Software Design Patterns

Email: Walaagad@cis.asu.edu.eg

Design Patterns Space

SCOPE	PURPOSE			
		CREATIONAL	STRUCTURAL	BEHAVIORAL
	CLASS	FACTORY METHOD	ADAPTER	INTERPRETER
				TEMPLATE METHOD
	OBJECT	ABSTRACT FACTORY	ADAPTER (OBJECT)	CHAIN OF RESPONSIBILITY
		BUILDER	BRIDGE	COMMAND
		PROTOTYPE	COMPOSITE	ITERATOR
		SINGELTON	DECORATOR	MEDIATOR
			FACADE	MEMENTO
			FLYWEIGHT	OBSERVER
			PROXY	STATE
				STRATEGY
				VISITOR

Structural Patterns

- Proxy
- Decorator
- Adapter
- Façade
- Flyweight
- Composite
- Bridge

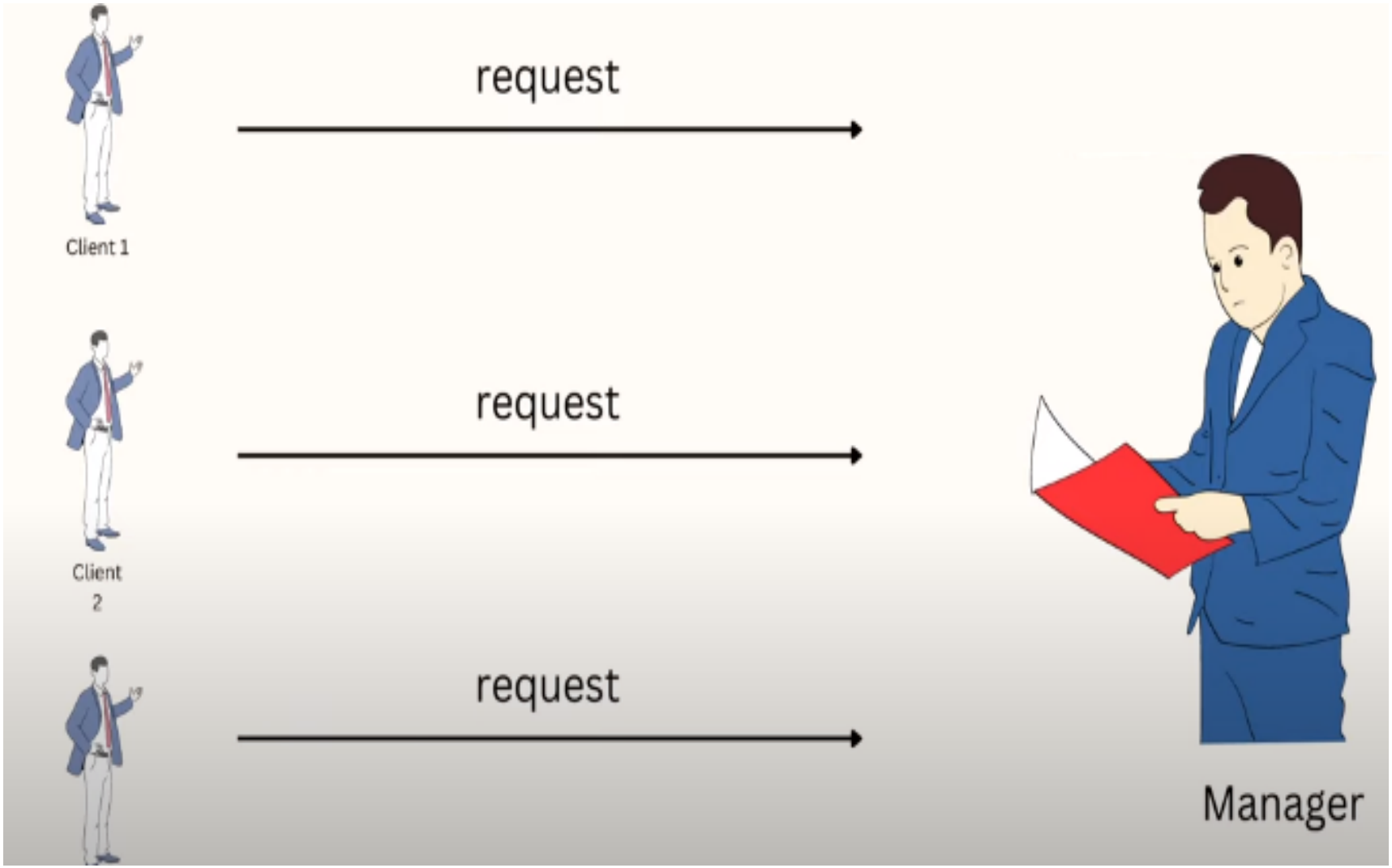
Structural Patterns

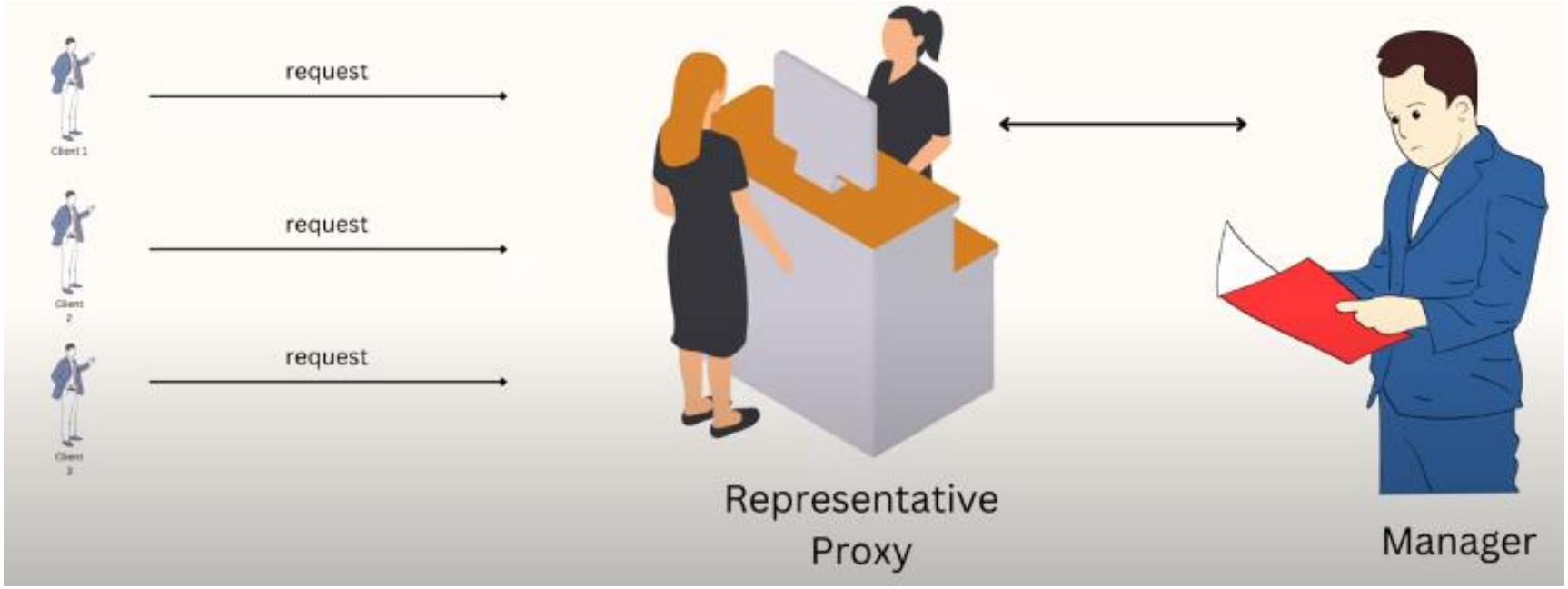
- Proxy
- Decorator
- Adapter
- Façade
- Flyweight
- Composite
- Bridge

