ACCOLITE UNIVERSITY '21

Design Patterns

Agenda

- Why Design Pattern
- Objective of Design Pattern
- Design Principle
- Design Pattern Classification
- Creational Design Pattern
- Structural Design Pattern
- Behavioral Design Pattern

© Copyright 2021 : Accolite : All Rights Reserved

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:

- Improve understandability of design.
- 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.

© Copyright 2021 : Accolite : All Rights Reserved

Objective of Design Patterns

- Always think the future aspects of design and requirement changing.
- If requirements are changed, minimum number of java files should be changed (We can live with changes in xml, properties and other files as we will prefer not to build the application for a change).
- Always think about the minimum duplicity.
- Always think that your design should be easily understood to other.
- Reusability.
- Integrity.
- Low Coupling

© Copyright 2021 : Accolite : All Rights Reserved

OOAD Design Principle

© Copyright 2021 : Accolite : All Rights Reserved

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.

© Copyright 2021 : Accolite : All Rights Reserved

Design Principle Classification

- **Design Principle [SOLID]**
  - **Single Responsibility**
  - **Open Close**
  - **Liskov's Substitution**
  - **Interface Segregation**
  - **Dependency Inversion**

© Copyright 2021 : Accolite : All Rights Reserved

SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

© Copyright 2021 : Accolite : All Rights Reserved

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.

© Copyright 2021 : Accolite : All Rights Reserved

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.

© Copyright 2021 : Accolite : All Rights Reserved

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.

© Copyright 2021 : Accolite : All Rights Reserved

LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

© Copyright 2021 : Accolite : All Rights Reserved

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.

© Copyright 2021 : Accolite : All Rights Reserved

DEPENDENCY INVERSION

Would you solder a lamp directly to the electrical wiring in a wall?

© Copyright 2021 : Accolite : All Rights Reserved

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.

© Copyright 2021 : Accolite : All Rights Reserved

GoF OOAD Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

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 realize relationships between entities. Examples of Structural Patterns include: Adapter pattern: 'adapts' one interface for a class into one that a client expects.

Behavioural: **Behavioral** 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.

© Copyright 2021 : Accolite : All Rights Reserved

Creational Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

Creational Patterns Classification

- **Creational Patterns**
  - **Singleton**
  - **Factory Method**
  - **Abstract Factory Method**
  - **Builder**
  - **Prototype**
  - **Object Pool**

© Copyright 2021 : Accolite : All Rights Reserved

Singleton Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

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.

© Copyright 2021 : Accolite : All Rights Reserved

Lazy Loaded Singleton Class

```java
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;
    }
}
```

© Copyright 2021 : Accolite : All Rights Reserved

Early Loaded Singleton Class

```java
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;
    }
}
```

© Copyright 2021 : Accolite : All Rights Reserved

Multiple Singletons

- If we are loading our class by different class loaders.
- If our class is serialized.
- If instantiate the class with reflection framework.
- Parent class is cloneable and clone method is not being overridden properly.
- Click below link for more information
  http://java.sun.com/developer/technicalArticles/Programming/singletons/

© Copyright 2021 : Accolite : All Rights Reserved

Factory Design Patterns

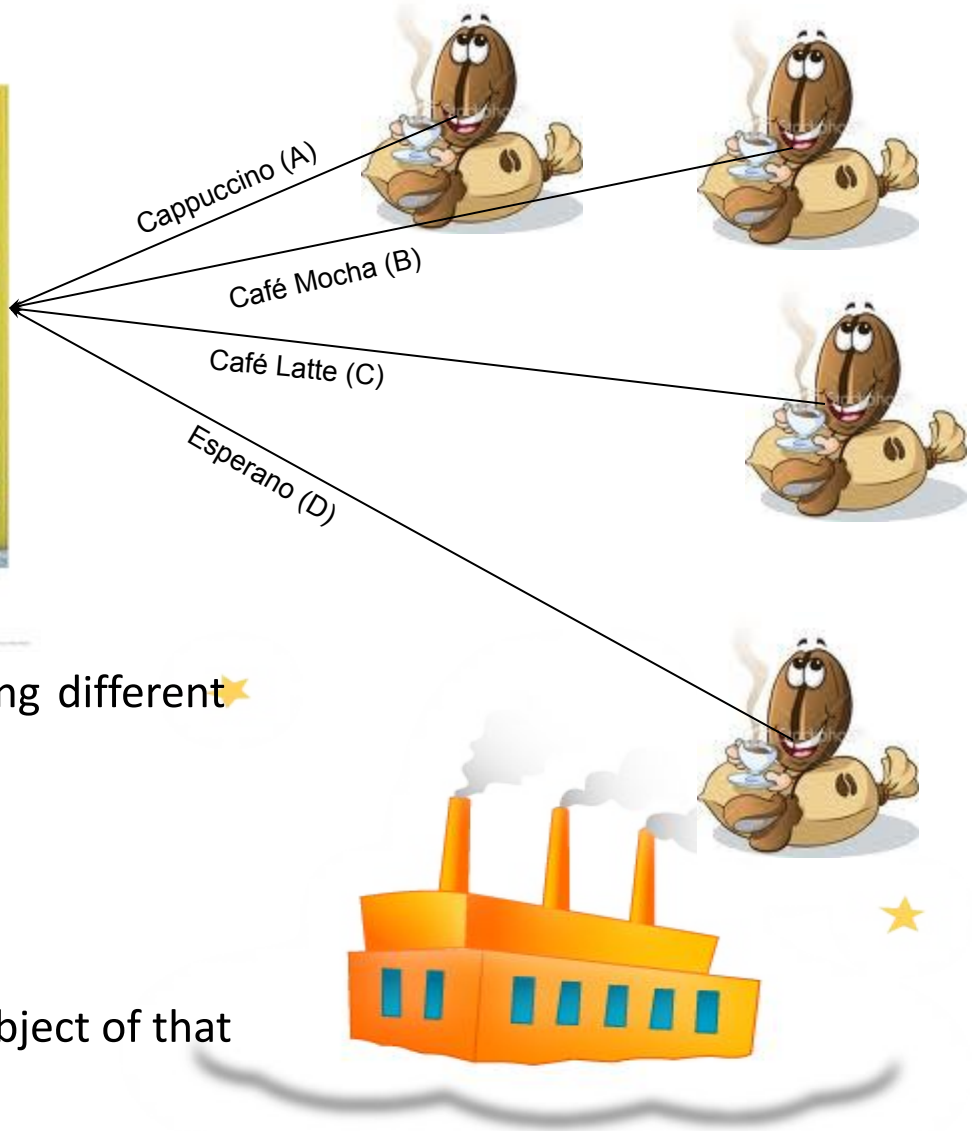
© Copyright 2021 : Accolite : All Rights Reserved

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".

© Copyright 2021 : Accolite : All Rights Reserved

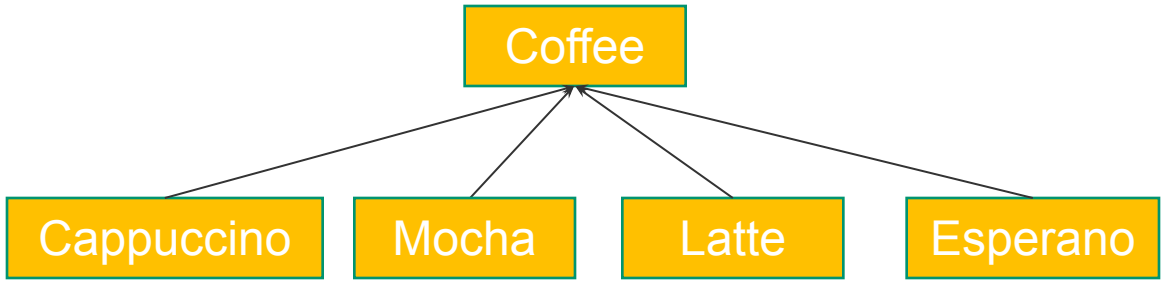
Coffee Shop Problem



- 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.

© Copyright 2021 : Accolite : All Rights Reserved

Coffee Shop Initial Design




```java
package hs.design.factory.cofee;

public abstract class Coffee {
    public void serve() {
        System.out.println("Serve coffee");
    }

    public abstract void makeCofee();

    public void deliver() {
        serve();
        makeCofee();
    }
}
```

```java
public class BadCoffeeTest {
    public static void main(String[] args) {
        if (args == null || args.length < 1) {
            System.out.println("Pass appropriate parameter");
            return;
        }
        String option = args[0];
        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();
        } else {
            System.out.println("Invalid option");
        }
        coffee.deliver();
    }
}
```

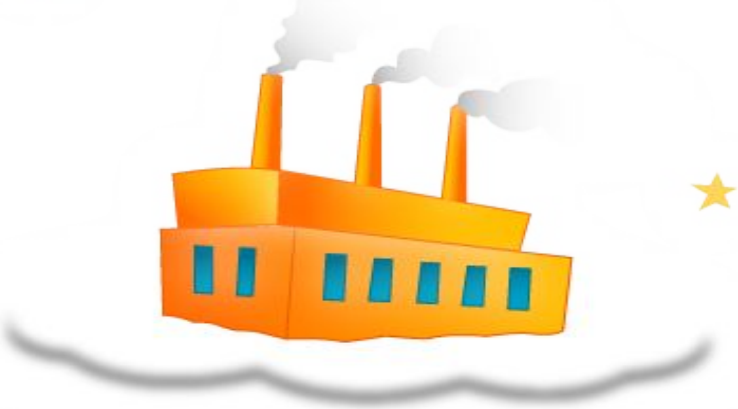
© Copyright 2021 : Accolite : All Rights Reserved

Problem with Initial Design

- Creation of objects can takes place in so many java files.

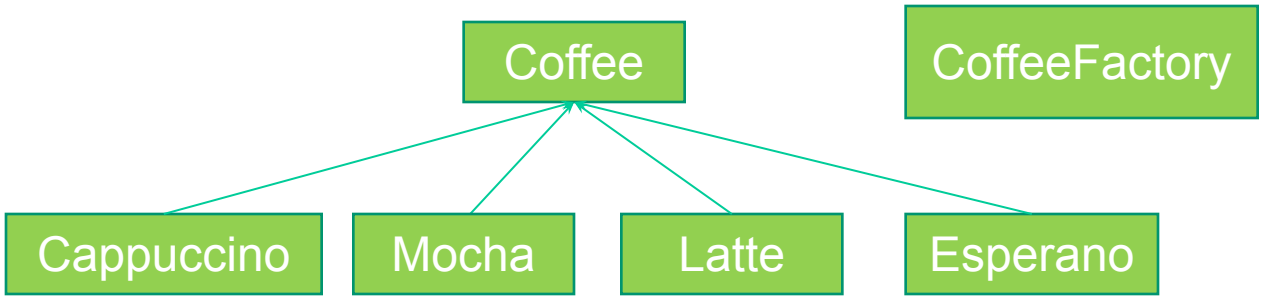
```
if ("A".equals(option)) {
    coffee = new Cappuccino();
} else if ("B".equals(option)) {
    coffee = new MochaCoffee();
} else if ("C".equals(option)) {
    coffee = new Latte();
} else if ("D".equals(option)) {
    coffee = new Esperano();
} else {
    System.out.println("Invalid option");
}
```

- If one more coffee will be added to the coffee shop (always think future aspects), we will have to change all the java files.



