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 №5

з дисципліни "Технології розроблення програмного забезпечення"

Виконав

студент групи ІА–33:

Китченко Д.В.

Перевірив:

Мягкий М.Ю.

Київ 2025

Зміст

Вступ

Патерни проєктування є фундаментальним інструментом в арсеналі сучасного розробника програмного забезпечення. Вони являють собою перевірені часом, елегантні рішення для типових проблем, що виникають в процесі проєктування архітектури програмних систем. Використання патернів дозволяє створювати більш гнучкі, розширювані та легкі для супроводу додатки.

У даній лабораторній роботі детально розглядається структурний патерн «Adapter», який дозволяє організувати спільну роботу класів з несумісними інтерфейсами.

Мета роботи

Вивчити структуру та призначення шаблону «Adapter» та навчитися застосовувати його на практиці для інтеграції сторонніх або застарілих компонентів у існуючу архітектуру програмної системи без її модифікації.

5.2.1. Шаблон «Adapter»

Призначення патерну: Шаблон "Adapter" (Адаптер) використовується для адаптації інтерфейсу одного об'єкту до іншого [6]. Наприклад, існує декілька бібліотек для роботи з принтерами, проте кожна має різний інтерфейс (хоча однакові можливості і призначення). Має сенс розробити уніфікований інтерфейс (сканування, асинхронне сканування, двостороннє сканування, потокове сканування і тому подібне), і реалізувати відповідні адаптери для приведення бібліотек до уніфікованого інтерфейсу. Це дозволить в програмі звертатися до загального інтерфейсу, а не приводити різні сценарії роботи залежно від способу реалізації бібліотеки. Адаптери також називаються "wrappers" (обгортками).

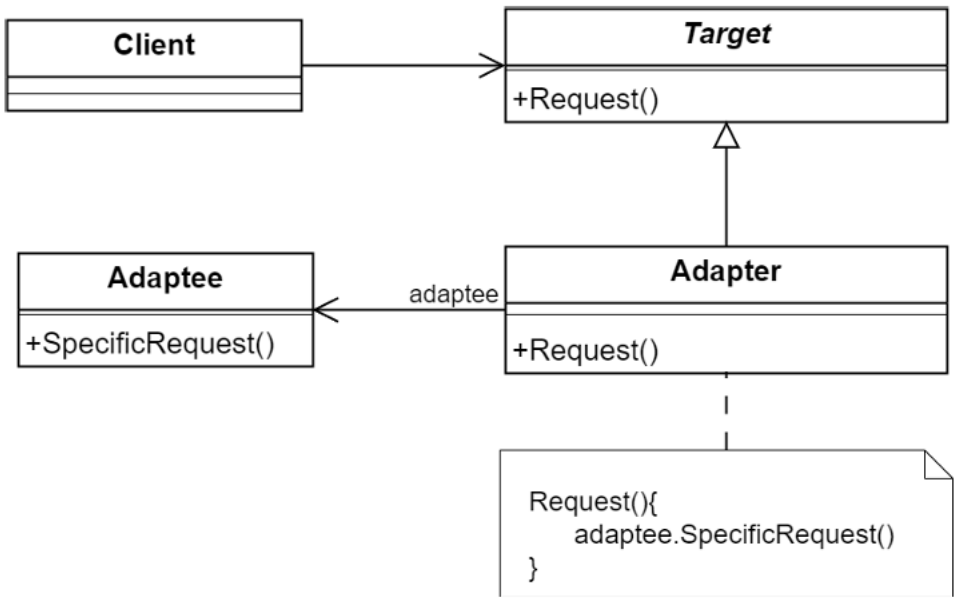


Рисунок 5.1. Структура патерну Адаптер на рівні об’єктів

Проблема: Ви реалізовуєте аудіо-плеєр, який може програвати аудіо різних форматів. Ви почали аналізувати і бачите що для програвання аудіо із різних форматів краще використовувати різні компоненти.

Таким чином ви реалізуєте складну логіку роботи з різними компонентами. У вас у коді з'являється багато логіки з перевіркою, якщо формат один, то викликаємо такий метод у такого компонента, якщо інший, то викликаємо декілька методів у іншого компонента, і т.д. Логіка роботи з компонентами стає досить складною та заплутаною.

Коли потрібно добавити підтримку нового аудіо-формату, то потрібно добавити роботу з новим компонентом і при цьому внести зміну в уже існуючу логіку, що може привести до помилок там де все коректно працювало.

Рішення: Для вирішення цих проблем можна використати патерн Адаптер. Визначаємо загальний інтерфейс IPlayer для програвання музики через компоненти. Далі для кожного компонента робимо свій адаптер. Адаптер має посилання на об'єкт компонента та реалізує інтерфейс викликаючи методи з компонента, який він адаптує. Таким чином, адаптер ніби обгортає специфічний компонент і далі весь клієнтський код буде працювати з адаптерами через інтерфейс IPlayer. Класи, які будуть працювати з адаптером через цей інтерфейс не будуть знати інтерфейси кожного специфічного компонента.

При такій реалізації, якщо буде потрібно добавити підтримку нового формату, то просто створюється новий адаптер, який обгортає новий компонент, але робота з цим адаптером буде виконуватися через існуючий інтерфейс. При цьому існуючий код не буде зазнавати змін, а значить і не "поламаємо" те що вже працює.

Переваги та недоліки:

+ Відокремлює інтерфейс або код перетворення даних від основної бізнес-логіки.

+ Можна добавляти нові адаптери не змінюючи код у класі Client.

- Умовним недоліком можна назвати збільшення кількості класів, але за рахунок використання патерна Адаптер програмний код, як привило, стає легше читати.

5.2.2. Шаблон «Builder»

Призначення патерну: Шаблон «Builder» (Будівельник) використовується для відділення процесу створення об'єкту від його представлення [6]. Це доречно у випадках, коли об'єкт має складний процес створення (наприклад, Web-сторінка як елемент повної відповіді web- сервера) або коли об'єкт повинен мати декілька різних форм створення (наприклад, при конвертації тексту з формату у формат).

Проблема: Візьмемо процес побудови відповіді на запит web-сервера. Побудова складається з наступних частин: додавання стандартних заголовків (дата/час, ім'я сервера, інш.), код статусу (після пошуку відповідної сторінки на сервері), заголовки відповіді (тип вмісту, інш.), утримуване, інше.

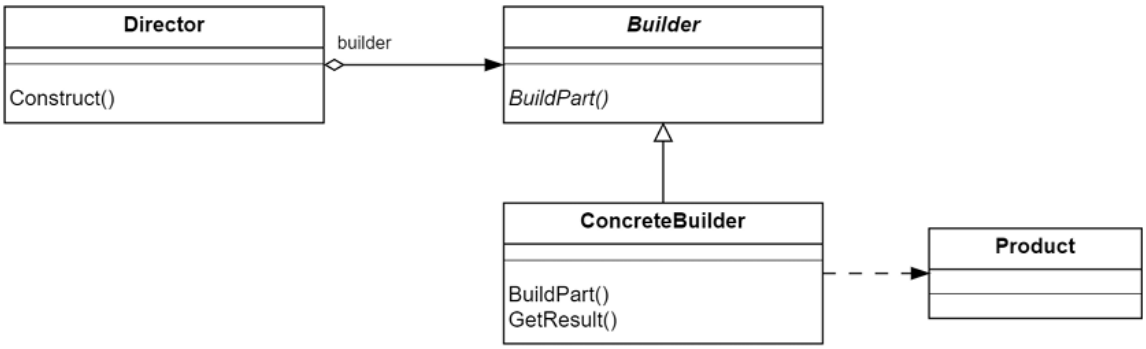


Рисунок 5.2. Структура патерну Builder

Рішення: Кожен з цих етапів може бути абстрагований в окремий метод будівельника. Це дасть наступні вигоди:

- Гнучкіший контроль над процесом створення сторінки;

- Незалежність від внутрішніх змін – наприклад, зміна назви сервера не сильно порушить процес побудови відповіді;

Переваги та недоліки:

+ Дозволяє використовувати один і той самий код для створення різноманітних продуктів.

- Клієнт буде прив’язаний до конкретних класів будівельників, тому що в інтерфейсі будівельника може не бути методу отримання результату.

5.2.3. Шаблон «Command»

Призначення патерну: Шаблон "command" (команда) перетворить звичайний виклик методу в клас [6]. Таким чином дії в системі стають повноправними об'єктами. Це зручно в наступних випадках:

- Коли потрібна розвинена система команд – відомо, що команди будуть добавлятися;
- Коли потрібна гнучка система команд – коли з'являється необхідність додавати командам можливість відміни, логування і інш.;
- Коли потрібна можливість складання ланцюжків команд або виклику команд в певний час.

Об'єкт команда сама по собі не виконує ніяких фактичних дій окрім перенаправлення запиту одержувачеві (тобто команди все ж виконуються одержувачем), однак ці об'єкти можуть зберігати дані для підтримки додаткових функцій відміни, логування і інш. Наприклад, команда вставки символу може запам'ятовувати символ, і при виклику відміни викликати відповідну функцію витирання символу. Можна також визначити параметр «застосовності» команди (наприклад, на картинці писати не можна) – і використати цей аттрибут для засвічування піктограми в меню.

Такий підхід до команд дозволяє побудувати дуже гнучку систему команд, що настроюється. У більшості додатків це буде зайвим (використовується спрощений варіант), проте життєво важливий в додатках з великою кількістю команд (редактори).

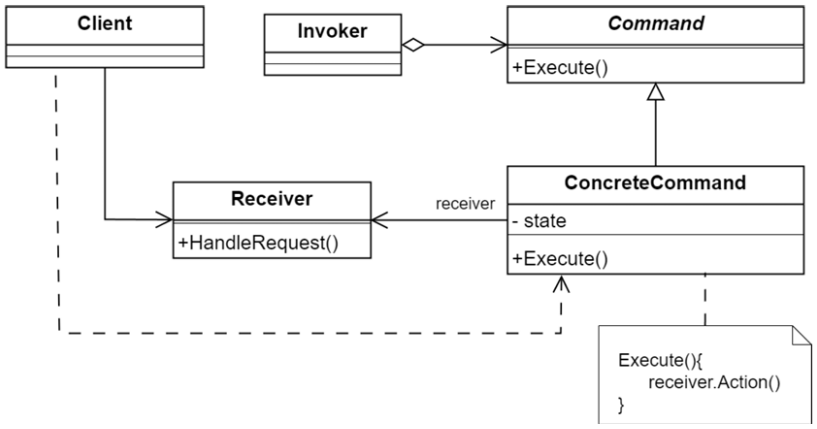


Рисунок 5.3. Структура патерну Команда

Проблема: Ви реалізуєте товстий клієнт який має багатий візуальний інтерфейс: має меню, кнопки і контекстне меню. Кожна дія, яку можна виконати, має три варіанти виконання – через меню, натисканням кнопки та через контекстне меню. Реалізацію кожної дії можна розмістити в обробнику

візуального елементу, але тоді потрібно буде продублювати цей функціонал для меню, кнопки та контекстного меню. Таким чином ми будемо мати дублювання коду і при зміні функціоналу змінювати його в трьох місцях.

Додатковим викликом буде реалізувати автоматизоване тестування такої системи, тому що нам необхідно емулювати натискання кнопок користувачем.

Рішення: Виділення функціоналу який виконується по натисканню на кнопку в окремий клас дозволяє відвязати візуальну частину від логіки обробки. Таким чином ми будемо мати шар з UI елементами і шар з логікою обробки дій виконаних на UI. Це дозволяє один і той самий об'єкт з шару логіки обробки дій використати для реакцію на натискання кнопки і пункту меню і контекстного меню. Кнопки тепер не знають про конкретні класи реалізації команд, а взаємодіють з об'єктами команд через загальний інтерфейс. Ще одна перевага, що тепер ми можемо достатньо просто реалізувати механізм enable-disable для кнопок та контекстного меню через виклик у об'єкта команди метода IsEnabled() або через прив'язку (binding) на поле IsEnabled у об'єкта команди.

При такій реалізації також набагато простіше організувати тестування застосунку, тому що ми тепер не потрібно емулювати дії користувача, а достатньо протестувати шар логіки обробки використовуючи модульні тести (unit tests).

Переваги та недоліки:

+ Ініціатор виконання команди не знає деталей реалізації виконавця команди.

+ Підтримує операції скасування та повторення команд.

+ Послідовність команд можна логувати і при необхідності виконати цю послідовність ще раз.

+ Простота розширення за рахунок додавання нових команд без необхідності внесення змін в уже існуючий код (принцип відкритості-закритості).

5.2.4. Шаблон «Chain of Responsibility»

Призначення патерну: Шаблон «Chain of responsibility» (Ланцюжок відповідальності) частково можна спостерігати в житті, коли підписання відповідного документу проходить від його складання у одного із співробітників компанії через менеджера і начальника до головного начальника, який ставить свій підпис [5].

Проблема: Ви розробляєте систему зі складними UI формами, які містять багато вкладених один в один візуальних компонентів, по кліку правою кнопкою миші вам потрібно сформувати контекстне меню. В залежності від того на якому компоненті був виконаний клік мишкою, вміст контекстного меню повинен відрізнятися, також в контекстне меню потрібно добавляти пункти, якщо компонент, в який входить поточний, теж має свої пункти. Один із очевидних підходів – ви робити метод, який викликається по кліку мишки. В методі ви робити перевірки і якщо в даний момент конкретний елемент, добавляєте в контекстне меню підходящі до нього пункти, потім перевіряєте інші елементи і перевіряєте чи вони не є такими, що містять в собі вибраний елемент. Якщо так, то добавляєте в контекстне меню ще пункти.

З часом алгоритм формування меню стає більш складним, тому що добавляються нові елементи. Також з часом в алгоритмі залишилися перевірки на компоненти, які з форми були вже видалені.

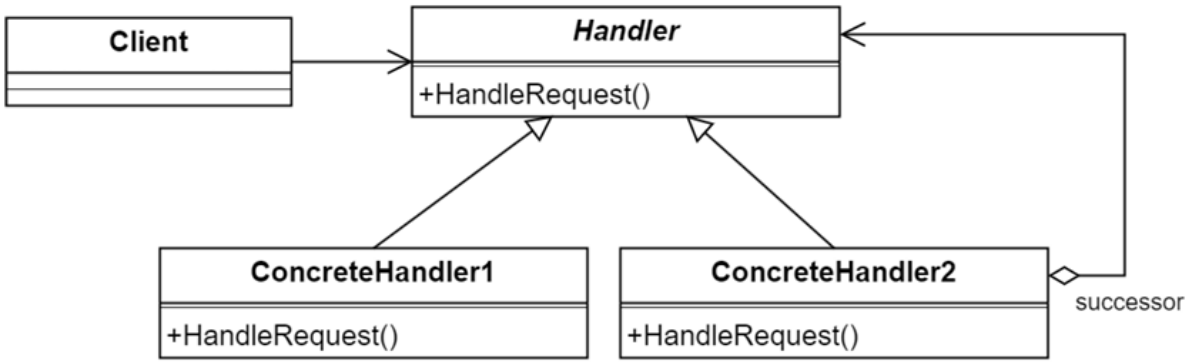


Рисунок 5.4. Структура патерна Ланцюжок відповідальності

Рішення: Для вирішення проблеми зі зростаючою складністю формування контестного меню підходить патерн «Ланцюжок відповідальності».

Основна ідея в тому, що нам не потрібно мати загальний метод формування контекстного меню. Ми можемо зробити загальний інтерфейс з методом UpdateContextMenu() і реалізувати цей метод в усіх візуальних компонентах. Додатково для візуальних компонентів добавляємо поле для переходу на "батьківський" елемент, якій в собі містить поточний візуальний компонент, а в реалізації UpdateContextMenu в кінці додаємо виклик цього метода у "батьківського" елемента. Якщо елемент не потребує додавання пунктів в контекстне меню, він просто викликає цей метод у наступного елемента в ланцюжку. Тепер обробка правого кліку мишки буде виглядати наступним чином: По кліку правою кнопкою миші викликаємо UpdateContextMenu на тексті, він, наприклад, додає пункт меню «Копіювати текст», далі викликає аналогічний метод у компонента «текст-блок» в якому цей текст знаходиться, але текст-блок нічого в контекстне меню не добавляє, а викликає UpdateContextMenu в елемента Grid, в якому знаходиться текст-блок, а Grid при відпрацюванні метода добавляє пункт меню «Контестна допомога» і так далі.

Коли ми змінюємо структуру форми, то між компонентами змінюються зв'язки в ланцюжку, але самі алгоритми вже переробляти не потрібно. якщо видаляється з форми якийсь компонент, то разом з ним видаляється і логіка пов'язана з цим компонентом. Таким чином складність роботи з контекстним

меню залишається контрольованою.

Слід добавити що елемент, що обробляє виклик, не обов'язково повинен передавати виклик далі по ланцюжку. Це залежить від того, яку поведінку ви хочете отримати.

Переваги та недоліки:

+ Зменшує залежність між клієнтом та обробниками: клієнт не знає хто обробить запит, а обробники не знають структуру ланцюжка.

+ Реалізовує додаткову гнучкість в обробці запиту: легко добавити або вилучити з ланцюжка нові обробники.

- Запит може залишитися ніким не опрацьованим: запит не має вказаного обробника, тому може бути не опрацьованим.

5.2.5. Шаблон «Prototype»

Призначення патерну: Шаблон «Prototype» (Прототип) використовується для створення об'єктів за «шаблоном» (чи «кресленням», «ескізом») шляхом копіювання шаблонного об'єкту, який називається прототипом. Для цього визначається метод «клонувати» в об'єктах цього класу [6].

Цей шаблон зручно використати, коли заздалегідь відомо як виглядатиме кінцевий об'єкт (мінімізується кількість змін до об'єкту шляхом створення шаблону), а також для видалення необхідності створення об'єкту – створення відбувається за рахунок клонування, і самій програмі немає необхідності знати, як створювати об'єкт.

Також, це дозволяє маніпулювати об'єктами під час виконання програми шляхом настроювання відповідних прототипів; значно зменшується ієрархія наслідування (оскільки в іншому випадку це були б не прототипи, а вкладені класи, що наслідують).

Проблема: Ви розробляєте редактор рівнів для 2D гри на основі спрайтів. В панелі інструментів ви маєте багато кнопок для різних елементів, які можна розташовувати на екрані, такі як сходи, стіни, підлога, оздоблення та інші. Ці елементи у вас об'єднані в ієрархію з базовим класом GameObject. Під кожен елемент можна зробити свій тип кнопки, але тоді ми отримаємо паралельну ієрархію кнопок і при додаванні нового типу ігрового об’єкту потрібно буде додавати і новий тип кнопки.

Рішення: Використовуючи патерн прототип, додаємо до базового об'єкта GameObject метод Clone(), а кнопки будуть зберігати посилання на об'єкт базового типу GameObject. При натисканні на кнопку об'єкт який потрібно добавити на ігровому полі отримуємо не створенням нового, а клонуванням прототипу, який прив'язаний до кнопки. Таким чином, коли ми будемо додавати нові типи ігрових об'єктів, то логіка роботи з кнопками не буде змінюватися, тому що не має прив'язки до конкретних типів.

Також слід відзначити, що при копіюванні об'єкту, навіть приватні поля будуть скопійовані, тому що реалізація методу копіювання знаходиться в цьому класі, що копіюється і таким чином є доступ для копіювання і до відкритих і до закритих полів.

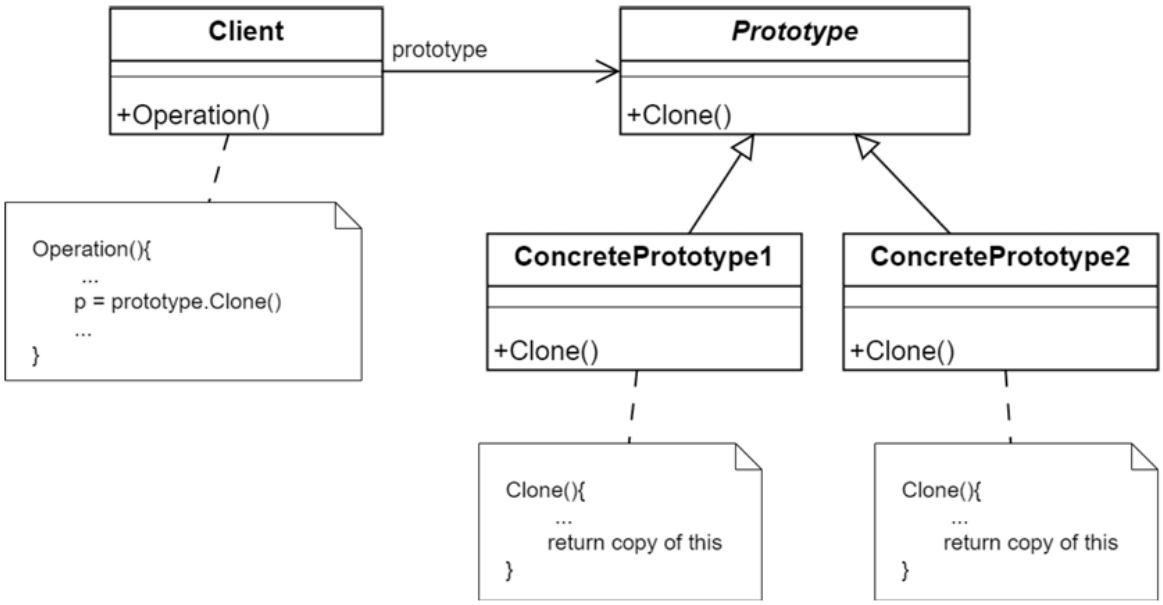


Рисунок 5.5. Структура патерну «Прототип»

Переваги та недоліки:

+ За рахунок клонування складних об’єктів замість їх створення, підвищується продуктивність.

+ Різні варіації об’єктів можна отримувати за рахунок клонування, а не розширення ієрархії класів.

+ Вища гнучкість, тому що клоновані об’єкти можна модифікувати незалежно, не впливаючи на об’єкт з якого була зроблена копія.

- Реалізація глибокого клонування досить проблематична, коли об’єкт що клонується містить складну внутрішню структуру та посилання на інші об’єкти.

- Надмірне використання патерну Прототип може привести до ускладнення коду та проблем із супроводом такого коду.

Хід роботи

В рамках розробки програми "Архіватор" виникла задача реалізувати підтримку декількох форматів архівів (RAR та SevenZip) з можливістю легкого додавання нових форматів у майбутньому. При цьому для роботи з цими форматами доступні тільки дві "старі" бібліотеки LegacyRarEngine та LegacySevenZipEngine, API яких є несумісним з сучасною архітектурою системи.

Пряма інтеграція цих legacy бібліотек у код системи призвела б до значного ускладнення архітектури та порушення принципів об'єктно-орієнтованого проєктування. Обидві legacy бібліотеки мають однакові "незручності" які роблять їх несумісними з існуючою системою:

- Повертає коди помилок (-1, 0) замість сучасних винятків.
- Надає доступ до файлів в архіві тільки за індексом, а не за іменем.
- Використовує застарілі формати даних (наприклад, long timestamp замість LocalDateTime).
- Використовує власну структуру даних RarRecord та SevenZipRecord з публічними полями.

Для вирішення цієї проблеми було застосовано патерн проєктування Adapter який дозволяє інтегрувати несумісні компоненти без зміни їхнього коду. Ключова ідея полягає в створенні проміжного класу адаптера який знає як розмовляти зі старими бібліотеками та перетворює їхній незручний API у зручний сучасний інтерфейс.

Особливістю реалізації є те що замість створення окремого адаптера для кожної legacy бібліотеки було створено один універсальний адаптер який може працювати з різними бібліотеками. Це як універсальна розетка яка приймає різні типи вилок європейські та американські та перетворює їх у єдиний стандартний вихід.

Архітектура складається з трьох основних компонентів:

LegacyRarEngine це перша стара бібліотека яка вміє працювати з RAR архівами. Вона має всі описані вище незручності але ми не можемо змінити її код бо це зовнішня бібліотека. Це Adaptee тобто компонент який потребує адаптації. LegacySevenZipEngine це друга стара бібліотека для роботи з SevenZip архівами.

UniversalArchiveAdapter це клас адаптер який є серцем всієї системи. При створенні адаптера йому передається формат з яким треба працювати RAR чи SevenZip. Залежно від формату адаптер автоматично обирає потрібну legacy бібліотеку всередині. Після цього адаптер надає зручний API для роботи з архівами незалежно від того яка legacy бібліотека працює всередині. Адаптер виконує всі необхідні перетворення коди помилок у винятки доступ за індексами у списки Unix timestamp у LocalDateTime публічні поля у інкапсульовані об'єкти.

Така архітектура дозволила інтегрувати дві несумісні legacy бібліотеки в систему зберігши чистоту коду та не змінюючи самі бібліотеки. Клієнтський код тепер працює з адаптером через зручний API і не знає про існування складних legacy бібліотек всередині.

Діаграма класів реалізованої системи

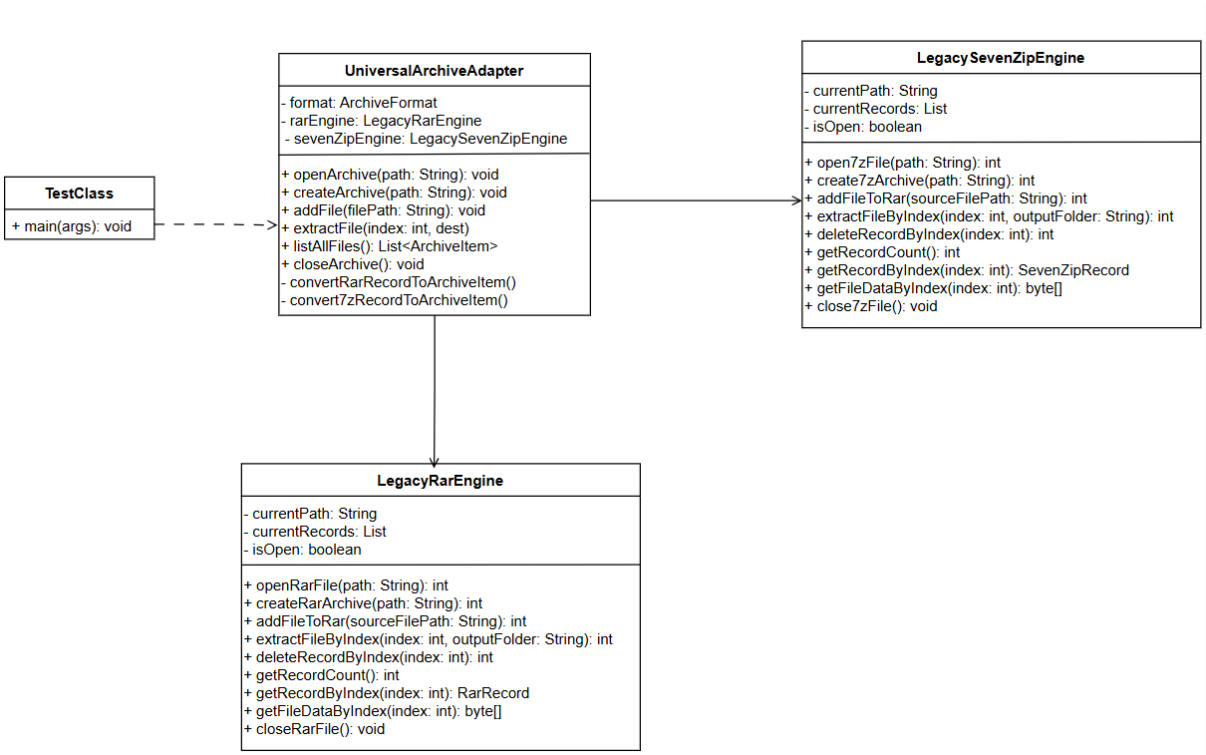


Рисунок 1. Діаграма класів для реалізації патерну Adapter

Фрагменти коду реалізації

1. "Незручний" API класу LegacyRarEngine. Цей клас навмисно спроєктований зі застарілими підходами: повертає коди помилок та надає доступ до файлів за індексом.
2. UniversalArchiveAdapter адаптує обидві legacy бібліотеки. Клас адаптер є серцем реалізації патерну. Він приймає формат при створенні та автоматично обирає потрібну legacy бібліотеку всередині.
3. Код клієнта виглядає чистим та зрозумілим. Не треба перевіряти коди помилок після кожного виклику. Не треба писати цикли для отримання списку файлів. Не треба розбиратись з Unix timestamp. Не треба знати про існування legacy бібліотек. Адаптер приховує всю складність і надає простий інтерфейс. При цьому той самий клієнтський код працює з різними форматами бо адаптер універсальний.

Тестування реалізації

Для перевірки коректності роботи патерну Adapter було створено тестовий клас Test який демонструє роботу універсального адаптера з двома різними legacy бібліотеками

Factory вибирає потрібну Strategy залежно від формату

1. Запит на RAR формат:

Factory обрав: RarStrategy

2. Запит на 7Z формат:

Factory обрав: SevenZipStrategy

ADAPTER ПАТЕРН

UniversalAdapter адаптує РІЗНІ legacy бібліотеки

3. RarStrategy створює UniversalAdapter для RAR:

RarStrategy: створення RAR архіву demo.rar

обрано RAR

RAR архів створено: demo.rar

4. SevenZipStrategy створює UniversalAdapter для 7Z:

SevenZipStrategy: створення 7Z архіву

обрано 7Z

7Z архів створено: demo.7z

ПЕРЕВІРКА

5. Відкриття RAR архіву:

RarStrategy: відкриття RAR архіву

обрано RAR

Відкрито RAR

6. Витягування з RAR:

RarStrategy: витягування з RAR

обрано RAR

Витягнуто з RAR

7. Відкриття 7Z архіву:

SevenZipStrategy: відкриття 7Z архіву

обрано 7Z

Відкрито 7Z

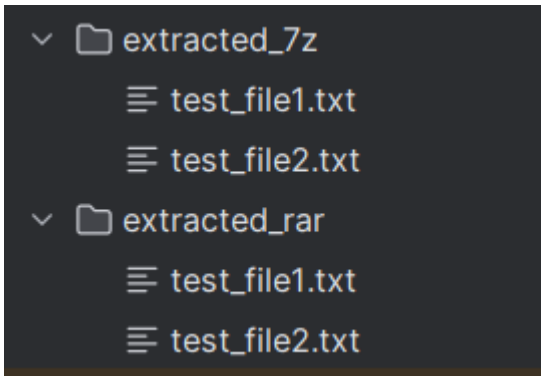
8. Витягування з 7Z:

SevenZipStrategy: витягування з 7Z

обрано 7Z

Витягнуто з 7Z

Process finished with exit code 0



Результати виконання тесту:

Універсальний адаптер успішно працює з двома різними legacy бібліотеками LegacyRarEngine та LegacySevenZipEngine надаючи однаковий зручний інтерфейс для обох.

Критично важливим результатом є те що клієнтський код однаковий для обох форматів. Методи createArchive addFile openArchive listAllFiles extractFile викликаються однаково незалежно від того чи це RAR чи SevenZip архів. Адаптер автоматично обирає потрібну legacy бібліотеку залежно від формату переданого у конструкторі і клієнт не знає про цю деталь реалізації.

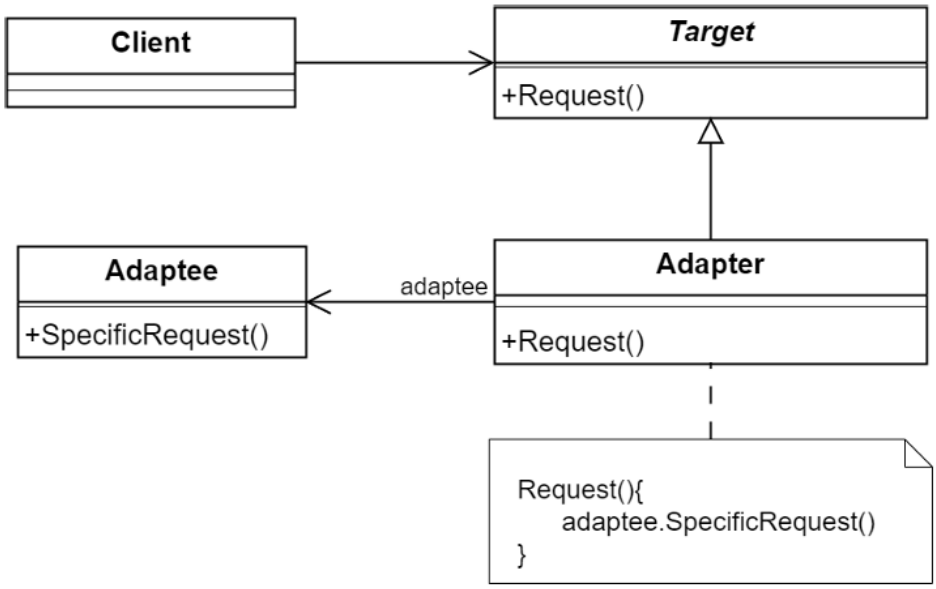
Успішне витягування файлів та їх наявність у папках extracted_rar та extracted_7z підтверджує що адаптер не тільки надає зручний інтерфейс але й коректно виконує всі операції. Файли створюються стискаються зберігаються у архів та витягуються назад без втрат даних.

Питання до лабораторної роботи

1. Яке призначення шаблону «Адаптер»?

   Призначення шаблону «Адаптер» - забезпечити спільну роботу класів з несумісними інтерфейсами. Він діє як "перехідник" або "обгортка" навколо існуючого класу (Adaptee), перетворюючи його інтерфейс на інший, очікуваний клієнтським кодом (Client).

2. Нарисуйте структуру шаблону «Адаптер».



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

   Client (Клієнт): Клас, який хоче використовувати функціонал, але очікує певний інтерфейс (Target). У нашій роботі це RarStrategy.

   Target (Цільовий інтерфейс): Інтерфейс, який використовує Client.

   Adapter (Адаптер): Клас, який реалізує інтерфейс Target і містить посилання на об'єкт Adaptee. Він перенаправляє виклики від Client до Adaptee, виконуючи необхідні перетворення. У нашій роботі це RarAdapter.

   Adaptee (Об'єкт, що адаптується): Існуючий клас з несумісним інтерфейсом. У нашій роботі це LegacyRarEngine.

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

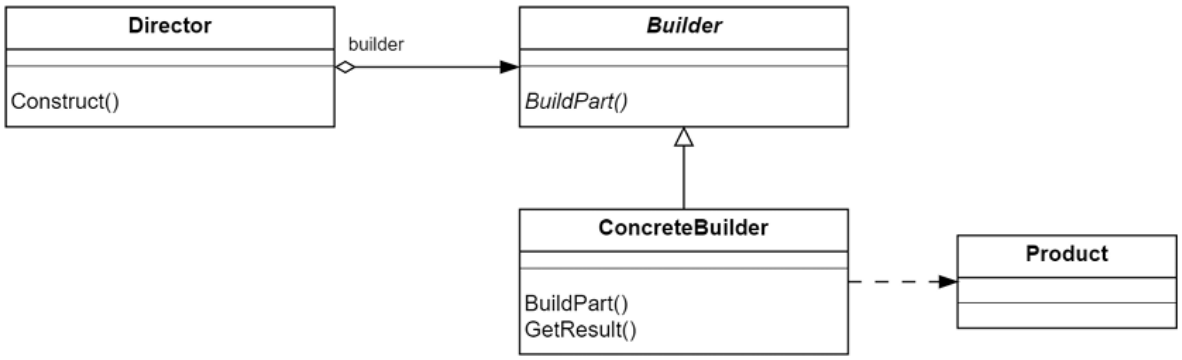
Адаптер об'єктів (використано в роботі): Використовує композицію. Клас Adapter містить екземпляр класу Adaptee. Цей підхід є більш гнучким, оскільки дозволяє адаптувати будь-який підклас Adaptee.

Адаптер класів: Використовує множинне успадкування (в Java реалізується через успадкування класу та реалізацію інтерфейсу). Клас Adapter одночасно успадковує Adaptee та реалізує інтерфейс Target. Цей підхід менш гнучкий.

5. Яке призначення шаблону «Будівельник»?

Шаблон «Будівельник» (Builder) використовується для покрокового створення складних об'єктів. Він дозволяє відокремити процес конструювання об'єкта від його представлення, завдяки чому один і той самий процес конструювання може створювати різні представлення.

6. Нарисуйте структуру шаблону «Будівельник».



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

Product (Продукт): Складний об'єкт, який створюється.

Builder (Будівельник): Абстрактний інтерфейс для створення частин об'єкта Product.

ConcreteBuilder (Конкретний будівельник): Реалізує інтерфейс Builder і конструює конкретне представлення продукту.

Director (Директор): Клас, який керує процесом побудови, використовуючи об'єкт Builder.

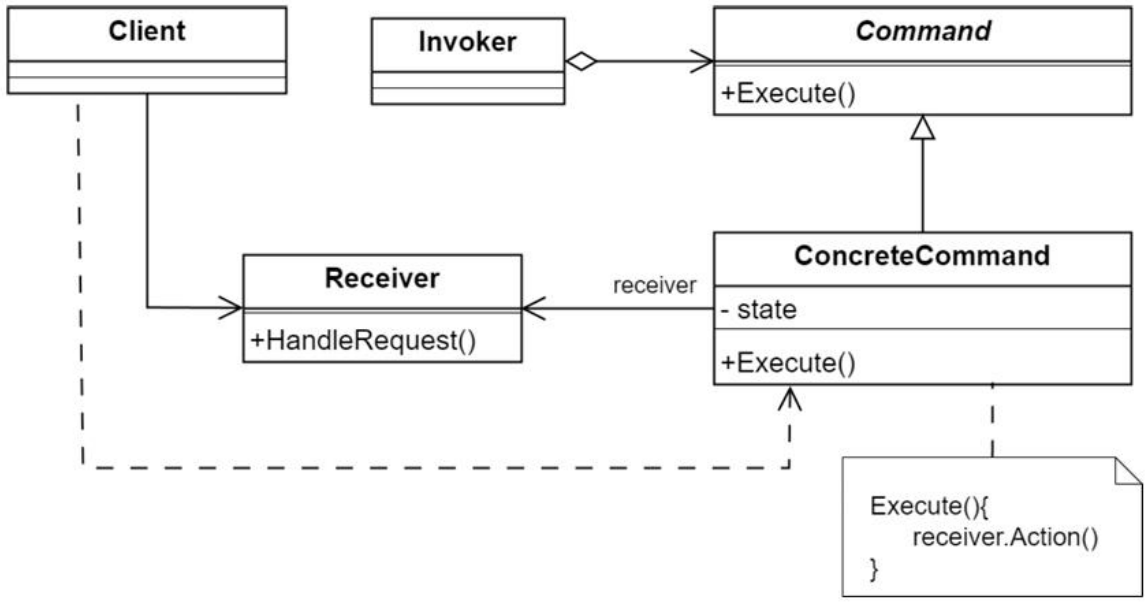
8. У яких випадках варто застосовувати шаблон «Будівельник»"?

Коли процес створення об'єкта є складним, багатоетапним, або коли конструктор має занадто багато параметрів (особливо опціональних). Також, коли потрібно створювати різні представлення одного й того ж об'єкта.

9. Яке призначення шаблону «Команда»?

Шаблон «Команда» (Command) інкапсулює запит на виконання дії як об'єкт. Це дозволяє параметризувати клієнтські об'єкти різними запитами, ставити запити в чергу, логувати їх, а також підтримувати операції скасування (undo).

10. Нарисуйте структуру шаблону «Команда».



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

Command: Інтерфейс, що оголошує метод для виконання операції (execute).

ConcreteCommand: Реалізує інтерфейс Command, містить посилання на Receiver і викликає його методи.

Client: Створює об'єкт ConcreteCommand і встановлює його одержувача.

Invoker: Просить команду виконати запит.

Receiver: "Одержувач", який знає, як виконати операцію.

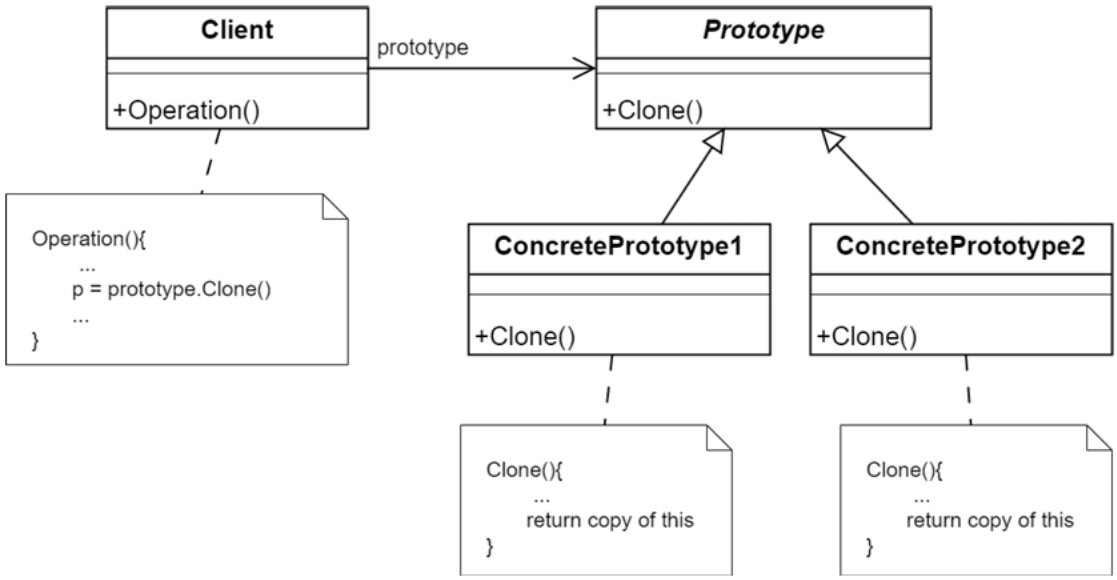
12. Розкажіть як працює шаблон «Команда».

Клієнт створює об'єкт-команду, пов'язуючи її з конкретним одержувачем. Потім цей об'єкт-команда передається ініціатору (Invoker), наприклад, кнопці в меню. Коли користувач натискає кнопку, ініціатор викликає метод execute() у команди, а команда, в свою чергу, викликає потрібний метод у свого одержувача.

13. Яке призначення шаблону «Прототип»?

Шаблон «Прототип» (Prototype) дозволяє створювати нові об'єкти шляхом копіювання існуючого об'єкта (прототипу). Це дозволяє уникнути прив'язки до класів об'єктів, що створюються.

14. Нарисуйте структуру шаблону «Прототип».



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

Prototype: Інтерфейс, що оголошує метод клонування (clone).

ConcretePrototype: Реалізує інтерфейс Prototype та метод clone.

Client: Створює новий об'єкт, викликаючи метод clone у прототипу.

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

Обробка подій в GUI: Коли користувач клікає на кнопку, подія спочатку обробляється кнопкою, потім може бути передана батьківській панелі, потім вікну, і так далі, доки не буде оброблена.

Системи логування: Повідомлення може проходити через ланцюжок обробників: один записує в консоль, інший - у файл, третій - відправляє по email, залежно від рівня важливості.

Системи авторизації та валідації: Запит користувача може проходити через ланцюжок перевірок: перевірка автентифікації, перевірка прав доступу, перевірка валідності даних.

Висновки

В ході виконання даної лабораторної роботи було успішно реалізовано патерн проєктування Adapter для інтеграції двох legacy бібліотек у сучасну систему архівації.

Ключовим досягненням роботи є створення універсального адаптера який може працювати з різними legacy бібліотеками замість окремого адаптера для кожної. Такий підхід значно зменшив дублювання коду та спростив підтримку системи. UniversalArchiveAdapter приймає формат при створенні та автоматично обирає потрібну legacy бібліотеку всередині що робить його схожим на універсальну розетку яка приймає різні типи вилок.