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 «Теорія розробки програмного забезпечення»

Тема: «Вступ до паттернів проектування»

Виконав:
Студент групи ІА-34
Цап Андрій

Перевірив:
Мягкий Михайло Юрійович

Київ – 2025

Зміст

Вступ

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

У сучасному програмуванні розробка складних систем вимагає чіткої архітектури, зрозумілої структури та можливості легкого розширення функціоналу. Для цього використовуються шаблони проєктування — перевірені практикою рішення типових задач проєктування програмного забезпечення. У даній роботі розглядаються шаблони **Singleton**, **Iterator**, **Proxy**, **State** та **Strategy**.

У рамках лабораторної роботи було проаналізовано функціональні частини створеного програмного забезпечення для управління проєктами та вибрано найбільш доцільний шаблон для впровадлення.

Теоретичні відомості

Будь-який патерн проєктування, використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проєктування, вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях.

Відповідне використання патернів проєктування дає розробнику ряд незаперечних переваг. Наведемо деякі з них. Модель системи, побудована в межах патернів проєктування, фактично є структурованим виокремленням тих елементів і зв’язків, які значимі при вирішенні поставленого завдання. Крім цього, модель, побудована з використанням патернів проєктування, більш проста і наочна у вивченні, ніж стандартна модель. Проте, не дивлячись на простоту і наочність, вона дозволяє глибоко і всебічно опрацювати архітектуру розроблюваної системи з використанням спеціальної мови.

Шаблон «Singleton» - призначення патерну: «Singleton» (Одинак) являє собою клас в термінах ООП, який може мати не більше одного об'єкта (звідси і назва «одинак»). Насправді, кількість об'єктів можна задати (тобто не можна створити більш n об'єктів даного класу). Даний об'єкт найчастіше зберігається як статичне поле в самому класі.

Переваги та недоліки: Однак слід зазначити, що в даний час патерн «Одинак» багато хто вважає т.зв. «анти-шаблоном», тобто поганою практикою проєктування. Це пов'язано з тим, що «одинаки» представляють собою глобальні дані (як глобальна змінна), що мають стан. Стан глобальних об'єктів важко відслідковувати і підтримувати коректно; також глобальні об'єкти важко тестуються і вносять складність в

програмний код (у всіх ділянках коду виклик в одне єдине місце з «одинаком»; при зміні підходу доведеться змінювати масу коду).

Шаблон «Iterator» - призначення: «Iterator» (Ітератор) являє собою шаблон реалізації об'єкта доступу до набору (колекції, агрегату) елементів без розкриття внутрішніх механізмів реалізації. Ітератор виносить функціональність перебору колекції елементів з самої колекції, таким чином досягається розподіл обов'язків: колекція відповідає за зберігання даних, ітератор – за прохід по колекції.

Переваги та недоліки: цей шаблон дозволяє уніфікувати операції проходження по наборам об'єктів для всіх наборів. Тобто, незалежно від реалізації (масив, зв'язаний список, незв'язаний список, дерево та ін.), кожен з наборів може використовувати будь-який з реалізованих ітераторів.

Шаблон «Proxy» - призначення: «Proxy» (Проксі) – об'єкти є об'єктами-заглушками або двійниками/замінниками для об'єктів конкретного типу. Зазвичай, проксі об'єкти вносять додатковий функціонал або спрощують взаємодію з реальними об'єктами.

Переваги та недоліки:

+ Легкість впровадження проміжного рівня без переробки клієнтського коду.

+ Додаткові можливості по керуванню життєвим циклом об’єкту.

- Існує ризик падіння швидкості роботи через впровадження додаткових операцій.

- Існує ризик неадекватної заміни відповіді клієнтському коду.

Шаблон «State» - призначення: Шаблон «State» (Стан) дозволяє змінювати логіку роботи об'єктів у випадку зміни їх внутрішнього стану [6]. Наприклад, відсоток нарахованих на картковий рахунок грошей залежить від стану картки: Visa Electron, Classic, Platinum і т.д. Або обсяг послуг, які надані хостинг компанією, змінюється в залежності від обраного тарифного плану (стану членства – бронзовий, срібний або золотий клієнт).