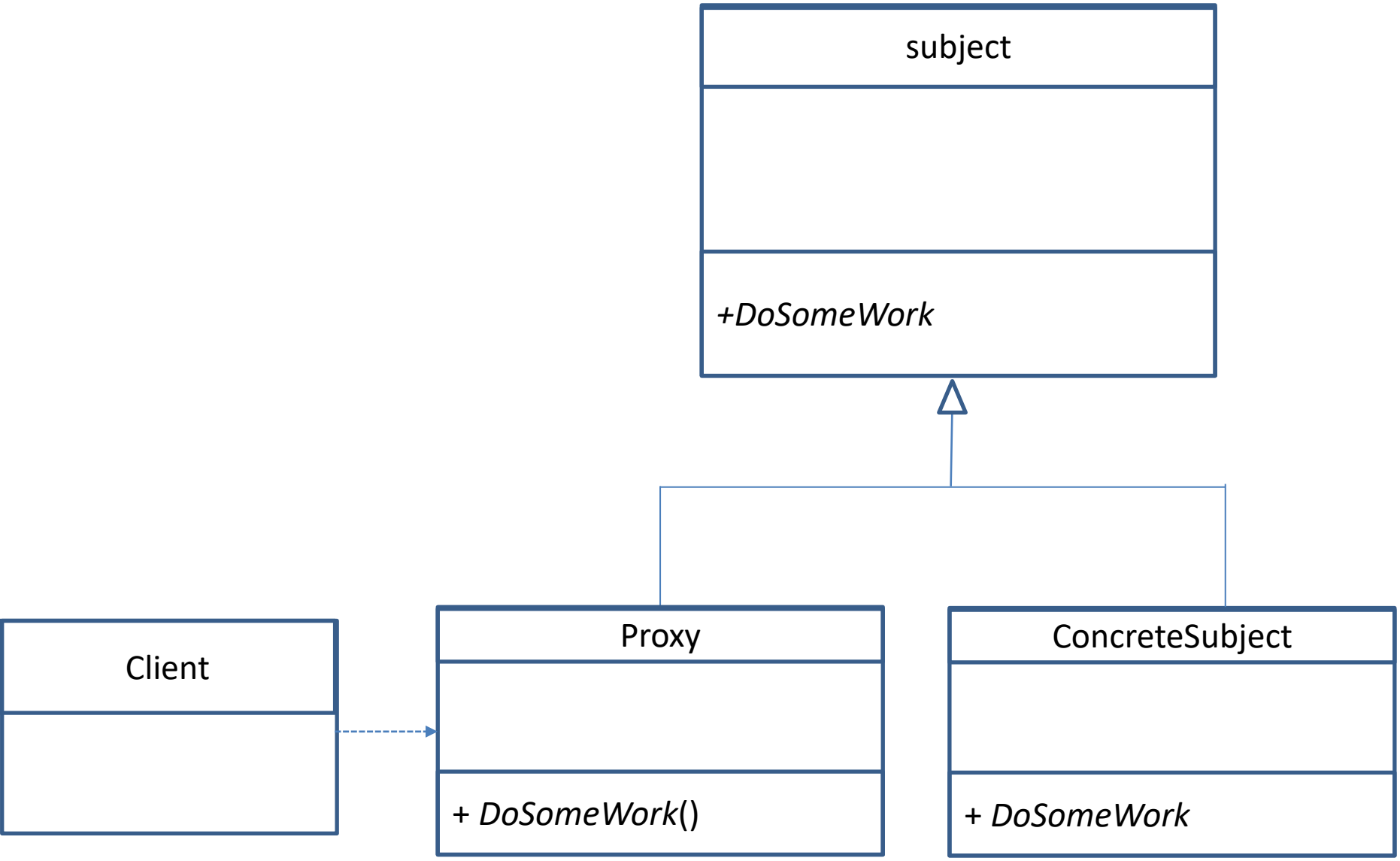
Proxy Pattern

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

Like a gate for an object


request
Client 1
request
Client
2
request
Manager


request
request
request
Representative
Proxy
Manager


subject
+DoSomeWork
Client
Proxy
+ DoSomeWork()
ConcreteSubject
+ DoSomeWork

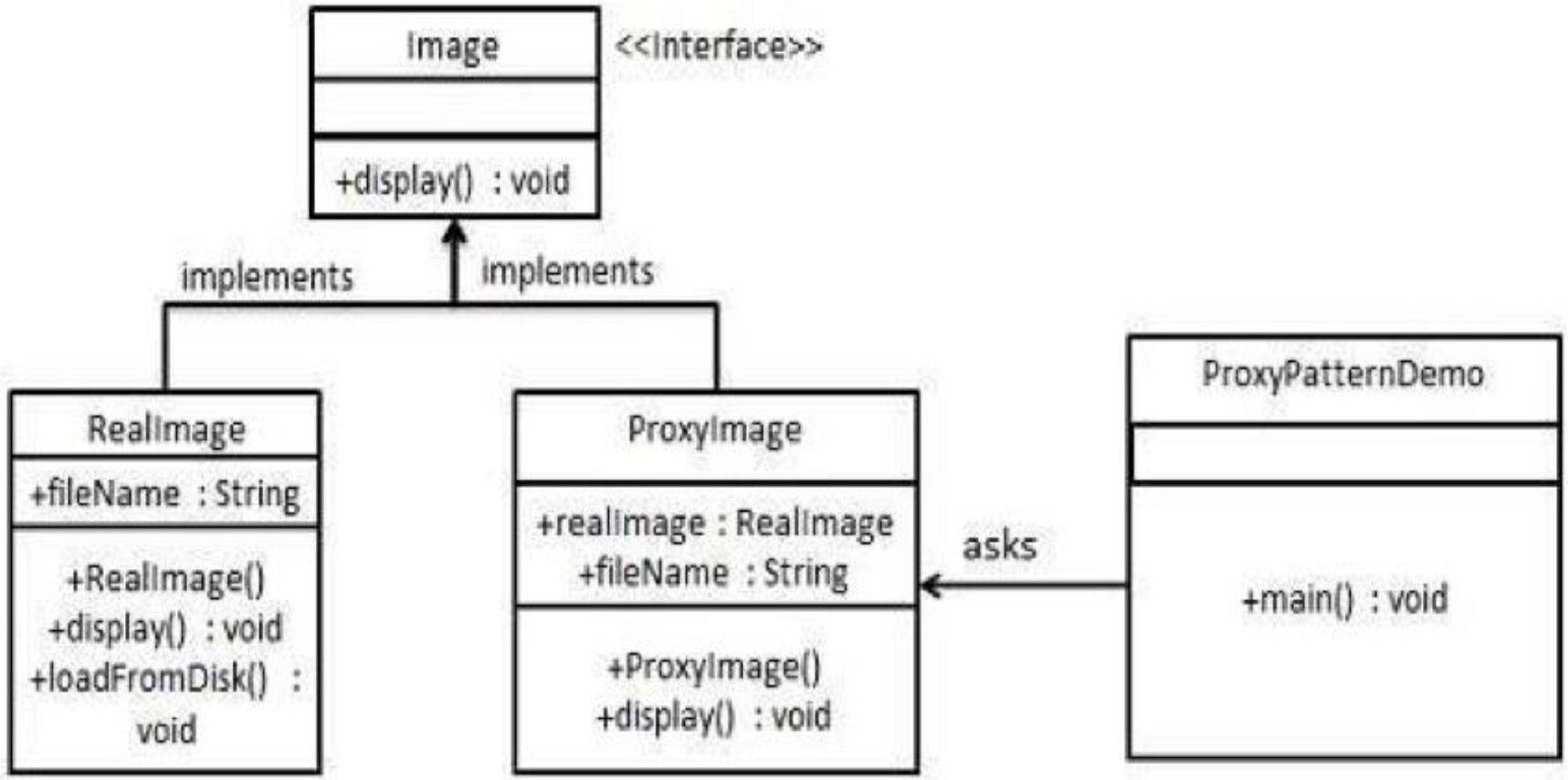
Proxy Pattern

- The proxy design pattern can be best understood with the help of a real-world example.
- In computer networks, we usually come across the term proxy server.
- It is a server application that acts as an intermediary for web requests from clients.

Proxy Pattern

- The client, instead of connecting directly to a server, directs its request to the proxy server which performs the intended filtration and other network transaction.
- The purpose of a proxy server is to simplify and control the complexity of the requests by providing additional benefits such as privacy and security.
- Proxies have been designed to add structure and encapsulation to distributed systems in computer networks.

Proxy Pattern (example1)



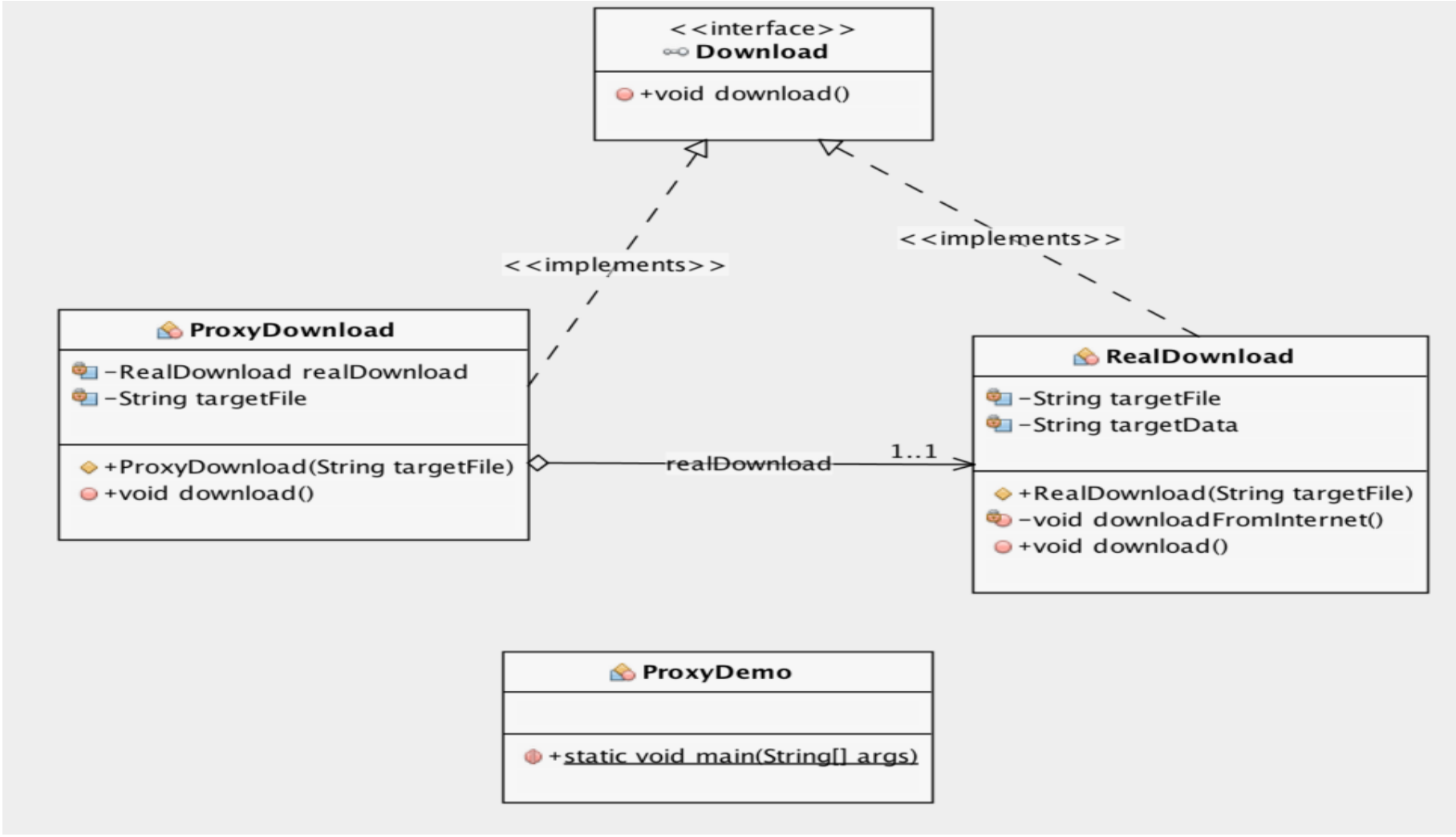
```java
public interface Image {
    void display();
}
```

```java
public class RealImage implements Image {

    private String fileName;

    public RealImage(String fileName){
        this.fileName = fileName;
        loadFromDisk(fileName);
    }

    @Override
    public void display() {
        System.out.println("Displaying " + fileName);
    }

    private void loadFromDisk(String fileName){
        System.out.println("Loading " + fileName);
    }
}
```

```
public class ProxyImage implements Image{

    private RealImage realImage;
    private String fileName;

    public ProxyImage(String fileName){
        this.fileName = fileName;
    }

    @Override
    public void display() {
        if(realImage == null){
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}
```

```
public class ProxyPatternDemo {

    public static void main(String[] args) {
        Image image = new ProxyImage("test_10mb.jpg");

        //image will be loaded from disk
        image.display();
        System.out.println();

        //image will not be loaded from disk
        image.display();
    }
}
```

