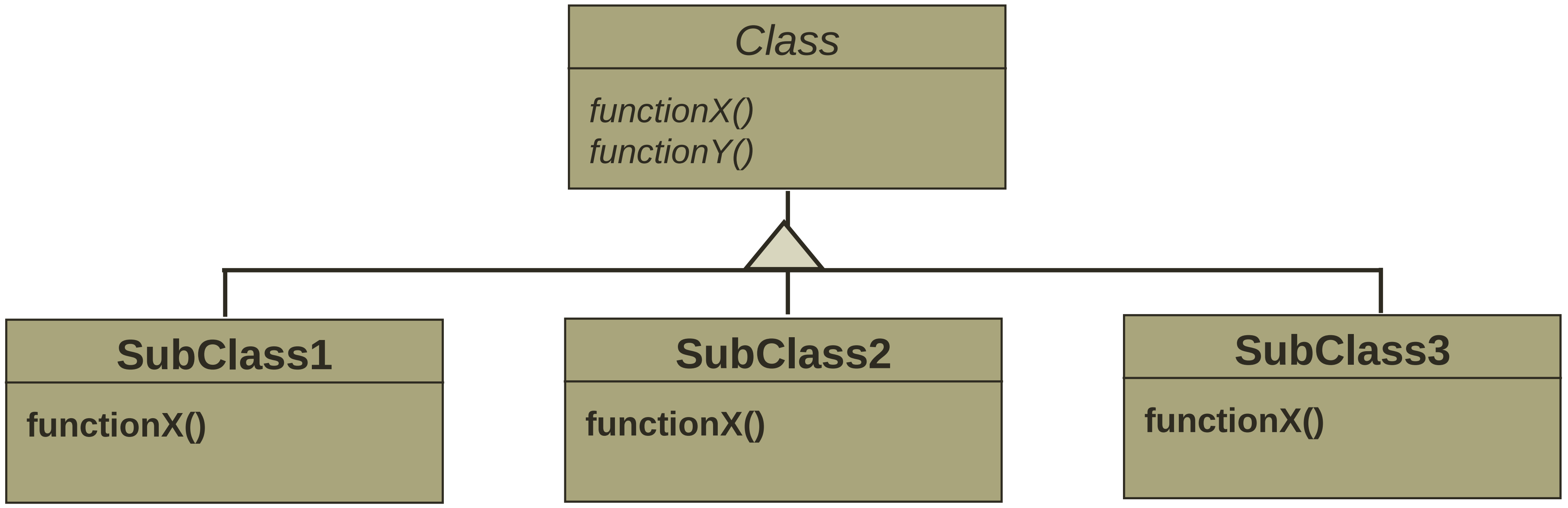
- Multiple strategies can be mixed in any combination where subclassing would be difficult

