Design Patterns

Agenda

- Why Design Pattern
- Objective of Design Pattern
- Design Principles
- Design Pattern Classification
- Creational Design Pattern
- Structural Design Pattern
- Functional Design Pattern

Why Design Pattern

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software **design**. A **design pattern** isn't a finished **design** that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Look at below points to understand the need of design patterns:

- Design Patterns are the approaches to be adopted for similar kind of situations (Time Management).
- Facilitate communication among developers by providing a common language (Hey!! I have used Decorator design pattern here).
- Improve design documentation.
- Improve understandability of design.

OOAD Design Principle

OOAD Design Principle

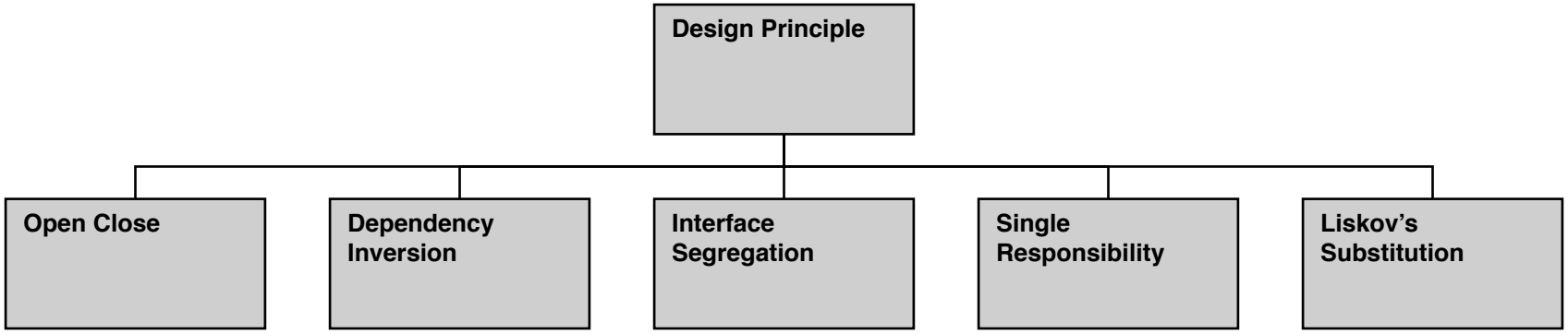
Software design principles represent a set of guidelines that helps us to avoid having a bad design. We can say that a design is bad if it has below characteristics:

- Rigidity - It is hard to change because every change affects too many other parts of the system.
- Fragility - When you make a change, unexpected parts of the system break.
- Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

Rule Of Thumb: We can define certain criteria of a bad design but there is no perfect way to have a best design which can not be improved. We should choose the best available approach in terms of reusability, simplicity, maintainability and, of course, better performance in available infrastructure.

We always have possibility of improvements.

Design Principle Classification



Open Close Design Principle

Software entities like classes, modules and functions should be open for extension but closed for modifications.

The **Open Close Principle** states that the design and writing of the code should be done in a way such that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they don't use.

The **Interface Segregation Principle** states that clients should not be forced to implement interfaces they don't use. Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one sub-module.

Single Responsibility Principle

A class should have only one reason to change..

In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

Liskov's Substitution Principle

Derived types must be completely substitutable for their base types.

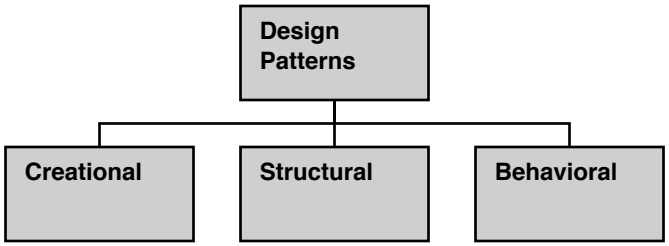
All the time we design a program module and we create some class hierarchies. Then we extend some classes creating some derived classes.

We must make sure that the new derived classes just extend without replacing the functionality of old classes. Otherwise the new classes can produce undesired effects when they are used in existing program modules.

GoF OOAD Design Patterns

Design Pattern Classification

Creational: In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or in added complexity to the design.

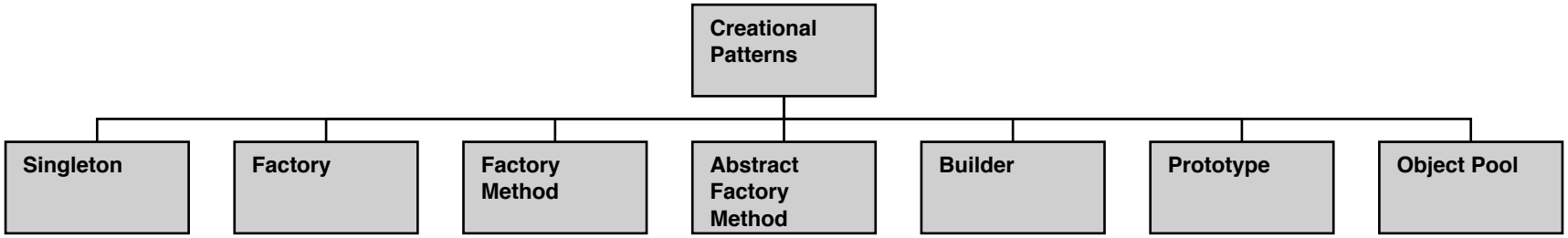


Structural: Structural design patterns are design patterns that ease the design by identifying a simple way to realise relationships between entities. Examples of Structural Patterns include: Adapter pattern: 'adapts' one interface for a class into one that a client expects.

Behavioural: Behavioural design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Creational Design Patterns

Creational Patterns Classification



Singleton Design Patterns

Single Object of a class

The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance; in the same time it provides a global point of access to that instance. In this case the same instance can be used from everywhere, being impossible to invoke directly the constructor each time. We use singleton design patterns in below scenarios:

- If there is no need to have more than one object of the class.
- If a class do not have instance variables.
- If state of a class does not effect the business.

Lazy Loaded Singleton Class

```
package hs.design.singleton;

public class Singleton {
    private static Singleton obj;

    // Private constructor suppresses generation of a (public) default
    // constructor
    private Singleton() {
    }

    public synchronized static Singleton getInstance() {
        if (obj == null){
            // lazy,when this method will get called first time
            obj = new Singleton();
        }
        return obj;
    }
}
```

Early Loaded Singleton Class

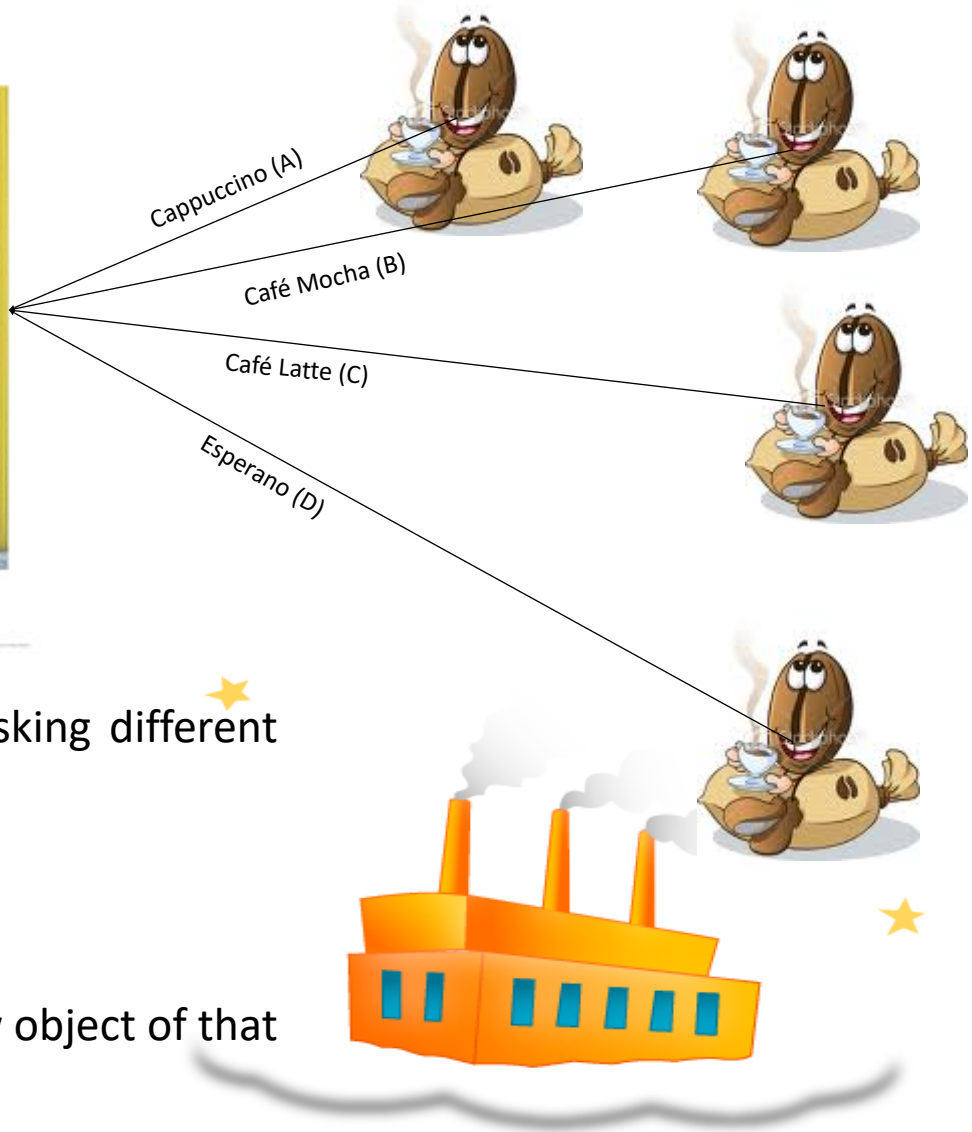
```
package hs.design.singleton;

public class EarlySingletone {
    // early, when this class get loaded
    private static EarlySingletone obj = new EarlySingletone();

    // Private constructor suppresses generation of a (public) default
    // constructor

    private EarlySingletone() {
    }

    public static EarlySingletone getInstance() {
        return obj;
    }
}
```

