innovate achieve lead

BITS Pilani
Pilani Campus

DESIGN PATTERNS (Lecture-11)

Motivation and Concept

- OO systems exploit recurring design structures that promote
  - Abstraction
  - Flexibility
  - Modularity
  - Elegance

What Is a Design Pattern?

- A design pattern
  - **Is a common solution to a recurring problem in design**
  - Abstracts a recurring design structure
  - Comprises class and/or object
    - Dependencies
    - Structures
    - Interactions
    - Conventions
- A design pattern has 4 basic parts:
  - 1. Name
  - 2. Problem
  - 3. Solution
  - 4. Consequences and trade-offs of application
- Language- and implementation-independent

Three Types of Patterns

- **Creational patterns:**
  - Deal with initializing and configuring classes and objects
- **Structural patterns**:
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
- **Behavioral patterns**:
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

Classification of GoF Design Pattern

Creational	Structural	Behavioral
Factory Method	Adapter	Interpreter
Abstract Factory	Bridge	Template Method
Builder	Composite	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Flyweight	Iterator
	Facade	Mediator
	Proxy	Memento
		Observer
		State
		Strategy
		Visitor

Creational Patterns

- **Abstract Factory**:
  - Factory for building related objects
- **Builder**:
  - Factory for building complex objects incrementally
- **Factory Method**:
  - Method in a derived class creates associates
- **Prototype**:
  - Factory for cloning new instances from a prototype
- **Singleton**:
  - Factory for a singular (sole) instance

Structural patterns

- Describe ways to assemble objects to realize new functionality
  - Added flexibility inherent in object composition due to ability to change composition at run-time
  - not possible with static class composition
- Example: Proxy
  - ***Proxy***: acts as convenient surrogate or placeholder for another object.
    - Remote Proxy: local representative for object in a different address space
    - Virtual Proxy: represent large object that should be loaded on demand
    - Protected Proxy: protect access to the original object

Structural Patterns

- **Adapter:**
  - Translator adapts a server interface for a client
- **Bridge**:
  - Abstraction for binding one of many implementations
- **Composite**:
  - Structure for building recursive aggregations
- **Decorator**:
  - Decorator extends an object transparently
- **Facade**:
  - Simplifies the interface for a subsystem
- **Flyweight**:
  - Many fine-grained objects shared efficiently.
- **Proxy**:
  - One object approximates another

Behavioral Patterns

- **Chain of Responsibility**:
  - Request delegated to the responsible service provider
- **Command**:
  - Request or Action is first-class object, hence re-storable
- **Iterator**:
  - Aggregate and access elements sequentially
- **Interpreter**:
  - Language interpreter for a small grammar
- **Mediator**:
  - Coordinates interactions between its associates
- **Memento**:
  - Snapshot captures and restores object states privately

Which ones do you think you have seen somewhere?

Creational Patterns

- Today's class
  - Singleton
  - Factory

Singleton Pattern

- Singleton pattern is used when we want to **allow only a single instance of a class to be created inside our application. Using this pattern ensures that a class only have a single instance by protecting the class creation** process, by setting the class constructor into private access modifier. To get the class instance, the singleton class can provide a method for example a *getInstance()* method, this will be the only method that can be accessed to get the instance.

```
public class Service {
  private static Service instance = null;
  private Service() { }
  public static synchronized Service
                              getInstance() {
      if(null == instance){
            instance = new Service();
            return instance;
      }
  }
  protected Object clone() throws
  CloneNotSupportedException {
      throw
            new CloneNotSupportedException("Clone is
                                    not allowed.");
  }
}
```

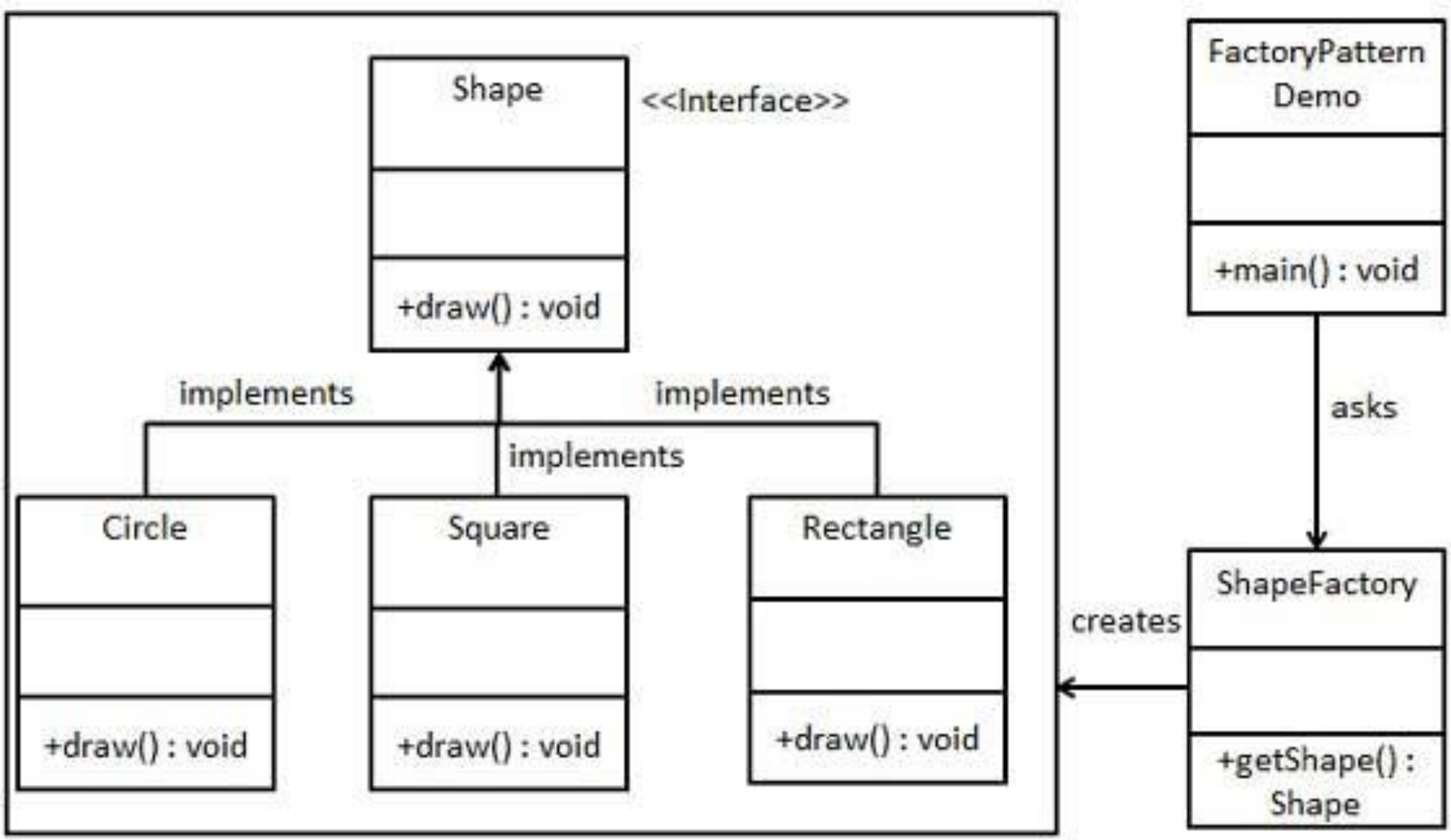
There are some rules that need to be followed when we want to implement a singleton:

- A singleton has a static variable to keep it sole instance.
- You need to set the class constructor into private access modifier. By this you will not allow any other class to create an instance of this singleton because they have no access to the constructor.
- Because no other class can instantiate this singleton, the singleton class should provide a service to its users by providing a method that returns its singleton instance, for example getInstance().
- When we use our singleton in a multi threaded application we need to make sure that instance creation process doesn’t result in more than one instance, so we add a synchronized keywords to protect more than one thread access this method at the same time.
- It is also advisable to override the clone() method of the java.lang.Object class and throw CloneNotSupportedException so that another instance cannot be created by cloning the singleton object.

Factory Pattern

- Factory pattern is one of most used design pattern. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Factory Pattern (contd.)



Factory Pattern (contd.)

- We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.
- *FactoryPatternDemo*, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE* / *RECTANGLE* / *SQUARE*) to *ShapeFactory* to get the type of object it needs.