STRATEGY. SUBCLASSING

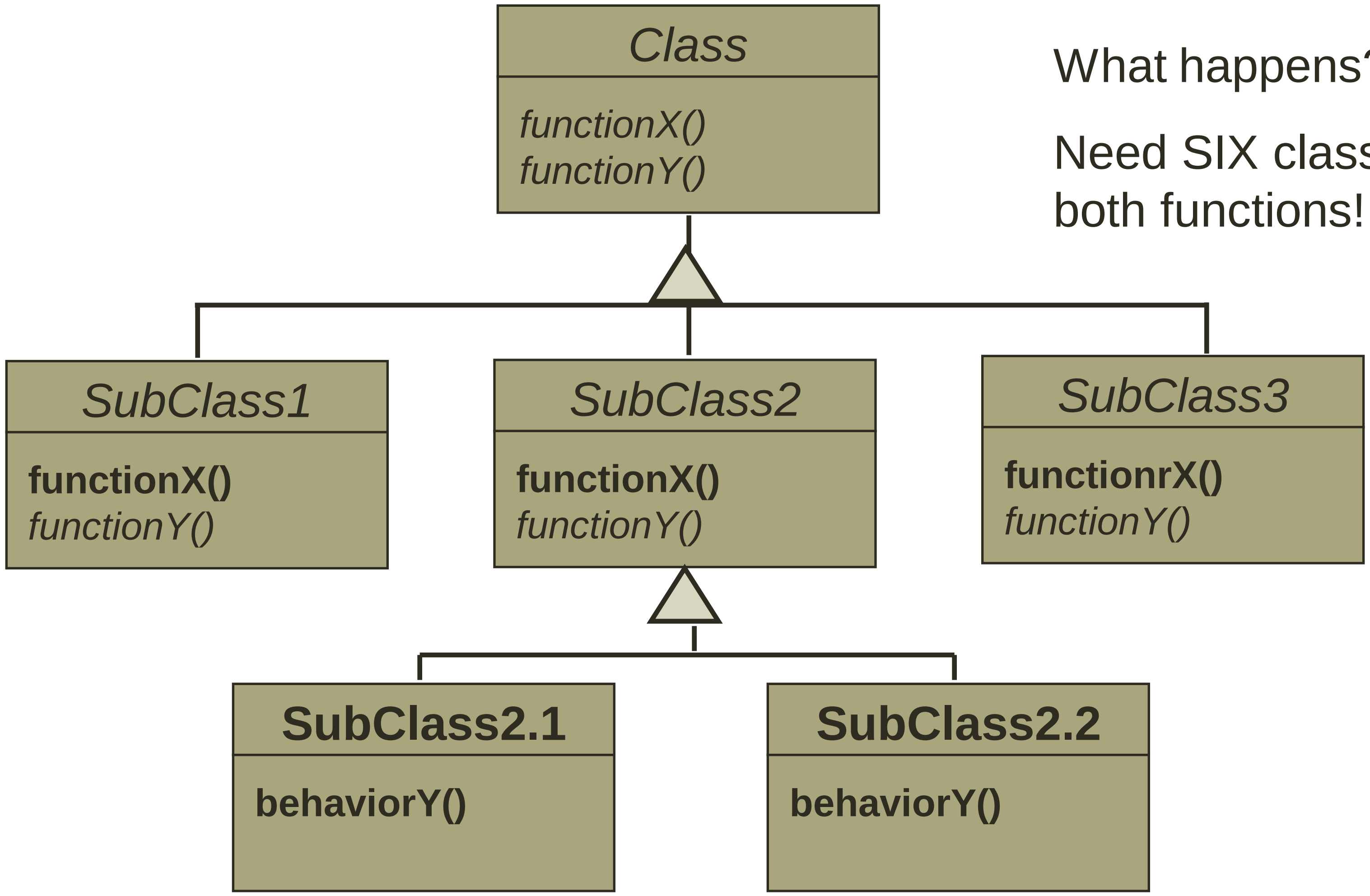


STRATEGY. SUBCLASSING

Add a new method



STRATEGY. SUBCLASSING

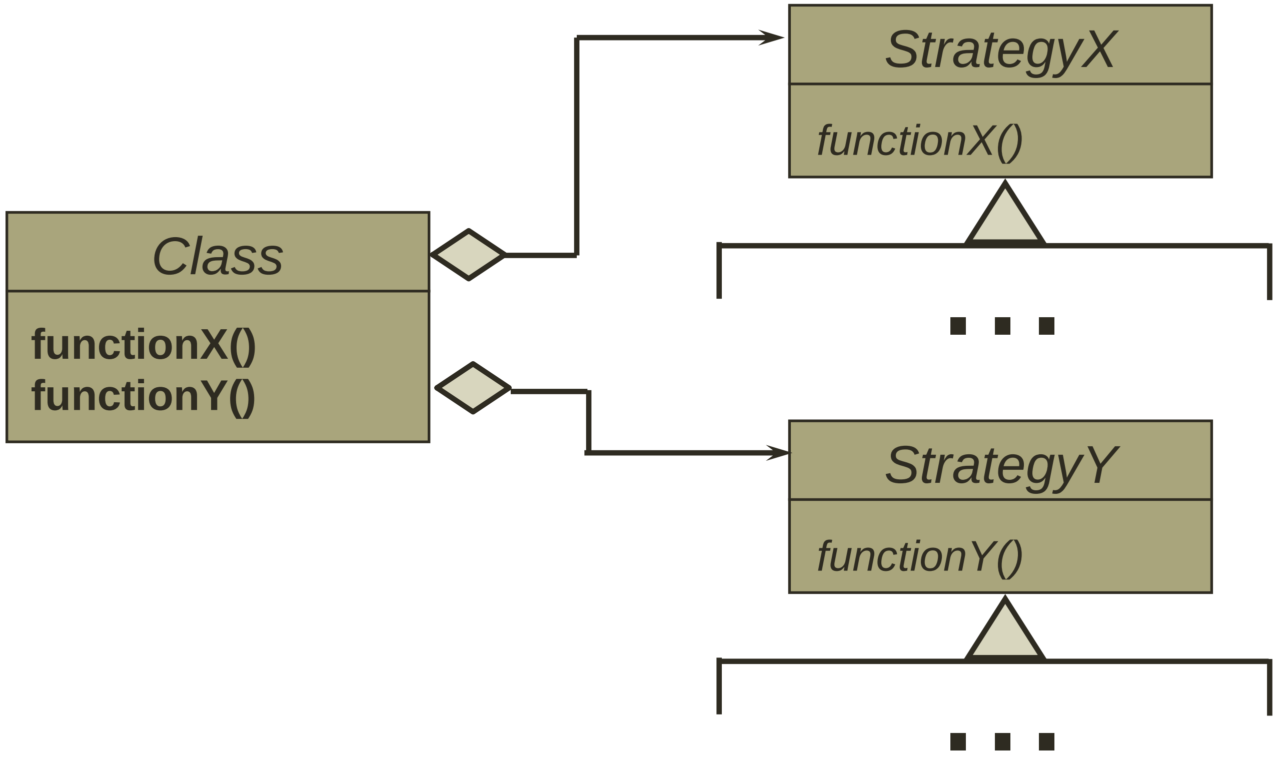


What happens?

