ABOUT JAVA™ DESIGN PATTERNS

JAVA™ 디자인 패턴에 관하여

Nathan Cho *조나단 [20425]*

Department of Web Operation

Sunrin Internet High School

Seoul, Korea

dev.bedrock@gmail.com

목차

1. 소프트웨어 디자인 패턴

소프트웨어 디자인 패턴이란 특정한 문제나 작업을 위한 적절하고 재사용이 가능한 해결 방안이다. 감이 잡히지 않는다면 다음 예시에서 어떤 문제가 '특정한' 문제이고 어떤 해결 방안이 '디자인 패턴' 에 해당되는지 파악해 보자.

문제 사항: 하나의 객체만 생성할 클래스를 만들어야 한다. 이 객체는 모든 다른 클래스들에서 사용할 수 있어야 한다.

해결 방안: 싱글턴 패턴이 위 문제 사항에 가장 적합한 해결 방안이다.

이처럼 모든 디자인 패턴은 각각의 특정한 규칙이나 특징을 가지고 있다. 이러한 규칙과 특징에 대하여는 밑의 디자인 패턴들에서 알아보도록 하자.

하지만 기억해야 하는 점은 **디자인 패턴은 객체 지향적 디자인 패턴에서 발생하는 문제들에 대한 해결 방법이지 직접적인 프로그래밍 언어에 사용하는 해결 방법이 아니다.** 즉 디자인 패턴은 해결책에 대한 표준적인 아이디어를 주는 것이다.

이것을 처음 알게 되었다면 이러한 질문을 할 수 있다. '문제에 대한 직접적인 해결책을 주는 것이 아니라면 굳이 사용할 이유가 있는가?' 다음의 디자인 패턴을 사용함으로써 얻을 수 있는 장점들에 대해서 알아보자.

디자인 패턴의 장점

1. **재사용**이 가능하다.
2. 유연성을 높여 **유지보수가 쉽다.**
3. 많은 개발자들에 의해 검증되어 문제 해결에 걸리는 시간을 단축할 수 있다.

그러면 이러한 디자인 패턴들은 소프트웨어 개발의 어떤 단계에서 사용해야 하는가? 디자인 패턴은 코드 설계의 부분에서 이루어지기 때문에 코드 작성 전 단계에서 수행해야

한다. 소프트웨어 개발 수명 주기 (Software Development Life Cycle, **SDLC**) 에서는 [**요구 사항 분석**] 부분에서 이루어져야 한다.

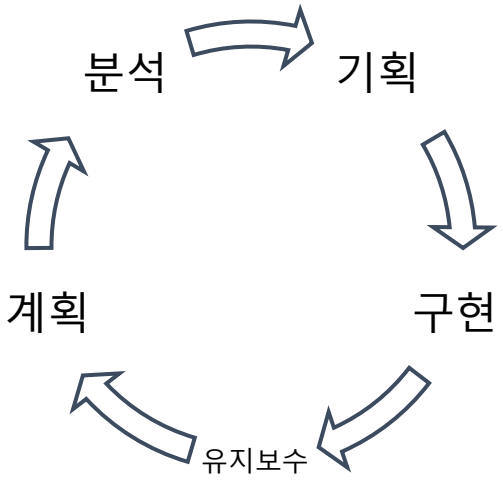


그림 1.1 소프트웨어 개발 수명 주기 (Software Development Life Cycle, **SDLC**)

디자인 패턴의 종류는 *GoF - 재사용 가능 객체 지향 소프트웨어의 디자인 패턴과 원소들 (Gang of Four - Design patterns, elements of reusable object-oriented software)* 이라는 책에 따르면 23 개의 디자인 패턴 (Gof 디자인 패턴으로 알려져 있다) 이 나온다. 다음은 디자인 패턴들의 세 가지 분류이다.

디자인 패턴의 분류

1. 생성 (Creational) 디자인 패턴
2. 구조 (Structural) 디자인 패턴
3. 동작 (Behavioral) 디자인 패턴

이 보고서에서는 생성 디자인 패턴의 대표적인 패턴들의 설명과 예시들을 설명한다.

1) Java 디자인 패턴

자바의 가장 중요한 부분 중 하나는 디자인 패턴을 사용하는 것이다. 자바는 내부적으로 디자인 패턴을 사용하고 있기 때문에 디자인 패턴을 사용하는 것이 편하고도 중요하다.