Step-1 Create an interface.

```
public interface Shape { void draw(); }
```

Step-2 Create concrete classes implementing the same interface.

```
public class Rectangle implements Shape {
    public void draw() {
        System.out.println("Inside Rectangle::draw()
    method.");
    }
}
```

Similar code for Square and Circle

Step-3 Create a Factory to generate object of concrete class based on given information.

```
public class ShapeFactory {
    //use getShape method to get object of type shape
    public Shape getShape(String shapeType) {
            if(shapeType == null) { return null; }
            if(shapeType.equalsIgnoreCase("CIRCLE")) {
                        return new Circle();
            }
            else if(shapeType.equalsIgnoreCase("RECTANGLE")) {
                        return new Rectangle();
            } else if(shapeType.equalsIgnoreCase("SQUARE")) {
                        return new Square();
            }
            return null;
    }
}
```

Step-4 Use the Factory to get object of concrete class by passing an information such as type.

```
public class FactoryPatternDemo {
    public static void main(String[] args) {
            ShapeFactory shapeFactory = new ShapeFactory();
            //get an object of Circle and call its draw method.
            Shape shape1 = shapeFactory.getShape("CIRCLE");
            //call draw method of Circle shape1.draw();
            //get an object of Rectangle and call its draw method.
            Shape shape2 = shapeFactory.getShape("RECTANGLE");          //call
    draw method of Rectangle shape2.draw();
            //get an object of Square and call its draw method.
            Shape shape3 = shapeFactory.getShape("SQUARE");
            //call draw method of square shape3.draw();
    }
}
```