Need SIX classes to handle both functions!!!

STRATEGY. SUBCLASSING

A mode simple approach



STRATEGY

Benefits

- Families of related algorithms
- Alternative to subclassing
  - This lets you vary the algorithm dynamically, which makes it easier to change and extend
  - You also avoid complex inheritance structures

Drawbacks

- Clients must be aware of different strategies
  - Clients must know how strategies differ so it can select the appropriate one
- Communication overhead between strategy and context
  - Sometimes the context will create and initialize parameters that are never used

STRATEGY

JDK example

-java.util.Comparator#compare()
-javax.servlet.http.HttpServlet
-javax.servlet.Filter#doFilter()

CURSE CONTENT

- Behavioral patterns
  - Chaim of responsibility
  - Strategy
- Partitioning patterns
  - Filter
  - Composite object
  - Read-only Interface

FILTER

Problem

- Use filter or criteria pattern when you need to filter a set of objects, using different criteria, changing them in a decoupled way throw logical application

Pattern Type

- Strategy pattern

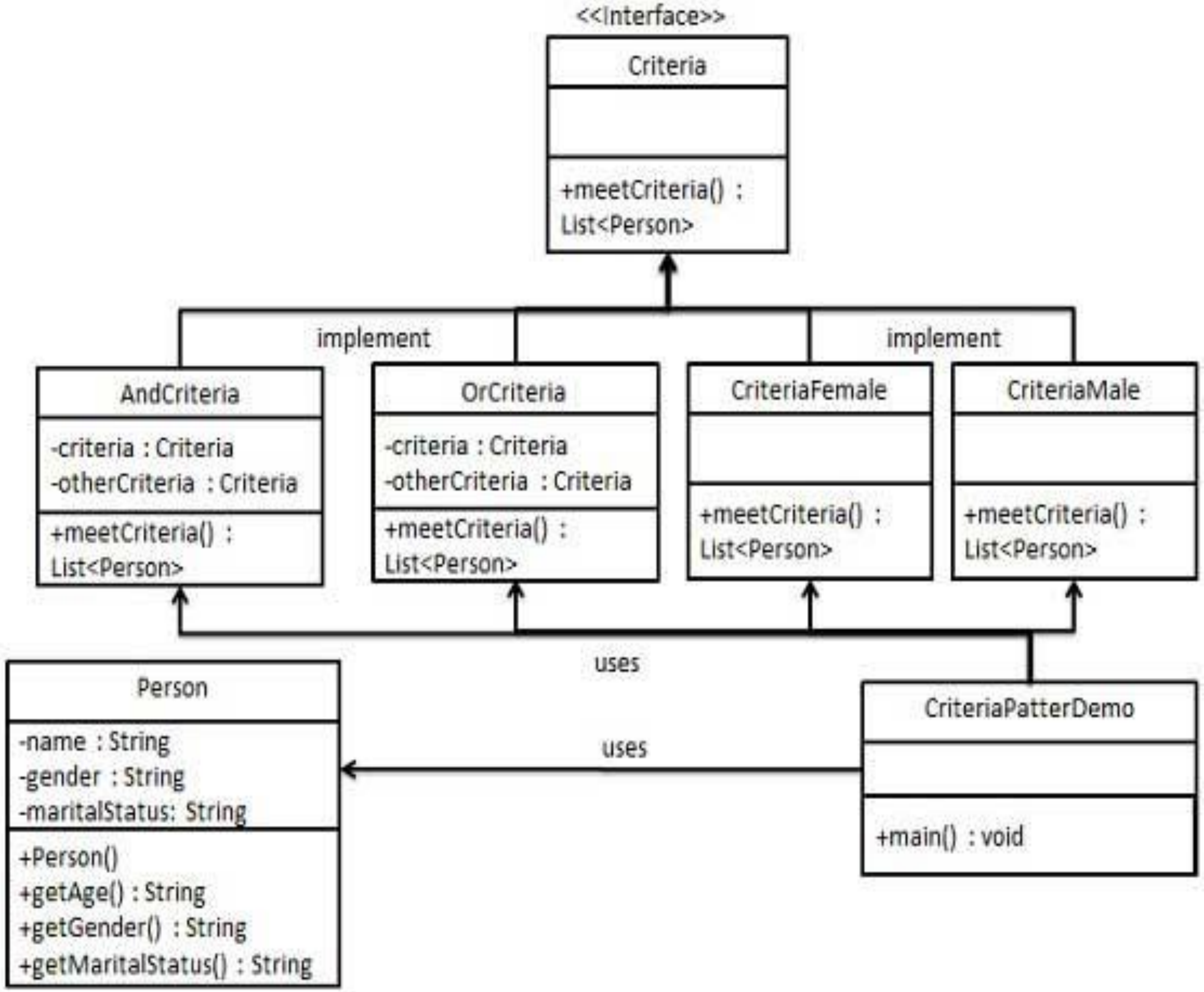
Usage

- Use when the search results for a query are very numerous and reviewing them would be very time consuming.
- Use when search results can be categorized into filters: the search must be contextual.
- Do not use when your search is not easily categorized into filters.

FILTER

Example

Filter a list of persons



FILTER.EXAMPLE

```
public class Person {
   private String name;
   private String gender;
   private String maritalStatus;
   public Person(String name, String gender, String maritalStatus){
      this.name = name;
      this.gender = gender;
      this.maritalStatus = maritalStatus;
   }
public String getName() {
      return name;
   }
   public String getGender() {
      return gender;
   }
   public String getMaritalStatus() {
      return maritalStatus;
   }
}
```

FILTER.EXAMPLE

```
public interface Criteria {
   public List<Person> 
meetCriteria(List<Person> persons);
}
```

```
public class CriteriaMale implements Criteria {
   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> malePersons = new ArrayList<Person>();
      for (Person person : persons) {
         if(person.getGender().equalsIgnoreCase("MALE")){
            malePersons.add(person);
         }
      }
      return malePersons;
   }
}
```

FILTER.EXAMPLE

```
public class CriteriaFemale implements Criteria {
   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> femalePersons = new ArrayList<Person>();
      for (Person person : persons) {