Переваги та недоліки:

+ Код специфічний для окремого стану реалізується в класі стану.

+ Класи та об’єкти станів можна використовувати з різними контекстами, за рахунок чого збільшується гнучкість системи.

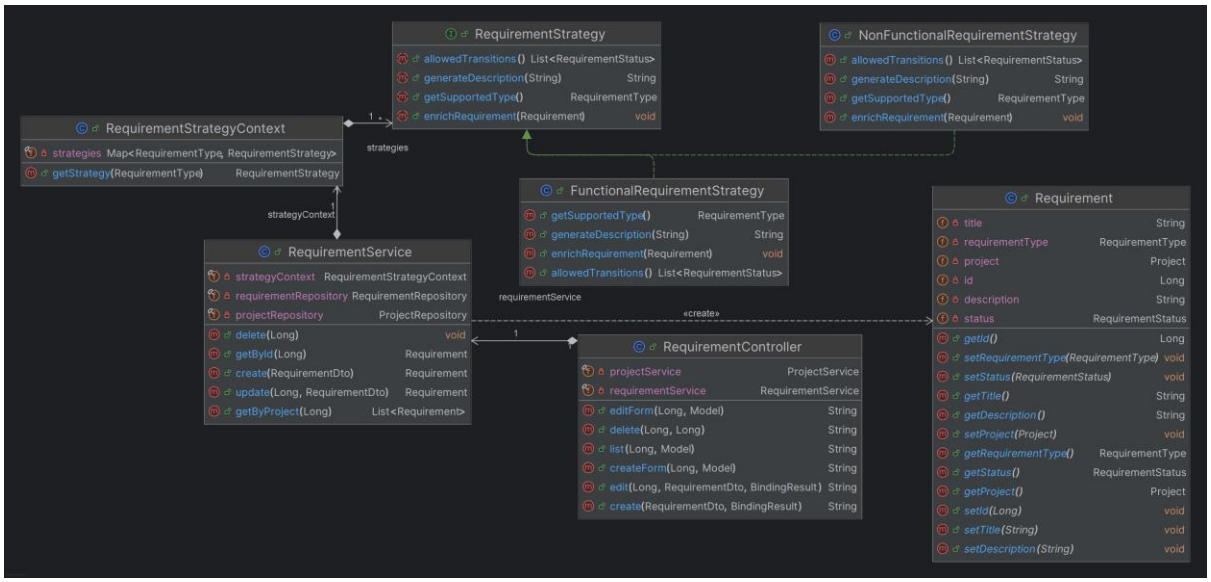
+ Код контексту простіше читати, тому що вся залежна від станів логіка винесена в інші класи.

+ Відносно легко додавати нові стани, головне правильно змінити переходи між станами.

- Клас контекст стає складніше через ускладнений механізм переключення станів.

Хід роботи

Діаграма класів для реалізації шаблону Strategy:



1. Requirement

Requirement — це доменна модель, що представляє окрему вимогу в проєкті. Саме цей клас є об’єктом, над яким працюють стратегія та сервіс.

2. RequirementStrategy

RequirementStrategy — інтерфейс патерна Strategy, який визначає поведінку для різних типів вимог.

Містить універсальні методи:

getSupportedType() — який тип вимоги підтримується

generateDescription() — як формується опис

allowedTransitions() — які статуси доступні

enrichRequirement() — як модифікується вимога при створенні

3. FunctionalRequirementStrategy / NonFunctionalRequirementStrategy

Це конкретні реалізації стратегії.

FunctionalRequirementStrategy:

- додає префікс FUNC-
- створює структурований опис поведінки
- визначає workflow для робочих вимог

NonFunctionalRequirementStrategy:

- додає префікс NFR-
- генерує блок критеріїв (продуктивність, безпека, якість)
- має свій набір дозволених статусів

4. RequirementStrategyContext

RequirementStrategyContext — механізм вибору стратегії.
Він:

- автоматично збирає всі стратегії через Spring
- зберігає їх у Map<RequirementType, RequirementStrategy>
- повертає відповідну стратегію за типом вимоги

5. RequirementService

RequirementService — бізнес-логіка роботи з вимогами.

