МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет ім. М. Є. Жуковського

«Харківський авіаційний інститут»

факультет програмної інженерії та бізнесу

кафедра інженерії програмного забезпечення

**Розрахунково-графічна робота**

з дисципліни «**Архітектура та проектування програмного забезпечення**»

*назва дисципліни*

на тему: «Шаблони проектування»

Виконав: студент 2 курсу групи № 621пмб

напряму підготовки (спеціальності)

121 інженерія программного забезпечення

(шифр і назва напряму підготовки (спеціальності))

Рахліс Д.С.

(прізвище й ініціали студента)

Прийняв: доцент Лучшев П.О.

(посада, науковий ступінь, прізвище й ініціали)

Національна шкала:

Кількість балів:

Оцінка ECTS:

Харків – 2023

**Зміст**

[**1 Мета роботи** 3](#_gjdgxs)

[**2 Постановка задачі** 3](#_1fob9te)

[**3 Система оцінювання** 3](#_2et92p0)

[**4 Порядок виконання роботи** 4](#_3dy6vkm)

[**4.1 Creational pattern – Builder** 4](#_1t3h5sf)

[**4.1.1 Опис шаблону Builder** 4](#_4d34og8)

[**4.1.2 Опис реалізації шаблону Builder мовлю C#** 6](#_2s8eyo1)

[**4.1.3 Опис реалізації шаблону Builder мовлю Java** 10](#_17dp8vu)

[**4.2 Structural pattern – Bridge** 14](#_3rdcrjn)

[**4.2.1 Опис шаблону Bridge** 14](#_26in1rg)

[**4.2.2 Опис реалізації шаблону Bridge мовлю C#** 16](#_lnxbz9)

[**4.2.3 Опис реалізації шаблону Bridge мовлю Java** 19](#_35nkun2)

[**4.3 Behavioral pattern – Command pattern** 21](#_1ksv4uv)

[**4.3.1 Опис шаблону Command pattern** 21](#_44sinio)

[**4.3.2 Опис реалізації шаблону Command pattern мовлю C#** 23](#_2jxsxqh)

[**4.3.3 Опис реалізації шаблону Command pattern мовлю Java** 26](#_z337ya)

[**4.4 Concurrency pattern – Balking pattern** 28](#_3j2qqm3)

[**4.4.1 Опис шаблону Balking pattern** 28](#_1y810tw)

[**4.4.2 Опис реалізації шаблону Balking pattern мовлю Java** 29](#_4i7ojhp)

[**Висновок** 32](#_2xcytpi)

**1 Мета роботи**

Вивчення стандартних ситуацій в процесі розробки складних програмних проектів та використання шаблонів (Design patterns) проектування для їх вирішення.

**2 Постановка задачі**

Самостійно знайти за допомогою мережі Інтернет (відповідно до варіанта завдання) опис шаблонів проектування (Design patterns) наступних типів:

* породжує (Creational patterns);
* структурного (Structural patterns);
* поведінкового (Behavioral pattern);
* паралельних обчислень (Concurrency pattern).

На підставі зібраного матеріалу, представити текстовий опис шаблона і його основних UML-моделей:

* статичну (діаграма класів і / або діаграма модулів);
* динамічну модель (діаграма взаємодії і / або стану);

На основі зібраного матеріалу розробити програмний проект, який демонструє особливості застосування заданих шаблонів проектування на практиці.

Для завдання підвищеного рівня складності (Таблиця 3.1) необхідно шаблон проектування (design pattern) оформити у вигляді zip-файлу, який є шаблоном проекту або елемента (Project / Item Tempalte) для середовища розробки Visual Studio [1-3].

**3 Система оцінювання**

|  |  |  |  |
| --- | --- | --- | --- |
| Таблиця 3.1 | | | |
| № | Складність | Вимоги | Бал |
| 1 | Базовий рівень | Текстовий опис шаблона, його основних складових частин, їх призначення | 5 |
| 2 | UML модель шаблону виконана в Visio, Visual Studio 2015 (Modeling project) або в іншому спеціалізованому редакторі (окремим файлом / проектом) | 5 |
| 3 | Програмний проект на С #, який демонструє специфіку шаблону проектування (за кожен шаблон) | 5 |
| 4 | Реалізація шаблону проектування на двох і більше мовами програмування | 5 |
| 5 | Підвищений рівень | Створення шаблону проектування (design pattern) у вигляді шаблону проекту або елемента (project / item template) для Visual Studio (zip-файл) | 5 |

**4 Порядок виконання роботи**

**Варіант 16**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Creational pattern** | **Structural pattern** | **Behavioral pattern** | **Concurrency pattern** |
| 3.12 | Builder | Bridge | Command pattern | Balking pattern |

**4.1 Creational pattern – Builder**

**4.1.1 Опис шаблону Builder**

Будівельник — це породжувальний патерн проектування, що дає змогу створювати складні об’єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображень об’єктів.

Патерн Будівельник пропонує винести конструювання об’єкта за межі його власного класу, доручивши цю справу окремим об’єктам, які називаються будівельниками.

Патерн пропонує розбити процес конструювання об’єкта на окремі кроки (наприклад, побудуватиСтіни, встановитиДвері і т. д.) Щоб створити об’єкт, вам потрібно по черзі викликати методи будівельника. До того ж не потрібно викликати всі кроки, а лише ті, що необхідні для виробництва об’єкта певної конфігурації.

Зазвичай один і той самий крок будівництва може відрізнятися для різних варіацій виготовлених об’єктів. Наприклад, дерев’яний будинок потребує будівництва стін з дерева, а кам’яний — з каменю.

У цьому випадку ви можете створити кілька класів будівельників, які по-різному виконуватимуть ті ж самі кроки. Використовуючи цих будівельників в одному й тому самому будівельному процесі, ви зможете отримувати на виході різні об’єкти.

Ви можете піти далі та виділити виклики методів будівельника в окремий клас, що називається «Директором». У цьому випадку директор задаватиме порядок кроків будівництва, а будівельник — виконуватиме їх.

Окремий клас директора не є суворо обов’язковим. Ви можете викликати методи будівельника і безпосередньо з клієнтського коду. Тим не менш, директор корисний, якщо у вас є кілька способів конструювання продуктів, що відрізняються порядком і наявними кроками конструювання. У цьому випадку ви зможете об’єднати всю цю логіку в одному класі.

Така структура класів повністю приховає від клієнтського коду процес конструювання об’єктів. Клієнту залишиться лише прив’язати бажаного будівельника до директора, а потім отримати від будівельника готовий результат.

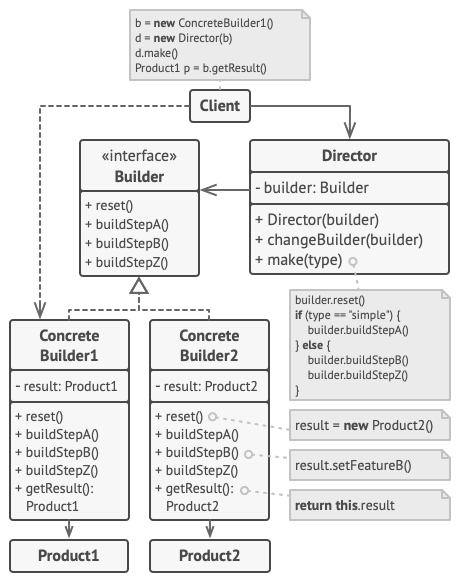


Рисунок 4.1.1.1 – Загальна UML діаграма патерну Builder

На рисунку 4.1.1.1 зображена загальна структура використання патерну Builder. Інтерфейс будівельника оголошує кроки конструювання продуктів, спільні для всіх видів будівельників. Конкретні будівельники реалізують кроки будівництва, кожен по-своєму. Конкретні будівельники можуть виготовляти різнорідні об’єкти, що не мають спільного інтерфейсу. Продукт — об’єкт, що створюється. Продукти, зроблені різними будівельниками, не зобов’язані мати спільний інтерфейс. Директор визначає порядок виклику кроків будівельників, необхідних для виробництва продуктів тієї чи іншої конфігурації. Зазвичай Клієнт подає до конструктора директора вже готовий об’єкт-будівельник, а директор надалі використовує тільки його. Але можливим є також інший варіант, коли клієнт передає будівельника через параметр будівельного методу директора. У такому випадку можна щоразу використовувати різних будівельників для виробництва різноманітних відображень об’єктів.

