REPORT

중간고사 보고서

자바프로그래밍2 1분반

제출일 2023. 10.31

소속 컴퓨터공학과

학번 32183520

이름 이 주성

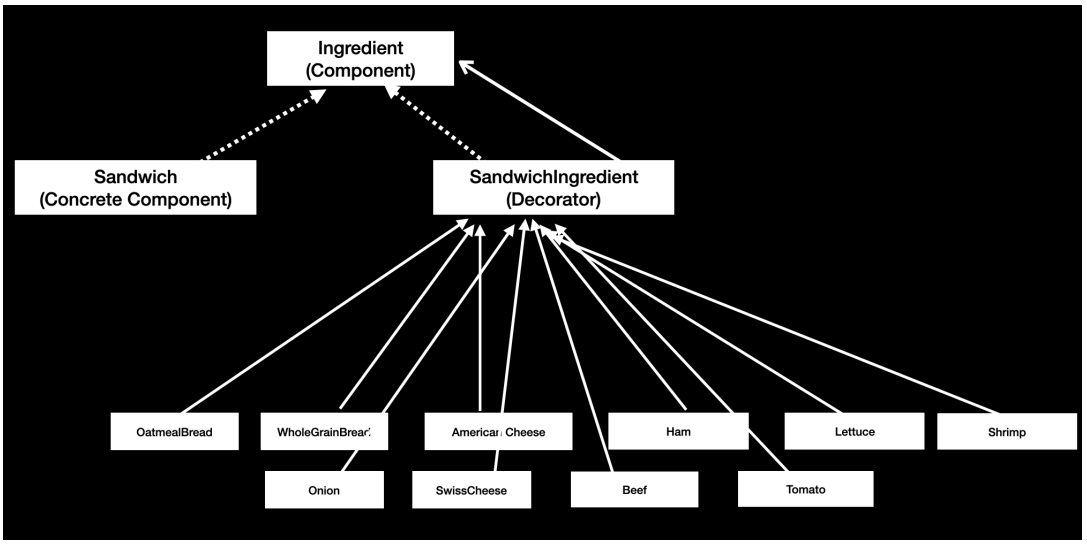
Singleton Pattern

- 문제의 코드는 싱글톤의 classic한 구현 방법으로 멀티 스레드 환경에서는 잘 동작하지 않는다.
- Double Checking Lock을 사용해 동기화를 시켜줘서 멀티스레드 환경에서도 잘 동작하도록 한다.
- voloatile과 synchronized를 사용해 동기화를 처리한다.

```
public class Singleton {
    // 동기화된 싱글톤으로 관리되어야 하므로 private volatile static으로 선언
    private volatile static Singleton instance = null;

    // 다른곳에서 생성할 수 없도록 생성자를 private로 선언
    private Singleton() {
        System.out.println("Singleton constructor");
    }

    // 싱글톤 객체를 얻을 유일한 메서드
    public static Singleton getInstance() {
        // 아직 생성되지 않았을 경우
        if (instance == null) {
            // synchronized 동기화
            synchronized (Singleton.class) {
                if (instance == null) {
                    // 싱글톤 객체 생성
                    instance = new Singleton();
                }
            }
        }

        return instance;
    }
}
```

Decorator Pattern

- 샌드위치를 Decorator Pattern을 이용해 생성한다.
- Component인 Ingredient를 Sandwich와 Decorator들이 구현한다.



Ingredient (Component)

```
public interface Ingredient {
   String getDescription();
   int cost();
}
```

Sandwich (Concrete Component)

- 구체 컴포넌트로 데코레이터로 꾸밀 예정

```
public class Sandwich implements Ingredient {

   public Sandwich() {   }

   @Override
   public String getDescription() {
       return "Sandwich";
   }

   @Override
   public int cost() {
       return 0;
   }
}
```

SandwichIngredient (Decorator)

- Ingredient를 구현하면서 참조변수를 가짐
- 데코레이터로 꾸밀 ingredient를 생성자의 파마리터로 받음

```
// Component인 Ingredient를 구현
public abstract class SandwichIngredient implements Ingredient {
    // Component인 Ingredient의 참조변수를 가짐
    private Ingredient ingredient;

    // 생성자로 데코레이터로 꾸밀 Ingredient을 받음
    public SandwichIngredient(Ingredient ingredient) {
        this.ingredient = ingredient;
    }

    public String getDescription() {
        return ingredient.getDescription();
    }

    @Override
    public int cost() {
        return ingredient.cost();
    }
}
```

Beef (Concrete Decorator)