Proxy Pattern (example2)

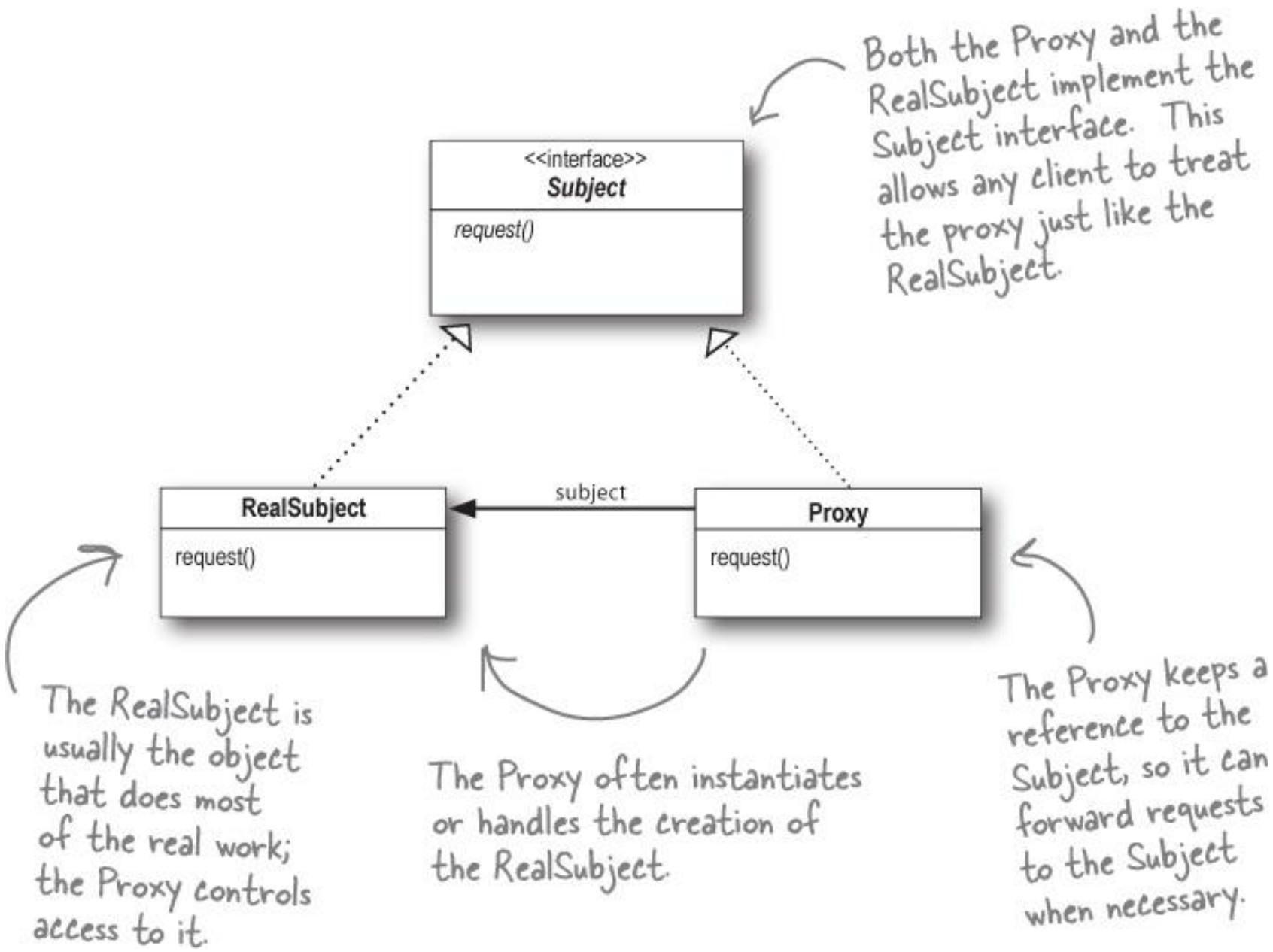


```java
public interface Download
{
    public void download();
}
```

```java
public class RealDownload implements Download
{

    private String targetFile;
    private String targetData;

    public RealDownload(String targetFile)
    {
        this.targetFile = targetFile;
        downloadFromInternet();
    }

    private void downloadFromInternet()
    {
        this.targetData="This is a test data ";
    }

    @Override
    public void download()
    {
        System.out.println(this.targetData);
    }

}
```

```java
public class ProxyDownload implements Download
{

    private RealDownload realDownload;
    private String targetFile;

    public ProxyDownload(String targetFile)
    {
        this.targetFile=targetFile;
    }

    @Override
    public void download()
    {
        if(realDownload==null)
        {
            realDownload=new RealDownload(targetFile);
        }
        realDownload.download();
    }

}
```

```java
public class ProxyDemo
{
    public static void main(String[] args)
    {
        Download download=new ProxyDownload("xyz.movie");
        download.download();
        System.out.println("");

        download.download();
    }
}
```


Both the Proxy and the RealSubject implement the Subject interface. This allows any client to treat the proxy just like the RealSubject.
<<interface>>
Subject
request()
RealSubject
request()
subject
Proxy
request()
The RealSubject is usually the object that does most of the real work; the Proxy controls access to it.
The Proxy often instantiates or handles the creation of the RealSubject.
The Proxy keeps a reference to the Subject, so it can forward requests to the Subject when necessary.

Usages and Applications of Proxy Pattern

Remote Proxy – Using a remote proxy, clients can access objects in a remote location as if they are co-located with them.

Virtual Proxy – A virtual proxy creates an instance of an expensive Object only on demand. I.e., it saves on resources by not creating an instance of an Object heavy on resources until it is needed.

Protection Proxy – A protection proxy regulates access to the original object. Its similar to authorization i.e. object access is controlled based on access rights defined for that Object.