**4.1.2 Опис реалізації шаблону Builder мовлю C#**

Нижче приведено діаграму класів отриману в результаті реалізації шаблону Builder (рисунок 4.1.2.1).

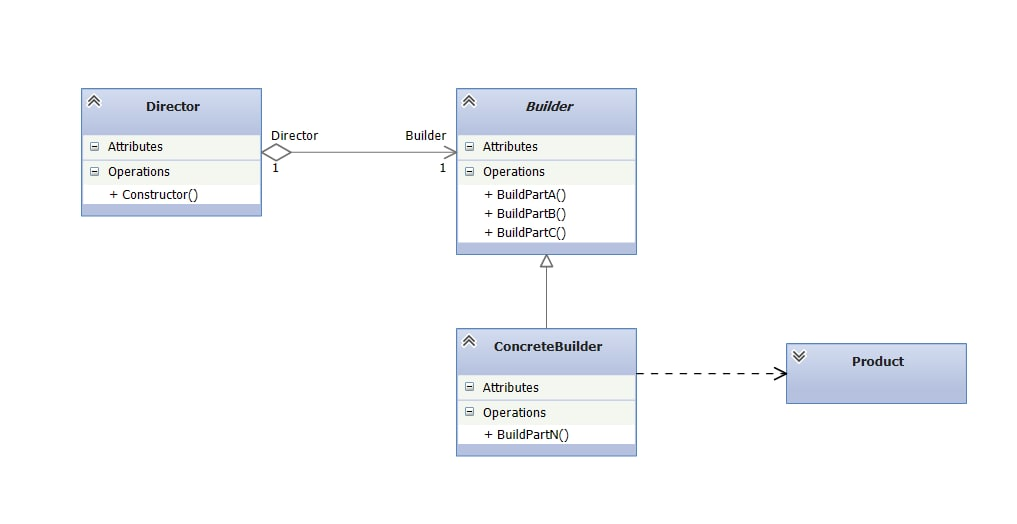


Рисунок 4.1.2.1 – Діаграма класів шаблону Builder

Для вирішення завдання згідно індивідуального варіанту у середовищі Visual Studio 2019 мною було розроблено програмний код який приведено нижче.

**Простір імен: BuilderConsoleDemo**

**Файл: Bottle.cs**

using System;

namespace BuilderConsoleDemo

{

public class Bottle : Packing {

public String pack() {

return "Bottle";

}

}

}

**Файл: Burger.cs**

namespace BuilderConsoleDemo

{

public abstract class Burger : Item {

public abstract string name();

public Packing packing() {

return new Wrapper();

}

public abstract float price();

}

}

**Файл: ChickenBurger.cs**

using System;

namespace BuilderConsoleDemo

{

public class ChickenBurger : Burger {

public override float price() {

return 50.5f;

}

public String name() {

return "Chicken Burger";

}

}

}

**Файл: Coke.cs**

using System;

namespace BuilderConsoleDemo

{

public class Coke : ColdDrink {

public override float price() {

return 30.0f;

}

public String name() {

return "Coke";

}

}

}

**Файл: ColdDrink.cs**

namespace BuilderConsoleDemo

{

public abstract class ColdDrink : Item {

public abstract string name();

public Packing packing() {

return new Bottle();

}

public abstract float price();

}

}

**Файл: Item.cs**

using System;

namespace BuilderConsoleDemo

{

public interface Item {

public String name();

public Packing packing();

public float price();

}

}

**Файл: Meal.cs**

using System;

using System.Collections.Generic;

namespace BuilderConsoleDemo

{

public class Meal

{

private List<Item> items = new List<Item>();

public void addItem(Item item)

{

items.add(item);

}

public float getCost()

{

float cost = 0.0f;

foreach (Item item in items)

{

cost += item.price();

}

return cost;

}

public void showItems()

{

foreach (Item item in items)

{

Console.WriteLine("Item : " + item.name());

Console.WriteLine(", Packing : " + item.packing().pack());

Console.WriteLine(", Price : " + item.price());

}

}

}

}

**Файл: Packing.cs**

using System;

namespace BuilderConsoleDemo

{

public interface Packing {

public String pack();

}

}

**Файл: Pepsi.cs**

using System;

namespace BuilderConsoleDemo

{

public class Pepsi : ColdDrink {

public override float price() {

return 35.0f;

}

public String name() {

return "Pepsi";

}

}

}

**Файл: VegBurger.cs**

using System;

namespace BuilderConsoleDemo

{

public class VegBurger : Burger {

public override float price() {

return 25.0f;

}

public String name() {

return "Veg Burger";

}

}

}

**Файл: Wrapper.cs**

using System;

namespace BuilderConsoleDemo

{

public class Wrapper : Packing

{

public String pack()

{

return "Wrapper";

}

}

}

Нижче приведено результат роботи програми (рисунок 4.1.2.2).

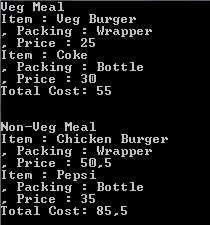


Рисунок 4.1.2.2 – Результат роботи програмної реалізації шаблону Builder мовою C#

**4.1.3 Опис реалізації шаблону Builder мовлю Java**

Нижче приведено діаграму класів отриману в результаті реалізації шаблону Builder (рисунок 4.1.3.1).

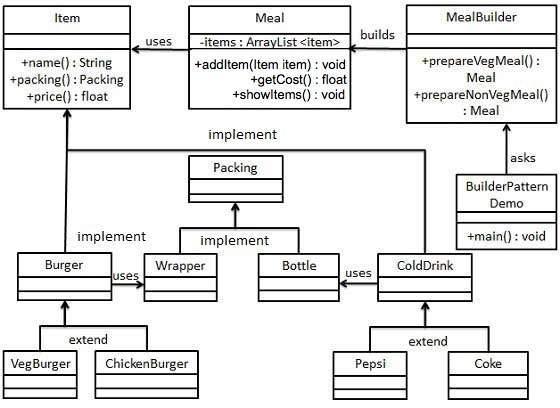


Рисунок 4.1.3.1 – Діаграма класів шаблону Builder

Для вирішення завдання згідно індивідуального варіанту у середовищі Eclipse Jee 2019-06 мною було розроблено програмний код який приведено нижче.

