ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«УЖГОРОДСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ»
ФАКУЛЬТЕТ МАТЕМАТИКИ ТА ЦИФРОВИХ ТЕХНОЛОГІЙ

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ З ШАБЛОНІВ ПРОЄКТУВАННЯ. ЧАСТИНА 1

(для студентів спеціальностей 124 «Системний аналіз» та 113 «Прикладна математика»)

Ужгород – 2021

Методичні рекомендації до виконання лабораторних робіт із шаблонів проектування. Частина 1. (для студентів спеціальностей 124 «Системний аналіз» та 113 «Прикладна математика») / Упоряд.: Ю.В. Андрашко. Ужгород: УжНУ, 2021. 36 с

Упорядники:

Андрашко Юрій Васильович, к.т.н., доцент кафедри системного аналізу та теорії оптимізації;

Рецензент: Антосяк Павло Павлович, кандидат фізико-математичних наук, доцент кафедри системного аналізу та теорії оптимізації.

Рекомендовано до друку Науково-методичною комісією факультету математики та цифрових технологій (протокол №5 від 3 березня 2021 року).

Зміст

Вступ

Метою вивчення навчальної дисципліни «Шаблони проєктування» вивчення основних принципів і засади розробки моделей інформаційних систем; сучасні підходи і засоби для проектування програмного забезпечення; застосування шаблонів при розробці складних інформаційних систем. Оволодіння принципами проектування архітектури програмного забезпечення та баз даних; володіння сучасними інструментаріями при розробці архітектури програмного забезпечення; знання та вміння застосовувати шаблони проектування.

Шаблони проєктування (паттерни) – це типова архітектурна конструкція, що представляє собою рішення типової проблеми проектування Використання шаблонів є ефективним способом вирішення задач проектування програмного забезпечення.

Шаблон не можна безпосередньо транслювати в програмний код. Він є зразком вирішення проблеми проектування і відображає відношення між класами та об'єктами, без вказівки на те, як це відношення буде реалізоване. Розрізняють такі типи шаблонів: породжуючі, структурні та поведінкові шаблони.

Породжуючі шаблони – це шаблони проектування, що абстрагують процес створення об’єктів. Вони допомагають зробити систему незалежною від способу створення, композиції та представлення її об'єктів. Шаблон, що створює об'єкти, делегує інстанціювання іншому об'єктові. Ці шаблони важливі, коли система більше залежить від композиції об'єктів, ніж від спадкування класів. Таким чином, замість прямого кодування фіксованого набору поведінок, визначається невеликий набір фундаментальних поведінок, за допомогою композиції яких можна отримувати складніші. Твірні шаблони інкапсулюють знання про конкретні класи, які застосовуються у системі та приховують деталі того, як ці класи створюються і стикуються між собою. Єдина інформація про об'єкти, що відома системі – їхні інтерфейси.

Породжуючі шаблони.

1. Одинак (Singleton).

Проблеми, які вирішує шаблон:

1. Повинен існувати єдиний екземпляр певного об’єкту.
2. Цей об’єкт повинен бути легко доступним з будь-якого місця проекту.

Перша задача зазвичай виникає коли йде доступ до певного ресурсу, наприклад робота з базою даних чи запис логу до файлу. Існування декількох об’єктів, що мають відповідальність доступу до такого ресурсу може спричинити проблеми через доступність ресурсу, а також спричинити не консистентність даних.

Друга задача полягає в спрощенні доступу до об’єкту, адже використовувати звичайний конструктор класу неможливо, а використання глобальних змінних також не завжди є зручним.

Ідея шаблону:

Необхідно зробити конструктор недоступним та створити публічний статичний метод, який і контролюватиме життєвий цикл об’єкта-одинака. Якщо у вас є доступ до класу одинака, то є доступ до цього статичного методу. З якої точки коду ви б його не викликали, він завжди віддаватиме один і той самий об’єкт.

Одинак визначає статичний метод getInstance, який повертає один екземпляр свого класу. Конструктор Одинака повинен бути не публічним для приховування від клієнтів. Виклик методу getInstance це єдиний спосіб отримати об’єкт цього класу (рис 1.).

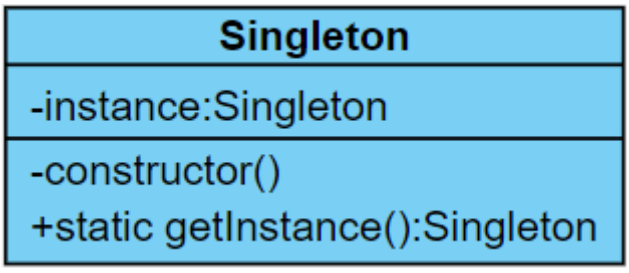

Рис. 1. UML діаграма шаблону одинак

Переваги використання шаблону:

- Вирішує задачу існування єдиної глобальної точки доступу
- Можлива відкладена ініціалізація об’єкта-одинака.

