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

Технології розроблення програмного забезпечення

Тема: «Особиста бухгалтерія»

Виконала:
Студентка групи ІА-34
Дригант А.С
Перевірив:
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.).

Теоритичні відомості

Патерн Adapter (Адаптер)

Призначення: дозволяє “пристосувати” (адаптувати) інтерфейс одного класу до іншого, який очікує клієнт.

Суть: створюється проміжний клас-обгортка (adapter), який реалізує потрібний інтерфейс і всередині викликає методи несумісного об’єкта.

Приклад: різні бібліотеки для відтворення музики мають різні інтерфейси — адаптер уніфікує їх через спільний IPlayer.

Переваги:

- відокремлює інтерфейс від бізнес-логіки;
- легко додавати нові адаптери без зміни існуючого коду.

Недолік: збільшується кількість класів.

Патерн Builder (Будівельник)

Призначення: відділяє процес побудови складного об’єкта від його представлення.

Суть: будівельник крок за кроком створює об’єкт (наприклад, web-сторінку з заголовками, тілом тощо).

Переваги:

- дає контроль над процесом створення;
- дозволяє створювати різні представлення одного продукту.

Недоліки:

- клієнт залежить від конкретних класів будівельників.

Патерн Command (Команда)

Призначення: перетворює дію (виклик методу) в окремий об’єкт.

Суть: кожна команда має метод execute(), який викликає необхідну дію в отримувача.

Приклад: у графічному застосунку команда “Копіювати” може бути викликана з меню, кнопки або контекстного меню.

Переваги:

- відокремлює ініціатора від виконавця;
- легко реалізувати скасування (undo), повтор (redo) і логування дій;
- просте додавання нових команд.

Патерн Chain of Responsibility (Ланцюжок відповідальності)

Призначення: дозволяє передавати запит по ланцюжку обробників, поки хтось не обробить його.

Суть: кожен обробник має посилання на наступного в ланцюжку. Якщо не може обробити - передає далі.

Приклад: формування контекстного меню — кожен компонент додає свої пункти і викликає UpdateContextMenu() у батьківського елемента.

Переваги:

- клієнт не знає, хто саме обробить запит;
- легко додати або видалити обробників.

Недолік: запит може залишитися необробленим.

Патерн Prototype (Прототип)

Призначення: створення нових об’єктів шляхом копіювання існуючих

(прототипів).

Суть: замість створення через new, використовується метод clone(), який повертає копію об’єкта.

Приклад: у редакторі рівнів 2D-гри кнопки зберігають прототипи елементів (стіна, сходи) і додають їх шляхом клонування.

Переваги:

- швидке створення складних об’єктів;
- дозволяє уникнути надмірного наслідування;
- гнучкість - клоновані об’єкти можна змінювати незалежно.

Недоліки:

- складність реалізації глибокого копіювання;
- надмірне використання ускладнює код.