© Copyright 2021 : Accolite : All Rights Reserved

Design with Factory Pattern

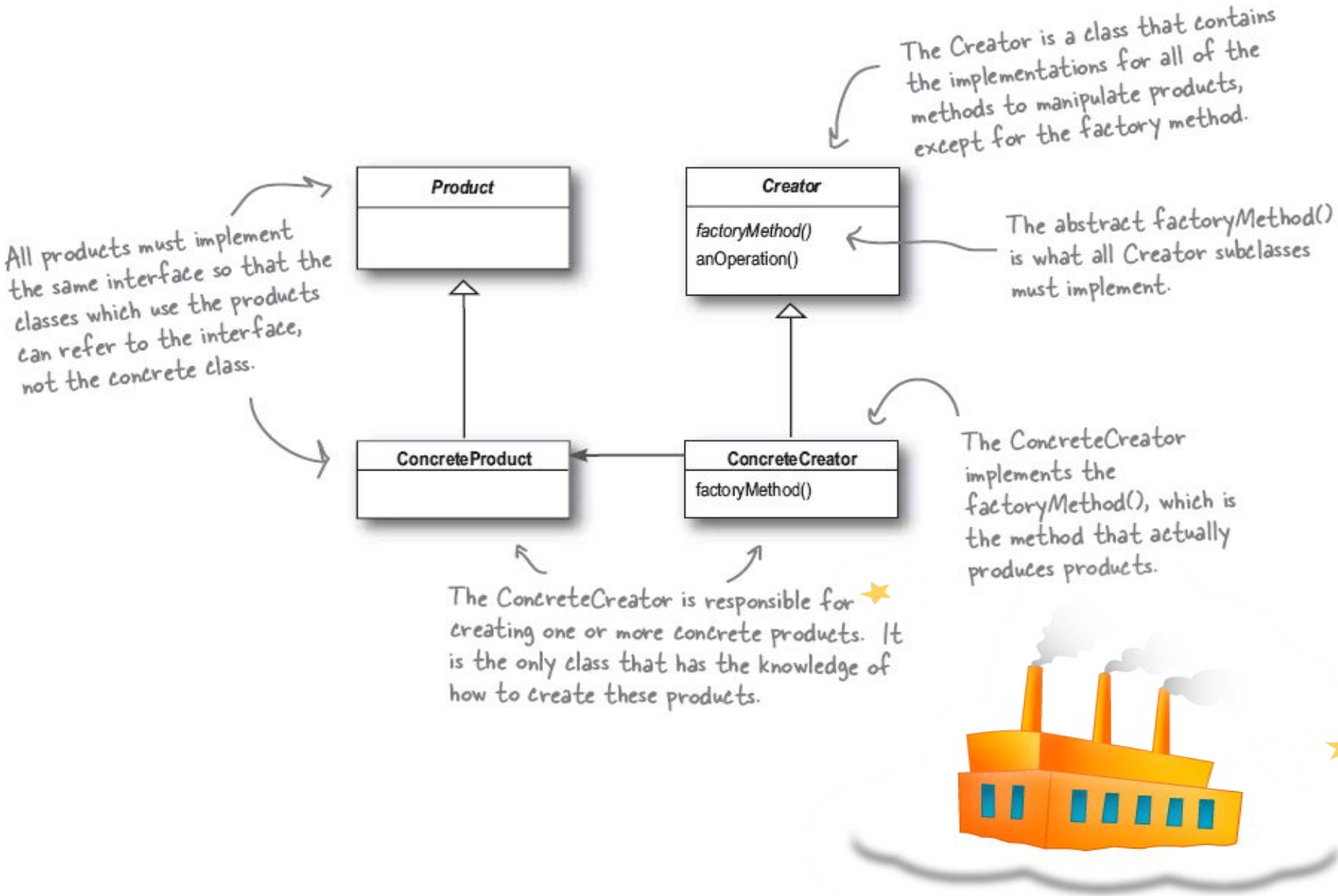



```java
public class CoffeeFactory {

    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;
    }
}
```

```java
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();
    }
}
```

© Copyright 2021 : Accolite : All Rights Reserved

Factory Pattern Class Diagram



© Copyright 2021 : Accolite : All Rights Reserved

Builder Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

Coffee Shop Design With Builder Pattern

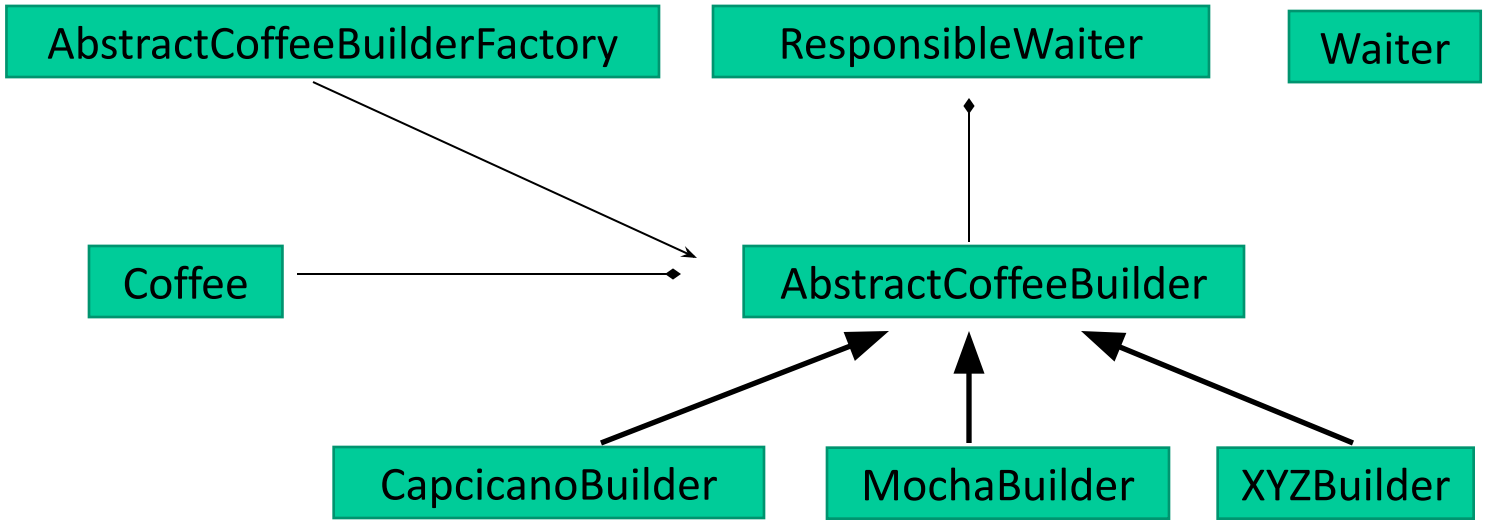
- Builder is used to separate the construction of complex objects from their representation so the construction process can be used to create different representations.
- The construction logic is isolated from the actual steps used to create the complex object and, therefore, the construction process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

The main advantages of Builder Pattern are as follows:

- It provides clear separation between the construction and representation of an object.
- It provides better control over construction process.
- It supports to change the internal representation of objects.

© Copyright 2021 : Accolite : All Rights Reserved

Coffee Shop Design With Builder Pattern



© Copyright 2021 : Accolite : All Rights Reserved

Coffee Shop Code with Builder Pattern

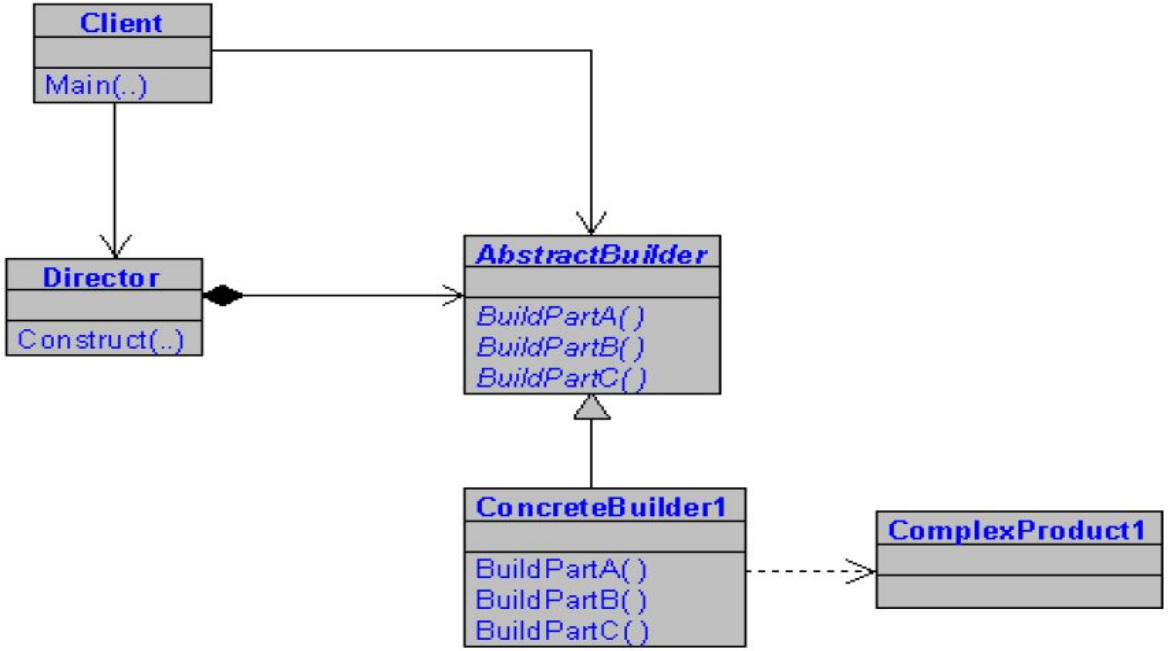
```java
package builder.hs.builder;

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() {
        coffee.setType("Mocha");
    }
}
```

© Copyright 2021 : Accolite : All Rights Reserved

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

© Copyright 2021 : Accolite : All Rights Reserved

Structural Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

Structural Patterns Classification

- Structural Patterns
  - Adapter
  - Composite
  - Decorator
  - Bridge
  - Flyweight
  - Proxy

© Copyright 2021 : Accolite : All Rights Reserved

Composite Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

New Requirement of Barista

- In the next client discussion, Barista ask some changes in there software.