Structural Patterns

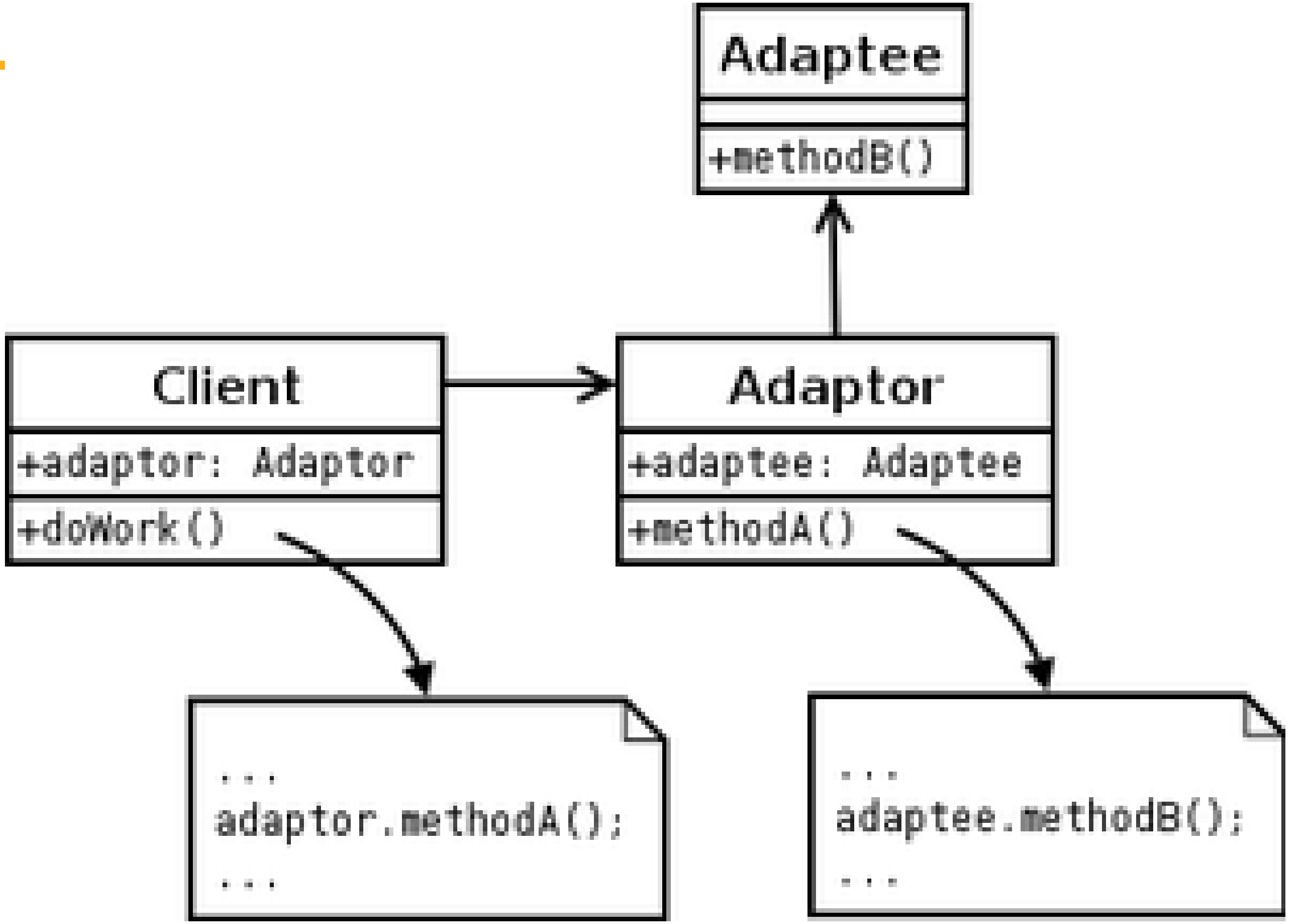
- Adapter
- Composite
- Decorator

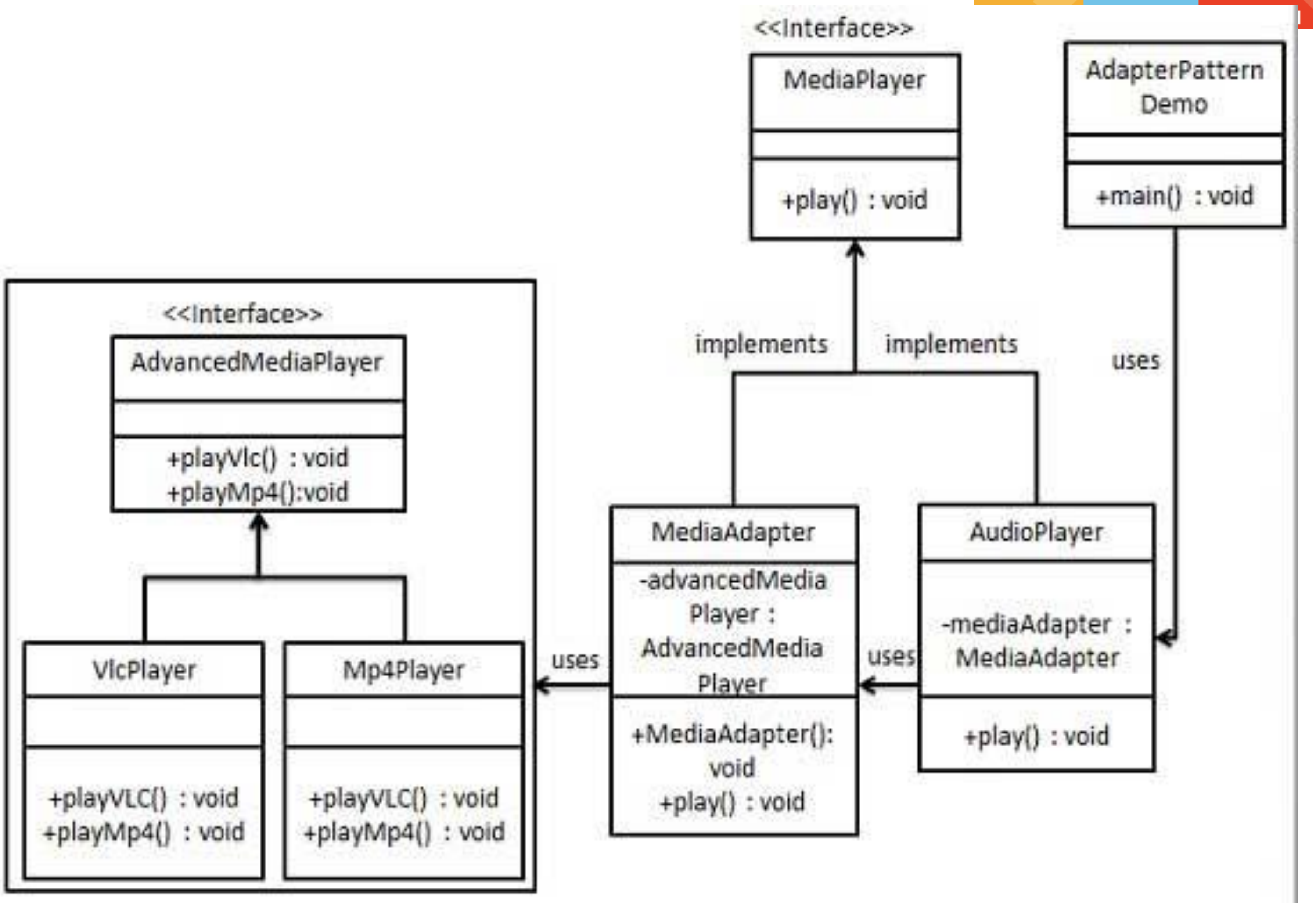
Adapter Pattern

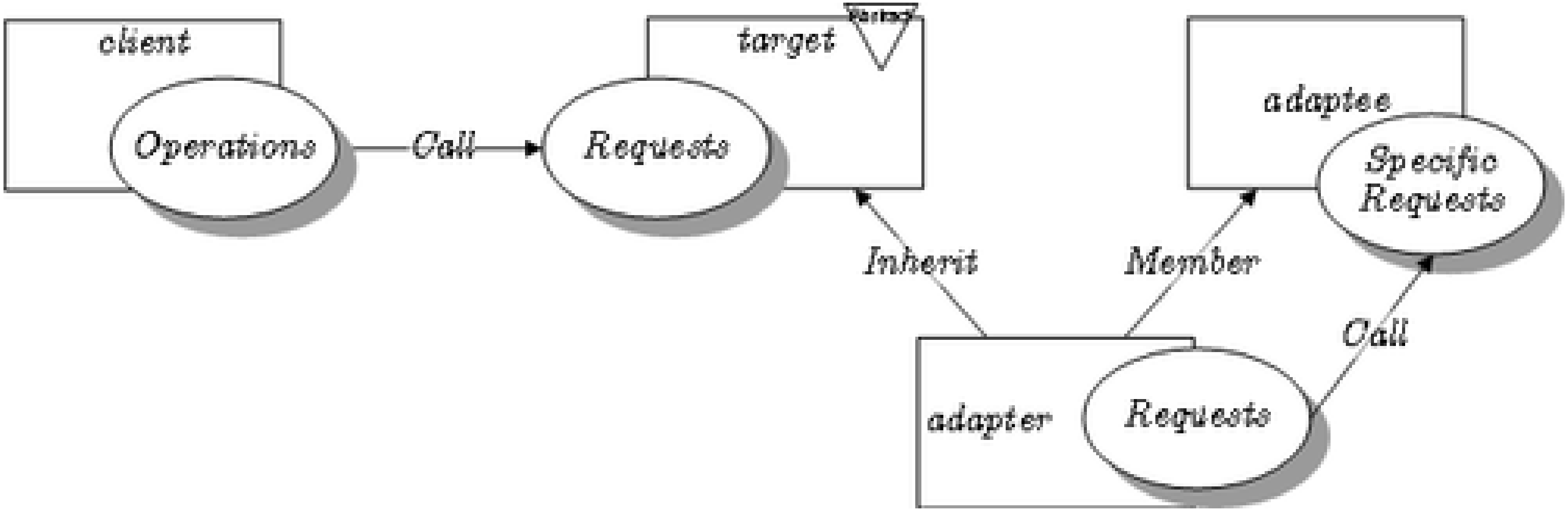
- Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.
- This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugins the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

Adapter Pattern (Contd.)

- We've an interface *MediaPlayer* interface and a concrete class *AudioPlayer* implementing the *MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default.
- We're having another interface *AdvancedMediaPlayer* and concrete classes implementing the *AdvancedMediaPlayer* interface.These classes can play vlc and mp4 format files.
- We want to make *AudioPlayer* to play other formats as well. To attain this, we've created an adapter class *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format.
- *AudioPlayer* uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format. *AdapterPatternDemo*, our demo class will use *AudioPlayer* class to play various formats.


Adaptee
+methodB()
Client
+adaptor: Adaptor
+doWork()
Adaptor
+adaptee: Adaptee
+methodA()
...
adaptor.methodA();
...
...
adaptee.methodB();
...


<<Interface>>
MediaPlayer
+play() : void
AdapterPattern Demo
+main() : void
implements
implements
uses
<<Interface>>
AdvancedMediaPlayer
+playVlc() : void
+playMp4():void
MediaAdapter
-advancedMedia Player : AdvancedMedia Player
+MediaAdapter(): void
+play() : void
AudioPlayer
-mediaAdapter : MediaAdapter
+play() : void
uses
uses
VlcPlayer
+playVLC() : void
+playMp4() : void
Mp4Player
+playVLC() : void
+playMp4() : void


client
Operations
Call
Requests
target
Abstract
adaptee
Specific Requests
Inherit
Member
Call
adapter
Requests

Adapter Implementation

```
public class MediaAdapter implements MediaPlayer {

    AdvancedMediaPlayer advancedMusicPlayer;

    public MediaAdapter(String audioType){
          if(audioType.equalsIgnoreCase("vlc") ) {
                    advancedMusicPlayer = new VlcPlayer();
          } else if (audioType.equalsIgnoreCase("mp4")) {
          advancedMusicPlayer = new Mp4Player();
          }
    }

    public void play(String audioType, String fileName) {
          if(audioType.equalsIgnoreCase("vlc")) {
          advancedMusicPlayer.playVlc(fileName);
          }else if(audioType.equalsIgnoreCase("mp4")) {
                    advancedMusicPlayer.playMp4(fileName);
          }
    }
}
```

Client Implementation

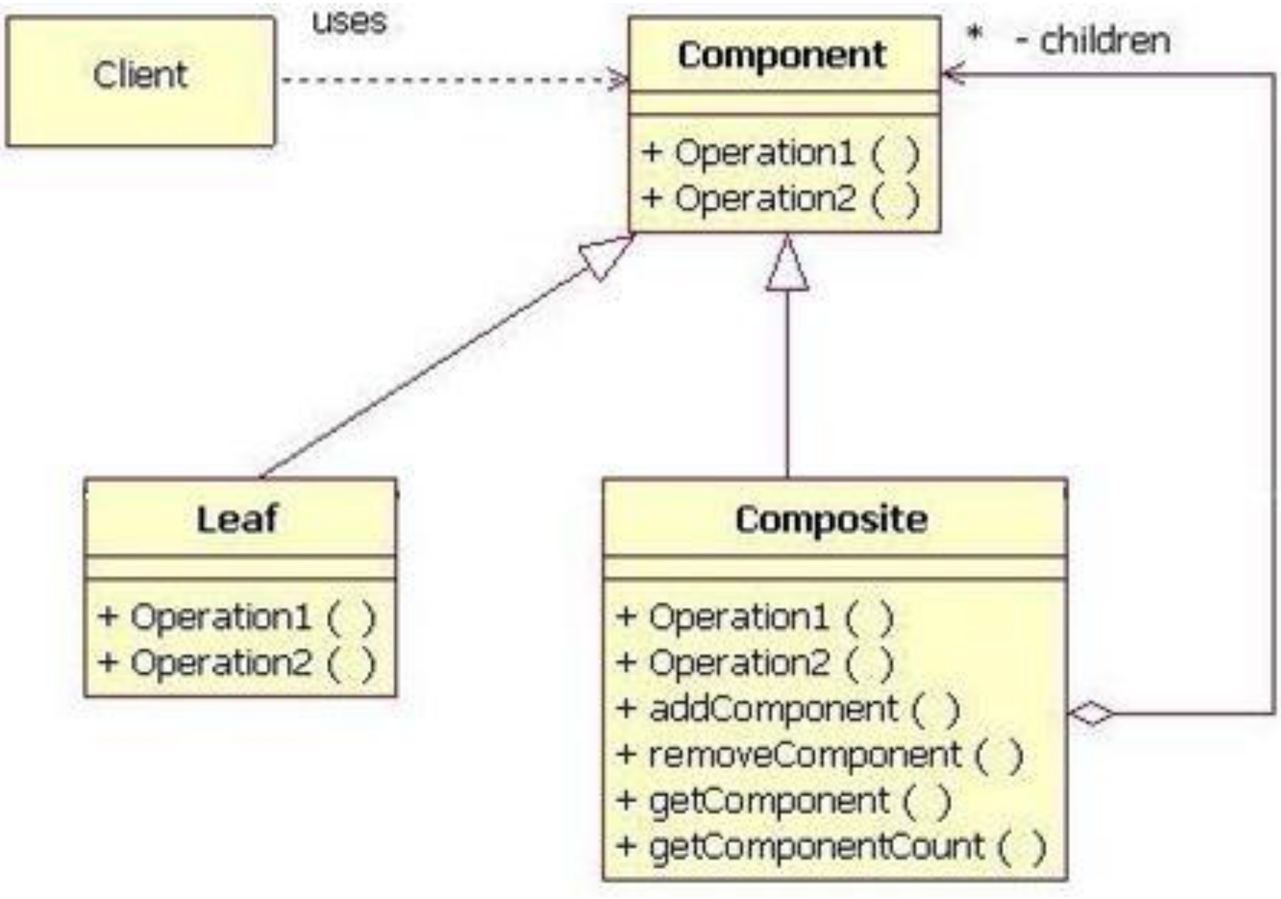
```
public class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;
    public void play(String audioType, String fileName) {
    //inbuilt support to play mp3 music files
    if(audioType.equalsIgnoreCase("mp3")) {    System.out.println("Playing mp3
    file. Name: "+ fileName); }              //mediaAdapter is providing support to play
    other file formats else    if(audioType.equalsIgnoreCase("vlc") ||

           audioType.equalsIgnoreCase("mp4")) {
           mediaAdapter = new MediaAdapter(audioType);
           mediaAdapter.play(audioType, fileName);
           } else {
                      System.out.println("Invalid media. "+ audioType + " format
                                                          not supported");
           }
    }
}
```