**Файл: Bottle.java**

**public** **class** Bottle **implements** Packing {

@Override

**public** String pack() {

**return** "Bottle";

}

}

**Файл: BuilderPatternDemo.java**

**public** **class** BuilderPatternDemo {

**public** **static** **void** main(String[] args) {

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("\n\nNon-Veg Meal");

nonVegMeal.showItems();

System.***out***.println("Total Cost: " + nonVegMeal.getCost());

}

}

**Файл: Burger.java**

**public** **abstract** **class** Burger **implements** Item {

@Override

**public** Packing packing() {

**return** **new** Wrapper();

}

@Override

**public** **abstract** **float** price();

}

**Файл: ChickenBurger.java**

**public** **class** ChickenBurger **extends** Burger {

@Override

**public** **float** price() {

**return** 50.5f;

}

@Override

**public** String name() {

**return** "Chicken Burger";

}

}

**Файл: Coke.java**

**public** **class** Coke **extends** ColdDrink {

@Override

**public** **float** price() {

**return** 30.0f;

}

@Override

**public** String name() {

**return** "Coke";

}

}

**Файл: ColdDrink.java**

**public** **abstract** **class** ColdDrink **implements** Item {

@Override

**public** Packing packing() {

**return** **new** Bottle();

}

@Override

**public** **abstract** **float** price();

}

**Файл: Item.java**

**public** **interface** Item {

**public** String name();

**public** Packing packing();

**public** **float** price();

}

**Файл: 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());

}

}

}

**Файл: 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;

}

}

**Файл: Packing.java**

**public** **interface** Packing {

**public** String pack();

}

**Файл: Pepsi.java**

**public** **class** Pepsi **extends** ColdDrink {

@Override

**public** **float** price() {

**return** 35.0f;

}

@Override

**public** String name() {

**return** "Pepsi";

}

}

**Файл: VegBurger.java**

**public** **class** VegBurger **extends** Burger {

@Override

**public** **float** price() {

**return** 25.0f;

}

@Override

**public** String name() {

**return** "Veg Burger";

}

}

**Файл: Wrapper.java**

**public** **class** Wrapper **implements** Packing {

@Override

**public** String pack() {

**return** "Wrapper";

}

}

Нижче приведено результат роботи програми (рисунок 4.1.3.2).

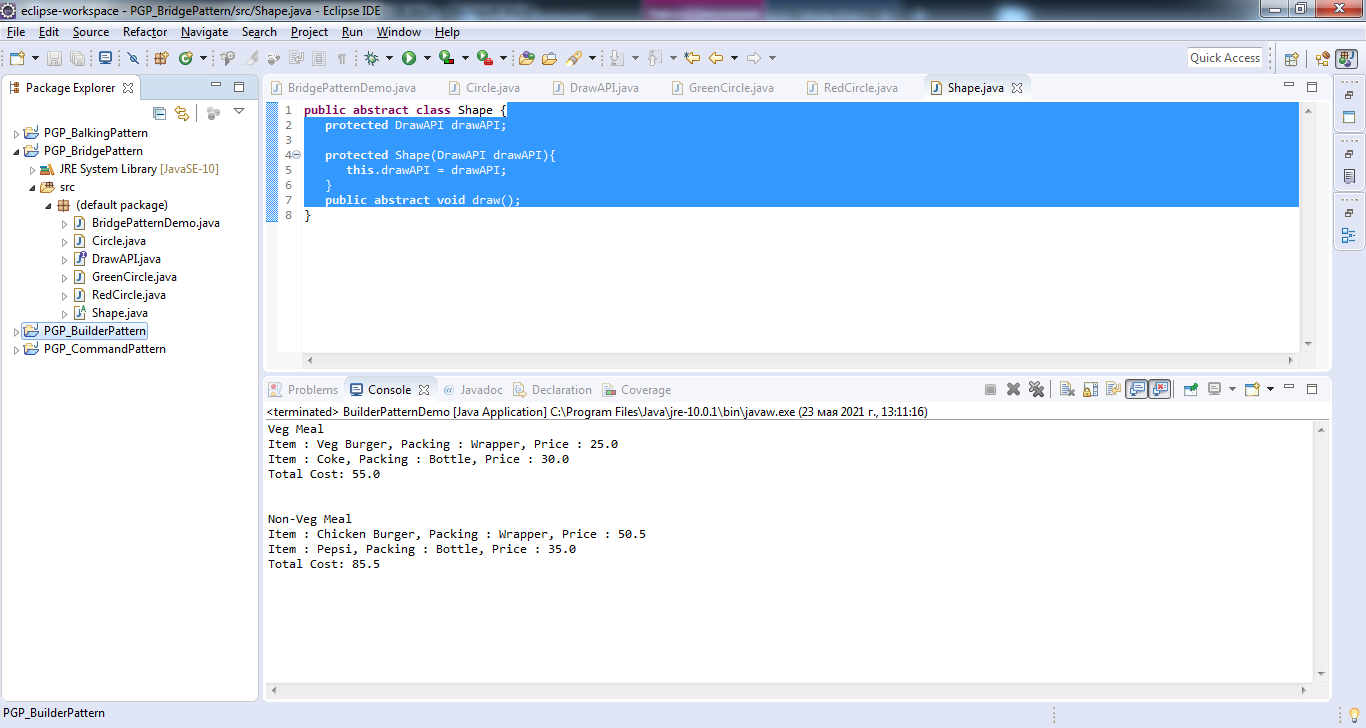


Рисунок 4.1.3.2 – Результат роботи програмної реалізації шаблону Builder мовою Java

**4.2 Structural pattern – Bridge**

**4.2.1 Опис шаблону Bridge**

Міст — це структурний патерн проектування, який розділяє один або кілька класів на дві окремі ієрархії — абстракцію та реалізацію, дозволяючи змінювати код в одній гілці класів, незалежно від іншої.

Патерн Міст пропонує замінити спадкування на делегування. Для цього потрібно виділити одну з таких «площин» в окрему ієрархію і посилатися на об’єкт цієї ієрархії, замість зберігання його стану та поведінки всередині одного класу.

Таким чином, ми можемо зробити Колір окремим класом з підкласами Червоний та Синій. Клас Фігур отримає посилання на об’єкт Кольору і зможе делегувати йому роботу, якщо виникне така необхідність. Такий зв’язок і стане мостом між Фігурами та Кольором. При додаванні нових класів кольорів не потрібно буде звертатись до класів фігур і навпаки.

Якщо говорити про реальні програми, то абстракцією може виступати графічний інтерфейс програми (GUI), а реалізацією — низькорівневий код операційної системи (API), до якого графічний інтерфейс звертається, реагуючи на дії користувача.

Ви можете розвивати програму у двох різних напрямках:

* мати кілька різних GUI (наприклад, для звичайних користувачів та адміністраторів).
* підтримувати багато видів API (наприклад, працювати під Windows, Linux і macOS).

Така програма може виглядати як один великий клубок коду, в якому змішано умовні оператори рівнів GUI та API.

Ми можемо вирішити цю проблему, застосувавши Міст. Патерн пропонує розплутати цей код, розділивши його на дві частини:

* Абстракцію: рівень графічного інтерфейсу програми.
* Реалізацію: рівень взаємодії з операційною системою.

Абстракція делегуватиме роботу одному з об’єктів реалізації. Причому, реалізації можна буде взаємозаміняти, але тільки за умови, що всі вони слідуватимуть єдиному інтерфейсу.

Таким чином, ви зможете змінювати графічний інтерфейс програми, не чіпаючи низькорівневий код роботи з операційною системою. І навпаки, ви зможете додавати підтримку нових операційних систем, створюючи нові підкласи реалізації, без необхідності правити код у класах графічного інтерфейсу.