자바의 디자인 패턴은 두 가지로 분류할 수 있다.

1. JSE (Java 기본) 디자인 패턴
2. JEE 디자인 패턴

이 보고서에서는 주로 기본적인 디자인 패턴인 JSE 디자인 패턴을 다룰 것이다.

2. 생성 (Creational) 패턴

생성 디자인 패턴은 객체를 기본적인 new 생성자를 통해 생성하는 방법 외의 생성 방법을 숨기면서 객체를 생성하는 방법을 제시한다. 상황과 객체 생성에 따라서 다양하게 방법을 제시한다.

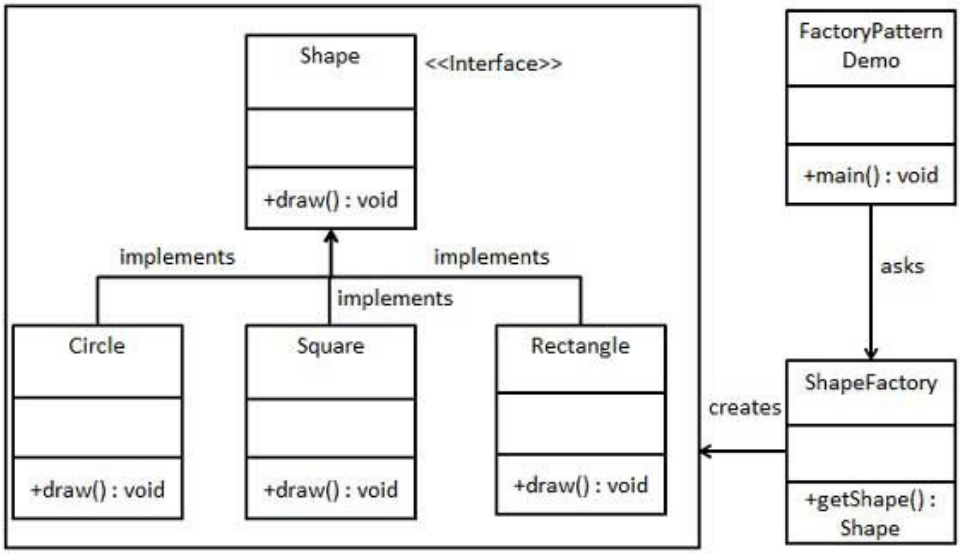
생성 디자인 패턴에는 대표적으로 6 가지 종류가 있다.

1. 팩토리(Factory) 패턴
2. 추상 팩토리 (Abstract Factory) 패턴
3. 싱글턴 (Singleton) 패턴
4. 프로토타입 (Prototype) 패턴
5. 빌더 (Builder) 패턴
6. 오브젝트 풀 (Object Pool) 패턴

여기서는 팩토리 패턴, 싱글턴 패턴, 빌더 패턴만을 다룬다.

1) 팩토리 (Factory) 패턴

팩토리 패턴은 자바에서 가장 많이 쓰이는 디자인 패턴 중 하나이다. 팩토리 패턴에서는 클라이언트에게 객체 생성의 방법을 드러내지 않고 객체를 생성하고 인터페이스를 통해서 객체를 전달해준다.



Shape.java

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

Shape 인터페이스를 implement 하는 클래스들을 만든다.

Rectangle.java

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

Square.java

```
public class Square implements Shape {
   @Override
```

```
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

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

클래스 객체를 반환하는 팩토리 클래스를 만든다.

ShapeFactory.java

```
public class ShapeFactory {
   //use getShape method to get object of type shape
   //getShape 메소드로 새로운 Shape 객체를 생성할 수 있음
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

팩토리 클래스를 실행하는 예시 클래스 코드이다.