Хід роботи

Діаграма класів

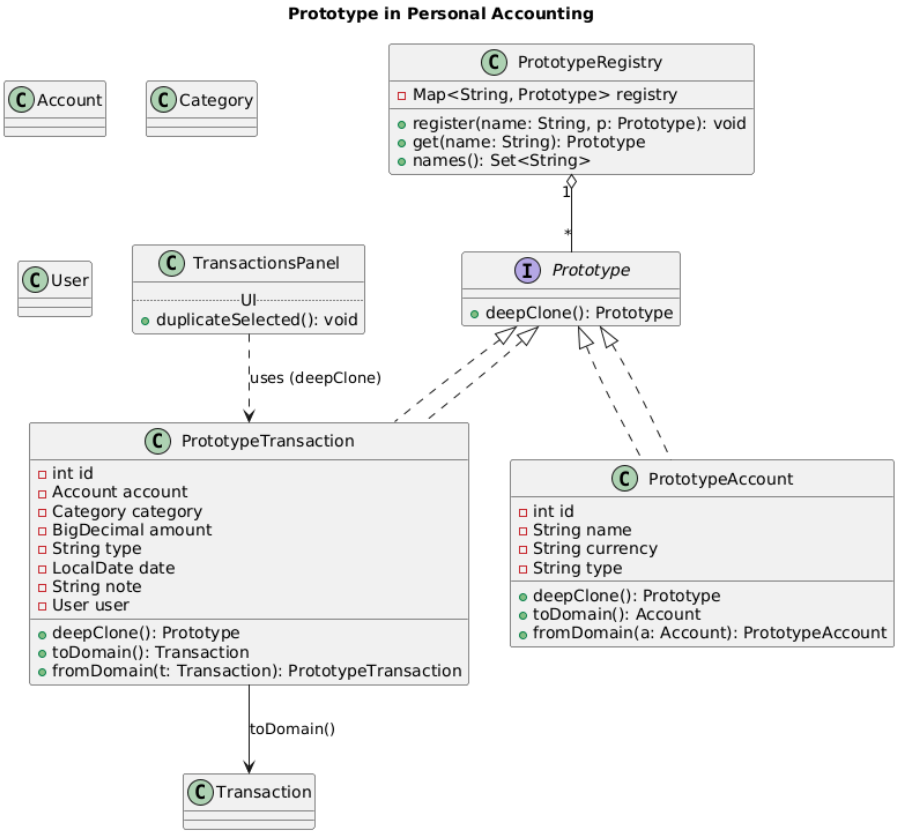


Рисунок 5.1 - Діаграма класів

На діаграмі показано реалізацію патерну Prototype у системі особистої бухгалтерії. Центральним елементом є інтерфейс Prototype, який визначає єдиний метод клонування (deepClone() або copy()), що дозволяє створювати нові об’єкти без використання конструктора. Від нього наслідуються конкретні реалізації - PrototypeTransaction і PrototypeAccount, які відповідають за копіювання відповідно транзакцій і рахунків.

Кожен конкретний прототип містить методи toDomain() і fromDomain(), які забезпечують перетворення між об’єктом-доменом (наприклад, Transaction) і його прототипом. Таким чином можна як створити шаблон із наявного об’єкта, так і відновити новий екземпляр на його основі.

Клас PrototypeRegistry реалізує реєстр усіх доступних шаблонів системи - він зберігає об’єкти-прототипи у структурі Map і дозволяє створювати їх копії за ключем. Це спрощує доступ до часто використовуваних шаблонів, таких як «зарплата», «комунальні послуги» чи «переказ».

У графічному інтерфейсі користувача (TransactionsPanel) реалізовано приклад практичного застосування патерну - метод duplicateSelected() створює копію виділеної транзакції, використовуючи PrototypeTransaction. Користувач може дублювати записи без повторного введення даних, а система автоматично оновлює таблицю з транзакціями.

Допоміжні класи Account, Category, User і Transaction показані для контексту - вони відображають реальні сутності доменної моделі, з якими працюють прототипи.