Недоліки використання шаблону:

- Порушує принцип єдиного обов’язку класу.
- Серйозні проблеми в багатопочному коді.
- Необхідність створення Mock-об’єктів при юніт-тестуванні.

Приклад реалізації шаблону одинак на мові C#

```
using System;
namespace Creational
{
    namespace Singleton
    {
        class Singleton
        {
            private static Singleton instance = null;

            private Singleton()
            {
                Random rnd = new Random(DateTime.Now.Millisecond);
                this.randomNumber = rnd.Next();
            }

            public static Singleton getInstance()
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
                return instance;
            }

            public double randomNumber;
            private int counter = 0;

            public void Print()
            {
                Console.WriteLine($"My random number =  {this.randomNumber}");
            }

            public void IncCounter()
            {
                Console.WriteLine($"Counter = {++this.counter}");
```

```
            }
        }
    }
}
```

Приклад тестування шаблону

```
        static void TestSingleton()
        {
            Singleton s1 = Singleton.getInstance();
            Singleton s2 = Singleton.getInstance();
            s1.Print();
            s2.Print();
            s1.IncCounter();
            s2.IncCounter();
            s1.IncCounter();
        }
```

Приклад реалізації майже шаблону одинак на мові JavaScript

```
class Singleton {
    constructor (){
        if (Singleton.instance instanceof Singleton){
            return Singleton.instance;
        }

        this.randomNumber = Math.random();
        this.counter = 0;
        Singleton.instance = this;
    }

    print (){
        console.log(`My random number = ${this.randomNumber}`);
    }

    incCounter(){
        console.log(`Counter =${++this.counter}`);
    }
}

export default Singleton;
```

Приклад тестування шаблону

```
testSingleton: function () {
    let s1 = new Singleton();
    let s2 = new Singleton();
    s1.print();
    s2.print();
    s1.incCounter();
    s2.incCounter();
```

```
    s1.incCounter();
}
```

Приклад реалізації справжнього шаблону одинак на мові JavaScript

```
let singleton = Symbol();
let singletonEnforcer = Symbol();

class Singleton {

    constructor(enforcer) {
        if (enforcer !== singletonEnforcer)
            throw "Instantiation failed: use Singleton.getInstance() instead of new
.";
        this.randomNumber = Math.random();
        this.counter = 0;
    }

    static get _instance() {
        if (!this[singleton])
            this[singleton] = new Singleton(singletonEnforcer);
        return this[singleton];
    }

    static set _instance(v) { throw "Can't change constant property!" }

    static getInstance() { return this._instance; }

    print() {
        console.log(`My random number = ${this.randomNumber}`);
    }

    incCounter() {
        console.log(`Counter =${++this.counter}`);
    }

}

export default Singleton;
```

2. Фабричний метод (Factory method).

Проблему, які вирішує шаблон:

- Необхідно мати можливість в різних місцях створювати об’єкти різних класів (які зазвичай мають спільний базовий клас.) залежно від певних умов, які виникають в процесі функціонування процесу.

Фабричний метод відокремлює код створення об’єктів від решти коду, який використовує ці об’єкти. Завдяки цьому код створення можна розширювати, не зачіпаючи основний код. Щоб додати підтримку нового класу, вам потрібно створити новий підклас та визначити в ньому фабричний метод, повертаючи звідти екземпляр нового класу.

Ідея шаблону:

Фабричний метод пропонує відмовитись від безпосереднього створення об’єктів за допомогою виклику конструктора, замінивши його викликом особливого фабричного методу. Насправді об’єкти все одно будуть створюватися за допомогою виклику конструктору, але робити це буде фабричний метод. На перший погляд це може здатись безглуздим — ми просто перемістили виклик конструктора з одного кінця програми в інший. Проте тепер ви зможете перевизначити фабричний метод у підкласі, щоб змінити тип створюваного продукту. Щоб ця система запрацювала, всі об’єкти, що повертаються, повинні мати спільний інтерфейс. Підкласи зможуть виготовляти об’єкти різних класів, що відповідають одному і тому самому інтерфейсу (рис 2.).