- ALL
  - SMALL
    - COLD
      - Latte
      - Esperrano
      - Mocha
  - LARGE
    - Cappuccino

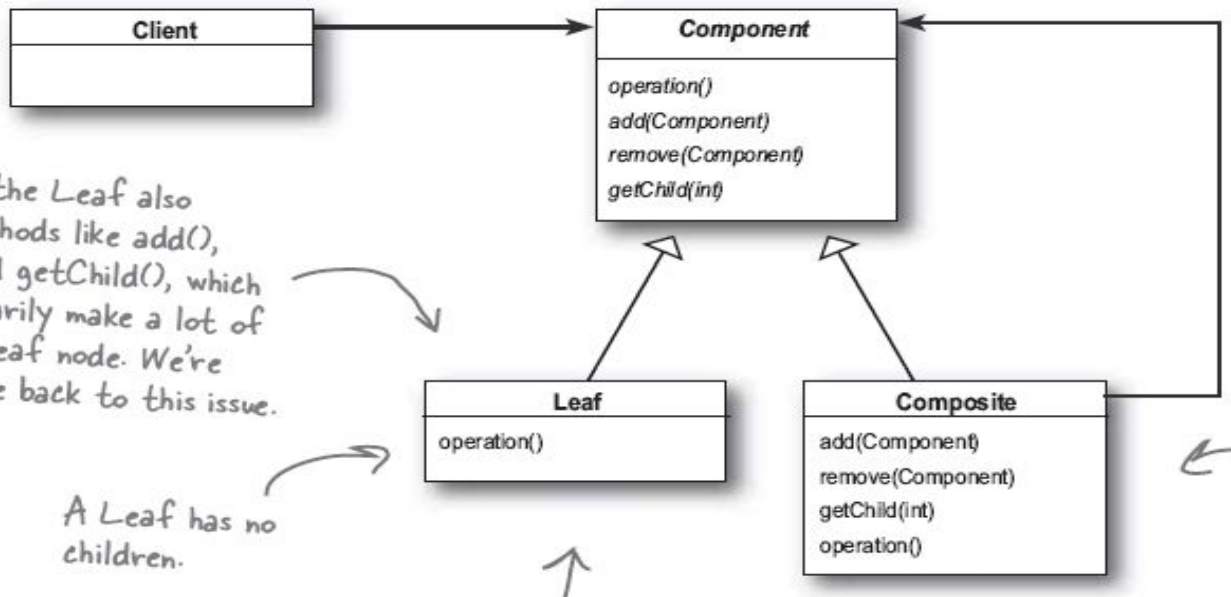
© Copyright 2014: Accolite : All Rights Reserved

Composite Design Pattern

The Client uses the Component interface to manipulate the objects in the composition.

The Component defines an interface for all objects in the composition: both the composite and the leaf nodes.

The Component may implement a default behavior for add(), remove(), getChild() and its operations.



Note that the Leaf also inherits methods like add(), remove() and getChild(), which don't necessarily make a lot of sense for a leaf node. We're going to come back to this issue.

A Leaf has no children.

A Leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports.

The Composite's role is to define behavior of the components having children and to store child components.

The Composite also implements the Leaf-related operations. Note that some of these may not make sense on a Composite, so in that case an exception might be generated.

© Copyright 2021 : Accolite : All Rights Reserved

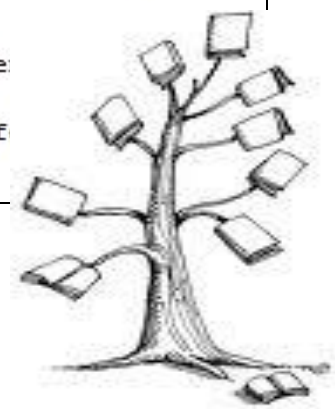
Class Diagram With Composite Pattern

```java
public interface Component {
    ArrayList<? extends Component> getChildren();
    boolean addComponent(Component c);
    boolean removeComponent(Component c);
}
```

```java
public interface CoffeeComponent extends Component{
    String getDetails();
}
```

```java
public class CoffeeCategory implements CoffeeComponent{
    private String name = null;
    ArrayList<CoffeeComponent> coffeeCategoryComponents
        = new ArrayList<CoffeeComponent>();
    public CoffeeCategory(String name){
        this.name = name;
    }
```

```java
public abstract class CoffeeLeaf
        implements CoffeeComponent {
    protected Beverage coffee = null;
    public CoffeeLeaf(String type, String client) {
        ClientFactory factory = ClientFactory.
        getClientFactory(client);
        if (factory == null) {
            throw new IllegalArgume
        }
        coffee = factory.createCoff
    }
```



© Copyright 2021 : Accolite : All Rights Reserved

Decorator Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

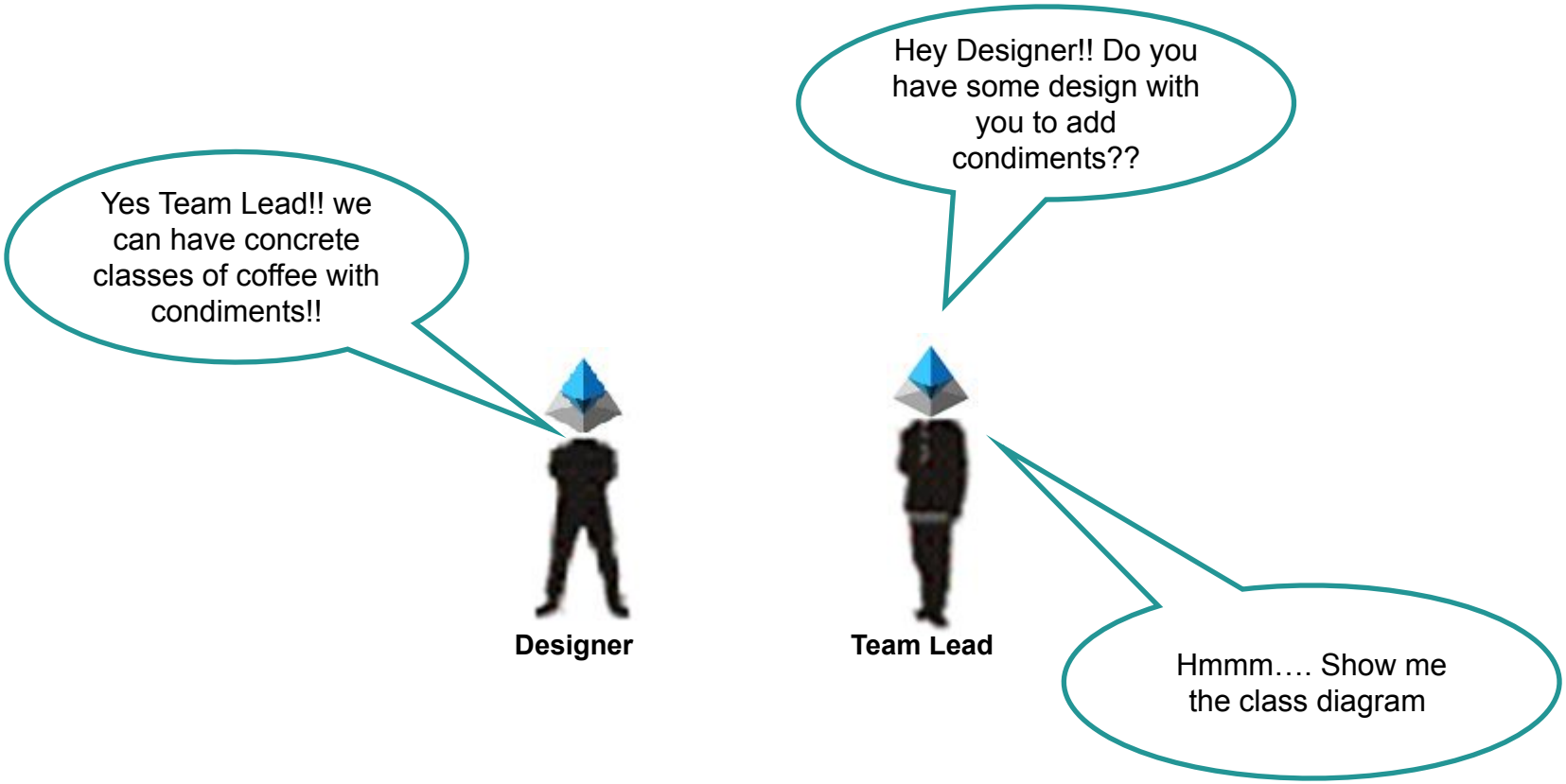
New Requirements of Barista

- 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.

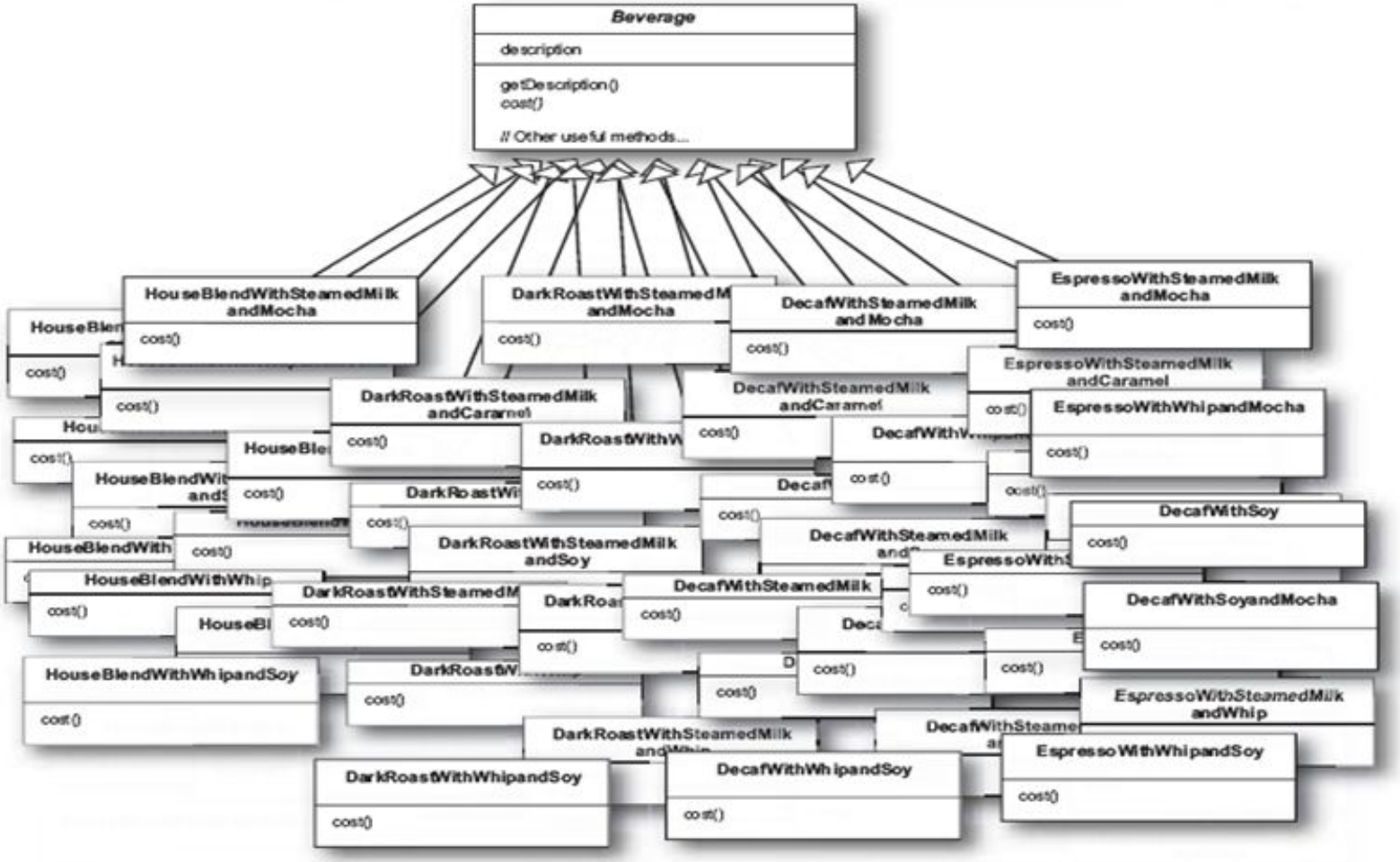
We have some condiments to be added to coffee. We will charge for condiment.