         if(person.getGender().equalsIgnoreCase("FEMALE")){
            femalePersons.add(person);
         }
      }
      return femalePersons;
   }
}
```

FILTER.EXAMPLE

```
public class CriteriaSingle implements Criteria {
   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> singlePersons = new ArrayList<Person>();
      for (Person person : persons) {
         if(person.getMaritalStatus().equalsIgnoreCase("SINGLE")){
            singlePersons.add(person);
         }
      }
      return singlePersons;
   }
}
```

FILTER.EXAMPLE

```
public class AndCriteria implements Criteria {
  private Criteria criteria;
  private Criteria otherCriteria;
  public AndCriteria(Criteria criteria,
                    Criteria otherCriteria) {
    this.criteria = criteria;
    this.otherCriteria = otherCriteria;
  }
```

```
@Override
  public List<Person> meetCriteria(List<Person> persons) {

    List<Person> firstCriteriaPersons =
            criteria.meetCriteria(persons);
    return
          otherCriteria.meetCriteria(firstCriteriaPersons);
  }
}
```

FILTER.EXAMPLE

```
public class OrCriteria implements Criteria
{
  private Criteria criteria;
   private Criteria otherCriteria;
   public OrCriteria(Criteria criteria,
                 Criteria otherCriteria) {
      this.criteria = criteria;
      this.otherCriteria = otherCriteria;
   }
```

```
@Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> firstCriteriaItems =
criteria.meetCriteria(persons);
      List<Person> otherCriteriaItems =
                        otherCriteria.meetCriteria(persons);
      for (Person person : otherCriteriaItems) {
         if(!firstCriteriaItems.contains(person)){
            firstCriteriaItems.add(person);
         }
      }
      return firstCriteriaItems;
   }}
```

FILTER.EXAMPLE

```
public class CriteriaPatternDemo {
   public static void main(String[] args) {
      List<Person> persons = new ArrayList<Person>();

      persons.add(new Person("Robert","Male", "Single"));
      persons.add(new Person("John", "Male", "Married"));
      persons.add(new Person("Laura", "Female", "Married"));
      persons.add(new Person("Diana", "Female", "Single"));
      persons.add(new Person("Mike", "Male", "Single"));
      persons.add(new Person("Bobby", "Male", "Single"));
```

```
Criteria male = new CriteriaMale();
Criteria female = new CriteriaFemale();
Criteria single = new CriteriaSingle();
Criteria singleMale = new AndCriteria(single, male);
Criteria singleOrFemale = new OrCriteria(single, female);

System.out.println("Males: ");
printPersons(male.meetCriteria(persons));
```