1) Інтерфейс прототипа

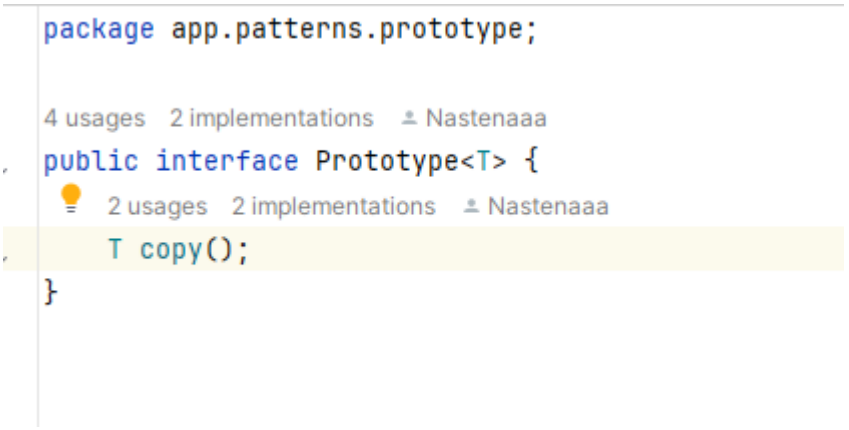


Рисунок 5.2 - файл Prototype

У файлі Prototype.java оголошено узагальнений інтерфейс, який є базою для реалізації патерну Prototype. Його призначення - визначити єдиний контракт для всіх класів, що підтримують створення копій своїх об’єктів. Інтерфейс містить лише один метод - copy(), який реалізується у конкретних класах-прототипах. Цей метод повертає новий об’єкт із такими самими властивостями, як у вихідного, що дозволяє створювати нові екземпляри без використання конструктора. Таким чином, інтерфейс забезпечує гнучкість і незалежність коду, дозволяючи дублювати об’єкти будь-якого типу єдиним способом.

2) Конкретний прототип

2.1) Транзакція

```java
package app.patterns.prototype;

import ...

4 usages  Nastenaaa
public class PrototypeTransaction implements Prototype<Transaction> {
    8 usages
    private final Transaction src;

    1 usage  Nastenaaa
    private PrototypeTransaction(Transaction src) { this.src = src; }

    Nastenaaa
    public static PrototypeTransaction from(Transaction t) { return new PrototypeTransaction(t); }

    2 usages  Nastenaaa
    @Override
    public Transaction copy() {
        Transaction t = new Transaction();
        t.setAccount(src.getAccount());
        t.setCategory(src.getCategory());
        t.setType(src.getType());
        t.setAmount(src.getAmount());
        t.setDate(src.getDate());
        t.setNote(src.getNote());
        t.setUser(src.getUser());
        return t;
    }


    1 usage  Nastenaaa
    public Transaction copyWith(LocalDate date, BigDecimal amount, String note) {
        Transaction t = copy();
        if (date   != null) t.setDate(date);
        if (amount != null) t.setAmount(amount);
        if (note   != null) t.setNote(note);
        return t;
    }
}
```

Рисунок 5.3 - клас PrototypeTransaction

У класі PrototypeTransaction реалізовано логіку створення нових екземплярів транзакцій шляхом копіювання вже існуючих, що повністю відповідає ідеї патерну Prototype. Клас зберігає посилання на вихідний об’єкт Transaction, який виступає шаблоном, і на його основі створює нову копію з тими самими властивостями. Такий підхід усуває необхідність повторного заповнення всіх полів вручну, що особливо зручно при роботі з повторюваними операціями, наприклад щомісячними витратами або регулярними доходами.

Метод copy() виконує глибоке копіювання основних характеристик транзакції - рахунку, категорії, типу, суми, дати, нотатки та користувача, але не дублює ідентифікатор, що гарантує створення нового запису. Завдяки цьому забезпечується коректна поведінка під час збереження в базу даних.

Метод copyWith() розширює можливості копіювання, дозволяючи змінювати окремі параметри, наприклад дату або суму, при збереженні решти атрибутів без змін. Це дає змогу швидко створювати нові транзакції на основі існуючих, змінюючи лише кілька полів.

Загалом, цей клас забезпечує ефективне повторне використання вже створених об’єктів, підвищує гнучкість системи та спрощує користувачеві введення однотипних даних, оскільки більшість значень копіюються автоматично.

2.2) Рахунок

```java
package app.patterns.prototype;

import app.model.Account;

2 usages  Nastenaaa
public class PrototypeAccount implements Prototype<Account> {
    5 usages
    private final Account src;

    1 usage  Nastenaaa
    private PrototypeAccount(Account src) { this.src = src; }

    Nastenaaa
    public static PrototypeAccount from(Account a) { return new PrototypeAccount(a); }

    2 usages  Nastenaaa
    @Override
    public Account copy() {
        Account c = new Account();
        c.setUser(src.getUser());
        c.setName(src.getName());
        c.setCurrency(src.getCurrency());
        c.setType(src.getType());

        return c;
    }
}
```