Composite Pattern

- **Composite objects: objects that contain other objects; for example, a drawing may be composed of** graphic primitives, such as lines, circles, rectangles, text, and so on. Developers need the Composite pattern because we often must manipulate composites exactly the same way we manipulate primitive objects. For example, graphic primitives such as lines or text must be drawn, moved, and resized. But we also want to perform the same operation on composites, such as drawings, that are composed of those primitives. Ideally, we'd like to perform operations on both primitive objects and composites in exactly the same manner, *without distinguishing between the two. If* we must distinguish between primitive objects and composites to perform the same operations on those two types of objects, our code would become more complex and more difficult to implement, maintain, and extend. Implementing the Composite pattern is easy. Composite classes extend a base class that represents primitive objects.

Figure-1


Client
uses
Component
+ Operation1 ( )
+ Operation2 ( )
* - children
Leaf
+ Operation1 ( )
+ Operation2 ( )
Composite
+ Operation1 ( )
+ Operation2 ( )
+ addComponent ( )
+ removeComponent ( )
+ getComponent ( )
+ getComponentCount ( )

Composite Pattern (contd.)

- **Component represents a base class (or possibly an interface) for primitive objects, and Composite** represents a composite class. For example, the **Component class might represent a base class for graphic** primitives, whereas the **Composite class might represent a Drawing class. Leaf class represents a** concrete primitive object; for example, a **Line class or a Text class. The Operation1() and Operation2() methods represent domain-specific methods implemented by both the Component** and **Composite classes.**
- The **Composite class maintains a collection of components. Typically, Composite** methods are implemented by iterating over that collection and invoking the appropriate method for each **Component in the collection. For example, a Drawing class might implement its draw() method** like this:

Composite Pattern (contd.)

```
// This method is a Composite method
public void draw() {
        // Iterate over the components
        for(int i=0; i < getComponentCount(); ++i) {
                // Obtain a reference to the component and invoke its draw method
                Component component = getComponent(i);
                component.draw();
        }
}
```

- For every method implemented in the **Component class, the Composite class implements a method** with the same signature that iterates over the composite's components, as illustrated by the **draw()** method listed above.

Composite Pattern (contd.)

- The **Composite class extends the Component class, so you can pass a composite to a method that** expects a component; for example, consider the following method:

```
// this method is implemented in a class that's unrelated to the
// Component and Composite classes
public void repaint(Component component) {
    // the component can be a composite, but since it extends
    // the Component class, this method need not
    // distinguish between components and composites
    component.draw();
}
```

- The preceding method is passed a component either a simple component or a composite then it invokes that component's **draw() method. Because the Composite class extends Component, the repaint()method need not distinguish between components and composites it simply invokes the draw() method for the component (or composite).**

Composite Pattern (contd.)

- Figure-1's Composite pattern class diagram illustrate one problem with the pattern: *you must distinguish between* ***components and composites when you reference a Component, and you must invoke a composite specific method, such as addComponent(). You typically fulfill that requirement by*** adding a method, such as ***isComposite(), to the Component class. That method returns false for*** components and is overridden in the **Composite class to return true. Additionally, you must also cast** the **Component reference to a Composite instance, like this:**

```
...
if(component.isComposite()) {
Composite composite = (Composite)component;
composite.addComponent(someComponentThatCouldBeAComposite);
}
...
```

- Notice that the **addComponent() method is passed a Component reference, which can be either a** primitive component or a composite.