Factory Design Patterns

Factory Pattern Definition

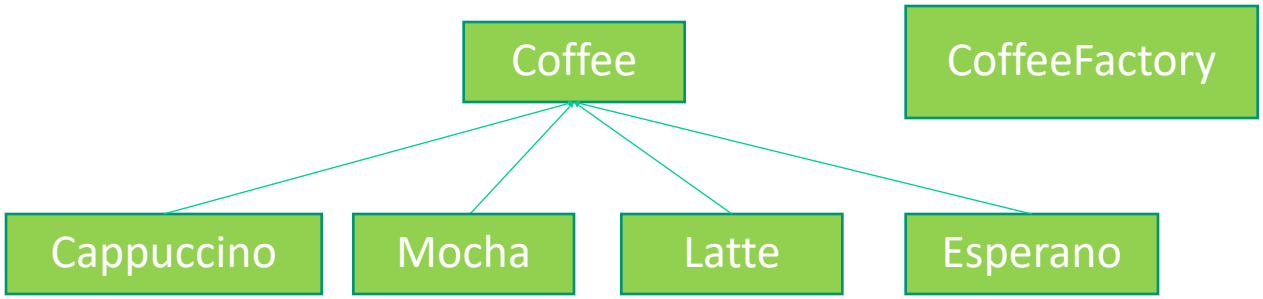
- The Factory Method Pattern is all about "Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses" Thus, as defined by Gamma et al, "The Factory Method lets a class defer instantiation to subclasses".

Coffee Shop Problem



- Coffee Shop has different customers asking different coffee.
- Every coffee is associated with a code.
- Each order comes up with the code.
- For each order, we need to create a new object of that type of coffee.

Design with Factory Pattern




```
public class CoffeeFactory {

    public static Coffee getCoffeeObject(String option) {
        Coffee coffee = null;
        if ("A".equals(option)) {
            coffee = new Cappuccino();
        } else if ("B".equals(option)) {
            coffee = new MochaCoffee();
        } else if ("C".equals(option)) {
            coffee = new Latte();
        } else if ("D".equals(option)) {
            coffee = new Esperano();
        }
        return coffee;
    }
}
```

```
public class CoffeeTest {
    public static void main(String[] args) {
        if (args == null || args.length < 1) {
            System.out.println("Pass appropriate parameter");
            return;
        }
        String option = args[0];
        //Create the object from Factory
        Coffee coffee = CoffeeFactory.getCoffeeObject(option);
        if (coffee == null) {
            System.out.println("Invalid option");
            return;
        }
        coffee.deliver();
    }
}
```

Builder Design Patterns

Coffee Shop Design With Builder Pattern

- Builder is used to separate the construction of complex objects from their representation so the construction process can be used to create different representations.
- The construction logic is isolated from the actual steps used to create the complex object and, therefore, the construction process can be reused to create different complex objects from the same set of simple objects.

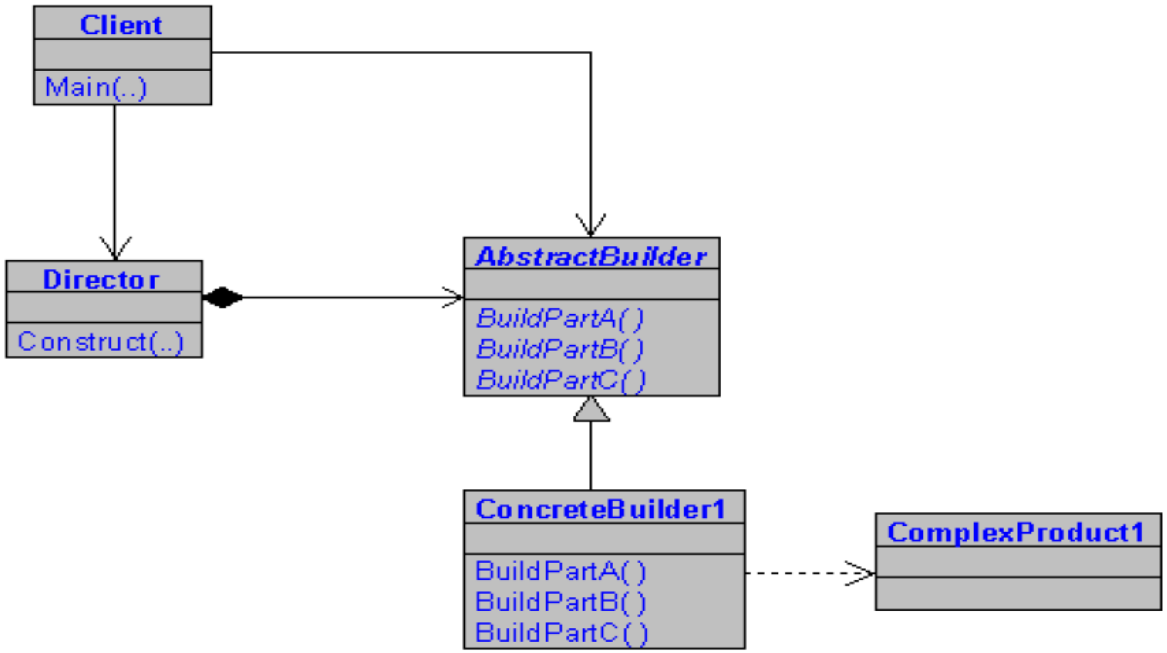
Coffee Shop Code with Builder Pattern

```java
package builder.hs.builder;

import builder.hs.coffee.Coffee;

public abstract class AbstractCoffeBuilder {

    protected Coffee coffee = new Coffee();

    protected abstract void setType();

    protected abstract void setCoffeeQuantity();

    protected abstract void setMilkQuantity();

    public Coffee createCoffee(){
        setCoffeeQuantity();
        setMilkQuantity();
        setType();
        return coffee;
    }
}
```

```java
public class CappuccinoCoffeeBuilder
        extends AbstractCoffeBuilder {
    public void setCoffeeQuantity() {
        coffee.setCoffeeQuantity(5);
    }
    public void setMilkQuantity() {
        coffee.setMilkQuantity(200);
    }
    public void setType() {
        coffee.setType("Cappuccino");
    }
}
```

```java
public class MochaCoffeeBuilder
        extends AbstractCoffeBuilder {
    public void setCoffeeQuantity() {
        coffee.setCoffeeQuantity(4);
    }
    public void setMilkQuantity() {
        coffee.setMilkQuantity(150);
    }
    public void setType() {
        cof
    }
}
```

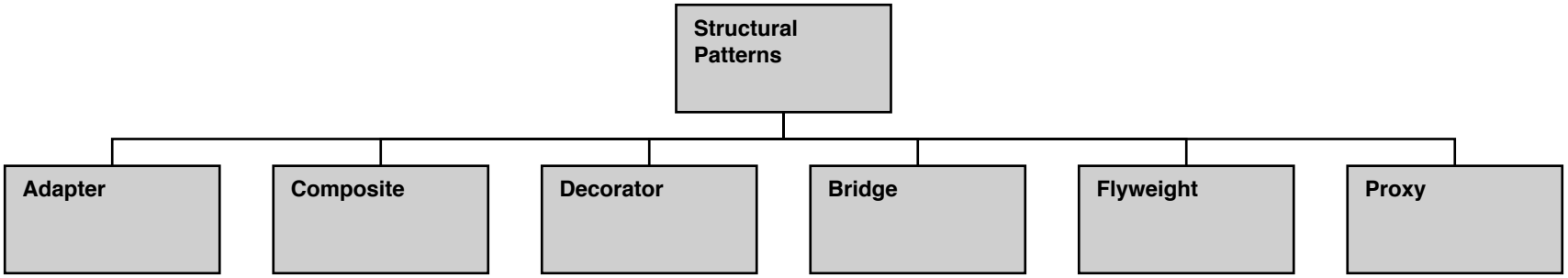
Think About Rest of The Code



- Builder - Abstract interface for creating products.
- Concrete Builder - Provides implementation for Builder and constructs and assembles parts to build the products.
- Director - Construct an object using the Builder pattern.
- Product - The complex object under construction

