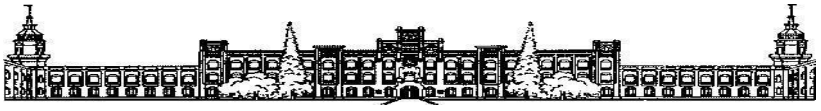


Міністерство освіти і науки України
Національний технічний
університет України
“Київський політехнічний інститут імені Ігоря
Сікорського” Факультет інформатики та обчислювальної
техніки Кафедра інформаційних систем та технологій

ЛАБОРАТОРНА РОБОТА №4

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Вступ до паттернів проектування»

Тема роботи: 3. Текстовий редактор

Виконав

студент групи ІА–33

Супик Андрій Олександрович

Київ 2025

Тема: Вступ до паттернів проектування

Мета: Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» та навчитися застосовувати їх в реалізації програмної системи.

Посилання на репозиторій: https://github.com/insxlll/trpzlab

Короткі теоретичні відомості

Патерн проектування — це типовий спосіб вирішення певної проблеми, що часто зустрічається при проектуванні архітектури програм.

На відміну від готових функцій чи бібліотек, патерн не можна просто взяти й скопіювати в програму. Патерн являє собою не якийсь конкретний код, а загальний принцип вирішення певної проблеми, який майже завжди треба підлаштовувати для потреб тієї чи іншої програми.

Патерни часто плутають з алгоритмами, адже обидва поняття описують типові рішення відомих проблем. Але якщо алгоритм — це чіткий набір дій, то патерн — це високорівневий опис рішення, реалізація якого може відрізнятися у двох різних програмах.

Якщо провести аналогії, то алгоритм — це кулінарний рецепт з чіткими кроками, а патерн — інженерне креслення, на якому намальовано рішення без конкретних кроків його отримання.

Крім цього, патерни відрізняються і за призначенням. Існують три основні групи патернів:

- Породжуючі патерни піклуються про гнучке створення об’єктів без внесення в програму зайвих залежностей.

- Структурні патерни показують різні способи побудови зв’язків між об’єктами

- Поведінкові патерни піклуються про ефективну комунікацію між об’єктами

Одинак (Singleton)

Шаблон проєктування "Одинак" гарантує, що клас матиме лише один екземпляр, і забезпечує глобальну точку доступу до цього екземпляра. Це корисно, коли потрібно контролювати доступ до деяких спільних ресурсів, наприклад, підключення до бази даних або конфігураційного файлу. Основна ідея полягає у тому, щоб закрити доступ до конструктора класу і створити статичний метод, що повертає єдиний екземпляр цього класу. У мовах, які підтримують багатопоточність, також важливо синхронізувати метод доступу, щоб уникнути створення кількох екземплярів в різних потоках. Один із способів реалізації одинак у Java – використання статичної ініціалізації, яка автоматично забезпечує безпечність у багатопоточному середовищі. Деякі розробники вважають одинак антипатерном, оскільки він створює глобальний стан програми, що ускладнює тестування та підтримку. Використання цього шаблону має бути обґрунтованим і обмеженим певними обставинами.

Ітератор (Iterator)

Шаблон "Ітератор" надає спосіб послідовного доступу до елементів колекції без розкриття її внутрішньої структури. Він особливо корисний для обходу різних типів колекцій, таких як списки, множини або деревовидні структури, незалежно від того, як вони реалізовані. Ітератор інкапсулює поточний стан перебору, тому може зберігати інформацію про те, який елемент є наступним. Цей шаблон дозволяє відокремити логіку роботи з колекцією від логіки обходу, що робить код більш гнучким і модульним. У Java цей шаблон реалізований у вигляді інтерфейсу Iterator, який надає методи hasNext() та next() для послідовного перебору елементів. Ітератор також можна використовувати для видалення елементів під час обходу колекції. Завдяки цьому шаблону можна використовувати поліморфізм для однакового доступу до елементів різних колекцій.