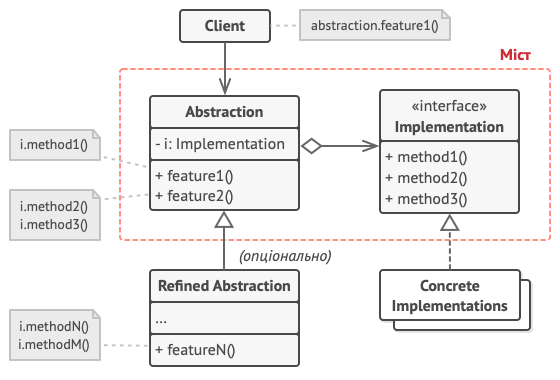


Рисунок 4.2.1.1 – Загальна UML діаграма патерну Builder

На рисунку 4.2.1.1 зображена загальна структура використання патерну Bridge. Абстракція містить керуючу логіку. Код абстракції делегує реальну роботу пов’язаному об’єктові реалізації. Реалізація описує загальний інтерфейс для всіх реалізацій. Всі методи, які тут описані, будуть доступні з класу абстракції та його підкласів. Інтерфейси абстракції та реалізації можуть або збігатися, або бути абсолютно різними. Проте, зазвичай в реалізації живуть базові операції, на яких будуються складні операції абстракції. Конкретні реалізації містять платформо-залежний код. Розширені абстракції містять різні варіації керуючої логіки. Як і батьківский клас, працює з реалізаціями тільки через загальний інтерфейс реалізацій. Клієнт працює тільки з об’єктами абстракції. Не рахуючи початкового зв’язування абстракції з однією із реалізацій, клієнтський код не має прямого доступу до об’єктів реалізації.

**4.2.2 Опис реалізації шаблону Bridge мовлю C#**

Нижче приведено діаграму класів отриману в результаті реалізації шаблону Bridge (рисунок 4.2.2.1).

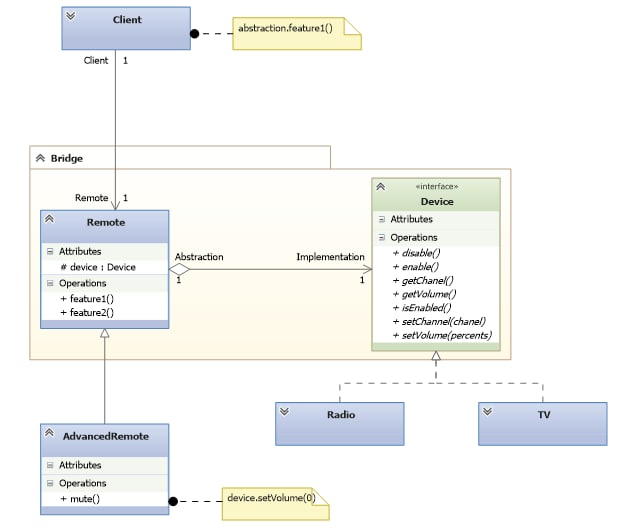


Рисунок 4.2.2.1 – Діаграма класів шаблону Bridge

Для вирішення завдання згідно індивідуального варіанту у середовищі Visual Studio 2019 мною було розроблено програмний код який приведено нижче.

**Простір імен: BridgeDemo**

**Файл: Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace RefactoringGuru.DesignPatterns.Bridge.Conceptual

{

// The Abstraction defines the interface for the "control" part of the two

// class hierarchies. It maintains a reference to an object of the

// Implementation hierarchy and delegates all of the real work to this

// object.

class Abstraction

{

protected IImplementation \_implementation;

public Abstraction(IImplementation implementation)

{

this.\_implementation = implementation;

}

public virtual string Operation()

{

return "Abstract: Base operation with:\n" +

\_implementation.OperationImplementation();

}

}

// You can extend the Abstraction without changing the Implementation

// classes.

class ExtendedAbstraction : Abstraction

{

public ExtendedAbstraction(IImplementation implementation) : base(implementation)

{

}

public override string Operation()

{

return "ExtendedAbstraction: Extended operation with:\n" +

base.\_implementation.OperationImplementation();

}

}

// The Implementation defines the interface for all implementation classes.

// It doesn't have to match the Abstraction's interface. In fact, the two

// interfaces can be entirely different. Typically the Implementation

// interface provides only primitive operations, while the Abstraction

// defines higher- level operations based on those primitives.

public interface IImplementation

{

string OperationImplementation();

}

// Each Concrete Implementation corresponds to a specific platform and

// implements the Implementation interface using that platform's API.

class ConcreteImplementationA : IImplementation

{

public string OperationImplementation()

{

return "ConcreteImplementationA: The result in platform A.\n";

}

}

class ConcreteImplementationB : IImplementation

{

public string OperationImplementation()

{

return "ConcreteImplementationA: The result in platform B.\n";

}

}

class Client

{

// Except for the initialization phase, where an Abstraction object gets

// linked with a specific Implementation object, the client code should

// only depend on the Abstraction class. This way the client code can

// support any abstraction-implementation combination.

public void ClientCode(Abstraction abstraction)

{

Console.Write(abstraction.Operation());

}

}

class Program

{

static void Main(string[] args)

{

Client client = new Client();

Abstraction abstraction;

// The client code should be able to work with any pre-configured

// abstraction-implementation combination.

abstraction = new Abstraction(new ConcreteImplementationA());

client.ClientCode(abstraction);

Console.WriteLine();

abstraction = new ExtendedAbstraction(new ConcreteImplementationB());

client.ClientCode(abstraction);

}

}

}

Нижче приведено результат роботи програми (рисунок 4.2.2.2).

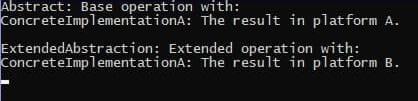


Рисунок 4.2.2.2 – Результат роботи програмної реалізації шаблону Bridge мовою C#

**4.2.3 Опис реалізації шаблону Bridge мовлю Java**

Нижче приведено діаграму класів отриману в результаті реалізації шаблону Bridge (рисунок 4.2.3.1).

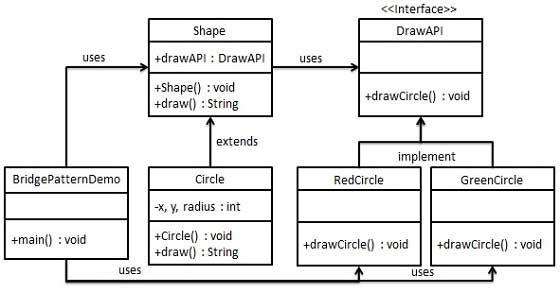


Рисунок 4.2.3.1 – Діаграма класів шаблону Bridge

Для вирішення завдання згідно індивідуального варіанту у середовищі Eclipse Jee 2019-06 мною було розроблено програмний код який приведено нижче.

**Файл: BridgePatternDemo.java**

**public** **class** BridgePatternDemo {

**public** **static** **void** main(String[] args) {

Shape redCircle = **new** Circle(100,100, 10, **new** RedCircle());

Shape greenCircle = **new** Circle(100,100, 10, **new** GreenCircle());

redCircle.draw();

greenCircle.draw();

}

}

**Файл: Circle.java**

**public** **class** Circle **extends** Shape {

**private** **int** x, y, radius;

**public** Circle(**int** x, **int** y, **int** radius, DrawAPI drawAPI) {

**super**(drawAPI);

**this**.x = x;

**this**.y = y;

**this**.radius = radius;

}

**public** **void** draw() {

drawAPI.drawCircle(radius,x,y);

}

}

**Файл: DrawAPI.java**

**public** **interface** DrawAPI {

**public** **void** drawCircle(**int** radius, **int** x, **int** y);

}

**Файл: GreenCircle.java**

**public** **class** GreenCircle **implements** DrawAPI {

@Override

**public** **void** drawCircle(**int** radius, **int** x, **int** y) {

System.***out***.println("Drawing Circle[ color: green, radius: " + radius + ", x: " + x + ", " + y + "]");

}

}

**Файл: RedCircle.java**

**public** **class** RedCircle **implements** DrawAPI {

@Override

**public** **void** drawCircle(**int** radius, **int** x, **int** y) {

System.***out***.println("Drawing Circle[ color: red, radius: " + radius + ", x: " + x + ", " + y + "]");

}

}

**Файл: Shape.java**

**public** **abstract** **class** Shape {

**protected** DrawAPI drawAPI;

**protected** Shape(DrawAPI drawAPI){

**this**.drawAPI = drawAPI;

}

**public** **abstract** **void** draw();

}

Нижче приведено результат роботи програми (рисунок 4.2.3.2).

