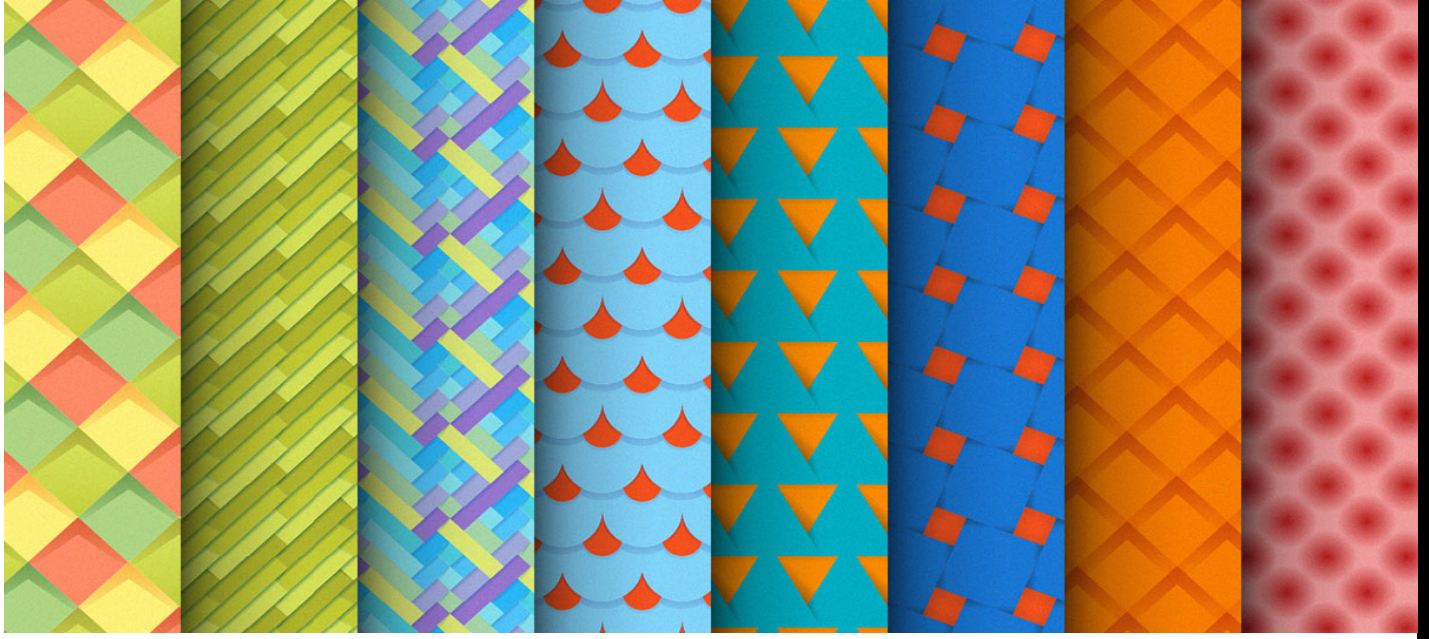


DESIGN PATTERNS

COURSE 5

PREVIOUS COURSE

- **Creational Patterns**
  - Adapter
    - interface converter
  - Bridge
    - decouple abstraction from its implementation
  - Façade
    - provide a unified interface to a subsystem
  - Flyweight
    - using sharing to support a large number of fine-grained objects efficiently
  - Proxy
    - provide a surrogate for another object to control access
  - Composite
    - compose objects into tree structures, treating all nodes uniformly
  - Decorator
    - attach additional responsibilities dynamically

CONTENT

- **Structural patterns**
  - Adapter
  - Bridge
  - Façade
  - Flyweight
  - Proxy
  - Composite
  - Decorator
- **Behavioral patterns**

STRUCTURAL PATTERNS

- **Help identify and describe relationships between entities**

- **Address how classes and objects are composed to form large structures**
  - Class-oriented patterns use inheritance to compose interfaces or implementations
  - Object-oriented patterns describe ways to compose objects to realize new functionality, possibly by changing the composition at run-time

- **Example**
  - Proxy in distributed programming
  - Bridge in JDBC drivers

STRUCTURAL PATTERNS

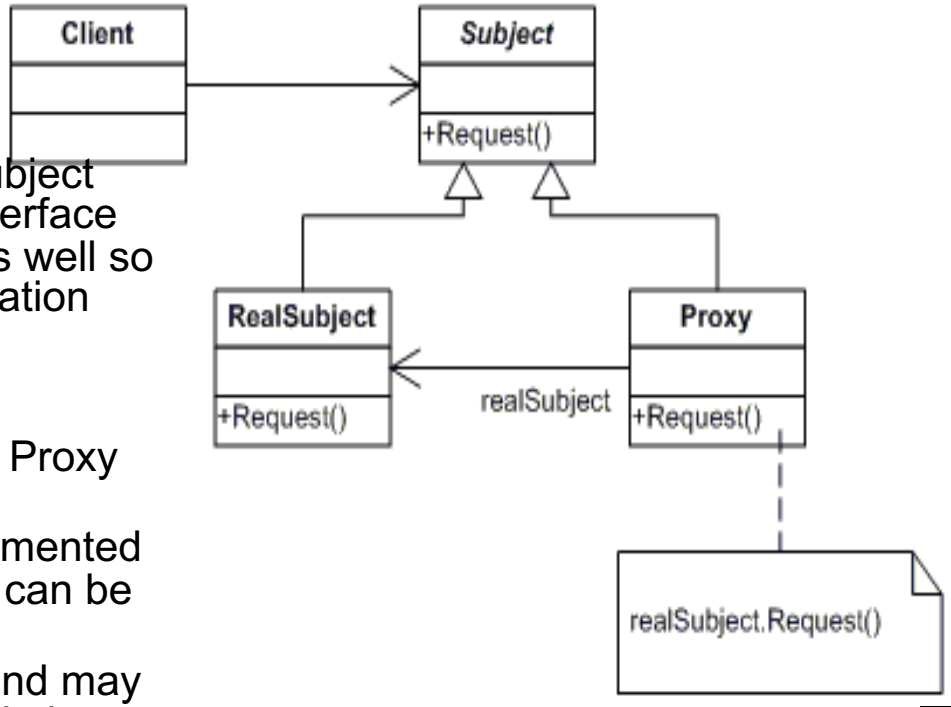
- **Adapter**
  - interface converter
- **Bridge**
  - decouple abstraction from its implementation
- **Façade**
  - provide a unified interface to a subsystem
- **Flyweight**
  - using sharing to support a large number of fine-grained objects efficiently
- **Proxy**
  - provide a surrogate for another object to control access
- **Composite**
  - compose objects into tree structures, treating all nodes uniformly
- **Decorator**
  - attach additional responsibilities dynamically