Сервіс не залежить від конкретних стратегій, що робить систему розширюваною.

6. RequirementController

RequirementController — контролер MVC, який: приймає дані з UI, передає їх у сервісний шар, запускає процес створення/редагування вимог

Переваги використання

Гнучке розширення функціональності без зміни існуючого коду

У модулі роботи з вимогами різні типи (функціональні та нефункціональні) мають різну логіку опрацювання. Завдяки шаблону Strategy нову поведінку можна додати шляхом створення нового класу-стратегії, не змінюючи вже написаного коду.

Відсутність умовної логіки у сервісах

Без використання стратегії логіка RequirementService швидко перетворилася б на набір умов: if/else та switch.
У майбутньому це ускладнювало б підтримку проєкту. Strategy дозволяє винести всю мінливу поведінку в окремі класи, зберігаючи RequirementService компактним, чистим і відповідальним лише за загальні операції.

Чистіша архітектура та чітке розділення відповідальностей

Strategy дозволяє зберегти архітектуру модульною:

- контролер відповідає за маршрутизацію,
- сервіс — за бізнес-логіку створення та збереження,
- стратегія — за специфічну поведінку конкретного типу вимог.

Недоліки використання

Збільшення кількості класів у проєкті

Для реалізації шаблону потрібно створити:

- інтерфейс стратегії,
- декілька реалізацій,
- фабрику для вибору правильної стратегії.

Ускладнення початкового розуміння архітектури

Новачку або перевіряючому складніше швидко зрозуміти всі переходи

Вихідні коди реалізації шаблону Strategy

RequirementStrategy.java

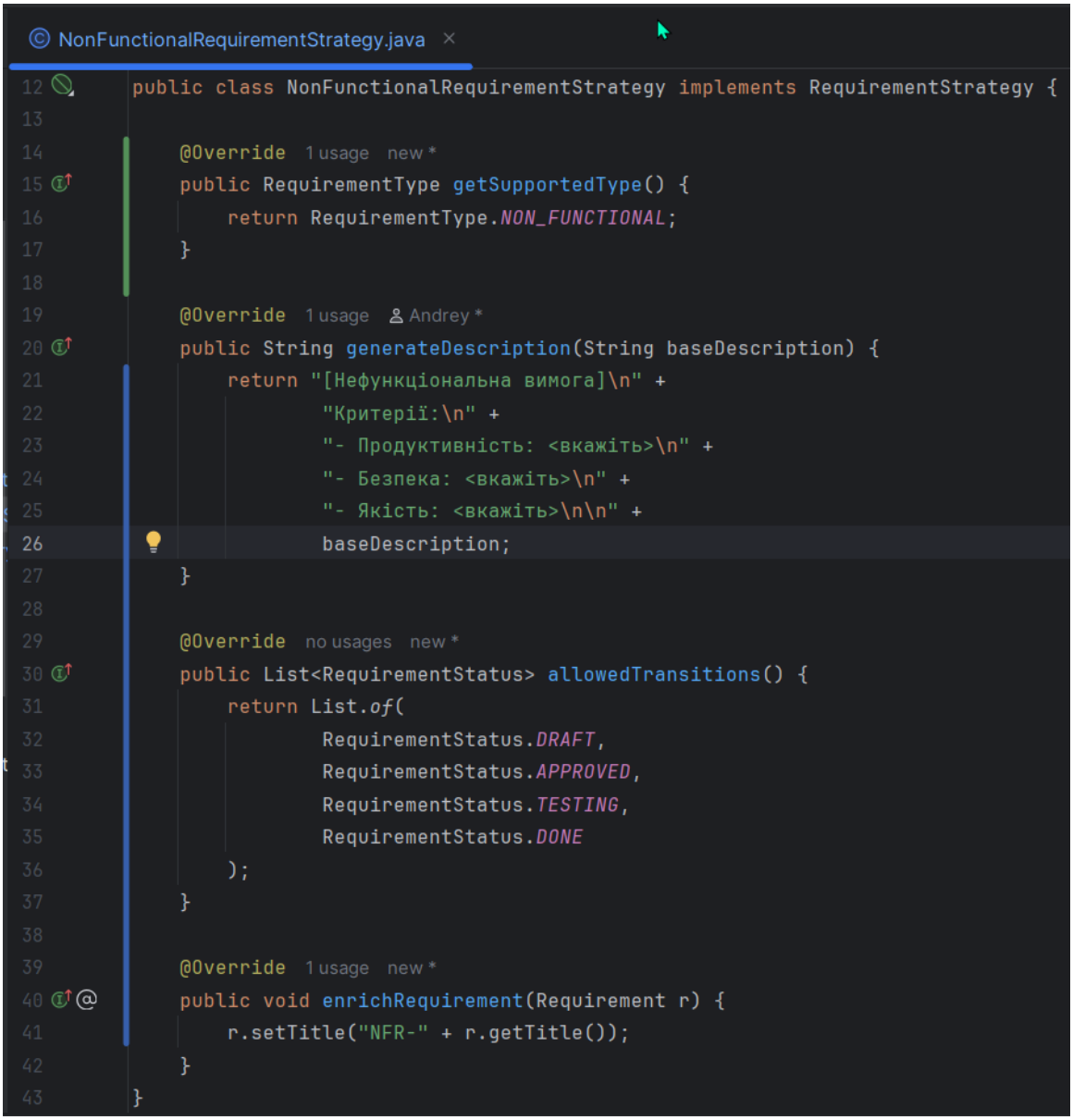
```java
package org.example.projectmanagementsoftware.pattern.strategy.intefraces;

import org.example.projectmanagementsoftware.domain.Requirement;
import org.example.projectmanagementsoftware.domain.enums.RequirementStatus;
import org.example.projectmanagementsoftware.domain.enums.RequirementType;

import java.util.List;

public interface RequirementStrategy {  9 usages  2 implementations  Andrey *

    RequirementType getSupportedType();  1 usage  2 implementations  new *

    String generateDescription(String baseDescription);  1 usage  2 implementations  Andr

    List<RequirementStatus> allowedTransitions();  no usages  2 implementations  new *

    void enrichRequirement(Requirement requirement);  1 usage  2 implementations  new *
}
```