Проксі (Proxy)

Шаблон "Проксі" створює замісник або посередника для іншого об’єкта, що контролює доступ до цього об’єкта. Проксі може виконувати додаткову роботу перед передачею викликів реальному об'єкту, як-от перевірку прав доступу або відкладену ініціалізацію. Існує декілька типів проксі, серед яких захисний проксі, який контролює доступ, і віртуальний проксі, який затримує створення об'єкта, поки він не буде потрібен. У Java цей шаблон часто використовується для створення динамічних проксі за допомогою інтерфейсів, де проксі-клас реалізує той самий інтерфейс, що й реальний об’єкт. Проксі ефективний для оптимізації роботи з ресурсами або для контролю доступу до важких у створенні об'єктів. Це дозволяє зберігати оригінальний об'єкт захищеним і надає додатковий шар для маніпуляцій.

Стан (State)

Шаблон "Стан" дозволяє об'єкту змінювати свою поведінку при зміні внутрішнього стану, надаючи йому різні стани для різних контекстів. Він ефективно інкапсулює різні стани об'єкта як окремі класи і делегує дії поточному стану. Наприклад, кнопка може мати різні дії залежно від того, чи вона активована чи деактивована. Це дозволяє замість довгих умовних операторів використовувати поліморфізм, де кожен клас стану реалізує свою поведінку, визначену інтерфейсом стану. У Java цей шаблон може бути реалізований як клас з інтерфейсом для станів, де кожен стан є окремим підкласом, що відповідає за певну поведінку. Це полегшує масштабування та тестування, оскільки додавання нового стану не вимагає змін у вихідному коді

Стратегія (Strategy)

Шаблон "Стратегія" дозволяє вибирати алгоритм або поведінку під час виконання, забезпечуючи взаємозамінність різних алгоритмів для конкретного завдання. Він передбачає інкапсуляцію різних варіантів поведінки в окремих класах, які реалізують один інтерфейс, що спрощує заміну і додавання алгоритмів. Клас контексту отримує об'єкт стратегії і

викликає відповідні методи, не знаючи деталей реалізації конкретної стратегії. Наприклад, клас сортування може мати кілька стратегій: швидке сортування, сортування вставкою чи сортування вибором, і залежно від контексту обирається відповідний метод. У Java шаблон реалізується через інтерфейс стратегії, який мають різні класи конкретних стратегій

Хід роботи

У цій роботі для реалізації функціоналу обробки тексту було обрано патерн проєктування "Стратегія" (Strategy). Цей патерн був обраний як архітектурне рішення для вирішення фундаментальної проблеми: необхідно надати клієнтському коду (компоненту редактора) можливість динамічно змінювати алгоритм обробки тексту, не змінюючи сам клієнтський код.

У процесі роботи над проєктом стало очевидно, що "обробка тексту" — це не єдина дія, а ціле сімейство взаємозамінних алгоритмів. Наприклад, логіка перетворення тексту у верхній регістр, нижній регістр або очищення поля є різними, але, по суті, виконують один тип завдання — "трансформація тексту".

Без патерну "Стратегія" реалізація такого функціоналу призвела б до написання великої кількості умовних операторів (if/else або switch) безпосередньо у методі-обробнику подій (наприклад, ActionListener кнопки "Застосувати"). Це робить код жорстким, складним для тестування та неможливим для розширення без модифікації існуючого коду.

Патерн "Стратегія" вирішує цю проблему елегантно. Він дозволяє інкапсулювати кожен конкретний алгоритм в окремий клас, який реалізує спільний інтерфейс. Клієнт (у даному випадку, JStrategyTextPane) делегує виконання завдання об'єкту-стратегії, не вникаючи в деталі його реалізації.

Застосування цього патерну природним чином привело нас до визначення наступного сімейства алгоритмів: ClearText, UpperCaseText, LowerCaseText та

SentenceCaseText.

Реалізація класів

Реалізація шаблону “Strategy” вимагає створення окремих класів для реалізації кожного конкретного алгоритму.