PROXY

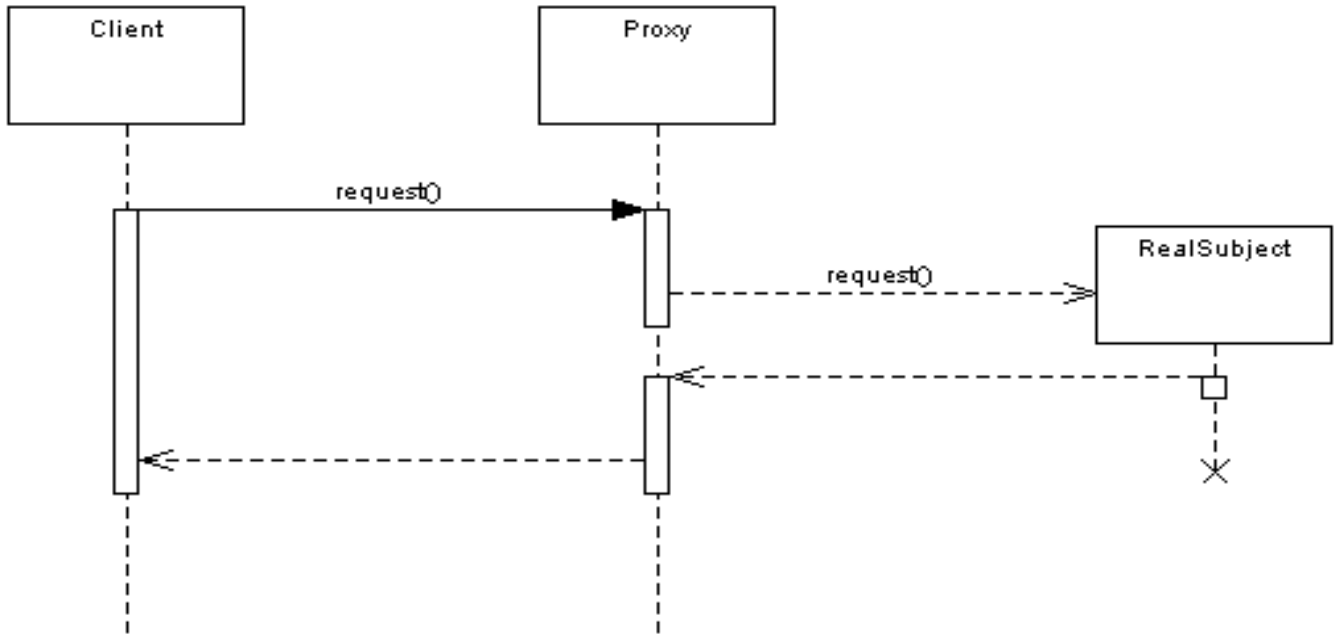
- **Intent**
  - Provide a surrogate or placeholder for another object to control access to it.
  - Use an extra level of indirection to support distributed, controlled, or intelligent access.
  - Add a wrapper and delegation to protect the real component from undue complexity.
- **Problem**
  - You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

PROXY. STRUCTURE

- **Subject**
  - Interface implemented by the RealSubject and representing its services. The interface must be implemented by the proxy as well so that the proxy can be used in any location where the RealSubject can be used.
- **Proxy**
  - Maintains a reference that allows the Proxy to access the RealSubject.
  - Implements the same interface implemented by the RealSubject so that the Proxy can be substituted for the RealSubject.
  - Controls access to the RealSubject and may be responsible for its creation and deletion.
  - Other responsibilities depend on the kind of proxy.
- **RealSubject**
  - The real object that the proxy represents



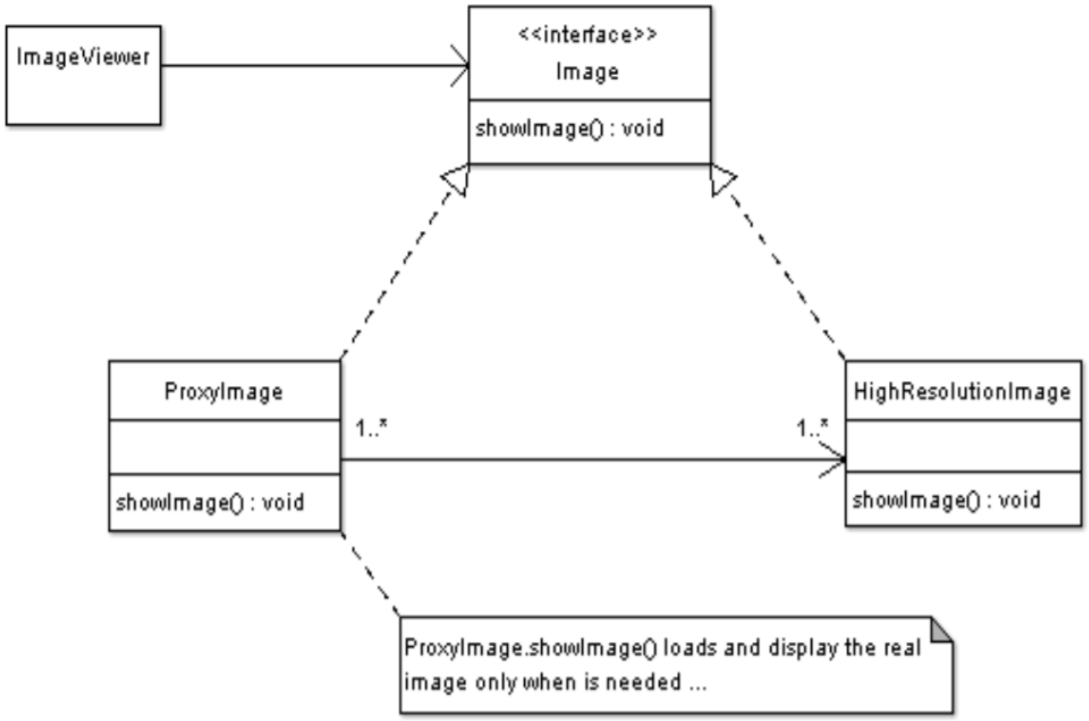
PROXY



Proxy is providing a barrier between the client and the real implementation.

PROXY. EXAMPLE

- **Image viewer program that lists and displays high resolution photos.**
  - The program has to show a list of all photos however it does not need to display the actual photo until the user selects an image item from a list.



PROXY. EXAMPLE

```
/**
* Subject Interface
*/
public interface Image {
    public void showImage();
}

/*** Proxy */
public class ImageProxy
                  implements Image {
    /** Private Proxy data */
    private String imageFilePath;

    /**  Reference to RealSubject */    }
     private Image proxifiedImage;
```

```
public ImageProxy(String imageFilePath) {
     this.imageFilePath= imageFilePath;
}

@Override
public void showImage() {
      // create the Image Object only when the
      // image is  required to be shown
     proxifiedImage = new
           HighResolutionImage(imageFilePath);
      // now call showImage on realSubject
      proxifiedImage.showImage();
}
```

PROXY. EXAMPLE

```
/**
RealSubject
*/
public class HighResolutionImage implements Image {
    public HighResolutionImage(String imageFilePath) {
                loadImage(imageFilePath);
    }
    private void loadImage(String imageFilePath) {
           // load Image from disk into memory
           // this is heavy and costly operation
    }
   @Override
    public void showImage() {
             // Actual Image rendering logic
    }
 }
```

PROXY. EXAMPLE

```
/** * Image Viewer program */
public class ImageViewer {
 public static void main(String[] args) {
   // assuming that the user selects a folder
   // that has 3 images
   //create the 3 images
   Image highResolutionImage1 =
      new ImageProxy(”img/veryHighResPhoto1.jpeg");
   Image highResolutionImage2 =
      new ImageProxy(”img/veryHighResPhoto2.jpeg");
   Image highResolutionImage3 =
      new ImageProxy(”img/veryHighResPhoto3.jpeg");
// assume that the user clicks on Image one item
//in a list
// this would cause the program to call showImage()
// for that image only
// note that in this case only image one
// was loaded into memory
   highResolutionImage1.showImage();
// consider using the high resolution image object
directly
Image highResolutionImageNoProxy1 = new
  HighResolutionImage(”img/veryHighResPhoto1.jpeg");
Image highResolutionImageNoProxy2 =  new
  HighResolutionImage(”img/veryHighResPhoto2.jpeg");
Image highResolutionImageBoProxy3 = new
  HighResolutionImage(”img/veryHighResPhoto3.jpeg");
// assume that the user selects image two item
 // from images list
highResolutionImageNoProxy2.showImage();
// note that in this case all images have
 // been loaded into memory
// and not all have been actually displayed
// this is a waste of memory resources
}
 }
```

PROXY. EXAMPLE

- **Java API Usage**
  - “Remote Method Invocation” (java.rmi library)
  - Allows objects in separate virtual machines to be used as if local (language specific)

- **Security Proxies that controls access to objects can be found in many object oriented languages including java, C#, C++**