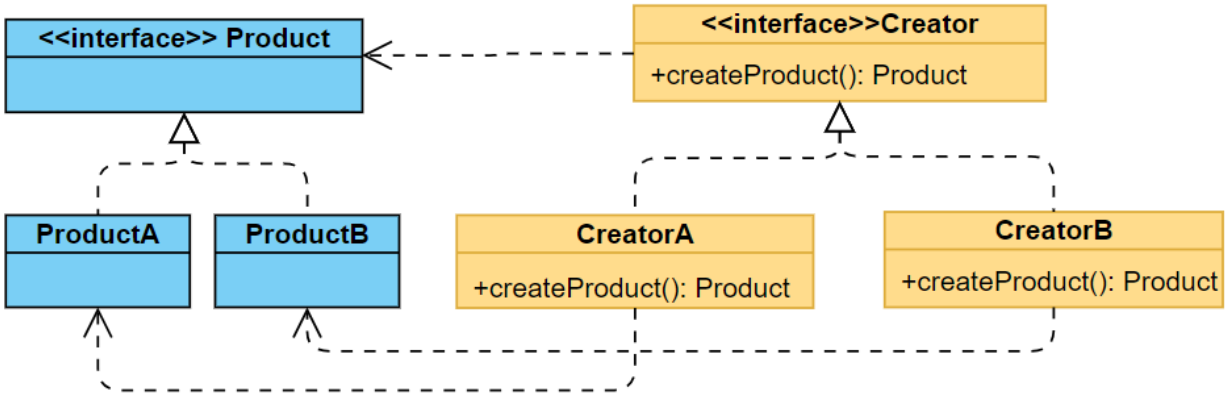

Рис. 2. UML діаграма шаблону Фабричний метод

Переваги використання шаблону:

- Виділяє код створення класів в одне місце.
- Спрощує додавання нових класів до програми.
- Реалізує принцип відкритості/закритості

Недоліки використання шаблону:

- Створює паралельну ієрархію класів.

Приклад реалізації шаблону фабричний метод на мові C#

```
using System.Collections.Generic;

namespace Creational
{
    namespace FactoryMethod
    {
        interface IProduct
        {
            string Operation();
        }
        abstract class ICreator
        {
            public abstract IProduct CreateProduct();
            public List<IProduct> CreateProductList(int count)
            {
                List<IProduct> productList = new List<IProduct>(count);
                for (int i = 0; i < count; i++)
                {
                    productList.Add(this.CreateProduct());
                }
                return productList;
            }
```

```
        }

        class ProductA : IProduct
        {
            public string Operation()
            {
                return "This is A";
            }
        }
        class ProductB : IProduct
        {
            public string Operation()
            {
                return "This is B";
            }
        }

        class ProductC : IProduct
        {
            public string Operation()
            {
                return "This is C";
            }
        }

        class ProductACreator : ICreator
        {
            public override IProduct CreateProduct()
            {
                return new ProductA();
            }
        }

        class ProductBCreator : ICreator
        {
            public override IProduct CreateProduct()
            {
                return new ProductB();
            }

        }

        class ProductCCreator : ICreator
        {
            public override IProduct CreateProduct()
            {
                return new ProductC();
            }
        }
    }
}
```

Приклад тестування шаблону

```
static void TestFabricMethod()
{
    Console.Write("Enter products number:");
    int count = int.Parse(Console.ReadLine());
    Console.Write("Select the product type A, B or C:");
    string choise = Console.ReadLine();
    ICreator productCreator;
    if (choise == "A")
        productCreator = new ProductACreator();
    else if (choise == "B")
        productCreator = new ProductBCreator();
    else if (choise == "C")
        productCreator = new ProductCCreator();
    else
        throw new Exception("Wrong product type");
    List<IProduct> productList = productCreator.CreateProductList(count);
    for (int i = 0; i < count; i++)
    {
        Console.WriteLine(productList[i].Operation());
    }
}
```

Приклад реалізації шаблону фабричного методу мові JavaScript

```
class ProductA
{
    Operation()
    {
        return "This is A";
    }
}

class ProductB
{
    Operation()
    {
        return "This is B";
    }
}

class ProductCreator{
    CreateProduct(){
        return null;
    }

    CreateProductList(count){
        let productList = [];
```

```
        for(let i=0;i<count; i++){
            productList.push(this.CreateProduct())
        }
        return productList;
    }
}

class ProductACreator extends ProductCreator
{
    CreateProduct()
    {
        return new ProductA();
    }
}

class ProductBCreator extends ProductCreator
{
    CreateProduct()
    {
        return new ProductB();
    }
}

export {ProductACreator, ProductBCreator};
```

Приклад тестування шаблону

```
testFactoryMethod: function () {
    console.log("Enter products number");
    let count = parseInt(readLine());
    console.log("Select the product type A or B:");
    let choise = readLine();
    let productCreator;
    if (choise == "A")
        productCreator = new ProductACreator();
    else
        productCreator = new ProductBCreator();
    let productList = productCreator.CreateProductList(count);
    for (let i = 0; i < count; i++) {
        console.log(productList[i].Operation());
    }
},
```

Приклад реалізації шаблону фабричного методу мові JavaScript як функції.

```
class ProductA
{
    Operation()
    {
        return "This is A";
    }
}
```

```
class ProductB
{
    Operation()
    {
        return "This is B";
    }
}

function CreateProduct(product){
    if (product == "A")
        return new ProductA();
    if (product == "B")
        return new ProductB();
}

export default CreateProduct;

   testFactoryMethod2: function () {
        console.log("Enter products number");
        let count = parseInt(readLine());
        console.log("Select the product type A or B:");
        let choise = readLine();
        let productList = [];
        for (let i = 0; i < count; i++) {
            productList.push(CreateProduct(choise));
            console.log(productList[i].Operation());
        }
    },
```

3. Абстрактна фабрика (Abstract Factory).