Alternative Composite pattern implementation

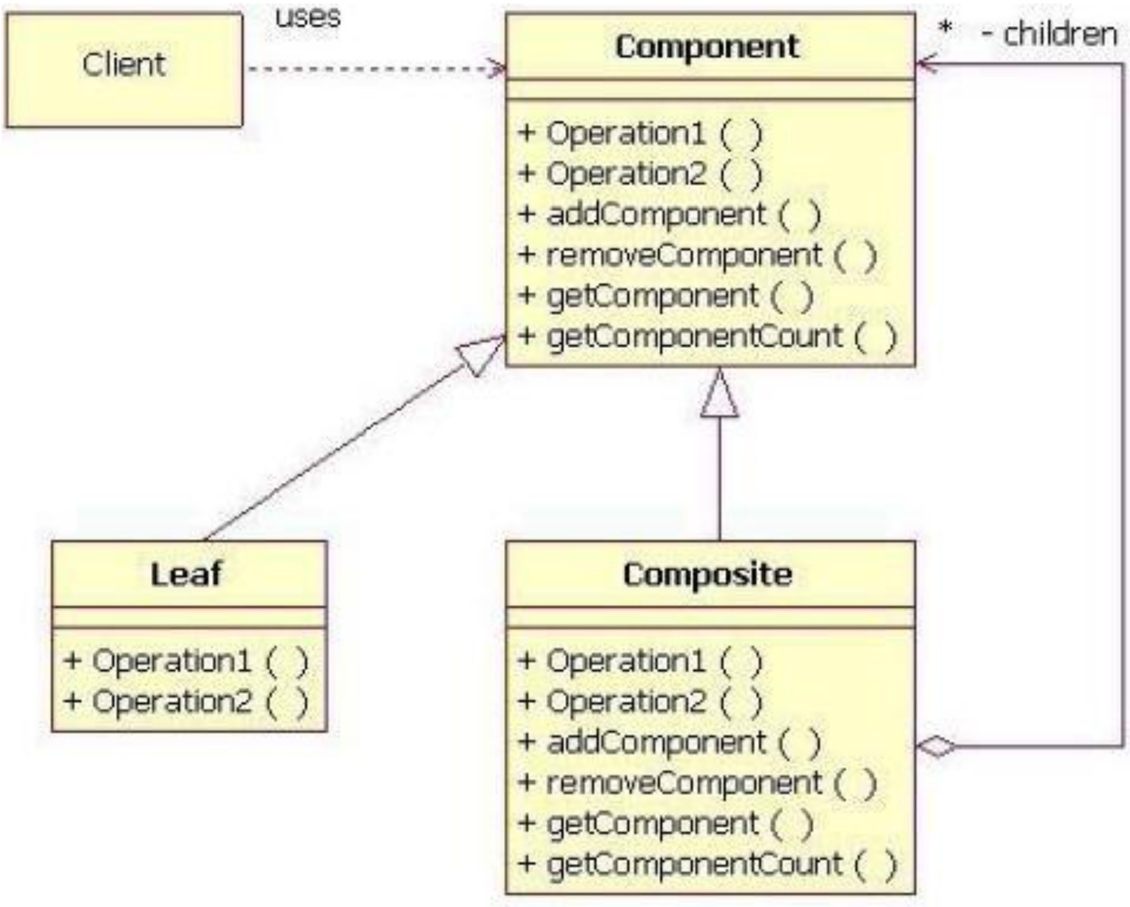


Figure-2

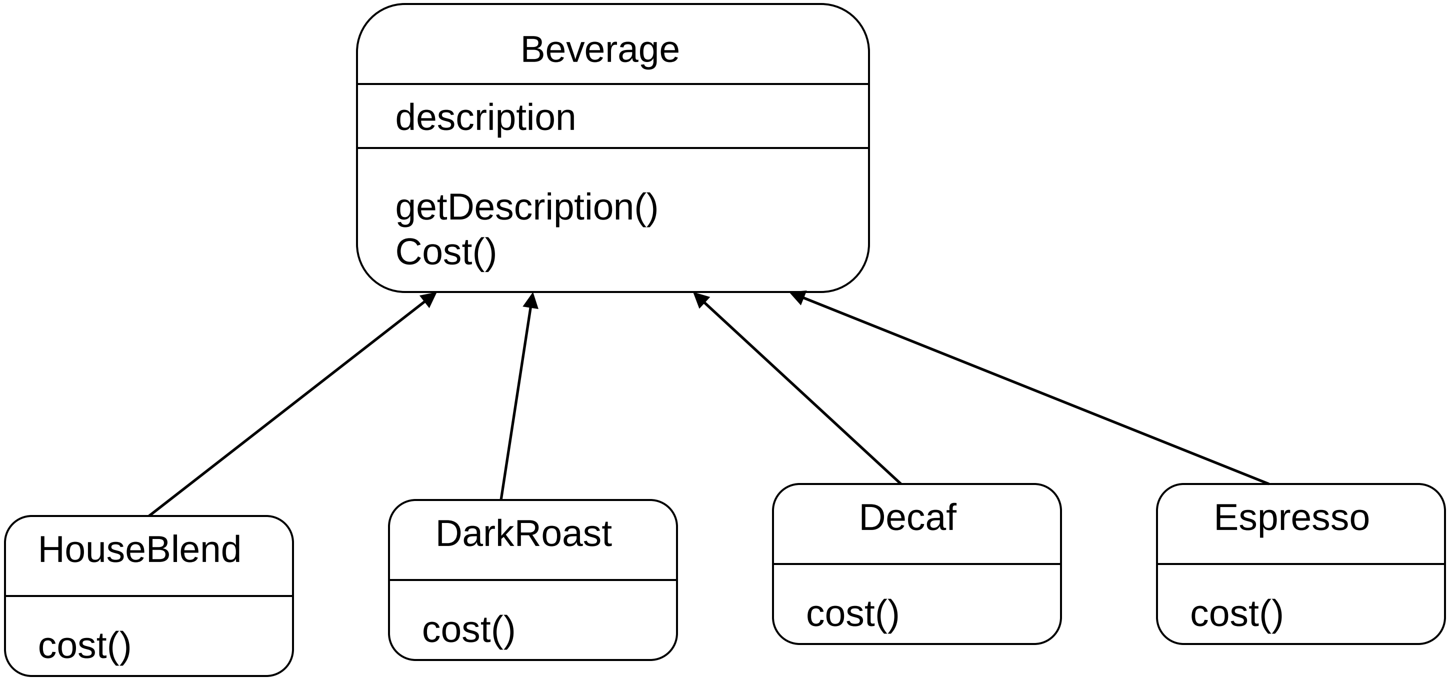
- If you implement Figure-2's Composite pattern, you don't ever have to distinguish between components and composites, and you don't have to cast a Component reference to a Composite instance. So the code fragment listed above reduces to a single line:

```
...
component.addComponent(someComponentThatCouldBeAComposite);
...
```

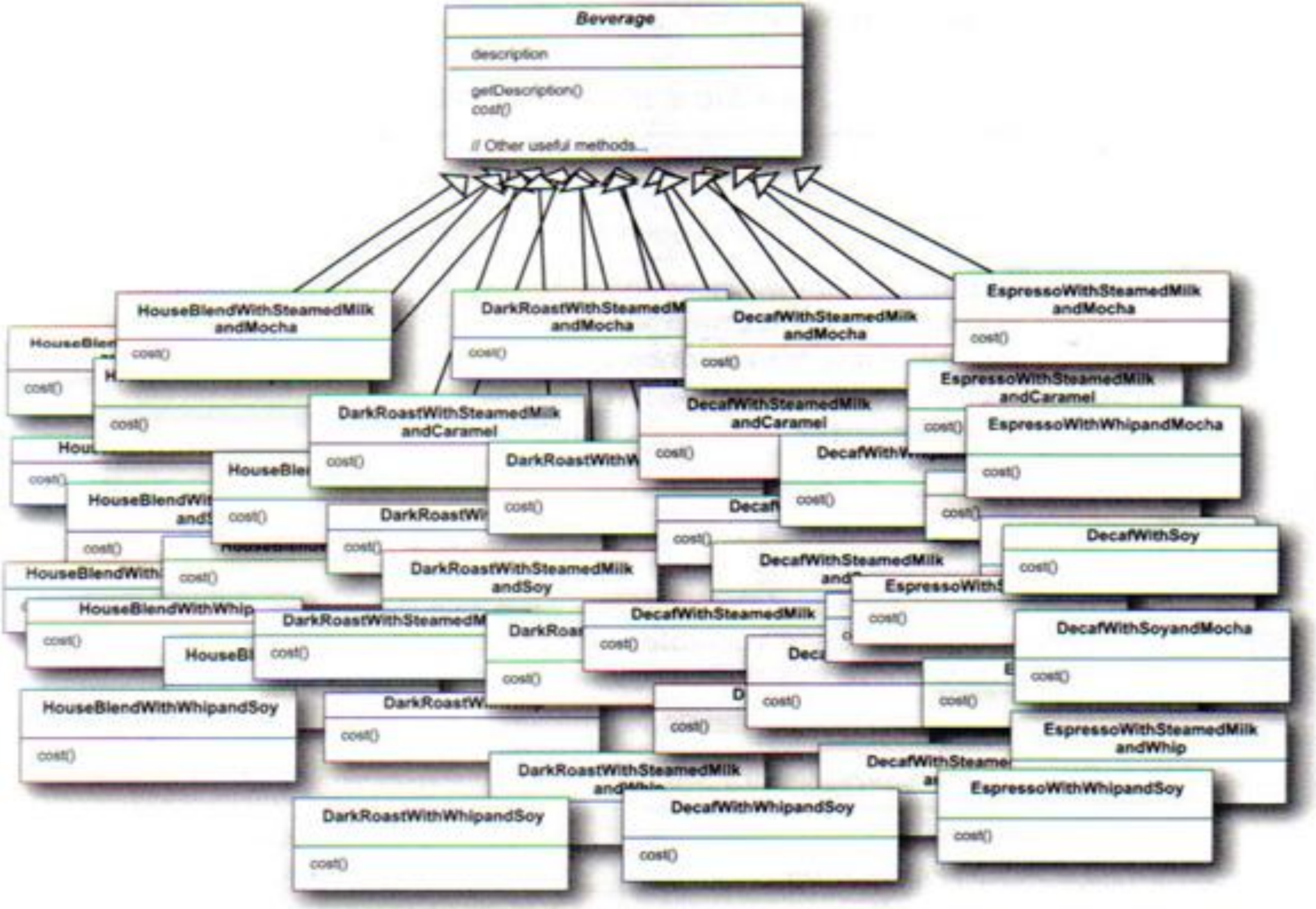
- But, if the Component reference in the preceding code fragment does not refer to a **Composite, what** should the **addComponent() do? That's a major point of contention with Figure 2's Composite pattern** implementation. Because primitive components do not contain other components, adding a component to another component makes no sense, so the **Component.addComponent() method can either fail** silently or throw an exception. Typically, adding a component to another primitive component is considered an error, so throwing an exception is perhaps the best course of action.

- Which Composite pattern implementation would you prefer and why?