© Copyright 2021 : Accolite : All Rights Reserved

Team Lead– Designer Discussion



© Copyright 2021 : Accolite : All Rights Reserved



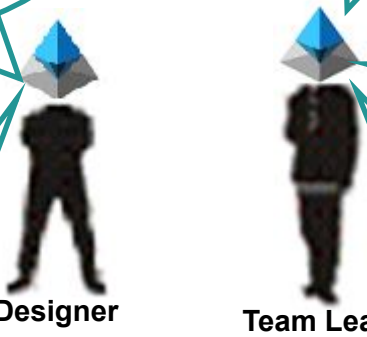
Each cost method computes the cost of the coffee along with the other condiments in the order.

© Copyright 2021 : Accolite : All Rights Reserved

Team Lead– Designer Discussion

What are you doing? Customer is demanding. He can ask any number of condiments.

Hmm… I can not predict the mood of the customer.



Note that base component is same in all and condiments are addition…

…and after addition of condiments, coffee remains coffee

Yes. Design condiments as coffee container. Also condiment will be treated as coffee after adding coffee object. Decorate the coffee.

© Copyright 2021 : Accolite : All Rights Reserved

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**

© Copyright 2021 : Accolite : All Rights Reserved

Meet with Decorator Pattern

1 We start with our DarkRoast object.

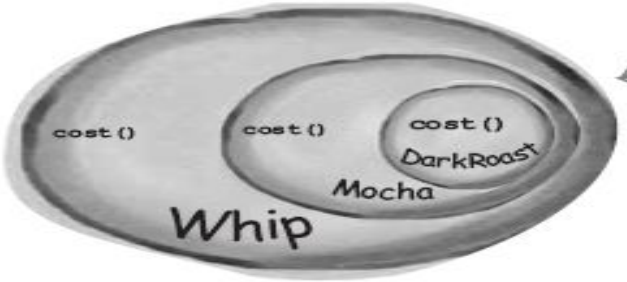
Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

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.

© Copyright 2021 : Accolite : All Rights Reserved

Time to Calculate the Cost

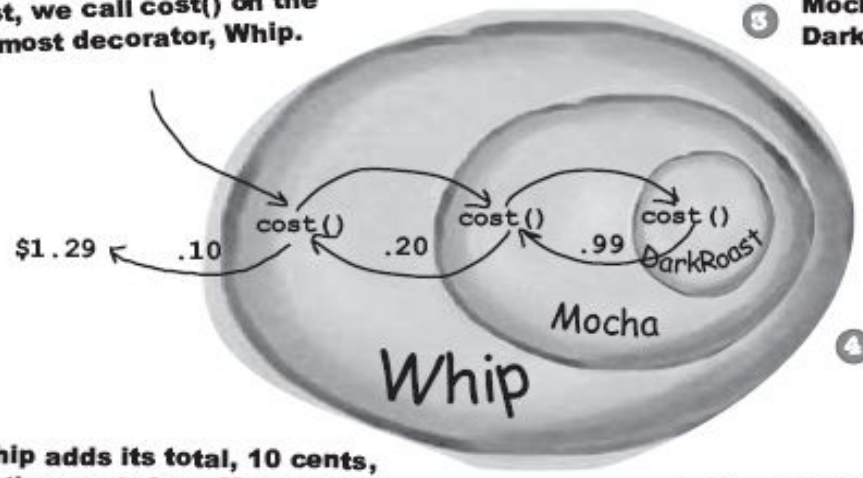
4 Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.

(You'll see how in a few pages.)

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.

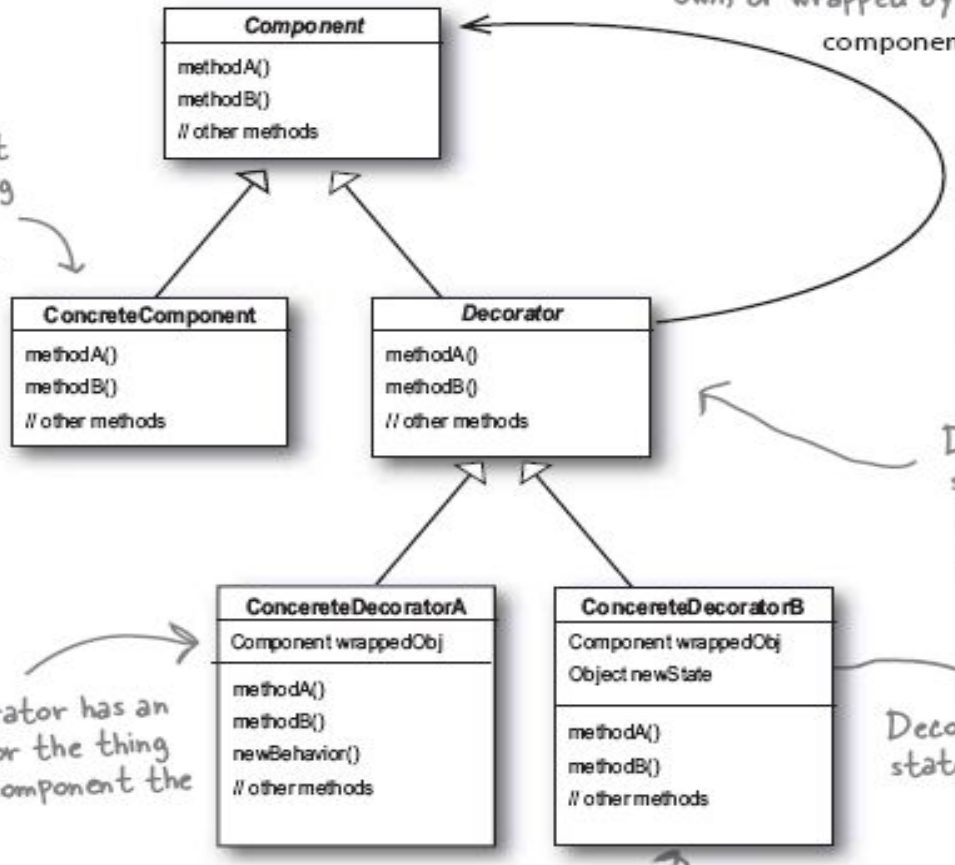
6 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

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

© Copyright 2021 : Accolite : All Rights Reserved

Decorator class diagram

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.

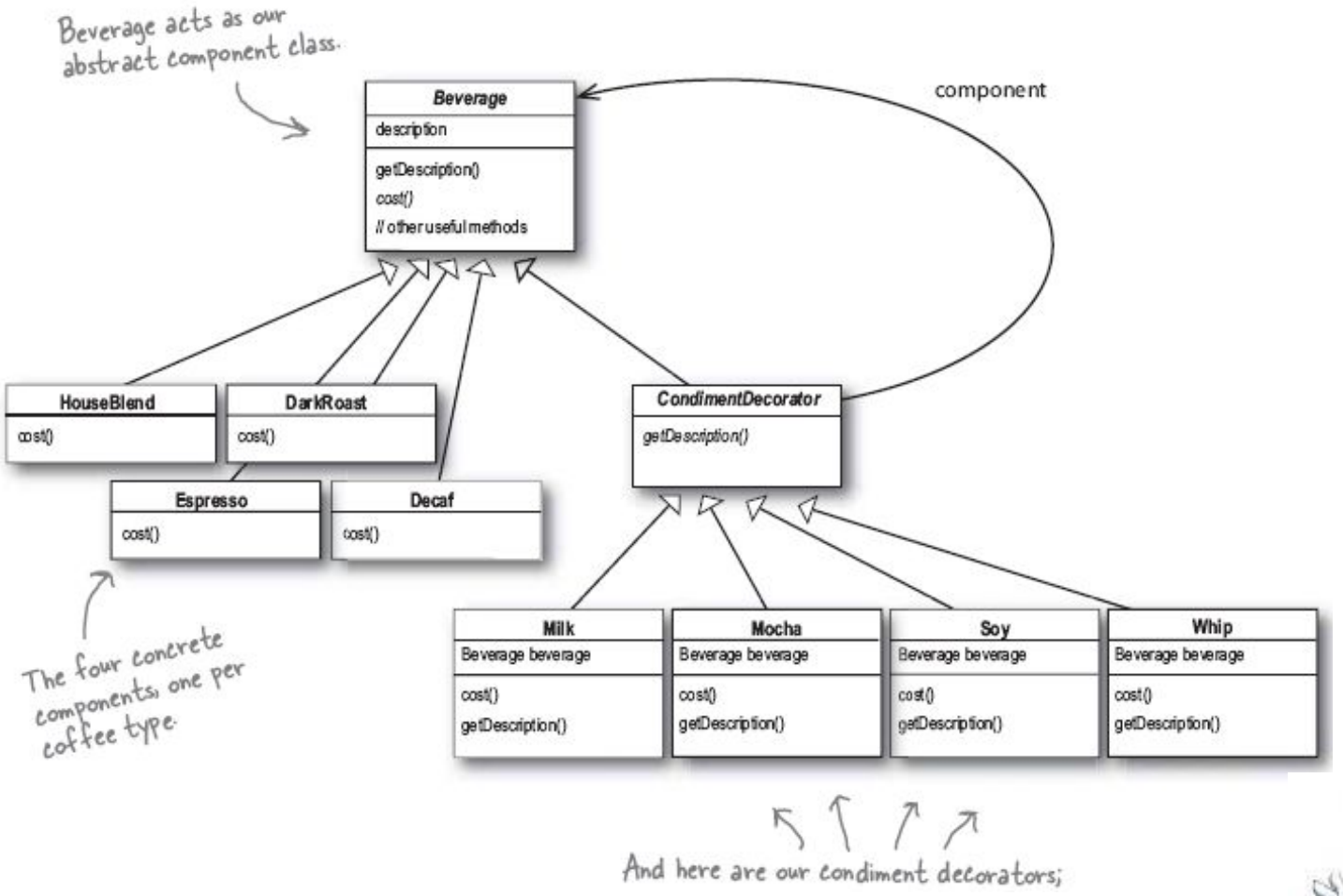
The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).

Decorators can extend the state of the component.

Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the compone

© Copyright 2021 : Accolite : All Rights Reserved

Our Coffee Shop Design With Decorator Pattern



© Copyright 2021 : Accolite : All Rights Reserved

Our Coffee Shop Coding With Decorator Pattern

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

```java
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");
    }
}
```

```java
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");
    }
}
```

© Copyright 2021 : Accolite : All Rights Reserved

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();
    }
}
```

© Copyright 2021 : Accolite : All Rights Reserved

Adapter Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

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.

We Also need to sale Britania cake. They will deliver cake and we will sale.

We do not want to use Britania Software. Can we do it with Coffee Software??

© Copyright 2021 : Accolite : All Rights Reserved

Britania will provide its own classes

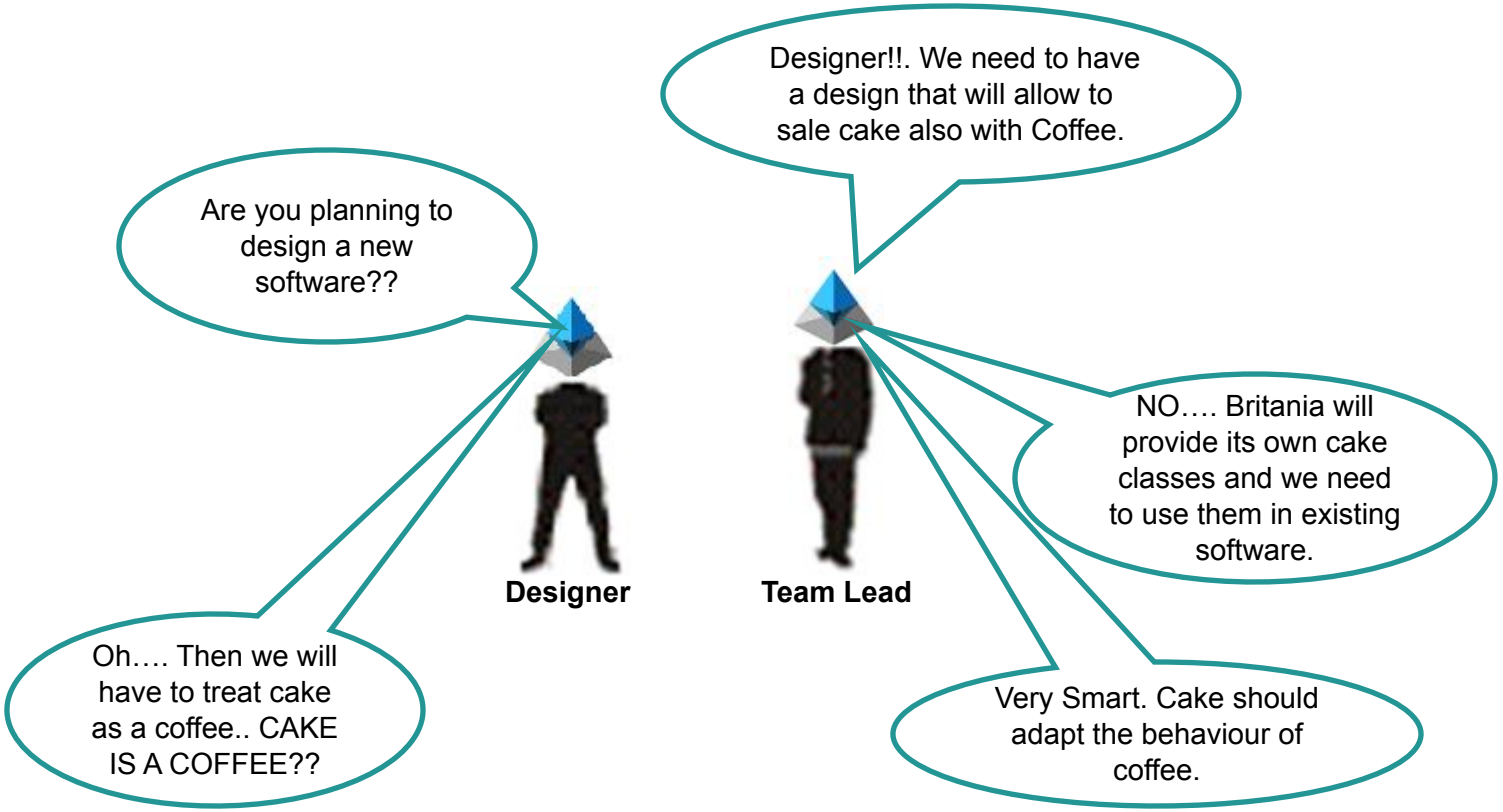
```java
public interface Cake {
    String getCakeName();
    int cost();
}
```

```java
public class ChocolateCake implements Cake{
    public int cost() {
        return 80;
    }
    public String getCakeName() {
        return "Chocolate cake";
    }
}
```

```java
public class VanillaCake implements Cake{
    public int cost() {
        return 60;
    }
    public String getCakeName() {
        return "Vanilla Cake";
    }
}
```

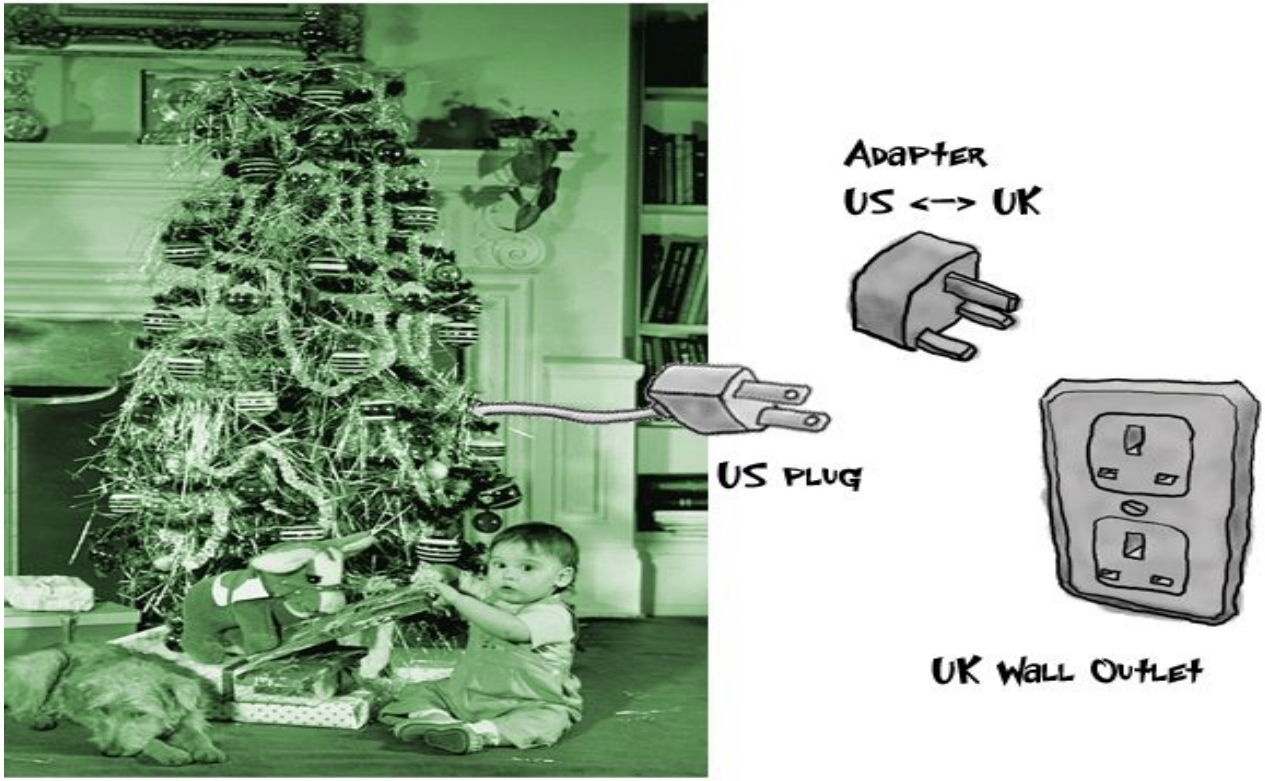
© Copyright 2021 : Accolite : All Rights Reserved

Team Lead– Designer Discussion



© Copyright 2021 : Accolite : All Rights Reserved

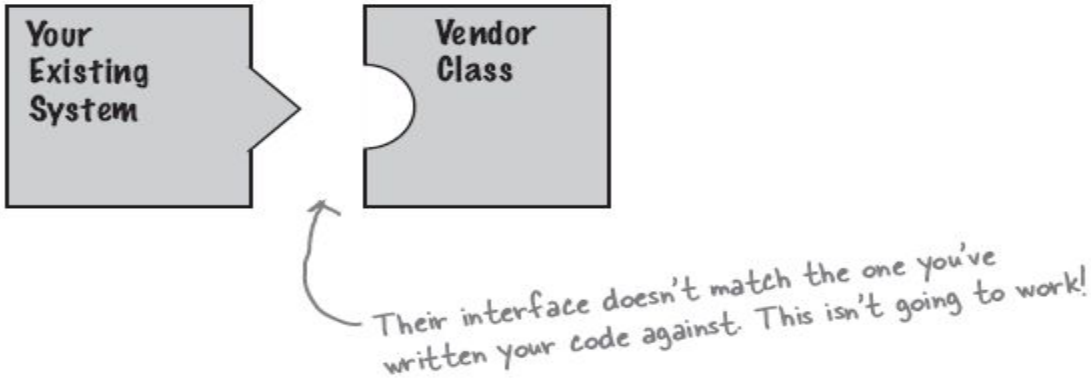
Adapter All Around Us



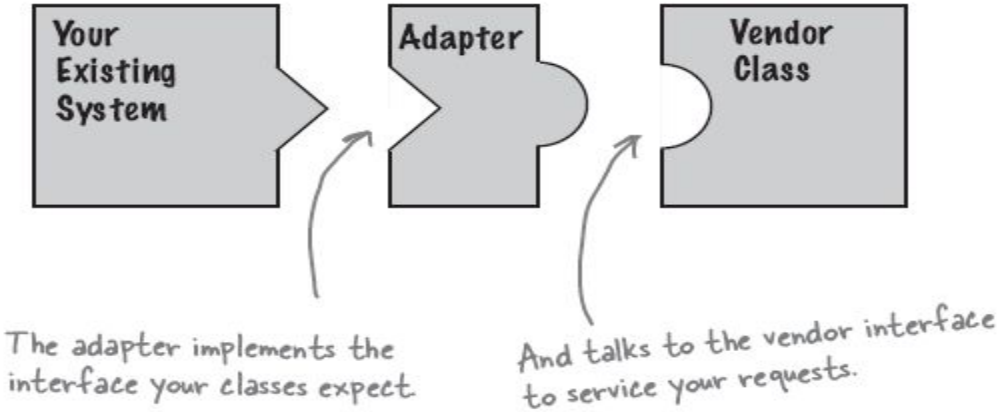
© Copyright 2021 : Accolite : All Rights Reserved

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:



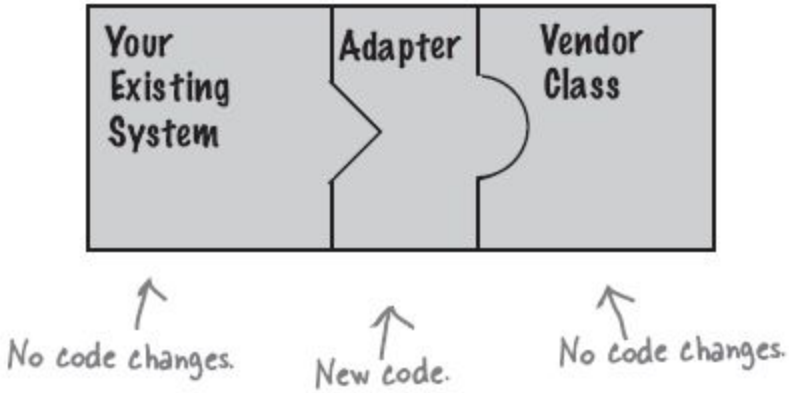
Okay, you don't want to solve the problem by changing your existing code (and you can't change the vendor's code). So what do you do? Well, you can write a class that adapts the new vendor interface into the one you're expecting.



© Copyright 2021 : Accolite : All Rights Reserved

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.

© Copyright 2021 : Accolite : All Rights Reserved

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() {
    }
}
```