PROXY. TYPES

- **Remote Proxy**
  - Provides a reference to an object located in a different address space on the same or different machine
- **Virtual Proxy**
  - Allows the creation of a memory intensive object on demand. The object will not be created until it is really needed.
- **Copy-On-Write Proxy**
  - Defers copying (cloning) a target object until required by client actions. Really a form of virtual proxy.
- **Protection (Access) Proxy**
  - Provides different clients with different levels of access to a target object
- **Cache Proxy**
  - Provides temporary storage of the results of expensive target operations so that multiple clients can share the results
- **Firewall Proxy**
  - Protects targets from bad clients
- **Synchronization Proxy**
  - Provides multiple accesses to a target object
- **Smart Reference Proxy**
  - Provides additional actions whenever a target object is referenced such as counting the number of references to the object

PROXY

- **When to use**
  - The object being represented is external to the system.
  - Objects need to be created on demand.
  - Access control for the original object is required
  - Added functionality is required when an object is accessed

STRUCTURAL PATTERNS

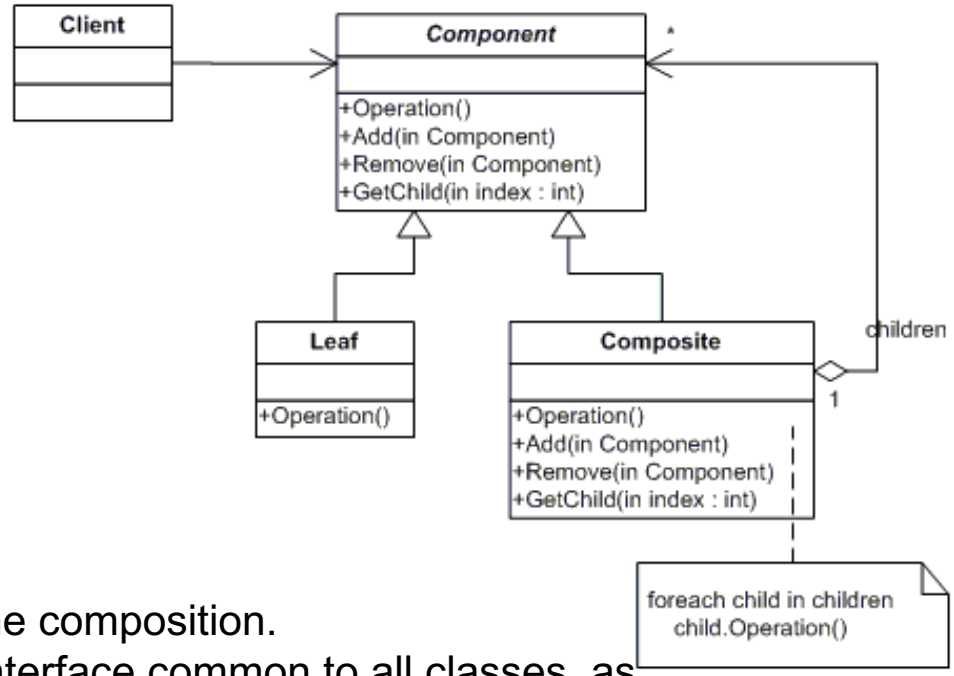
- **Adapter**
  - interface converter
- **Bridge**
  - decouple abstraction from its implementation
- **Façade**
  - provide a unified interface to a subsystem
- **Flyweight**
  - using sharing to support a large number of fine-grained objects efficiently
- **Proxy**
  - provide a surrogate for another object to control access
- **Composite**
  - compose objects into tree structures, treating all nodes uniformly
- **Decorator**
  - attach additional responsibilities dynamically

COMPOSITE

- **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. STRUCTURE



- **Component**
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- **Leaf**
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.
- **Composite**
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- **Client**
  - manipulates objects in the composition through the Component interface.

COMPOSITE. EXAMPLE

- **File System**
- **XML**
- **Html tags**
- **Hierarchy of an office**
  - starting from the president to employees

COMPOSITE. EXAMPLE

- **File System**
  - Stating from the following abstraction how you would refectory in order to follow the composite 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. EXAMPLE

```
class Directory {
     public Directory(String name) { m_name = name; }
     public void add(Object obj) { m_files.add(obj); }
     public void ls() {
         System.out.println(m_name);
          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();
          }
       }
       private String m_name;
       private ArrayList<Object> m_files = new ArrayList<>();
}
```

COMPOSITE. EXAMPLE

```
class CompositeDemo {
    public static void main(String[] args) {
        Directory one = new Directory("dir111");
        Directory two = new Directory("dir222");
        Directory thr = new Directory("dir333");
        File a = new File("a"), b = new File("b");
        File 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. EXAMPLE. REFACTOR

