DESIGN PATTERNS

Course 5

PREVIOUS COURSE CONTENT

- Structural patterns
  - Adapter
  - Bridge
  - Façade
  - Flyweight
- Behavioral patterns
  - Iterator

CURRENT CURSE CONTENT

- Structural patterns
  - Decorator
  - Proxy
- Behavioral patterns
  - State
  - Visitor
  - Null Object

DECORATOR

Intent

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

Problem

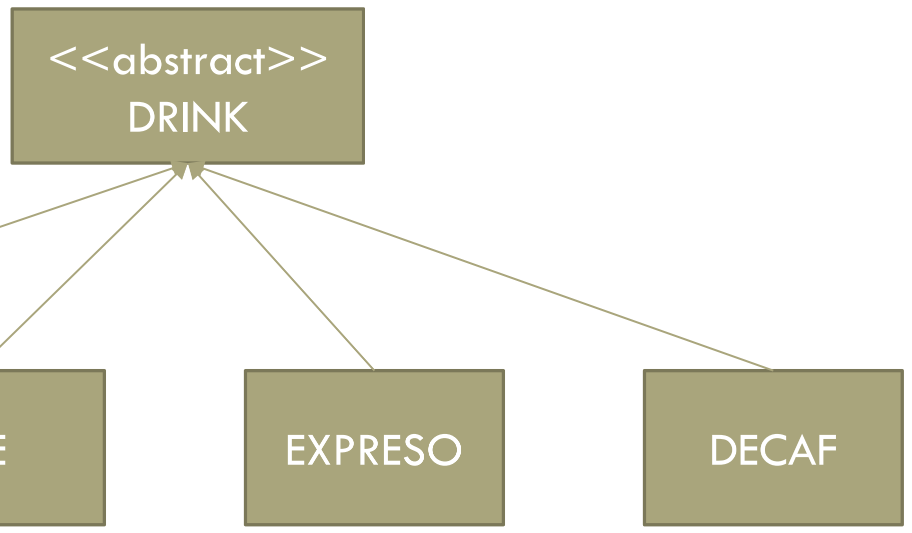
- Want to add properties to an existing object
  - Add borders or scrollbars to a GUI component
  - Add headers and footers to an advertisement
  - Add stream functionality such as reading a line of input or compressing a file before sending it over the wire

