Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №6

Технології розроблення програмного забезпечення

«ШАБЛОНИ «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»»

Виконав:
студент групи ІА-24

Чайка А.П

Перевірив:

Мягкий М. Ю.

Київ 2024

Тема лабораторних робіт:

IRC client (singleton, builder, abstract factory, template method, composite, client-server)

Клієнт для IRC-чатів з можливістю вказівки порту і адреси з'єднання, підтримкабазових команд (підключення до чату, створення чату, установка імені,
реєстрація, допомога і т.д.), отримання метаданих про канал.

Завдання:

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціонала робочої програми у вигляді класів і їхвзаємодій для досягнення конкретних функціональних можливостей.
3. Застосування одного з даних шаблонів при реалізації програми.

Зміст

Хід роботи:

Крок 1. Теоретичні відомості
Ось оновлений текст:

1. Single Responsibility Principle (Принцип єдиної відповідальності)

Кожен клас повинен мати лише одну відповідальність (або причину для змін).

- Ідея: Клас або модуль повинен відповідати за одну чітко визначену задачу. Якщо клас виконує кілька завдань, це ускладнює його зміну, тестування та повторне використання.
- Приклад порушення: Клас UserManager обробляє авторизацію користувача та генерує PDF-звіти.
- Як виправити: Розділіть функціонал на окремі класи, наприклад, UserAuthentication для авторизації та PDFReportGenerator для створення звітів.

2. Open/Closed Principle (Принцип відкритості/закритості)

Класи повинні бути відкритими для розширення, але закритими для модифікації.

- Ідея: Ми повинні мати можливість додавати новий функціонал до класу, не змінюючи існуючий код.
- Приклад порушення: Метод calculateSalary(Employee employee) у класі PayrollService змінюється кожного разу, коли додається новий тип працівника.
- Як виправити: Використовуйте поліморфізм. Створіть інтерфейс SalaryCalculator і реалізуйте його для кожного типу працівника.

3. Liskov Substitution Principle (Принцип підстановки Барбари Лісков)

Об’єкти базового класу повинні мати можливість бути заміненими об’єктами підкласу без порушення логіки програми.

- Ідея: Клас-нащадок повинен успадковувати всі можливості базового класу та не порушувати його поведінки.
- Приклад порушення: У базовому класі Bird є метод fly(), але клас-нащадок Penguin викликає виключення, якщо цей метод використовується.
- Як виправити: Перемістіть метод fly() до інтерфейсу FlyingBird, а клас Penguin залиште як окремий без реалізації цього інтерфейсу.

4. Interface Segregation Principle (Принцип розділення інтерфейсів)

Краще мати багато вузькоспеціалізованих інтерфейсів, ніж один загальний, який змушує реалізовувати невикористовуваний функціонал.

- Ідея: Не змушуйте класи реалізовувати методи, які їм не потрібні.
- Приклад порушення: Інтерфейс Vehicle має методи drive(), sail(), fly(). Клас Car змушений реалізовувати методи sail() і fly(), навіть якщо вони йому не потрібні.
- Як виправити: Розділіть інтерфейси на LandVehicle, WaterVehicle, AirVehicle. Клас Car реалізуватиме лише LandVehicle.

5. Dependency Inversion Principle (Принцип інверсії залежностей)

Модулі верхнього рівня не повинні залежати від модулів нижнього рівня. Обидва типи модулів мають залежати від абстракцій.

- Ідея: Код не повинен бути жорстко пов’язаний із

конкретними реалізаціями. Замість цього слід використовувати абстракції.

- Приклад порушення: Клас OrderService напряму створює об’єкт MySQLDatabase, що ускладнює зміну бази даних.
- Як виправити: Використовуйте інтерфейс Database і залежність через ін’єкцію (Dependency Injection), щоб OrderService міг працювати з будь-якою реалізацією Database, наприклад, MySQLDatabase чи PostgreSQLDatabase.

Переваги використання SOLID:

1. Масштабованість: Код легше адаптувати до нових вимог.
2. Тестування: Простіше створювати модульні тести.
3. Підтримка: Код зрозумілий для інших розробників і легкий для підтримки.
4. Перевикористання: Частини коду можна використовувати в інших проєктах.

Дотримуючись цих принципів, ви отримуєте чистий, структурований та гнучкий код.