Проблему, які вирішує шаблон:

- Необхідно мати можливість в різних місцях створювати множини об’єктів різних класів залежно від певних умов, які виникають в процесі функціонування програми. При цьому об’єкти однієї із множин сумісні тільки між собою та не сумісні з об’єктами іншої із множин.

Абстрактна фабрика оголошує список створюючих методів, які клієнтський код може використовувати для отримання тих чи інших різновидів обєктів. Конкретні фабрики відносяться до різних систем і створюють елементи, сумісні з цією системою.

Ідея шаблону:

Абстрактна фабрика приховує від клієнтського коду подробиці того, як і які конкретно об’єкти будуть створені. Внаслідок цього, клієнтський код може працювати з усіма типами створюваних продуктів, так як їхній загальний інтерфейс був визначений заздалегідь (рис 3.).

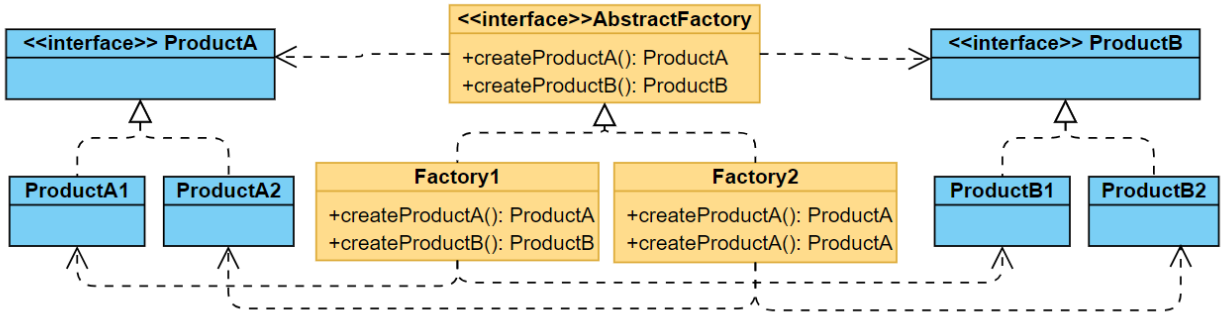

Рис. 3. UML діаграма шаблону Абстрактна фабрика

Переваги використання шаблону:

- Гарантує поєднання створюваних об’єктів.
- Звільняє клієнтський код від прив’язки до конкретних класів.
- Виділяє код створення класів в одне місце.
- Спрощує додавання нових класів до програми.

- Реалізує принцип відкритості/закритості

Недоліки використання шаблону:

- Ускладнює код програми внаслідок введення великої кількості додаткових класів.
- Вимагає наявності всіх типів продукту в кожній варіації.

Приклад реалізації шаблону абстрактна фабрика на мові C#

```
namespace Creational
{
    namespace AbstractFactory
    {
        interface IProductA
        {
            string OperationA();
        }
        interface IProductB
        {
            string OperationB();
            string OperationWithProductA(IProductA product);
        }

        interface IAbstractFactory
        {
            IProductA CreateProductA();
            IProductB CreateProductB();
        }

        class ProductAFirst : IProductA
        {
            public string OperationA()
            {
                return "This is A of first class";
            }
        }

        class ProductASecond : IProductA
        {
            public string OperationA()
            {
                return "This is A of second class";
            }
        }

        class ProductBFirst : IProductB
        {
            public string OperationB()
            {
```

```
            return "This is B of first class";
        }

        public string OperationWithProductA(IProductA product)
        {
            return $"{this.OperationB()} AND {product.OperationA()}";
        }
    }

    class ProductBSecond : IProductB
    {
        public string OperationB()
        {
            return "This is B of second class";
        }

        public string OperationWithProductA(IProductA product)
        {
            return $"{this.OperationB()} AND {product.OperationA()}";
        }
    }

    class FactoryFirstClass : IAbstractFactory
    {
        public IProductA CreateProductA()
        {
            return new ProductAFirst();
        }

        public IProductB CreateProductB()
        {
            return new ProductBFirst();
        }
    }

    class FactorySecondClass : IAbstractFactory
    {
        public IProductA CreateProductA()
        {
            return new ProductASecond();
        }

        public IProductB CreateProductB()
        {
            return new ProductBSecond();
        }
    }
}
```

Приклад тестування шаблону

```
static void TestAbstractFabric()
{
    Console.Write("Select category first or second:");
    int category = int.Parse(Console.ReadLine());
    IAbstractFactory factory;
    if (category == 1)
        factory = new FactoryFirstClass();
    else
        factory = new FactorySecondClass();
    IProductA productA = factory.CreateProductA();
    IProductB productB = factory.CreateProductB();
    Console.WriteLine(productA.OperationA());
    Console.WriteLine(productB.OperationB());
    Console.WriteLine(productB.OperationWithProductA(productA));
}
```