Decorator Pattern: A coffee shop example…



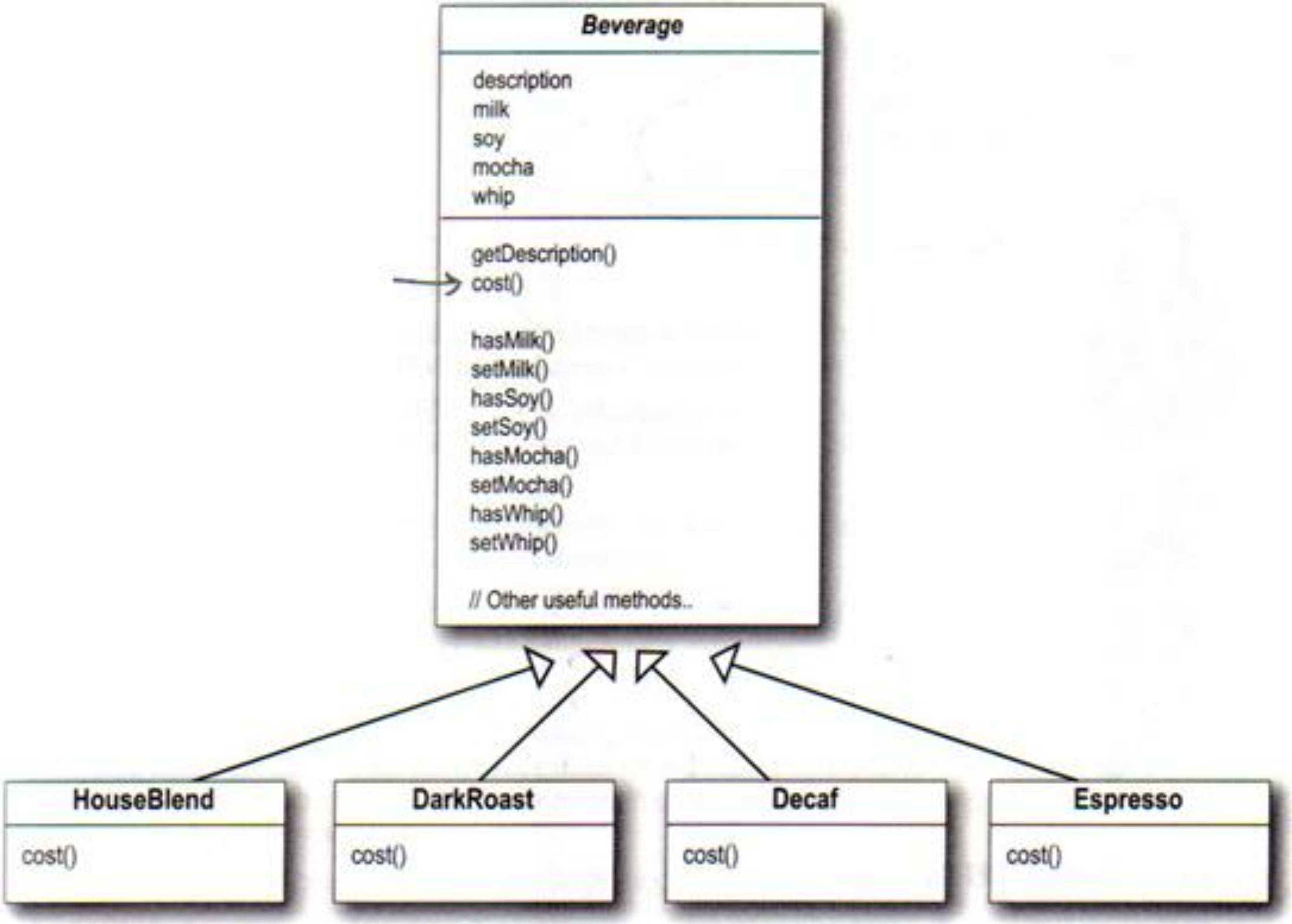
What if you want to show the addition of condiments such as steamed milk, soy, mocha and whipped milk?


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
cost()
DecafWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andCaramel
cost()
EspressoWithWhipandMocha
cost()
DarkRoastWithSteamedMilk andCaramel
cost()
DecafWithSteamedMilk andCaramel
cost()
DarkRoastWithSteamedMilk andSoy
cost()
DecafWithSoy
cost()
DecafWithSteamedMilk
cost()
DecafWithSoyandMocha
cost()
HouseBlendWithWhip
cost()
HouseBlendWithWhipandSoy
cost()
EspressoWithSteamedMilk andWhip
DarkRoastWithSteamedMilk andWhip
EspressoWithWhipandSoy
cost()
DarkRoastWithWhipandSoy
cost()
DecafWithWhipandSoy
cost()

Whoa!
Can you say
"class explosion?"

Each cost method computes the cost of the coffee along with the other condiments in the order.

Beverage class redone



Potential problems with the design so far?

- Solution is not easily extendable
  - How to deal with
    - new condiments
    - Price changes
    - New beverages that may have a different set of condiments – a smoothie?
    - Double helpings of condiments

Design Principle

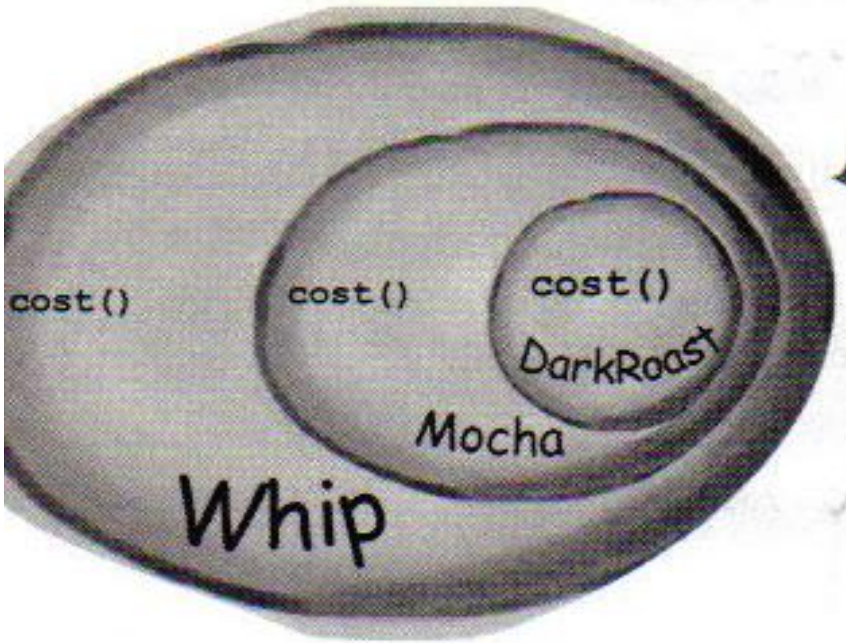
The Open-Closed Principle

Classes should be open for extension, but closed for modification.

The Decorator Pattern

- Take a coffee beverage object – say DarkRoast object
- Decorate it with Mocha
- Decorate it with Whip
- Call the cost method and rely on delegation to correctly compute the composite cost

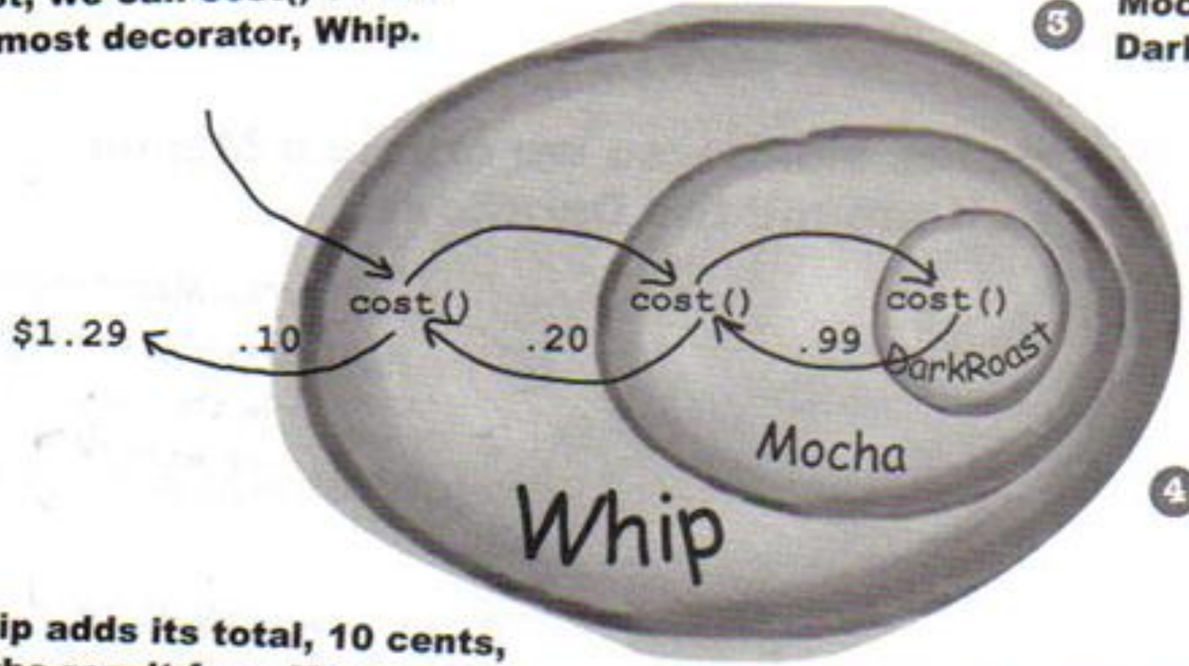
Decorator Pattern approach



Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

DarkRoast wrapped in Mocha and Whip is still
rage and we can do anything with it we can do
DarkRoast, including call its cost() method.

Computing Cost using the decorator pattern



2 Whip calls cost() on Mocha.

1 First, we call cost() on the outmost decorator, Whip.

3 Mocha calls cost() on DarkRoast.

4 DarkRoast returns its cost, 99 cents.

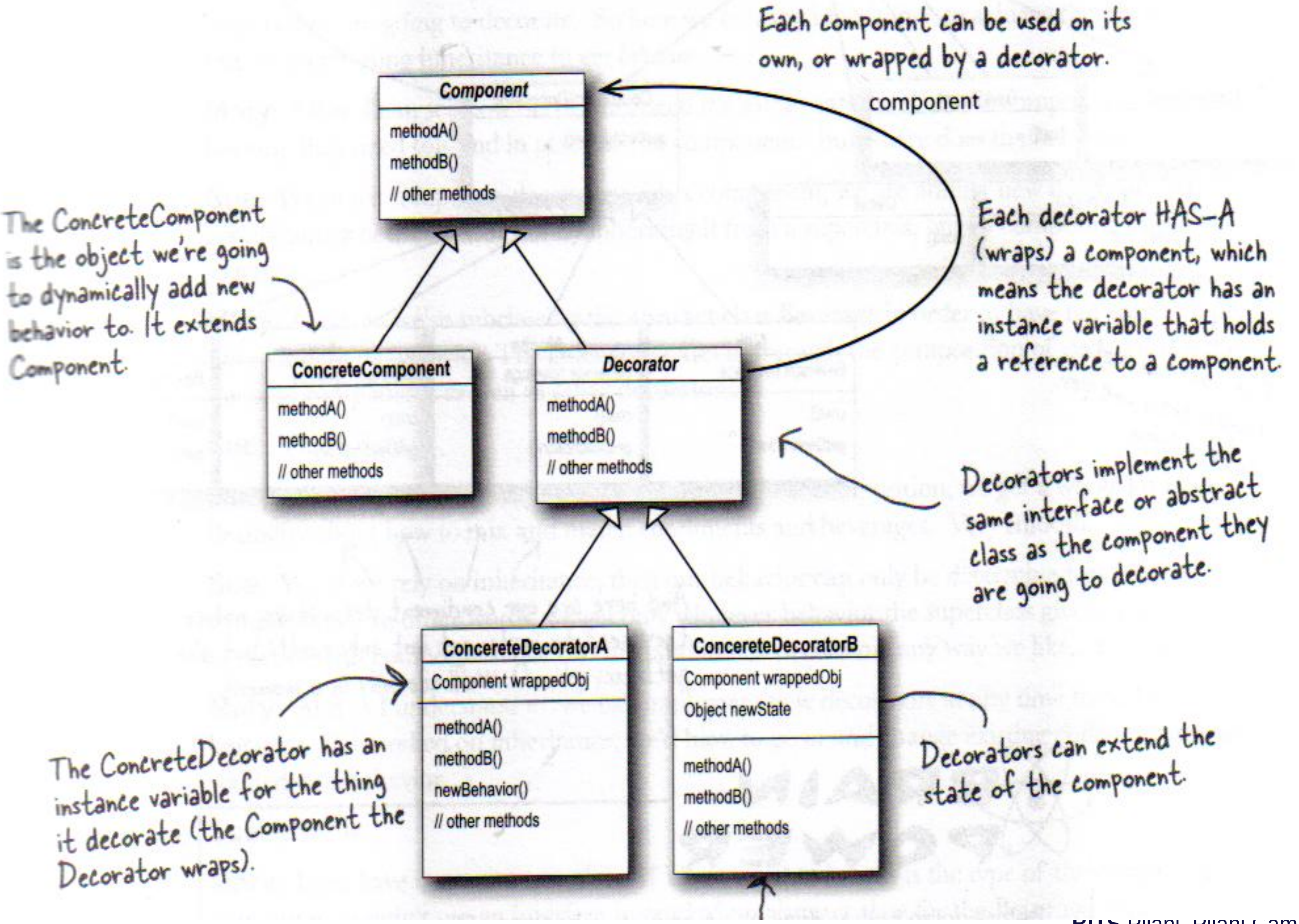
5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

6 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

Decorator Pattern

The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

Decorator Pattern Defined



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

The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.

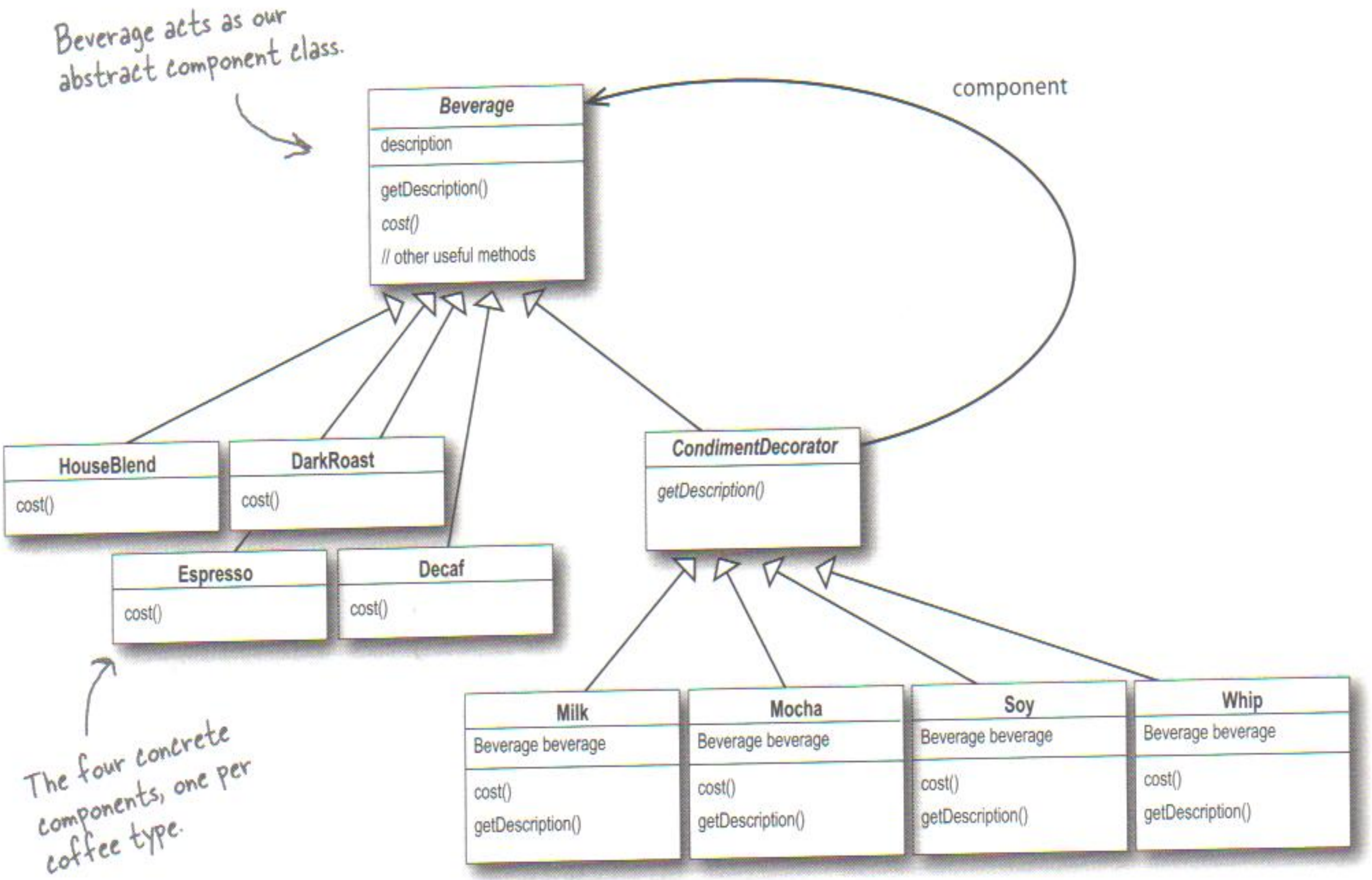
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