Structural Patterns

- Proxy
- Decorator
- Adapter
- Façade
- Flyweight
- Composite
- Bridge

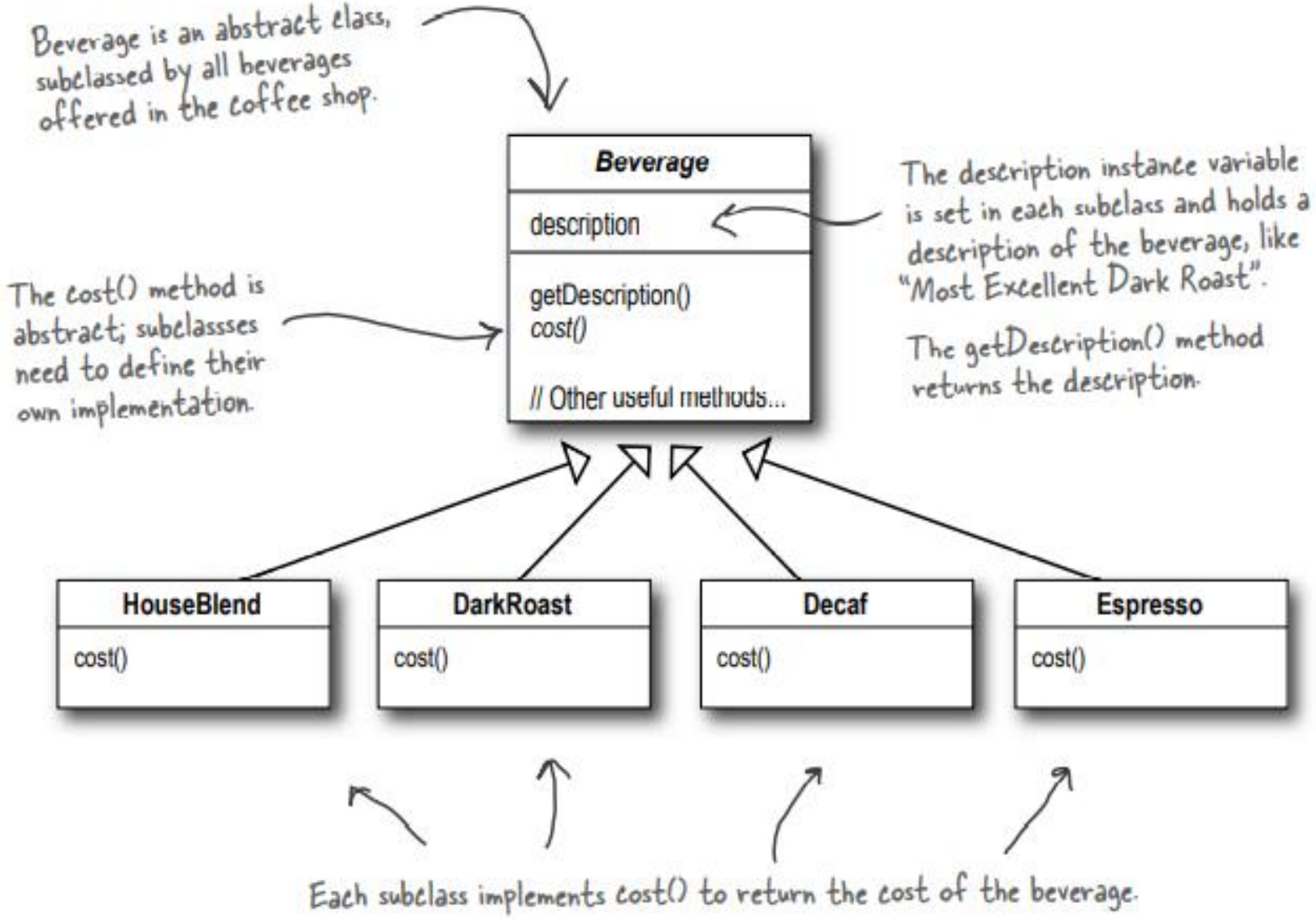
Structural Patterns

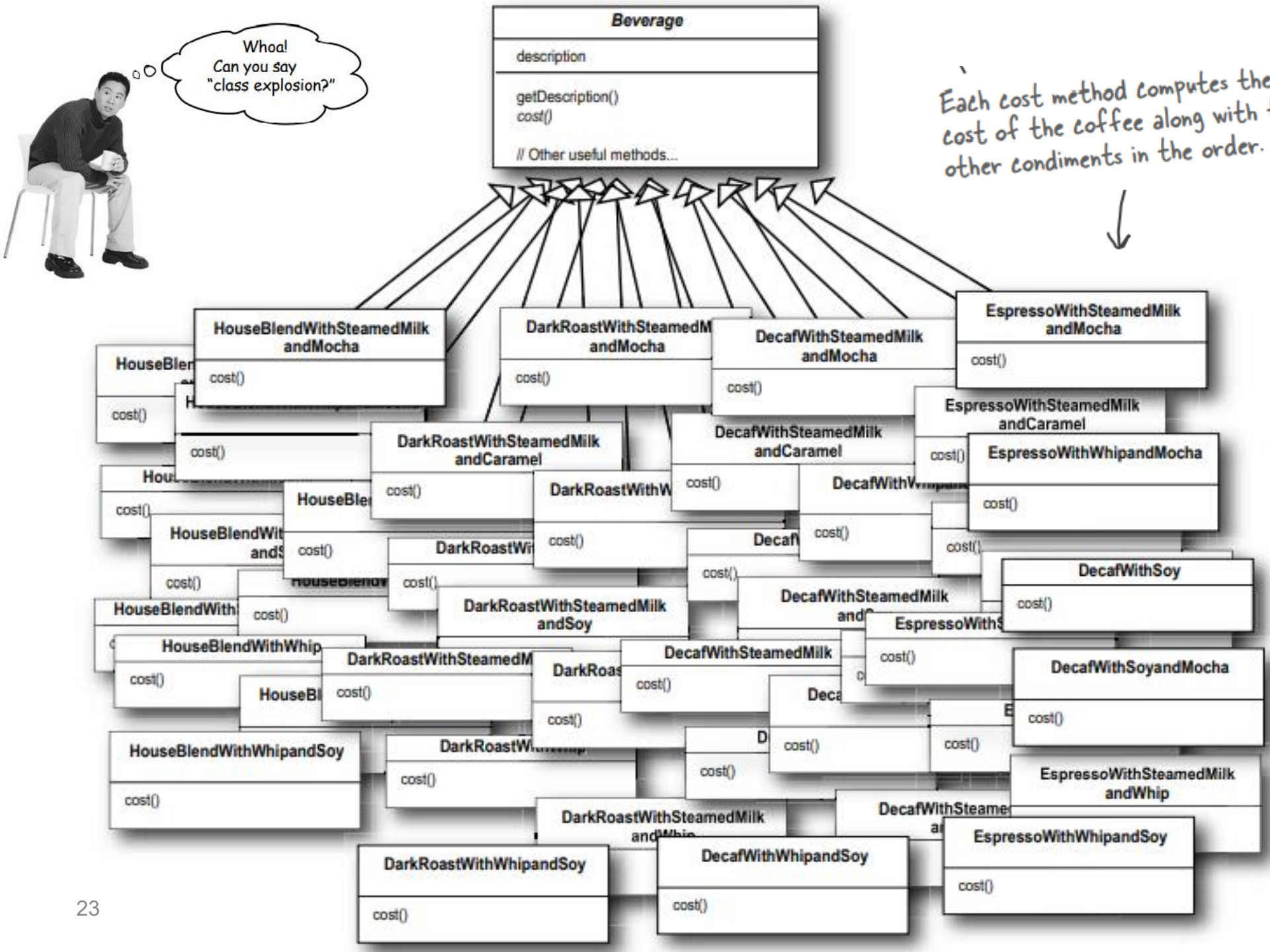
- Proxy
- Decorator
- Adapter
- Façade
- Flyweight
- Composite
- Bridge