Рисунок 5.4 – клас PrototypeAccount

У класі PrototypeAccount реалізовано конкретний прототип для створення копій об’єктів типу Account. Його головне завдання - спростити процес створення нових

рахунків, базуючись на вже існуючих, зберігаючи при цьому їх основні характеристики. Клас містить посилання на вихідний об’єкт Account, який виступає шаблоном для копіювання. Під час виклику методу copy() створюється новий екземпляр рахунку, у який копіюються дані користувача, назва рахунку, валюта та тип. Такі поля, як ідентифікатор чи баланс, не дублюються - вони залишаються порожніми або ініціалізуються пізніше, що забезпечує створення дійсно нового об’єкта в системі.

Ця реалізація дозволяє швидко створювати однотипні рахунки, наприклад для сімейного бюджету чи різних валютних фондів, без необхідності повторного введення всіх параметрів. Клас чітко демонструє переваги патерну Prototype, адже замість створення нових об’єктів через конструктор використовується механізм копіювання з готового зразка, що підвищує зручність та ефективність роботи програми.

3) Реєстр прототипів (шаблонів)

```java
package app.patterns.prototype;

import java.util.HashMap;
import java.util.Map;

no usages  Nastenaaa
public class PrototypeRegistry {
    2 usages
    private final Map<String, Object> map = new HashMap<>();

    no usages  Nastenaaa
    public <T> void put(String key, Prototype<T> proto) { map.put(key, proto); }

    /unchecked/
    public <T> T create(String key) {
        Object p = map.get(key);
        if (p == null) throw new IllegalArgumentException("No prototype: " + key);
        return ((Prototype<T>) p).copy();
    }
}
```

Рисунок 5.5 - клас PrototypeRegistry

Клас PrototypeRegistry виконує роль централізованого сховища для всіх створених прототипів системи. Він реалізує механізм реєстрації та отримання шаблонів, що дозволяє легко керувати різними типами об’єктів, які можна клонувати. У цьому

класі використовується колекція Map, де кожному прототипу призначається унікальний ключ.

Метод put() додає новий прототип у реєстр, забезпечуючи можливість швидкого доступу до нього в будь-якому місці програми. Метод create() отримує збережений прототип за заданим ключем і викликає його метод copy(), створюючи новий об’єкт на основі зареєстрованого зразка. Якщо вказаного шаблону не існує, система генерує виняток, що запобігає помилкам при використанні.

Завдяки цьому класу вся робота з шаблонами стає централізованою: замість створення об’єктів вручну, достатньо один раз зареєструвати їх у реєстрі та потім викликати create(). Такий підхід підвищує гнучкість, розширюваність і зменшує залежність між компонентами системи, чітко відображаючи ідею патерну Prototype - створення об’єктів на основі готових прототипів.