Decorators implement the same interface or abstract class as the component they are going to decorate.

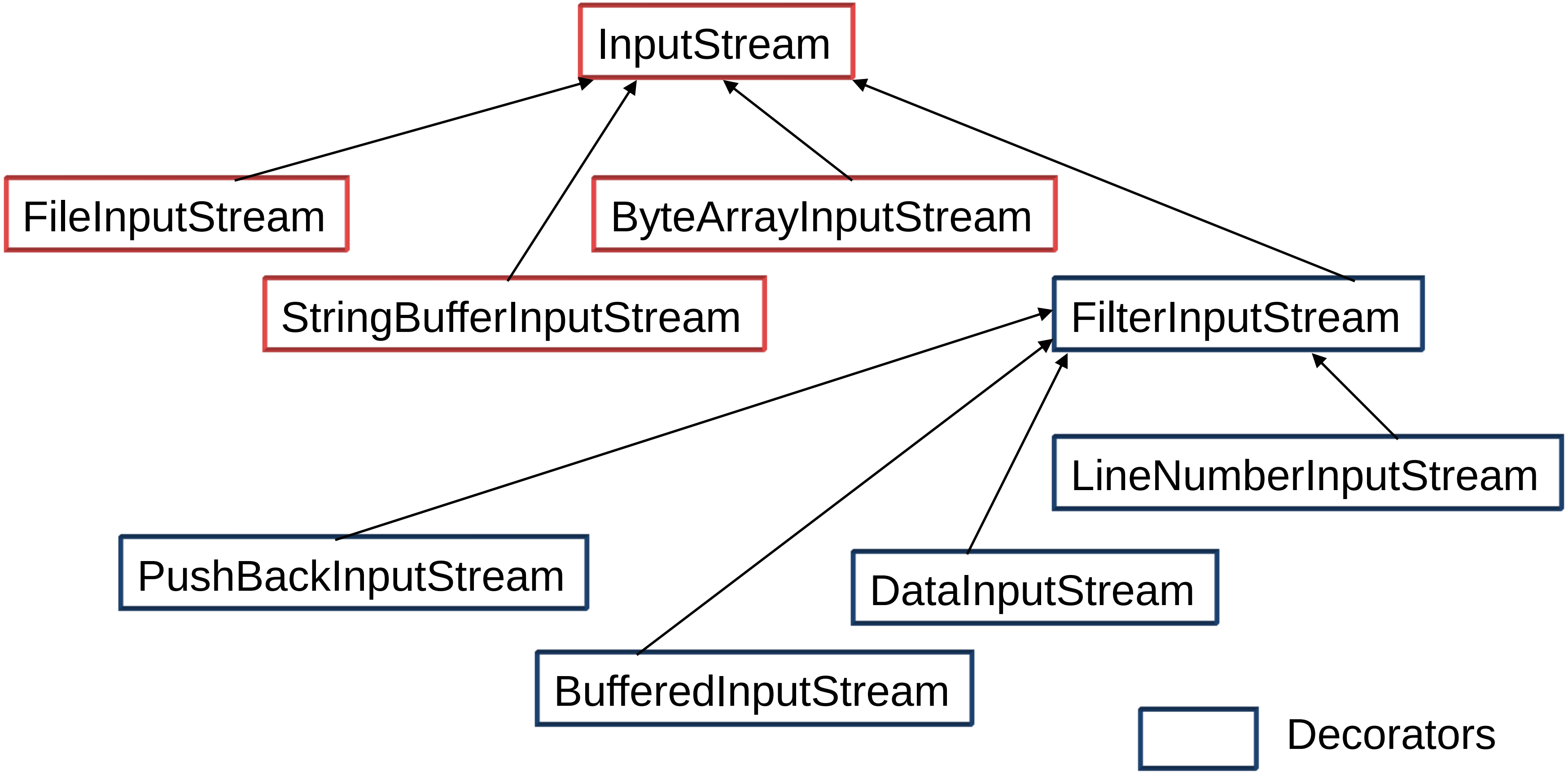
The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).

Decorators can extend the state of the component.

Decorator Pattern for Beverage Example



Decorating Java I/O Classes



InputStream

FileInputStream

StringBufferInputStream

ByteArrayInputStream

FilterInputStream

LineNumberInputStream

PushBackInputStream

DataInputStream

BufferedInputStream

□ Decorators

Keyboard Input

The **System** class in java provides an **InputStream** object: **System.in**

And a buffered **PrintStream** object **System.out**

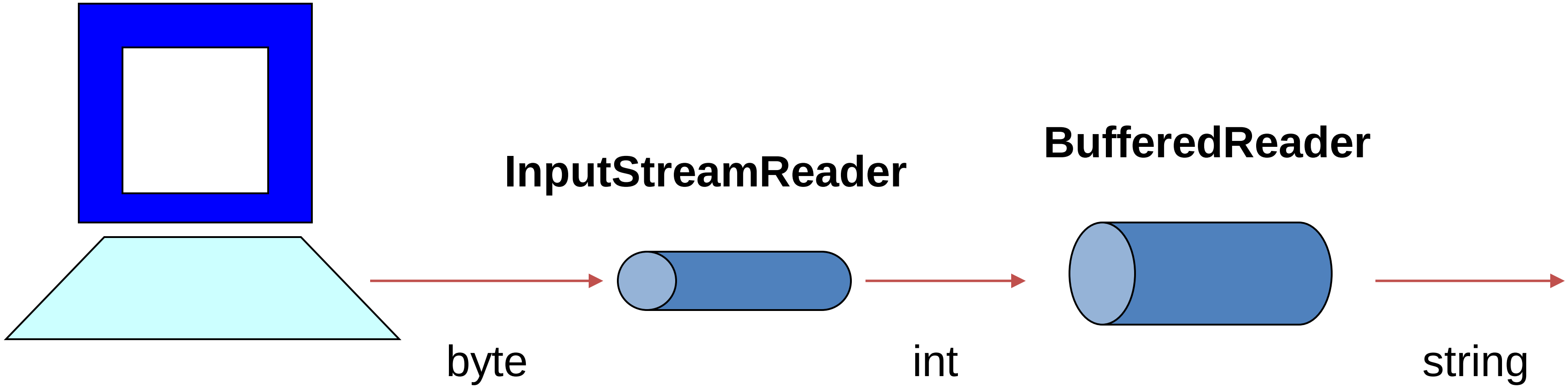
The PrintStream class (**System.out**) provides support for outputting primitive data type values.

However, the InputStream class only provides methods for reading byte values.

To extract data that is at a “higher level” than the byte, we must “encase” the **InputStream**, **System.in**, inside an **InputStreamReader** object that converts byte data into 16-bit character values (**returned as an int**).

We next “wrap” a **BufferedReader** object around the **InputStreamReader** to enable us to use the methods **read**( ) which returns a char value and **readLine**( ), which return a String.

Keyboard Input



import java.io.*; //for keyboard input stream

InputStreamReader isr = **new** InputStreamReader(System.in);

BufferedReader br = **new** BufferedReader(isr);

String st = br.readLine( ); //reads chars until eol and forms string

Thank You