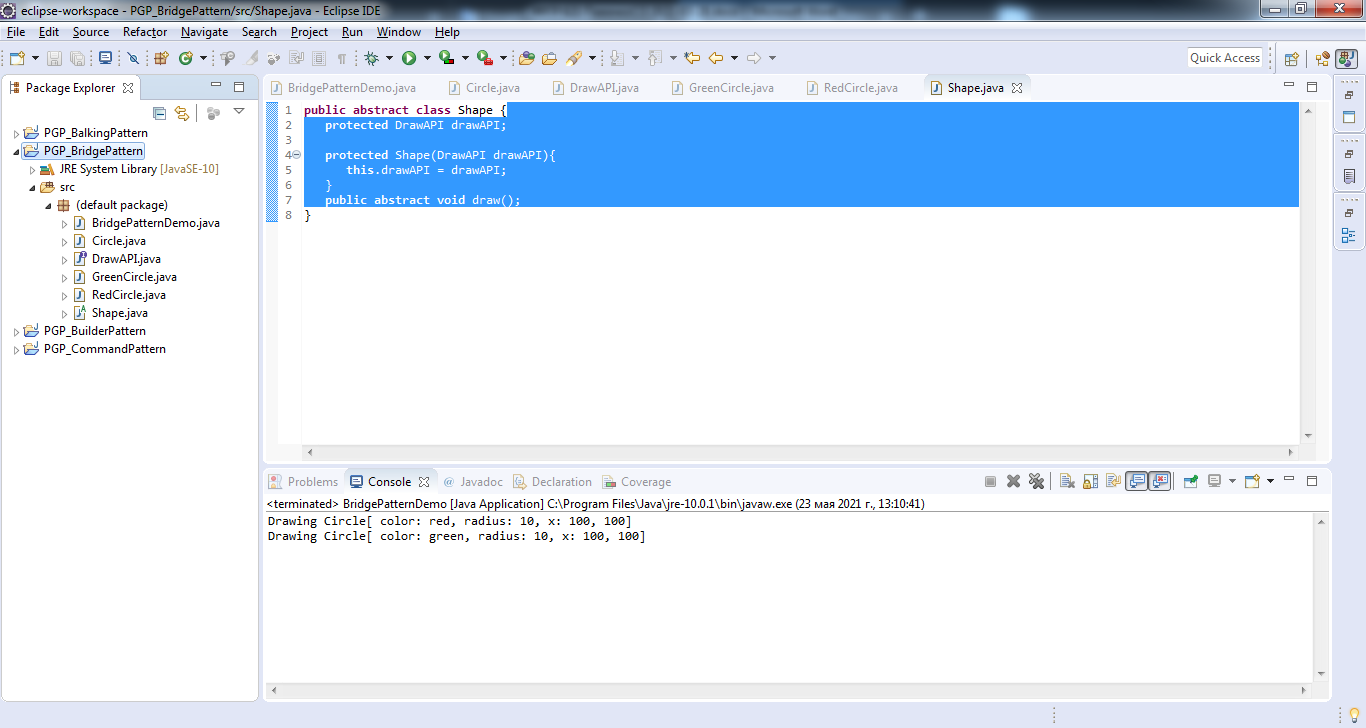


Рисунок 4.2.3.2 – Результат роботи програмної реалізації шаблону Bridge мовою Java

**4.3 Behavioral pattern – Command pattern**

**4.3.1 Опис шаблону Command pattern**

Команда — це поведінковий патерн проектування, який перетворює запити на об’єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій.

Хороші програми зазвичай структурують у вигляді шарів. Найпоширеніший приклад — це шари користувацького інтерфейсу та бізнес-логіки. Перший лише малює гарне зображення для користувача, але коли потрібно зробити щось важливе, інтерфейс користувача «просить» шар бізнес-логіки зайнятися цим.

У дійсності це виглядає так: один з об’єктів інтерфейсу користувача викликає метод одного з об’єктів бізнес-логіки, передаючи до нього якісь параметри.

Патерн Команда пропонує більше не надсилати такі виклики безпосередньо. Замість цього кожен виклик, що відрізняється від інших, слід звернути у власний клас з єдиним методом, який і здійснюватиме виклик. Такий зветься командою.

До об’єкта інтерфейсу можна буде прив’язати об’єкт команди, який знає, кому і в якому вигляді слід відправляти запити. Коли об’єкт інтерфейсу буде готовий передати запит, він викличе метод команди, а та — подбає про все інше.

Класи команд можна об’єднати під загальним інтерфейсом, що має єдиний метод запуску команди. Після цього одні й ті самі відправники зможуть працювати з різними командами, не прив’язуючись до їхніх класів. Навіть більше, команди можна буде взаємозаміняти «на льоту», змінюючи підсумкову поведінку відправників.

Параметри, з якими повинен бути викликаний метод об’єкта одержувача, можна заздалегідь зберегти в полях об’єкта-команди. Завдяки цьому, об’єкти, які надсилають запити, можуть не турбуватися про те, щоб зібрати необхідні дані для одержувача. Навіть більше, вони тепер взагалі не знають, хто буде одержувачем запиту. Вся ця інформація прихована всередині команди.

Таким чином, команди стануть гнучким прошарком між користувацьким інтерфейсом та бізнес-логікою. І це лише невелика частина тієї користі, яку може принести патерн Команда!

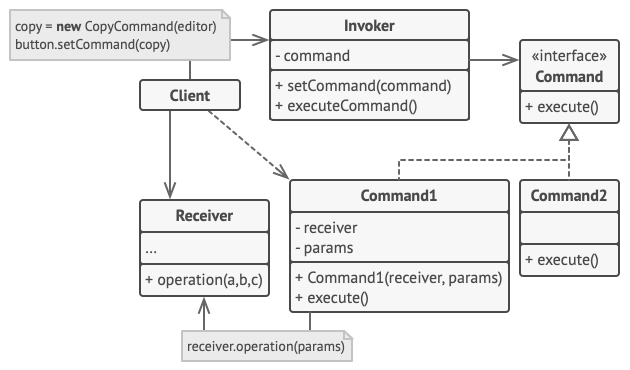


Рисунок 4.3.1.1 – Загальна UML діаграма патерну Command

На рисунку 4.3.1.1 зображена загальна структура використання патерну Command. Відправник зберігає посилання на об’єкт команди та звертається до нього, коли потрібно виконати якусь дію. Відправник працює з командами тільки через їхній загальний інтерфейс. Він не знає, яку конкретно команду використовує, оскільки отримує готовий об’єкт команди від клієнта. Команда описує інтерфейс, спільний для всіх конкретних команд. Зазвичай тут описується лише один метод запуску команди. Конкретні команди реалізують різні запити, дотримуючись загального інтерфейсу команд. Як правило, команда не робить всю роботу самостійно, а лише передає виклик одержувачу, яким виступає один з об’єктів бізнес-логіки. Параметри, з якими команда звертається до одержувача, необхідно зберігати у вигляді полів. У більшості випадків об’єкти команд можна зробити незмінними, передаючи у них всі необхідні параметри тільки через конструктор. Одержувач містить бізнес-логіку програми. У цій ролі може виступати практично будь-який об’єкт. Зазвичай, команди перенаправляють виклики одержувачам, але іноді, щоб спростити програму, ви можете позбутися від одержувачів, «зливши» їхній код у класи команд. Клієнт створює об’єкти конкретних команд, передаючи до них усі необхідні параметри, серед яких можуть бути і посилання на об’єкти одержувачів. Після цього клієнт зв’язує об’єкти відправників зі створеними командами.

