МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Інститут ІКНІ

Кафедра Систем Штучного Інтелекту

Звіт

Про проходження курсу «Design Pattern»

Виконав:

Студент групи КН-209

Кіндрат Володимир

Викладач:

Шамуратов О.В.

Львів – 2020

Factory Pattern

Фабричний метод — це породжувальний шаблон проектування, який визначає загальний інтерфейс для створення об’єктів у суперкласі, дозволяючи підкласам змінювати тип створюваних об’єктів. В даному випадку ми використаємо цей шаблон проектування, адже він дозволить зробити код створення об’єктів більш універсальним та ефективніше працювати з багатьма подібними класами (Складами, де зберігаються продукти) та не дублювати код при їх створенні.

Warehouse class

```
package Project;
import Project.Main;
import java.util.List;

public abstract class Warehouse {
    public void getProduct(String product, int count){
        System.out.println("To warehouse brought " + product+ " product in "+ count + "
count");
    }
    public int CountProduct(){
        System.out.println("In warehouse items");
        return 5;
    }
    public void TakeOutProduct(String product){
        System.out.println("From warehouse take out " + product+" product");
    }
}
```

FoodWarehouse class

```
package Project;

import java.util.List;

public class FoodWarehouse extends Warehouse {
    List<String> ItemList;
}
```

ClosesWarehouse class

```
package Project;

import java.util.List;

public class ClosesWarehouse extends Warehouse {
    List<String> ItemList;
}
```

CleaningWarehouse class

```java
package Project;

import java.util.List;

public class CleaningWarehouse extends Warehouse{
    List <String> ItemList;
}
```

Store class

```java
package Project;

public class Store {

    String fruit = "Banana";
    String fruit1 = "Carrot";
    String fruit2 = "Rasbery";
    String vegetable = "Tomato";
    String closes = "TShirt";

    public void getOutProduct() {
        FoodWarehouse foodWarehouse = new FoodWarehouse();
        foodWarehouse.TakeOutProduct(fruit1);
    }
}
```

Main class

```java
package Project;

public class Main {
    public static FoodWarehouse foodWarehouse = new FoodWarehouse();
    static String item  = "Carrot";
    static int count = 13;
    public static void main(String[] args) {
        Main main = new Main();
        main.setItemToWharehouse(item, count);
        foodWarehouse.CountProduct();
        foodWarehouse.TakeOutProduct(item);
    }

    public void setItemToWharehouse(String nameItem,int countItem ){
        foodWarehouse.getProduct(nameItem, countItem);
    }
}
```

Результат роботи програми:

```
To warehouse brought Carrot product in 13 count
In warehouse items
From warehouse take out Carrot product
```

Adapter Pattern

Адаптер — це структурний патерн проектування, що дає змогу об’єктам із несумісними інтерфейсами працювати разом. В даному випадку ми використаємо даний шаблон, аби зробити прийом продуктів на склад більш універсальним. Уявимо, що різні компанії використовують різні фомати звітності для товару, які рібно розмістити на складі, щоб не виникало конфліктів ми створимо клас Адаптер, який дозволить об’єктам з різними інтерфейсами працювати разом.

ClosesCompany class

```
package Project;

public class ClosesCompany {
    char closes1 = 'T';
    char closes2 = 'B';
    char closes3 = 'H';
    Adapter adapter = new Adapter();

    public void setAdapter(Adapter adapter) {
        this.adapter = adapter;
    }
    public void setItem(){
        adapter.addItem(closes1);
        adapter.addItem(closes2);
    }
}
```

FoodCompany class

```
package Project;

public class FoodCompany {
    Adapter adapter = new Adapter();
    String[] items = {"Onion", "Banana", "Tomato"};

    public void setAdapter(Adapter adapter) {
        this.adapter = adapter;
    }
    public void setItem(){
        adapter.addItem(items);
    }
}
```