Приклад реалізації шаблону абстрактна фабрика на мові JavaScript

```
class ProductAFirst {
    OperationA() {
        return "This is A of first class";
    }
}

class ProductASecond {
    OperationA() {
        return "This is A of second class";
    }
}

class ProductBFirst {
    OperationB() {
        return "This is B of first class";
    }

    OperationWithProductA(product) {
        return `${this.OperationB()} AND ${product.OperationA()}`;
    }
}

class ProductBSecond {
    OperationB() {
        return "This is B of second class";
    }

    OperationWithProductA(product) {
        return `${this.OperationB()} AND ${product.OperationA()}`;
    }
}
```

```
class FactoryFirstClass {
    CreateProductA() {
        return new ProductAFirst();
    }

    CreateProductB() {
        return new ProductBFirst();
    }
}

class FactorySecondClass {
    CreateProductA() {
        return new ProductASecond();
    }

    CreateProductB() {
        return new ProductBSecond();
    }
}
export {FactoryFirstClass, FactorySecondClass};

   testAbstractFactory: function () {
        console.log("Select category first or second:");
        let category = parseInt(readLine());
        let factory;
        if (category == 1)
            factory = new FactoryFirstClass();
        else
            factory = new FactorySecondClass();
        let productA = factory.CreateProductA();
        let productB = factory.CreateProductB();
        console.log(productA.OperationA());
        console.log(productB.OperationB());
        console.log(productB.OperationWithProductA(productA));
    },
```

Приклад іншого підходу реалізації шаблону абстрактна фабрика на мові JavaScript

```
class ProductAFirst {
    OperationA() {
        return "This is A of first class";
    }
}

class ProductASecond {
    OperationA() {
        return "This is A of second class";
    }
```

```
}

class ProductBFirst {
    OperationB() {
        return "This is B of first class";
    }

    OperationWithProductA(product) {
        return `${this.OperationB()} AND ${product.OperationA()}`;
    }
}

class ProductBSecond {
    OperationB() {
        return "This is B of second class";
    }

    OperationWithProductA(product) {
        return `${this.OperationB()} AND ${product.OperationA()}`;
    }
}

class Factory {
    constructor(type) {
        if (type == "first") {
            this.productA = ProductAFirst;
            this.productB = ProductBFirst;
        }
        else if (type == "second") {
            this.productA = ProductASecond;
            this.productB = ProductBSecond;
        }
        else
            throw `Wrong type of fabric. ${type} is not "first" or "second"`;
    }

    CreateProduct(product) {
        if (product == "A")
            return new this.productA();
        if (product == "B")
            return new  this.productB();
    }
}

export default Factory;

    testFactory: function () {
        console.log("Select category first or second:");
        let category = parseInt(readLine());
        let factory;
        if (category == 1)
```

```
            factory = new Factory("first");
        else
            factory = new Factory("second");
        let productA = factory.CreateProduct("A");
        let productB = factory.CreateProduct("B");
        console.log(productA.OperationA());
        console.log(productB.OperationB());
        console.log(productB.OperationWithProductA(productA));
    },
```

4.

5. Будівельник (Builder)

Проблему, які вирішує шаблон:

- Необхідно створювати об’єкти що мають складний процес ініціалізації, деякі із кроків якої є опціональними та застосовуються не для всіх екземплярів класу.

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

Ідея шаблону:

Процес конструювання об’єкта необхідно розбити на окремі кроки. Для створення об’єкту, потрібно по черзі викликати відповідні методи будівельника. Не потрібно викликати всі кроки, а лише ті, що необхідні для створення об’єкта певної конфігурації. Іноді доцільно 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 обов’язковим. Можна викликати методи будівельника і безпосередньо з клієнтського коду. Тим не менш, директор корисний, якщо у вас є кілька наперед визначених шаблонів 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 (рис 4.).