FILTER.EXAMPLE

```
 System.out.println("Males: ");
      printPersons(male.meetCriteria(persons));

      System.out.println("\nFemales: ");
      printPersons(female.meetCriteria(persons));

      System.out.println("\nSingle Males: ");
printPersons(singleMale.meetCriteria(persons));
```

```
       System.out.println("\nSingle Or Females: ");
      printPersons(singleOrFemale.meetCriteria(persons));
   }

public static void printPersons(List<Person> persons){

      for (Person person : persons) {
         System.out.println("Person : [ Name : " + person.getName() + ", 
Gender : " + person.getGender() + ", Marital Status : " + 
person.getMaritalStatus() + " ]");
      }
   }
}
```

FILTER.EXAMPLE

Males:

Person : [Name : Robert, Gender : Male, Marital Status : Single]

Person : [Name : John, Gender : Male, Marital Status : Married]

Person : [Name : Mike, Gender : Male, Marital Status : Single]

Person : [Name : Bobby, Gender : Male, Marital Status : Single]

Females:

Person : [Name : Laura, Gender : Female, Marital Status : Married]

Person : [Name : Diana, Gender : Female, Marital Status : Single]

Single Males:

Person : [Name : Robert, Gender : Male, Marital Status : Single]

Person : [Name : Mike, Gender : Male, Marital Status : Single]

Person : [Name : Bobby, Gender : Male, Marital Status : Single]

Single Or Females:

Person : [Name : Robert, Gender : Male, Marital Status : Single]

Person : [Name : Diana, Gender : Female, Marital Status : Single]

Person : [Name : Mike, Gender : Male, Marital Status : Single]

Person : [Name : Bobby, Gender : Male, Marital Status : Single]

Person : [Name : Laura, Gender : Female, Marital Status : Married]

FILTER

JDK example

- Stream API
- Criteria API JDBC

CURSE CONTENT

- Behavioral patterns
  - Chaim of responsibility
  - Strategy
- Partitioning patterns
  - Filter
  - Composite object
  - Read-only Interface

COMPOSITE OBJECT

Intent

- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- "Directories contain entries, each of which could be a directory."
- 1-to-many "has a" up the "is a" hierarchy

Problem

- Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

COMPOSITE OBJECT

Component

- Component is the abstraction for leafs and composites. It defines the interface that must be implemented by the objects in the composition.

Leaf

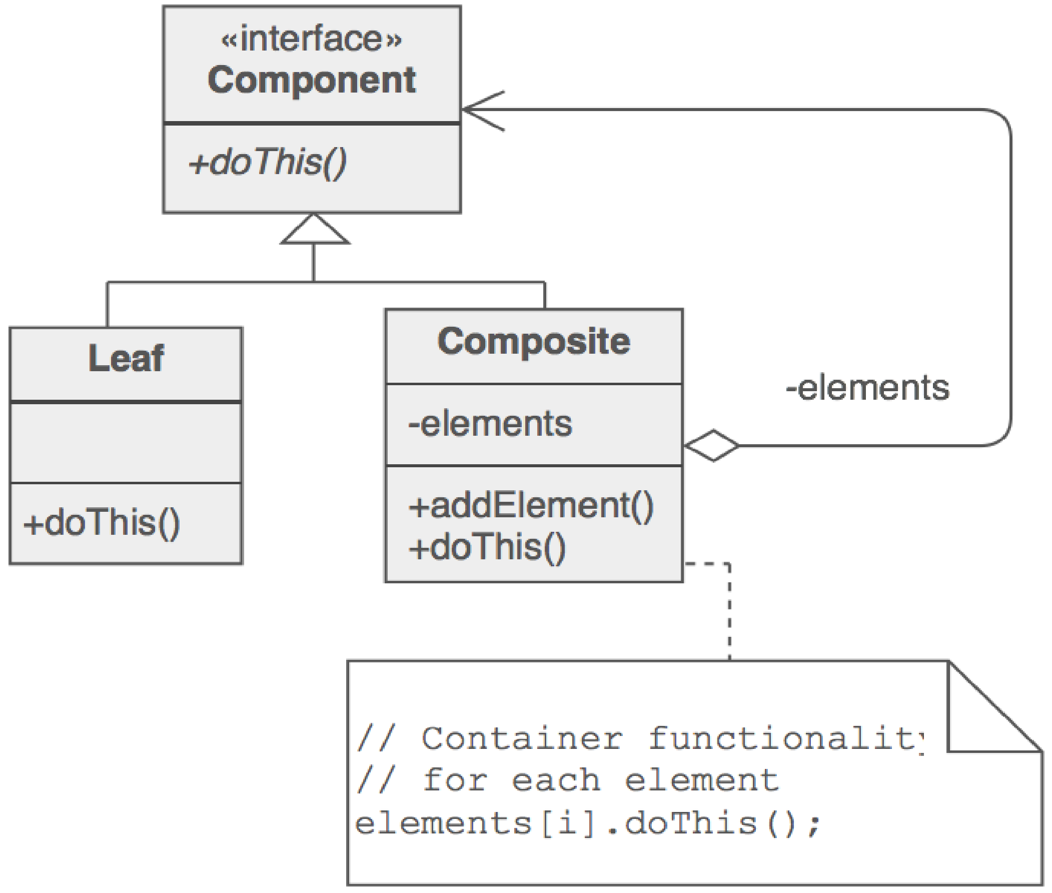
- Leafs are objects that have no children. They implement services described by the Component interface.