DECORATOR. DRINK AUTOMATE

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

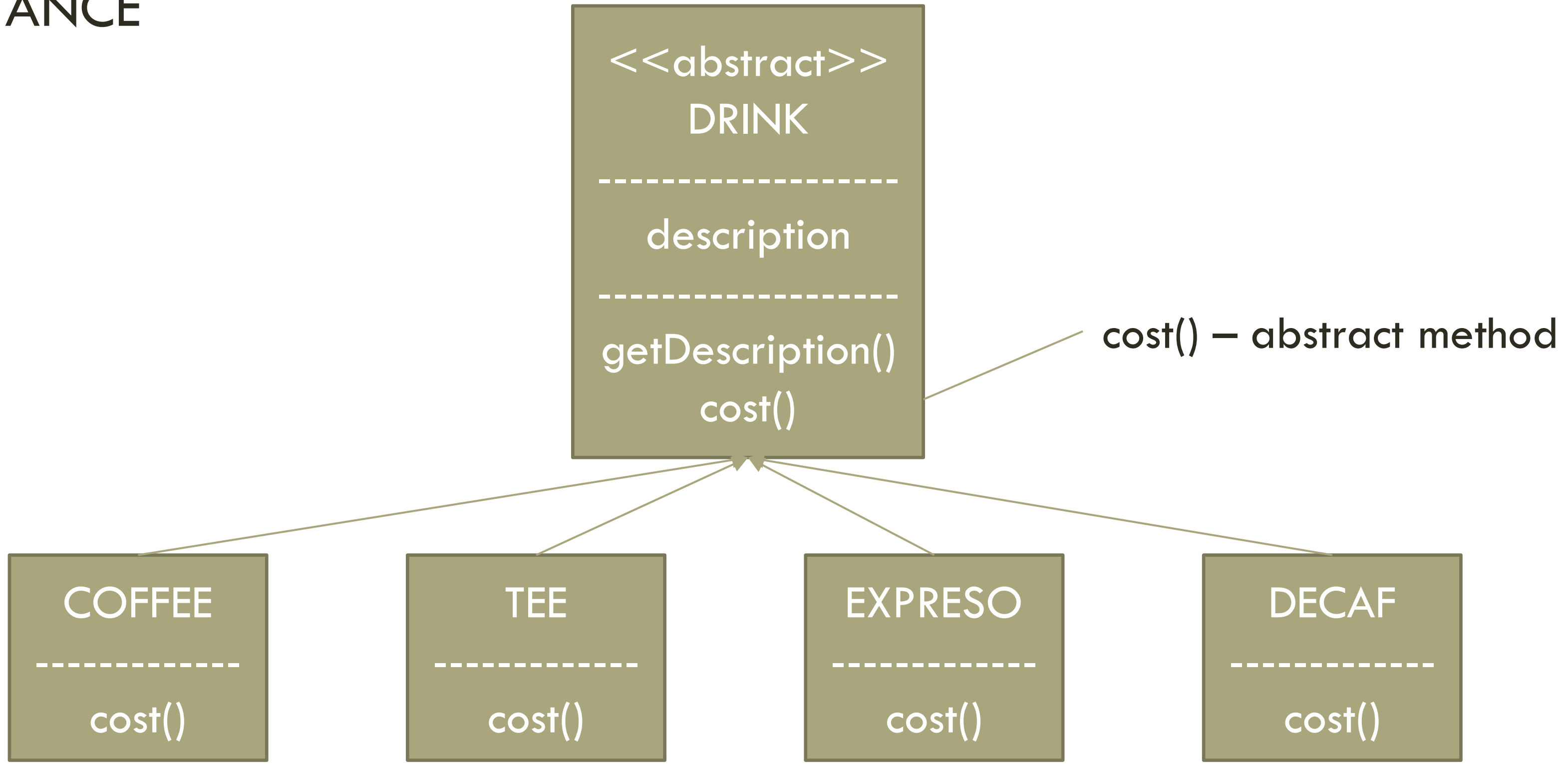
DECORATOR. COFFEE EXAMPLE

- INHERITANCE



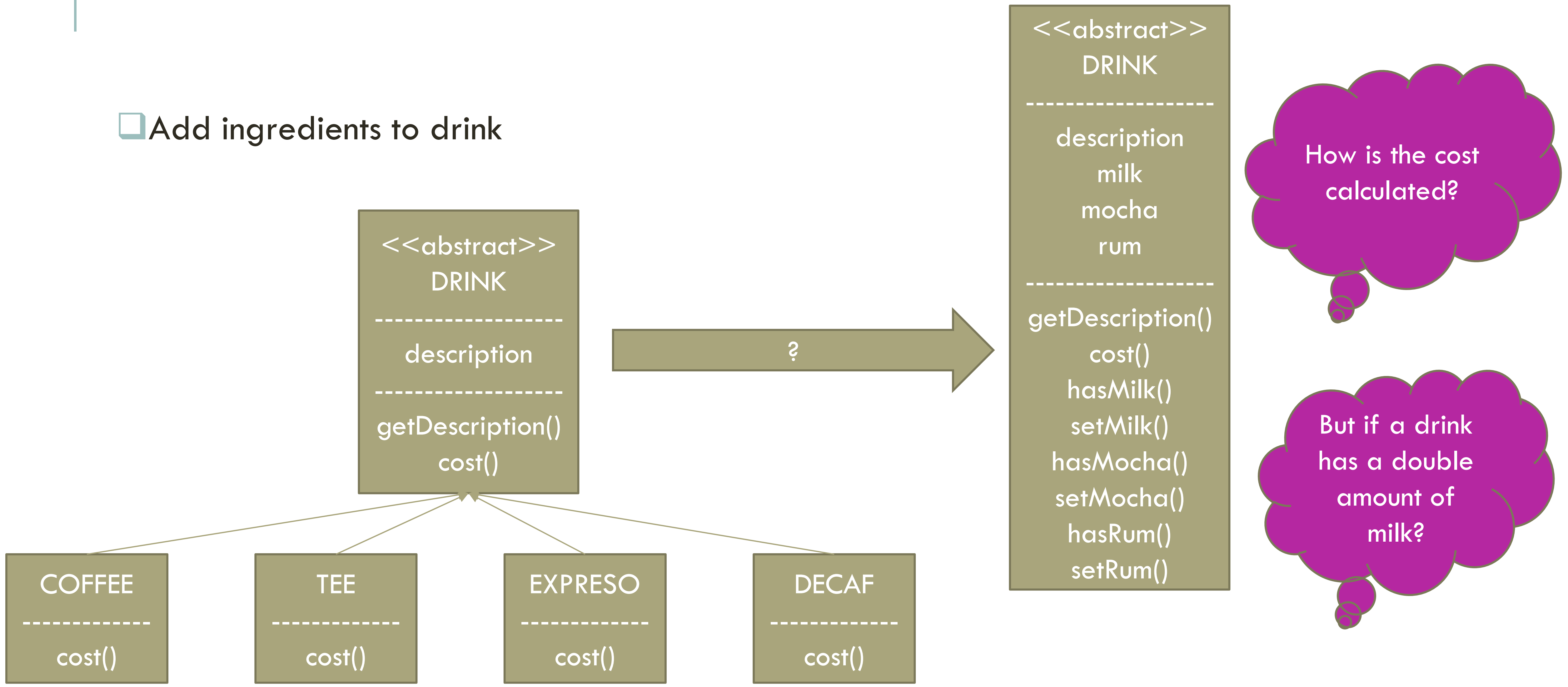
DECORATOR. COFEE EXAMPLE

- INHERITANCE



DECORATOR. COFFEE EXAMPLE

- ❑Add ingredients to drink



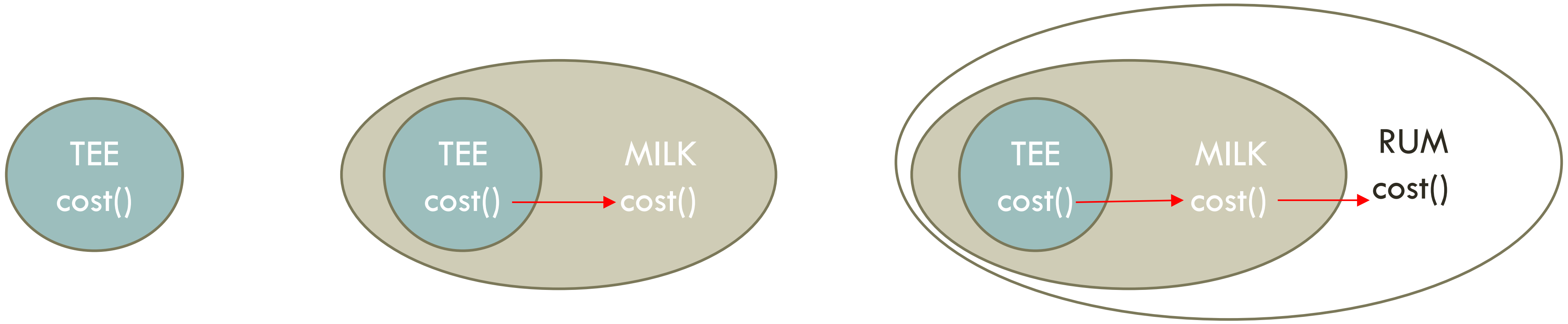
DECORATOR. COFFEE EXAMPLE

Problems

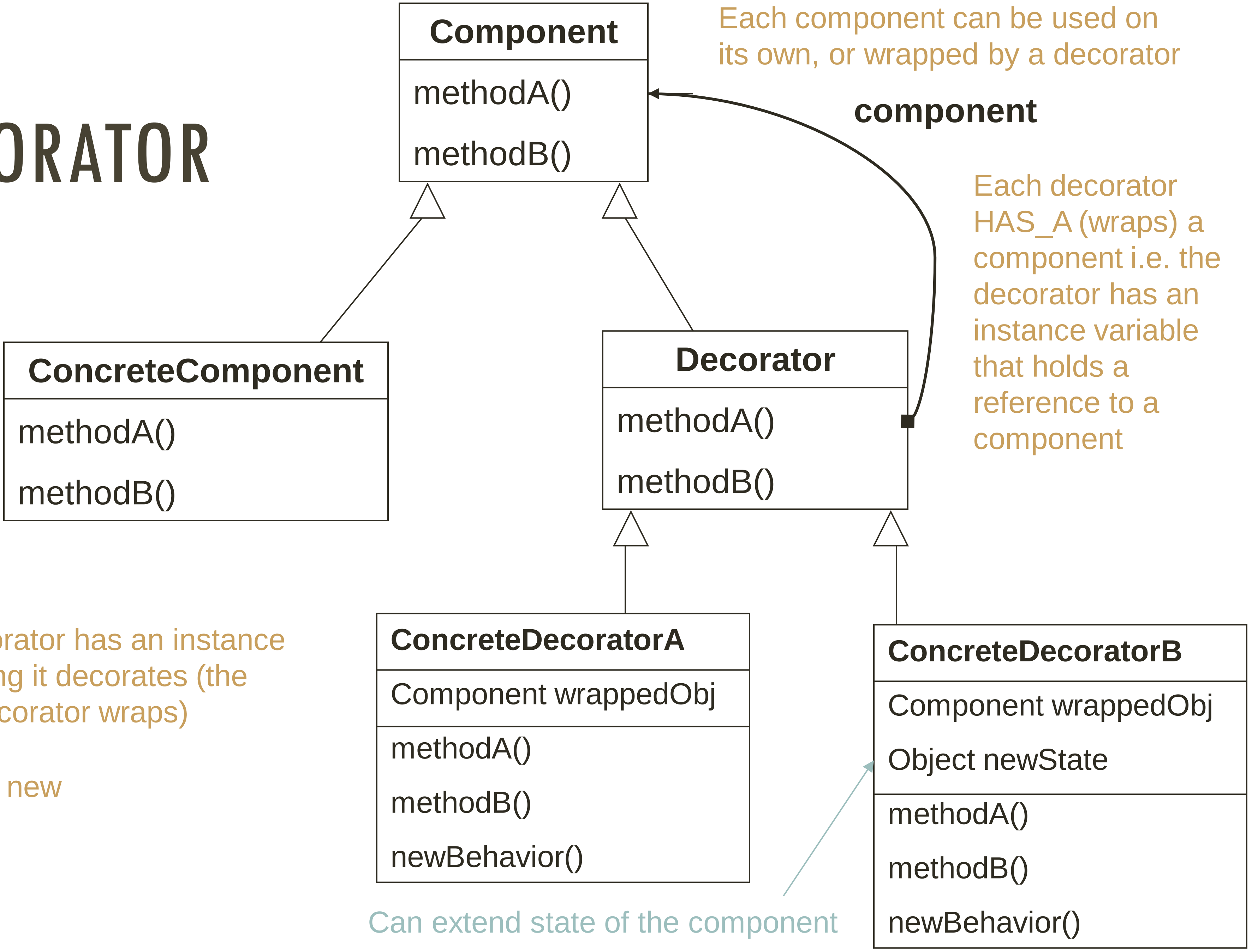
- Goal is to simplify maintenance.
  - Prices can change – code change
  - New condiments – add new methods and alter cost method in superclass
  - New drinks e.g. cappuccino
  - Double mocha?
- Code changes in the superclass when the above happens or in using combinations

DECORATE. CAFE EXAMPLE

- How to resolve this?
  - Using the decoration => start from a base drink and add ingredients



DECORATOR