**4.3.2 Опис реалізації шаблону Command pattern мовлю C#**

Нижче приведено діаграму класів отриману в результаті реалізації шаблону Command pattern (рисунок 4.3.2.1).

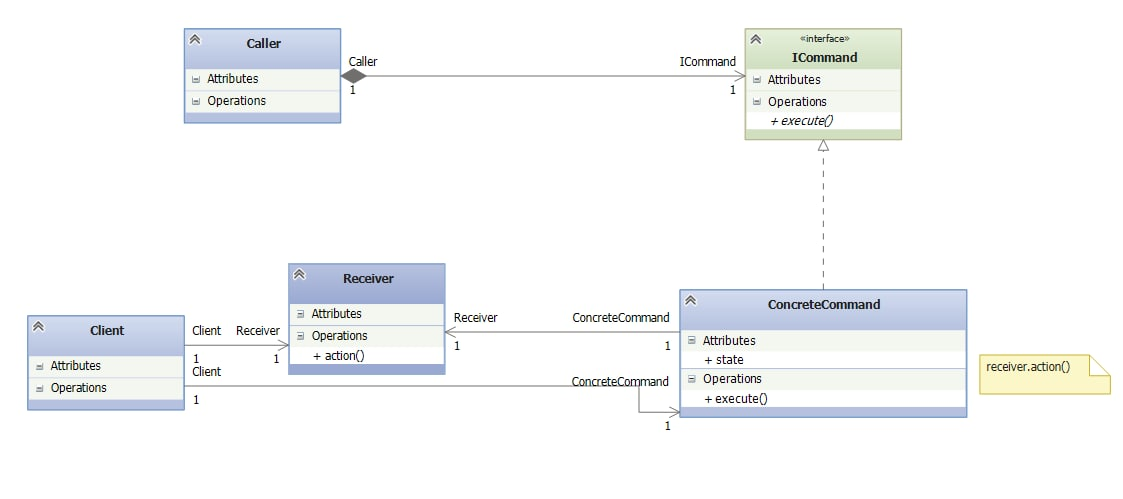


Рисунок 4.3.2.1 – Діаграма класів шаблону Command pattern

Для вирішення завдання згідно індивідуального варіанту у середовищі Visual Studio 2019 мною було розроблено програмний код який приведено нижче.

**Простір імен: CommandDemo**

**Файл: ComplexCommand.cs**

using System;

namespace ComandDemo

{ // However, some commands can delegate more complex operations to other

// objects, called "receivers."

class ComplexCommand : ICommand

{

private Receiver \_receiver;

// Context data, required for launching the receiver's methods.

private string \_a;

private string \_b;

// Complex commands can accept one or several receiver objects along

// with any context data via the constructor.

public ComplexCommand(Receiver receiver, string a, string b)

{

this.\_receiver = receiver;

this.\_a = a;

this.\_b = b;

}

// Commands can delegate to any methods of a receiver.

public void Execute()

{

Console.WriteLine("ComplexCommand: Complex stuff should be done by a receiver object.");

this.\_receiver.DoSomething(this.\_a);

this.\_receiver.DoSomethingElse(this.\_b);

}

}

}

**Файл: ICommand.cs**

namespace ComandDemo

{

// The Command interface declares a method for executing a command.

public interface ICommand

{

void Execute();

}

}

**Файл: Invoker.cs**

using System;

namespace ComandDemo

{

// The Invoker is associated with one or several commands. It sends a

// request to the command.

class Invoker

{

private ICommand \_onStart;

private ICommand \_onFinish;

// Initialize commands.

public void SetOnStart(ICommand command)

{

this.\_onStart = command;

}

public void SetOnFinish(ICommand command)

{

this.\_onFinish = command;

}

// The Invoker does not depend on concrete command or receiver classes.

// The Invoker passes a request to a receiver indirectly, by executing a

// command.

public void DoSomethingImportant()

{

Console.WriteLine("Invoker: Does anybody want something done before I begin?");

if (this.\_onStart is ICommand)

{

this.\_onStart.Execute();

}

Console.WriteLine("Invoker: ...doing something really important...");

Console.WriteLine("Invoker: Does anybody want something done after I finish?");

if (this.\_onFinish is ICommand)

{

this.\_onFinish.Execute();

}

}

}

}

**Файл: Program.cs**

namespace ComandDemo

{

class Program

{

static void Main(string[] args)

{

// The client code can parameterize an invoker with any commands.

Invoker invoker = new Invoker();

invoker.SetOnStart(new SimpleCommand("Say Hi!"));

Receiver receiver = new Receiver();

invoker.SetOnFinish(new ComplexCommand(receiver, "Send email", "Save report"));

invoker.DoSomethingImportant();

}

}

}

**Файл: Receiver.cs**

using System;

namespace ComandDemo

{

// The Receiver classes contain some important business logic. They know how

// to perform all kinds of operations, associated with carrying out a

// request. In fact, any class may serve as a Receiver.

class Receiver

{

public void DoSomething(string a)

{

Console.WriteLine($"Receiver: Working on ({a}.)");

}

public void DoSomethingElse(string b)

{

Console.WriteLine($"Receiver: Also working on ({b}.)");

}

}

}

**Файл: SimpleCommand.cs**

using System;

namespace ComandDemo

{

// Some commands can implement simple operations on their own.

class SimpleCommand : ICommand

{

private string \_payload = string.Empty;

public SimpleCommand(string payload)

{

this.\_payload = payload;

}

public void Execute()

{

Console.WriteLine($"SimpleCommand: See, I can do simple things like printing ({this.\_payload})");

}

}

}

Нижче приведено результат роботи програми (рисунок 4.3.2.2).

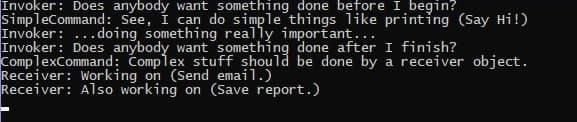


Рисунок 4.3.2.2 – Результат роботи програмної реалізації шаблону Command pattern мовою C#

**4.3.3 Опис реалізації шаблону Command pattern мовлю Java**

Нижче приведено діаграму класів отриману в результаті реалізації шаблону Command pattern (рисунок 4.3.3.1).

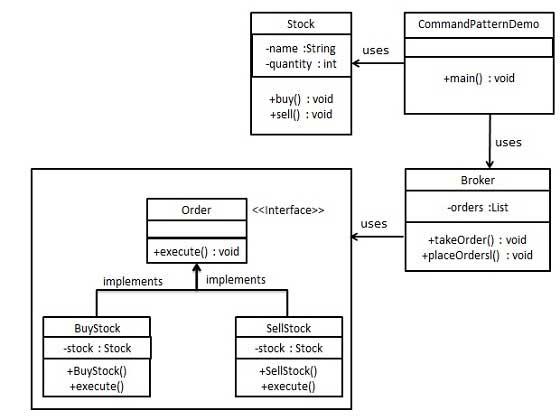


Рисунок 4.3.3.1 – Діаграма класів шаблону Command pattern

Для вирішення завдання згідно індивідуального варіанту у середовищі Eclipse Jee 2019-06 мною було розроблено програмний код який приведено нижче.