Structural Design Patterns

Structural Patterns Classification

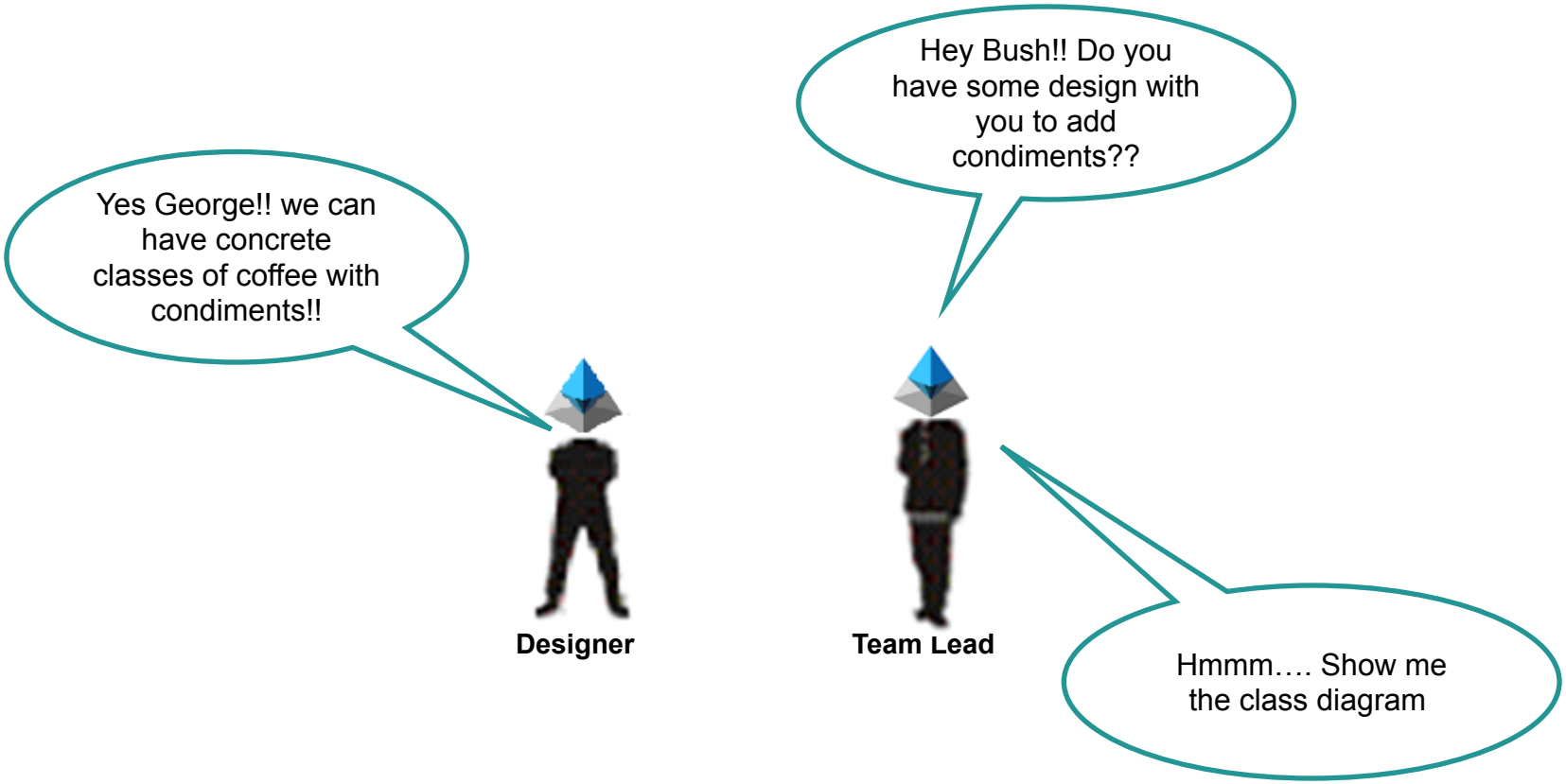


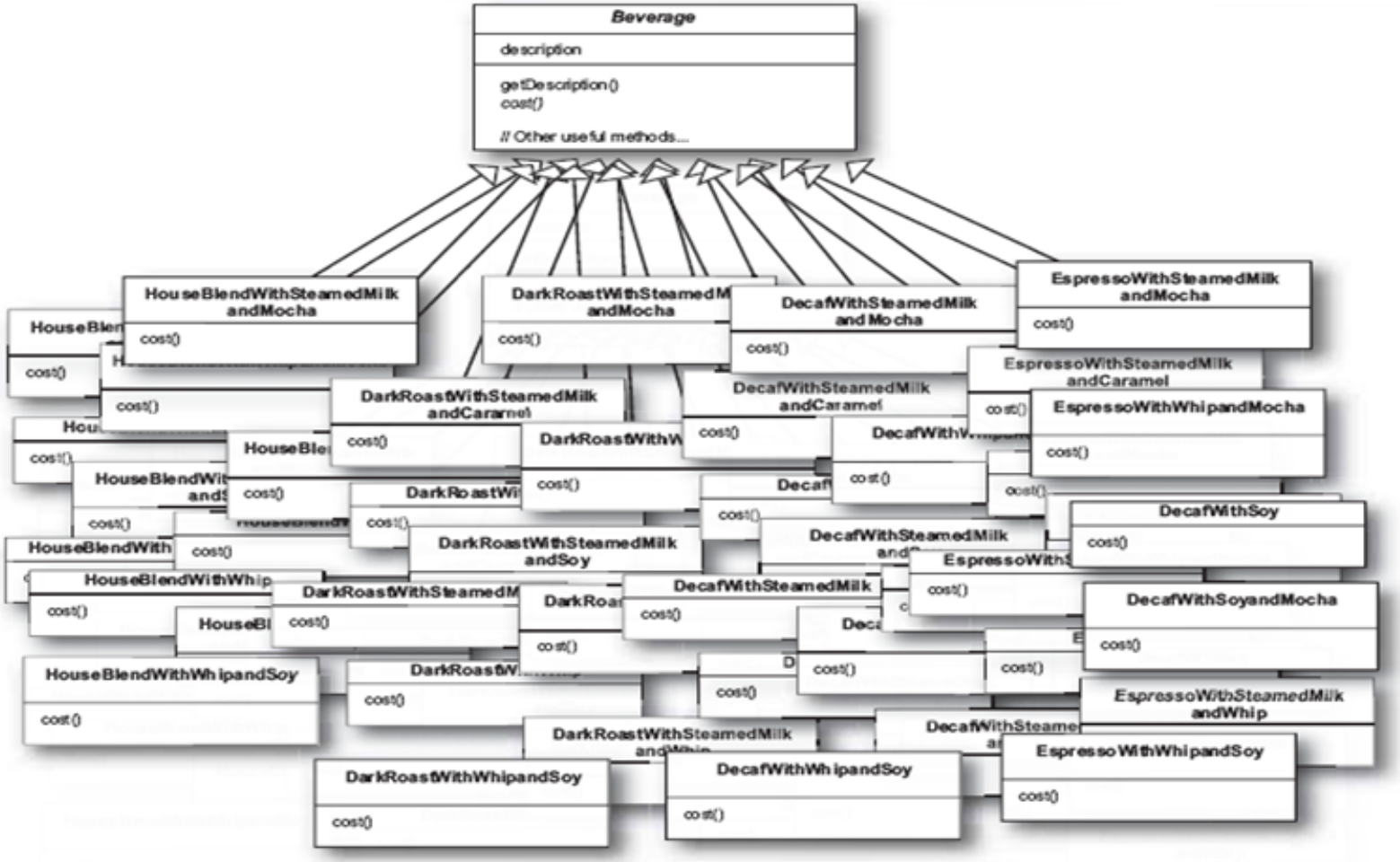
Decorator Design Patterns

New Requirements of Barista

- In addition to the coffee, a customer can also ask for several condiments like steamed milk, sugar, soy etc. Barista will charge a bit for each of these, so they really need to get them built into our order system.

George – Bush Discussion

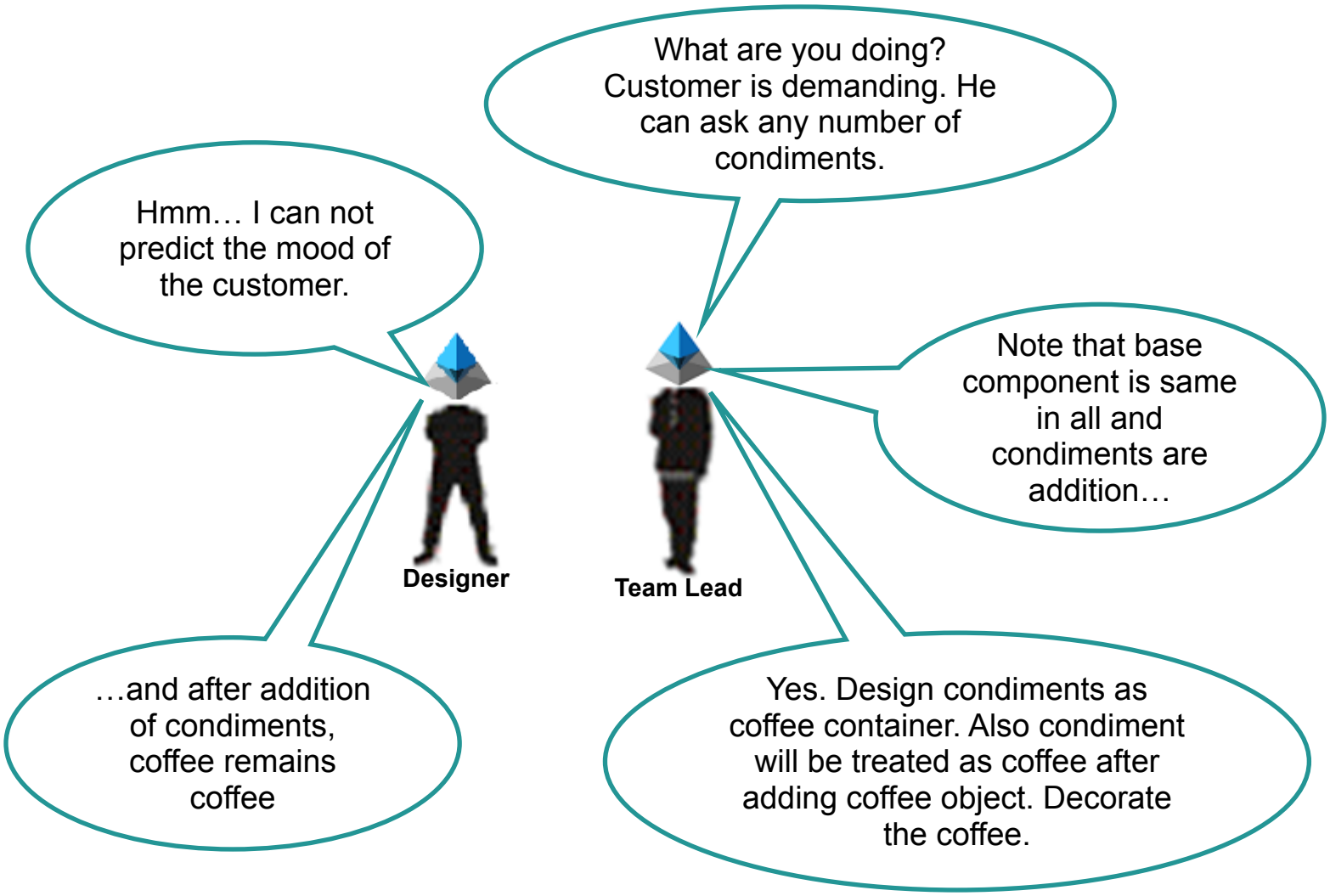




Whoa! Can you say "class explosion?"