```
interface AbstractFile { public void ls(); }

class File implements AbstractFile { … }

class Directory implements AbstractFile {
    public void ls() {
         for (int i = 0; i < m_files.size(); ++i) {
              // Leverage the "lowest common denominator"
              AbstractFile obj = m_files.get(i);
             obj.ls();
          }
   private ArrayList< AbstractFile> m_files = new ArrayList<>();
…
}
```

COMPOSITE

- **Propose a implementation of composite pattern for the following examples**
  - Html tags
  - Hierarchy of an office
    - starting from the president to employees

COMPOSITE

- **When to use Composite Pattern?**
  - When you want to represent part-whole hierarchies of objects.
  - When you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

STRUCTURAL PATTERNS

- **Adapter**
  - interface converter
- **Bridge**
  - decouple abstraction from its implementation
- **Façade**
  - provide a unified interface to a subsystem
- **Flyweight**
  - using sharing to support a large number of fine-grained objects efficiently
- **Proxy**
  - provide a surrogate for another object to control access
- **Composite**
  - compose objects into tree structures, treating all nodes uniformly
- **Decorator**
  - attach additional responsibilities dynamically

DECORATOR

- **Intent**
  - Attach additional responsibilities to an object dynamically.
  - Decorators provide a flexible alternative to subclassing for extending functionality.

- **Problem**
  - Want to add properties to an existing object
  - Add borders or scrollbars to a GUI component
  - Add headers and footers to an advertisement
  - Add stream functionality such as reading a line of input or compressing a file before sending it over the wire

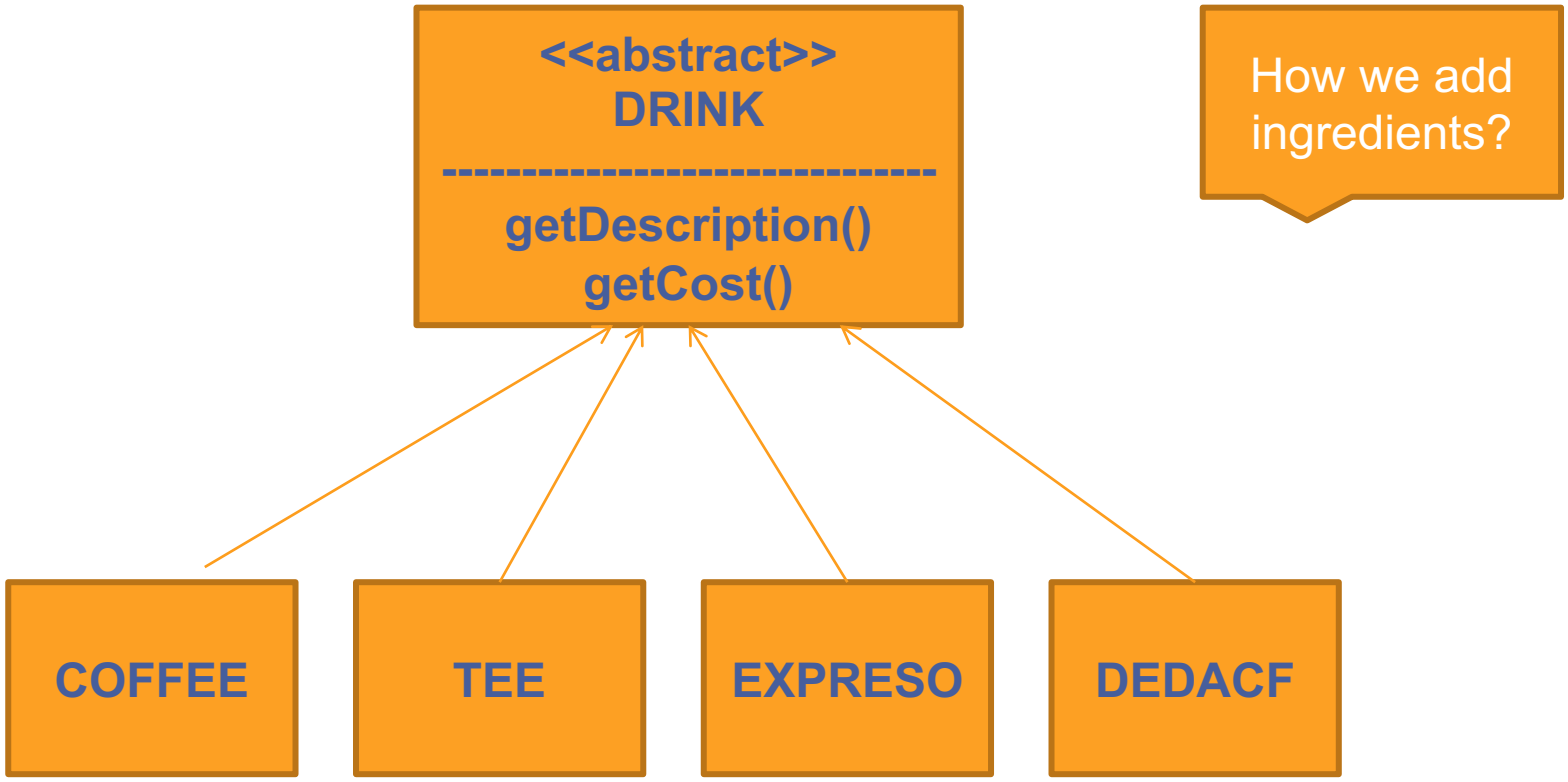
DECORATOR

- **How could we design the following example?**
  - An automated machine that prepares drinks.
    - For a drink we have the following base drinks: coffee, tee, espresso, decaf
    - And the following ingredients: milk, mocha, rum
  - Based on the base drink and ingredients that are chose in order to prepare a drink the price of the drink varies

DECORATOR

Drink classes hierarchy

DECORATOR



Drink classes hierarchy, adding methods to display the description and cost

DECORATOR

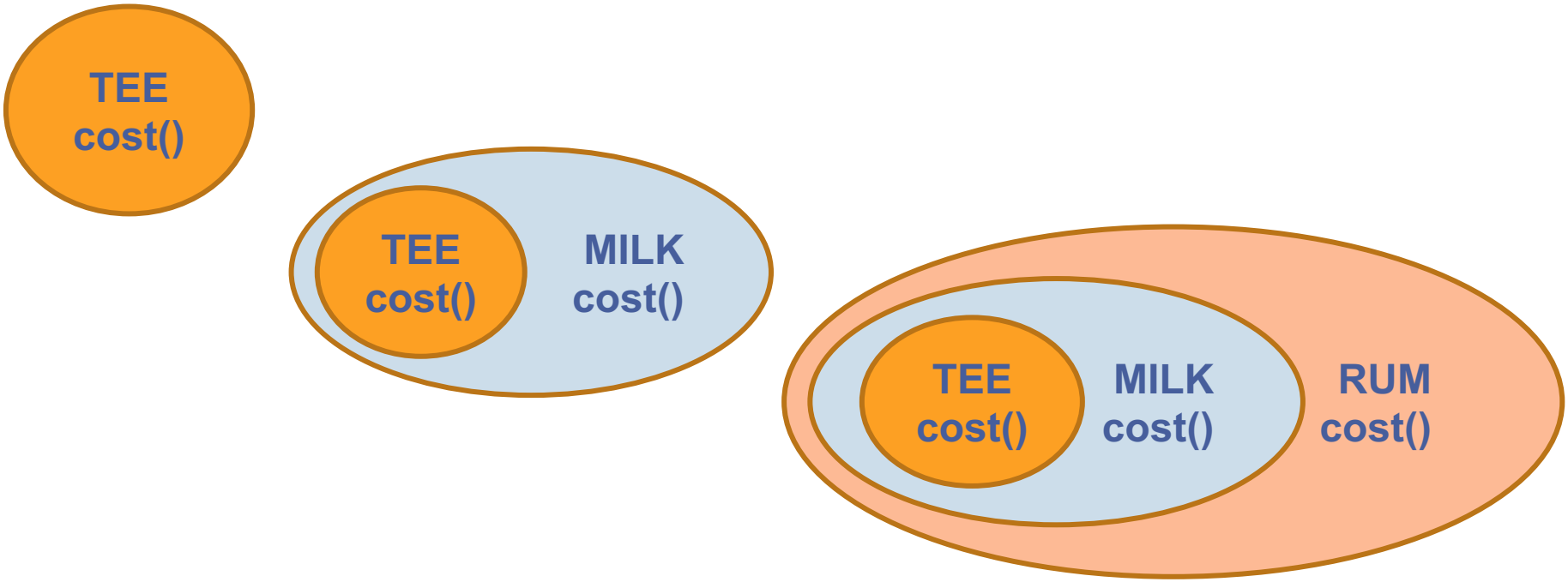
- **How to add ingredients?**
  - Member fields to Drink class
    - Possible problems?
  - Other variants?

DECORATOR

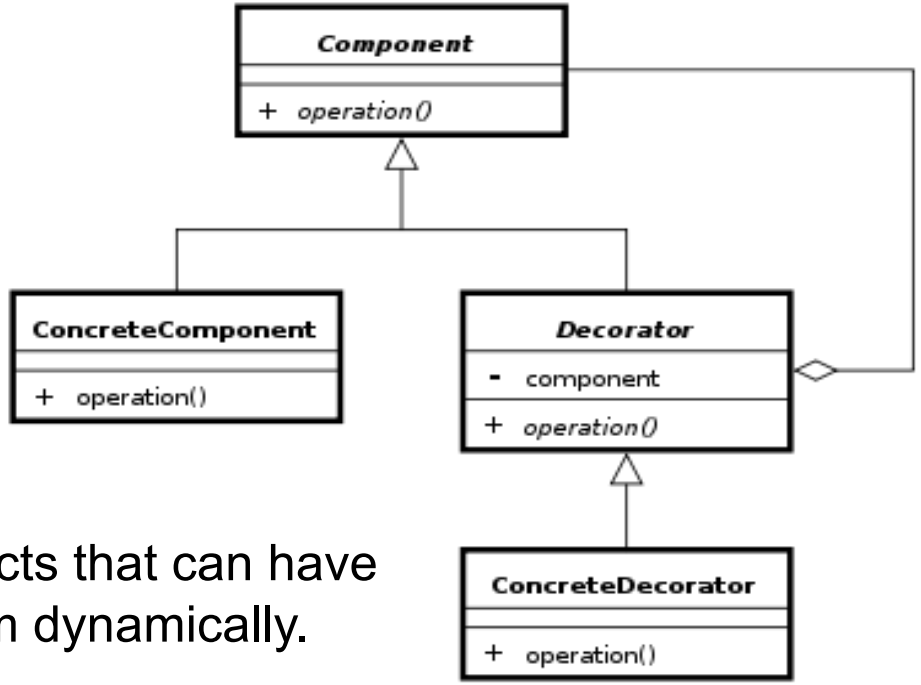
- **How to add ingredients?**
  - Member fields to Drink class
    - Possible problems?
      - Code changes in the superclass when a new ingredient is added
      - Prices can change => code change
      - Double the amount of milk
  - Other variants?

DECORATOR

- **How to add ingredients?**
  - Solution using decorator
    - start from a base drink and add ingredients

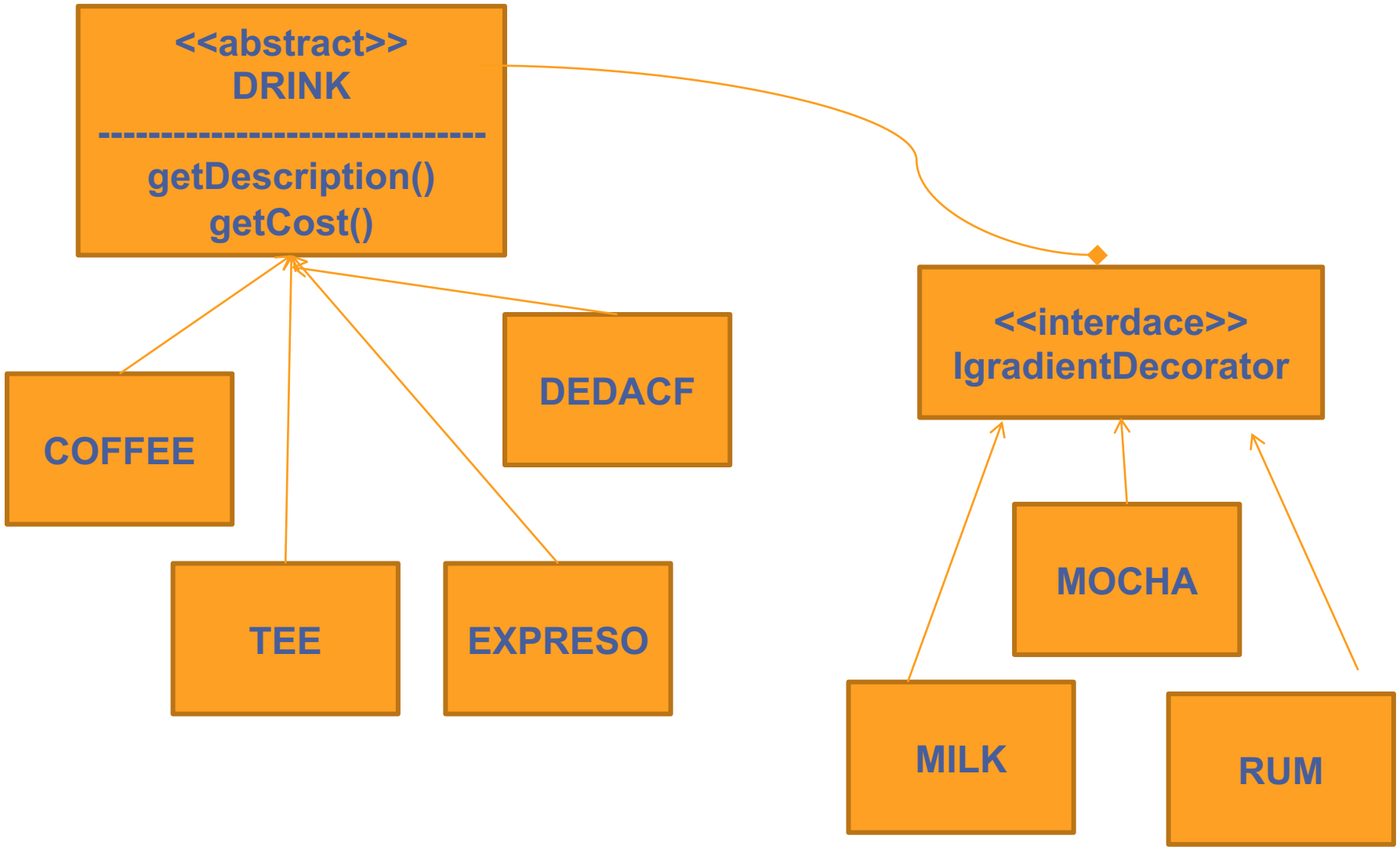


DECORATOR. STRUCTURE



- **Component**
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent**
  - defines an object to which additional responsibilities can be attached.
- **Decorator**
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator**
  - adds responsibilities to the component.

DECORATOR. EXAMPLE



DECORATOR. EXAMPLE

DRINK