Each component can be used on its own, or wrapped by a decorator

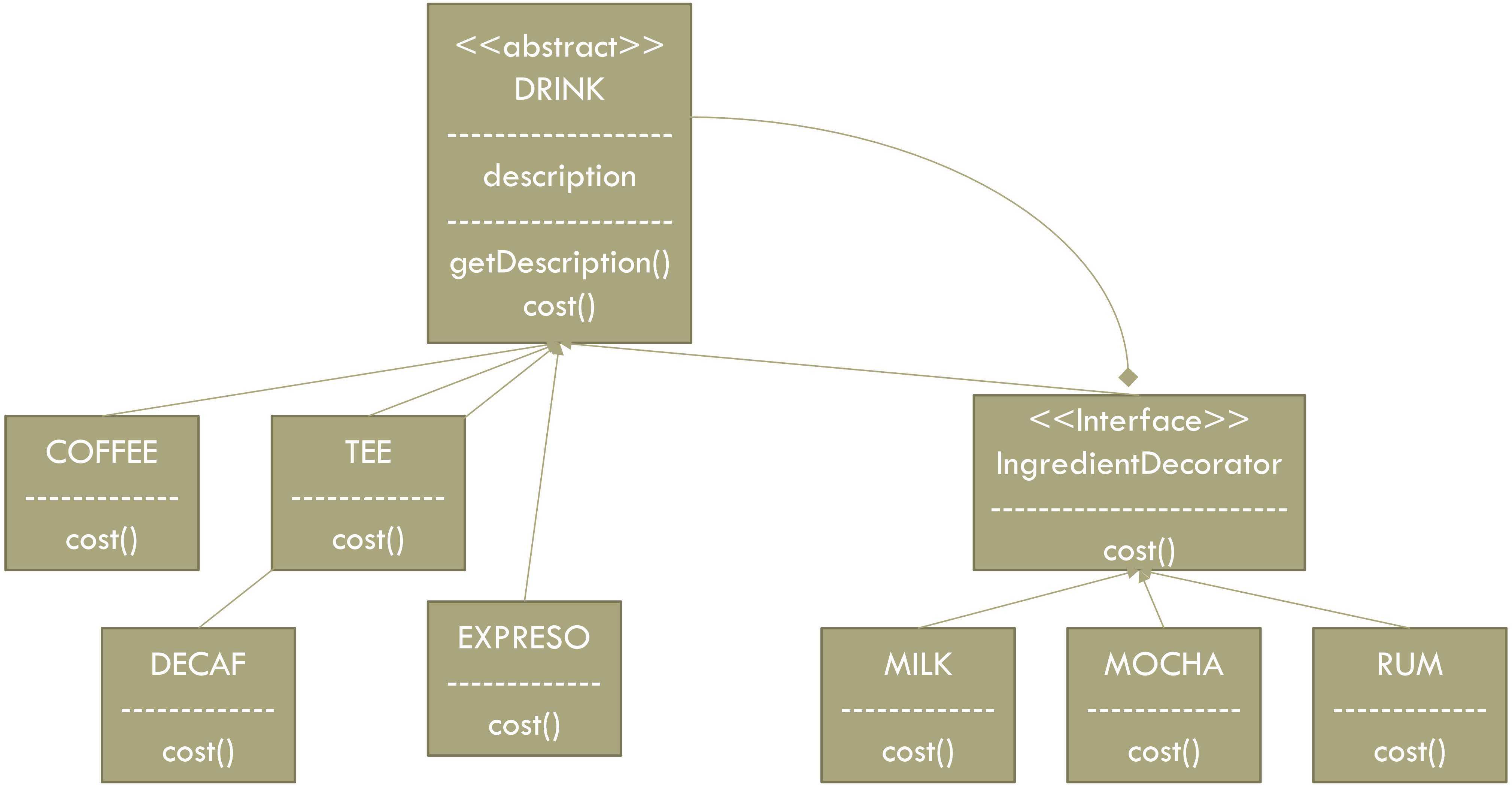
Each decorator HAS_A (wraps) a component i.e. the decorator has an instance variable that holds a reference to a component

The ConcreteDecorator has an instance variable for the thing it decorates (the component the Decorator wraps)

Decorator can add new methods

Can extend state of the component

DECORATOR. COFFEE EXAMPLE



DECORATOR. COFFEE EXAMPLE

```
public abstract class Drink{
        String description = "Unknown Drink";

        public String getDescription() {  // already implemented
                return description;
        }

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

DECORATOR. COFFEE EXAMPLE

We need to be interchangeable with a Drink, so extend the Drink class – not to get its behavior but its type