Decorator Pattern

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclasses to extend flexibility

It is like wrapper

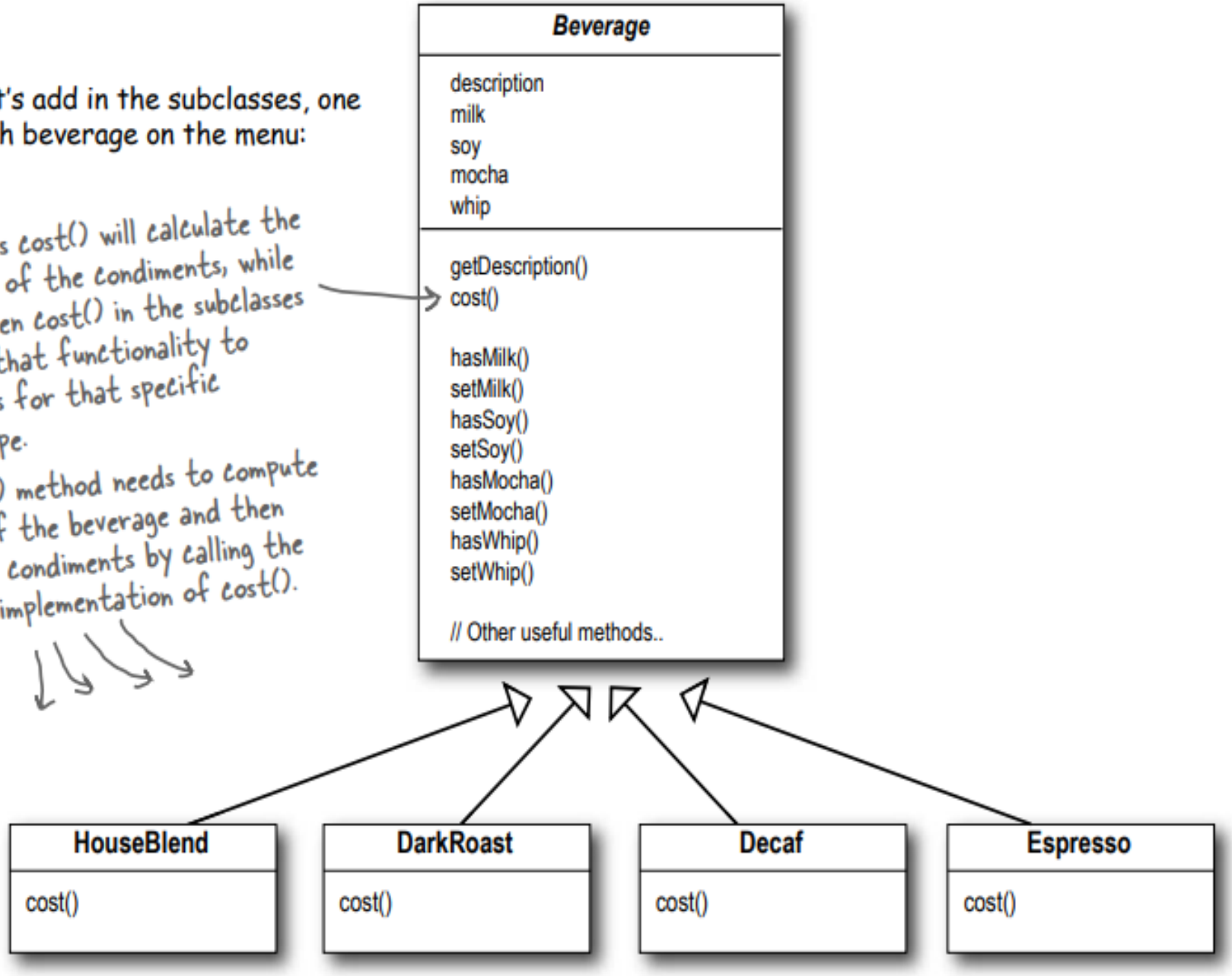

Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.
Beverage
description
getDescription()
cost()
// Other useful methods...
The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".
The getDescription() method returns the description.
The cost() method is abstract; subclassses need to define their own implementation.
HouseBlend
cost()
DarkRoast
cost()
Decaf
cost()
Espresso
cost()
Each subclass implements cost() to return the cost of the beverage.


Whoa! Can you say "class explosion?"
Beverage
description
getDescription()
cost()
// Other useful methods...
Each cost method computes the cost of the coffee along with other condiments in the order.
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
cost()
DecafWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andCaramel
cost()
DarkRoastWithSteamedMilk andCaramel
cost()
DecafWithSteamedMilk andCaramel
cost()
EspressoWithWhipandMocha
cost()
DecafWithSoy
cost()
DarkRoastWithSteamedMilk andSoy
DecafWithSteamedMilk
cost()
DecafWithSoyandMocha
cost()
HouseBlendWithWhipandSoy
cost()
EspressoWithSteamedMilk andWhip
DarkRoastWithSteamedMilk
DarkRoastWithWhipandSoy
cost()
DecafWithWhipandSoy
cost()
EspressoWithWhipandSoy
cost()

Now let's add in the subclasses, one for each beverage on the menu:

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().



What requirements or other factors might change that will impact this design?

Price changes for condiments will force us to alter existing code.

New condiments will force us to add new methods and alter the cost method in the superclass.

We may have new beverages. For some of these beverages (iced tea?), the condiments may not be appropriate, yet the Tea subclass will still inherit methods like hasWhip().

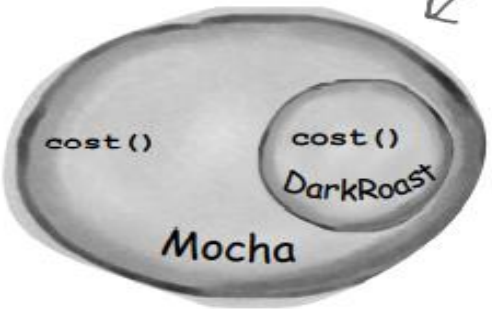
What if a customer wants a double mocha?

Your turn:

❶ We start with our DarkRoast object.

Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

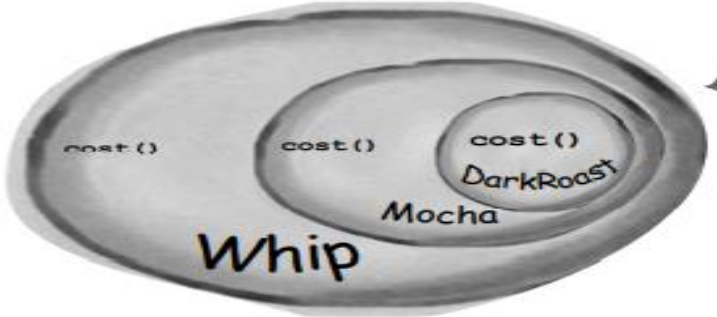
❷ The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.