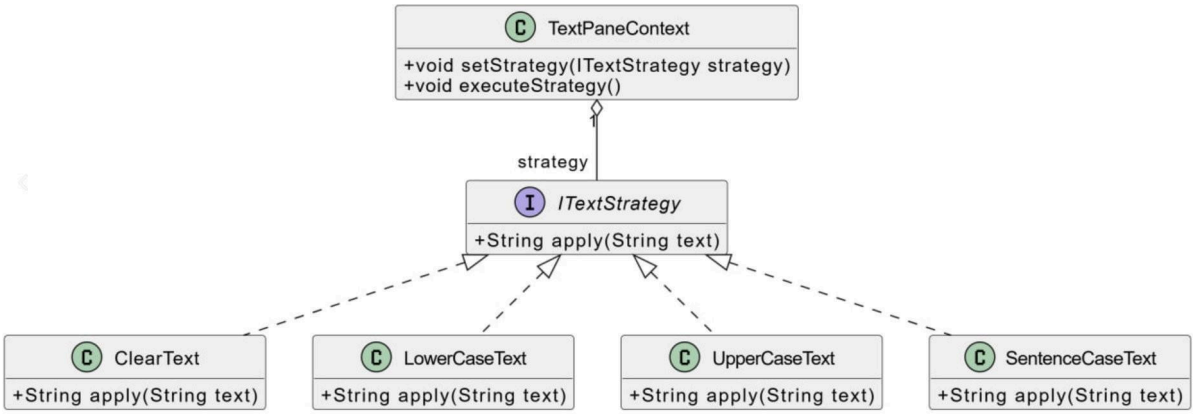


Рисунок 1. - Діаграма класів, яка представляє використання шаблону в реалізації системи

Опис створених об’єктів:

ITextStrategy (інтерфейс). ITextStrategy є абстрактною основою патерну. Він визначає загальний інтерфейс (контракт) для всіх конкретних стратегій обробки тексту. Інтерфейс декларує єдиний метод void processText(JTextPane textPane), який має бути реалізований усіма конкретними стратегіями. Він приймає JTextPane як аргумент, оскільки алгоритмам потрібен прямий доступ до компонента для отримання та встановлення тексту.

```
  Файл: ITextStrategy.java (Интерфейс Стратегии)
package com.example.client;
import javax.swing.JTextPane;
public interface ITextStrategy {
    void processText(JTextPane textPane);
}
```

ClearText (конкретна стратегія). Цей клас реалізує ITextStrategy і представляє алгоритм "Очищення текстового поля". Його реалізація методу processText є найпростішою: він викликає textPane.setText(""), повністю ігноруючи поточний вміст поля.

```
 Файл: ClearText.java (Конкретная Стратегия)
package com.example.client;
import javax.swing.JTextPane;
public class ClearText implements ITextStrategy {
    @Override
    public void processText(JTextPane textPane) {
        textPane.setText("");
    }
}
```

LowerCaseText (конкретна стратегія). Аналогічно до UpperCaseText, цей клас реалізує ITextStrategy для операції "Перетворення у нижній регістр". Його реалізація методу processText використовує метод .toLowerCase() для обробки тексту.

```
 Файл: LowerCaseText.java (Конкретная Стратегия)
package com.example.client;
import javax.swing.JTextPane;
public class LowerCaseText implements ITextStrategy {
    @Override
    public void processText(JTextPane textPane) {
        String text = textPane.getText();
        textPane.setText(text.toLowerCase());
    }
}
```

UpperCaseText (конкретна стратегія). Цей клас реалізує ITextStrategy і відповідає за алгоритм "Перетворення у верхній регістр". Його ключова відповідальність — отримати поточний текст з textPane (через textPane.getText()), виконати рядкову операцію (.toUpperCase()) і встановити оброблений результат назад у компонент (textPane.setText(...)).

```
 Файл: UpperCaseText.java (Конкретная Стратегия)
package com.example.client;
import javax.swing.JTextPane; public class UpperCaseText implements
ITextStrategy {
    @Override
    public void processText(JTextPane textPane) {
        String text = textPane.getText();
        textPane.setText(text.toUpperCase());
    }
```

