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»

Тема: «Патерни проектування»

Виконала:
студентка групи ІА-33
Самойленко Анастасія

Перевірив:
асистент кафедри ІСТ
Мягкий Михайло Юрійович

Київ 2025

Тема: Патерни проектування.
Мета: Вивчити структуру шаблонів «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» та навчитися застосовувати їх в реалізації програмної системи.
Тема роботи:
27. Особиста бухгалтерія (state, prototype, decorator, bridge, flyweight, SOA)
Програма повинна бути наочним засобом для ведення особистих фінансів: витрат і прибутку; з можливістю встановлення періодичних витрат / прибутку (зарплата і орендна плата); введення сканованих чеків з відповідними статтями витрат; побудова статистики; експорт/імпорт в Excel, реляційні джерела даних; різні рахунки; ведення єдиного фонду на всі рахунки (всією сім'єю) – на особливі потреби (ремонт, автомобіль, відпустка); можливість введення вкладів / кредитів для контролю банківських рахунків (звірка нарахованих відсотків з необхідними і т.д.).

Вступ

Під час розробки складних програмних систем важливо забезпечити зрозумілу, гнучку та легко підтримувану архітектуру. Для цього використовуються шаблони проєктування — узагальнені рішення типових завдань, що виникають у процесі програмування. Вони допомагають організувати взаємодію між об’єктами, зменшити дублювання коду та спростити внесення змін у програму.

У цій лабораторній роботі розглядаються такі структурні та поведінкові шаблони, як Adapter, Builder, Command, Chain of Responsibility та Prototype. Кожен із них має своє призначення: адаптер узгоджує несумісні інтерфейси, будівельник спрощує створення складних об’єктів, команда інкапсулює запит у вигляді об’єкта, ланцюг відповідальностей дозволяє передавати запит послідовно між обробниками, а прототип забезпечує копіювання об’єктів без прив’язки до їх конкретних класів.

Метою роботи є вивчення структури зазначених шаблонів та набуття практичних навичок їх використання під час реалізації програмних систем. Результатом стане розуміння способів застосування цих шаблонів для побудови більш ефективного, масштабованого та зручного у підтримці програмного забезпечення.

Теоретичні відомості

Шаблон (патерн) проєктування — це формалізований опис типового рішення, яке багаторазово зустрічається під час розроблення інформаційних систем. Він містить узагальнений спосіб вирішення певної задачі проєктування, рекомендації щодо його застосування та має загальновживану назву.

Головна мета використання шаблонів — повторне застосування вже знайдених ефективних рішень у нових ситуаціях. Важливою умовою є коректне моделювання предметної області, що дозволяє правильно визначити задачу та обрати відповідний патерн.

Застосування шаблонів проєктування забезпечує низку переваг:

- створення структурованої та логічної моделі системи;
- підвищення наочності та спрощення вивчення архітектури;
- глибше опрацювання структури програмного продукту;
- підвищення стійкості системи до змін вимог;
- спрощення подальшого доопрацювання та розширення;
- полегшення інтеграції систем і взаєморозуміння між розробниками завдяки спільному “словнику проєктування”.

Шаблон «Adapter»

Патерн *Adapter* дозволяє узгодити несумісні інтерфейси, щоб об’єкти з різними структурами могли взаємодіяти між собою. Він діє як «обгортка» (wrapper), яка перетворює інтерфейс одного класу у вигляд, сумісний з іншим.

Проблема:

У програмі потрібно працювати з різними компонентами, що мають схожу функціональність, але різні інтерфейси. Наприклад, аудіо-плеєр, який повинен підтримувати різні формати файлів, але кожен формат має свій спосіб взаємодії. Це ускладнює код і робить його важким для розширення.

Рішення:

Визначається єдиний інтерфейс (наприклад, IPlayer), через який клієнтський код взаємодіє з усіма компонентами. Для кожного конкретного формату або бібліотеки створюється власний адаптер, який реалізує цей інтерфейс і викликає

методи відповідного компонента. Таким чином, при додаванні нового формату достатньо створити новий адаптер без змін у вже працюючому коді.

Переваги:

- Спрощує інтеграцію сторонніх або несумісних класів.
- Дозволяє розширювати систему без зміни існуючої логіки.
- Підвищує зрозумілість і структурованість коду.

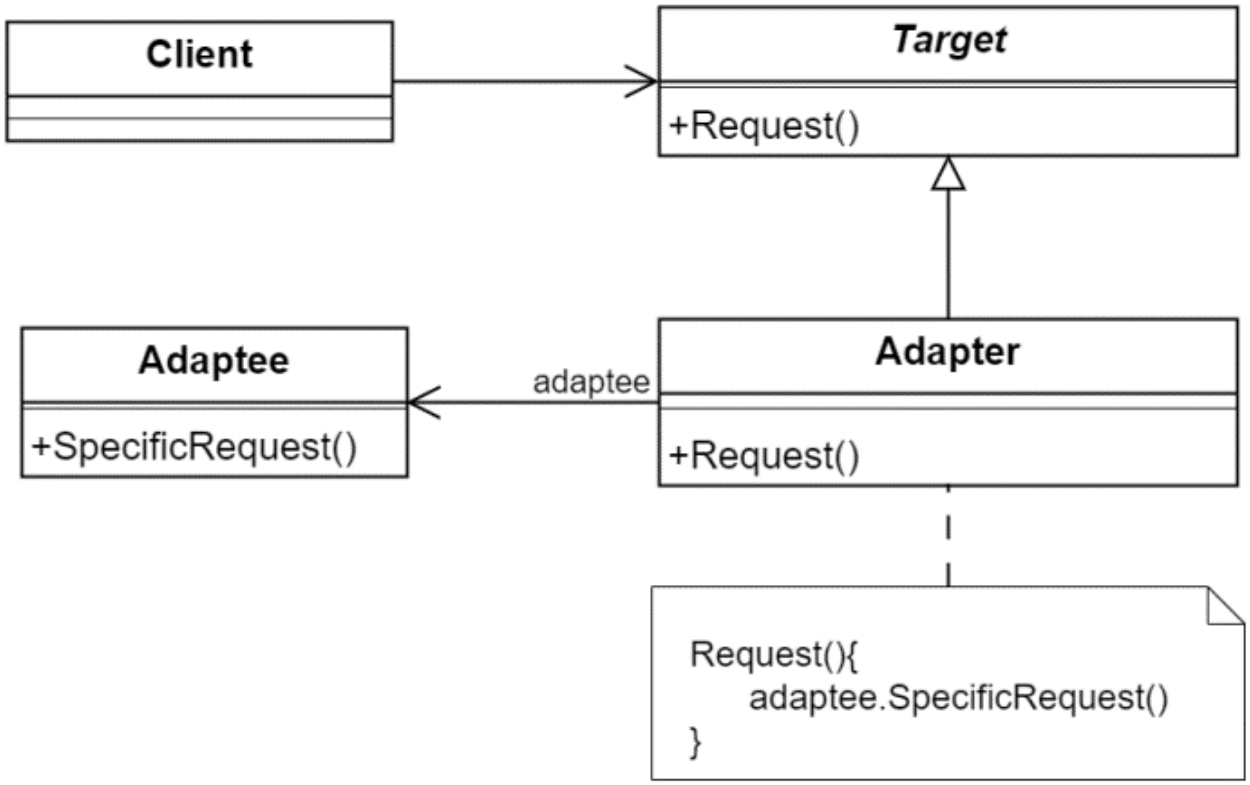
Недоліки:

- Збільшує кількість класів у проєкті.
- Може ускладнити відстеження зв’язків між адаптерами та оригінальними класами.

Основна ідея:

Adapter перетворює інтерфейс одного класу в інтерфейс, очікуваний клієнтом, забезпечуючи сумісність без зміни існуючого коду.

Структура патерну Адаптер на рівні об’єктів:



Шаблон «Builder»

Патерн *Builder* використовується для розділення процесу створення складного об’єкта та його представлення. Завдяки цьому один і той самий процес побудови може створювати різні варіації об’єкта.

Проблема:

Якщо об’єкт має багато складових частин або параметрів (наприклад, відповідь web-сервера, що містить заголовки, статус, тіло тощо), то його створення може стати громіздким і незручним. У коді з’являється багато кроків ініціалізації, що ускладнює підтримку й зміну структури об’єкта.

Рішення:

Процес створення об’єкта розділяється на окремі кроки — методи будівельника. Для кожної частини створюється свій метод, що дозволяє послідовно формувати кінцевий результат. Керуючий клас (*Director*) визначає порядок виклику методів, а сам *Builder* відповідає за створення і налаштування об’єкта. Це забезпечує гнучкість, зручність розширення і незалежність від змін у внутрішній структурі продукту.

Переваги:

- Дозволяє створювати різні варіації складних об’єктів за допомогою одного і того ж процесу побудови.
- Спрощує контроль над етапами створення об’єкта.
- Забезпечує кращу роздільність коду та зручність модифікації.

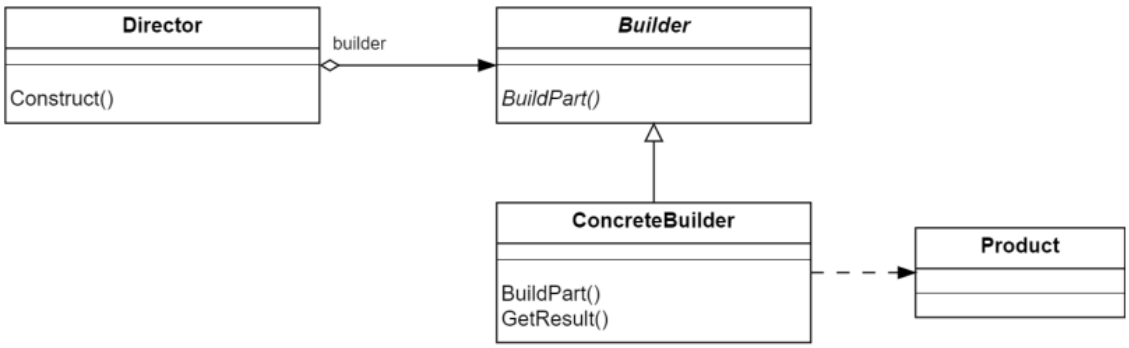
Недоліки:

- Може ускладнити архітектуру за рахунок додаткових класів.
- Клієнт може бути прив’язаний до конкретних реалізацій будівельників.

Основна ідея:

Builder ізолює логіку створення складного об’єкта від його структури, дозволяючи легко змінювати як процес побудови, так і кінцеве представлення без втручання у внутрішній код.

Структура патерну Builder:



Шаблон «Command»

Патерн *Command* перетворює виклик операції в окремий об’єкт, що інкапсулює дію та її параметри. Це дозволяє поводитися з командами як з об’єктами — передавати, зберігати, скасовувати або повторювати їх виконання.

Проблема:

У програмах із розвиненим графічним інтерфейсом (меню, кнопки, контекстне меню) одна й та сама дія може викликатися різними способами. Якщо логіка кожної дії дублюється в різних обробниках подій, це ускладнює підтримку коду та унеможливлює централізоване тестування.

Рішення:

Функціонал кожної дії виділяється в окремий клас-команду, який реалізує спільний інтерфейс (наприклад, ICommand). Елементи інтерфейсу (кнопки, меню тощо) не містять логіки виконання — вони лише викликають відповідну команду. Такий підхід відокремлює графічний інтерфейс від бізнес-логіки, спрощує тестування, додає можливість реалізації відміни (undo), повтору (redo), логування або об’єднання команд у ланцюжки.

Переваги:

- Повна незалежність між ініціатором дії та її виконавцем.
- Підтримка скасування, повторного виконання та історії команд.
- Просте розширення системи шляхом додавання нових команд без змін у наявному коді.
- Можливість об’єднувати кілька команд у складні послідовності.

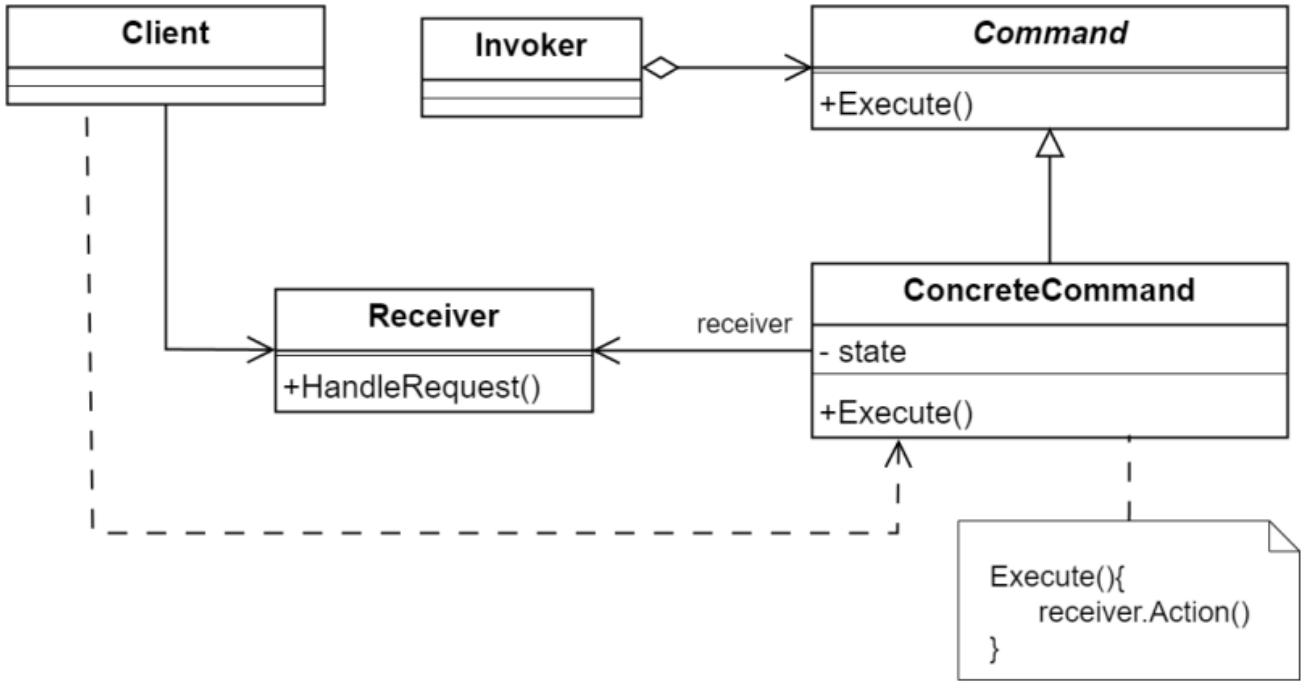
Недоліки:

- Збільшення кількості класів у системі.
- Необхідність додаткового управління станами команд.

Основна ідея:

Command інкапсулює запит у вигляді об’єкта, відокремлюючи відправника від виконавця, що забезпечує гнучкість, розширюваність і можливість керування командами як даними.

Структура патерну Команда:



Шаблон «Chain of Responsibility»

Патерн *Chain of Responsibility* дозволяє передавати запит послідовно через ланцюжок обробників, поки один із них не опрацює його. Такий підхід усуває жорстку залежність між відправником запиту і його отримувачами, забезпечуючи гнучкість і масштабованість системи.

Проблема:

У складних інтерфейсах із багатьма вкладеними компонентами (наприклад, у формі з текстовими блоками, панелями, сітками тощо) потрібно формувати контекстне меню залежно від того, де саме користувач виконав клік. Якщо вся логіка перевірок і додавання пунктів меню зосереджена в одному методі, код стає заплутаним, важким для підтримки та розширення.

Рішення:

Кожен компонент інтерфейсу реалізує спільний метод, наприклад UpdateContextMenu(), який додає власні пункти меню і передає запит далі "вгору" по ієрархії до свого батьківського елемента. Якщо компонент не потребує змін у меню, він просто делегує виклик далі по ланцюгу. Таким чином формується послідовність обробників, де кожен елемент сам вирішує, чи опрацьовувати запит, чи передати його наступному.

Переваги:

- Зменшує залежність між джерелом запиту та його обробниками.

- Легко додавати, змінювати або вилучати обробників без змін у клієнтському коді.
- Підвищує гнучкість і контрольованість обробки запитів.

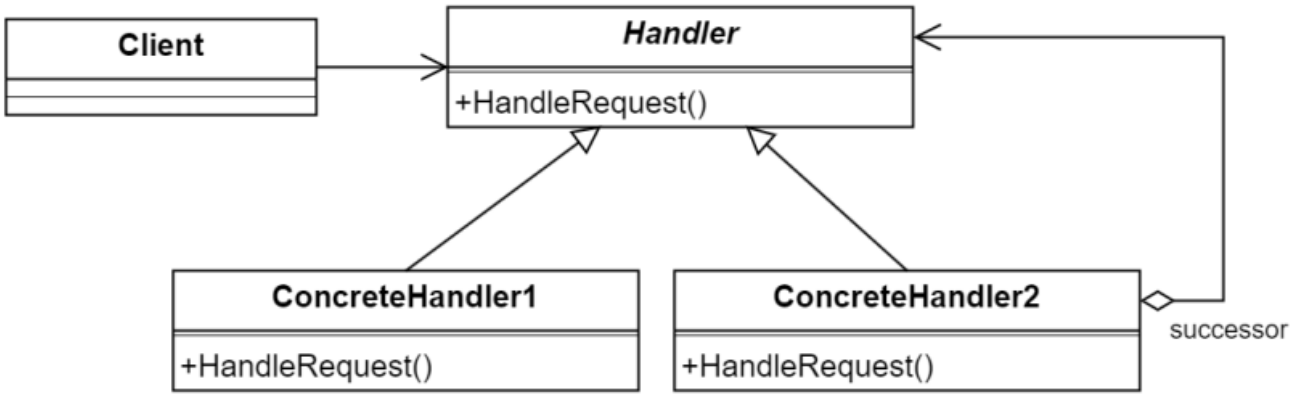
Недоліки:

- Існує ризик, що запит залишиться необробленим, якщо жоден елемент у ланцюгу його не перехопить.
- Відстеження потоку запиту може бути складним при великій кількості обробників.

Основна ідея:

Chain of Responsibility забезпечує передачу запиту вздовж послідовності потенційних обробників, дозволяючи кожному з них вирішувати, чи обробляти запит самостійно, чи передати його далі, що зменшує зв'язність системи та спрощує її розширення.

Структура патерна Ланцюжок відповідальності:



Шаблон «Prototype»

Патерн «Прототип» (Prototype) використовується для створення нових об'єктів шляхом копіювання вже існуючого зразка (прототипу). У межах класу визначається метод clone(), який дозволяє створювати нові екземпляри, не використовуючи конструктор безпосередньо.

