МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО»

ЛАБОРАТОРНА РОБОТА №4

ТЕМА: «Вступ до паттернів проектування»

Виконала:

Студентка групи ІА-34

Потейчук С.А.

Перевірив:

Мягкий М.Ю.

Київ 2025

Зміст

Теоретичні відомості

Патерн проєктування є формалізованим описом вдалого рішення типової задачі, який можна багаторазово застосовувати. Кожен патерн має загальновживану назву, що створює єдину мову для спілкування розробників. Для ефективного використання патернів необхідним етапом є адекватне моделювання предметної області, що дозволяє коректно формалізувати задачу та вибрати відповідні рішення.

Використання патернів надає розробнику низку переваг. Вони дозволяють створити структуровану модель системи, яка виокремлює найважливіші елементи та зв'язки. Така модель є більш простою та наочною для вивчення, але при цьому дозволяє глибоко опрацювати архітектуру. Крім того, патерни підвищують стійкість системи до змін вимог та спрощують її подальше доопрацювання та інтеграцію. Таким чином, патерни є перевіреними роками практики «ескізами» архітектурних рішень для типових ситуацій.

Шаблон «Singleton» (Одинак)

Призначенням патерну «Одинак» є гарантування існування лише одного екземпляра класу та надання глобальної точки доступу до нього. Він вирішує проблему, коли в системі не може бути більше одного фізичного об'єкта певного класу, наприклад, файлу налаштувань, або коли необхідно жорстко контролювати всі операції, що проходять через цей клас, як у випадку керування єдиним мережевим з'єднанням.

Рішення полягає у зберіганні єдиного екземпляра як статичного поля всередині самого класу та закритті його конструктора для блокування створення нових об'єктів. Однак слід зазначити, що зараз цей патерн часто розглядають як антипатерн через його недоліки. Головним чином, він порушує принцип єдиної відповідальності класу і, будучи глобальним станом, значно ускладнює тестування та підтримку коду.

Шаблон «Iterator» (Ітератор)

Патерн «Ітератор» призначений для послідовного доступу до елементів будь-якої колекції без необхідності знати її внутрішню структуру. Він вирішує проблему обходу складних колекцій, таких як дерева або графи, коли необхідно мати різні алгоритми перебору. Замість того, щоб додавати цю функціональність в сам клас колекції, що розмиває його основне призначення, логіка обходу виноситься в окремий об'єкт-ітератор.

Ітератор відповідає за відстеження стану обходу та поточної позиції в колекції. Типовий ітератор містить методи для переходу до першого елемента, переходу до наступного, перевірки завершення обходу та отримання поточного елемента. Головною перевагою цього підходу є можливість реалізації різних способів обходу однієї й тієї ж колекції без змін в її коді. Недоліком є те, що для простих колекцій, де достатньо звичайного циклу, використання ітератора може бути надмірним.

Шаблон «Proxy» (Заступник)

Патерн «Заступник» служить для створення спеціальних об'єктів-замінників, які виступають посередниками між клієнтським кодом і реальними об'єктами. Основна мета заступника – контролювати доступ до справжнього об'єкта, додаючи додатковий функціонал, такий як кешування, відкладна ініціалізація або логування, без зміни оригінального коду.

Типовою проблемою, яку вирішує цей патерн, є необхідність оптимізації взаємодії з ресурсом, що є витратним за часом або грошима. Для вирішення цієї проблеми створюється клас-заступник, який імплементує той самий інтерфейс, що і оригінальний об'єкт для роботи з сервісом. Однак, замість негайної відправки кожного запиту, заступник накопичує їх у внутрішній черзі і періодично відправляє всі разом. Аналогічно він може кешувати відповіді від сервісу, повертаючи їх клієнтам без зайвих звернень.

Таким чином, основний об'єкт, що безпосередньо працює з сервісом, використовується рідше, а клієнтський код навіть не помічає змін, оскільки продовжує працювати через той самий інтерфейс.

Головною перевагою патерну є легкість впровадження додаткової логіки без необхідності змін у клієнтському коді або в самому реальному об'єкті. До недоліків можна віднести деяке зниження швидкості роботи через додатковий рівень абстракції та потенційний ризик того, що заступник не зможе точно відтворити поведінку реального об'єкта в усіх ситуаціях.