Ось опис шаблонів без прикладів коду:

1. Abstract Factory (Абстрактна фабрика)

Призначення:
Абстрактна фабрика використовується для створення сімейств взаємопов’язаних об’єктів без зазначення їхніх конкретних класів.

Основна ідея:
Цей шаблон дозволяє створювати групи об’єктів (наприклад, елементи інтерфейсу користувача для різних платформ) через єдиний інтерфейс. Це забезпечує незалежність коду від конкретних реалізацій об’єктів і полегшує зміну платформ або конфігурацій.

Застосування:

- У кросплатформних програмах.
- Коли необхідно створювати взаємопов’язані об’єкти певного “сімейства”.

2. Factory Method (Фабричний метод)

Призначення:
Цей шаблон дозволяє делегувати створення об’єктів підкласам, визначаючи загальний інтерфейс для їх створення.

Основна ідея:
Замість того, щоб безпосередньо створювати об’єкти в коді, ви викликаєте метод фабрики, який вирішує, який конкретно об’єкт створювати. Це підвищує гнучкість програми і полегшує додавання нових типів об’єктів.

Застосування:

- Коли потрібно дозволити підкласам вирішувати, які об’єкти створювати.
- Для створення об’єктів, залежних від контексту або логіки програми.

3. Memento (Сувенір/Знімок)

Призначення:
Шаблон використовується для збереження та відновлення стану об’єкта без порушення його інкапсуляції.

Основна ідея:
Об’єкт-знімок дозволяє зафіксувати стан іншого об’єкта, щоб пізніше відновити його до цього стану. Цей шаблон часто застосовують у системах із функціями “Скасувати” або “Історія змін”.

Застосування:

- У текстових редакторах для збереження та відновлення вмісту.
- У програмах, що працюють із транзакціями.

4. Observer (Спостерігач)

Призначення:
Даний шаблон створює залежність «один-до-багатьох», коли один

об’єкт (суб’єкт) повідомляє кілька залежних об’єктів (спостерігачів) про зміну свого стану.

Основна ідея:
Коли об’єкт змінює свій стан, всі “підписані” спостерігачі автоматично отримують повідомлення про це. Такий підхід часто використовується в системах сповіщення або потоків даних.

Застосування:

- У системах сповіщення (електронні листи, SMS).
- Для зв’язку між частинами програми, які мають бути синхронізованими.

5. Decorator (Декоратор)

Призначення:
Шаблон дозволяє динамічно додавати нові функціональні можливості об’єкту без зміни його коду.

Основна ідея:
Об’єкт обгортається іншими об’єктами (декораторами), які додають нову поведінку або функціонал. Це дозволяє створювати багато комбінацій функціональності з мінімальними змінами коду.

Застосування:

- У системах, де необхідно динамічно змінювати поведінку об’єкта.
- Для додавання додаткових функцій до об’єктів (наприклад, логування, кешування, валідація).

Ці шаблони є основою для створення гнучкого, розширюваного та зрозумілого коду, що легко адаптується до змін.

Крок 2. Реалізація шаблону проєктування для майбутньої системи

У програмі IRClient використано шаблон проектування Abstract Factory для організації створення різних типів команд, таких як JoinCommand та MetadataCommand. Це дозволяє централізовано управляти створенням об’єктів команд. Завдяки застосуванню абстрактної фабрики, процес створення команд передається окремій фабриці, що забезпечує гнучкість при додаванні нових типів команд без необхідності змінювати вже існуючий код.

Шаблон Abstract Factory дозволяє створювати сімейства взаємопов’язаних об’єктів, що належать до певного типу команд, зокрема об’єктів типу Command. Це дає змогу абстрагувати процес створення команд від конкретних реалізацій, що дає можливість змінювати типи команд (наприклад, додавати нові типи) без значних змін в інших частинах.

```
1   package org.example.AbstractFactory;
2
3   public interface Command {  7 usages  2 implementations
4       void comply();  2 usages  2 implementations
5   }
```

Рис. 1 – Код інтерфейсу Command

```
1   package org.example.AbstractFactory;
2
3   public interface CommandFactory {  3 usages  1 implementation
4       Command createCommand(String type, String... args);  2 usages  1 implementation
5   }
```