```
public abstract class Drink{
  String description = "Unknown Drink";
  public String  getDescription () {
            // already implemented
            return description;
  }

  public abstract double cost();
  // Need to implement cost()
}
```

INGREDIENT

```
public abstract
    class  IngredientDecorator
                  extends  Drink{

  Drink drink;
  public Drink( Drink drink) {
       this.drink = drink;
  }

}
```

DECORATOR. EXAMPLE

DRINK

```
public class Tee  extends Drink{
  public Tee() {
    description = "Tee";
  }
  public double cost() {
     return .89;
  }
}
```

INGREDIENT

```
public class Mocha
                  extends IngredientDecorator{
  public Mocha( Drink drink) {
      super (drink);
  }
   public String getDescription () {
       return  drink.getDescription() +
                                      ", Mocha";
   }
    public double cost() {
        return .20 + drink.cost ();
    }
}
```

DECORATOR. EXAMPLE

```
public class Client{
 public static void main(String args []) {
  // espresso order, no condiments
  Drink drink=  new Espresso();
  System.out.println(drink.getDescription() + " $" + drink.cost());

  Drink drink2 = new Coffee(); //get a Coffee with milk
  drink2 = new Milk(drink2);
  drink2 = new Milk(drink2); // wrap it with Milk
  System.out.println(drink2.getDescription() +" $" +drink2.cost());

