Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Лабораторна робота №5

«Патерни проектування»

З дисципліни «**Технології розроблення програмного забезпечення**»

Виконав:

Студент групи ІА-31

Дук М. Д.

Перевірив:

Мягкий М. Ю.

Київ 2025

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

Теоретичні відомості

Шаблон «Адаптер» (Adapter)

- **Призначення:** Дозволяє клієнтському коду працювати з різними специфічними компонентами через єдиний стандартизований інтерфейс (адаптер). Клієнт взаємодіє лише з адаптером і не знає про внутрішні, несумісні інтерфейси компонентів, які він обгортає.
- **Переваги:**
    - Відокремлює інтерфейс або код перетворення даних від основної бізнес-логіки.
    - Дозволяє додавати підтримку нових компонентів (адаптерів), не змінюючи існуючий клієнтський код (принцип відкритості-закритості).
- **Недоліки:**
    - Збільшення загальної кількості класів у системі.

Шаблон «Будівельник» (Builder)

- **Призначення:** Відділяє процес створення складного об'єкту від його кінцевого представлення. Це дозволяє використовувати той самий процес конструювання для створення різних варіацій об'єкта.
- **Переваги:**
    - Дозволяє використовувати один і той самий код конструювання для створення різноманітних представлень продукту.
- **Недоліки:**
    - Клієнт може бути прив’язаний до конкретних класів будівельників, оскільки загальний інтерфейс будівельника може не містити методу для отримання готового результату.

Шаблон «Команда» (Command)

- **Призначення:** Інкапсулює запит або виклик методу як повноцінний об'єкт. Це дозволяє параметризувати об'єкти різними діями, ставити команди в чергу, логувати їх, а також реалізовувати операції скасування (Undo).
- **Переваги:**
    - Ініціатор (наприклад, кнопка UI) не знає деталей реалізації виконавця команди.
    - Нативна підтримка операцій скасування (Undo) та повторення (Redo).
    - Можливість логування послідовності команд для подальшого відтворення.
    - Легке розширення системи шляхом додавання нових класів команд.

Шаблон «Ланцюжок відповідальності» (Chain of Responsibility)

- **Призначення:** Дозволяє передавати запит вздовж ланцюжка потенційних обробників, доки один з них не опрацює цей запит.
- **Переваги:**
    - Зменшує залежність між клієнтом (відправником запиту) та обробниками. Клієнт не знає, хто саме обробить запит.
    - Дозволяє гнучко додавати або видаляти обробники з ланцюжка під час виконання.
- **Недоліки:**
    - Запит може залишитися ніким не опрацьованим, якщо жоден об'єкт у ланцюжку не зможе його обробити.

Шаблон «Прототип» (Prototype)

- **Призначення:** Використовується для створення нових об'єктів шляхом копіювання (клонування) існуючого об'єкта-шаблону (прототипу), замість створення об'єкта "з нуля" через конструктор.
- **Переваги:**
    - Підвищує продуктивність, якщо процес створення об'єкта є "важким" (наприклад, вимагає багато ресурсів або часу).

    - Дозволяє отримувати різні варіації об'єктів шляхом клонування та модифікації копій, уникаючи розширення ієрархії класів.
    - Клоновані об'єкти можна модифікувати незалежно, не впливаючи на оригінальний прототип.
- **Недоліки:**
    - Реалізація глибокого клонування (deep copy) може бути складною, якщо об'єкт містить посилання на інші об'єкти.
    - Надмірне використання патерну може призвести до ускладнення коду.

Хід Роботи

Шаблон "Adapter" (Адаптер) використовується для адаптації інтерфейсу одного об'єкту до іншого. Цей патерн дозволяє класам з несумісними інтерфейсами працювати разом, не змінюючи їх вихідний код.

У системі архівування є необхідність підтримки різних форматів архівів (ZIP, TAR, тощо), кожен з яких має власний API з різними методами та інтерфейсами. Пряме використання цих бібліотек у клієнтському коді призводить до:

- Складної логіки з перевіркою типів архівів
- Дублювання коду
- Труднощів при додаванні нових форматів
- Порушення принципу відкритості/закритості (Open/Closed Principle)

Рішення

Використання патерну Adapter дозволяє:

1. Створити уніфікований інтерфейс IArchiver для всіх типів архівів
2. Реалізувати адаптери для кожного формату архіву
3. Інкапсулювати специфічну логіку роботи з різними бібліотеками
4. Забезпечити легке розширення новими форматами

У проєкті шаблон «Адаптер» використано для уніфікації роботи з архівами різних форматів (.zip та .tar). Замість того, щоб клієнтський код (наприклад,

ArchiveManager або MainForm) містив складну логіку для розрізнення типів архівів, ми визначаємо єдиний інтерфейс і створюємо класи-адаптери для кожного формату.

Ключові компоненти шаблону в проєкті:

1. Target (Цільовий інтерфейс): Це інтерфейс IArchiveStrategy. Він визначає єдиний набір операцій, які клієнтський код може виконувати над будь-яким архівом, незалежно від його формату (створення, додавання файлів, видалення, читання тощо).
2. Adaptee (Об'єкти, що адаптуються): Це конкретні класи з бібліотек .NET для роботи з архівами. У нашому випадку це System.IO.Compression.ZipArchive для .zip архівів та System.Formats.Tar.TarReader / System.Formats.Tar.TarWriter для .tar архівів. Вони мають різні API, несумісні між собою та з нашим інтерфейсом IArchiveStrategy.
3. Adapter (Адаптери): Це класи ZipAdapter та TarAdapter. Вони реалізують цільовий інтерфейс IArchiveStrategy. Всередині своїх методів кожен адаптер "обгортає" логіку викликів до відповідного Adaptee (ZipArchive або Tar*), транслюючи універсальні запити від клієнта у специфічні команди для конкретної бібліотеки.
4. Client (Клієнт): Це класи ArchiveManager та ArchiveFactory. ArchiveFactory виступає як "постачальник" адаптерів: він аналізує розширення файлу і повертає потрібний екземпляр адаптера (ZipAdapter або TarAdapter), але з типом IArchiveStrategy. ArchiveManager (і, як наслідок, MainForm) працює виключно з інтерфейсом IArchiveStrategy, не знаючи, з яким саме форматом архіву він має справу.

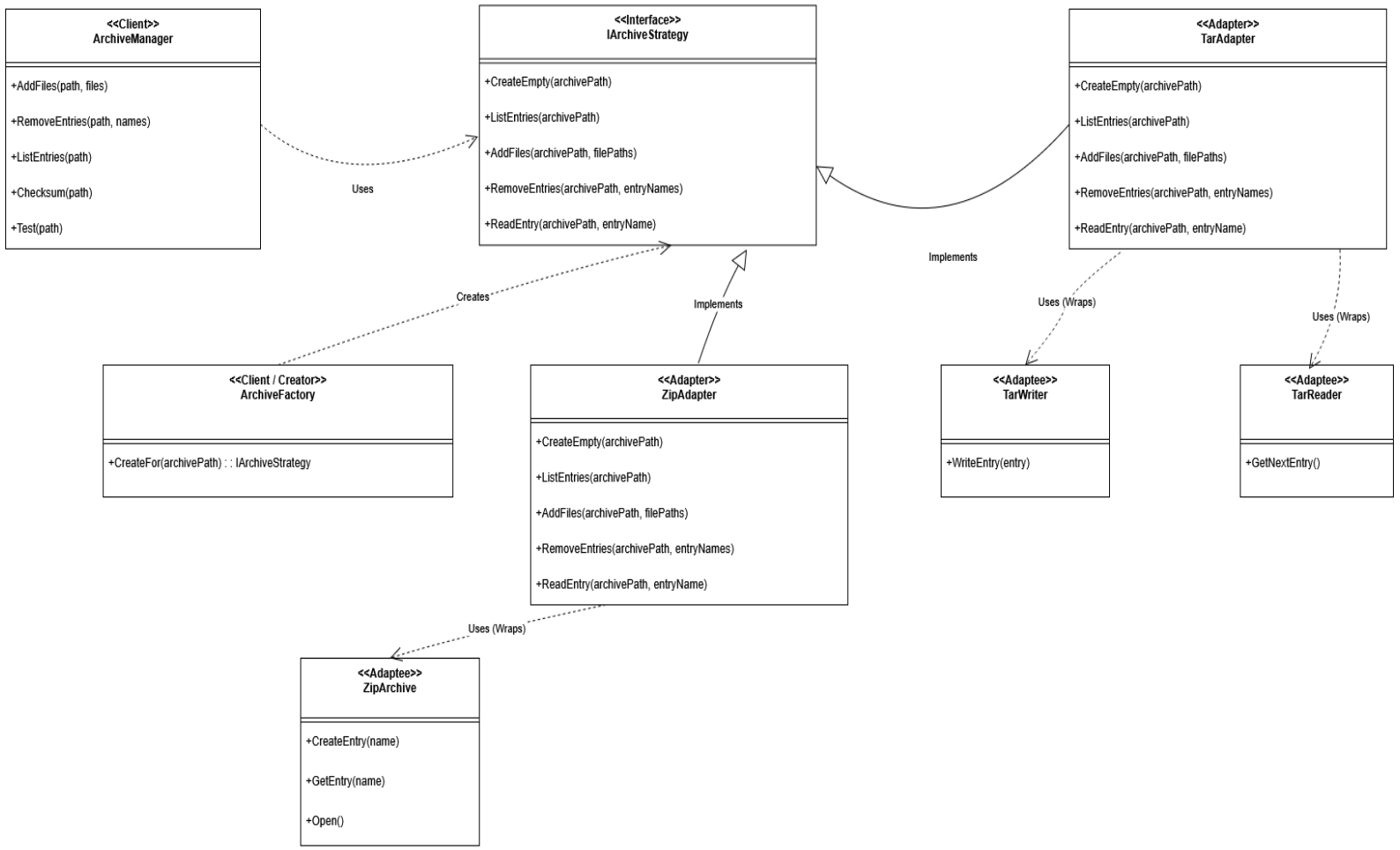
Переваги реалізації

- **Відокремлення інтерфейсу** - код перетворення даних відокремлений від основної бізнес-логіки
- **Розширюваність** - можна додавати нові адаптери без зміни клієнтського коду
- **Уніфікація** - всі типи архівів використовують однаковий інтерфейс
- **SOLID принципи** - дотримання принципів єдиної відповідальності та відкритості/закритості

Недоліки

- **Збільшення кількості класів** - додаткові класи адаптерів
- **Складність** - додаткові рівні абстракції

Діаграма Класів реалізованого шаблону



[Посилання на діаграму]

Пояснення діаграми:

- ArchiveManager (Клієнт) містить посилання на IArchiveStrategy (Цільовий інтерфейс).
- ArchiveFactory (Клієнт) створює конкретні адаптери (ZipAdapter або TarAdapter), але повертає їх клієнту як IArchiveStrategy.
- ZipAdapter і TarAdapter (Адаптери) реалізують IArchiveStrategy.
- Кожен адаптер всередині себе використовує відповідні системні класи (Adaptee), такі як ZipArchive або TarWriter, для виконання реальної роботи.

Код реалізованого шаблону

```
namespace ArchiverApp.Core.Facade
{
    using System.Collections.Generic;
    using System.IO;
    using System.Linq;
    using ArchiverApp.Data;

    4 usages  PriMerro23
    public class ArchiveManager
    {
        2 usages  PriMerro23
        public static void CreateEmptyArchive(string path){...}

        1 usage  PriMerro23
        public IEnumerable<string> ListEntries(string path){...}

        1 usage  PriMerro23
        public void AddFiles(string path, IEnumerable<string> files){...}

        1 usage  PriMerro23
        public void RemoveEntries(string path, IEnumerable<string> names){...}

        1 usage  PriMerro23
        public Dictionary<string, string> Checksum(string path){...}

        1 usage  PriMerro23
        public bool Test(string path){...}

        1 usage  PriMerro23
        public void SplitArchive(string path, string outputFolder, int partSizeBytes){...}

        1 usage  PriMerro23
        public void ExtractAll(string path, string destination){...}
    }
}
```

Рис. 1 –ArchiveManager.cs

```
namespace ArchiverApp.Core
{
    7 usages  PriMerro23
    public static class ArchiveFactory
    {
        7 usages  PriMerro23
        public static IArchiveStrategy CreateFor(string archivePath){...}
    }
}
```

Рис. 2 –ArchiveFactory.cs

```
namespace ArchiverApp.Core
{
    2 usages  3 inheritors  PriMerro23  1 exposing API
    public interface IArchiveStrategy
    {
        1 usage  2 implementations  PriMerro23
        void CreateEmpty(string archivePath);
        6 usages  2 implementations  PriMerro23
        IEnumerable<string> ListEntries(string archivePath);
        1 usage  2 implementations  PriMerro23
        void AddFiles(string archivePath, IEnumerable<string> filePaths);
        1 usage  2 implementations  PriMerro23
        void RemoveEntries(string archivePath, IEnumerable<string> entryNames);
        3 usages  2 implementations  PriMerro23
        byte[] ReadEntry(string archivePath, string entryName);
    }

    2 usages  2 inheritors  PriMerro23
    public interface IArchiveAdapter : IArchiveStrategy { }

    2 usages  2 inheritors  PriMerro23
    public interface IArchiveVisitor
    {
        2 implementations  PriMerro23
        void Visit(string entryName, byte[] content);
    }
}
```

Рис. 3 –IArchiveStrategy.cs

```
namespace ArchiverApp.Core
{
    1 usage  PriMerro23
    public class TarAdapter : IArchiveAdapter
    {
        0+1 usages  PriMerro23
        public void CreateEmpty(string archivePath){...}

        0+6 usages  PriMerro23
        public IEnumerable<string> ListEntries(string archivePath){...}

        0+1 usages  PriMerro23
        public void AddFiles(string archivePath, IEnumerable<string> filePaths){...}

        0+1 usages  PriMerro23
        public void RemoveEntries(string archivePath, IEnumerable<string> entryNames)
        {
            var temp :string  = Path.GetTempFileName();
            var keep :List<(name,path)>  = ExtractAllToTemp(archivePath).Where(e :(name,path)  => !entryNames.Contains(e.name)
            using (var outFs :FileStream  = File.Create(temp))
            using (var tar = new TarWriter(outFs))
            {...}
            File.Copy( sourceFileName: temp,  destFileName: archivePath, overwrite: true);
            File.Delete(temp);
        }

        0+3 usages  PriMerro23
        public byte[] ReadEntry(string archivePath, string entryName){...}

        2 usages  PriMerro23
        private List<(string name, string path)> ExtractAllToTemp(string archivePath){...}
    }
}
```

Рис. 4 –TarAdapter, TarReader TarWriter.

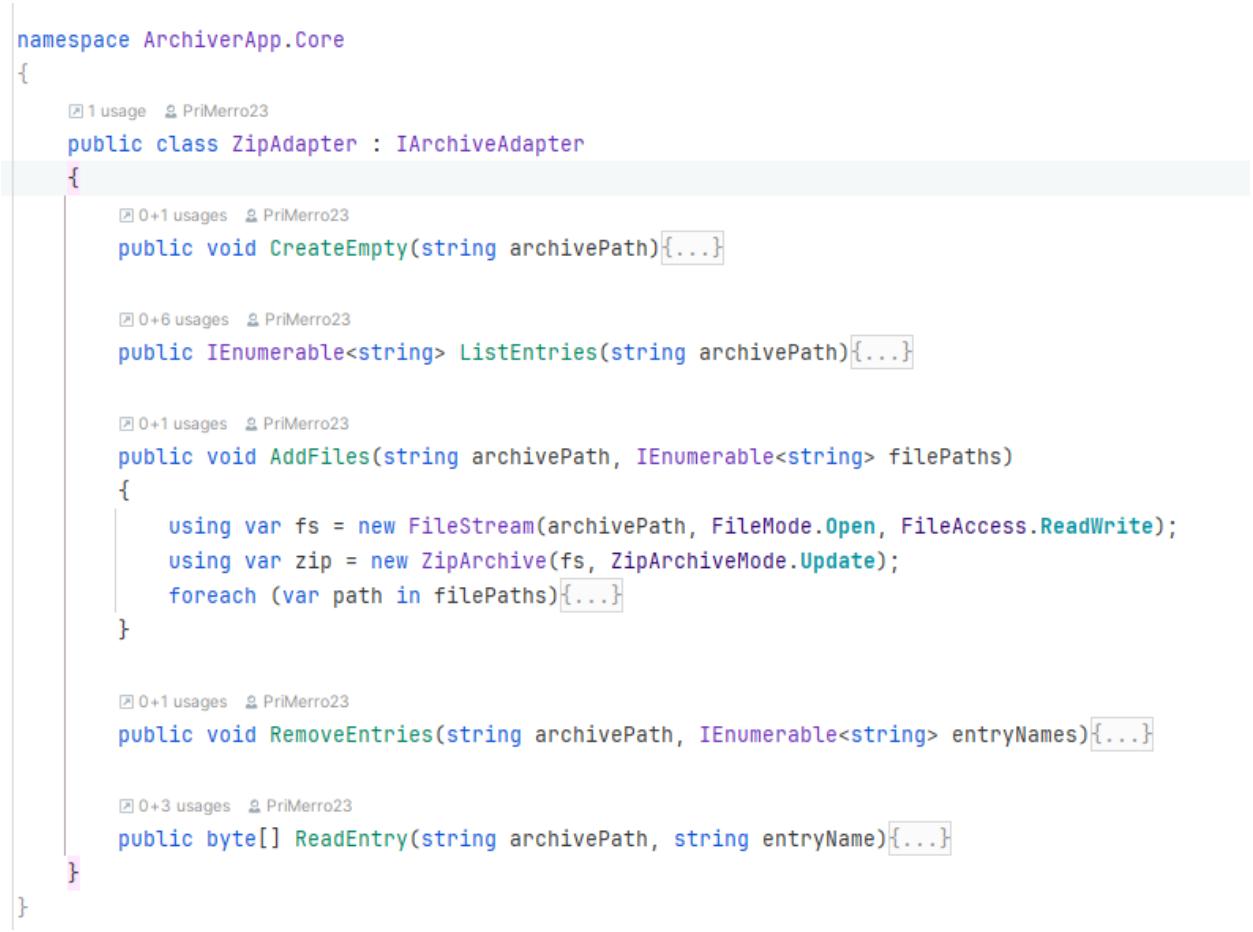


Рис. 5 –ZipAdapter, ZipArchive.

Висновок: Під час виконання лабораторної роботи я ознайомився з теоретичними основами шаблону проєктування «Адаптер» та успішно застосував його на практиці у проєкті.

Було реалізовано уніфікований інтерфейс IArchiveStrategy, який визначає загальний контракт для роботи з архівами. Для підтримки конкретних форматів (.zip та .tar) були створені класи-адаптери ZipAdapter та TarAdapter, які реалізують цей інтерфейс. Кожен адаптер "обгортає" виклики до специфічних системних бібліотек (ZipArchive та TarWriter/TarReader відповідно), приховуючи складність їх реалізації від решти програми.

Використання шаблону «Адаптер» дозволило значно спростити клієнтський код (клас ArchiveManager), який тепер працює з усіма типами архівів однаково через інтерфейс IArchiveStrategy. Це відповідає принципу відкритості/закритості: для додавання підтримки нового формату архіву

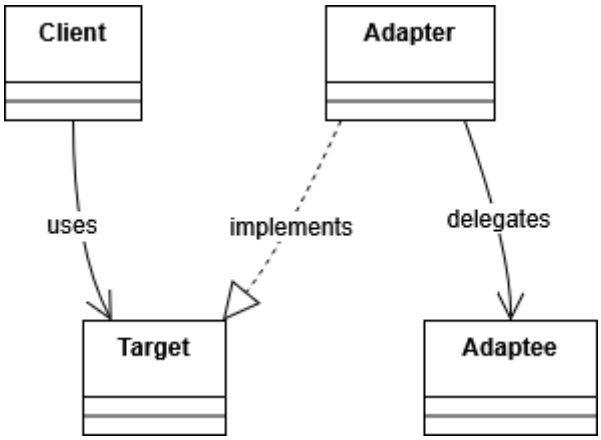
(наприклад, .rar або .7z) достатньо буде створити новий клас-адаптер, не вносячи жодних змін у вже існуючий та протестований код.

Питання до лабораторної роботи

1. Яке призначення шаблону «Адаптер»?

Шаблон «Адаптер» перетворює інтерфейс наявного класу до очікуваного клієнтом, дозволяючи взаємодіяти об’єктам із несумісними інтерфейсами без зміни їхнього коду. Це часто використовують для інтеграції сторонніх бібліотек або легасі-коду через уніфікований інтерфейс і «обгортки»

2. Нарисуйте структуру шаблону «Адаптер».



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

- Client: використовує інтерфейс Target, не знаючи деталей адаптації.
- Target: інтерфейс, який очікує клієнт.
- Adaptee: існуючий клас із «несумісним» інтерфейсом.
- Adapter: реалізує Target і делегує виклики до Adaptee, транслюючи запити.

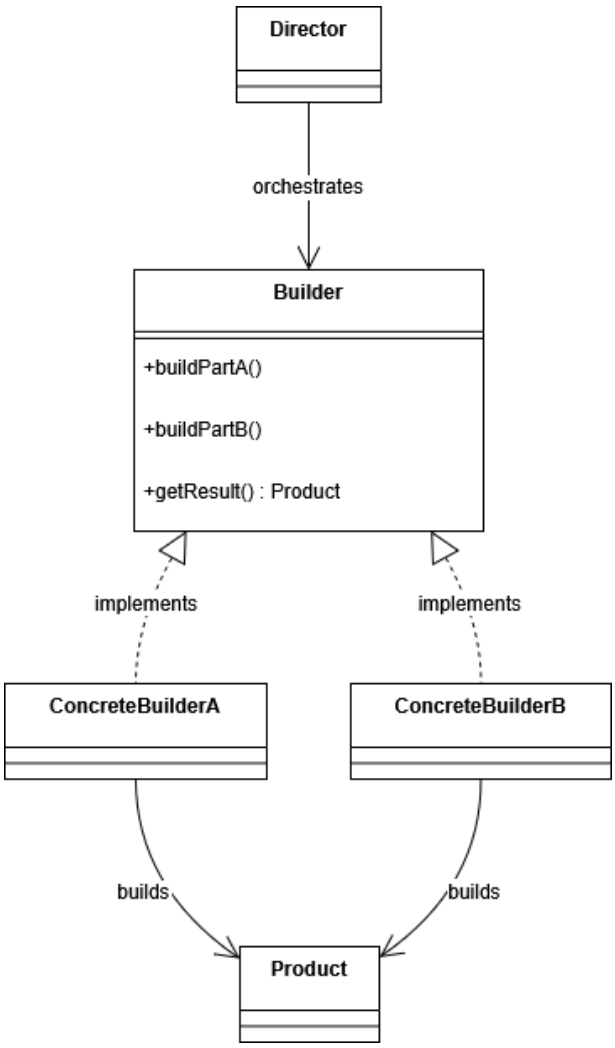
4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

Критерій	**Об’єктний адаптер (композиція)**	**Класовий адаптер (успадкування)**
Механізм	Тримає посилання на Adaptee і делегує виклики.	Наслідує Target і Adaptee для адаптації інтерфейсу.
Гнучкість	Може працювати з кількома Adaptee і замінювати їх під час виконання.	Жорстко зв’язаний із конкретним Adaptee через спадкування.
Підтримка мовою	Підходить для Java/C#, де немає множинного успадкування.	Потрібне множинне успадкування(C++).
Складність	Просте впровадження й тестування.	Складніша ієрархія, ризик ламкого зв’язування.

5. Яке призначення шаблону «Будівельник»?

Шаблон «Будівельник» відокремлює процес побудови складного об’єкта від його подання, щоб той самий процес міг створювати різні представлення продукту. Це корисно, коли є багато кроків/варіантів складання або численні опційні параметри, які потрібно налаштовувати поетапно.

6. Нарисуйте структуру шаблону «Будівельник».



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

- Product: складний об’єкт, який збирається по кроках.
- Builder: інтерфейс кроків побудови частин продукту.
- ConcreteBuilder: реалізує кроки для конкретної варіації продукту й повертає результат.
- Director: визначає порядок виклику кроків і координацію побудови.
- Client: ініціює побудову і забирає готовий продукт.

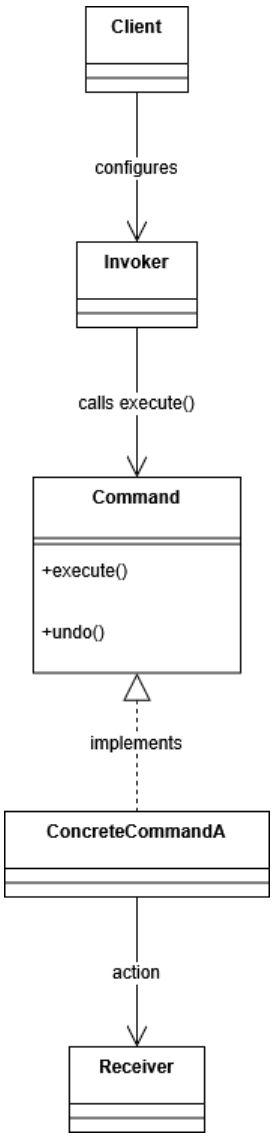
8. У яких випадках варто застосовувати шаблон «Будівельник»"?

- Потрібно створювати складні об’єкти поетапно з різними конфігураціями чи представленнями.
- Є багато опційних параметрів та ризик «телескопічних конструкторів», що погіршує читабельність.
- Потрібен стабільний процес побудови, незалежний від внутрішніх змін подання продукту.

9. Яке призначення шаблону «Команда»?

Шаблон «Команда» інкапсулює запит у окремий об’єкт, відв’язуючи ініціатора від отримувача та спрощуючи додавання скасування, чергування й логування.Це дає змогу параметризувати виклики, будувати історії операцій і виконувати їх у потрібний час або повторно.

10. Нарисуйте структуру шаблону «Команда».



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

- Command: загальний інтерфейс із методом виконання (наприклад, execute).
- ConcreteCommand: зберігає посилання на Receiver і параметри дії, реалізуючи execute.
- Receiver: знає, як виконати конкретну операцію.
- Invoker: ініціює виконання команд і може вести облік/режими.
- Client: збирає команди, задає отримувачів і передає команди викликачеві.

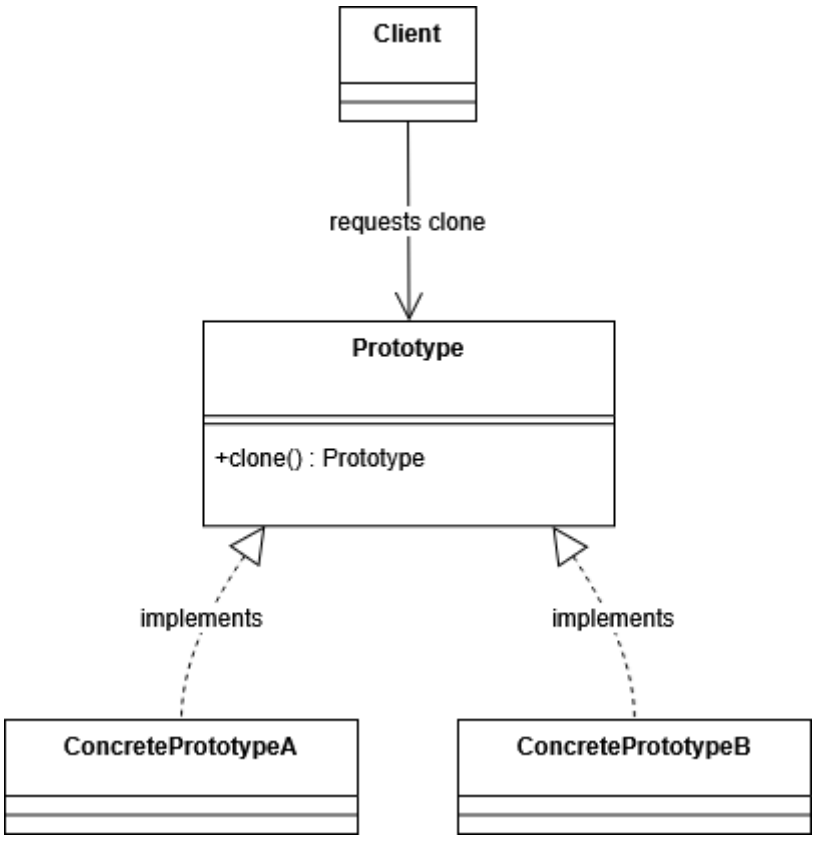
12. Розкажіть як працює шаблон «Команда».

1. Client створює ConcreteCommand, прив’язуючи до неї потрібний Receiver і параметри.
2. Invoker отримує команду і викликає її execute, не знаючи деталей бізнес-логіки.
3. ConcreteCommand усередині викликає відповідні методи Receiver для фактичної роботи.
4. За потреби Invoker/Command підтримують undo/redo, логування або черги виконання.

13. Яке призначення шаблону «Прототип»?

Шаблон «Прототип» створює нові об’єкти шляхом клонування прототипів, мінімізуючи залежність від конкретних класів і вартісну ініціалізацію.
Це корисно, коли створення «з нуля» дороге або потрібні численні подібні екземпляри з незначними відмінностями.

14. Нарисуйте структуру шаблону «Прототип».



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

- Prototype: інтерфейс/базовий клас із операцією клонування.

- ConcretePrototype: реалізує клонування (поверхневе або глибоке).
- Client: отримує нові об’єкти, викликаючи clone на прототипі замість new

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

- Формування контекстного меню у складних UI: кожен компонент додає свої пункти й передає запит батьківському елементу по ланцюжку.
- Служба підтримки з ескалацією: рівні L1→L2→L3 послідовно перевіряють, чи можуть обробити звернення.
- Логування за рівнями: INFO/WARN/ERROR пропускають повідомлення до відповідного обробника в ланцюжку.
- HTTP-middleware/фільтри безпеки: кожен проміжний обробник перевіряє/модифікує запит і, за потреби, передає далі.
- Банкомат-диспенсер: купюри різних номіналів видаються послідовними обробниками, що віднімають «свою» частину суми.
- Узгодження заявок на закупівлі: менеджери/директори затверджують або ескалюють далі по ієрархії