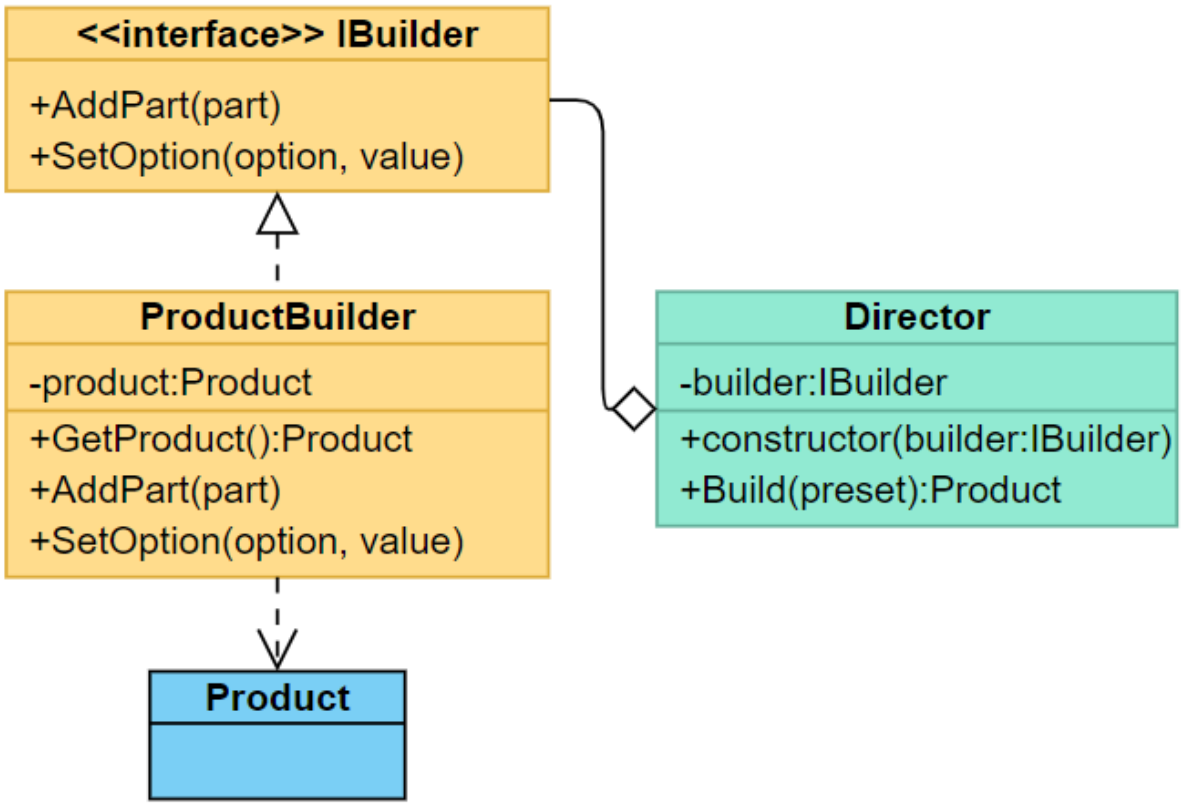

Рис. 4. UML діаграма шаблону Будівельник

Переваги використання шаблону:

- Декомпозиція створення на окремі кроки
- Повторне використання коду.
- Ізоляція створення від клієнтського коду.

Недоліки використання шаблону:

- Ускладнює код програми.
- Прив’язка коду до окремих класів будівельників, адже в інтерфейс неможливо винести метод отримання результату.

Приклад реалізації шаблону абстрактна фабрика на мові C#

```
using System.Collections.Generic;
using System;

namespace Creational
{
    namespace Builder
    {
        class Product
```

```
{
    private List<object> parts = new List<object>();
    public string Name = "No name";
    public void Add(string part)
    {
        this.parts.Add(part);
    }

    public override string ToString ()
    {
        string str = string.Empty;
        foreach (string part in this.parts)
        {
            str += $"\t{part},\n ";
        }
        return $"Product <{this.Name}> parts: \n {str}";
    }
}

interface IBuilder
{
    IBuilder AddPart(object part);
    IBuilder SetDateStemp();
    IBuilder SetName(string name);
}

class Builder : IBuilder
{
    private Product product  = new Product();

    public void Reset()
    {
        this.product = new Product();
    }
    public IBuilder SetName(string name)
    {
        this.product.Name = name;
        return this;
    }

    public IBuilder AddPart (object part)
    {
        this.product.Add(part as string);
        return this;
    }

    public IBuilder SetDateStemp ()
    {
        this.product.Add($"Date stemp: {DateTime.Now.ToString()}");
        return this;
    }
```

```
            public Product GetProduct()
            {
                Product result = this.product;
                this.Reset();
                return result;
            }
        }

        class Director
        {
            private Builder builder;
            public Director(Builder builder)
            {
                this.builder = builder;
            }

            public Product Empty()
            {
                return this.builder.GetProduct();
            }

            public Product BuildFromParts (string[] parts)
            {
                foreach (string part in parts)
                {
                    this.builder.AddPart(part);
                }
                return this.builder.GetProduct();
            }

            public Product Example()
            {
                return this.builder
                        .SetName("Example")
                        .AddPart("Part One")
                        .AddPart("Part Two")
                        .SetDateStemp()
                        .AddPart("Part Three")
                        .GetProduct();
            }
        }
    }
}
```

Приклад тестування шаблону

```
    static void TestBuilder()
    {
        IBuilder builder = new Builder();
```

```
        Product product = builder
                                .SetName("Custom product")
                                .SetDateStemp()
                                .AddPart("Part One")
                                .SetDateStemp()
                                .AddPart("Part Two")
                                .SetDateStemp()
                                .AddPart("Part Three")
                                .GetProduct();
        Console.WriteLine(product.ToString());
        Director director = new Director(builder);
        Console.WriteLine(director.Empty().ToString());
        Console.WriteLine(director.Example().ToString());
        string[] parts = new string[3] { "One", "Two", "Tree" };
        Console.WriteLine(director.BuildFromParts(parts).ToString());
    }
```