SentenceCaseText (конкретна стратегія). Цей клас реалізує ITextStrategy і представляє найбільш складний алгоритм: "Перетворення у регістр речення" (перша літера речення — велика, решта — малі). Він так само отримує, обробляє та встановлює текст через переданий об'єкт textPane.

```
 Файл: SentenceCaseText.java (Конкретная Стратегия)
package com.example.client;
```

```
import javax.swing.JTextPane; public class SentenceCaseText implements
ITextStrategy {
    @Override
    public void processText(JTextPane textPane) {
        String text = textPane.getText().toLowerCase();
        StringBuilder result = new StringBuilder();
        boolean capitalizeNext = true;
        for (char c : text.toCharArray()) {
            if (capitalizeNext && Character.isLetter(c)) {
                result.append(Character.toUpperCase(c));
                capitalizeNext = false;
            } else {
                result.append(c);
            }
            if (c == '.' || c == '!' || c == '?' || c == '\n') {
                capitalizeNext = true;
            }
        }
        text = result.toString();
        textPane.setText(text);
    }
}
```

JStrategyTextPane (Клас "Контекст"). Цей клас виступає "Контекстом" для патерну. Він є нащадком класу JTextPane, що дозволяє йому безпосередньо бути компонентом UI. Клас містить приватне поле ITextStrategy strategy для зберігання посилання на поточний обраний алгоритм. Він надає публічні методи setStrategy(ITextStrategy strategy) (для зміни стратегії ззовні, наприклад, при виборі з меню) та executeStrategy(). При виклику executeStrategy(), "Контекст" делегує виконання, викликаючи strategy.processText(this).

```
  Файл: JStrategyTextPane.java (Контекст)
package com.example.client;
import javax.swing.JTextPane;
public class JStrategyTextPane extends JTextPane {
    private ITextStrategy textStrategy;
    void setTextStrategy(ITextStrategy textStrategy) {
        this.textStrategy = textStrategy;
    }
    void executeTextStrategy() {
        if (textStrategy != null) {
            textStrategy.processText(this);
        }
    }
}
```

Висновок: Виконуючи цю лабораторну роботу, я ознайомився з такими патернами, як Singleton, Ітератор, Проксі, Стан та Стратегія. Особливу увагу я приділив реалізації шаблону State, детально описавши його основну логіку та функціональність у контексті мого проєкту. Завдяки цьому досвіду я краще зрозумів, як ці патерни можуть підвищити модульність, зручність використання та читабельність коду, а також як вони сприяють ефективному управлінню об'єктами в програмуванні.

Контрольні запитання

1. Що таке шаблон проєктування?

Це формалізований, перевірений часом опис вдалого рішення типової проблеми, що часто зустрічається при проєктуванні програмних систем

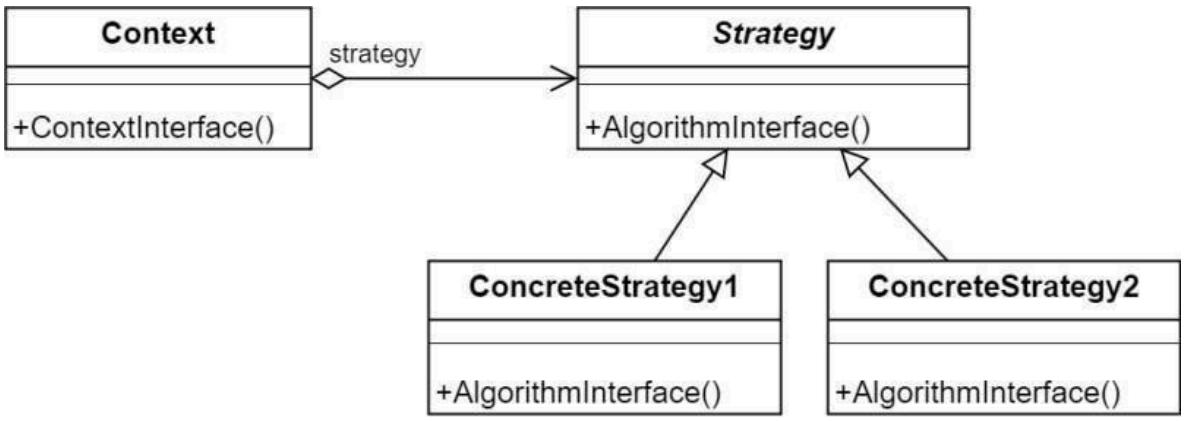
2. Навіщо використовувати шаблони проєктування?