Шаблон «State» (Стан)

Патерн «Стан» призначений для тих випадків, коли об'єкт повинен кардинально змінювати свою поведінку в залежності від свого внутрішнього стану. Замість того, щоб використовувати безліч умовних операторів, які перевіряють поточний стан, ця логіка інкапсулюється в окремих об'єктах-станах.

Яскравим прикладом проблеми, де застосовується цей патерн, є розробка модуля Listener для сервера. Цей модуль має поводити себе по-різному на різних етапах життєвого циклу: ігнорувати всі вхідні запити під час ініціалізації системи, нормально їх обробляти після повного запуску та відправляти лише відповіді, але не приймати нові запити, під час процедури завершення роботи.

Відповіддю на це виклик є визначення загального інтерфейсу для всіх станів, наприклад, IListenerState. Кожен конкретний стан реалізує цей інтерфейс у власному класі, такому як InitializingState, OpenState чи ClosingState. Основний клас Listener містить посилання на об'єкт поточного стану і всі вхідні виклики просто делегує цьому об'єктові. Коли виникає необхідність змінити поведінку, наприклад, після успішного запуску сервера, в полі стану встановлюється новий об'єкт – OpenState. Ця проста заміна призводить до миттєвої і повної зміни всієї поведінки Listener.

Такий підхід робить код значно чистішим і гнучкішим. Усі алгоритми, специфічні для певного стану, зосереджені в одному місці, що спрощує їх супровід і додавання нових станів у майбутньому. Основним недоліком є певне ускладнення структури програми через збільшення кількості класів і необхідність чітко керувати переходами між станами.

Шаблон «Strategy» (Стратегія)

Патерн «Стратегія» використовується для того, щоб визначити сімейство схожих алгоритмів, упакувати кожен з них в окремий клас і зробити їх взаємозамінними. Це дозволяє вибирати потрібний алгоритм прямо під час виконання програми, не змінюючи клієнтський код, який його використовує.

Сутність цього патерну полягає у винесенні алгоритмів, які часто змінюються або мають кілька варіантів реалізації, за межі основного класу контексту. Замість безлічі умовних операторів if або switch, які вибирають потрібну поведінку, контекст отримує посилання на об'єкт-стратегію. Коли потрібно виконати певну операцію, контекст просто делегує її виконання поточній стратегії. Для зміни алгоритму досить підставити в контекст інший об'єкт стратегії.

Головною перевагою патерну «Стратегія» є те, що він дозволяє легко додавати нові алгоритми без модифікації існуючого коду, дотримуючись принципу відкритості/закритості. Він також сприяє чистоті коду, оскільки ізолює деталі реалізації різних алгоритмів. Однак, його використання може бути надмірним, якщо алгоритмів небагато і вони дуже прості, оскільки це призводить до невиправданого ускладнення архітектури. Крім того, клієнтський код повинен розуміти відмінності між стратегіями, щоб вибрати найбільш підходящу.

Тема: Вступ до паттернів проектування.

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

Хід роботи

7. Редактор зображень (state)

Редактор зображень має такі функціональні можливості: відкриття/збереження зображень у найпопулярніших форматах, застосування ефектів, наприклад поворот, розтягування, стиснення, кадрування зображення, можливість створення колажів шляхом «нашарування» зображень.

Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

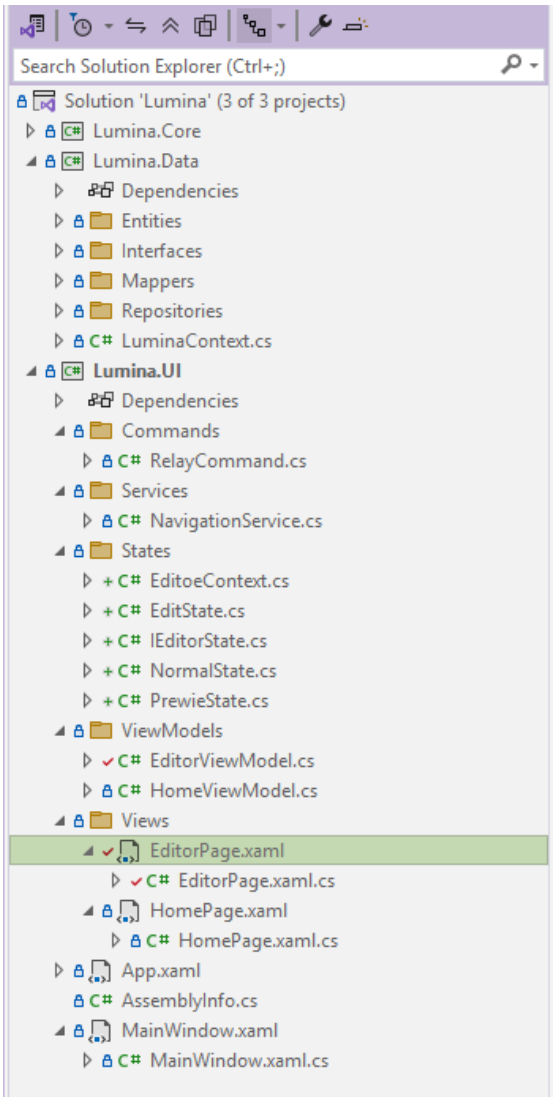

Рисунок 1 – Структура проєкту

Загальна структура проєкту:

- Lumina.Core – Бізнес-логіка, сервіси, моделі.
- Lumina.Data – Робота з базою даних, реалізації репозиторіїв, контекст EF Core, сутності та мапери.
- Lumina.UI – Графічний інтерфейс (WPF), користувацькі дії, візуальне відображення, керування станами.

Lumina.UI:

- App.xaml / App.xaml.cs – Точка входу. Ініціалізація головного вікна.
- MainWindow.xaml / MainWindow.xaml.cs – Головне вікно, у якому завантажуються сторінки (HomePage, EditorPage).
- HomePage.xaml / HomePage.xaml.cs – Головна сторінка з кнопками “Open Image”, “Create Collage”, “Go to Editor”, “Exit”.
- EditorPage.xaml / EditorPage.xaml.cs – Головна сторінка редактора: полотно, панель інструментів, ефекти, стани.
- Controls/ – Користувацькі елементи управління (наприклад, панель інструментів, область зображення).
- ViewModels/ – ViewModel-и для зв’язку UI та Core.
- State/ – Реалізація патерну State для поведінки редактора.

State – режими роботи редактора

Інтерфейс: IEditorState (визначає поведінку редактора).

States:

- NormalState – просто перегляд
- EditState – редагування, застосування ефектів
- PreviewState – перегляд результату

Контекст: EditorContext (зберігає поточний стан, викликає його логіку, використовується на EditorPage)

У проєкті патерн State (Стан) використовується для динамічного керування поведінкою редактора зображень залежно від поточного режиму роботи (Normal, Edit, Preview).

Патерн State дозволяє об’єкту змінювати свою поведінку при зміні внутрішнього стану, не використовуючи складні if або switch. Тобто об’єкт просто делегує дії об’єкту, що представляє поточний стан.

```
namespace Lumina.UI.States
{
    3 references
    public class EditorContext
    {
        private IEditorState _state;
        1 reference
        public IEditorState State => _state;

        1 reference
        public EditorContext()
        {
            _state = new NormalState();
        }

        3 references
        public void SetState(IEditorState newState)
        {
            _state = newState;
        }

        3 references
        public string PerformAction()
        {
            return _state.OnAction();
        }
    }
}
```

Рисунок 2 – клас EditorContext.cs

EditorContext працює з поточним станом. Він зберігає активний стан (NormalState, EditState або PreviewState) і викликає в нього методи.

```
namespace Lumina.UI.States
{
    2 references
    public class NormalState : IEditorState
    {
        2 references
        public string Name => "Normal";

        2 references
        public string OnAction()
        {
            return "Editor is in normal mode. You can view and navigate the image.";
        }
    }
}
```

Рисунок 3 – клас NormalState.cs

```
namespace Lumina.UI.States
{
    1 reference
    public class EditState : IEditorState
    {
        2 references
        public string Name => "Edit";

        2 references
        public string OnAction()
        {
            return "Editor is in edit mode. You can apply transformations and effects.";
        }
    }
}
```

Рисунок 4 – клас EditState.cs

```
namespace Lumina.UI.States
{
    1 reference
    public class PreviewState : IEditorState
    {
        2 references
        public string Name => "Preview";

        2 references
        public string OnAction()
        {
            return "Editor is in preview mode. You can see the final image.";
        }
    }
}
```

Рисунок 5 – клас PreviewState.cs

NormalState – Користувач просто переглядає зображення.