Рис 1 - Код RequirementStrategy

RequirementStrategyContext.java

```java
package org.example.projectmanagementsoftware.pattern.strategy.implementations;

import org.example.projectmanagementsoftware.domain.enums.RequirementType;
import org.example.projectmanagementsoftware.pattern.strategy.intefraces.RequirementStrategy;
import org.springframework.stereotype.Component;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Component  2 usages  new *
public class RequirementStrategyContext {

    private final Map<RequirementType, RequirementStrategy> strategies = new HashMap<>();  2 u

    public RequirementStrategyContext(List<RequirementStrategy> strategyList) {  new *
        for (RequirementStrategy s : strategyList) {
            strategies.put(s.getSupportedType(), s);
        }
    }

    public RequirementStrategy getStrategy(RequirementType type) {  1 usage  new *
        return strategies.get(type);
    }
}
```

Рис 2 - Код RequirementStrategyContext



```java
public class NonFunctionalRequirementStrategy implements RequirementStrategy {

    @Override  1 usage  new *
    public RequirementType getSupportedType() {
        return RequirementType.NON_FUNCTIONAL;
    }

    @Override  1 usage  Andrey *
    public String generateDescription(String baseDescription) {
        return "[Нефункціональна вимога]\n" +
                "Критерії:\n" +
                "- Продуктивність: <вкажіть>\n" +
                "- Безпека: <вкажіть>\n" +
                "- Якість: <вкажіть>\n\n" +
                baseDescription;
    }

    @Override  no usages  new *
    public List<RequirementStatus> allowedTransitions() {
        return List.of(
                RequirementStatus.DRAFT,
                RequirementStatus.APPROVED,
                RequirementStatus.TESTING,
                RequirementStatus.DONE
        );
    }

    @Override  1 usage  new *
    public void enrichRequirement(Requirement r) {
        r.setTitle("NFR-" + r.getTitle());
    }
}
```