Composite

- A Composite stores child components in addition to implementing methods defined by the component interface. Composites implement methods defined in the Component interface by delegating to child components

Client

- The client manipulates objects in the hierarchy using the component interface.



```
// Container functionalit
// for each element
elements[i].doThis();
```

COMPOSITE OBJECT

Example

Graphics Drawing Editor

- In graphics editors a shape can be basic or complex. An example of a simple shape is a line, where a complex shape is a rectangle which is made of four line objects. Since shapes have many operations in common such as rendering the shape to screen, and since shapes follow a part-whole hierarchy, composite pattern can be used to enable the program to deal with all shapes uniformly.

File System

- A file system is a tree structure that contains Branches which are Folders as well as Leaf nodes which are Files. A folder object usually contains one or more file or folder objects and thus is a complex object where a file is a simple object. Files and folders have many operations and attributes in common, such as moving and copying a file or a folder, listing file or folder attributes such as file name and size, it would be easier and more convenient to treat both file and folder objects uniformly by defining a File System Resource Interface.

COMPOSITE OBJECT

Example

❑ Implement a file system

COMPOSITE OBJECT. WITHOUT APPLYING THE PATTERN

```
class File
{
    public File(String name)
    {
        m_name = name;
    }
    public void ls()
    {
        System.out.println(Composite.g_indent + m_name);
    }
    private String m_name;
}
```

COMPOSITE OBJECT. WITHOUT APPLYING THE PATTERN

```
class Directory {
    private String m_name;
    private ArrayList m_files = new ArrayList();
    public Directory(String name)
    {
        m_name = name;
    }
    public void add(Object obj)
    {
        m_files.add(obj);
    }
```

```
public void ls() {
        System.out.println(Composite.g_indent + m_name);
        Composite.g_indent.append("   ");
        for (int i = 0; i < m_files.size(); ++i) {
            Object obj = m_files.get(i);
            // Recover the type of this object
            if (obj.getClass().getName().equals("Directory")) ((Directory)obj).ls();
            else ((File)obj).ls();
        }
     Composite.g_indent.setLength(CompositeDemo.g_indent.length() - 3);
    }}
```

COMPOSITE OBJECT. WITHOUT APPLYING THE PATTERN

```
public class CompositeDemo
{
    public static StringBuffer g_indent = new StringBuffer();

    public static void main(String[] args)
    {
        Directory one = new Directory("dir111"),
                          two = new Directory("dir222"),
                          thr = new Directory("dir333");
        File a = new File("a"), b = new File("b"), c = new File("c"),
                  d = new File("d"), e = new File("e");
        one.add(a);
        one.add(two);
        one.add(b);
        two.add(c);
        two.add(d);
        two.add(thr);
        thr.add(e);
        one.ls();
    }
}
```

COMPOSITE OBJECT. APPLYING THE PATTERN

```
interface AbstractFile {
  public void ls();
}
class File implements AbstractFile {
    public File(String name) {
        m_name = name;
    }
    public void ls()  {
        System.out.println(CompositeDemo.g_indent + m_name);
    }
    private String m_name;
}
```

COMPOSITE OBJECT. APPLYING THE PATTERN

```
class Directory implements AbstractFile
{
    private String m_name;
    private ArrayList m_files = new ArrayList();
    public Directory(String name) {
        m_name = name;
    }
    public void add(Object obj)  {
        m_files.add(obj);
    }
```

```
public void ls()  {
        System.out.println(CompositeDemo.g_indent + m_name);
        CompositeDemo.g_indent.append("   ");
        for (int i = 0; i < m_files.size(); ++i) {
            // Leverage the "lowest common denominator"
            AbstractFile obj = (AbstractFile)m_files.get(i);
            obj.ls();
        }
CompositeDemo.g_indent.setLength(CompositeDemo.g_indent.length(
) - 3);
    }}
```

COMPOSITE OBJECT. APPLYING THE PATTERN

```
public class CompositeDemo
{
    public static StringBuffer g_indent = new StringBuffer();

    public static void main(String[] args)
    {
        Directory one = new Directory("dir111"),
          two = new Directory("dir222"),
          thr = new Directory("dir333");
        File a = new File("a"), b = new File("b"), c = new File("c"),
            d = new File("d"), e = new File("e");
one.add(a);
        one.add(two);
        one.add(b);
        two.add(c);
        two.add(d);
        two.add(thr);
        thr.add(e);
        one.ls();
    }
}
```

COMPOSITE OBJECT

- Defines class hierarchies consisting of primitive objects and composite objects.
  - Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.