Приклад реалізації шаблону абстрактна фабрика на мові JavaScript

```
class Product {
    constructor() {
        this.parts = [];
        this.Name = "No name";
    }

    Add(part) {
        this.parts.push(part);
    }

    toString() {
        let str = "";
        for (let part of this.parts) {
            str += `\t${part},\n `;
        }
        return `Product <${this.Name}> parts: \n ${str}`;
    }
}

class Builder {
    constructor() {
        this.product = new Product();
    }

    Reset() {
        this.product = new Product();
    }

    SetName(name) {
```

```
        this.product.Name = name;
        return this;
    }

    AddPart(part) {
        this.product.Add(part);
        return this;
    }

    SetDateStemp() {
        this.product.Add(`Date stemp: ${new Date(Date.now())}`);
        return this;
    }

    GetProduct() {
        let result = this.product;
        this.Reset();
        return result;
    }
}

class Director {

    constructor(builder) {
        this.builder = builder;
    }

    Empty() {
        return this.builder.GetProduct();
    }

    BuildFromParts(parts) {
        for (let part of parts) {
            this.builder.AddPart(part);
        }
        return this.builder.GetProduct();
    }

    Example() {
        return this.builder
            .SetName("Example")
            .AddPart("Part One")
            .AddPart("Part Two")
            .SetDateStemp()
            .AddPart("Part Three")
            .GetProduct();
    }
}

export { Builder, Director};
```

6. Прототип (Prototype).

Проблему, які вирішує шаблон:

- Необхідно скопіювати об’єкт, зі збереженням його поточного стану, в тому числі і приватних полів.

Не кожен об’єкт вдасться скопіювати у такий спосіб, адже частина його стану може бути приватною,

Ідея шаблону:

Прототип доручає процес копіювання самим об’єктам, які треба скопіювати. Він вводить загальний інтерфейс для всіх об’єктів, що підтримують клонування. Це дозволяє копіювати об’єкти, не прив’язуючись до їхніх конкретних класів. Зазвичай такий інтерфейс має всього один метод clone(). (див рис 5.).

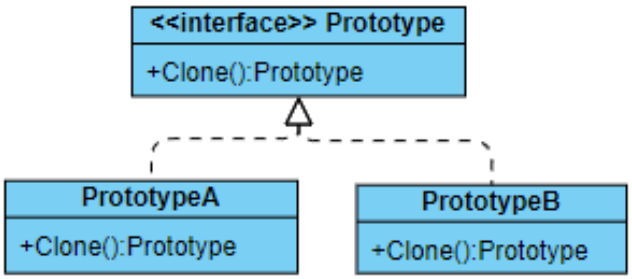

Рис. 5. UML діаграма шаблону Протоип

Переваги використання шаблону:

- Копіювання без прив’язки до класу
- Прискорення створення складних об’єктів.
- Зменшення коду ініціалізації

Недоліки використання шаблону:

- Труднощі з глибоким копіюванням полів що зберігають певні об’єкти які не реалізують інтерфейс клонування.

Приклад реалізації шаблону прототип на мові C#

```
using System;
```

```
namespace Creational
{
    namespace Prototype
    {
        interface IPrototype {
            IPrototype Clone();
        }
        class SomeType: IPrototype
        {
            public string Name = "No name";
            public int Count = 0;

            public IPrototype Clone()
            {
                return this.MemberwiseClone() as SomeType;
            }
        }
        class ProductPrototype: IPrototype
        {
            protected DateTime CreatedAt;
            protected int Id;

            public ProductPrototype()
            {
                this.CreatedAt = DateTime.Now;
                Random rnd = new Random();
                this.Id = rnd.Next();
            }

            public virtual IPrototype Clone()
            {
                return this.MemberwiseClone() as ProductPrototype;
            }
        }

        class CustomProduct : ProductPrototype
        {
            public SomeType obj;
            public CustomProduct(SomeType obj = null) : base()
            {
                this.obj = obj;
            }
            public override CustomProduct Clone()
            {
                CustomProduct clone = this.MemberwiseClone() as CustomProduct;
                clone.obj = this.obj.Clone() as SomeType;
                clone.CreatedAt = DateTime.Now;
                return clone;
                // return this.MemberwiseClone() as CustomProduct; // Deep copy pro
blem
            }
```

```
            public override string ToString()
            {
                return $"{{\n\tId:{this.Id},\n\tCreatedAt:{this.CreatedAt},\n\tobj:
{{ {(this.obj as SomeType).Name} }}\n}}";
            }
        }
    }
}
```

Приклад тестування шаблону

```
static void TestPrototype()
        {
            var p = new SomeType();
            CustomProduct product = new CustomProduct(p);
            CustomProduct productClone = product.Clone();
            (productClone.obj as SomeType).Name = "New name";
            Console.WriteLine(product.ToString());
            Console.WriteLine(productClone.ToString());
        }
```

Приклад реалізації шаблону прототип на мові JavaScript

```
class Prototype{
    clone(){
        return Object.assign({}, this);
    }
}

class SomeType {
    constructor() {
        this.name = "Noname";
        this.count = 0;
    }
}

class ProductPrototype extends Prototype {
    constructor() {
        this.createdAt = new Date(Date.now());
        this.id = Math.random();
    }
}

class CustomProduct extends ProductPrototype {
    constructor(obj) {
        super();
        this.obj = obj;
```

```
    }
    clone() {
        let clone = super.clone();
        clone.obj = Object.assign({}, this.obj);
        clone.createdAt = new Date(Date.now());
        return clone;
    }
}

export { SomeType, ProductPrototype, CustomProduct };
```

7. Умови індивідуальних завдань.