FactoryPatternDemo.java

```
public class FactoryPatternDemo {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");
```

```
      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of circle
      shape3.draw();
   }
}
```

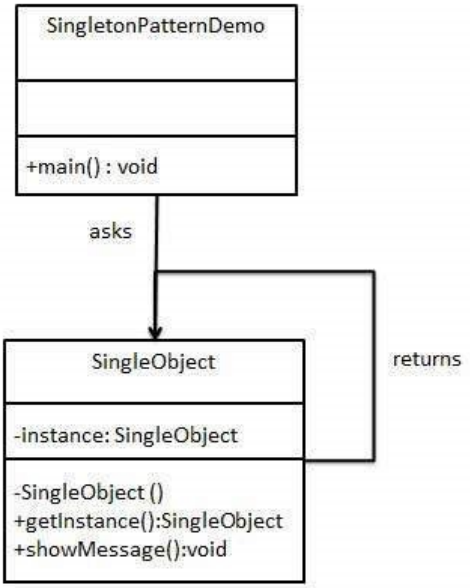
출력 결과

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

이렇게 객체를 직접적으로 생성하지 않더라도 팩토리 클래스를 사용해서 객체를 가져올 수 있다.

2) 싱글턴 (Singleton) 패턴

싱글턴 패턴은 자바에서 가장 간단한 디자인 패턴중 하나이다. 생성 패턴의 범주에 속해서 객체를 생성하는데 가장 좋은 방법을 제공한다. 객체를 직접 생성하는 것이 아닌 이미 생성된 객체의 인스턴스를 가져오는 형식으로 작동한다.



SingleObject.java

```
public class SingleObject {
   //create an object of SingleObject
   //SingleObject 객체를 생성함
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be instantiated
   // 생성자를 private 로 만들어서 객체를 직접 생성할 수 없게 한다
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //아래 코드는 SingleObject 의 생성자가 private 이기 때문에 컴파일 에러가 발생한다.
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

출력 결과

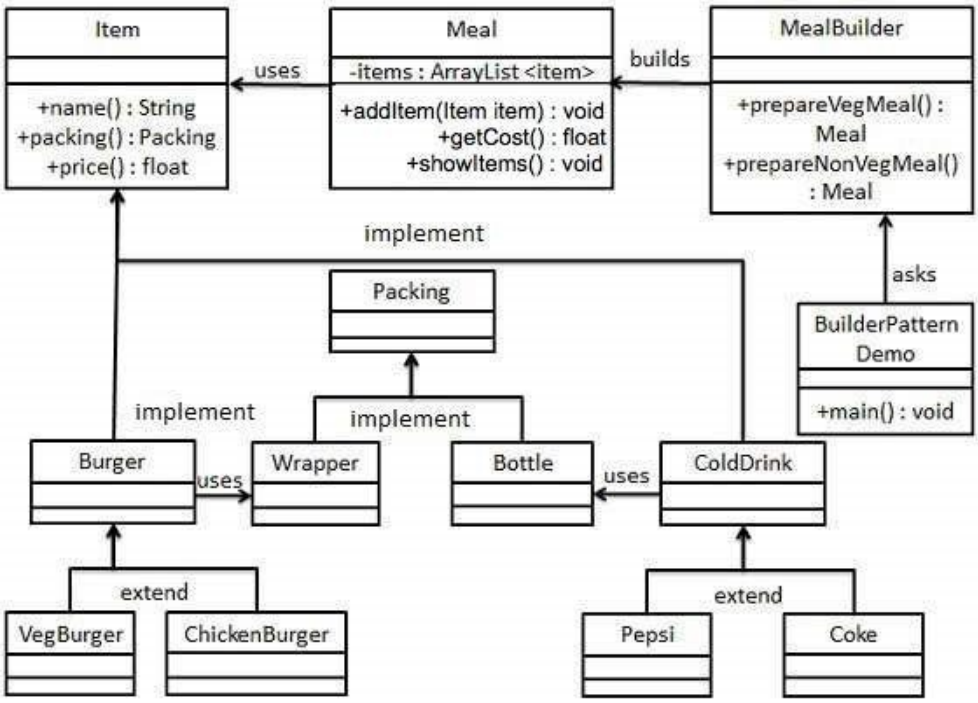
```
Hello World!
```

이렇게 직접 객체를 가져오는 것이 아니라 인스턴스로 가져옴으로써 객체 생성을 하드코딩 (hardcoded) 하지 않고 할 수 있다.

3) 빌더 (Builder) 패턴

빌더 디자인 패턴은 복잡한 객체를 간단한 객체들로 하나씩 초기화 하면서 생성할 수 있도록 해주는 디자인 패턴이다. 생성자에 많은 개수의 인자가 들어가면 코드의 가독성이 떨어지기 때문에 빌더 패턴을 사용하는 것이 매우 효율적이다.

빌터 패턴의 예시로는 실생활의 패스트푸드점을 구현해 보았다. 기본적인 식사는 버거 (Burger)와 음료수 (Cold Drink)로 버거는 치킨 버거나 채식 버거 (Veg Burger)로 포장지 (Wrapper) 에 싸서 제공되고, 음료수는 콜라나 펩시로 컵 (Bottle)에 담겨서 제공된다.



버거와 음료수가 implement 하는 Item 인터페이스를 만든다.

Item.java

```
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

버거 포장지와 음료수 컵을 포장하는 함수를 내장하는 Packing 인터페이스를 만든다.