© Copyright 2021 : Accolite : All Rights Reserved

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;
        }
    }
}
```

© Copyright 2021 : Accolite : All Rights Reserved

Behavioral Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

Structural Patterns Classification

- **Behavioral Patterns**
  - **Strategy**
  - **Command**
  - **Observer**
  - **Template**
  - **Chain Of Responsibility**
  - **Iterator**
  - **Mediator**
  - **Memento**

© Copyright 2021 : Accolite : All Rights Reserved

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.

© Copyright 2021 : Accolite : All Rights Reserved

Strategy Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

New Requirement of Barista

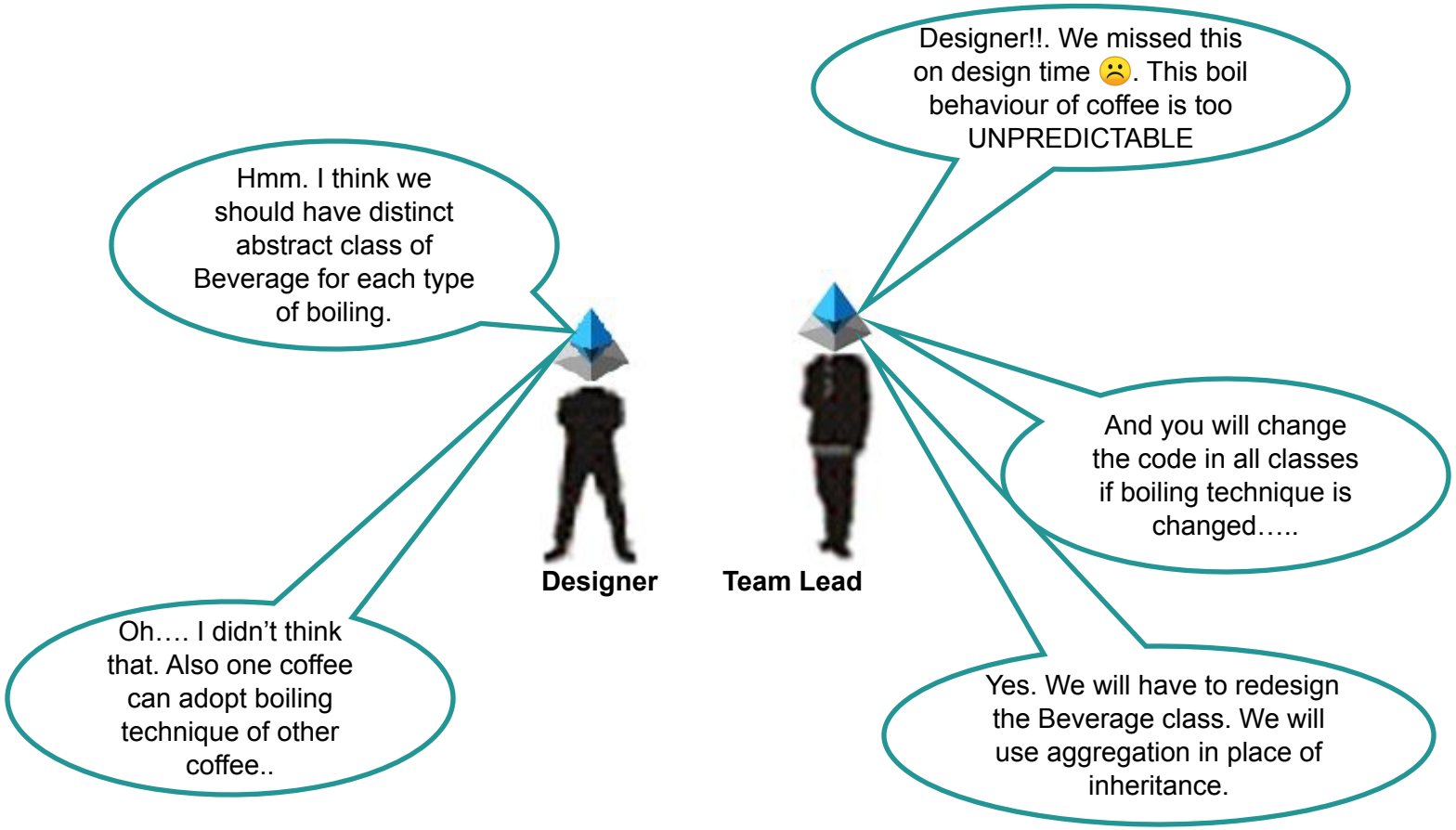
- Way of boiling the water is different for different coffee.
- Also the temperature of water is different for all type of coffee.
- Some products (Britania cake) at Barista do not have boiling water.
- Will you boil the water for cold coffee???

We are not happy with the way, you are boiling the water (boilWater method).

You boil the water in same way?? Boiling temperature and boiling method is different for all.

© Copyright 2021 : Accolite : All Rights Reserved

Team Lead– Developer Discussion



© Copyright 2021 : Accolite : All Rights Reserved

Very unstable behavior???
Strategy Design Pattern..

```java
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;
    }
}
```

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

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

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

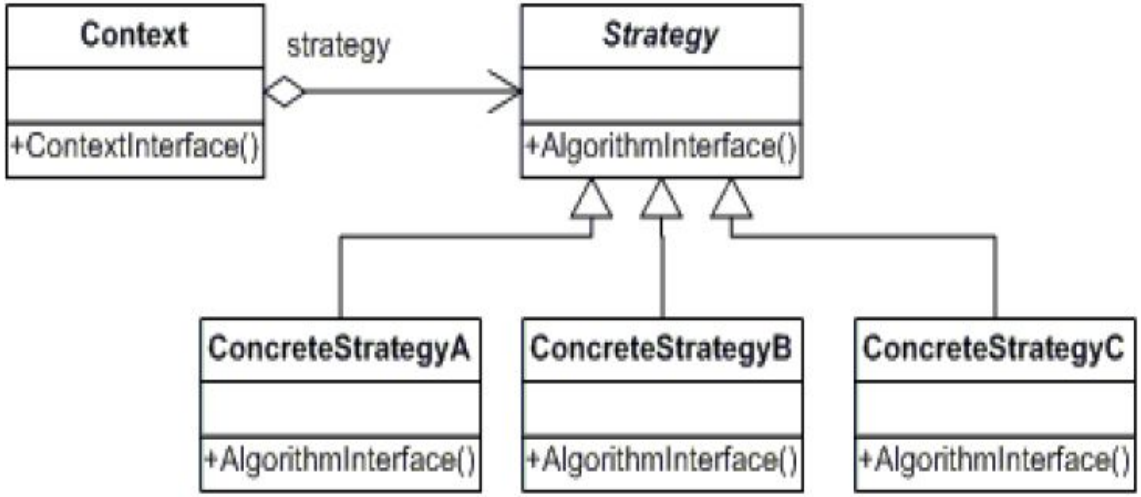
© Copyright 2021 : Accolite : All Rights Reserved

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

© Copyright 2021 : Accolite : All Rights Reserved

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.

© Copyright 2021 : Accolite : All Rights Reserved

Template Method Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

Tea also can be a product of Barista

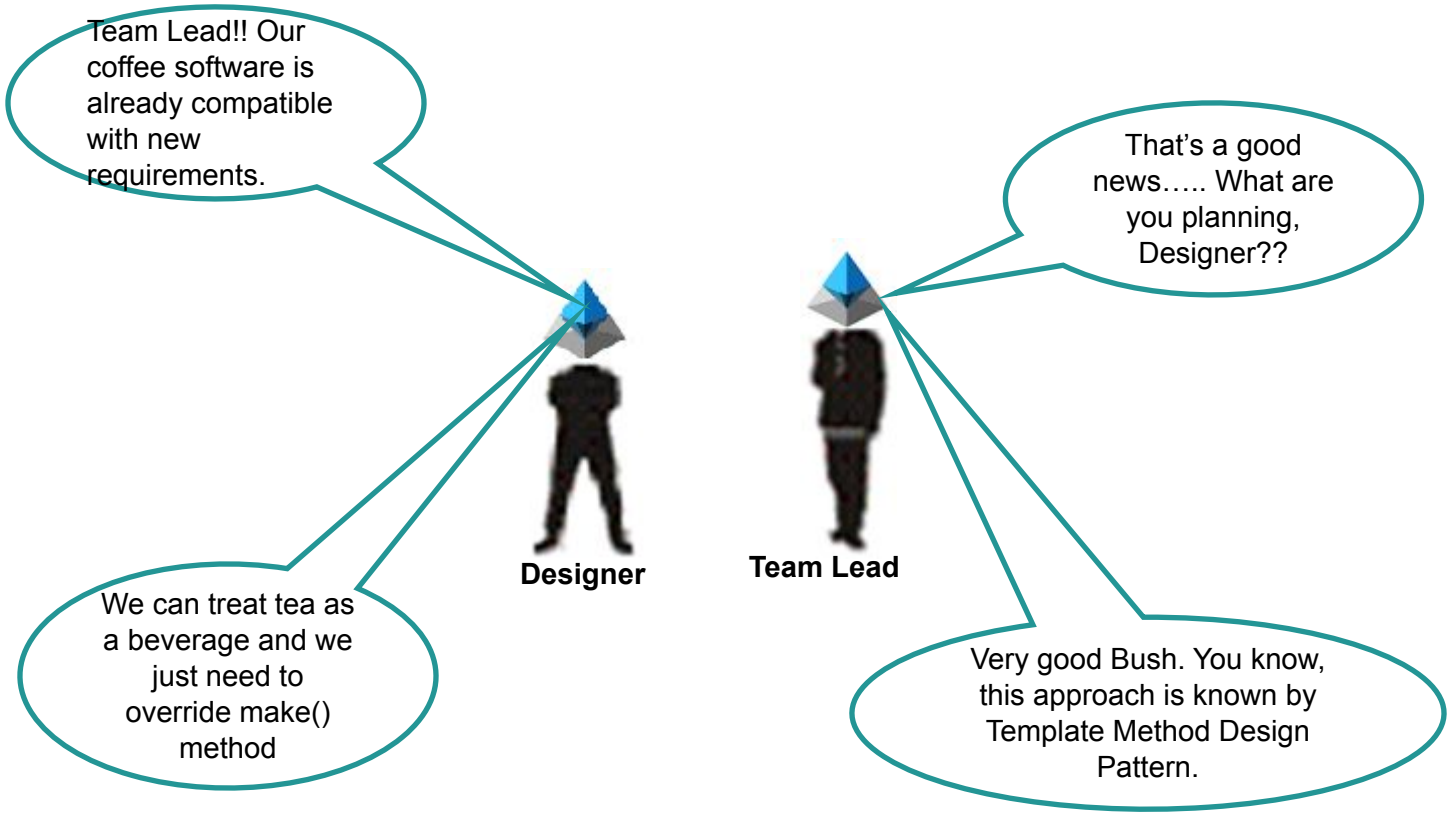
- Barista also want to take the order of Tea.
- All the steps to order a tea are similar to coffee. The only difference is that we use tea leaf in place of coffee bean.

We want to sale Tea also. All the to take the order of a tea are similar to coffee

But Making of tea follow different process. Obviously we will add tea leaf in place of coffee bean

© Copyright 2021 : Accolite : All Rights Reserved

Team Lead– Designer Discussion



© Copyright 2021 : Accolite : All Rights Reserved

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");
    }
}
```

© Copyright 2021 : Accolite : All Rights Reserved

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();
}
```

© Copyright 2014: Accolite : All Rights Reserved

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).

© Copyright 2021 : Accolite : All Rights Reserved

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");
    }
}
```

© Copyright 2021 : Accolite : All Rights Reserved

Template Method Design Pattern

```
public abstract class CaffeineBeverage {

    void final prepareRecipe() {

        boilWater();

        brew();

        pourInCup();

        addCondiments();

    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        // implementation
    }

    void pourInCup() {
        // implementation
    }

}
```

prepareRecipe() is our **template method**. Why?