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

George – Bush Discussion



Meet with Decorator Pattern

- We will start with a beverage and decorate it with the condiments at runtime. For example, if the customer wants Dark Roast with Mocha (coffee powder) and Whip, then we will:

1. **Take a DarkRoast object**
2. **Decorate it with a Mocha object**
3. **Decorate it with a Whip object**
4. **Call the cost() method and rely on delegation to add on the condiment costs**

Meet with Decorator Pattern

1 We start with our DarkRoast object.

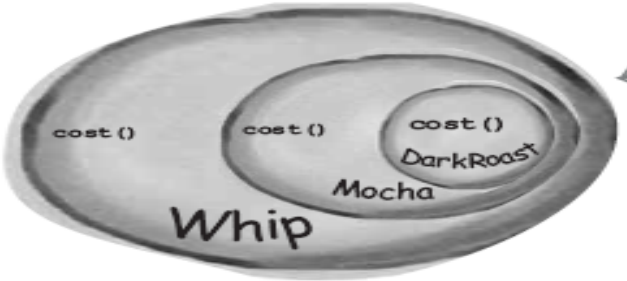
Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

2 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).

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

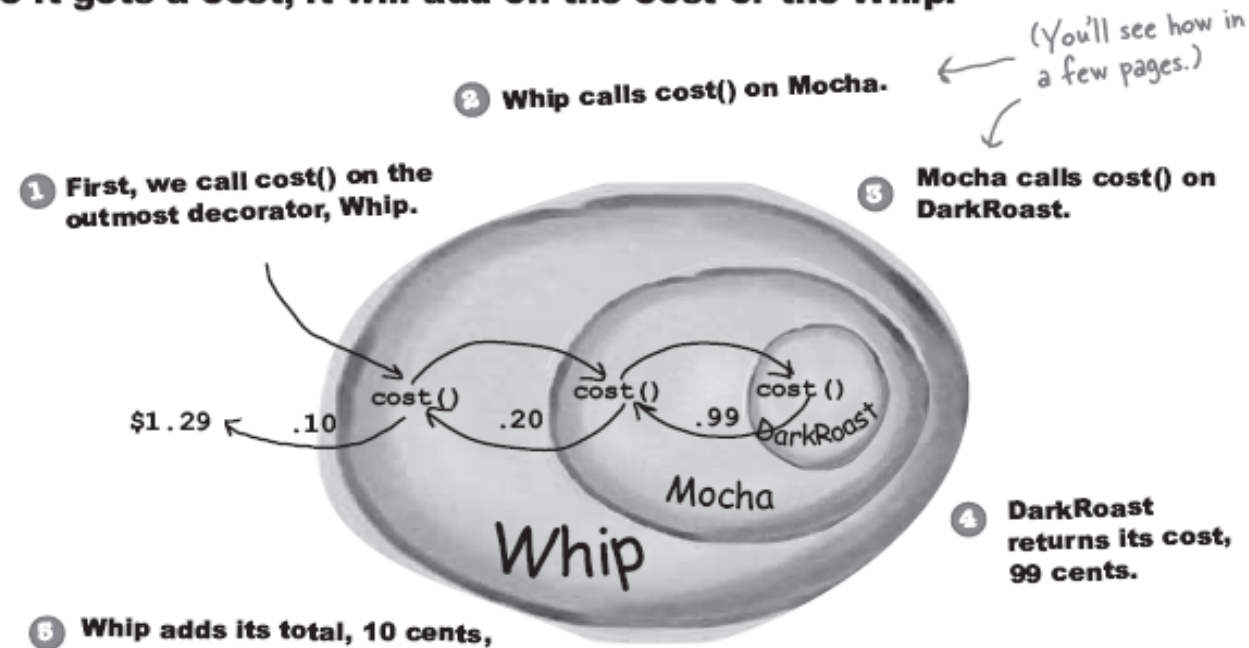


Whip is a decorator, so it mirrors DarkRoast's type includes a cost() method.

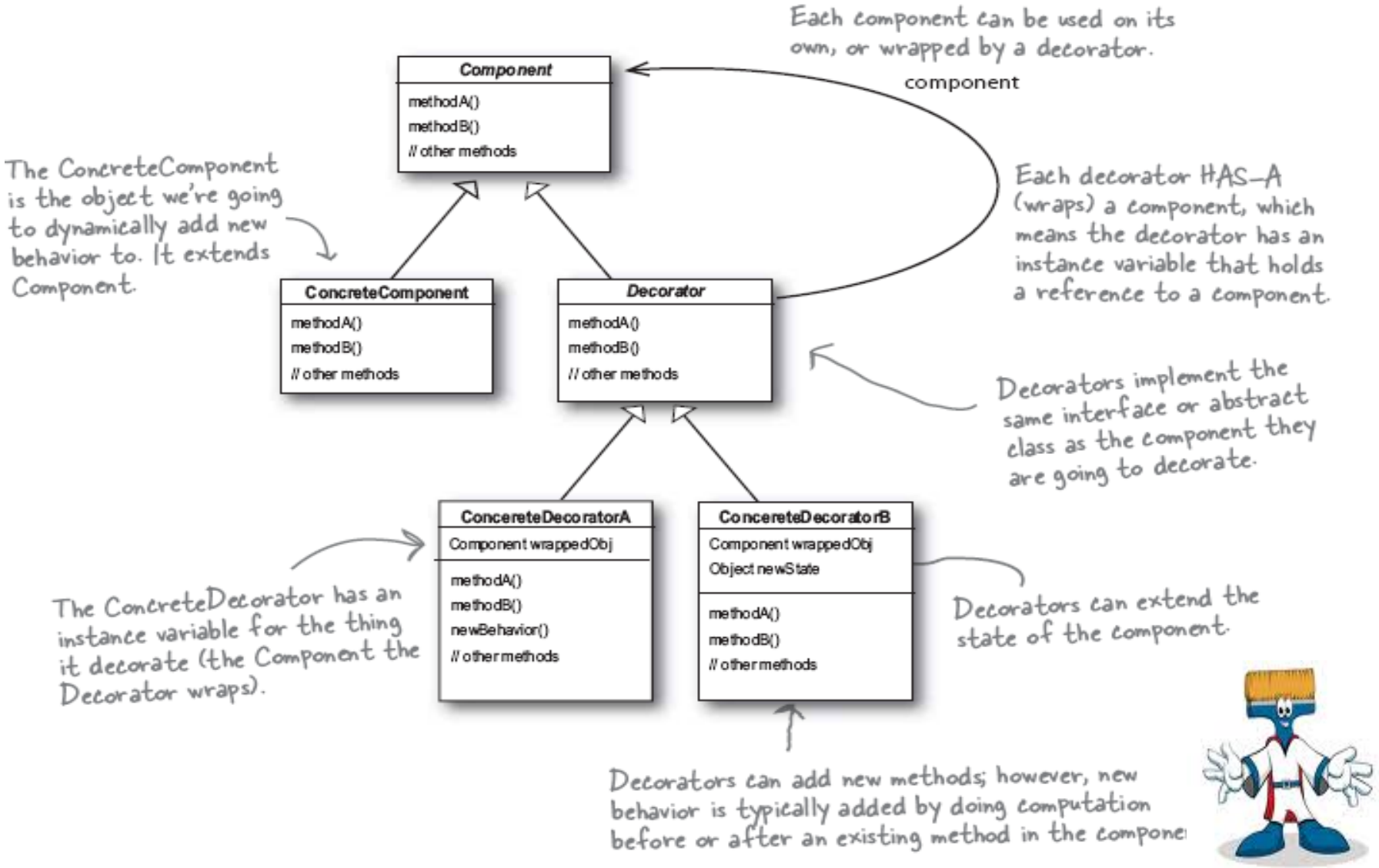
So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

Time to Calculate the Cost

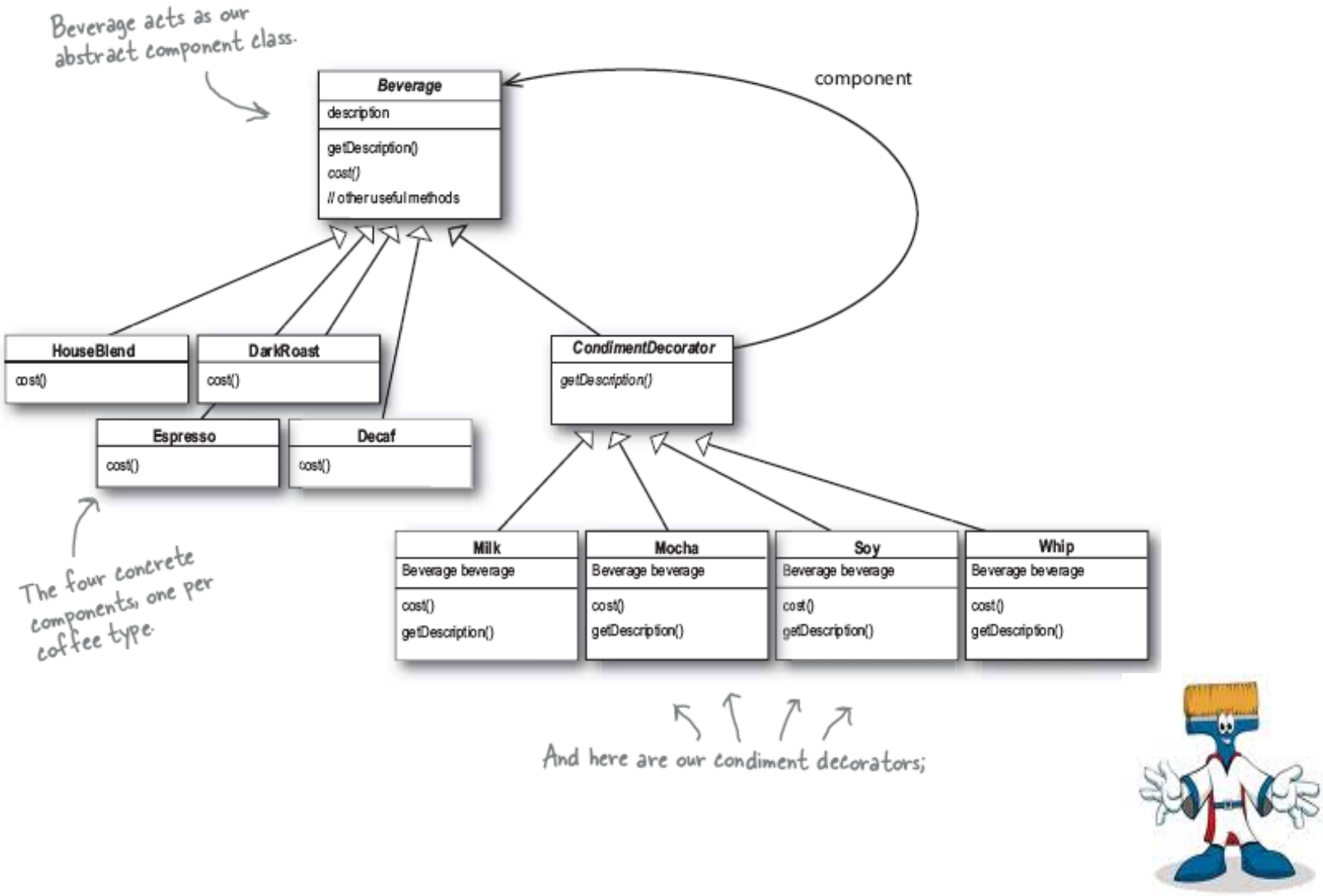
4 Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.