  Drink drink3 = new Tee(); // get a Tee
  drink3 = new Milk(drink3); // wrap with Milk
  drink3 = new Rum(drink3); // wrap with Rum
  System.out.println(drink3.getDescription() + " $" + drink3.cost())
}
}
```

DECORATOR. EXAMPLES

- **Java I/O**
  - InputStreamReader decorates InputStream
    - Bridge from byte streams to character streams: It reads bytes and translates them into characters using the specified character encoding
  - BufferReader decorates InputStreamReader
    - Read text from a character - input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
    - `BufferedReader keyboard = new BufferedReader(new InputStreamReader(System.in));`

- **Java Swing**
  - Any `JComponent` can have 1 or more borders
    - Borders are useful objects that, while not themselves components, know how to draw the edges of Swing components
    - Borders are useful not only for drawing lines and fancy edges, but also for providing titles and empty space around components

DECORATOR

- **Advantages**
  - It is flexible than inheritance because inheritance adds responsibility at compile time but decorator pattern adds at run time.
  - We can have any number of decorators and also in any order.
  - It extends functionality of object without affecting any other object

- **Disadvantages**
  - The main disadvantage of decorator design pattern is code maintainability because this pattern creates lots of similar decorators which are sometimes hard to maintain and distinguish.

BEHAVIORAL PATTERNS

- **Behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns**

- **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

BEHAVIORAL PATTERNS

- **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

BEHAVIORAL PATTERNS

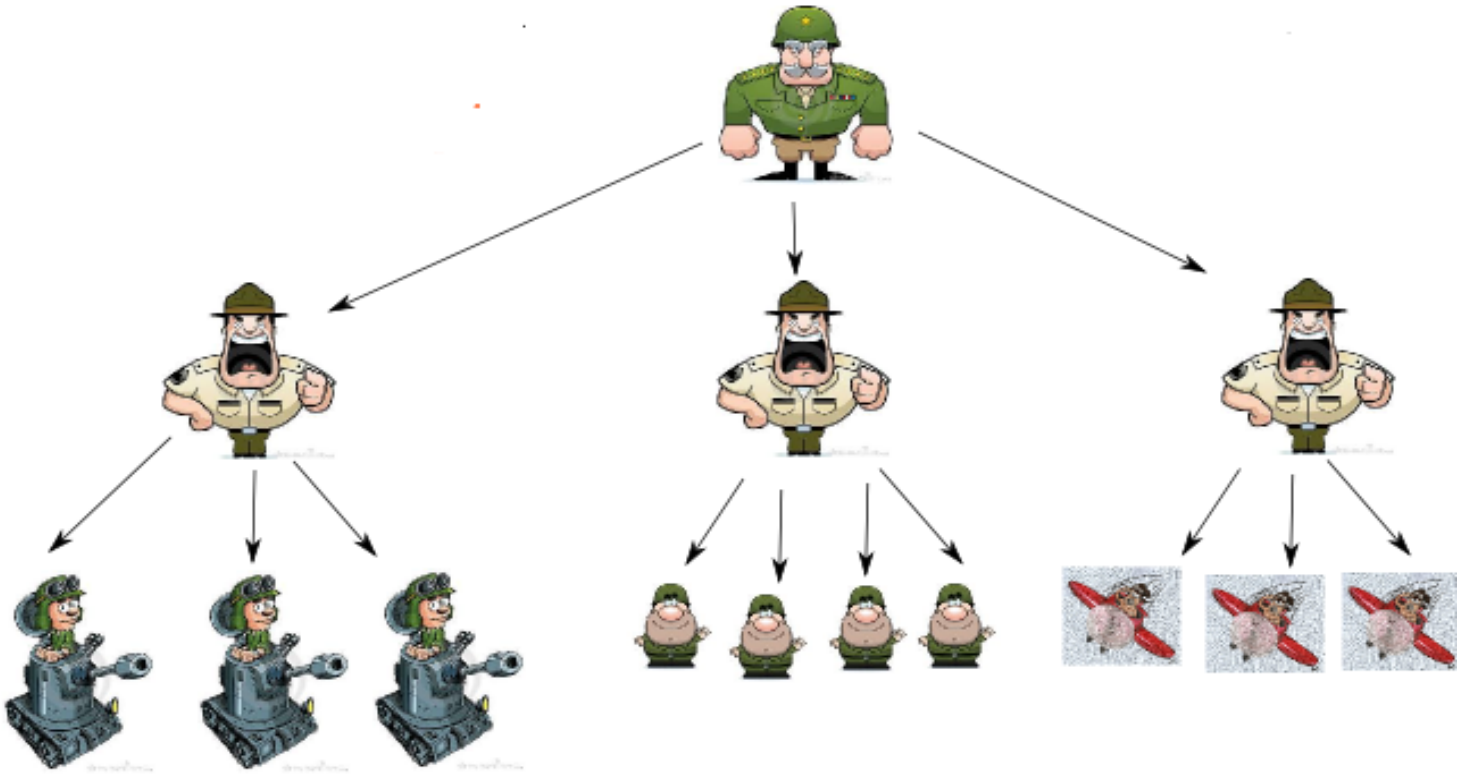
- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - 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

CHAIN 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.

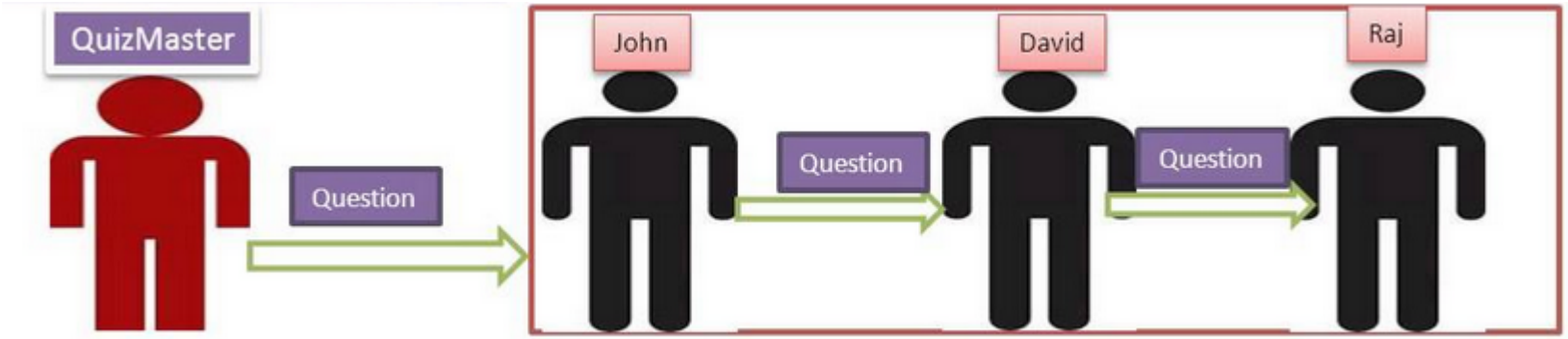
- **Problem**
  - There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings

CHAIN OF RESPONSIBILITY

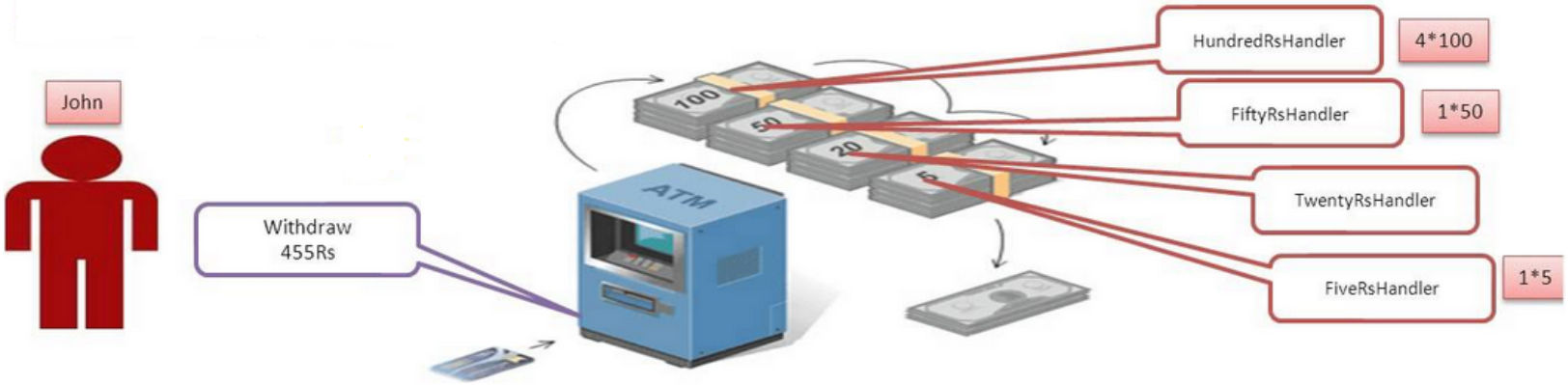


CHAIN OF RESPONSIBILITY

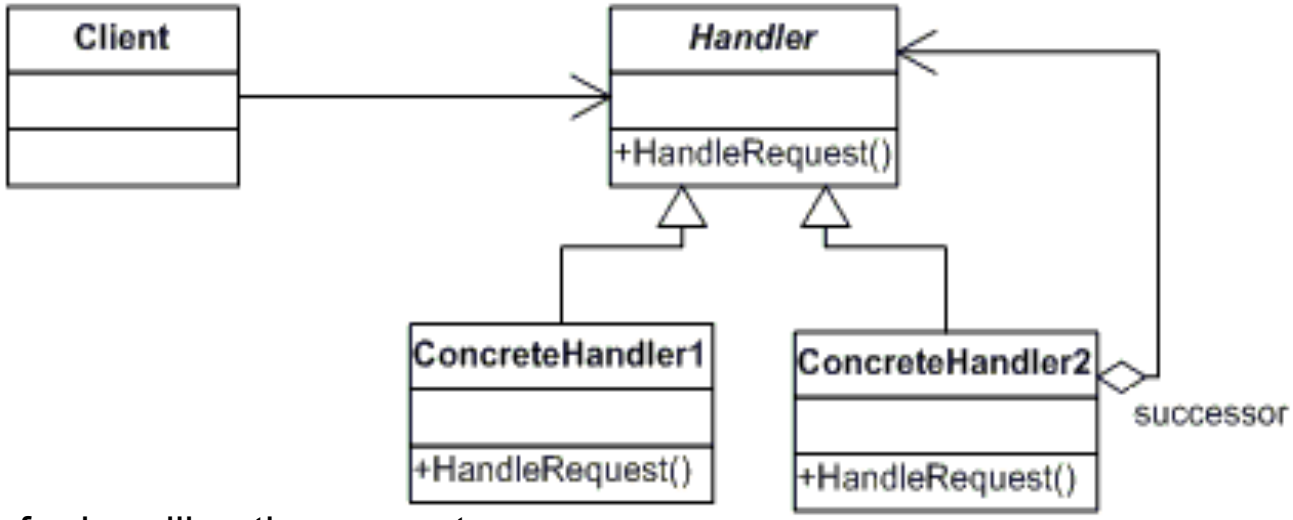
- Only one receiver in the chain handles the Request



- One or more Receivers in the chain handles the Request



CHAIN 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

CHAIN 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.

CHAIN OF RESPONSIBILITY

- **Simple chain of responsibility example**
  - Transmitting a message to a planet

CHAIN OF RESPONSIBILITY