The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

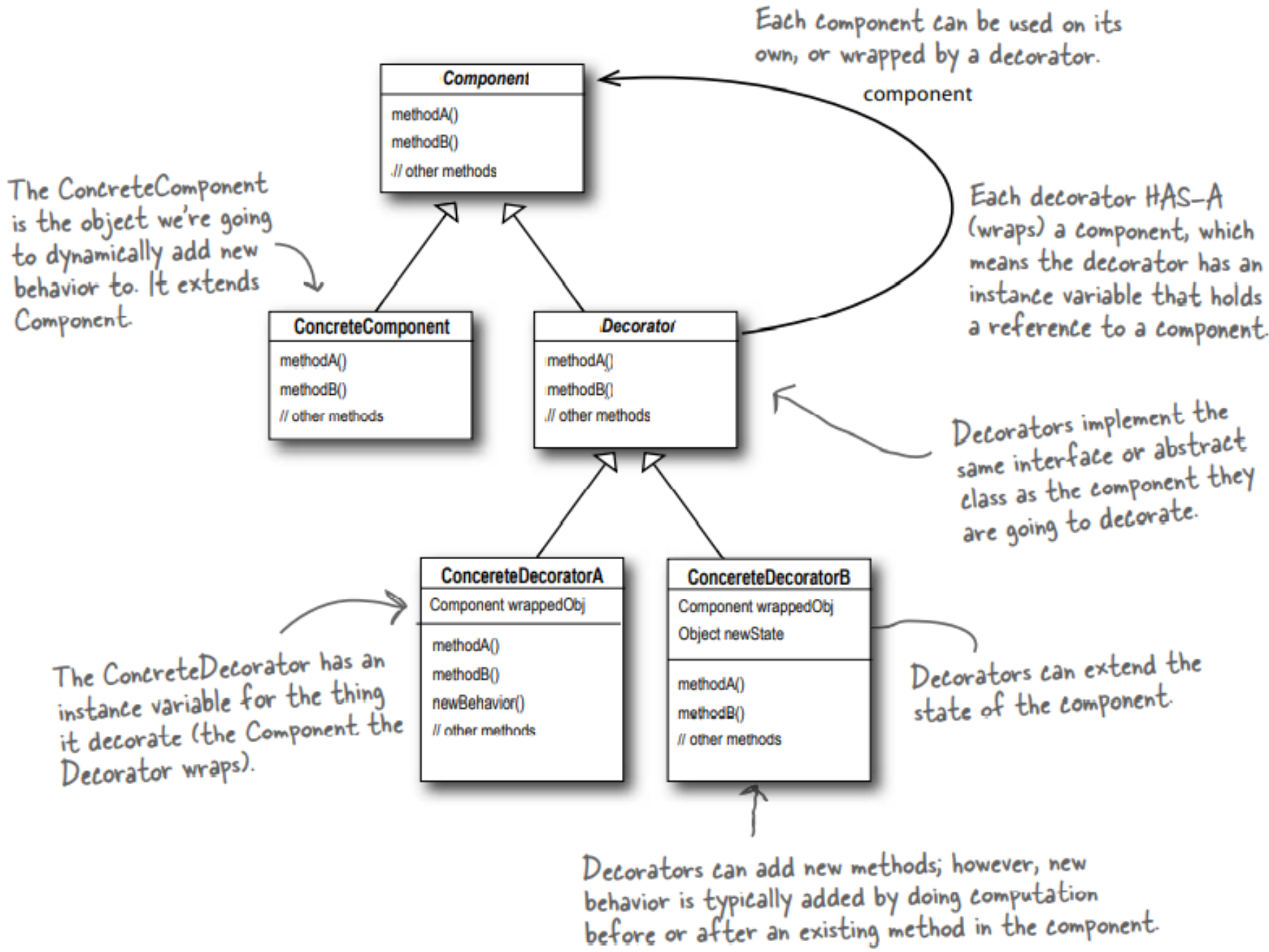
So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

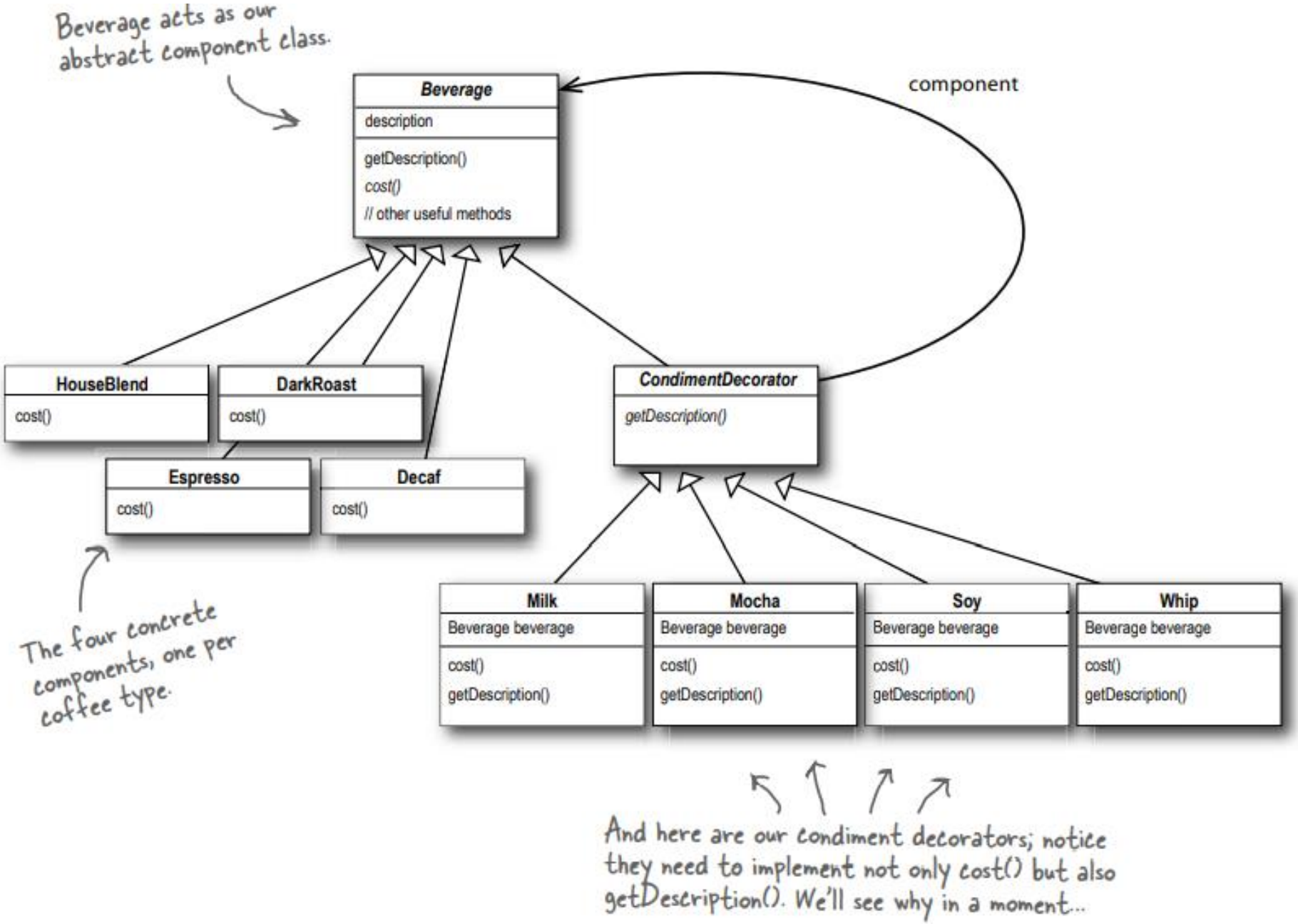
❸ The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