Вони надають спільний словник для розробників, підвищують стійкість системи до змін, спрощують інтеграцію та дозволяють повторно використовувати перевірені архітектурні рішення.

3. Яке призначення шаблону «Стратегія»?

Дозволяє визначати сімейство алгоритмів, інкапсулювати кожен з них і робити їх взаємозамінними. Це дає змогу змінювати алгоритм незалежно від клієнтського коду, що його використовує

4. Нарисуйте структуру шаблону «Стратегія».



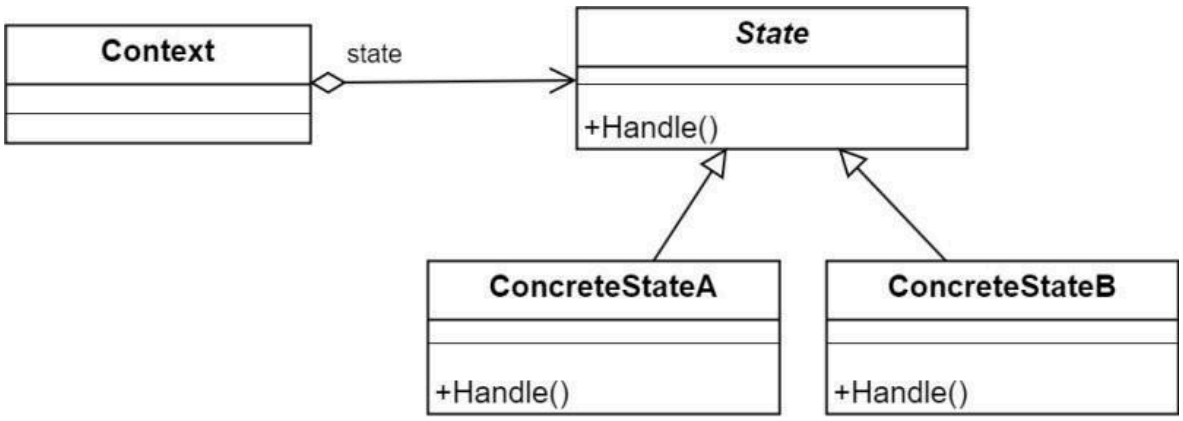
5. Які класи входять в шаблон «Стратегія», та яка між ними взаємодія?

Strategy (Інтерфейс): Оголошує загальний інтерфейс для всіх алгоритмів. ConcreteStrategy (Класи): Реалізують конкретні алгоритми. Context (Клас): Містить посилання на об'єкт-стратегію і взаємодіє з ним через загальний інтерфейс Strategy

6. Яке призначення шаблону «Стан»?

Дозволяє об'єкту змінювати свою поведінку при зміні його внутрішнього стану. Зовні це виглядає так, ніби об'єкт змінив свій клас.

7. Нарисуйте структуру шаблону «Стан».