```
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;
}
```

CHAIN OF RESPONSIBILITY

```
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);
         }
      }
    }
}
```

CHAIN OF RESPONSIBILITY

```
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);
                 }
            }
   }
}
```

CHAIN OF RESPONSIBILITY

```
public class EarthHandler extends PlanetHandler {

    public void handleRequest(PlanetEnum request) {
            if (request == PlanetEnum.EARTH) {
                System.out.println("EarthsHandler handles " + request);
                System.out.println("Earth is comfortable.\n");

            } else {
                 System.out.println("EarthHandler doesn't handle " + request);
                 if (successor != null) {
                    successor.handleRequest(request);
                }
            }
   }
}
```

CHAIN OF RESPONSIBILITY

CLIENT

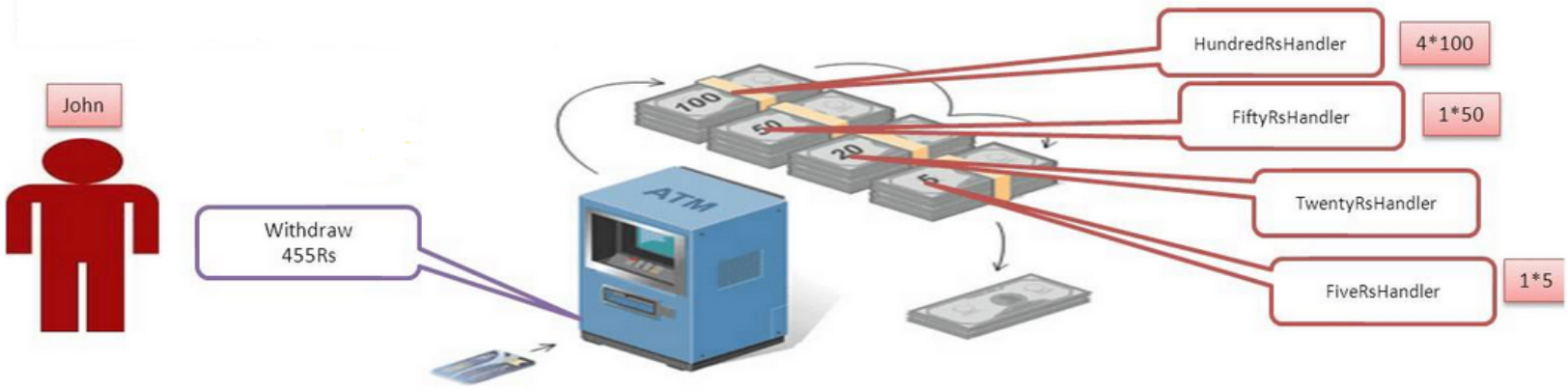
```
public class Demo {
  public static void main(String[] args) {
    PlanetHandler chain = setUpChain();
    chain.handleRequest(PlanetEnum.VENUS);
    chain.handleRequest(PlanetEnum.MERCURY);
    chain.handleRequest(PlanetEnum.EARTH);
    chain.handleRequest(PlanetEnum.JUPITER);
  }
  public static PlanetHandler setUpChain() {
     PlanetHandler mercuryHandler = new MercuryHandler();
     PlanetHandler venusHandler = new VenusHandler();
     PlanetHandler earthHandler = new EarthHandler();
     mercuryHandler.setSuccessor(venusHandler);
     venusHandler.setSuccessor(earthHandler);
     return mercuryHandler;
}
```

OUTPUT

```
MercuryHandler doesn't handle VENUS
VenusHandler handles VENUS
Venus is poisonous.

MercuryHandler handles MERCURY
Mercury is hot.

MercuryHandler doesn't handle EARTH
VenusHandler doesn't handle EARTH
EarthHandler handles EARTH
Earth is comfortable.

 MercuryHandler doesn't handle JUPITER
VenusHandler doesn't handle JUPITER
EarthHandler doesn't handle JUPITER
```

CHAIN OF RESPONSIBILITY

- Propose a chain of responsibility pattern implementation for ATM problem



CHAIN 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)

CHAIN OF RESPONSIBILITY

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

BEHAVIORAL PATTERNS

- **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

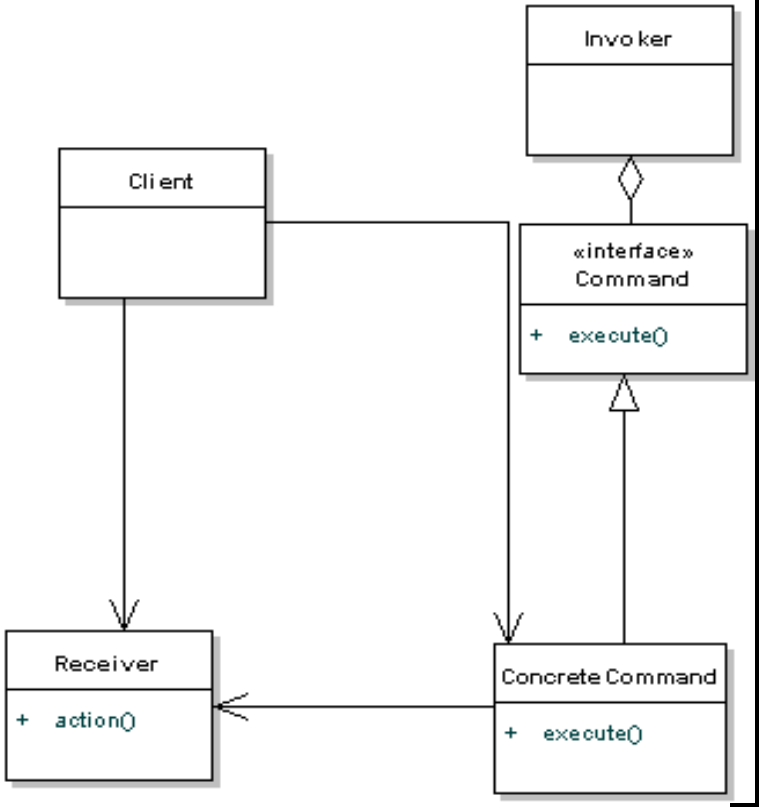
COMMAND

- **Intent**
  - encapsulate a request in an object
  - allows the parameterization of clients with different requests
  - allows saving the requests in a queue

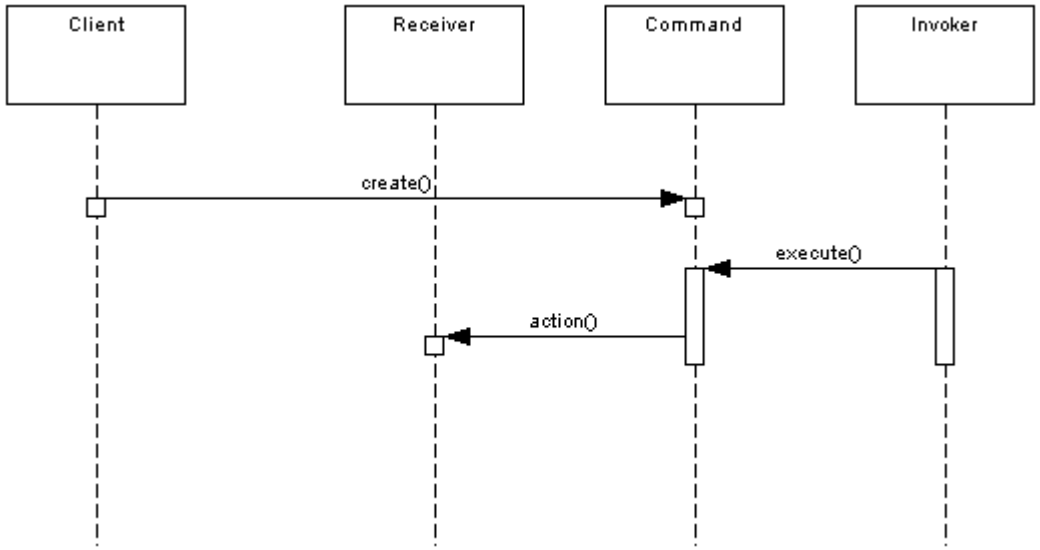
- **Problem**
  - Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request

COMMAND

- **Command**
  - declares an interface for executing an operation
- **ConcreteCommand**
  - defines a binding between a `Receiver` object and an action
  - implements *`execute()`* by invoking the corresponding operation(s) on `Receiver`
- **Client**
  - creates a `ConcreteCommand` object and sets its receiver
- **Invoker**
  - asks the command to carry out the request
- **Receiver**
  - knows how to perform the operations associated with carrying out the request.



COMMAND



- The client (main program) creates a concrete Command object and sets its Receiver.
- The Invoker issues a request by calling execute on the Command object. The concrete Command object invokes operations on its Receiver to carry out the request.
- The key idea here is that the concrete command registers itself with the Invoker and the Invoker calls it back, executing the command on the Receiver.

COMMAND

- **Example**
  - Adding actions to menus in java
    - Create a class that Extends ActionListerner interface and overwrite `actionPerformed ()` method

```
puclic class MyActionHandler extends ActionListener {
  public void actionPerformed(ActionEvent e) {
    Object o = e.getSource();
    if (o = fileNewMenuItem)  doFileNewAction();
    else if (o = fileOpenMenuItem) doFileOpenAction();
    else if (o = fileOpenRecentMenuItem) doFileOpenRecentAction();
    else if (o = fileSaveMenuItem) doFileSaveAction();
    // and more ... }
}

FileOpenMenuItem fomi = new FileOpenMenuItem("OpenFile")
fomi.addActionListener(new MyActionHandler());
```

COMMAND

- **Example**
  - Adding actions to menus in java
    - If we follow the command pattern first we create a command and after that each menu entry will implement the command

```
public interface Command { public void execute(); }

public class FileOpenMenuItem extends JMenuItem implements
  Command {
       public void execute() { // your business logic goes here }
}

FileOpenMenuItem fomi = new FileOpenMenuItem("OpenFile")
fomi.addActionListener(e->{
                Command command = (Command)e.getSource();
                command.execute();
});
```

COMMAND

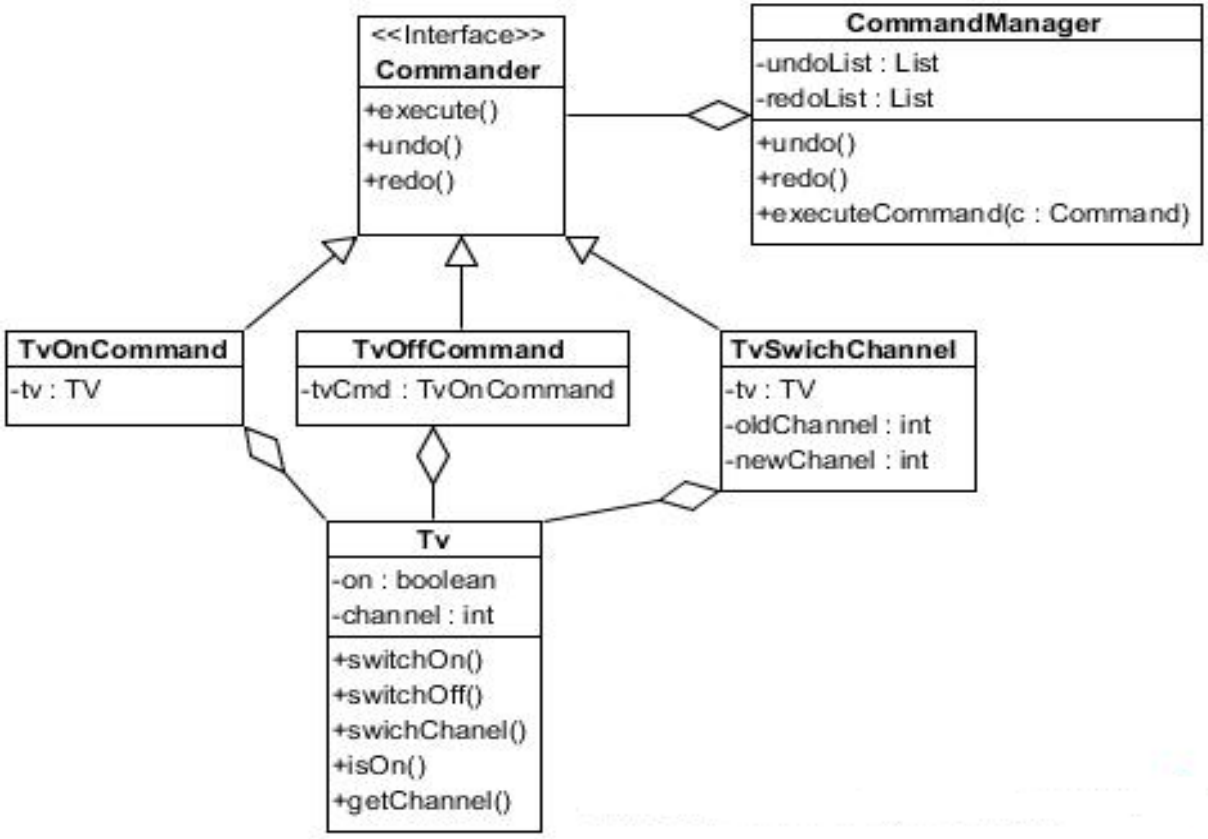
- **Java API examples**
  - ActionListener
  - Comparator
  - Runnable / Thread

COMMAND

- Applicability
  - Parameterizes objects depending on the action they must perform
  - Specifies or adds in a queue and executes requests at different moments in time
  - Offers support for undoable actions (the Execute method can memorize the state and allow going back to that state)
  - Structures the system in high level operations that based on primitive operations
  - Decouples the object that invokes the action from the object that performs the action. Due to this usage it is also known as Producer - Consumer design pattern.

COMMANDER. EXERCISE

- **Implement**
  - Undo/redo operation for a TV remote stating from the following class diagram



COMMAND

- **Advantages**
  - Command decouples the object that invokes the operation from the one that knows how to perform it.
  - Commands are first-class objects. They can be manipulated and extended like any other object.
  - You can assemble commands into a composite command. In general, composite commands are an instance of the Composite pattern.
  - It's easy to add new Commands, because you don't have to change existing classes.

- **Disadvantages**
  - Proliferation of little classes, that are more readable