**Файл: Broker.java**

import java.util.ArrayList;

import java.util.List;

public class Broker {

private List<Order> orderList = new ArrayList<Order>();

public void takeOrder(Order order){

orderList.add(order);

}

public void placeOrders(){

for (Order order : orderList) {

order.execute();

}

orderList.clear();

}

}

**Файл: BuyStock.java**

**public** **class** BuyStock **implements** Order {

**private** Stock abcStock;

**public** BuyStock(Stock abcStock){

**this**.abcStock = abcStock;

}

**public** **void** execute() {

abcStock.buy();

}

}

**Файл: CommandPatternDemo.java**

**public** **class** CommandPatternDemo {

**public** **static** **void** main(String[] args) {

Stock abcStock = **new** Stock();

BuyStock buyStockOrder = **new** BuyStock(abcStock);

SellStock sellStockOrder = **new** SellStock(abcStock);

Broker broker = **new** Broker();

broker.takeOrder(buyStockOrder);

broker.takeOrder(sellStockOrder);

broker.placeOrders();

}

}

**Файл: Order.java**

**public** **interface** Order {

**void** execute();

}

**Файл: SellStock.java**

**public** **class** SellStock **implements** Order {

**private** Stock abcStock;

**public** SellStock(Stock abcStock){

**this**.abcStock = abcStock;

}

**public** **void** execute() {

abcStock.sell();

}

}

**Файл: Stock.java**

**public** **class** Stock {

**private** String name = "ABC";

**private** **int** quantity = 10;

**public** **void** buy(){

System.***out***.println("Stock [ Name: "+name+", Quantity: " + quantity +" ] bought");

}

**public** **void** sell(){

System.***out***.println("Stock [ Name: "+name+", Quantity: " + quantity +" ] sold");

}

}

Нижче приведено результат роботи програми (рисунок 4.3.3.2).

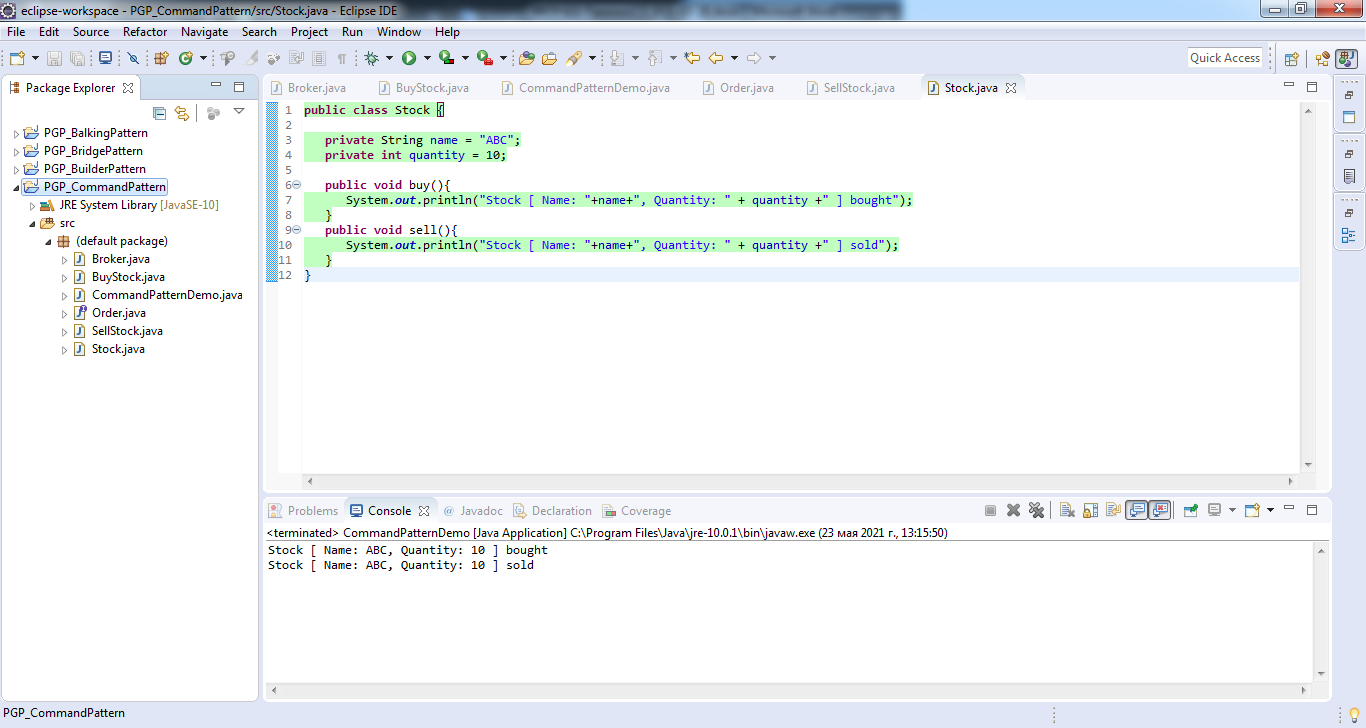


Рисунок 4.3.3.2 – Результат роботи програмної реалізації шаблону Command pattern мовою Java

**4.4 Concurrency pattern – Balking pattern**

**4.4.1 Опис шаблону Balking pattern**

У шаблоні знаходиться програма для створення шаблонів, яка виконує дію над об'єктом тільки тоді, коли об'єкт знаходиться в певному стані. Наприклад, якщо об'єкт читає zip-файли, а викликає метод викликає метод GET для об'єкта, коли zip-файл не відкритий, об'єкт "заблокується" за запитом. Наприклад, в мові програмування Java при таких обставинах може виникнути виключення IllegalStateException.

Є кілька експертів в цій області, які вважають, що відпочинок - це скоріше антипаттерн, ніж графічний дизайн. Якщо об'єкт не підтримує ваш API, він повинен або обмежити API, щоб викликає порушення виклик був недоступний, або щоб виклик міг бути виконаний без обмежень.

Об'єкти, які використовують цей шаблон, як правило, тільки в стані, яке схильне тимчасово відмовлятися, але на невідому кількість часу. Якщо об'єкти повинні залишатися в стані, яке схильне до відмови протягом відомого обмеженого періоду часу, то може бути краща схема захищеної підвіски.

На рисунку 4.4.1.1 зображено UML діаграму прикладу реалізації пральна машина - це об'єкт, який може знаходитися в двох станах: включено і прання. Якщо машина вимкнена, стан змінюється на WASHING, і жоден інший потік не може викликати цю дію для цього, а потім виконати завдання. З іншого боку, якщо він вже був промитий і будь-якої іншої потік виконує wash (), він не може зробити це ще раз і повертається, нічого не роблячи.

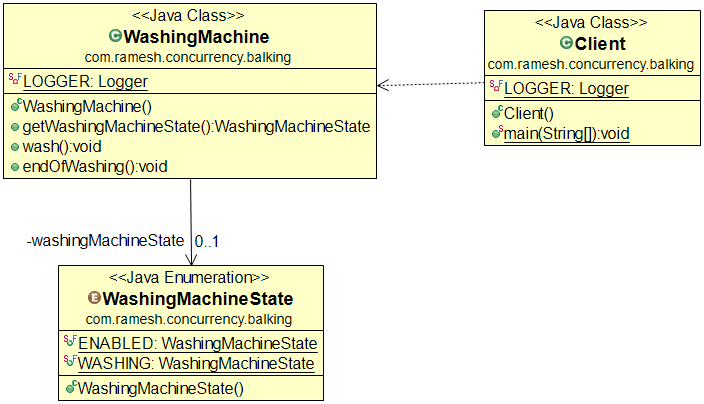


Рисунок 4.4.1.1 – UML діаграма прикладу реалізації патерну Balking

**4.4.2 Опис реалізації шаблону Balking pattern мовлю Java**

Нижче приведено діаграму класів отриману в результаті реалізації шаблону Balking pattern (рисунок 4.4.2.1).