Because:

(1) It is a method, after all.

(2) It serves as a template for an algorithm, in this case, an algorithm for making caffeinated beverages.

In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class...

...and some are handled by the subclass.

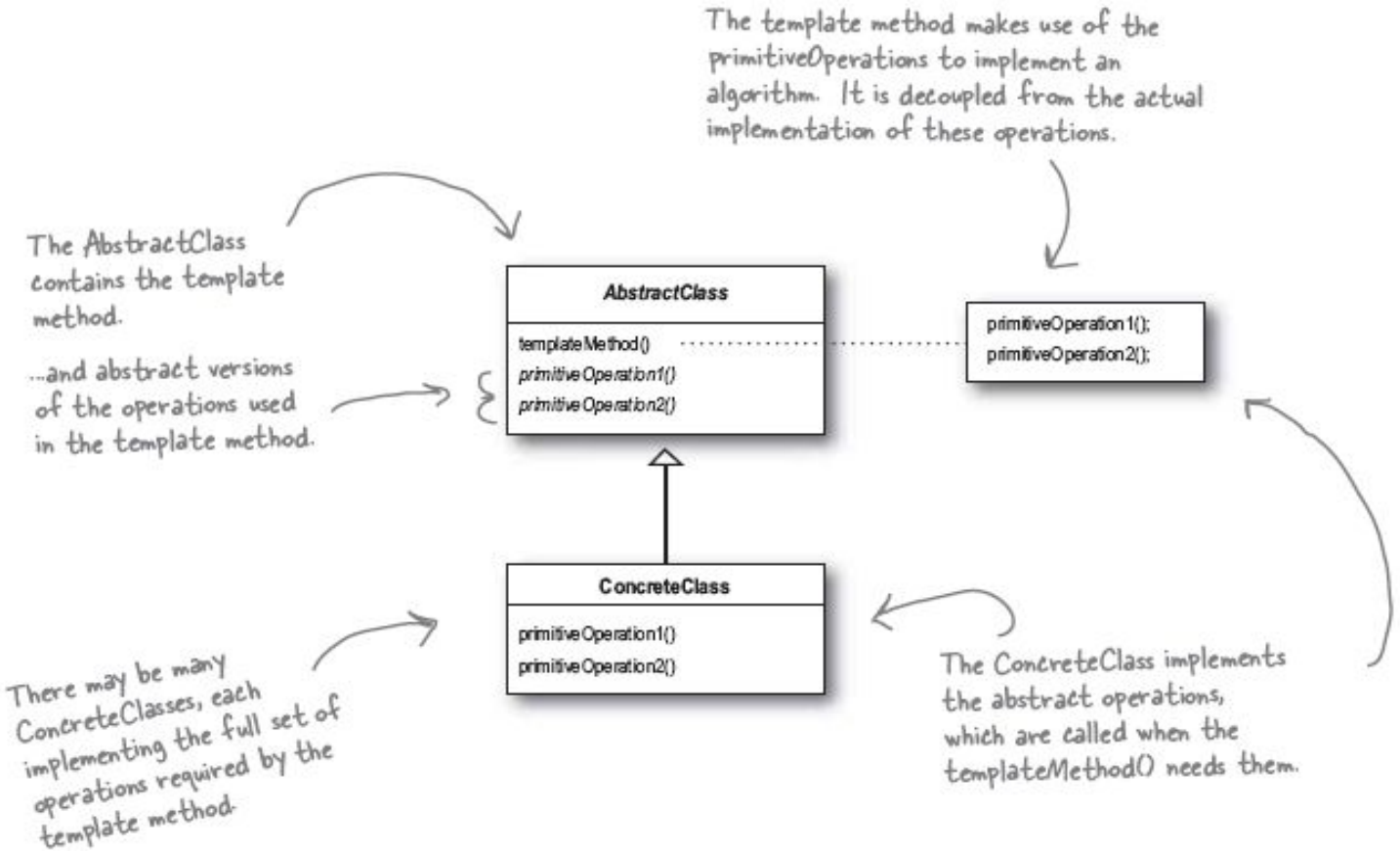
The methods that need to be supplied by a subclass are declared abstract.