```
public abstract class IngredientDecorator extends Drink{

        public abstract String getDescription();
}
```

Here we require all the ingredients decorators reimplement the getDescription() method.

DECORATOR. COFFEE EXAMPLE

```
public class Tee extends Drink{

        public Tee() {
                description = "Tee";
        }

        public double cost() {
                return .89;
        }
}
```

Compute the cost of a Tee. Need not worry about ingredients

Note the description variable is inherited from Drink. To take care of description, put this in the constructor for the class

DECORATOR. COFFEE EXAMPLE

Mocha is decorator, so extend IngredientDecorator, which extends Drink

```
public class Mocha extends IngredientDecorator {
    Drink drink;

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

    public String getDescription() {
        return drink.getDescription() + ", Mocha";
    }

    public double cost() {
        return .20 + drink.cost();
    }
}
```

Cost of ingredient+ cost of drink

The description include the drink – say Tee – and the condiments

Instantiate Mocha with a reference to a Drink using

1.An instance variable to hold the drink wrapped

2.A way to set this instance to the object wrapped – pass the drink wrapped to the decorator's constructor

DECORATOR. COFFEE EXAMPLE

```
public class Drink{

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

        Drink drink2 = new Caffee();                              // get a Caffee
        drink2 = new Milk(drink2);                                // wrap it with Milk
        drink2 = new Milk(drink2);                                // warp 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. EXAMPLE

- Java I/O
  - InputStreamReader decorates InputStrim
    - 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.

PROXY

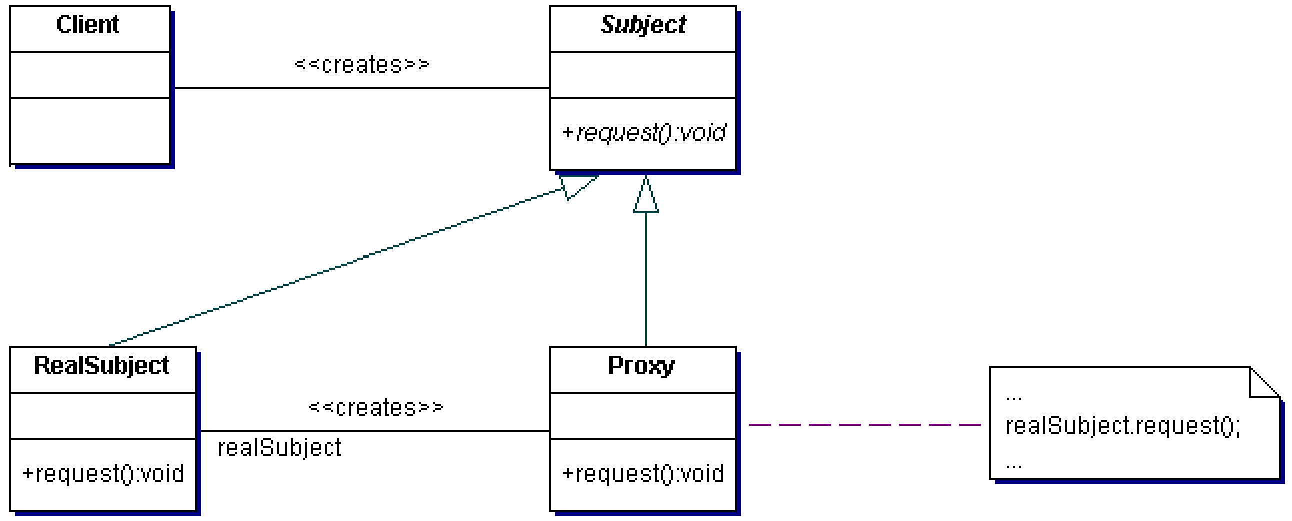
Intent

- Provide a surrogate or placeholder for another object to control access to it.