Decorator class diagram



Our Coffee Shop Design With Decorator Pattern



Our Coffee Shop Coding With Decorator Pattern

```
public abstract class CondimentDecorator
        extends Beverage{
    protected Beverage obj = null;
    public CondimentDecorator(Beverage beverage){
        obj = beverage;
    }
}
```

```
public class MilkCondiment extends
        CondimentDecorator{
    public MilkCondiment(Beverage beverage) {
        super(beverage);
    }
    public int cost() {
        return obj.cost() + 3;
    }
    public String getName() {
        return obj.getName() + ", Milk";
    }
    public void make() {
        obj.make();
        System.out.println(" Add Milk");
    }
}
```

```
public class SugerCondiment
    extends CondimentDecorator{
    public SugerCondiment(Beverage beverage) {
        super(beverage);
    }
    public int cost() {
        return obj.cost() + 1;
    }
    public String getName() {
        return obj.getName() + ", Suger";
    }
    public void make() {
        obj.make();
        System.out.println(" Add Suger");
    }
}
```

Our Coffee Shop Client With Decorator Pattern

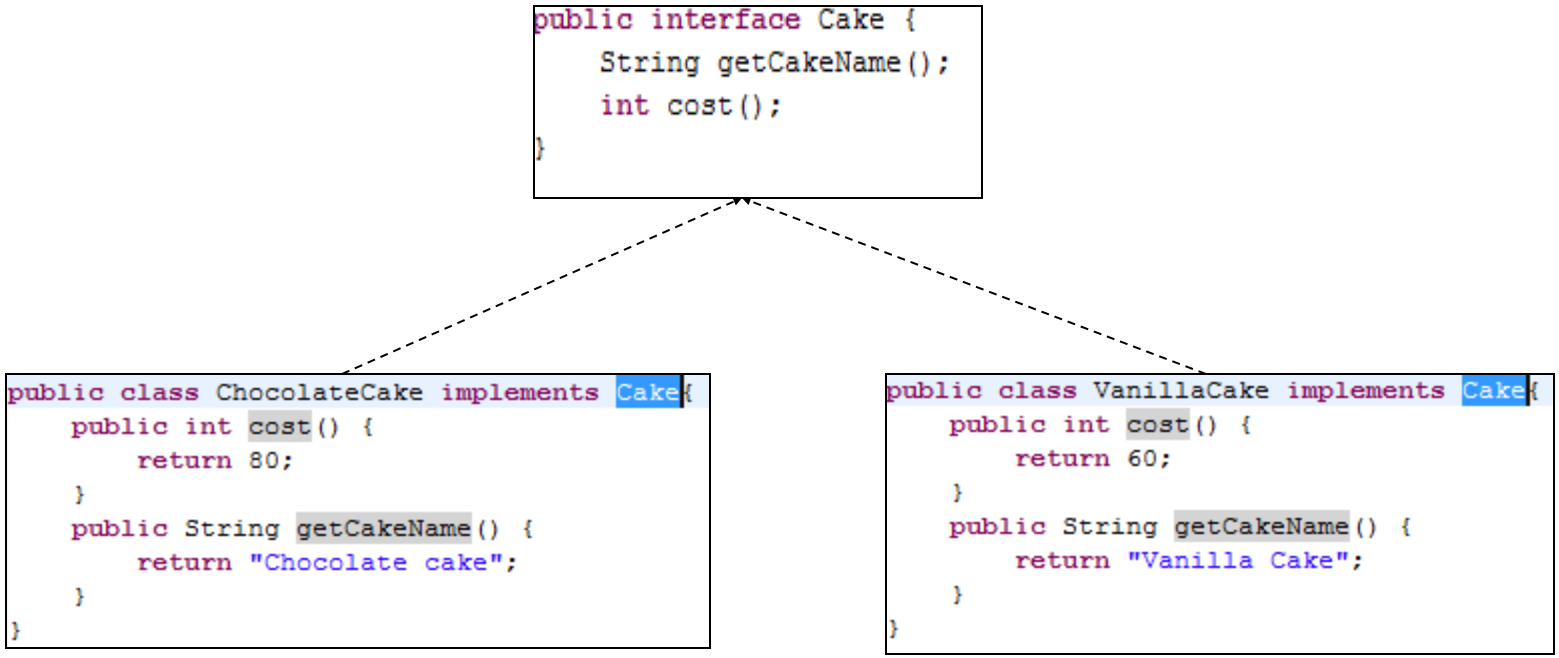
```
public class Test {
    public static void main(String[] args) {
        ClientFactory factory = ClientFactory.getClientFactory("BARISTA");
        if (factory == null) {
            return;
        }
        Beverage cappuccinoCoffee = factory.createCoffee("A");
        cappuccinoCoffee = new MilkCondiment(cappuccinoCoffee);
        cappuccinoCoffee = new SugerCondiment(cappuccinoCoffee);
        cappuccinoCoffee.order();
    }
}
```

Adapter Design Patterns

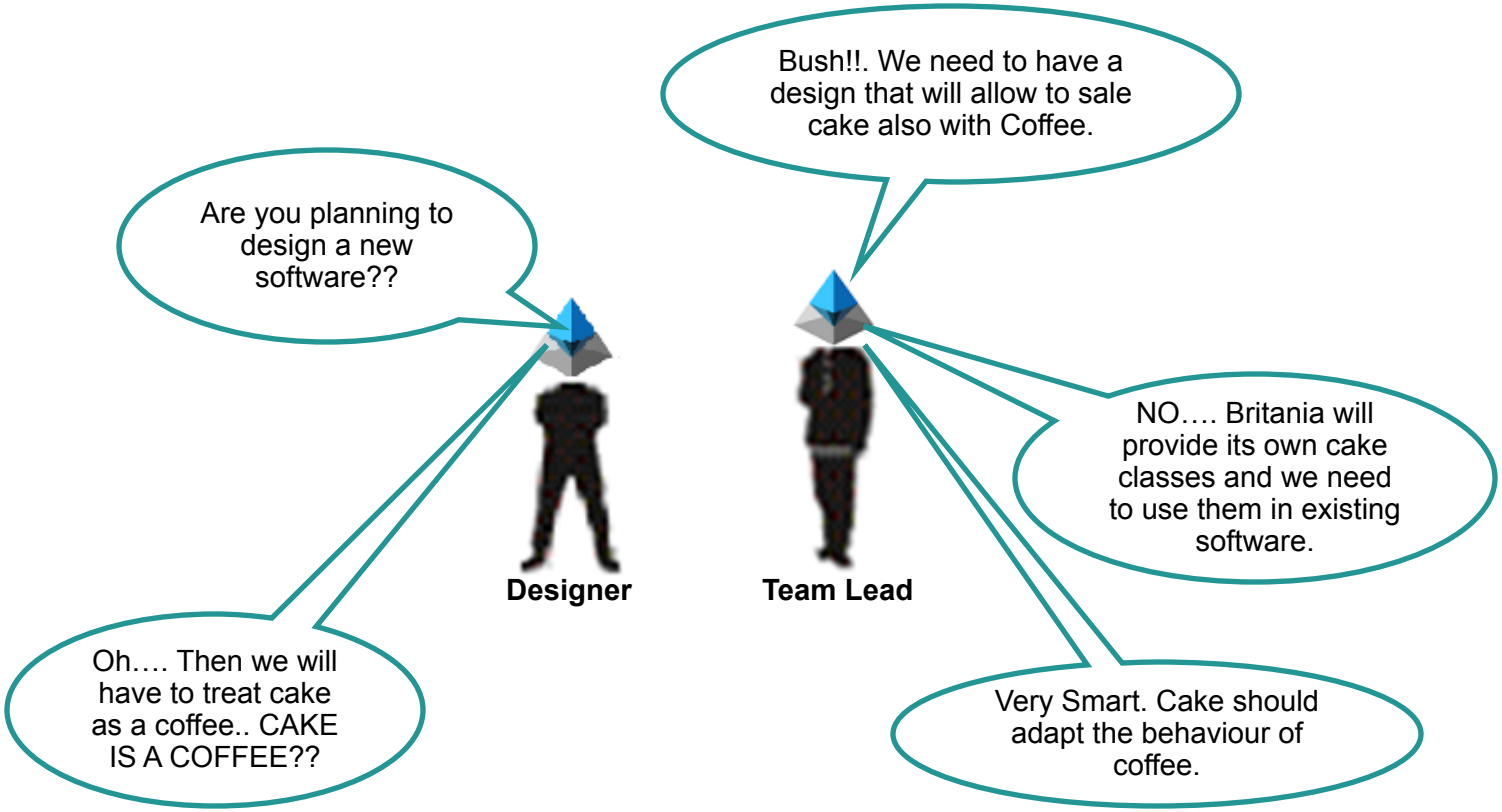
New Product of Barista

- Business of Barista is growing. Barista is planning to sale cake also. They have a deal with Britania to have some cake to sale on coffee counter.