EditState – Дозволено застосовувати ефекти, масштабувати, обертати.

PreviewState – Показується фінальний вигляд колажу або фото.

```
namespace Lumina.UI.Views
{
    2 references
    public partial class EditorPage : Page
    {
        private readonly EditorContext _context = new EditorContext();

        0 references
        public EditorPage()
        {
            InitializeComponent();
            Log($"Current state: {_context.State.Name}");
        }

        1 reference
        private void Normal_Click(object sender, RoutedEventArgs e)
        {
            _context.SetState(new NormalState());
            Log(_context.PerformAction());
        }

        1 reference
        private void Edit_Click(object sender, RoutedEventArgs e)
        {
            _context.SetState(new EditState());
            Log(_context.PerformAction());
        }

        1 reference
        private void Preview_Click(object sender, RoutedEventArgs e)
        {
            _context.SetState(new PreviewState());
            Log(_context.PerformAction());
        }

        4 references
        private void Log(string message)
        {
            LogBox.Items.Add(message);
        }
    }
}
```

Рисунок 6 – клас EditorPage.xaml.cs

На сторінці редактора є три кнопки: Normal, Edit, Preview. Коли користувач натискає кнопку:

- Контекст міняє стан;
- Стан повертає повідомлення;
- Повідомлення додається в лог (ListBox).