- 추가 기능을 구현하는 구체 데코레이터이다.
- 나머지 Concrete Decorator들도 동일

```
public class Beef extends SandwichIngredient {
    private String name = "Beef";

    // Beef 데코에이터로 꾸밀 Ingredient를 생성자의 파라미터로 받음
    public Beef(Ingredient ingredient) {
        super(ingredient);
    }

    // ingredient의 getDescription()에 Beef를 붙임
```

```
    @Override
    public String getDescription() {
        return super.getDescription() + "," + name;
    }

    // ingredient의 cost()에 Beef 가격 5000원 더함
    @Override
    public int cost() {
        return super.cost() + 5000;
    }
}
```

MainTest

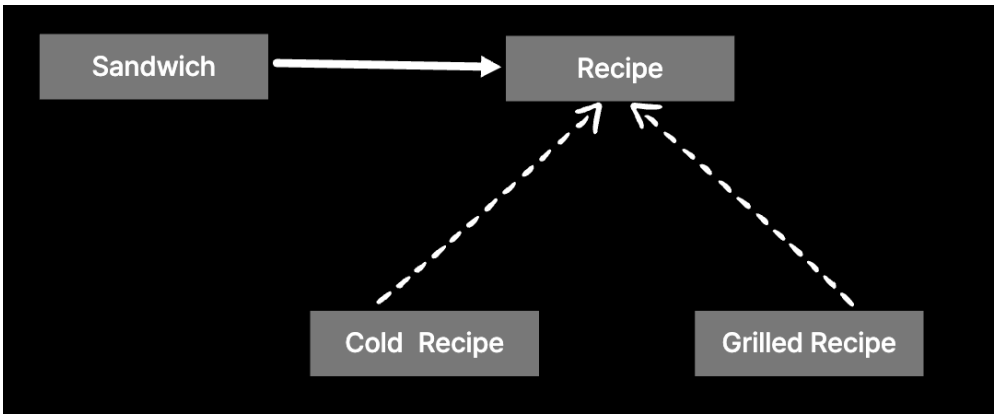
```
// Create sandwich
Ingredient sandwich = new Sandwich();
sandwich = new WholeGrainBread(sandwich);
sandwich = new Beef(sandwich);
sandwich = new SwissCheese(sandwich);
sandwich = new Lettuce(sandwich);
sandwich = new Onion(sandwich);
sandwich = new Tomato(sandwich);
```

- 결과

```
Sandwich,WholeGrainBread,Beef,SwissCheese,Lettuce,Onion,Tomato 10000
```

Strategy Pattern

- Recipe를 Strategy Pattern을 적용해 String cook() 메서드로 실행 시간에 동적으로 정한다.



Recipe

- cook() 메서드 보유한 인터페이스

```
public interface Recipe {
   String cook();
}
```

ColdRecipe

- Recipe 구현한 구체적인 레시피
- cook() 메서드를 통해 동적으로 정해짐

```
public class ColdRecipe implements Recipe {
   @Override
   public String cook() {
       return "Cold ";
   }
}
```

나머지 Concrete Recipe들도 같은 구조입니다.

Sandwich

- Recipe를 참조변수로 가지며 cook() 메서드를 사용해 Strategy Pattern으로 구체적인 레시피를 결정함

```
public class Sandwich implements Ingredient {
    // Recipe의 참조변수를 가지고 있음
    private Recipe recipe;
    public Sandwich(Recipe recipe) {
        this.recipe = recipe;
    }

    @Override
    public String getDescription() {
        // 구체적인 레시피 결정
        return this.recipe.cook() + "Sandwich";
    }

    @Override
    public int cost() {
        return 0;
    }
}
```

Factory Pattern

- Lecttuce, Tomato, Onion 재료는 팩토리 패턴을 사용해 객체를 생성한다.
- 팩토리 패턴을 이용해 객체 생성 부분을 캡슐화하고 getInstance()로 객체를 얻을 수 있도록 설계한다.

```
public class OtherIngredientFactory {
    public static Ingredient getInstance(String name, Ingredient ingredient) {
        switch (name) {
            case "Lettuce":
                return new Lettuce(ingredient);
            case "Tomato":
                return new Tomato(ingredient);
            case "Onion":
                return new Onion(ingredient);
            default:
                return ingredient;
        }
```

```
    }
}
```

MainTest

```
// Create sandwich
Ingredient sandwich = new Sandwich(new GrilledRecipe());
sandwich = new WholeGrainBread(sandwich);
sandwich = new Beef(sandwich);
sandwich = new SwissCheese(sandwich);
sandwich = OtherIngredientFactory.getInstance("Lettuce", sandwich);
sandwich = OtherIngredientFactory.getInstance("Onion", sandwich);
sandwich = OtherIngredientFactory.getInstance("Tomato", sandwich);
sandwich = OtherIngredientFactory.getInstance("other", sandwich);
```

- 결과

```
Sandwich,WholeGrainBread,Beef,SwissCheese,Lettuce,Onion,Tomato 10000
```

Observer Pattern

- 구독, 구독 취소, 구독자 알림 메서드를 가진 Subject와 update() 메서드를 가진 Observer를 이용해 Observer Pattern을 구현한다.