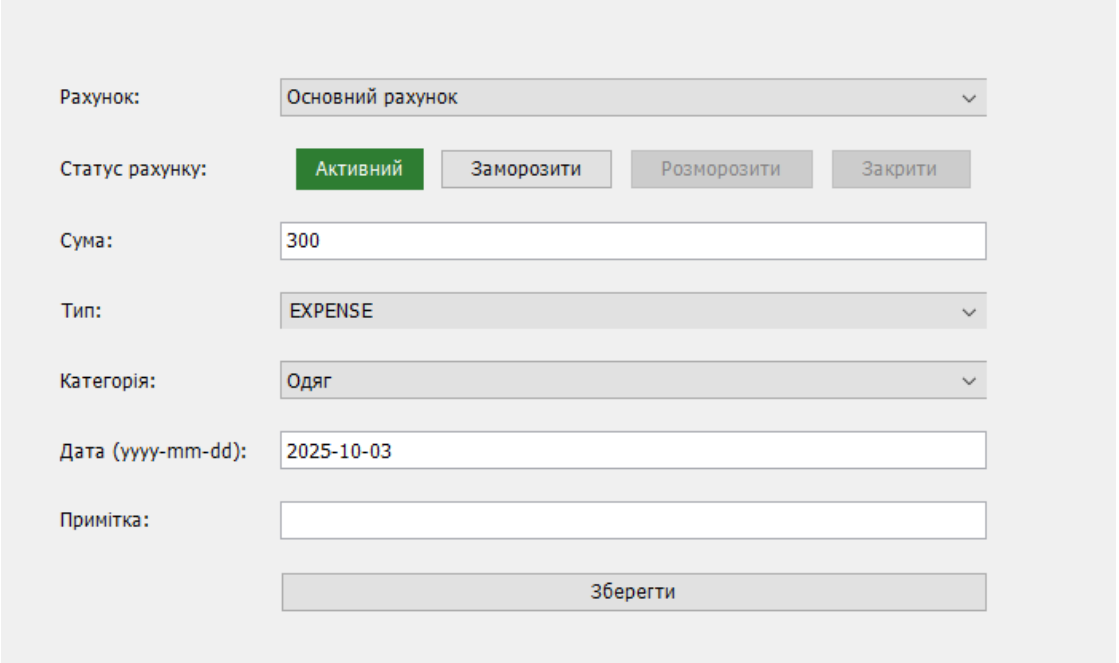
4) Використання в UI: дублювання транзакції з таблиці

```
private void duplicateSelected() {
    int viewRow = table.getSelectedRow();
    if (viewRow < 0) {
        JOptionPane.showMessageDialog( parentComponent: this,  message: "Виберіть рядок у таблиці для дублювання.");
        return;
    }
    int row = table.convertRowIndexToModel(viewRow);
    if (row < 0 || row >= current.size()) return;

    Transaction original = current.get(row);
    try {
        Transaction clone = PrototypeTransaction
                .from(original)
                .copyWith(LocalDate.now(),  amount: null,  note: "Повтор: " + (original.getNote() == null ? "" : original.getNote()));

        txRepo.save(clone);
        JOptionPane.showMessageDialog( parentComponent: this,  message: "Створено копію транзакції.");
        load();
    } catch (Exception ex) {
        ex.printStackTrace();
        JOptionPane.showMessageDialog( parentComponent: this,  message: "Не вдалося дублювати: " + ex.getMessage(),
                title: "Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

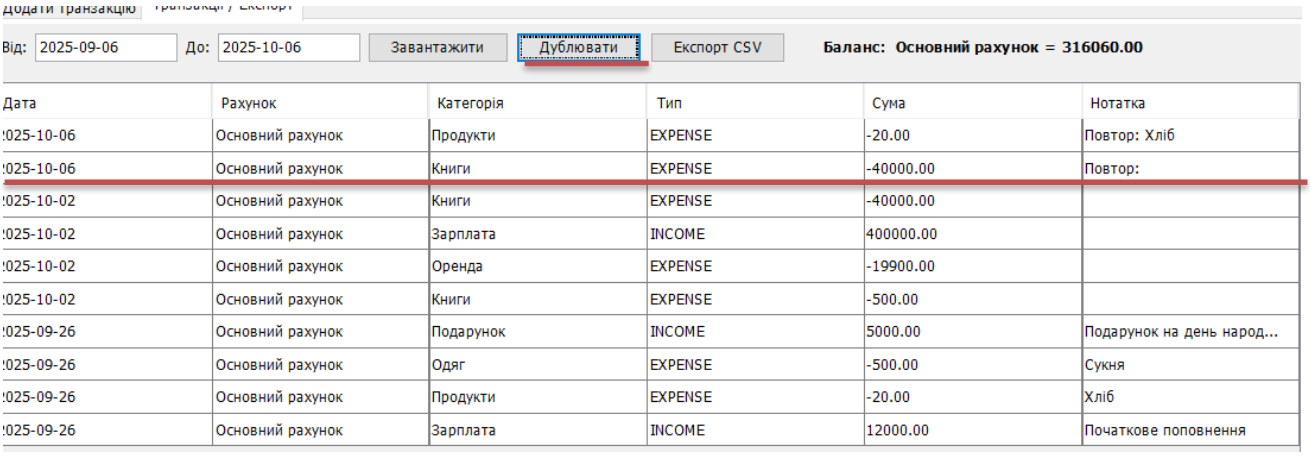
Рисунок 5.6 - використання в UI: дублювання транзакції з таблиці

У методі duplicateSelected() я реалізую живе застосування патерну Prototype прямо з інтерфейсу. Спочатку перевіряю, чи користувач виділив рядок у таблиці; якщо ні - показую повідомлення і зупиняю обробку. Далі перетворюю індекс видимого рядка на індекс у моделі, щоб коректно дістати відповідний об’єкт із кешованого списку current. Отримавши вихідну транзакцію, створюю її копію через прототип:

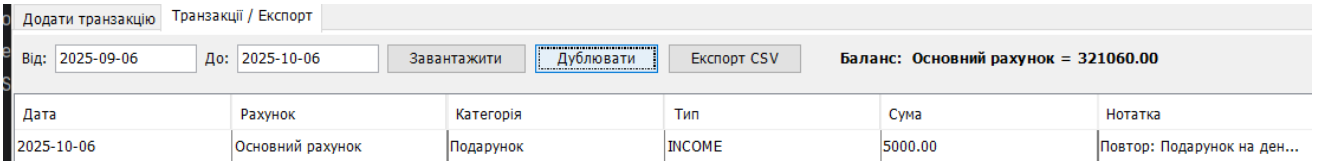