Рис 3 - Код NonFuctionalRequirementStrategy

FunctionalRequirementStrategy.java

```java
public class FunctionalRequirementStrategy implements RequirementStrategy {
    @Override  1 usage  new *
    public RequirementType getSupportedType() {
        return RequirementType.FUNCTIONAL;
    }

    @Override  1 usage  Andrey *
    public String generateDescription(String baseDescription) {
        return "[Функціональна вимога]\n" +
                "Опис: " + baseDescription + "\n" +
                "Вимога визначає конкретну поведінку системи.";
    }

    @Override  no usages  new *
    public List<RequirementStatus> allowedTransitions() {
        return List.of(
                RequirementStatus.DRAFT,
                RequirementStatus.APPROVED,
                RequirementStatus.IN_PROGRESS,
                RequirementStatus.DONE
        );
    }

    @Override  1 usage  new *
    public void enrichRequirement(Requirement r) {
        r.setTitle("FUNC-" + r.getTitle());
    }
}
```

Рис 4 - Код FuctionalRequirementStrategy

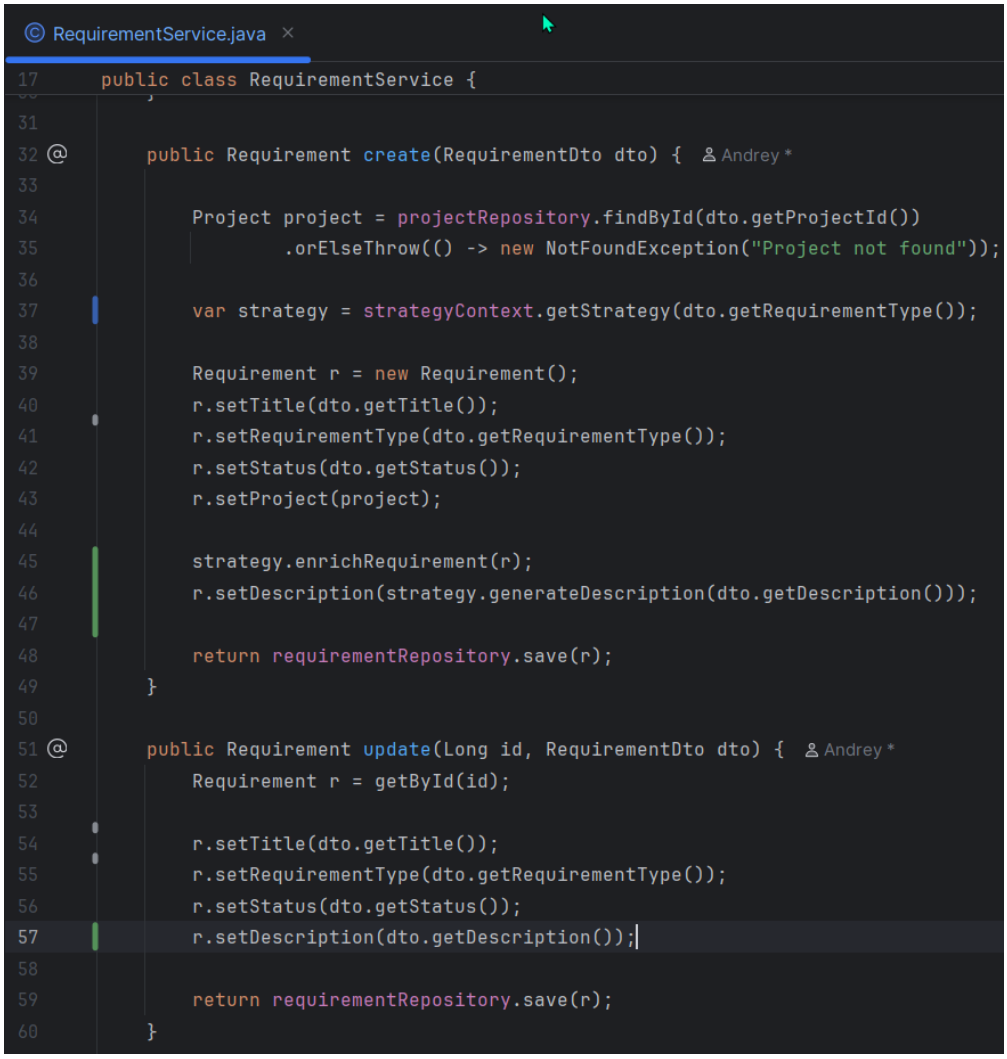


Рис 5 - Змінені методи з класу RequirementService

Висновки

Під час виконання лабораторної роботи були розглянуті основні структурні шаблони проєктування — Singleton, Iterator, Proxy, State та Strategy. Було визначено їх призначення, особливості використання, переваги та недоліки.

Після аналізу архітектури розроблюваної системи було встановлено, що найбільш доцільним шаблоном для інтеграції є Strategy, оскільки він дозволяє гнучко змінювати поведінку об’єктів без модифікації їхнього коду. Впровадження Strategy у модуль управління вимогами дало можливість реалізувати різні способи сортування даних (за назвою, типом, пріоритетом). Це покращило масштабованість системи, зробило код чистішим та підтримуваним.

Контрольні запитання

1. Що таке шаблон проєктування?

Формалізоване рішення типової задачі проєктування, яке показує взаємодію класів і об’єктів та містить рекомендації щодо застосування.

2. Навіщо використовувати шаблони проєктування?

Для підвищення гнучкості, повторного використання коду, зрозумілості архітектури та спрощення підтримки системи.

3. Яке призначення шаблону «Стратегія»?

Дозволяє змінювати поведінку об’єкта, вибираючи один з алгоритмів (стратегій), що досягають однієї мети різними способами.