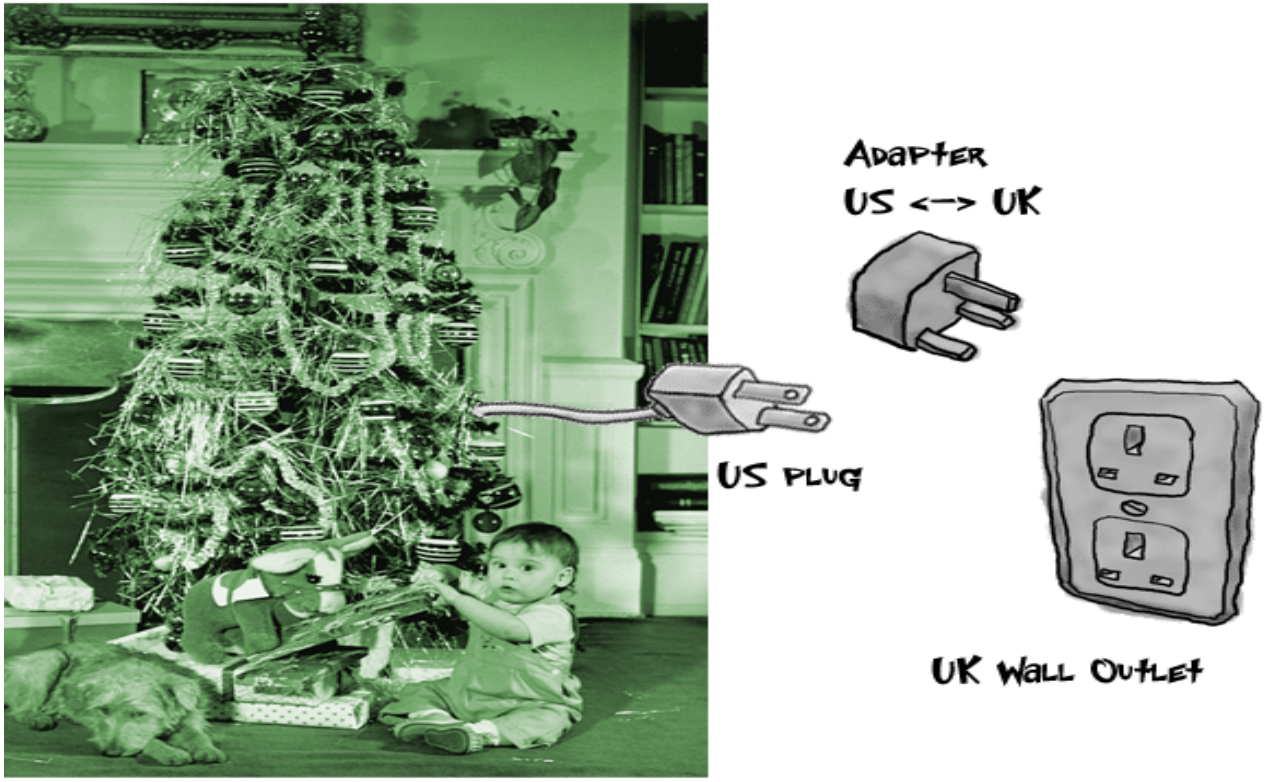
Britania will provide its own classes



George – Bush Discussion

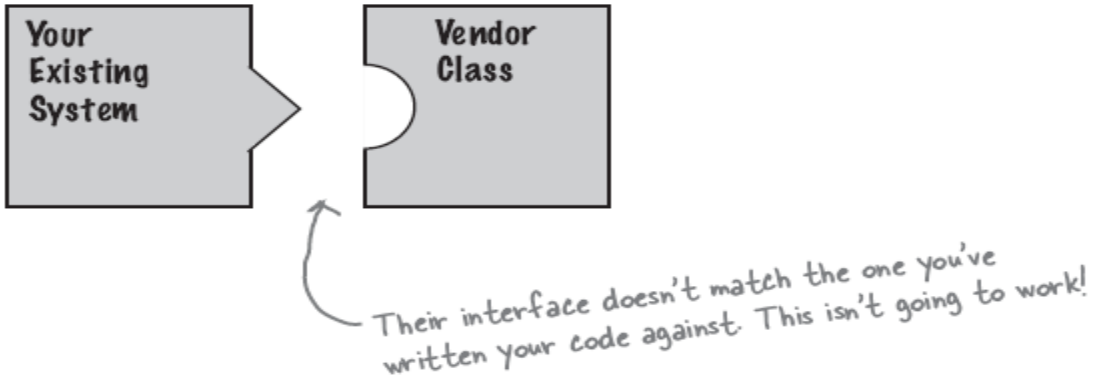


Adapter All Around Us

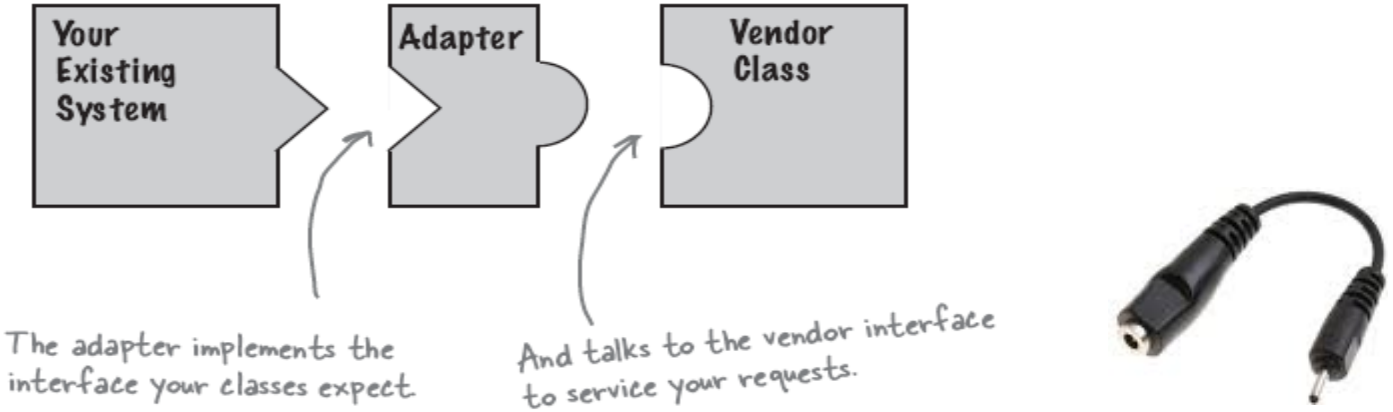


Object Oriented Adapter

Say you've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor:

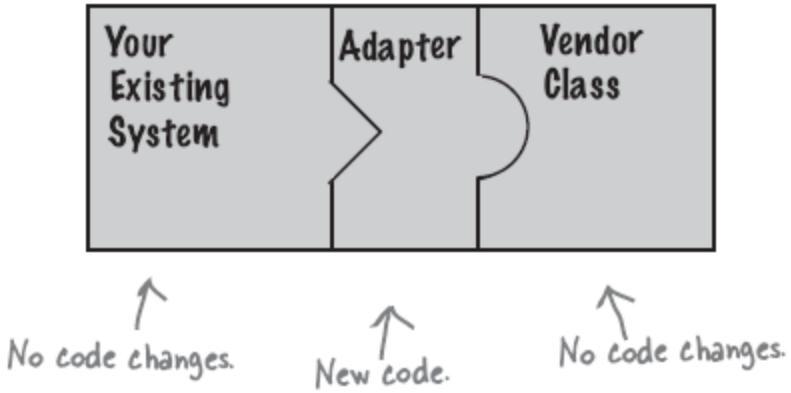


Okay, you don't want to solve the problem by changing your existing code (and you can't change the vendor's code). So what do you do? Well, you can write a class that adapts the new vendor interface into the one you're expecting.



Existing Code of Barista will remain same

The adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.



Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class.

Our Cake Adapter Class

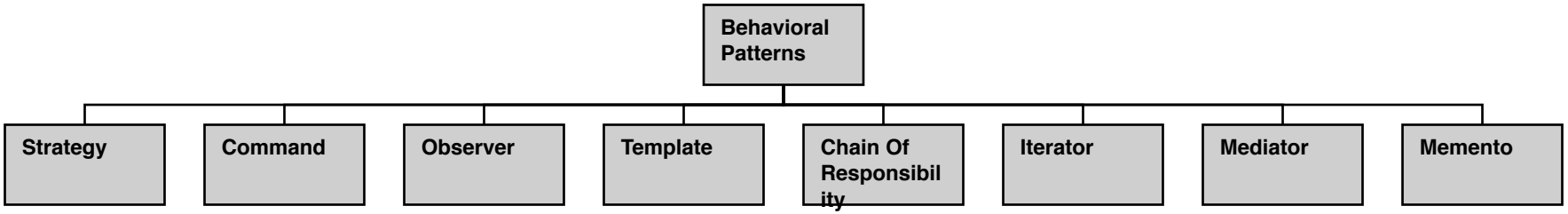
```
public class CakeAdapter extends Beverage{
    Cake cake = null;
    public CakeAdapter(Cake cake){
        this.cake = cake;
    }
    @Override
    public int cost() {
        return cake.cost();
    }
    @Override
    public String getName() {
        return cake.getCakeName();
    }
    @Override
    public void make() {
    }

    @Override
    protected void boilWater() {
    }
}
```

Barista Factory is ready to create object of CakeAdapter

```
public class BaristaFactory extends ClientFactory {
    public Beverage createCoffee(String type) {
        if ("A".equals(type)) {
            return new Cappuccino();
        } else if ("B".equals(type)) {
            return new MochaCoffee();
        } else if ("C".equals(type)) {
            return new Latte();
        } else if ("D".equals(type)) {
            return new Esperano();
        } else if ("E".equals(type)) {
            return new CakeAdapter(new ChocolateCake());
        } else if ("F".equals(type)) {
            return new CakeAdapter(new VanillaCake());
        } else {
            System.out.println("Invalid type");
            return null;
        }
    }
}
```

Behavioral Design Patterns

Structural Patterns Classification



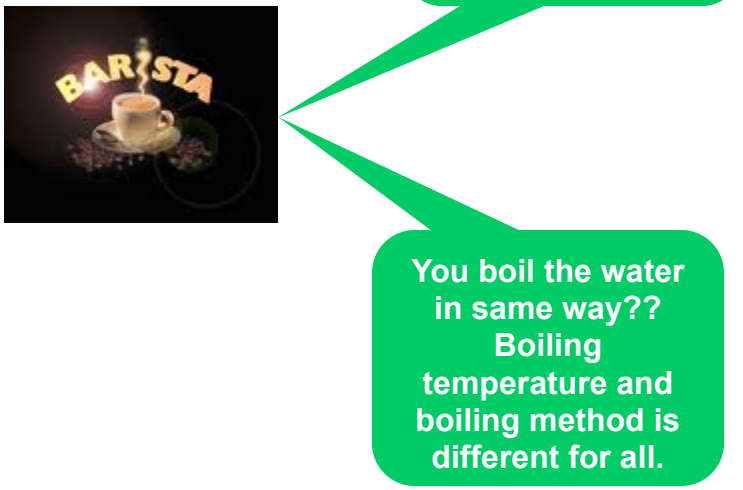
Behavioral Design Patterns

- Behavioral patterns are those, which are concerned with interactions between the objects.
- The interactions between the objects should be such that they are talking to each other and still are loosely coupled.
- The loose coupling is the key to n-tier architectures.
- In this, the implementation and the client should be loosely coupled in order to avoid hard-coding and dependencies.