- Makes the client simple.
  - Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component. This simplifies client code, because it avoids having to write tag-and-case-statementstyle functions over the classes that define the composition.

- Makes it easier to add new kinds of components.
  - Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes.

- Can make your design overly general.
  - The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

CURSE CONTENT

- Behavioral patterns
  - Chaim of responsibility
  - Strategy
- Partitioning patterns
  - Filter
  - Composite object
  - Read-only Interface

READ-ONLY INTERFACE

Intent

- An object can be modified by some of the clients and not by others

Problem

- How to create a situation where some classes see a class as read-only whereas others are able to make modifications?

READ-ONLY INTERFACE

Mutable

A class in the role has methods to get and set the values of its attributes. It also implements the ReadOnlyIf interface

ReadOnlyIf

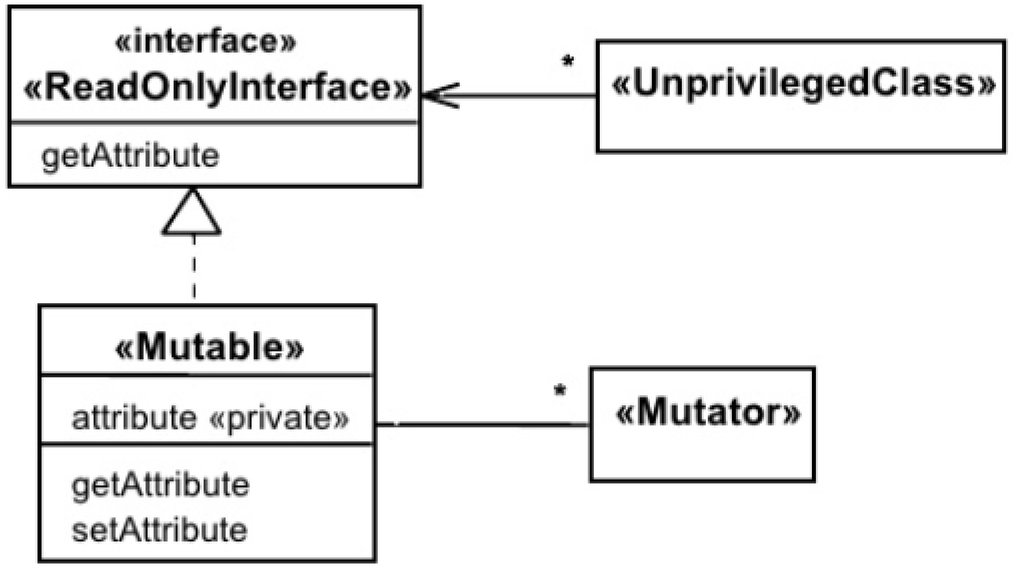
An interface in this role has the same get methods as the Mutable class, which implements the interface. However this interface does not include any methods that would cause a Mutable object to modify its content

MutatorClient

Classes in this role use the Mutable class directly and may call its methods that modify the state of a Mutable object.

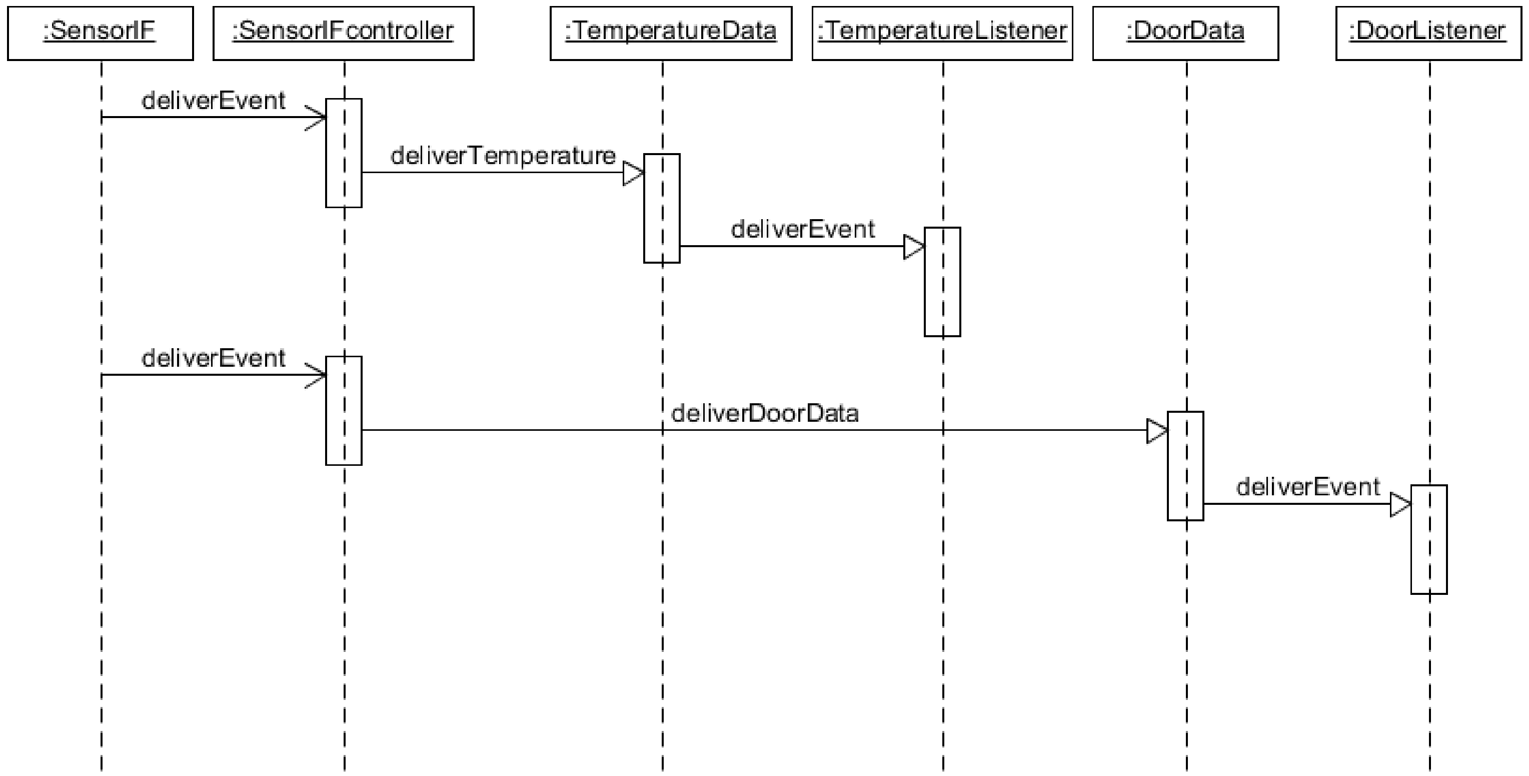
ReadOnlyClient

Classes in this role access the Mutable class through the ReadOnlyIf interface. Classes that access the Mutable class through the ReadOnlyIf interface are not able to access methods that modify Mutable object.



READ-ONLY INTERFACE

- You are designing software that will be part of a building security system. Part of its job is to monitor the state of physical sensors that detect the opening and closing of doors, the temperature of rooms and other such things. The sensors send messages to a computer running the security software. The security software may update a screen or take other actions as a result of receiving a message.



READ-ONLY INTERFACE

```
public interface TemperatureIF {
    /**  Return the temperature reading encapsulated in this object. */
    public double getTemperature() ;

    /** Add a listener to this object. */
    public void addListener(TemperatureIF listener) ;

   /** Remove a listener from this object; */
    public void removeListener(TemperatureIF listener) ;
    //…
}
```

READ-ONLY INTERFACE

```
class TemperatureData implements TemperatureIF {
    private double temperature;
    private ArrayList<TemperatureIF> listeners = new ArrayList<>();
    /** Set the temperature value stored in this object.  */
    public void setTemperature(double temperature) {
        this.temperature = temperature;
        fireTemperature();
    }
    /**Return the temperature reading encapsulated in this object. */
    public double getTemperature() {
        return temperature;
    }
```

```
/** Add a listener to this object. */
public void addListener(TemperatureIF listener)
{
    listeners.add(listener);
}

/** Remove a listener from this object; */
public void removeListener(
            TemperatureIF listener) {
    listeners.remove(listener);
}
```

READ-ONLY INTERFACE

```
/*** Send a TemperatureEvent to all registered TemperatureListener objects. */
    private void fireTemperature() {
        int count = listeners.size();
        TemperatureEvent evt;
        evt = new TemperatureEvent(this, temperature);
        for (int i = 0; i<count; i++) {
            TemperatureListener thisListener = (TemperatureListener)listeners.get(i);
            thisListener.temperatureChanged(evt) ;
        }
    } //…
}
```

READ-ONLY INTERFACE

- Using the Read-Only Interface pattern prevents programming mistakes that would result in classes being able to modify objects that they should not be able to modify.

- The Read-Only Interface pattern does not prevent objects from being improperly modified as a result of malicious programming.