4. Нарисуйте структуру шаблону «Стратегія».

Контекст → Стратегія (інтерфейс) → Конкретні стратегії.

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

- Контекст – виконує дії через стратегію.
- Інтерфейс стратегії – визначає метод.
- Конкретні стратегії – реалізують алгоритм. Контекст делегує роботу конкретній стратегії.

6. Яке призначення шаблону «Стан»?

Дозволяє змінювати поведінку об’єкта залежно від його внутрішнього стану, ізолюючи логіку станів в окремі класи.

7. Нарисуйте структуру шаблону «Стан».

Контекст → Інтерфейс стану → Конкретні стани.

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

- Контекст – делегує виклики стану.
- Інтерфейс стану – визначає методи поведінки.
- Конкретні стани – реалізують конкретну поведінку.

9. Яке призначення шаблону «Ітератор»?

Дозволяє послідовно обходити елементи колекції без розкриття її внутрішньої структури, виносячи логіку обходу з колекції.

10. Нарисуйте структуру шаблону «Ітератор».

Колекція → Ітератор → Клієнт.
Ітератор керує станом обходу та надає доступ до елементів.

11. Які класи входять в шаблон «Ітератор», та яка між ними взаємодія?

- Колекція – зберігає дані.
- Ітератор – відстежує стан обходу і надає елементи.
- Клієнт – використовує ітератор для перебору без знання внутрішньої структури.

12. В чому полягає ідея шаблона «Одинак»?

Гарантує існування лише одного екземпляра класу та надає глобальну точку доступу до нього.

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

Тому що він створює глобальні об’єкти зі станом, що ускладнює тестування, підтримку та порушує принцип єдиної відповідальності класу.

14. Яке призначення шаблону «Проксі»?

Створює об’єкт-замінник для реального об’єкта, контролює доступ та додає додаткову логіку або оптимізацію, не змінюючи клієнтський код.

15. Нарисуйте структуру шаблону «Проксі».

Клієнт → Проксі → Реальний об’єкт.

Проксі реалізує той самий інтерфейс, що й реальний об’єкт, і делегує йому виклики.

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

- Клієнт – взаємодіє через інтерфейс.
- Інтерфейс – визначає методи.
- Реальний об’єкт – виконує основну роботу.
- Проксі – контролює доступ, кешує або об’єднує запити, делегуючи їх реальному об’єкту.