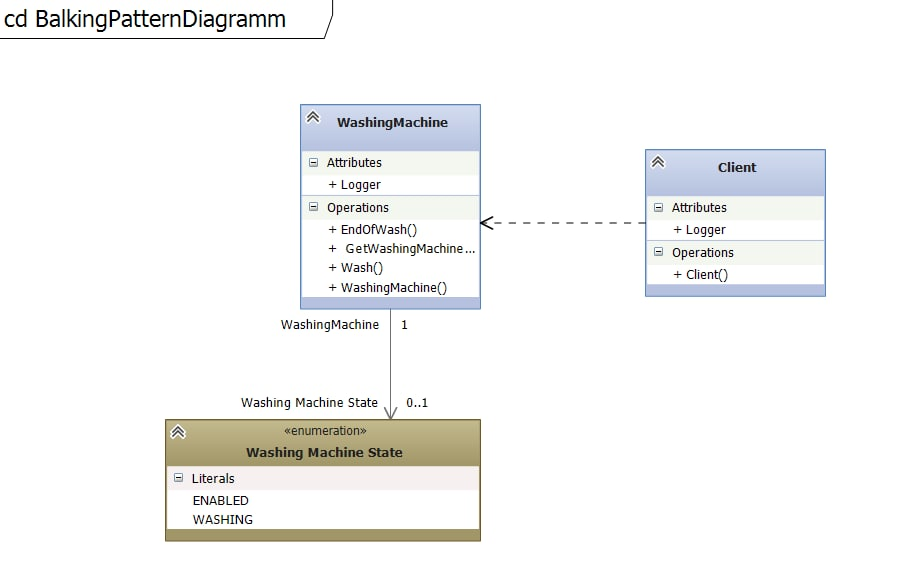


Рисунок 4.4.2.1 – Діаграма класів шаблону Balking pattern

Для вирішення завдання згідно індивідуального варіанту у середовищі Eclipse Jee 2019-06 мною було розроблено програмний код який приведено нижче.

**Файл: ChangerThread.java**

import java.io.IOException;

import java.util.Random;

//Modify thread modify and save

public class ChangerThread extends Thread {

private Data data;

private Random random = new Random();

public ChangerThread(String name, Data data) {

super(name);

this.data = data;

}

public void run() {

try {

for (int i = 0; true; i++) {

data.change("No." + i);

Thread.sleep(random.nextInt(1000));

data.save();

}

} catch (IOException e) {

e.printStackTrace();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

**Файл: Data.java**

import java.io.FileWriter;

import java.io.IOException;

import java.io.Writer;

public class Data {

//file name

private String filename;

//content

private String content;

//Identify whether the data is modified

private boolean changed;

public Data(String filename, String content) {

this.filename = filename;

this.content = content;

this.changed = true;

}

// change the data

public synchronized void change(String newContent) {

content = newContent;

changed = true;

}

// If the data is modified, save it, otherwise return directly

public synchronized void save() throws IOException {

if (!changed) {

System.out.println(Thread.currentThread().getName() + " balks");

return;

}

doSave();

changed = false;

}

private void doSave() throws IOException {

System.out.println(Thread.currentThread().getName() + " calls doSave, content = " + content);

//No need to close this way

try (Writer writer = new FileWriter(filename, true)) {

writer.write(content);

writer.write("\n");

writer.flush();

}

}

}

**Файл: SaverThread.java**

**import** java.io.IOException;

//Similar to automatic save

**public** **class** SaverThread **extends** Thread {

**private** Data data;

**public** SaverThread(String name, Data data) {

**super**(name);

**this**.data = data;

}

**public** **void** run() {

**try** {

**while** (**true**) {

data.save();

Thread.*sleep*(1000);

}

} **catch** (IOException e) {

e.printStackTrace();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**Файл: TestBalking.java**

// test

**public** **class** TestBalking {

**public** **static** **void** main(String[] args) {

Data data = **new** Data("data.txt", "empty");

**new** ChangerThread("Changer", data).start();

**new** SaverThread("Saver", data).start();

}

}

Нижче приведено результат роботи програми (рисунок 4.4.2.2).

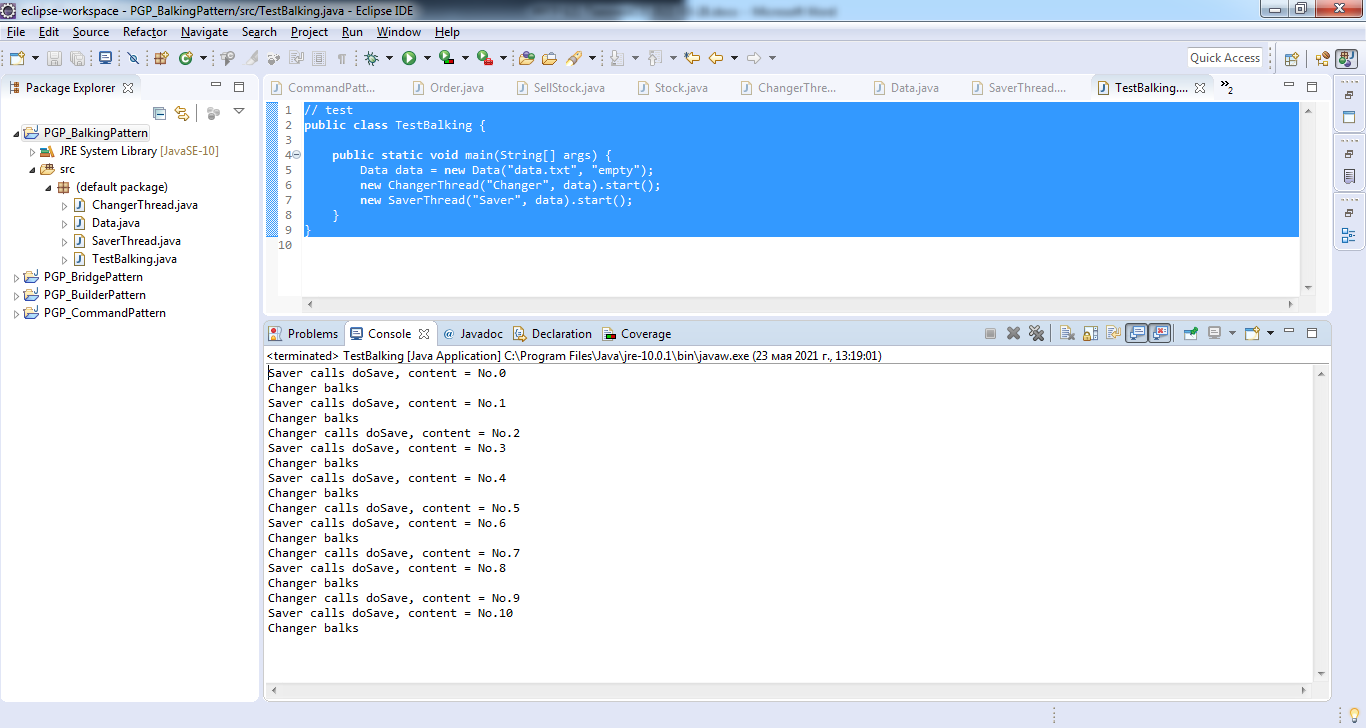


Рисунок 4.4.2.2 – Результат роботи програмної реалізації шаблону Balking pattern мовою Java

**Висновок**

Під час виконання лабораторної роботи ознайомився з стандартними ситуаціями в процесі розробки складних програмних проектів.

Набув практичних навичок використання шаблонів (Design patterns) проектування.

Виконав опис шаблонів програмування запропонованих індивідуальним варіантом.

Розробив програмні проекти мовою C# та Java де реалізував шаблони програмування запропоновані індивідуальним варіантом.