Варіант 1. Розробити підсистему для копіювання події (Дата події, опис, та секретний ключ для online сеансу зв’язку) в календарі на іншу дату. Передбачити можливість існування двох видів подій: Зустріч яка має час початку, час завершення та місце проведення та День народження, яка має контакти іменинника. Обґрунтувати вибір породжуючого патерна .

Основні вимоги: Подія - об'єкт, секретний ключ - приватне поле. Календар - об'єкт що містить список подій та методи: додавання нової події, та вивід всіх подій.

Варіант 2. Розробити підсистему збереження для генерування об’єктів, що вміють виводити інформацію про події (Дата події, опис) в форматі JSON та XML. Передбачити можливість існування двох видів подій: Зустріч, яка має час початку, час завершення та місце проведення та День народження, яка має дату та контакти іменинника. Вибір формату виводу здійснюється 1 раз на початку роботи програми. Обґрунтувати вибір породжуючого патерна .

Основні вимоги: Подія - об'єкт, який зберігає інформацію про подію. Підсистема збереження - сукупність об'єктів які приймають подію як аргумент і містять метод для її повернення в заданому форматі. Основна система містить список подій (можна задати літералом), для кожної із яких необхідно застосувати перетворення у відповідний формат даних та вивести в консоль (або зберегти до файлу).

Варіант 3. Розробити підсистему систему аналітики відвідування сайту. Система повинна реалізовувати методи:

1) Фіксування переходу в історії переходів.

2) Очищення статистики.

3) Вивід статистики про кількість переходів із кожного URL в консоль.

Система аналітики повинна існувати в єдиному екземплярі.

Обґрунтувати вибір породжуючого патерна .

Основні вимоги: Система містить список записів про переріхд - кожен із яких є об'єктом, який зберігає інформацію про адресу з якої здійснено перехід, кількість переходів та дату останнього переходу.

Варіант 4. Розробити підсистему для генерування HTML сторінки присвяченої подіям (Дата події, опис, ulr зображення). Сторінка може містити гедер, в якому вказати поточний рік, основну частину зі списком подій, блок із анонсом найближчої події та футер, де розміщено інформацію про авторів що виконали лабораторну роботу. Передбачити можливість генерування повної версії сторінки, сторінки без блоку анонсу, та сторінки без гедера та футера (тільки анонс та список подій). Обґрунтувати вибір породжуючого патерна.

Основні вимоги: Подія - об'єкт, який зберігає інформацію про подію. Основна система містить список подій (можна задати літералом), на основі цього списку згенерувати HTML сторінку присвячену подіям. Вибір одного із варіантів генерування здійснює користувач.

Варіант 5. Створити систему аналітики відвідування сайту. Система аналітики повинна існувати в єдиному екземплярі. Обґрунтувати вибір породжуючого патерна .

Система повинна реалізовувати методи:

1) Фіксування переходу із вказаного URL

2) Очищення статистики

3) Вивід статистики про кількість переходів із URL в консоль.

8. Умова творчого завдання.

Варіант 1. Запропонувати умову задачі проектування інформаційної для розв’язання якої доцільно використовувати шаблон Одинак.

Варіант 2. Запропонувати умову задачі проектування інформаційної для розв’язання якої доцільно використовувати шаблон Прототип.

Варіант 3. Запропонувати умову задачі проектування інформаційної для розв’язання якої доцільно використовувати шаблон Фабричний метод.

Варіант 4. Запропонувати умову задачі проектування інформаційної для розв’язання якої доцільно використовувати шаблон Будівельник.

Варіант 5. Запропонувати умову задачі проектування інформаційної для розв’язання якої доцільно використовувати шаблон Абстрактна фабрика.

Список рекомендованих джерел

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software.
2. Craig Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development
3. Robert Martin, Micah Martin. Agile Principles, Patterns, and Practices in C#.
4. Патерни проектування. URL: https://refactoring.guru/uk/design-patterns
5. Юр Т.В., Миронова Н.О.. Методичні вказівки до лабораторних робіт з дисципліни “Проектний практикум” для студентів напряму підготовки 6.050103 “Програмна інженерія” всіх форм навчання. Запоріжжя: ЗНТУ, 2013. 59 с.

Навчально-методичне видання

Андрашко Юрій Васильович

Методичні рекомендації до виконання лабораторних робіт із шаблонів проектування. Частина 1. (для студентів спеціальностей 124 «Системний аналіз» та 113 «Прикладна математика»)

В авторській редакції

Рекомендовано до друку Науково-методичною комісією факультету математики та цифрових технологій (протокол №5 від 3 березня 2021 року)