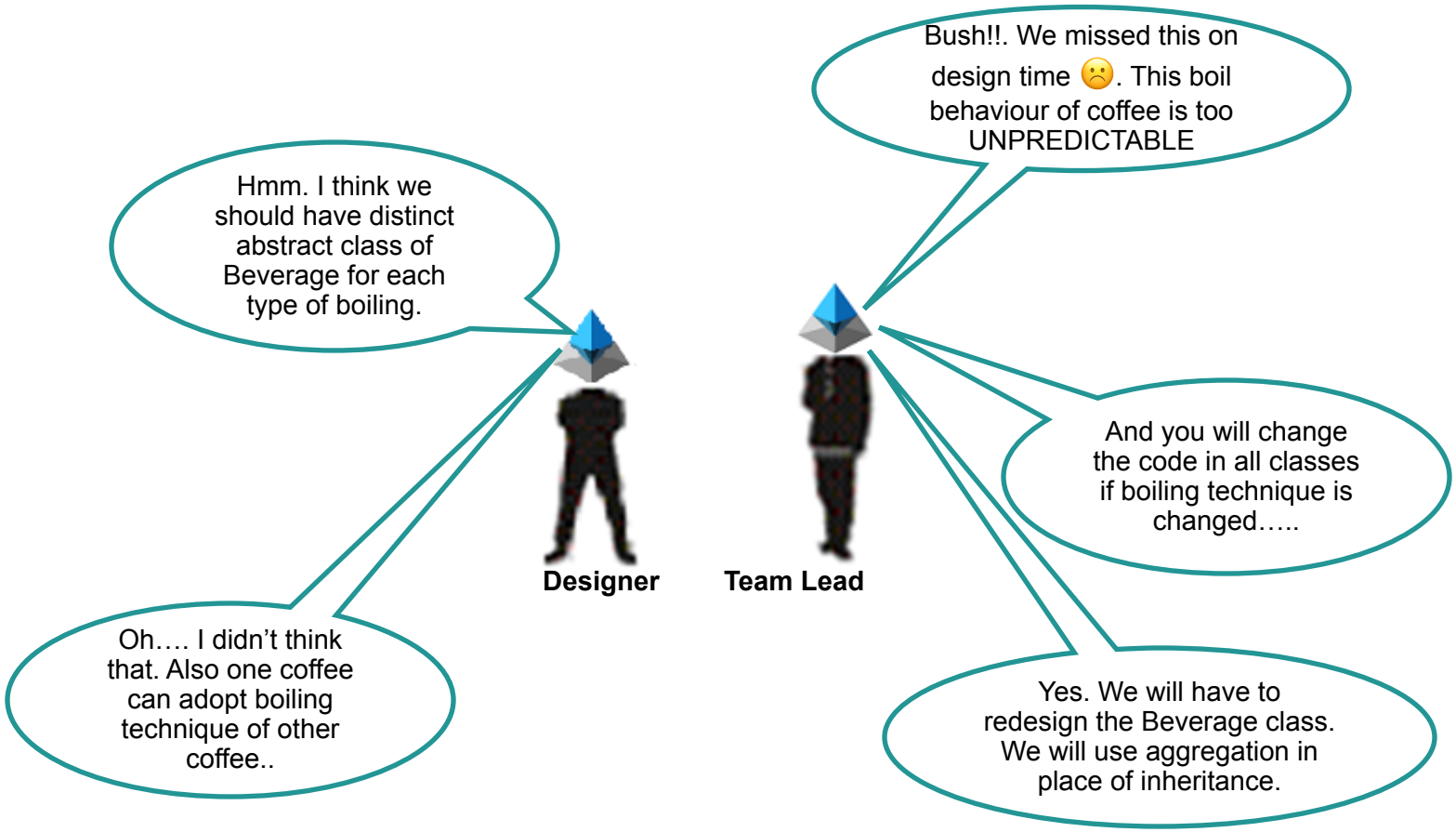
Strategy Design Patterns

New Requirement of Barista

- Way of boiling the water is different for different coffee.
- Also the temperature of water is different for all type of coffee.
- Some products (Britania cake) at Barista do not have boiling water. (Have you taken boiled cake ever ☺)
- Will you boil the water for cold coffee???



George – Bush Discussion



Very unstable behavior???
Strategy Design Pattern..

```
public abstract class Beverage
    extends creational.abstractFactoty.hs.factoryObject.Beverage {
    protected void boilWater() {
        throw new UnsupportedOperationException();
    }
    private BoilingBehaviour boilMethod = null;
    public void order() {
        System.out.println("Customer order " + getName());
        boilMethod.boilWater();
        make();
        serve();
        recieveMoney();
        System.out.println("Sold");
    }
    public void setBoilMethod(BoilingBehaviour boilMethod) {
        this.boilMethod = boilMethod;
    }
}
```

```
public interface BoilingBehaviour {
    void boilWater();
}
```

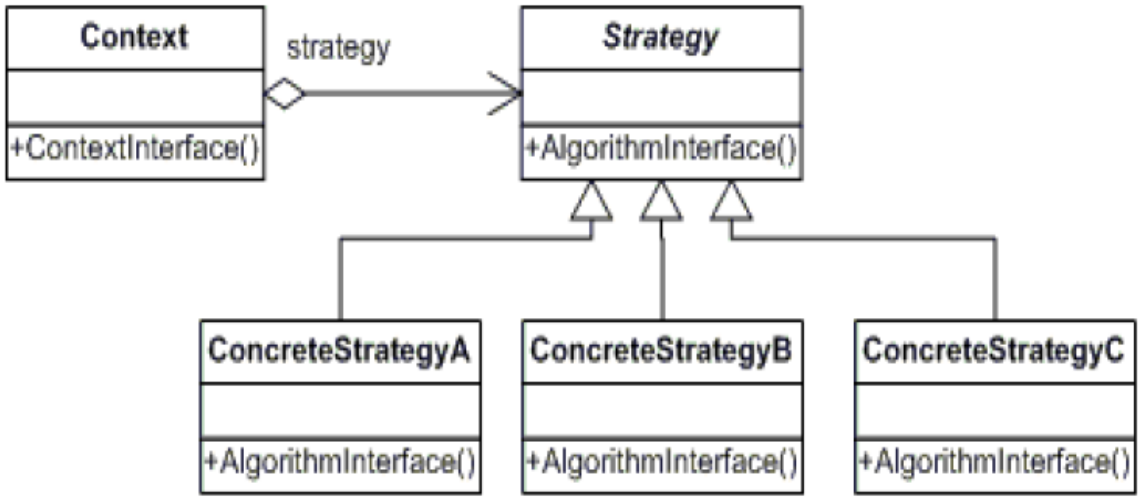
```
public class HardBoil implements BoilingBehaviour {
    public void boilWater() {
        System.out.println("Boiling water hardly at 100*C");
    }
}
```

```
public class NoBoil implements BoilingBehaviour{
    public void boilWater() {
        System.out.println("Why Boiling Water");
    }
}
```

Strategy Design Pattern Features

- Identify the aspects of your application that vary and separate them from what stays the same.
- Aggregation is sometimes better than association.
- The strategy pattern is useful for situations where it is necessary to dynamically swap the algorithms used in an application

Class Diagram of Strategy Design Pattern



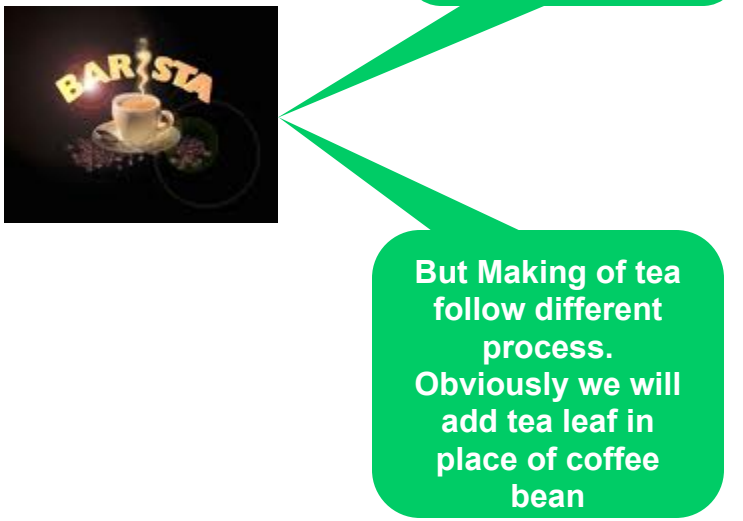
Participants

- **Strategy (SortStrategy)**
  - Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy (QuickSort, ShellSort, MergeSort)**
  - Implements the algorithm using the Strategy interface
- **Context (SortedList)**
  - Is configured with a ConcreteStrategy object.
  - Maintains a reference to a Strategy object.
  - May define an interface that lets Strategy access its data.