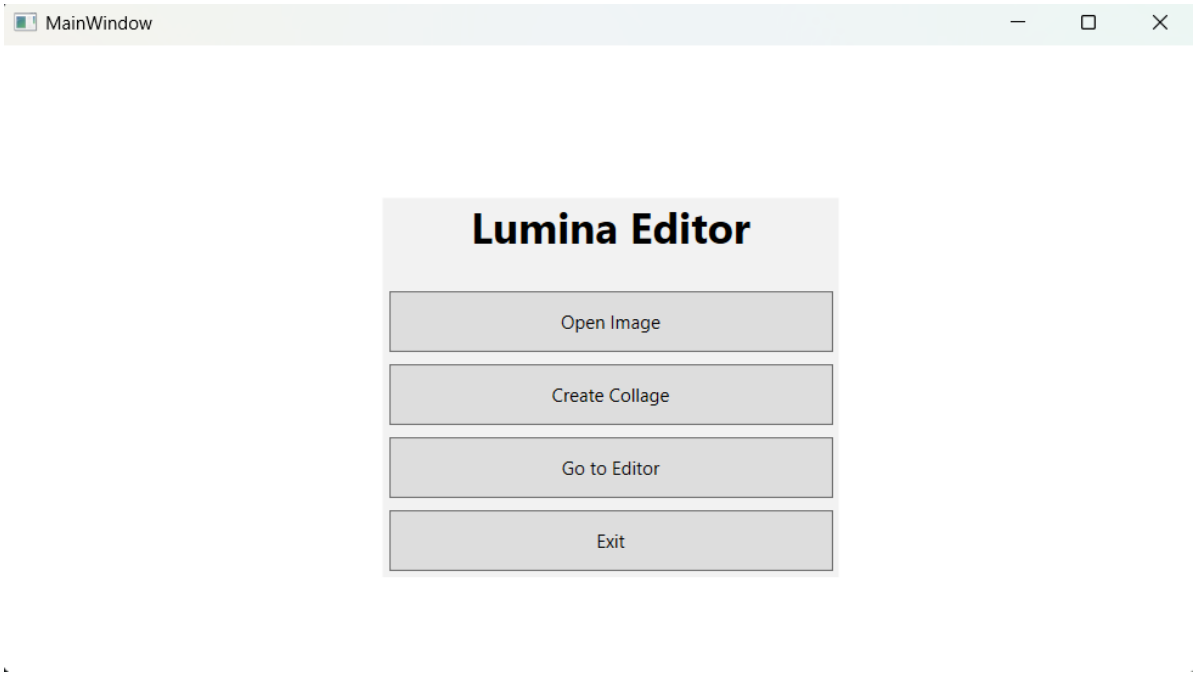

Рисунок 7 – Головна сторінка

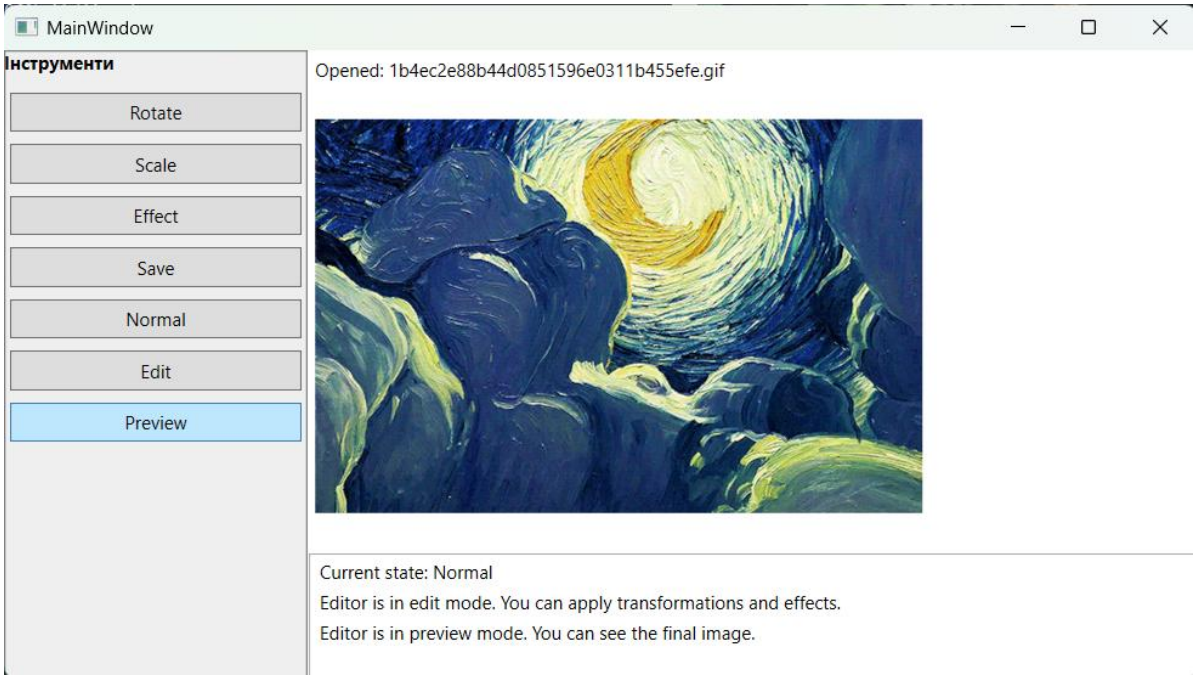

Рисунок 8 – Сторінка редактора

У цьому проєкті шаблон State дозволяє редактору зображень гнучко змінювати поведінку залежно від режиму роботи, не використовуючи складних умов. Кожен стан інкапсулює власну логіку, а EditorContext виступає посередником між користувачем та станами.

Висновки

Ефективність патерну State для управління поведінкою редактора зображень була практично підтверджена. Патерн дозволив уникнути використання складних умовних конструкцій та забезпечив чітку структуру коду.

Кожен стан інкапсулював власну логіку, що спростило розширення функціоналу. Гнучкість архітектури реалізована через розділення відповідальності між контекстом (EditorContext) та станами (NormalState, EditState, PreviewState). Контекст делегував виконання операцій поточному стану, що дозволило динамічно змінювати поведінку програми без змін у базовому коді.

Інтеграція з інтерфейсом користувача продемонструвала зручність патерну для UI-додатків. Перемикання режимів роботи через кнопки відображалося на доступності функцій редактора, що підвищило зрозумілість інтерфейсу для користувача.

Отже, патерн State є оптимальним рішенням для систем із динамічною зміною поведінки. Його реалізація в редакторі зображень забезпечила модульність, зрозумілість коду та готовність до майбутнього масштабування.

Контрольні питання

1. Що таке шаблон проєктування?

Шаблон проєктування – це універсальне, багаторазово використовуване рішення для типових проблем, що виникають під час розробки програмного забезпечення. Він описує підхід до організації коду, який дозволяє уникнути недоліків і покращити гнучкість системи.