Warehouse class

```
package Project;

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

public class Warehouse {
    public ArrayList<String> ItemList = new ArrayList<String>();
    private int counter = 0;
    public void addItem(String item){
        this.ItemList.add(counter, item);
        counter++;
```

```
        System.out.println(item +" was plased at warehouse");
    }
    public void getItems(){
        System.out.println("Now at warehouse:");
        for(String item : ItemList){
        System.out.print(" "+ item);
    }
    }
}
```

Adapter class

```
package Project;

import java.util.List;

public class Adapter {
    public String newItem;
    Warehouse warehouse = new Warehouse();
    public void addItem(char item){
        if(item == 'T')
            newItem = "Trousers";
        if(item == 'B')
            newItem = "Boots";
        if(item == 'H')
            newItem = "Hat";
        warehouse.addItem(newItem);
    }

    public void addItem(String[] strArray){
        int counter = 0;
        while(counter != 3){
            warehouse.addItem(strArray[counter]);
            counter++;
        }
    }
}
```

Main class

```
package Project;

public class Main {
    public static void main(String[] args){
    ClosesCompany closesCompany = new ClosesCompany();
    FoodCompany foodCompany = new FoodCompany();
    Warehouse warehouse = new Warehouse();
     closesCompany.setItem();
     foodCompany.setItem();
    }
}
```

Результат роботи програми:

```
Trousers was plased at warehouse
Boots was plased at warehouse
Onion was plased at warehouse
Banana was plased at warehouse
Tomato was plased at warehouse
```

Observer Pattern

Спостерігач — це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об'єктам стежити й реагувати на події, які відбуваються в інших об'єктах. Одним з випадків використання є, якщо один об'єкт повинен передавати повідомлення іншим об'єктам, але при цьому не може, або не повинен знати про їх налаштуванні. Я використаю цей патерн для реалізації коректного сповіщення магазину про наявність товару на складі, або його відсутність.

ClosesWarehouse class

```
package Project;

import java.util.ArrayList;

public class ClosesWarehouse {
    Observer observer = new Observer();
    public ArrayList<String> ItemList = new ArrayList<>();
    private int counter = 0;

    public void addItem(String item, int count){
        this.ItemList.add(counter, item);
        counter++;
    }
}
```

FoodWarehouse class

```
package Project;

import java.util.ArrayList;

public class FoodWarehouse {
    Observer observer = new Observer();
    public ArrayList<String> ItemList = new ArrayList<>();
    private int counter = 0;

    public void addItem(String item, int count){
        this.ItemList.add(counter, item);
        counter++;
    }
}
```

Company class

```
package Project;

public class Company {
    private int count;
    public String closes1 = "Tshirt";
    public String closes2 = "Trousers";
    public String closes3 = "Hat";
    public String food1 = "Hat";
    public String food2 = "Hat";
```

```
    Observer observer = new Observer();
    ClosesWarehouse closesWarehouse = new ClosesWarehouse();
    FoodWarehouse foodWarehouse = new FoodWarehouse();
    public void storeFood (String item, int count){
        closesWarehouse.addItem(item, count);
        observer.doNotification(item,count);
    }
    public void storeCloses(String item, int count){
        foodWarehouse.addItem(item, count);
        observer.doNotification(item, count);
    }
}
```

Observer class

```
package Project;

public class Observer {

    public void doNotification(String str, int count){
        System.out.println("To warehouse added " + str+ " in " + count+ " count");
    }
}
```

Storage class

```
package Project;

import java.util.ArrayList;

public class Storage {
    ArrayList <String> StorageItems = new ArrayList<>();
    public void Storage(String item){
        StorageItems.add(item);

    }
}
```

Main class

```
package Project;

public class Main {
    public static void main(String[] args){
        Company company = new Company();
        company.storeCloses(company.closes1, 21);
        company.storeCloses(company.closes2, 4);
        company.storeFood(company.food2, 120);
    }
}
```

Висновок:

Під час проходження курсу “Design patterns” я навчився використовувати шаблони проектування на практиці та дослідив їх більш поглиблено, що дало змогу витрачати менше часу, використовуючи готові рішення для певних класів проблем, також це дозволило оптимізувати код моїх програм та підвищити рівень його написання.