формую PrototypeTransaction з оригіналу і викликаю copyWith(...), де підміняю лише потрібні поля - ставлю поточну дату і додаю службову позначку в нотатку, а решта даних (рахунок, категорія, тип, сума, користувач) переходять із шаблону без змін. Нову транзакцію зберігаю через репозиторій (txRepo.save(...)), показую успішне повідомлення, після чого оновлюю таблицю викликом load() (і, за потреби, підсумки балансів). Якщо під час процесу виникає помилка (наприклад, не знайдено прототип чи проблеми з БД), я перехоплюю виняток, логую стек і виводжу повідомлення про помилку. Таким чином користувач одним кліком отримує новий запис, створений клонуванням вибраної транзакції без повторного ручного введення даних.



Повтор транзакцій:



Дата	Рахунок	Категорія	Тип	Сума	Нотатка
2025-10-06	Основний рахунок	Продукти	EXPENSE	-20.00	Повтор: Хліб
2025-10-06	Основний рахунок	Книги	EXPENSE	-40000.00	Повтор:
2025-10-02	Основний рахунок	Книги	EXPENSE	-40000.00	
2025-10-02	Основний рахунок	Зарплата	INCOME	400000.00	
2025-10-02	Основний рахунок	Оренда	EXPENSE	-19900.00	
2025-10-02	Основний рахунок	Книги	EXPENSE	-500.00	
2025-09-26	Основний рахунок	Подарунок	INCOME	5000.00	Подарунок на день народ...
2025-09-26	Основний рахунок	Одяг	EXPENSE	-500.00	Сукня
2025-09-26	Основний рахунок	Продукти	EXPENSE	-20.00	Хліб
2025-09-26	Основний рахунок	Зарплата	INCOME	12000.00	Початкове поповнення



Висновок

У результаті виконання лабораторної роботи я реалізувала патерн Prototype у системі «Personal Accounting». Його використання дало змогу створювати нові об’єкти - транзакції та рахунки - шляхом клонування вже існуючих зразків, без необхідності повторного введення даних. Завдяки цьому вдалося спростити логіку програми, підвищити гнучкість і зменшити кількість дублювання коду. Реєстр прототипів забезпечив централізоване зберігання шаблонів, а інтеграція з інтерфейсом користувача дозволила дублювати записи безпосередньо з таблиці. Такий підхід показав практичну користь патерну Prototype у розробці застосунків, де часто виникає потреба створювати однотипні об’єкти з невеликими змінами.

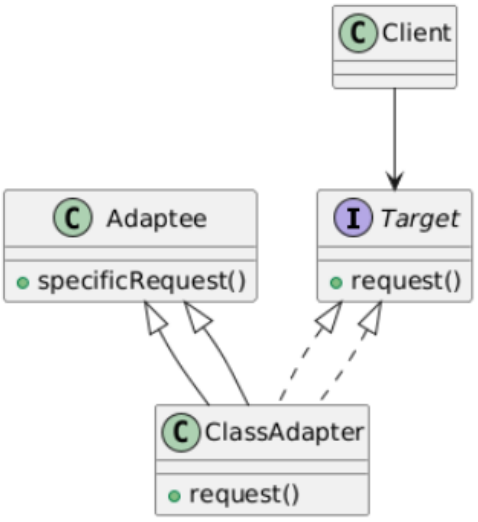