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

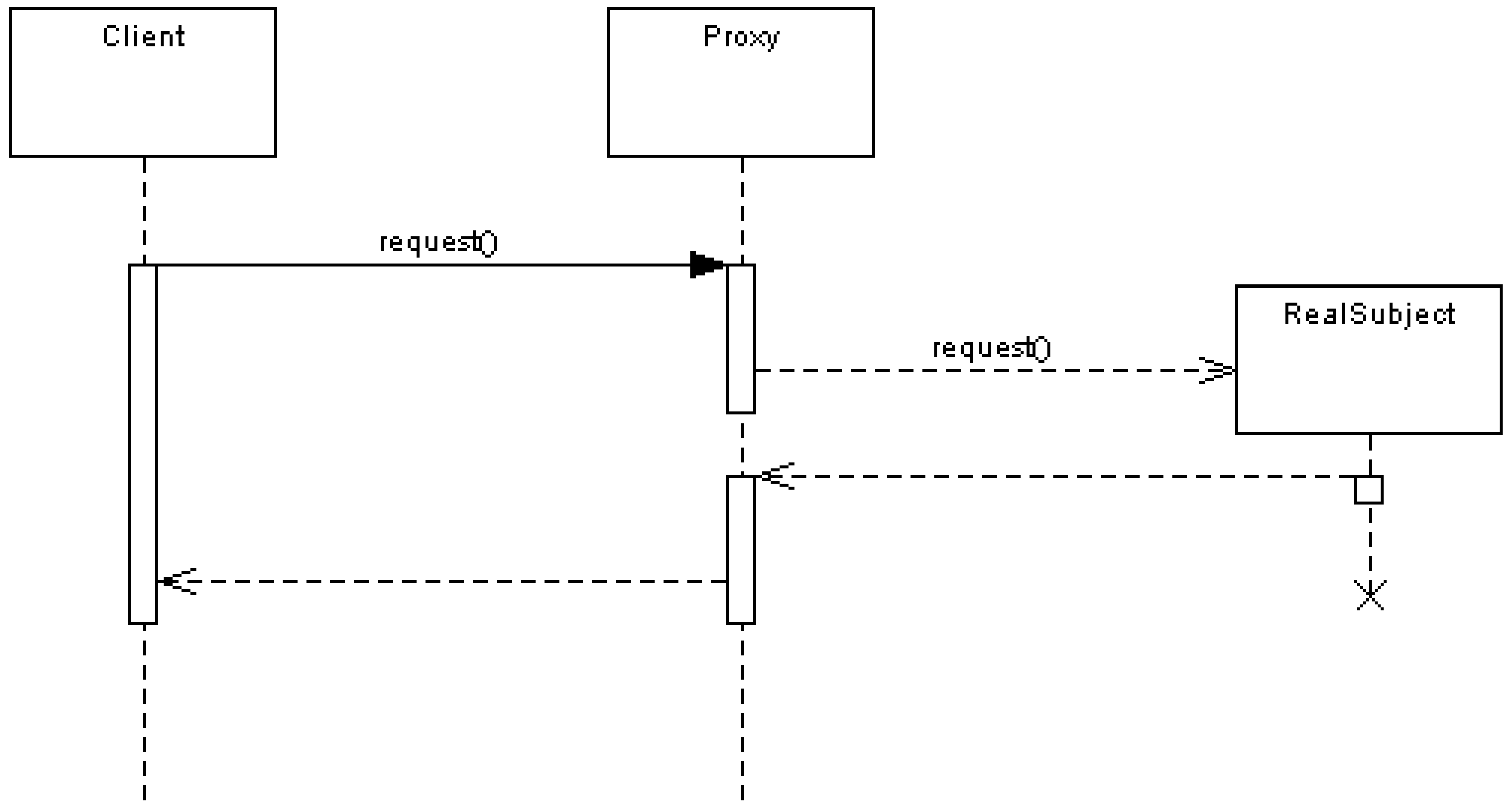


- Subject
  - Interface implemented by the RealSubject and representing its services. The interface must be implemented by the proxy as well so that the proxy can be used in any location where the RealSubject can be used.

- 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.

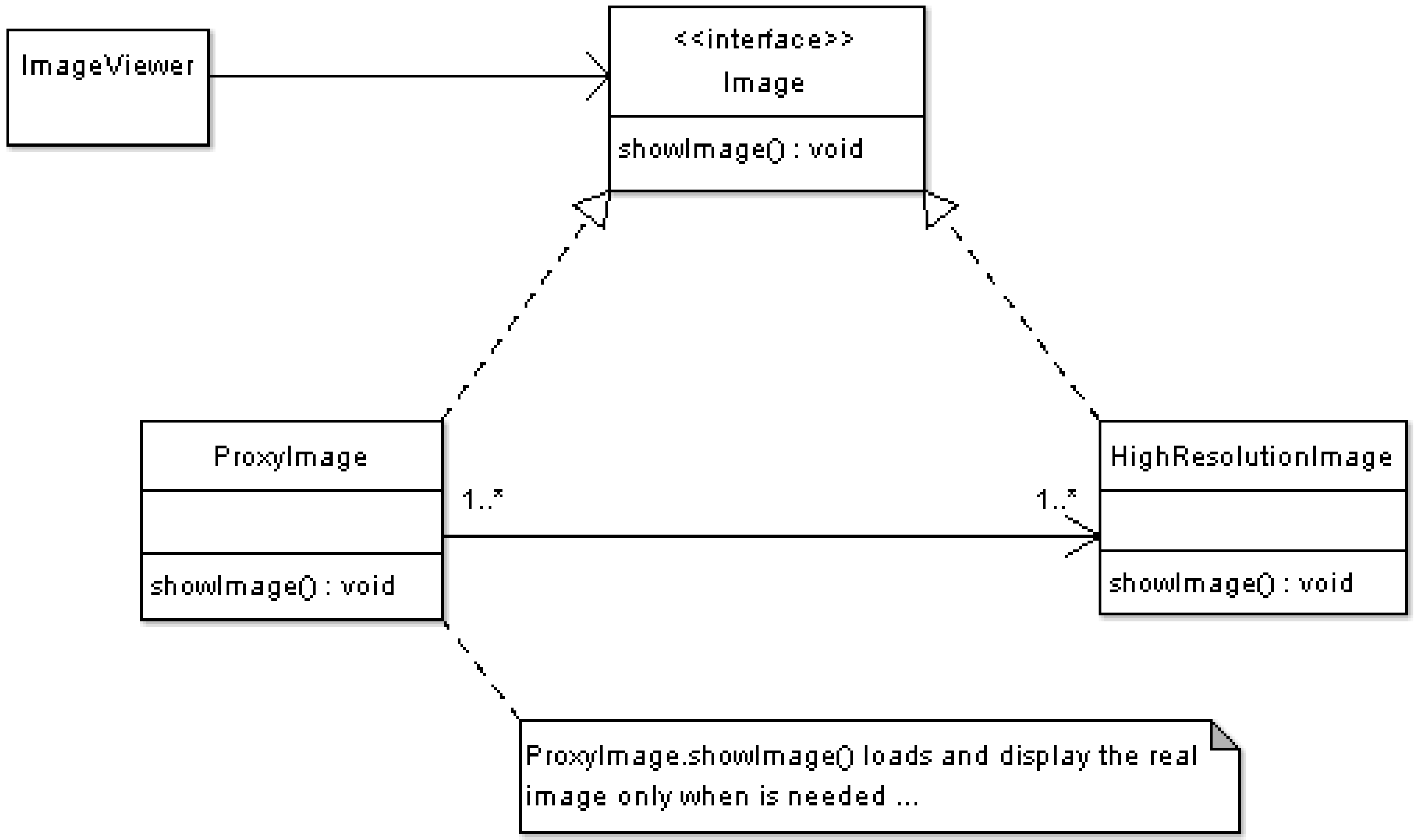
- RealSubject
  - The real object that the proxy represents

PROXY



PROXY. 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. EXAMPLE

```
/*** 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();
}
}
```