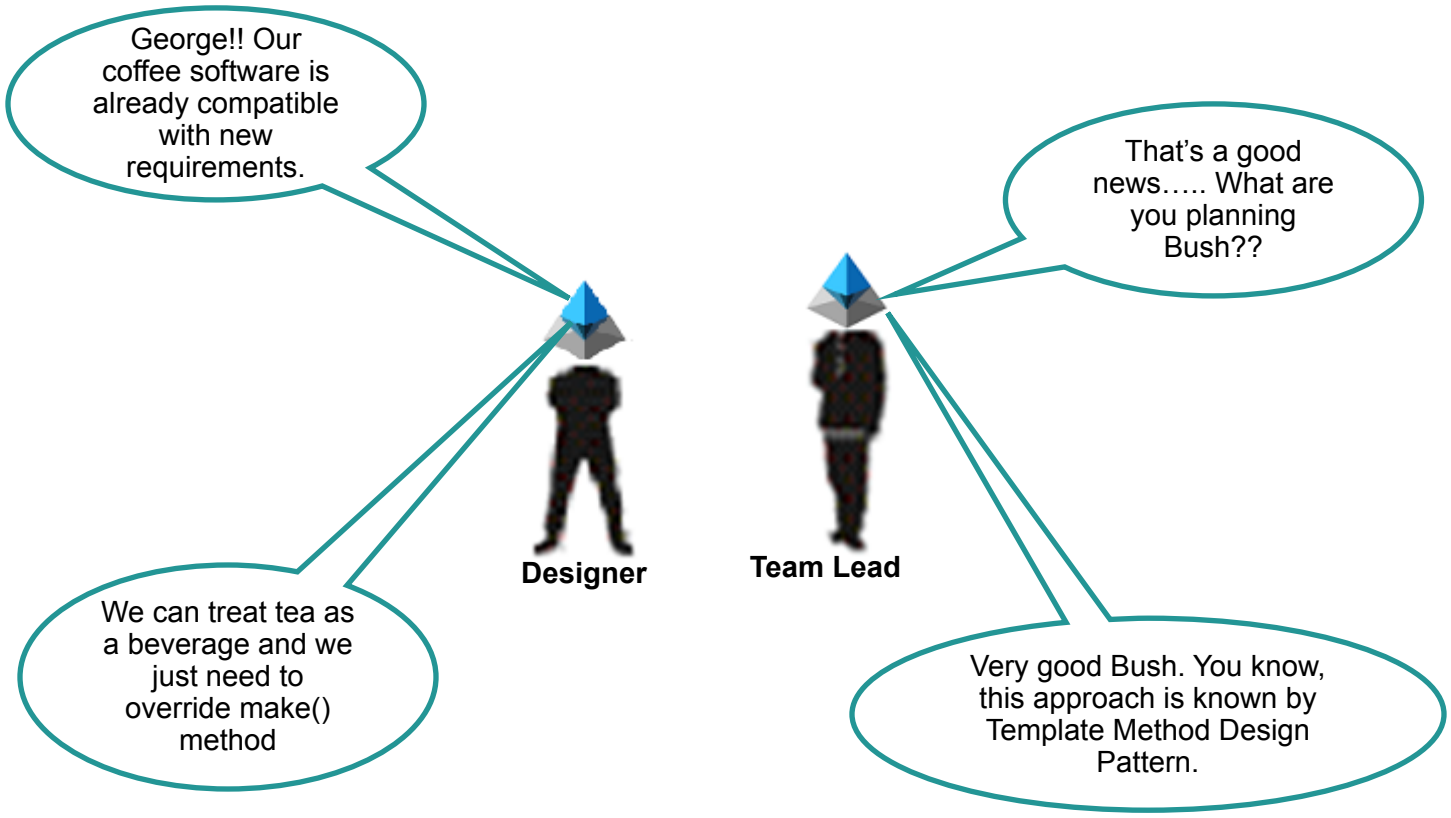
Template Method Design Patterns

Tea also can be a product of Barista

- Barista also want to take the order of Tea.
- All the steps to order a tea are similar to coffee. The only difference is that we use tea leaf in place of coffee bean.



George – Bush Discussion



Barista Tea class

```
public class Tea extends Beverage{
    public Tea(){
        super.setBoilMethod(new HardBoil());
    }
    public int cost() {
        return 40;
    }
    public String getName() {
        return "Barista Tea";
    }
    public void make() {
        System.out.println("Put tea leaf");
        System.out.println("Make Barista Tea");
    }
}
```

Feel of Template Method

1. The first problem we have is that Coffee uses brewCoffeeGrinds() and addSugarAndMilk() methods while Tea uses steepTeaBag() and addLemon() methods.

Coffee

```
void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}
```

Tea

```
void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

Let's think through this: steeping and brewing aren't so different; they're pretty analogous. So let's make a new method name, say, brew(), and we'll use the same name whether we're brewing coffee or steeping tea.

Likewise, adding sugar and milk is pretty much the same as adding a lemon: both are adding condiments to the beverage. Let's also make up a new method name, addCondiments(), to handle this. So, our new prepareRecipe() method will look like this:

```
void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
}
```

Feel of Template Method

2 Now we have a new prepareRecipe() method, but we need to fit it into the code. To do this we are going to start with the CaffeineBeverage superclass:

CaffeineBeverage is abstract, just like in the class design.

```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

Now, the same prepareRecipe() method will be used to make both Tea and Coffee. prepareRecipe() is declared final because we don't want our subclasses to be able to override this method and change the recipe! We've generalized steps 2 and 4 to brew() the beverage and addCondiments().

Because Coffee and Tea handle these methods in different ways, they're going to have to be declared as abstract. Let the subclasses worry about that stuff!

Remember, we moved these into the CaffeineBeverage class (back in our class diagram).

Feel of Template Method

3 Finally we need to deal with the Coffee and Tea classes. They now rely on CaffeineBeverage to handle the recipe, so they just need to handle brewing and condiments:

As in our design, Tea and Coffee now extend CaffeineBeverage.

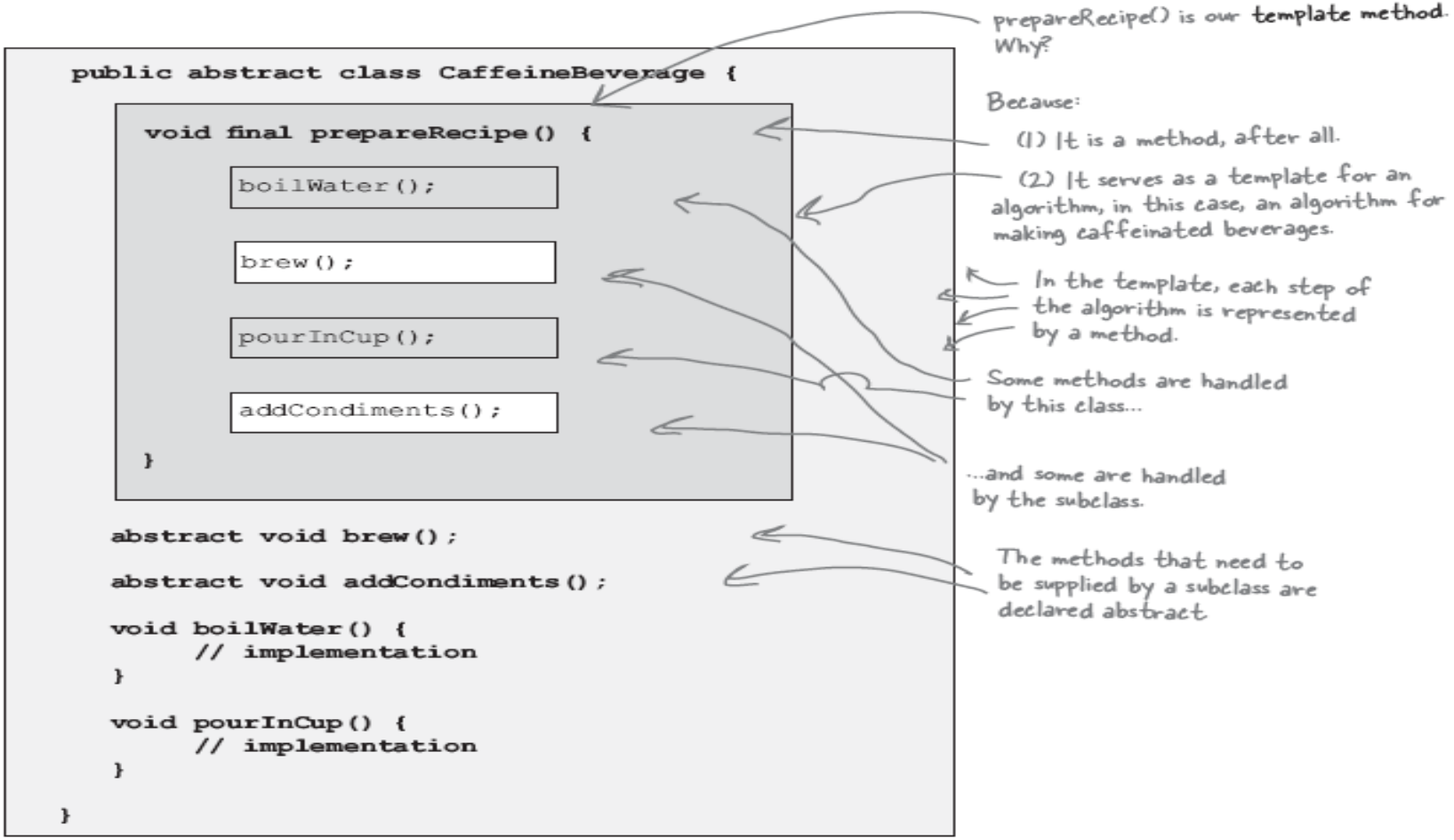
```
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

Tea needs to define brew() and addCondiments() – the two abstract methods from Beverage.

Same for Coffee, except Coffee deals with coffee, and sugar and milk instead of tea bags and lemon.

```
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Template Method Design Pattern



The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.

Template Method Pattern Defined

- The template method pattern defined the skeleton of an algorithm in a method, deferring some steps to subclass.
- Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure.