МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО»

# ЛАБОРАТОРНА РОБОТА №5

ТЕМА: «Патерни проектування»

Виконала:

Студентка групи ІА-34

Потейчук С.А.

Перевірив:

Мягкий М.Ю.

Київ 2025

Зміст

# Теоретичні відомості

## Шаблон «Adapter» (Адаптер)

Шаблон "Адаптер" використовується для забезпечення співпраці об'єктів з несумісними інтерфейсами. Він дозволяє об'єктам з різними інтерфейсами працювати разом, створюючи проміжний шар між ними. Цей патерн також відомий як "обгортка" (wrapper), оскільки він обгортає оригінальний об'єкт і надає новий, очікуваний інтерфейс.

При розробці аудіо-плеєра, який підтримує різні формати аудіо, виникає складність у роботі з різними компонентами програвання. Кожен компонент має власний унікальний інтерфейс, що призводить до появи великої кількості умовних перевірок у коді. Це ускладнює читання та підтримку коду, а також створює ризик виникнення помилок при додаванні підтримки нових форматів.

Для вирішення проблеми визначається загальний інтерфейс IPlayer, який описує методи програвання аудіо. Для кожного компонента програвання створюється окремий клас-адаптер, який реалізує цей інтерфейс та інкапсулює логіку роботи з конкретним компонентом. Клієнтський код тепер працює виключно через інтерфейс IPlayer, не знаючи про особливості реалізації кожного компонента. При необхідності додати підтримку нового формату створюється новий адаптер без змін у існуючому коді.

Головною перевагою є відокремлення інтерфейсу від бізнес-логіки та можливість легко додавати нові адаптери без змін у клієнтському коді. Недоліком можна вважати збільшення кількості класів, однак це компенсується поліпшенням читабельності та структурованості коду.

## Шаблон «Builder» (Будівельник)

Шаблон "Будівельник" відокремлює процес створення складного об'єкта від його фінального представлення. Це дозволяє використовувати той самий процес конструювання для отримання різних представлень об'єкта. Патерн особливо корисний, коли об'єкт має багато складових частин або коли існує кілька варіантів його створення.

Розглянемо процес побудови HTTP-відповіді веб-сервера, який включає додавання заголовків, встановлення коду статусу, формування тіла відповіді та інші кроки. Без використання патерну цей процес може бути розкиданий по різних частинах коду, що ускладнює його контроль та модифікацію.

Кожен крок створення відповіді інкапсулюється в окремі методи класу-будівельника. Це забезпечує гнучкий контроль над процесом побудови та робить систему стійкішою до змін у внутрішній структурі об'єкта. Наприклад, зміна назви сервера не вимагатиме модифікації всього процесу побудови відповіді.

Основним плюсом є можливість використання одного й того ж коду для створення різних продуктів. Мінусом є потенційна прив'язаність клієнта до конкретних класів будівельників, особливо якщо інтерфейс будівельника не містить методу отримання результату.

Шаблон «Command» (Команда)

Шаблон "Команда" перетворює запит на виконання певної дії на самостійний об'єкт. Це дозволяє параметризувати клієнтські об'єкти різними запитами, затримувати або ставити в чергу виконання запитів, а також підтримувати операції скасування. Патерн особливо корисний у системах з розвиненою системою команд, таких як графічні редактори.

У графічному інтерфейсі користувача одна й та сама дія часто може бути викликана різними способами - через меню, кнопку чи контекстне меню. Без використання патерну код, що відповідає за виконання дії, доводиться дублювати, що ускладнює підтримку та тестування.

Функціонал кожної дії виділяється в окремий клас команди. Візуальні елементи інтерфейсу отримують посилання на об'єкти цих команд і викликають їх у відповідний момент. Це дозволяє уникнути дублювання коду, спростити тестування та реалізувати додаткову функціональність, таку як скасування дій чи їх логування.

До переваг належать розділення відповідальності між ініціатором та виконавцем команди, підтримка операцій скасування та логування, а також

легкість розширення системи новими командами. Недоліком може бути деяке ускладнення архітектури програми через велику кількість класів команд.

Шаблон «Chain of Responsibility» (Ланцюжок відповідальності)

Шаблон "Ланцюжок відповідальності" дозволяє передавати запити послідовно по ланцюжку обробників. Кожен наступний обробник вирішує, чи може він обробити запит, чи передати його далі по ланцюжку. Цей підхід схожий на реальний процес затвердження документів в організації, коли документ проходить через кілька рівнів керівництва.

При розробці складних графічних інтерфейсів з багаторівневою структурою елементів виникає необхідність динамічного формування контекстних меню. Кожен елемент інтерфейсу може додавати свої пункти до меню, причому ці пункти повинні враховувати ієрархію вкладеності елементів. Традиційний підхід з єдиним методом формування меню призводить до складних умовних конструкцій, які важко підтримувати та розширювати.

Кожен візуальний компонент реалізує інтерфейс з методом оновлення контекстного меню та містить посилання на батьківський елемент. При кліку правою кнопкою миші викликається метод формування меню для конкретного елемента, який додає свої пункти меню, а потім передає управління батьківському елементу. Таким чином, запит проходить по всій ієрархії елементів, і кожен рівень додає свої пункти меню. Це дозволяє легко змінювати структуру інтерфейсу без необхідності переписування логіки формування меню.

Основним перевагою є зменшення залежності між клієнтом та обробниками запитів, а також гнучкість у додаванні нових обробників. Недоліком є потенційна ситуація, коли запит може залишитися необробленим, якщо жоден з обробників не взяв на себе відповідальність за його обробку.

Шаблон «Prototype» (Прототип)

Шаблон "Прототип" дозволяє створювати нові об'єкти шляхом копіювання існуючого об'єкта-прототипу, замість створення екземплярів через конструктор. Це особливо корисно, коли процес створення об'єкта є складним або коли необхідно створювати об'єкти з певними початковими налаштуваннями.

При розробці редактора рівнів для гри виникає необхідність створювати багато однотипних об'єктів з невеликими відмінностями. Створення окремих класів для кожного типу об'єкта та відповідних їм кнопок інтерфейсу призводить до роздування коду та ускладнення його підтримки.

Кожен ігровий об'єкт отримує метод клонування, а кнопки інтерфейсу зберігають посилання на прототипи об'єктів. При натисканні на кнопку створюється копія прототипу, яка потім може бути модифікована. Це дозволяє уникнути створення складної ієрархії класів та спрощує додавання нових типів об'єктів.

Головними перевагами є підвищення продуктивності за рахунок уникнення складного процесу створення об'єктів та гнучкість у створенні варіацій об'єктів. Основним недоліком є складність реалізації глибокого копіювання для об'єктів зі складною структурою, а також потенційне ускладнення коду при надмірному використанні патерну.

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» та навчитися застосовувати їх в реалізації програмної системи.

## Хід роботи

7. Редактор зображень (state, prototype, memento, facade, composite, client server)

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.

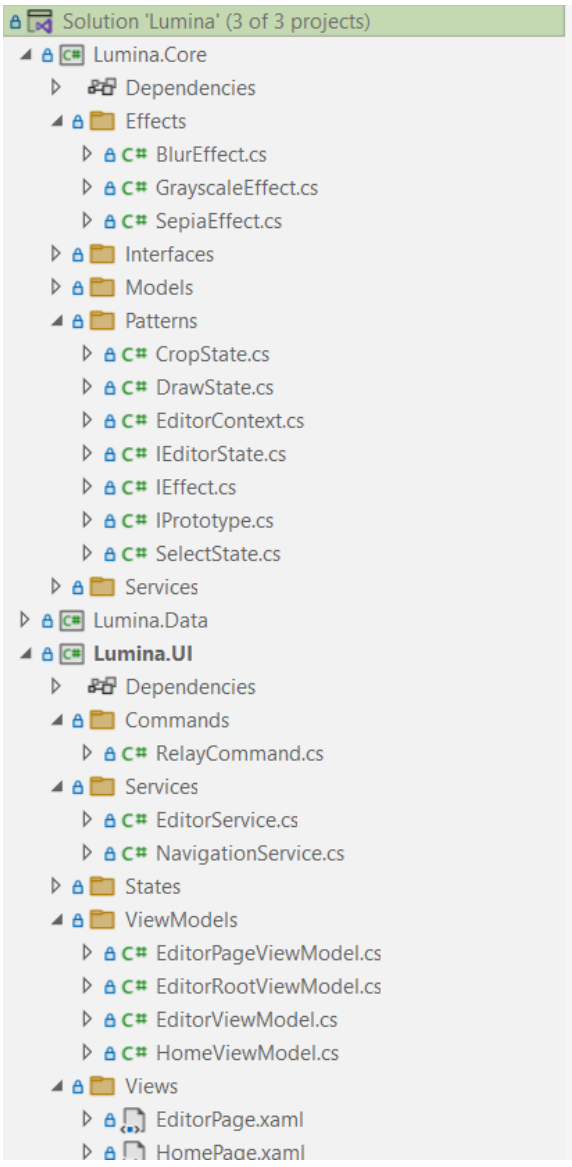



Рисунок 1 – Структура проєкту

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.

```
namespace Lumina.Core.Patterns
{
    7 references
    public interface IPrototype<T>
    {
        18 references
        T Clone();
    }
}
```

Рисунок 2 – Інтерфейс IPrototype.cs

Цей метод повертає повністю незалежну копію об’єкта.

```
4 references
public ImageLayer Clone()
{
    return new ImageLayer
    {
        Name = this.Name + " Copy",
        Image = this.Image,
        X = this.X,
        Y = this.Y,
        Width = this.Width,
        Height = this.Height,
        Rotation = this.Rotation,
        Opacity = this.Opacity,

        AppliedEffects = this.AppliedEffects
        .Select(e => e.CloneEffect())
        .ToList()
    };
}
```

Рисунок 3 – метод Clone в класі ImageLayer.cs

```
5 references
public class BlurEffect : IEffect, IPrototype<BlurEffect>
{
    1 reference
    public string Name => "Blur";
    2 references
    public int Radius { get; set; }

    2 references
    public BlurEffect(int radius)
    {
        Radius = radius;
    }

    2 references
    public BlurEffect Clone()
    {
        return new BlurEffect(this.Radius);
    }
```

Рисунок 4 – клас BlurEffect.cs

```
6 references
public class GrayscaleEffect : IEffect, IPrototype<GrayscaleEffect>
{
    1 reference
    public string Name => "Grayscale";
    3 references
    public double Strength { get; set; }
    1 reference
    public GrayscaleEffect() { Strength = 1.0; }

    1 reference
    public GrayscaleEffect(double strength)
    {
        Strength = strength;
    }

    2 references
    public GrayscaleEffect Clone()
    {
        return new GrayscaleEffect(this.Strength);
    }
    2 references
    public IEffect CloneEffect() => Clone();

}
```

Рисунок 5 – клас GrayscaleEffect.cs

```
6 references
public class SepiaEffect : IEffect, IPrototype<SepiaEffect>
{
    1 reference
    public string Name => "Sepia";
    3 references
    public double Amount { get; set; }
    1 reference
    public SepiaEffect() { Amount = 1.0; }

    1 reference
    public SepiaEffect(double amount)
    {
        Amount = amount;
    }

    2 references
    public SepiaEffect Clone()
    {
        return new SepiaEffect(this.Amount);
    }
```

Рисунок 6 – клас SepiaEffect.cs

Кожен ефект реалізує IPrototype, тому може бути скопійований перед застосуванням до шару.

Ефекти мають власні параметри (радіус розмиття, сила сепії), і вони створюються як окремі об’єкти для кожного шару.

Переваги застосування Prototype в проєкті:

Уникається дублювання коду, оскільки не потрібно щоразу створювати об’єкти через складні конструктори.

Зменшується ризик помилок, коли один ефект випадково використовується повторно кількома шарами.

Спрощується структуру редактора, оскільки логіка копіювання прихована всередині об’єктів.

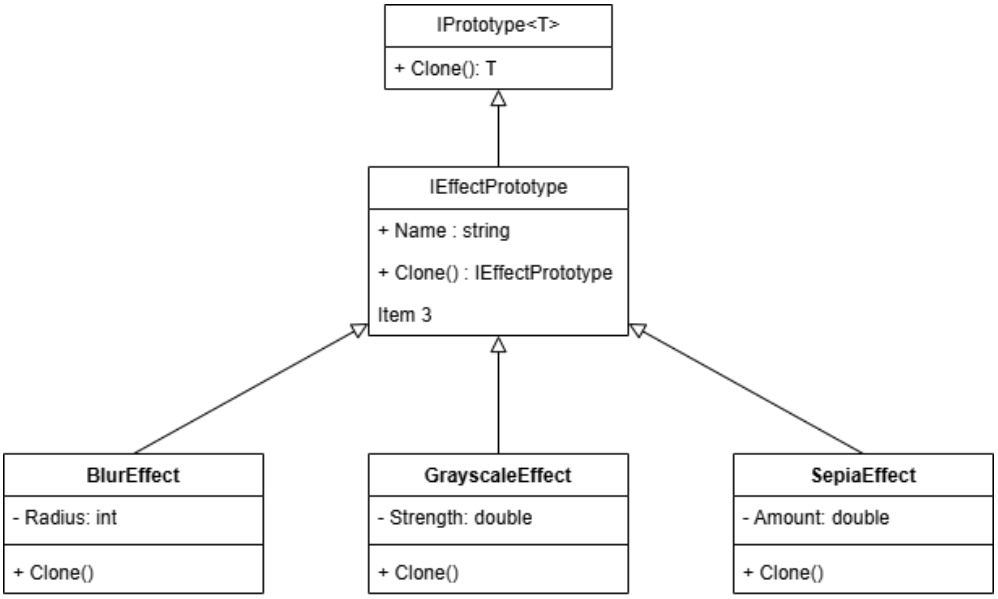



Рисунок 7 – Діаграма класів, яка представляє використання шаблону в реалізації системи

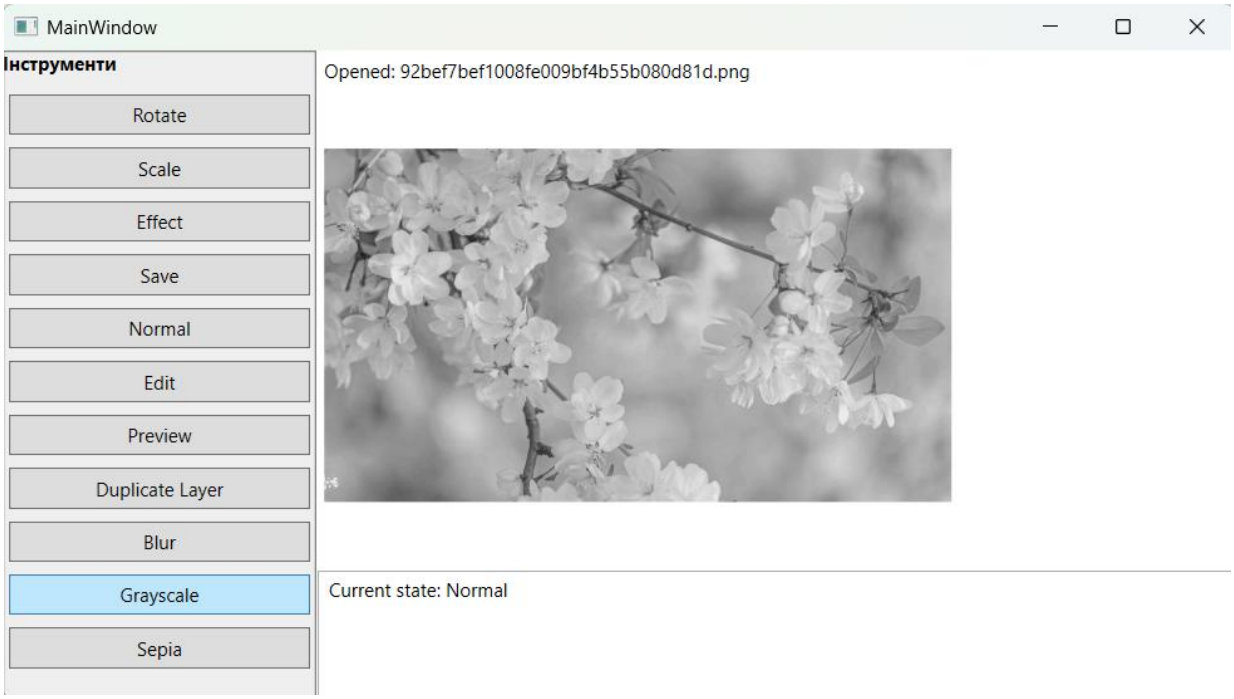



Рисунок 8 – Сторінка редактора (Застосування ефекту Grayscale)

При застосуванні ефекту спочатку створюється його копія через Clone(), після чого ця копія додається до списку ефектів шару. Це гарантує, що кожен застосований ефект має власні параметри і не впливає на інші шари.

Коли користувач натискає “Duplicate Layer”, створюється повна копія шару зі всіма ефектами.

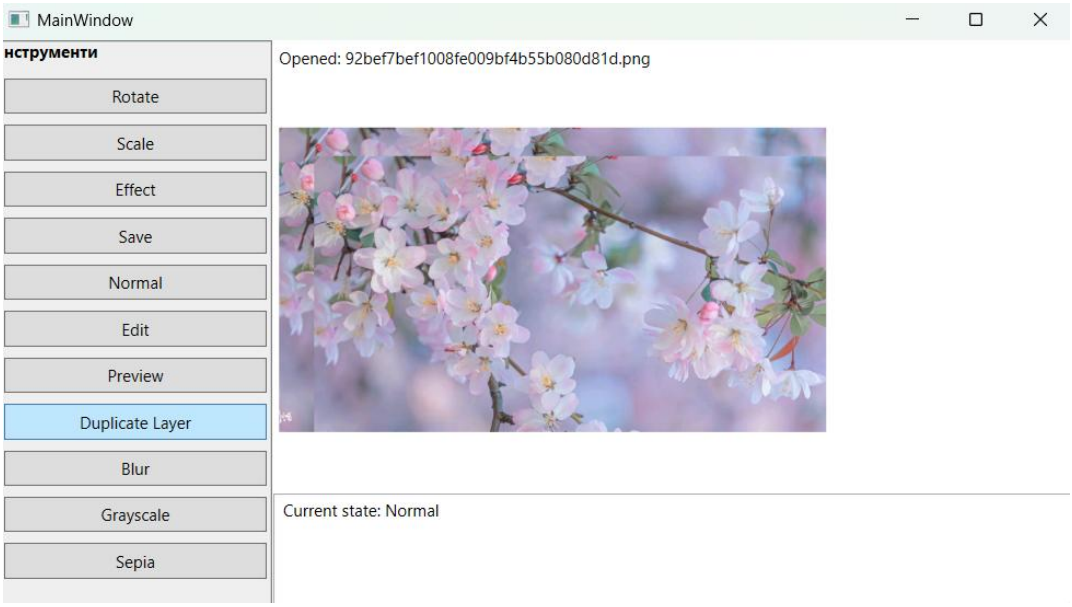



Рисунок 8 – Сторінка редактора (Застосування Duplicate Layer)

## Висновки

У ході лабораторної роботи було досліджено та реалізовано патерн проєктування Prototype, який призначений для створення копій об’єктів без необхідності повторного виклику їхніх конструкторів або ручного відтворення структури. Застосування цього патерну в редакторі зображень показало, що він є ефективним рішенням для роботи з об’єктами, що мають складну внутрішню структуру, зокрема з ефектами та шарами.

Prototype дозволив реалізувати механізм глибокого копіювання шарів зображення разом із прив’язаними до них ефектами. Це забезпечує незалежність копій, що є критично важливим під час редагування: дубльований шар може змінюватися без впливу на оригінал, а ефекти після копіювання зберігають власні параметри й не розділяються між різними елементами.

У процесі роботи було продемонстровано, що патерн Prototype спрощує структуру коду, усуває дублювання логіки створення об’єктів та зменшує кількість залежностей у системі. Завдяки Clone()-методам об’єкти самостійно відповідають за копіювання своїх даних, що робить систему більш гнучкою до змін та розширення.

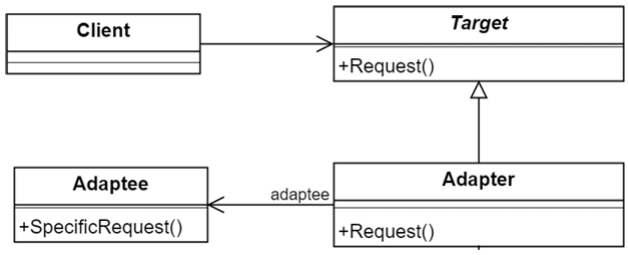
Завдяки використанню Prototype редактор стає більш гнучким, простішим у підтримці та стійким до майбутнього розширення функціональності.

Контрольні питання

1. Яке призначення шаблону «Адаптер»?

Призначення шаблону «Адаптер» – забезпечити спільну роботу об'єктів із несумісними інтерфейсами. Він перетворює інтерфейс одного класу на інший, очікуваний клієнтом, дозволяючи їм взаємодіяти, не змінюючи власного коду. По суті, це міст між двома різними інтерфейсами.

2. Нарисуйте структуру шаблону «Адаптер».




3. Які класи входять в шаблон «Адаптер», та яка між ними взаємодія?

- Клієнт (Client): Клас, який використовує об'єкти через Цільовий Інтерфейс.
- Цільовий Інтерфейс (Target): Інтерфейс, який очікує побачити Клієнт. Адаптер реалізує цей інтерфейс.
- Адаптер (Adapter): Клас, який реалізує Цільовий Інтерфейс. Він містить посилання на об'єкт Адаптованого Класу та "перекладає" виклики методів Цільового Інтерфейсу на зрозумілий Адаптованому Класу формат.
- Адаптований Клас (Adaptee): Існуючий клас з несумісним інтерфейсом, який потрібно інтегрувати.
- Взаємодія: Клієнт викликає метод Цільового Інтерфейсу в Адаптера. Адаптер делегує цей виклик об'єкту Адаптованого Класу, перетворюючи параметри та формат виклику.

4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

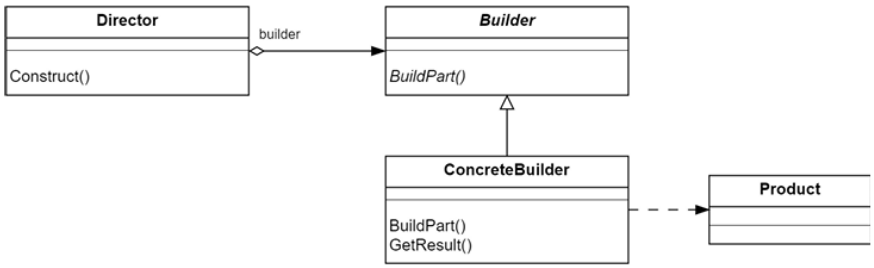
Адаптер на рівні об'єктів: Використовує композицію. Адаптер містить екземпляр Адаптованого Класу як поле. Ця реалізація більш гнучка, оскільки дозволяє адаптувати не лише один клас, а й його нащадків.

Адаптер на рівні класів: Використовує множинне наслідування. Адаптер успадковується від Адаптованого Класу та реалізує Цільовий Інтерфейс. Це дозволяє перевизначати методи Адаптованого Класу, але жорстко прив'язує Адаптер до конкретного класу.

5. Яке призначення шаблону «Будівельник»?

Призначення шаблону «Будівельник» – відокремити конструювання складного об'єкта від його представлення, щоб один і той самий процес конструювання міг створювати різні представлення. Він використовується для покрокового створення об'єкта, дозволяючи використовувати той самий код будівництва для отримання різних типів об'єктів.

6. Нарисуйте структуру шаблону «Будівельник».




7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?

- Будівельник (Builder): Інтерфейс для створення частин продукту.
- Конкретний Будівельник (Concrete Builder): Реалізує інтерфейс Будівельника, надає конкретні реалізації кроків побудови. Відповідає за створення та комплектацію конкретного продукту.
- Директор (Director): Контролює процес побудови, використовуючи інтерфейс Будівельника. Він знає, в якій послідовності викликати кроки побудови.
- Продукт (Product): Складний об'єкт, що створюється.
- Взаємодія: Клієнт створює об'єкт Конкретного Будівельника і передає його Директору. Директор пошагово викликає методи будівництва. Після завершення процесу Клієнт отримує готовий Продукт від Будівельника.

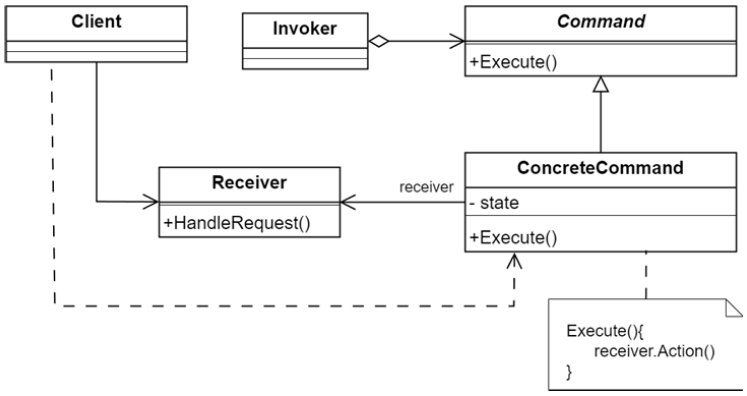
8. У яких випадках варто застосовувати шаблон «Будівельник»?

- Коли процес створення складного об'єкта має бути незалежним від частин, що утворюють об'єкт, і від способу їх комплектації.
- Коли необхідно створювати різні представлення одного об'єкта.
- Коли алгоритм створення складного об'єкта має розглядатися незалежно від частин, що утворюють цей об'єкт.

9. Яке призначення шаблону «Команда»?

Призначення шаблону «Команда» (Command) – інкапсулювати запит на виконання певної дії у вигляді об'єкта. Це дозволяє параметризувати клієнтські об'єкти різними запитами, ставити запити в чергу, підтримувати операції скасування (undo).

10. Нарисуйте структуру шаблону «Команда».




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

Інтерфейс Команди (Command): Оголошує метод виконання (наприклад, execute()).

Конкретна Команда (Concrete Command): Реалізує метод виконання, зв'язуючи дію з Отримувачем. Може зберігати параметри для дії та стан для скасування.

Отримувач (Receiver): Об'єкт, який знає, як виконати саму дію. Будь-який клас може виступати в ролі Отримувача.

Викликач (Invoker): Зберігає команду та ініціює її виконання.

Клієнт (Client): Створює конкретну команду та налаштовує її з Отримувачем.

Взаємодія: Клієнт створює об'єкт Конкретної Команди та пов'язує його з Отримувачем. Викликач зберігає посилання на цю команду. У певний момент (натискання кнопки) Викликач викликає метод execute() команди. Команда, у свою чергу, викликає відповідні методи Отримувача.

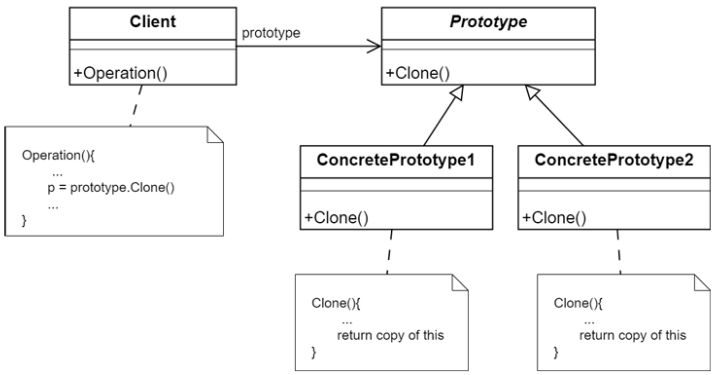
12. Розкажіть як працює шаблон «Команда».

1. Створення: Клієнт створює об'єкт Конкретної Команди, передаючи йому всі необхідні параметри (включаючи посилання на Отримувача).
2. Налаштування: Клієнт передає створену команду об'єкту-Викликачу (наприклад, встановлює команду для кнопки).
3. Виклик: У певний момент Викликач викликає метод execute() команди. Виконання: Команда виконує необхідні дії, викликаючи один або кілька методів Отримувача.
4. Скасування (опціонально): Якщо реалізовано, команда може зберігати стан, необхідний для скасування операції, і мати метод undo().

13. Яке призначення шаблону «Прототип»?

Призначення шаблону «Прототип» – створення нових об'єктів шляхом копіювання існуючого об'єкта-прототипу, замість створення через конструктор. Це дозволяє створювати об'єкти з певними початковими станами, уникнути складнощів створення об'єктів і зменшити залежність від конкретних класів.

14. Нарисуйте структуру шаблону «Прототип».



15. Які класи входять в шаблон «Прототип», та яка між ними взаємодія?

- Прототип (Prototype): Оголошує інтерфейс для клонування самого себе (метод clone).
- Конкретний Прототип (Concrete Prototype): Реалізує операцію клонування, створюючи копію самого себе.
- Клієнт (Client): Створює новий об'єкт, вимагаючи від прототипу клонувати самого себе.
- Взаємодія: Клієнт отримує посилання на об'єкт-прототип (може зберігатися в реєстрі прототипів) і викликає його метод clone(). Об'єкт-прототип створює і повертає свою точну копію.

16. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?

- Система контекстних меню в GUI: При кліку правою кнопкою миші запит на формування меню передається по ланцюжку від конкретного елемента до його контейнерів. Кожен елемент може додати свої пункти меню.
- Обробка HTTP-запитів у веб-фреймворках: Запит проходить через ланцюжок middleware (компонентів), кожен з яких виконує свою задачу: аутентифікацію, логування, компресію, валідацію тощо.
- Системи логування: Лог-повідомлення передаються ланцюжком обробників, кожен з яких вирішує, куди його записати (консоль, файл, база даних, email) залежно від рівня важливості (debug, info, error).
- Обробка платежів: Запит на оплату може спочатку перевірятися на валідність, потім списувати кошти з основного рахунку, а якщо їх не вистачає – з резервного, і так далі.