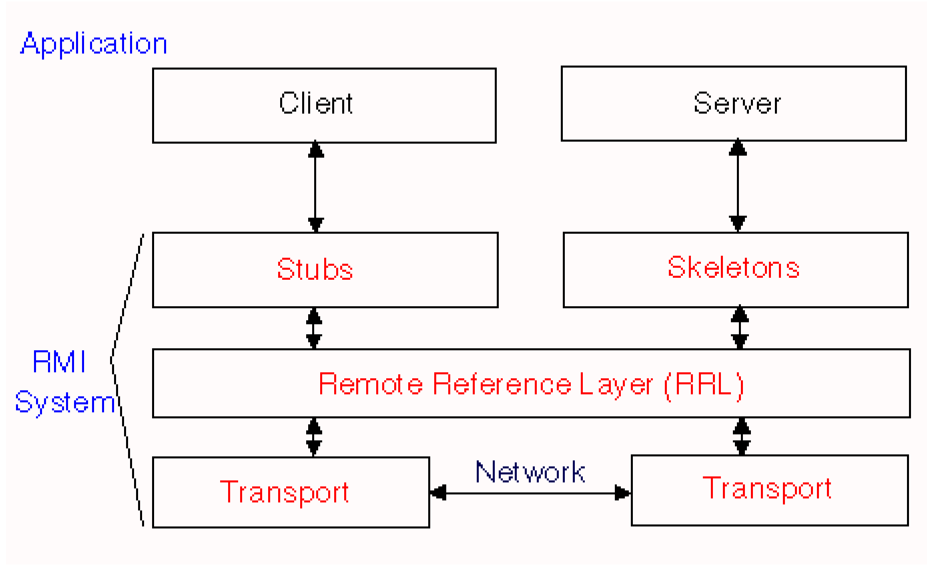
PROXY. EXAMPLE

```
/**
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
        }
}
```

```
/** * 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("sample/veryHighResPhoto1.jpeg");
        Image highResolutionImage2 =
             new ImageProxy("sample/veryHighResPhoto2.jpeg");
        Image highResolutionImage3 =
             new ImageProxy("sample/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("sample/veryHighResPhoto1.jpeg");
        Image highResolutionImageNoProxy2 =
              new HighResolutionImage("sample/veryHighResPhoto2.jpeg");
        Image highResolutionImageBoProxy3 =
              new HighResolutionImage("sample/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
  - java.rmi library
  - “Remote Method Invocation”
  - 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.

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

Memento
Capture and restore an object's internal state

BEHAVIORAL PATTERNS

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

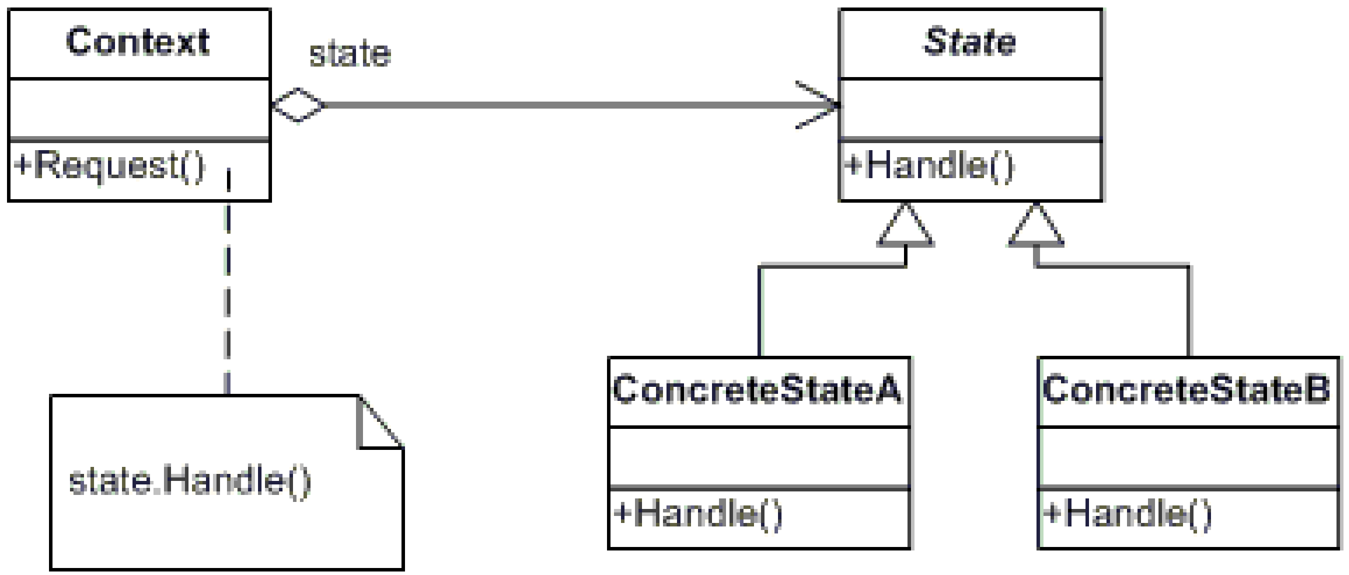
Intent

- 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 run-time 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



- 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 off
    - Low fan
    - Medium fan
    - High fan
    - Turn off

STATE. EXAMPLE

```
class CeilingFanPullChain
{
    private int m_current_state;

    public CeilingFanPullChain() {
        m_current_state = 0;
    }
    public void pull() {
        if (m_current_state == 0) {
            m_current_state = 1;
            System.out.println("   low speed");
        }
        else if (m_current_state == 1)  {
            m_current_state = 2;
            System.out.println("   medium speed");
        } else if (m_current_state == 2) {
            m_current_state = 3;
            System.out.println("   high speed");
        }  else  {
            m_current_state = 0;
            System.out.println("   turning off");
        }
    }
}
```

```
public class StateDemo
{
    public static void main(String[] args)
    {
        CeilingFanPullChain chain = new CeilingFanPullChain();
        while (true)
        {
            System.out.print("Press ");
            chain.pull();
        }
    }
}
```

STATE. EXAMPLE

```
class CeilingFanPullChain
{
    private State m_current_state;

    public CeilingFanPullChain()    {
        m_current_state = new Off();
    }
```

```
public void set_state(State s)  {
        m_current_state = s;
    }
    public void pull()   {
        m_current_state.pull(this);
    }
}
```

STATE. EXAMPLE

```
interface State {
  void pull(CeilingFanPullChain wrapper);
}

class Off implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.set_state(new Low());
        System.out.println("   low speed");
    }
}
```

```
class Low implements State  {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.set_state(new Medium());
        System.out.println("   medium speed");
    }
}

class Medium implements State {
    public void pull(CeilingFanPullChain wrapper)   {
        wrapper.set_state(new High());
        System.out.println("   high speed");
    }
}

class High implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.set_state(new Off());
        System.out.println("   turning off");
    }
}
```

STATE. EXAMPLE