Цей підхід особливо зручний, коли структура кінцевого об'єкта відома заздалегідь, і необхідно уникнути багаторазового створення схожих об'єктів. Замість того, щоб ініціалізувати об'єкт з нуля, система копіює вже готовий шаблон, що спрощує процес і підвищує ефективність. Таким чином, зменшується залежність програми від деталей створення об'єктів, адже сам процес ініціалізації прихований усередині методу клонування.

Використання прототипів дозволяє гнучко змінювати поведінку системи під час її виконання, настроюючи вихідні екземпляри-прототипи. Окрім цього, застосування даного патерну допомагає скоротити ієрархію класів, адже відпадає потреба створювати численні підкласи для кожної можливої конфігурації об’єкта.

Проблема:

Під час розроблення 2D-редактора рівнів для гри, що використовує спрайти, у панелі інструментів передбачено безліч елементів — стіни, підлоги, сходи, декорації тощо. Всі ці елементи походять від базового класу GameObject. Якщо для кожного типу створювати окрему кнопку, виникає дублювання структури класів: для кожного ігрового об’єкта потрібно розробляти відповідну кнопку.

Рішення:

Застосування патерну «Прототип» дозволяє уникнути цього. Базовий клас GameObject реалізує метод Clone(), а кнопки в панелі інструментів зберігають посилання на прототипи відповідних об’єктів. Коли користувач натискає кнопку, на сцену додається не новий об’єкт, а копія прототипу. Це означає, що при розширенні набору ігрових елементів логіка кнопок не змінюється, бо вона не залежить від конкретних класів.

Важливо, що під час копіювання об’єкта клонуються всі поля, у тому числі приватні, адже реалізація методу клонування знаходиться всередині самого класу, який має до них доступ.

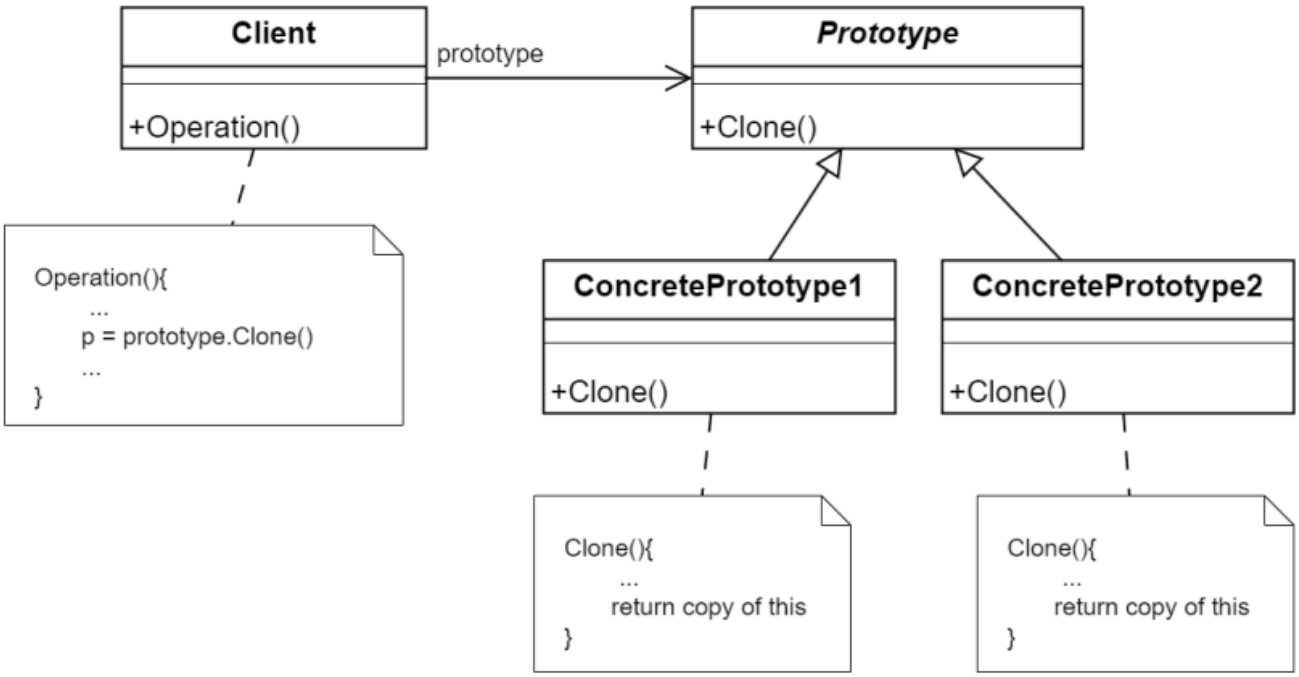
Переваги:

- Підвищення продуктивності за рахунок клонування складних об’єктів замість їх створення з нуля.
- Можливість отримувати різні варіації об’єктів без розширення ієрархії класів.
- Гнучкість у зміні й налаштуванні об’єктів під час виконання програми.

Недоліки:

- Реалізація глибокого клонування є складною, особливо якщо об’єкт містить посилання на інші об’єкти.
- Надмірне використання патерну може призвести до ускладнення структури коду та проблем із його підтримкою.

Структура патерну «Прототип»:



Хід роботи

1) Діаграма класів частини функціоналу робочої програми. (рис. 1)

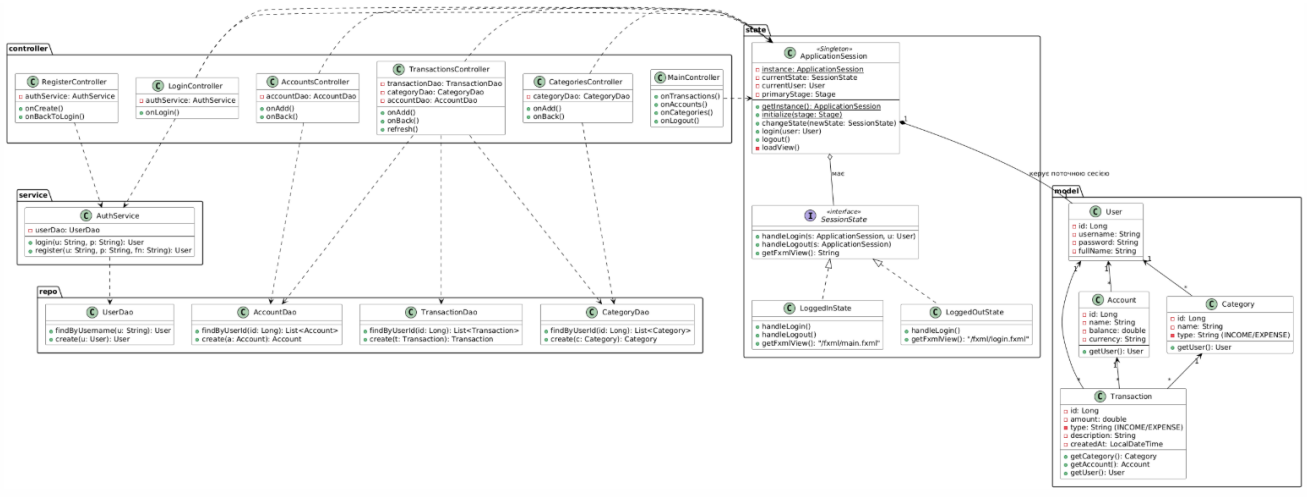


Рисунок 1 – Діаграма класів

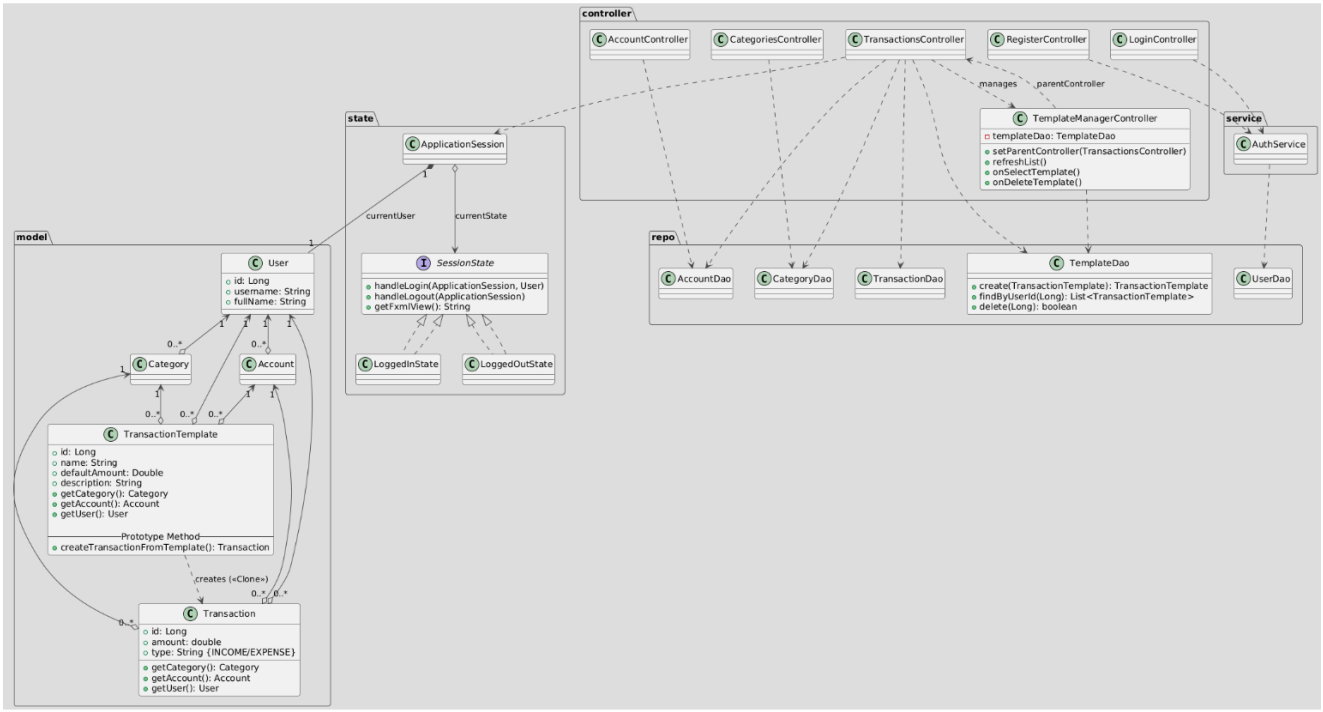


Рисунок 2 – Діаграма класів з патерном Prototype

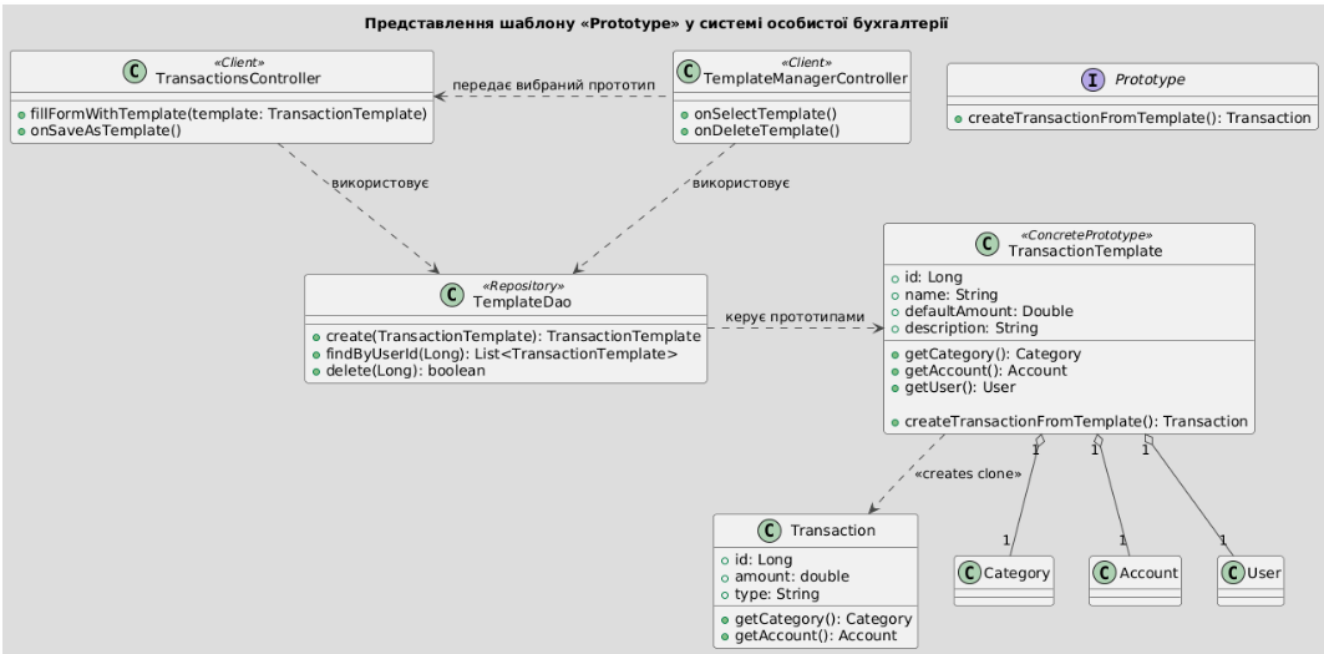


Рисунок 3 – Представлення самого шаблону на діаграмі

2) Реалізація патерну «Prototype» для формування об’єктів транзакцій за зразком (шаблоном)

Шаблон «Prototype» (Прототип) реалізовано для швидкого створення нових об'єктів Transaction (Транзакція) на основі існуючих, попередньо збережених конфігурацій (шаблонів).

Головна мета – уникнути повторної ініціалізації всіх полів (типу, рахунку, категорії, опису) та складного доступу до бази даних при кожному створенні нової транзакції. Замість цього використовується клонування існуючого, повністю ініціалізованого об'єкта.

Шаблон реалізовано через три основні елементи:

1. Prototype Interface — імпліцитно в TransactionTemplate.java

Хоча в Java не використовується явний інтерфейс Prototype (зазвичай використовують вбудований інтерфейс Cloneable), у нашому проєкті цю роль виконує метод, визначений у класі TransactionTemplate.

Призначення: Визначити механізм, за допомогою якого об'єкт-шаблон може створити свою робочу копію (Transaction).

Метод клонування: Transaction createTransactionFromTemplate().

- Цей метод виконує поверхневе копіювання незмінних довідників (Category, Account, User) та копіювання значень примітивних полів (amount, type, description) у новий об'єкт Transaction.
- Результат: Повертає об'єкт Transaction, готовий до редагування, який є ідентичним прототипу (крім ID).

2. Конкретний Прототип (TransactionTemplate.java та TemplateDao.java)

Клас TransactionTemplate є Конкретним Прототипом, а його екземпляри, які зберігаються у БД, є нашими готовими шаблонами.

Клас TransactionTemplate: Зберігає повний набір атрибутів, необхідних для створення транзакції (назву, суму за замовчуванням, посилання на Category, Account, User, тип та опис). Це є Сховище Прототипів (Prototype Repository) на рівні моделі.

TemplateDao.java: Це компонент, який керує Сховищем Прототипів у БД.

- create(): Зберігає новий прототип (клонований з поточної форми) у таблицю transaction_templates.
- findByUserId(): Завантажує всі прототипи для користувача, при цьому він виконує JOIN із таблицями categories та accounts, гарантуючи, що

кожен об'єкт TransactionTemplate є повністю ініціалізованим прототипом (тобто, готовий до негайного клонування без додаткових запитів).

3. Клієнт (Керуючий модуль) (TransactionsController.java та TemplateManagerController.java)

Ці класи використовують і керують прототипами, демонструючи переваги патерну.

TemplateManagerController:

- Працює як інтерфейс до сховища: завантажує (через TemplateDao) та відображає список готових прототипів.
- Надає можливість видалити прототип (templateDao.delete()).
- При виборі шаблону (подвійний клік або кнопка) він передає вибраний об'єкт-прототип до головного контролера через parentController.fillFormWithTemplate(selected).

TransactionsController:

- onSaveAsTemplate(): Використовує поточні дані форми для створення нового прототипу та збереження його через templateDao.create().
- fillFormWithTemplate(TransactionTemplate template): Це ключовий метод, де відбувається клонування. Він викликає template.createTransactionFromTemplate() і використовує отриманий клон для заповнення полів форми. Це гарантує, що форма заповнюється новим, незалежним об'єктом Transaction.

Впровадження патерну Прототип змінило логіку роботи з формою транзакцій та керування даними:

- Зміни в TemplateDao: Додано окрему таблицю transaction_templates для зберігання прототипів (шаблонів) у базі даних. TemplateDao тепер завантажує повністю ініціалізовані об'єкти TransactionTemplate (включно із завантаженими об'єктами Category та Account) при старті.
- Зміни в UI (Контролерах): Впроваджено зручний модальний діалог (TemplateManagerController) з функцією пошуку та видалення шаблонів.

- Прискорення створення: Коли користувач вибирає шаблон, TransactionsController не виконує додаткових запитів до БД. Він просто викликає метод на вибраному об'єкті-прототипі: Transaction clonedTx = template.createTransactionFromTemplate();. Це миттєво створює копію, яку можна редагувати та зберігати.
- Розділення відповідальності: Об'єкт TransactionTemplate сам знає, як створити свій клон (Transaction), знімаючи цю відповідальність із клієнтського коду (TransactionsController).

Переваги такого використання

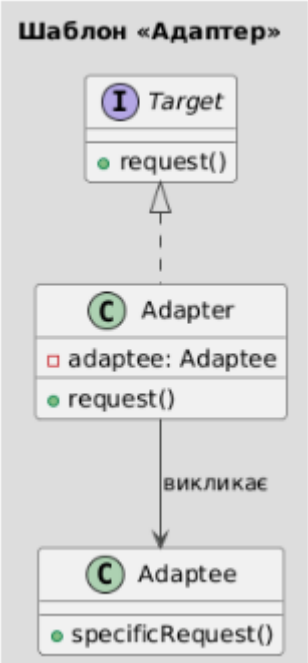
1. Швидкість (Продуктивність): Клонування об'єкта в пам'яті (зазвичай) значно швидше, ніж створення його з нуля, включаючи пошук і завантаження всіх залежних об'єктів (Category, Account) із бази даних.
2. Гнучкість у створенні: Патерн дозволяє створювати нові об'єкти з динамічною конфігурацією (шаблоном), не знаючи конкретного класу об'єкта, який створюється.
3. Зменшення залежностей: Клієнтський код (Controller) менше залежить від деталей створення (new Transaction()).

Відповіді на контрольні запитання:

1. Яке призначення шаблону «Адаптер»?

Адаптер — це структурний патерн проектування, що дає змогу об’єктам із несумісними інтерфейсами працювати разом. Ця реалізація використовує агрегацію: об’єкт адаптера «загортає», тобто містить посилання на службовий об’єкт. Такий підхід працює в усіх мовах програмування.

2. Нарисуйте структуру шаблону «Адаптер»



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

Target — інтерфейс, який очікує клієнт.

Adaptee — існуючий клас із несумісним інтерфейсом.

Adapter — адаптер, що перетворює інтерфейс Adaptee у Target.

Клієнт викликає метод request() у Adapter, який у свою чергу викликає specificRequest() у Adaptee. Адаптер повинен делегувати основну роботу сервісу.

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

1. Адаптер на рівні об'єктів (Композиція)

Це переважний підхід, оскільки він відповідає принципу "віддавати перевагу композиції над успадкуванням".

- Як працює: Клас-Адаптер містить посилання (або екземпляр) на об'єкт Adaptee (клас, який потрібно адаптувати). Адаптер реалізує цільовий Target-інтерфейс, який очікує клієнт. Коли клієнт викликає метод цільового

інтерфейсу, Адаптер просто делегує виклик відповідному методу об'єкта *Adaptee*.

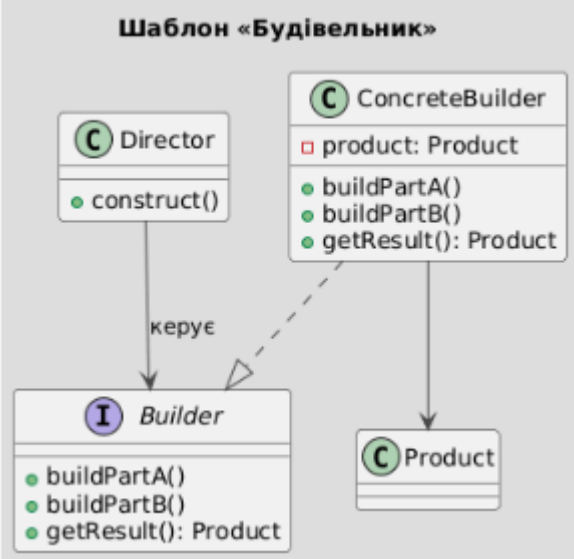
- Перевага: Гнучкість, оскільки об'єкт *Adaptee* можна змінити в рантаймі.

2. Адаптер на рівні класів (Множинне успадкування)

- Як працює: Клас-Адаптер успадковує як цільовий Target-інтерфейс, так і клас Adaptee. Завдяки успадкуванню від *Adaptee*, Адаптер отримує його поведінку, а реалізуючи *Target*-інтерфейс, він робить цю поведінку доступною для клієнта через потрібний інтерфейс.
- Обмеження: Необхідна підтримка множинного успадкування класів мовою програмування (наприклад, C++). У мовах, як Java або C#, де класи можуть успадковувати лише один клас, цей підхід можливий, лише якщо *Target* є інтерфейсом, а *Adaptee* — класом (або обидва є інтерфейсами).

5. Яке призначення шаблону «Будівельник»?
Будівельник — це породжувальний патерн проектування, що дає змогу створювати складні об’єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображень об’єктів.

6. Нарисуйте структуру шаблону «Будівельник».



7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?
Взаємодія між цими класами забезпечує покрокове створення складного об'єкта.
1. Продукт (Product)

- Це складний об'єкт, що створюється. Він містить набір частин, які збираються за допомогою Будівельника.
- *Взаємодія:* Конкретний Будівельник створює і збирає частини Продукту.

2. Будівельник (Builder)

- Це абстрактний інтерфейс або абстрактний клас, який визначає кроки для створення частин Продукту.
- Він декларує методи для конструювання частин, а також метод для отримання кінцевого результату.

- *Взаємодія:* Усі Конкретні Будівельники реалізують цей інтерфейс.

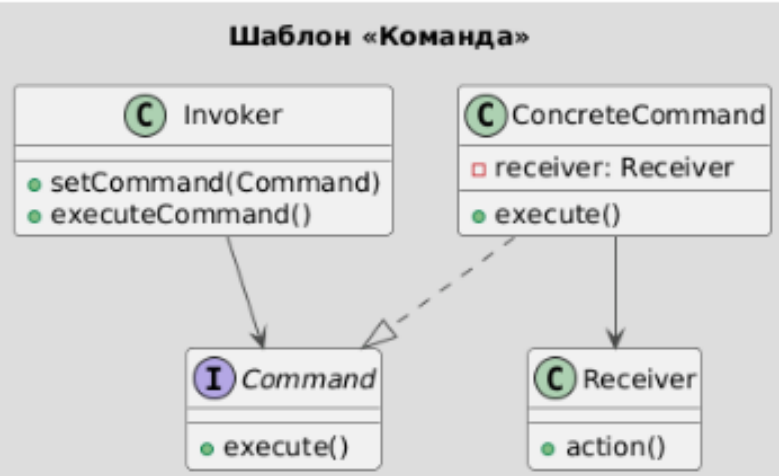
3. Конкретний Будівельник (Concrete Builder)
   - Реалізує інтерфейс Будівельника.
   - Виконує покрокове конструювання, зберігаючи та збираючи частини Продукту.
   - Кожен Конкретний Будівельник відповідає за створення одного конкретного подання (версії) Продукту.
   - *Взаємодія:* Отримує інструкції від Розпорядника та повертає кінцевий Продукт клієнту або Розпоряднику.
4. Розпорядник/Керівник (Director)
   - Керує процесом конструювання. Він знає, в якій послідовності викликати методи Будівельника, щоб створити певний Продукт.
   - Клієнт зазвичай створює Розпорядника та передає йому Конкретного Будівельника.
   - *Взаємодія:* Розпорядник працює лише з абстрактним інтерфейсом Будівельника, що дозволяє йому використовувати будь-якого Конкретного Будівельника без зміни свого коду.

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

- Коли процес створення об’єкта складний або має багато кроків.
- Коли потрібно мати різні представлення одного й того ж об’єкта.
- Коли потрібно відокремити створення об’єкта від його структури.

9. Яке призначення шаблону «Команда»?
Команда — це поведінковий патерн проектування, який перетворює запити на об’єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій.

10. Нарисуйте структуру шаблону «Команда».



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

Взаємодія між цими класами забезпечує повне відділення Ініціатора (об'єкта, який ініціює дію) від Отримувача (об'єкта, який знає, як виконати дію).

1. Команда (Command)
   - Це інтерфейс або абстрактний клас, який визначає єдиний метод для виконання операції, наприклад, execute().
   - *Взаємодія:* Усі Конкретні Команди реалізують цей інтерфейс.
2. Конкретна Команда (Concrete Command)
   - Реалізує інтерфейс Команди.
   - Зберігає посилання на об'єкт Отримувача.
   - Зберігає всі параметри, необхідні для виклику методу Отримувача.
   - Метод execute() передає виклик відповідному методу Отримувача (делегування).
   - *Взаємодія:* Створюється Клієнтом, передається Ініціатору, а при виконанні викликає методи Отримувача.
3. Отримувач (Receiver)
   - Це клас-виконавець, який знає, як виконати фактичну операцію.
   - Містить бізнес-логіку операції.
   - Він не знає і не залежить від того, хто його викликає.
   - *Взаємодія:* Конкретна Команда викликає його методи для виконання дії.
4. Ініціатор/Виклик (Invoker)
   - Це об'єкт, який запитує виконання Команди.
   - Він зберігає (або реєструє) посилання на об'єкт Команди і викликає її метод execute() у відповідний момент.
   - Не знає ні про Конкретну Команду, ні про Отримувача. Він працює лише з інтерфейсом Команди.
   - *Взаємодія:* Отримує Команду від Клієнта і запускає її виконання.

12. Розкажіть як працює шаблон «Команда».

Клієнт створює об’єкт ConcreteCommand, передаючи йому посилання на Receiver.

Invoker отримує команду і викликає метод execute().

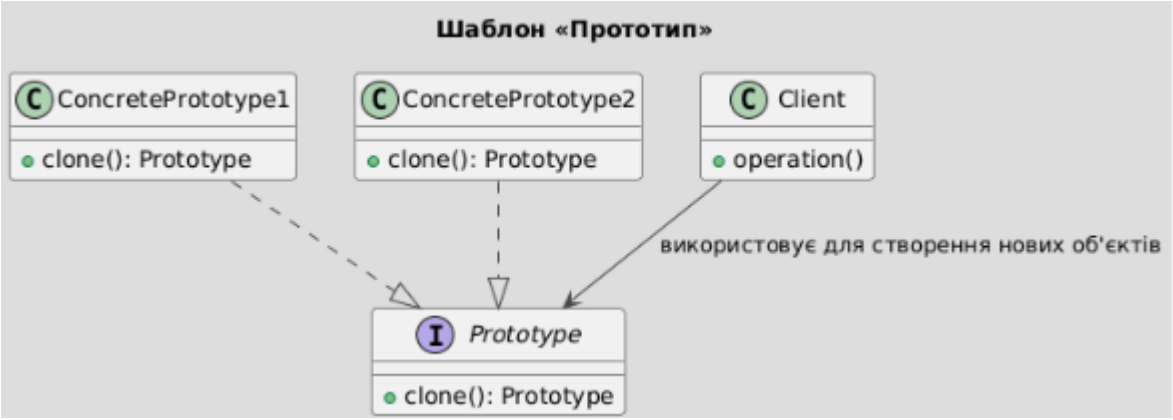
ConcreteCommand делегує виконання Receiver.

Таким чином дії можна зберігати, чергувати або скасовувати.

13. Яке призначення шаблону «Прототип»?

Прототип — це породжувальний патерн проектування, що дає змогу копіювати об’єкти, не вдаючись у подробиці їхньої реалізації.

14. Нарисуйте структуру шаблону «Прототип».



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

Взаємодія в цьому шаблоні дуже проста і зосереджена на механізмі клонування.

1. Прототип (Prototype)
   - Це інтерфейс або абстрактний клас, який оголошує операцію клонування (зазвичай метод clone() або copy()).
   - Цей метод може бути оголошений як абстрактний або містити реалізацію, якщо мова підтримує це (наприклад, реалізація інтерфейсу Cloneable в Java).
   - *Взаємодія:* Усі Конкретні Прототипи повинні реалізувати цей інтерфейс.
2. Конкретний Прототип (Concrete Prototype)
   - Це клас, який реалізує операцію клонування, оголошену в інтерфейсі Прототипу.
   - У реалізації методу клонування він створює копію самого себе, використовуючи, як правило, механізми копіювання на рівні мови програмування (поверхневе або глибоке копіювання).
   - *Взаємодія:* Клієнт викликає його метод clone() для отримання нового об'єкта.
3. Клієнт (Client)
   - Це об'єкт, який створює нові екземпляри.
   - Замість того, щоб викликати конструктор (оператор new), Клієнт викликає метод clone() на об'єкті-Прототипі.
   - Клієнт працює тільки з інтерфейсом Прототипу, що дозволяє йому копіювати будь-який Конкретний Прототип без знання його конкретного класу.
   - *Взаємодія:* Зберігає або отримує посилання на існуючий об'єкт-Прототип та викликає clone().

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

Middleware (Веб): Обробка HTTP-запитів. Ланцюжок перевіряє аутентифікацію, авторизацію, валідацію і лише потім передає запит основному контролеру.

Система затвердження: Заявка на фінансові витрати проходить ланцюжок ієрархічно: Менеджер відділу → Керівник департаменту → Директор. Кожен обробник має свій ліміт повноважень.

Логування: Повідомлення з різними рівнями (DEBUG, ERROR, FATAL) послідовно обробляються різними логерами: Консольний логер → Файловий логер → Логер електронної пошти.

Переваги:

Зниження зв'язаності: Клієнт не знає, який об'єкт оброблятиме його запит.

Гнучкість: Можна динамічно змінювати порядок або склад ланцюжка.

Висновки

У ході лабораторної роботи було вивчено структуру та принципи роботи шаблонів Adapter, Builder, Command, Chain of Responsibility і Prototype. Отримано практичні навички їх реалізації та побудови UML-діаграм.

У практичній частині на прикладі програми «Особиста бухгалтерія» ми реалізували патерн «Прототип», що дозволив швидко створювати нові об’єкти Transaction на основі вже існуючих шаблонів, підвищивши ефективність та зручність розробки.