2. Навіщо використовувати шаблони проєктування?

Стандартизація: Забезпечують єдиний підхід до вирішення задач.

Економія часу: Дозволяють використовувати готові рішення замість «винаходження велосипеда».

Підвищення якості: Зменшують ризик помилок через перевірені часом підходи.

Спрощення комунікації: Розробники використовують загальнозрозумілі назви патернів.

3. Яке призначення шаблону «Стратегія»?

Він дозволяє визначати сімейство алгоритмів, інкапсулювати їх у окремі класи та робити взаємозамінними. Клієнтський код може вибирати потрібний алгоритм без змін у своїй логіці.

4. Структура шаблону «Стратегія»:

Context містить посилання на інтерфейс Strategy

Strategy визначає спільний інтерфейс для всіх алгоритмів

ConcreteStrategy реалізують конкретні алгоритми

Контекст делегує виконання обраній стратегії

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

Інтерфейс Strategy: Оголошує метод виконання алгоритму (наприклад, execute()).

Конкретні стратегії (ConcreteStrategyA, B тощо): Реалізують алгоритми.

Контекст (Context): Містить посилання на стратегію та викликає її метод.

Взаємодія: Контекст делегує виконання алгоритму об’єкту стратегії. Зміна алгоритму відбувається шляхом підстановки іншої стратегії.

6. Яке призначення шаблону «Стан»?

Дозволяє об’єкту змінювати поведінку при зміні внутрішнього стану. Логіка кожного стану інкапсулюється в окремих класах.

7. Структура шаблону «Стан»:

Context зберігає посилання на поточний стан

State визначає інтерфейс для поведінки станів

ConcreteState реалізують поведінку, специфічну для стану

При зміні стану контекст змінює об'єкт стану

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

Інтерфейс State: Оголошує методи, специфічні для станів (наприклад, handle()).

Конкретні стани (ConcreteStateA, B тощо): Реалізують поведінку, властиву певному стану.

Контекст (Context): Має посилання на поточний стан і делегує йому виконання.

Взаємодія: При зміні стану контекст замінює об’єкт стану, що призводить до зміни поведінки.

9. Яке призначення шаблону «Ітератор»?

Надає спосіб послідовного доступу до елементів колекції без розкриття її внутрішньої структури.

10. Структура шаблону «Ітератор»

Aggregate визначає метод створення ітератора Iterator визначає інтерфейс для доступу до елементів ConcreteAggregate повертає конкретний ітератор ConcreteIterator реалізує алгоритм обходу колекції

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

Інтерфейс Iterator: Визначає методи для доступу до елементів (next(), hasNext() тощо).

Конкретний ітератор (ConcreteIterator): Реалізує логіку обходу колекції.

Інтерфейс Aggregate: Оголошує метод для створення ітератора (createIterator()).

Конкретна колекція (ConcreteAggregate): Повертає екземпляр ітератора.

Взаємодія: Клієнт отримує ітератор від колекції та використовує його для обходу елементів.

12. В чому полягає ідея шаблона «Одинак»?

Забезпечення існування лише одного екземпляра класу та надання глобального доступу до нього.

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

Порушення принципів ООП: Він часто використовується як глобальна змінна, що ускладнює тестування.

Приховані залежності: Класи, що використовують одинак, стають тісно пов’язаними.

Ускладнення підтримки: Важко відстежувати стан глобального об’єкта.

14. Яке призначення шаблону «Проксі»?

Створення об’єкта-замінника, який контролює доступ до реального об’єкта, додаючи додатковий функціонал (наприклад, кешування, логування).

15. Структура шаблону «Проксі»

Subject визначає спільний інтерфейс

RealSubject - реальний об'єкт, що виконує основну роботу

Proxy контролює доступ до RealSubject, додаючи додаткову функціональність

Клієнт працює з інтерфейсом Subject, не знаючи про використання проксі

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

Інтерфейс Subject: Оголошує спільні методи для реального об’єкта та проксі.

Реальний суб'єкт (RealSubject): Виконує основну логіку.

Проксі (Proxy): Контролює доступ до реального суб’єкта, додаючи додаткові операції. Взаємодія: Клієнт взаємодіє з проксі, який делегує виклики реальному об’єкту, якщо це необхідно.