Packing.java

```
public interface Packing {
   public String pack();
}
```

Wrapper.java

```
public class Wrapper implements Packing {
   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

Bottle.java

```
public class Bottle implements Packing {
   @Override
   public String pack() {
      return "Bottle";
   }
}
```

기본적인 작동을 포함하고 있는 음식 추상 클래스들을 만든다.

Burger.java

```
public abstract class Burger implements Item {
   @Override
   public Packing packing() {
      return new Wrapper();
   }

   @Override
   public abstract float price();
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {
```

```
            @Override
            public Packing packing() {
        return new Bottle();
            }

            @Override
            public abstract float price();
}
```

식사 클래스들을 상속받는 실제 식사 클래스들을 만든다.

VegBurger.java

```
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {

   @Override
   public float price() {
      return 50.5f;
   }

   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

Coke.java

```
public class Coke extends ColdDrink {

   @Override
   public float price() {
      return 30.0f;
   }
```

```
   @Override
   public String name() {
      return "Coke";
   }
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {

   @Override
   public float price() {
      return 35.0f;
   }

   @Override
   public String name() {
      return "Pepsi";
   }
}
```

위의 식사 메뉴 (Item) 클래스의 ArrayList 를 가지고 있는 식사 (Meal) 클래스를 만든다.

Meal.java

```
import java.util.ArrayList;
import java.util.List;

public class Meal {
   private List<Item> items = new ArrayList<Item>();

   public void addItem(Item item){
      items.add(item);
   }

   public float getCost(){
      float cost = 0.0f;

      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }

   public void showItems(){
      for (Item item : items) {
         System.out.print("Item : " + item.name());
         System.out.print(", Packing : " + item.packing().pack());
```

```
            System.out.println(", Price : " + item.price());
        }
    }
}
```

빌더 패턴의 핵심인 빌더 클래스 (MealBuilder) 를 만든다.

MealBuilder.java

```
public class MealBuilder {
   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }

   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

식사를 제공하는 예시 클래스를 만든다.

BuilderPatternDemo.java

```
public class BuilderPatternDemo {
   public static void main(String[] args) {
      // 빌더 클래스를 사용해서 객체들을 생성한다
      MealBuilder mealBuilder = new MealBuilder();

      Meal vegMeal = mealBuilder.prepareVegMeal();
      System.out.println("Veg Meal");
      vegMeal.showItems();
      System.out.println("Total Cost: " + vegMeal.getCost());

      Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
      System.out.println("\nNon-Veg Meal");
      nonVegMeal.showItems();
      System.out.println("Total Cost: " + nonVegMeal.getCost());
   }
}
```

출력 결과

```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0

Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

이렇게 복잡한 객체들을 빌더 클래스들을 사용해서 생성하고 사용할 수 있다.

3. 결론 / 자료출처

모든 소프트웨어 디자인 패턴을 담지 못해서 아쉽지만 많은 예제 코드들에서 봐오던 팩토리 패턴과 빌더 패턴이 코드 밑에서 어떻게 작성 되어 있는지 알 수 있게 되어 의미있었다.

간단한 코드들에서는 신경쓰지 않아도 될 객체지향의 캡슐화를 사용한 보안 같은 문제들을 이렇게 해결한다는 것을 새로 알게 되었다. 오랜 시간 동안 많은 개발자들이 어떻게 하면 가장 완벽한 방법으로 코드를 작성하고 클라이언트를 보호할지에 대해 고민했다는 것을 조금이나마 느낄 수 있었던 것 같다.

자료 출처


```
https://www.javatpoint.com/core-java-design-patterns
https://en.wikipedia.org/wiki/Systems_development_life_cycle
https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
```