Рис. 2 – Код інтерфейсу CommandFactory

```
1   package org.example.AbstractFactory;
2
3   public class IRCCommandFactory implements CommandFactory {  2 usages
4       @Override  2 usages
5       public Command createCommand(String type, String... args) {
6           return switch (type) {
7               case "join" -> new JoinCommand(args[0], args[1]);
8               case "metadata" -> new MetadataCommand(args[0]);
9               default -> throw new IllegalArgumentException("Unknown command type");
10          };
11      }
12  }
```

Рис. 3 – Код класу IRCCommandFactory

```java
package org.example.AbstractFactory;

import ...

public class JoinCommand implements Command {  1 usage
    private String channel;  2 usages
    private String message;  2 usages

    public JoinCommand(String channel, String message) {  1 usage
        this.channel = channel;
        this.message = message;
    }

    @Override  2 usages
    public void comply() {
        IRCClient client = IRCClient.getSpecimen();
        IRCConnection connection = client.getConnection();
        connection.joinChannel(channel);
        connection.sendMessage(message);
    }
}
```

Рис. 4 – Код класу JoinCommand

```java
package org.example.AbstractFactory;

import ...

public class MetadataCommand implements Command {  1 usage
    private String metadata;  3 usages

    public MetadataCommand(String metadata) { this.metadata = metadata; }

    @Override  2 usages
    public void comply() {
        IRCClient client = IRCClient.getSpecimen();
        IRCConnection connection = client.getConnection();
        connection.sendMetadata(metadata);
        System.out.println("Sent metadata: " + metadata);
    }
}
```

Рис. 5 – Код класу MetadataCommand

Використання шаблону Abstract Factory в даному контексті має кілька важливих переваг:

1. Централізоване створення команд: Завдяки фабриці команд IRCCommandFactory, процес створення різних типів команд організовано в одному місці, що дозволяє зберігати порядок у коді та спрощує додавання нових команд у майбутньому.
2. Гнучкість при додаванні нових команд: Шаблон Abstract Factory забезпечує простоту додавання нових команд без необхідності змінювати основний код програми. Для цього достатньо створити нову реалізацію фабрики та додати обробку нової команди.
3. Відокремлення логіки створення об’єктів: Логіка створення об’єктів команд ізольована від інших частин програми, що робить код більш структурованим і легким для розширення.

Завдяки використанню Abstract Factory, ми отримуємо ефективний механізм для роботи з різними типами команд, що полегшує підтримку та масштабування коду. Цей шаблон дозволяє додавати нові команди без змін у основній логіці програми, забезпечуючи високу гнучкість та надійність.

Крок 3. Зображення структури шаблону

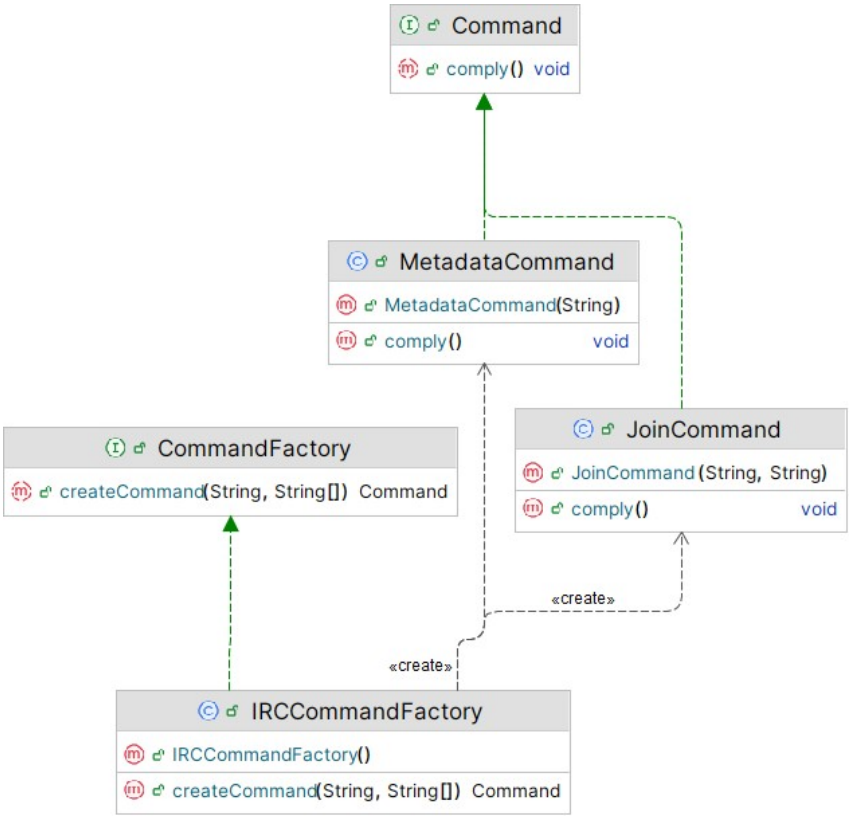

Рис. 6 – Структура шаблону Abstract Factory

Шаблон Builder, який використовується в класах IRCConnectionBuilder та IRCConnection, відокремлює процес створення об’єкта IRCConnection від його представлення. Це важливо для зручності налаштування та створення об’єкта з різними параметрами. Такий підхід дозволяє гнучко налаштовувати підключення до сервера, змінюючи лише окремі параметри без необхідності створювати складні конструкторами чи змінювати сам клас IRCConnection.

Проте, у деяких випадках використання шаблону Builder може бути надмірно складним, якщо об’єкти, які створюються, є дуже простими або мають лише кілька параметрів. У таких ситуаціях використання Builder може здатися зайвим, оскільки створення класу можна виконати без поетапного налаштування. Однак, основною перевагою цього шаблону є можливість створювати об’єкти з різними параметрами за допомогою одного коду, без складних конструкторів. Це також полегшує розширення класів і забезпечує чистоту та зрозумілість коду, що робить Builder чудовим вибором для складних об’єктів з багатьма параметрами.

Шаблон Abstract Factory в реалізації класів та інтерфейсів для IRC-клієнта допомагає централізувати створення різних типів команд, що спрощує розширення функціональності програми та підтримку коду. Він відокремлює логіку створення команд від решти програми, забезпечуючи чітку організацію коду та даючи можливість легко додавати нові типи команд без необхідності змінювати основну частину додатку.

Завдяки шаблону Abstract Factory, клас IRCCommandFactory може створювати різні конкретні команди (наприклад, JoinCommand, MetadataCommand) на основі переданого типу, що зберігає гнучкість і масштабованість програми. Це дозволяє додавати нові команди без зміни існуючого коду: для цього достатньо створити нову команду та оновити фабрику.

Однак, для простих або схожих команд, використання Abstract Factory може виглядати занадто складним. Якщо об’єкти команд мають лише кілька варіантів або не потребують великої гнучкості, шаблон може бути зайвим, адже створення простих об’єктів без фабрики також може бути ефективним і швидким.

Проте для великих і складних систем, де з’являються нові типи команд або змінюється бізнес-логіка, Abstract Factory значно спрощує процес підтримки та розширення програми, забезпечуючи чистоту та чіткість структури коду.

Код можна переглянути в даному **репозиторії**

Висновок:

Висновок

У рамках розробки IRC клієнта для чатів з підтримкою базових команд і можливістю вказівки порту та адреси з’єднання, основним шаблоном проектування, що був застосований, є Abstract Factory. Шаблон Abstract Factory дозволив централізувати створення різних типів команд IRC, що спростило додавання нових команд без необхідності змінювати основний код програми. Використання цього шаблону забезпечило гнучкість у додаванні нових типів команд, таких як JoinCommand чи MetadataCommand, при збереженні структури та організації коду.

Крім того, у лабораторній роботі було розглянуто й інші шаблони проектування, такі як Factory Method, Memento, Observer та Decorator. Кожен з цих шаблонів допоміг покращити гнучкість та масштабованість програми:

- Factory Method спростив створення конкретних об’єктів команд, делегуючи цей процес спеціальним фабрикам, що зберігає організованість коду.
- Memento дозволив зберігати та відновлювати стан об’єктів, що є важливим для відстеження змін і можливості повернення до попередніх значень.
- Observer забезпечив механізм спостереження за змінами в об’єктах, що корисно для реагування на події, наприклад, при зміні статусу підключення.
- Decorator надав можливість динамічно доповнювати функціональність об’єктів без зміни їх внутрішнього коду.