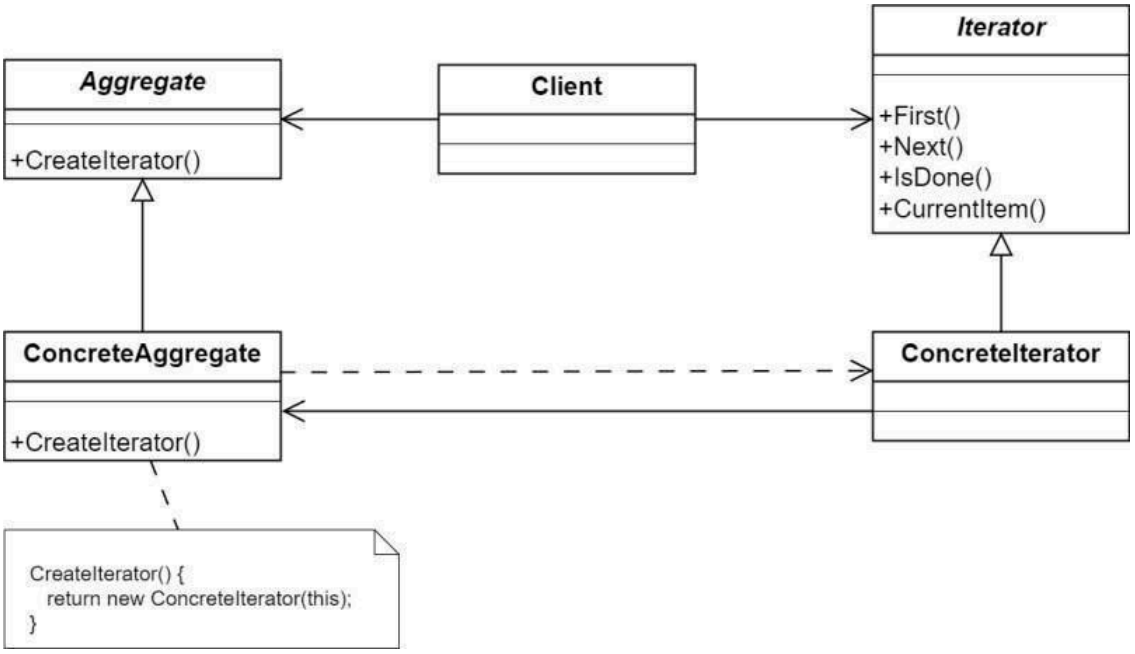
8. Які класи входять в шаблон «Стан», та яка між ними взаємодія?

State (Інтерфейс): Оголошує інтерфейс для поведінки, пов'язаної зі станом. ConcreteState (Класи): Реалізують поведінку для конкретних станів. Context (Клас): Зберігає посилання на поточний стан і делегує йому виконання роботи. Може змінювати свій стан.

9. Яке призначення шаблону «Ітератор»?

Надає уніфікований спосіб послідовного доступу до елементів колекції (агрегату), не розкриваючи її внутрішньої структури.

10. Нарисуйте структуру шаблону «Ітератор».



11. Які класи входять в шаблон «Ітератор», та яка між ними взаємодія? Iterator (Інтерфейс): Визначає методи для обходу (hasNext, next).

ConcreteIterator (Клас): Реалізує алгоритм обходу і відстежує поточну позицію. Aggregate (Інтерфейс): Визначає метод для створення ітератора. ConcreteAggregate (Клас): Реалізує метод створення ітератора, повертаючи екземпляр ConcreteIterator

12. В чому полягає ідея шаблона «Одинак»?

Гарантувати, що клас матиме лише один екземпляр (об'єкт), і надати глобальну точку доступу до цього екземпляра. Клас сам контролює створення екземпляра: приватний конструктор забороняє створення нових об'єктів ззовні, а статичний метод getInstance() повертає той самий єдиний екземпляр при кожному виклику.

13. Чому шаблон «Одинак» вважають «анти-шаблоном»?