```
public class StateDemo
{
    public static void main(String[] args)
    {
        CeilingFanPullChain chain = new CeilingFanPullChain();
        while (true)
        {
            System.out.print("Press ");
            chain.pull();
        }
    }
```

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.

VISITOR

Intent

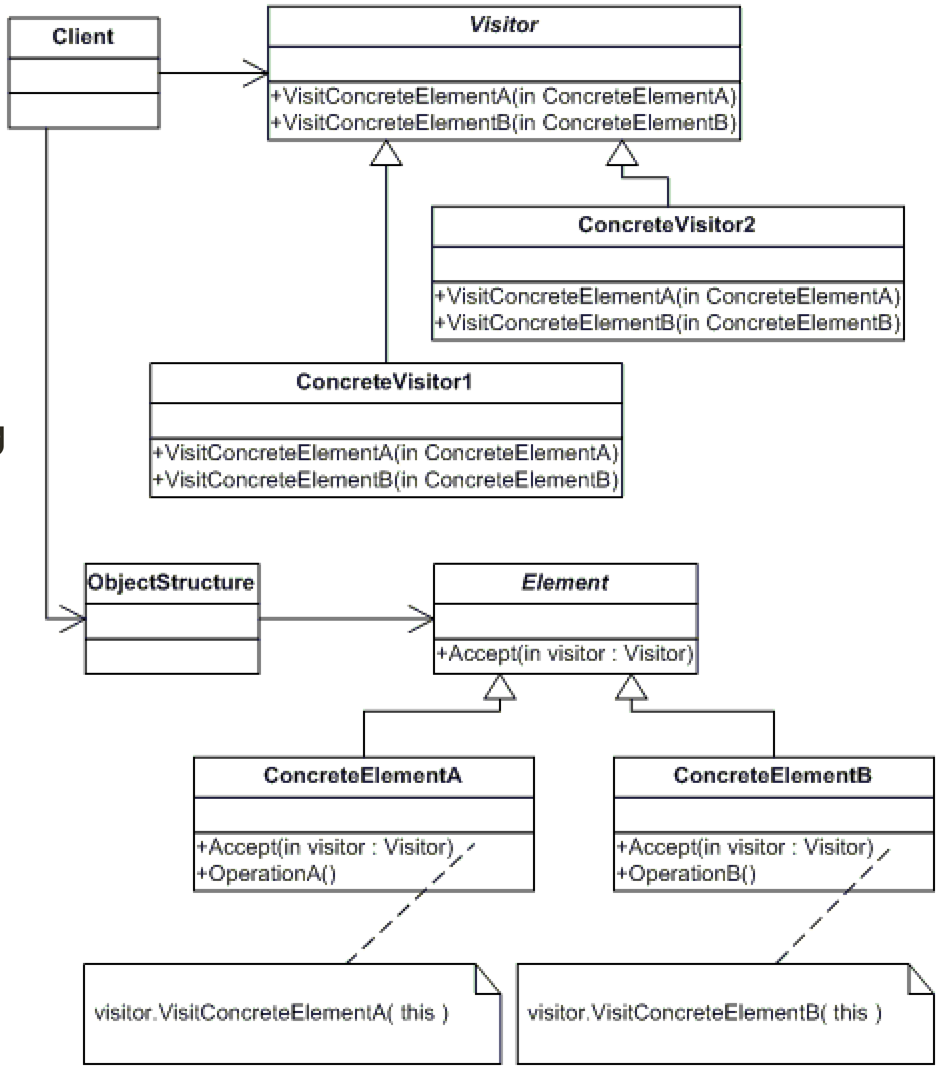
- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- The classic technique for recovering lost type information.
- Do the right thing based on the type of two objects.
- Double dispatch

Problem

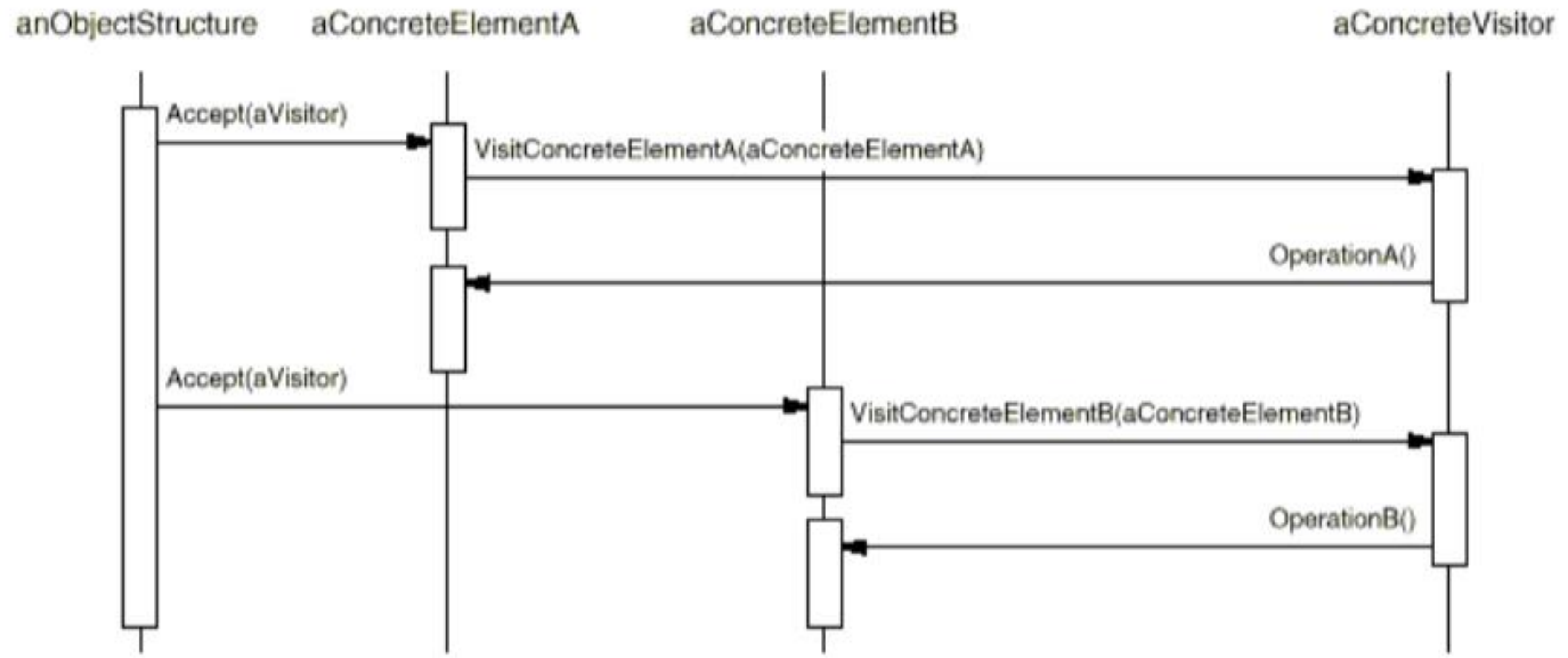
- Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

VISITOR

- **Visitor** - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface
- **ConcreteVisitor** - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- **Element** - defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement** - implements an Accept operation that takes a visitor as an argument
- **ObjectStructure**
  - can enumerate its elements
  - may provide a high-level interface to allow the visitor to visit its elements
  - may either be a Composite (pattern) or a collection such as a list or a set


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)

VISITOR


anObjectStructure
aConcreteElementA
aConcreteElementB
aConcreteVisitor
Accept(aVisitor)
VisitConcreteElementA(aConcreteElementA)
OperationA()
Accept(aVisitor)
VisitConcreteElementB(aConcreteElementB)
OperationB()

VISITOR. EXAMPLE

Example

- Shopping cart where different type of items (Elements) an be added
- When checkout button is clicked, it calculates the total amount to be paid.

VISITOR. EXAMPLE

```
public class Book implements ItemElement {

    private int price;
    private String isbnNumber;

    public Book(int cost, String isbn){
        this.price=cost;
        this.isbnNumber=isbn;
    }
    public int getPrice() {
        return price;
    }
    public String getIsbnNumber() {
        return isbnNumber;
    }
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}
```

```
public interface ItemElement {

    public int accept(ShoppingCartVisitor visitor);

}
```

```
public class Fruit implements ItemElement {
    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm){
        this.pricePerKg=priceKg;
        this.weight=wt;
        this.name = nm;
    }
    public int getPricePerKg() {
        return pricePerKg;
    }
    public int getWeight() {
        return weight;
    }
    public String getName(){
        return this.name;
    }
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }

}
```

VISITOR. EXAMPLE

```
public interface ShoppingCartVisitor {


    int visit(Book book);

    int visit(Fruit fruit);

}
```

```
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {
    @Override
    public int visit(Book book) {
        int cost=0;
        //apply 5$ discount if book price is greater than 50
        if(book.getPrice() > 50){
            cost = book.getPrice()-5;
        }else cost = book.getPrice();
        System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost ="+cost);
        return cost;
    }
    @Override
    public int visit(Fruit fruit) {
        int cost = fruit.getPricePerKg()*fruit.getWeight();
        System.out.println(fruit.getName() + " cost = "+cost);
        return cost;
    }
}
```

VISITOR. EXAMPLE

```
public class ShoppingCartClient {

    public static void main(String[] args) {
        ItemElement[] items = new ItemElement[]{
                new Book(20, "1234"),
                 new Book(100, "5678"),
                 new Fruit(10, 2, "Banana"),
                new Fruit(5, 5, "Apple")};

        int total = calculatePrice(items);
        System.out.println("Total Cost = "+total);
    }

    private static int calculatePrice(ItemElement[] items) {
        ShoppingCartVisitor visitor =
                new ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item : items){
            sum = sum + item.accept(visitor);
        }
        return sum;
    }

}
```

VISITOR

Benefits

- Adding new operations is easy
- Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor. Unrelated sets of behavior are partitioned in their own visitor subclasses.
- Visitors can accumulate state as they visit each element in the object structure.
- Without a visitor, this state would have to be passed as extra arguments to the operations that perform the traversal.

Disadvantages

- Adding new ConcreteElement classes is hard. Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class.
- The ConcreteElement interface must be powerful enough to let visitors do their job. You may be forced to provide public operations that access an element's internal state, which may compromise its encapsulation

NULL OBJECT

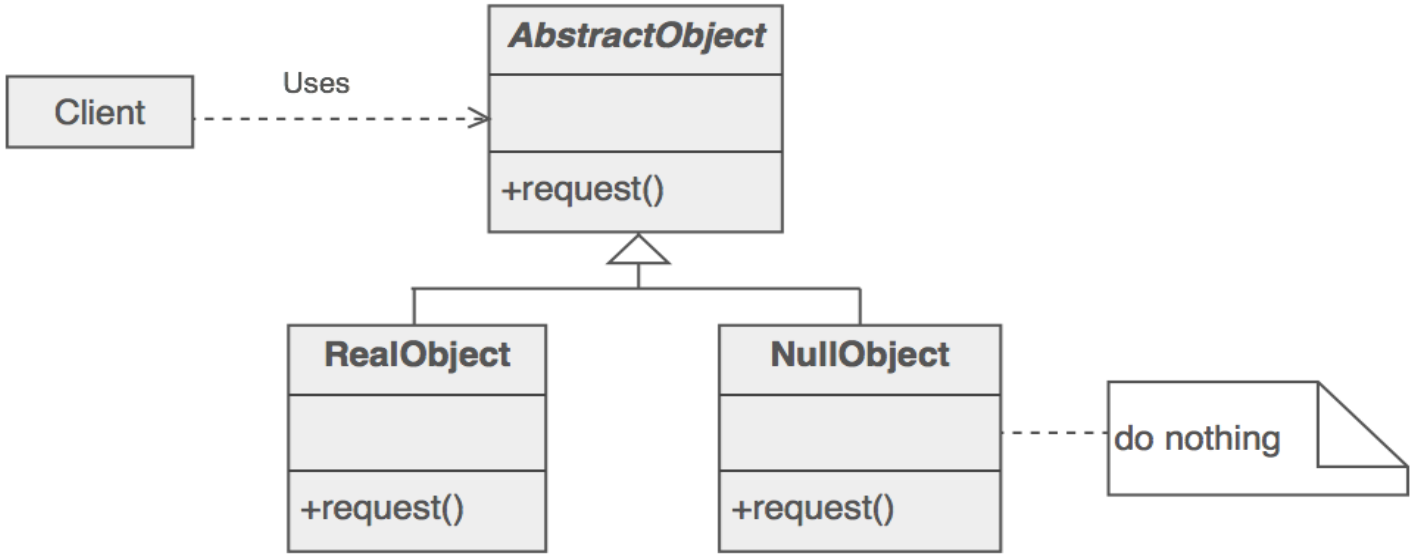
Intent

- 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



- **AbstractClass** - defines abstract primitive operations that concrete implementations have to define.
- **RealClass** - a real implementation of the AbstractClass performing some real actions.
- **NullClass** - 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 – Loging system

- Implement a logging framework in order to support the logging of an application. The framework must fulfill the following requirements:
  - The destination of the output messages should be selected from a configuration file and it can be one of the following options: Log File, Standard Console or Log Disabled.
- Must be open for extension; new logging mechanism can be added without touching the existing code.

NULL OBJECT

```
public abstract class AbstractCustomer {
   protected String name;
   public abstract boolean isNil();
   public abstract String getName();
}
```

```
public class RealCustomer extends AbstractCustomer {
   public RealCustomer(String name) {
      this.name = name;
   }
   @Override
   public String getName() {
      return name;
   }
@Override
   public boolean isNil() {
      return false;
   }
}
```

```
public class NullCustomer extends AbstractCustomer {
   @Override
   public String getName() {
      return "Not Available in Customer Database";
   }
   @Override
   public boolean isNil() {
      return true;
   }
}
```

NULL OBJECT

```
public class CustomerFactory {
   public static final String[] names = {"Rob", "Joe", "Julie"};
   public static AbstractCustomer getCustomer(String name){
      for (int i = 0; i < names.length; i++) {
         if (names[i].equalsIgnoreCase(name)){
            return new RealCustomer(name);
         }
      }
      return new NullCustomer();
   }
}
```

NULL OBJECT

```
public class NullPatternDemo {
   public static void main(String[] args) {
      AbstractCustomer customer1 = CustomerFactory.getCustomer("Rob");
      AbstractCustomer customer2 = CustomerFactory.getCustomer("Bob");
      AbstractCustomer customer3 = CustomerFactory.getCustomer("Julie");
      AbstractCustomer customer4 = CustomerFactory.getCustomer("Laura");

      System.out.println("Customers");
      System.out.println(customer1.getName());
      System.out.println(customer2.getName());
      System.out.println(customer3.getName());
      System.out.println(customer4.getName());
   }
}
```

NULL OBJECT

- 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.