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

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


Each component can be used on its own, or wrapped by a decorator.
component
Component
methodA()
methodB()
// other methods
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
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.
ConcereteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcereteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).
Decorators can extend the state of the component.
Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.


Beverage acts as our abstract component class.
Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
getDescription()
The four concrete components, one per coffee type.
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()
And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

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

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

Beverage is an abstract class with the two methods getDescription() and cost().

getDescription is already implemented for us, but we need to implement cost() in the subclasses.

```
public class Espresso extends Beverage {

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

    public double cost() {
        return 1.99;
    }
}
```

```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

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

```
public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}
```

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

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

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

```
public static void main(String args[]) {
    Beverage beverage = new Espresso();
    System.out.println(beverage.getDescription()
            + " $" + beverage.cost());

    Beverage beverage2 = new DarkRoast();
    beverage2 = new Mocha(beverage2);
    beverage2 = new Mocha(beverage2);
    beverage2 = new Whip(beverage2);
    System.out.println(beverage2.getDescription()
            + " $" + beverage2.cost());

    Beverage beverage3 = new HouseBlend();
    beverage3 = new Soy(beverage3);
    beverage3 = new Mocha(beverage3);
    beverage3 = new Whip(beverage3);
    System.out.println(beverage3.getDescription()
            + " $" + beverage3.cost());
}
```

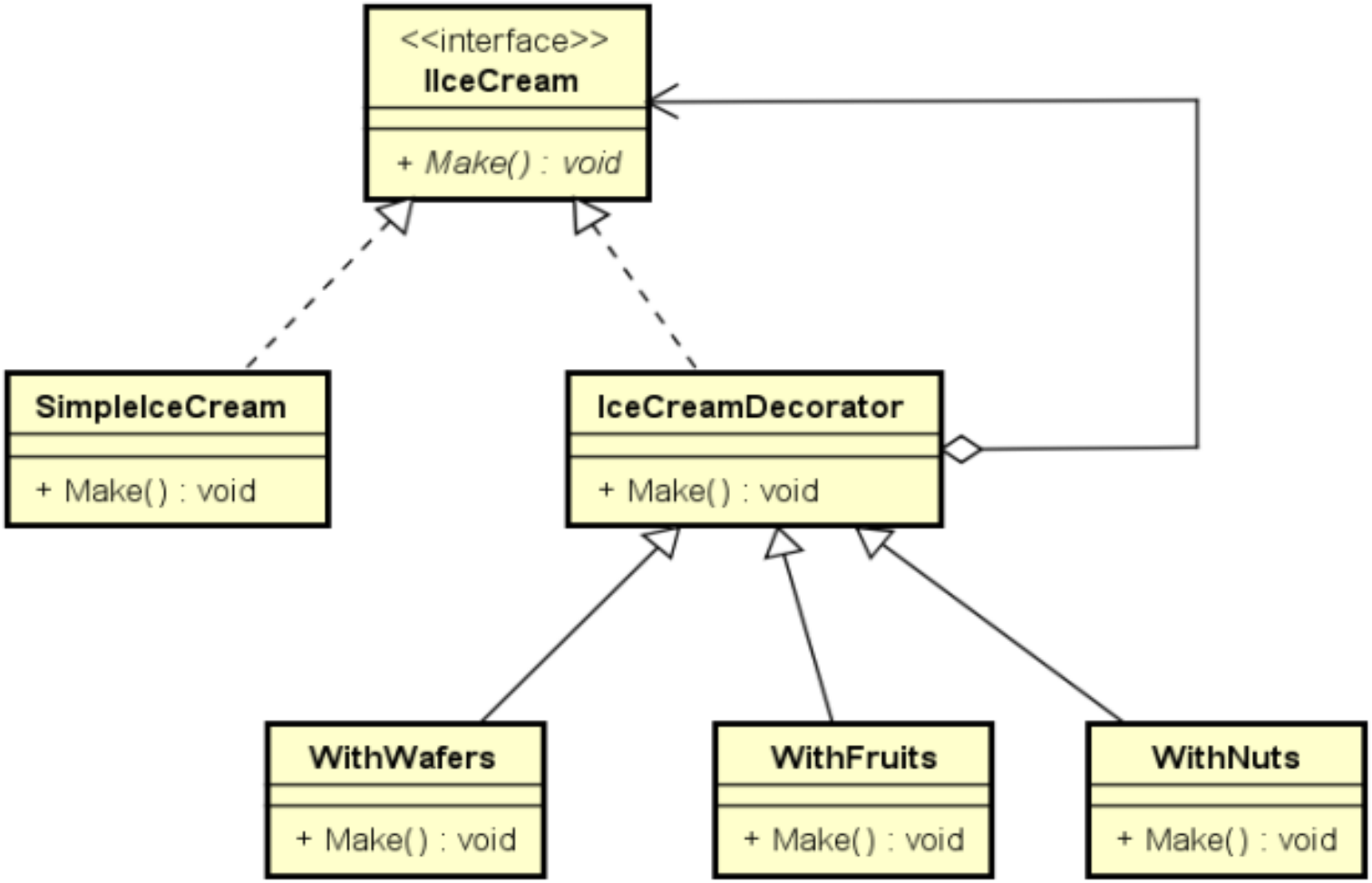
Finally, give us a HouseBlend with Soy, Mocha, and Whip.

```
File Edit Window Help CloudsInMyCoffee
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```

Decorator Pattern advantages

Decorator design pattern is useful in providing runtime modification abilities and hence more flexible. It is easy to maintain and extend when the amount of choices are more.

Ice Cream Example




PC CD-ROM
NEED FOR SPEED
UNDERGROUND
EA GAMES

Weapon
+aim_and_fire()
BaseWeapon
+aim_and_fire()
WeaponAccessory
+aim_and_fire()