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

© Copyright 2021 : Accolite : All Rights Reserved

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



© Copyright 2021 : Accolite : All Rights Reserved

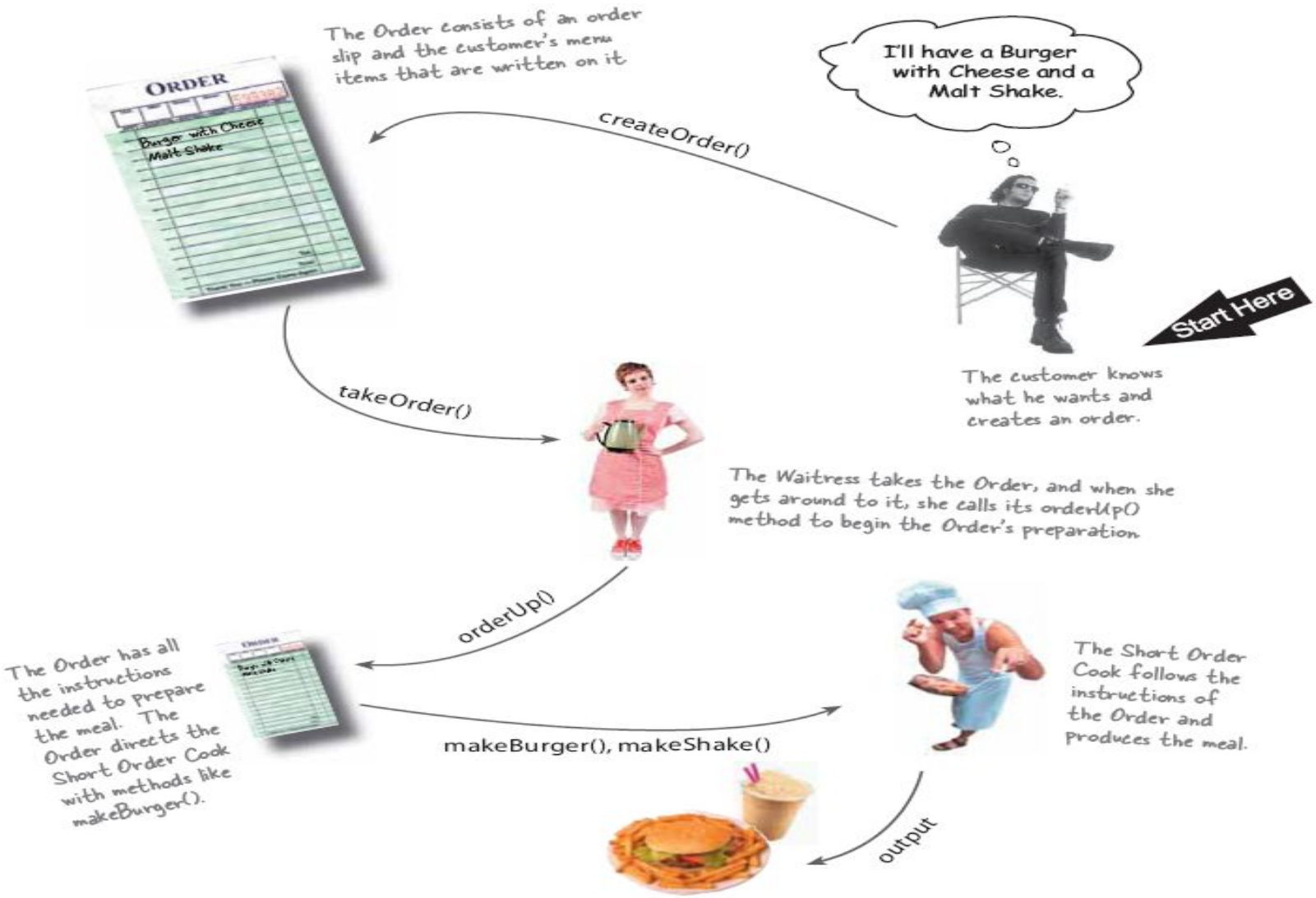
Command Design Patterns

© Copyright 2021 : Accolite : All Rights Reserved

Barista Introduces Waitress

© Copyright 2021 : Accolite : All Rights Reserved

Lets study the interaction in detail



© Copyright 2021 : Accolite : All Rights Reserved

Different responsibilities

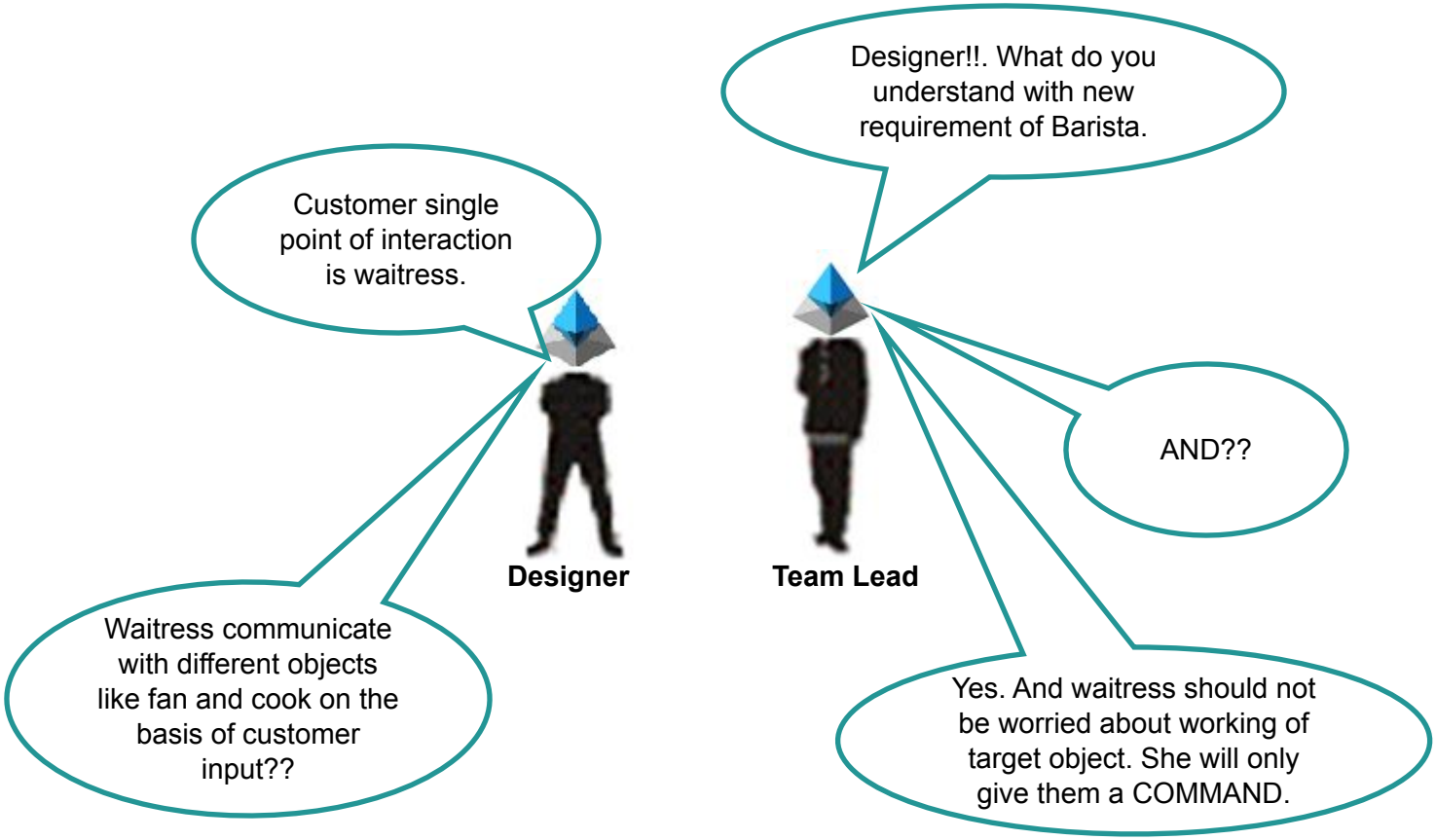
- Customer
  - Ask waitress for the Coffee
  - Ask waitress to switch on the fan
  - Ask Waitress to switch off the fan

- Waitress
  - Take the order from the customer.
  - Process the order

- Cook
  - Receive the order from waitress
  - Prepare the food.

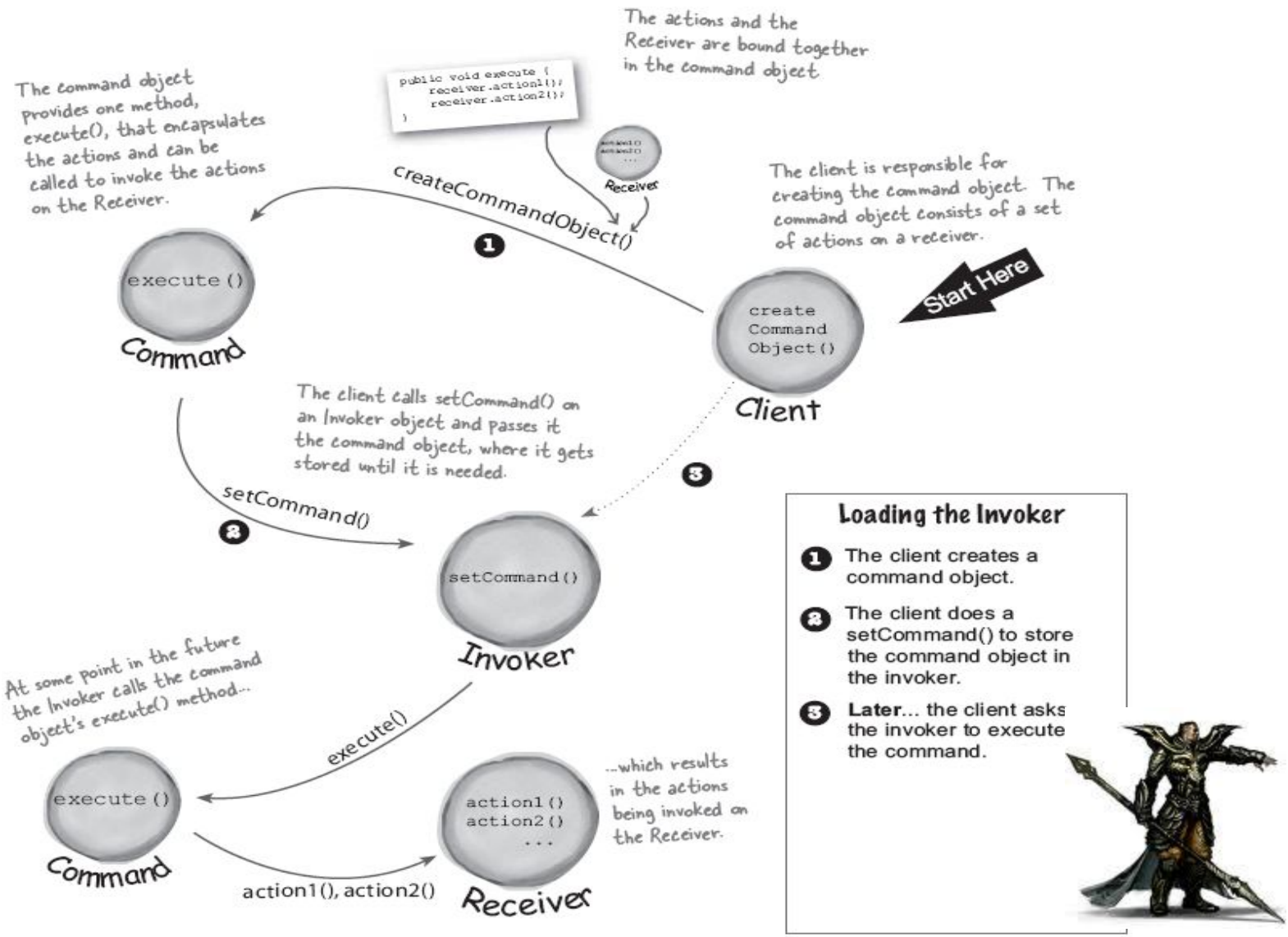
© Copyright 2021 : Accolite : All Rights Reserved

Team Lead– Designer Discussion



© Copyright 2021 : Accolite : All Rights Reserved

Command Design Pattern



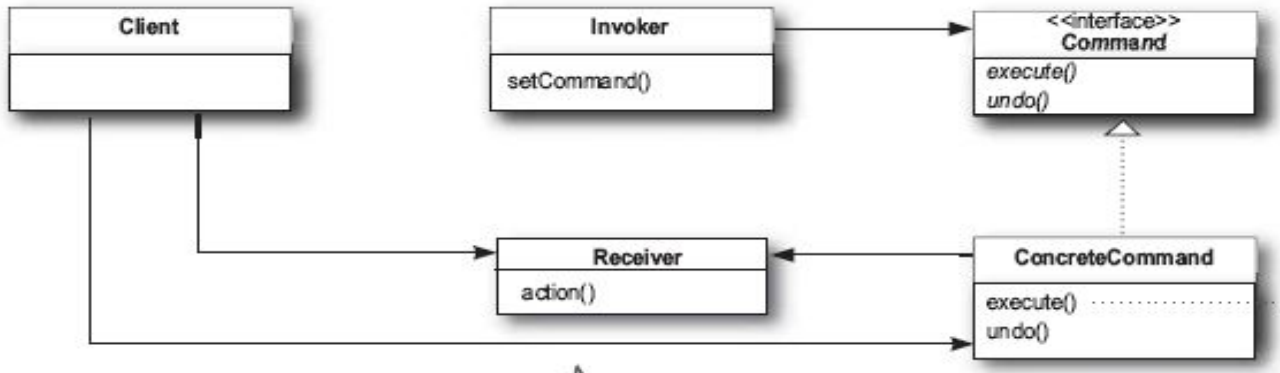
© Copyright 2021 : Accolite : All Rights Reserved

Command Pattern Class Diagram

The Client is responsible for creating a ConcreteCommand and setting its Receiver.

The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.

Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to preform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.



The execute method invokes the action(s) on the receiver needed to fulfill the request.

```
public void execute() {
    receiver.action()
}
```

The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.

The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.

© Copyright 2021 : Accolite : All Rights Reserved

Command Pattern Definition

- The command pattern encapsulates a request as an object, thereby letting you parameterized other objects with different requests.
- One object (Waitress) is a single point of contact for different kinds of request (from Customer) and map those request to different objects to perform an operation.

© Copyright 2021 : Accolite : All Rights Reserved

Barista Software with Command

```java
public class Waitress {
    private List<Command> commands = new ArrayList<Command>();
    private Map<String, Integer> nameIndexMap =
        new HashMap<String, Integer>();
    public Waitress() {
        //Some command object can be added here
    }
    public void addCommand(String name, Command command) {
        commands.add(command);
        nameIndexMap.put(name, commands.size() - 1);
    }
    public void executeCommand(String name) {
        Command command = commands.get(nameIndexMap.get(name));
        command.execute();
    }
    public Set<String> getCommandsName(){
        return nameIndexMap.keySet();
    }
}
```

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

```java
public class CoffeeCommand implements
        Command {
    Beverage beverage = null;
    public CoffeeCommand(Beverage beverage) {
        super();
        this.beverage = beverage;
    }
    public void execute() {
        beverage.order();
    }
}
```

```java
public class FanOnCommand implements
        Command{
    Fan fan = null;
    public FanOnCommand(Fan fan) {
        super();
        this.fan = fan;
    }
    public void execute() {
        fan.on();
    }
}
```

© Copyright 2021 : Accolite : All Rights Reserved

Time to write down the client code

```
public class TestCommand {
    public static void main(String[] args) {
        Waitress waitress = new Waitress();
        Fan fan = new Fan();
        waitress.addCommand("FanOn", new FanOnCommand(fan));
        waitress.addCommand("FanOff", new FanOnCommand(fan));
        try {
            BufferedReader br = new BufferedReader
                (new InputStreamReader(System.in));
            printCommands(waitress);
            Set<String> commandNames = waitress.getCommandsName();
            String str = null;
            while (!(str = br.readLine()).equals("X")) {
                if(!commandNames.contains(str)){
                    System.out.println("Enter the correct choice: ");
                    continue;
                }
                waitress.executeCommand(str);
                System.out.println("-----------------------");
                printCommands(waitress);
            }
        } catch (Exception e) {
        }
    }
    public static void printCommands(Waitress waitress){
        System.out.println("Choose one of the below command:");
        for(String command : waitress.getCommandsName()){
            System.out.println(command);
        }
        System.out.println("Print X to exit");
        System.out.println("Enter the Choice:");
    }
}
```

INVOKER

RECEIVER

Add Commands to INVOKER

Client ask INVOKER to execute command

© Copyright 2021 : Accolite : All Rights Reserved

Thank You

© Copyright 2021 : Accolite : All Rights Reserved