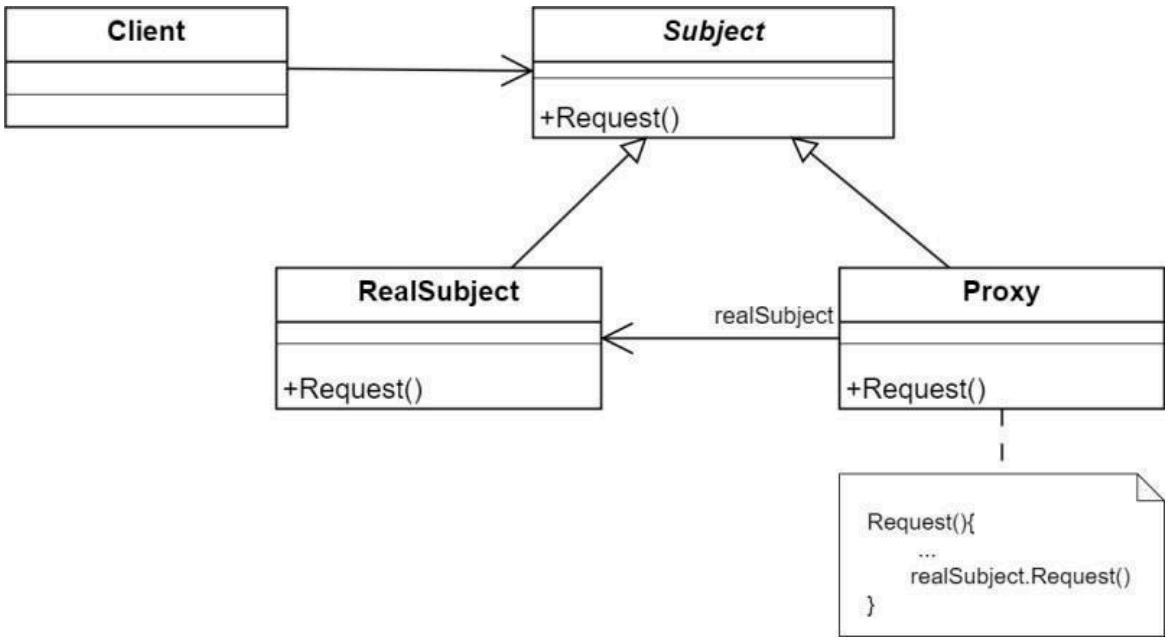
Singleton вважають анти-шаблоном через наступні проблеми: 1. Порушує Single Responsibility Principle – клас відповідає і за свою логіку, і за контроль кількості екземплярів. 2. Ускладнює тестування – складно

підмінити Singleton mock-об'єктом для unit-тестів, неможливо створити окремі екземпляри для паралельних тестів. 3. Прихований глобальний стан – створює неявні залежності між класами, що знижує модульність коду. 4. Проблеми з багатопоточністю – потребує додаткової синхронізації для коректної роботи в багатопоточному середовищі. 5. Порушує Dependency Injection – класи безпосередньо звертаються до Singleton, замість отримувати залежності ззовні.

14. Яке призначення шаблону «Проксі»?

Надає об'єкт-замінник (сурогат), який контролює доступ до іншого об'єкта. Проксі має той самий інтерфейс, що й реальний об'єкт, тому клієнт може працювати з ним прозорим чином. Проксі може виконувати додаткову логіку до або після делегування виклику реальному об'єкту.

15. Нарисуйте структуру шаблону «Проксі».



16. Які класи входять в шаблон «Проксі», та яка між ними взаємодія?

Subject (Інтерфейс) – визначає загальний інтерфейс для RealSubject та Proxy. Завдяки йому Proxy може бути використаний замість RealSubject. RealSubject (Клас) – реальний об'єкт, який виконує основну бізнес-логіку. Містить ресурсомісткі або захищені операції.
Proxy (Клас) – містить посилання на об'єкт RealSubject і реалізує той самий інтерфейс Subject. Перехоплює виклики клієнта, може виконувати додаткову логіку (перевірка прав, логування, кешування), а потім делегує виклик реальному об'єкту. Взаємодія: 1. Клієнт звертається до Proxy через інтерфейс Subject. 2. Proxy виконує додаткову логіку (наприклад, перевірка прав доступу). 3. Proxy делегує виклик об'єкту RealSubject. 4. RealSubject виконує реальну роботу і повертає результат. 5. Proxy може обробити результат (кешувати, логувати) і повернути його клієнту.
Клієнт не знає, працює він з Proxy чи з RealSubject – обидва мають однаковий інтерфейс