Subject 인터페이스

- 구독, 구독 취소, 구독자 알림 추상 메서드 보유

```
public interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String orderDetails);
}
```

SandwichOrderSubject

- Subject를 구현한 SandwichOrderSubject는 Observer들의 리스트를 가지고 있으며 실제로 옵저버에 대해 추가, 삭제, 알림 기능을 구현한다.

```
public class SandwichOrderSubject implements Subject {
    // 관찰 대상인 Observer들의 리스트
    public ArrayList<Observer> observers;

    public SandwichOrderSubject() {
        observers = new ArrayList<>();
    }

    // 옵저버 추가
    @Override
    public void addObserver(Observer observer) {
        this.observers.add(observer);
    }

    // 옵저버 삭제
    @Override
    public void removeObserver(Observer observer) {
        if (observers.indexOf(observer) >= 0) {
            observers.remove(observer);
        }
    }

    // 옵저버들에게 알림
    @Override
    public void notifyObservers(String orderDetails) {
        for (Observer o : observers) {
            o.update(orderDetails);
        }
    }
}
```

Observer

- Subject로부터 알림이 오면 그에 따른 update()

```
public interface Observer {
    void update(String orderDetails);
```

```
}
```

Customer

- Concrete Observer

```
public class Customer implements Observer {
    private String name;

    public Customer(String name) {
        this.name = name;
    }

    @Override
    public void update(String orderDetails) {
        System.out.println(name + ", your order is ready: " + orderDetails);
    }
}
```

MainTest

1. Subject 생성, Observer 생성 - 옵저버 패턴
2. Subject에 Observer들(Tom, Jetty) 등록 - 옵저버 패턴
3. 샌드위치 생성 - 팩토리 패턴, 데코레이터 패턴
4. Observer들에게 알림 - 옵저버 패턴
5. Subject에서 Observer들 제거 - 옵저버 패턴
6. Mitty도 2, 3, 4 과정 동일

```
public MainTest() {

    // Create a sandwich order subject
    SandwichOrderSubject s = new SandwichOrderSubject();

    // Subscribe customers "Tom" & "Jerry" to dorder notifications
    Customer tom = new Customer("Tom");
    Customer jerry = new Customer("Jerry");
```

```
    // subscribe
    s.addObserver(tom);
    s.addObserver(jerry);


    // Create sandwich
    Ingredient sandwich = new Sandwich(new GrilledRecipe());
    sandwich = new WholeGrainBread(sandwich);
    sandwich = new Beef(sandwich);
    sandwich = new SwissCheese(sandwich);
    sandwich = OtherIngredientFactory.getInstance("Lettuce", sandwich);
    sandwich = OtherIngredientFactory.getInstance("Onion", sandwich);
    sandwich = OtherIngredientFactory.getInstance("Tomato", sandwich);
    sandwich = OtherIngredientFactory.getInstance("other", sandwich);

    // Notify orderDetails
    String orderDetails = sandwich.getDescription() + " " + sandwich.cost();
    s.notifyObservers(orderDetails);

    // Unsubscribe "Tom" & "Jerry"
    s.removeObserver(tom);
    s.removeObserver(jerry);

    // subscribe "Mitty" to order notifications
    Customer mitty = new Customer("Mitty");
    s.addObserver(mitty);

    // Create sandwich
    Ingredient sandwich2 = new Sandwich(new ColdRecipe());
    sandwich2 = new OatmealBread(sandwich2);
    sandwich2 = new Ham(sandwich2);
    sandwich2 = new AmericanCheese(sandwich2);
    sandwich2 = OtherIngredientFactory.getInstance("Lettuce", sandwich2);
    sandwich2 = OtherIngredientFactory.getInstance("Tomato", sandwich2);
    sandwich2 = OtherIngredientFactory.getInstance("other", sandwich2);

    // notify orderDetails2
    String orderDetails2 = sandwich2.getDescription() + " " + sandwich2.cost();
    s.notifyObservers(orderDetails2);

}
```