Контрольні питання

1. Яке призначення шаблону «Адаптер»?

Патерн «Адаптер» (Adapter) використовується для узгодження несумісних інтерфейсів двох класів. Його головна мета - дозволити взаємодію об’єктів, які зазвичай не можуть співпрацювати через різні інтерфейси. Наприклад, коли новий компонент програми повинен використовувати старий модуль або бібліотеку, але їхні методи мають різні назви чи сигнатури.

Адаптер «обгортає» старий клас, реалізуючи очікуваний інтерфейс клієнта, і всередині делегує виклики до оригінального об’єкта. Це дозволяє клієнту працювати з єдиним уніфікованим інтерфейсом, не змінюючи вже написаний код.

2. Нарисуйте структуру шаблону «Адаптер».



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

1. Target - інтерфейс або абстрактний клас, який визначає набір методів, зручних для клієнта.
2. Adaptee - існуючий клас із відмінним інтерфейсом, який треба «пристосувати».
3. Adapter - клас-посередник, який реалізує Target і всередині містить об’єкт Adaptee.
4. Client - використовує лише Target, не знаючи про Adaptee.

Коли клієнт викликає метод з Target, адаптер перетворює цей виклик у виклик методів Adaptee. Таким чином, адаптер виступає посередником між двома несумісними системами, роблячи їх взаємодію можливою.

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

- Адаптер на рівні об’єктів (Object Adapter) використовує композицію - він має поле типу Adaptee і делегує йому виклики. Такий підхід гнучкий, бо можна змінити об’єкт під час виконання або адаптувати кілька різних класів одночасно.
- Адаптер на рівні класів (Class Adapter) використовує множинне наслідування (якщо це підтримується мовою). У цьому випадку адаптер одночасно наслідує Target і Adaptee, перевизначаючи методи

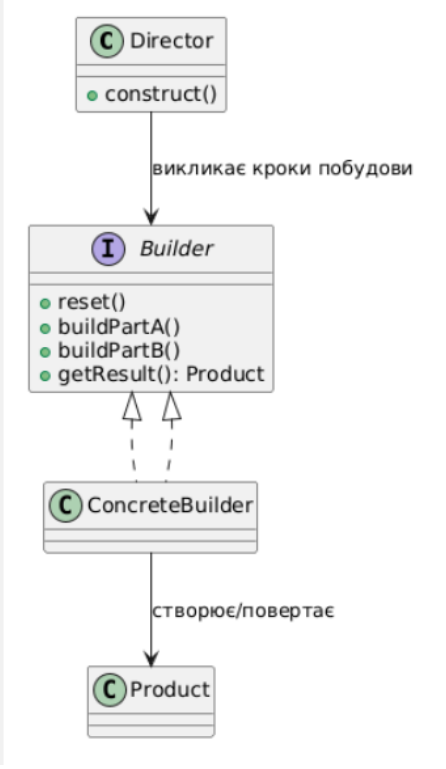
для узгодження. Такий підхід швидший, але менш гнучкий і сильніше прив’язаний до конкретної реалізації.

5. Яке призначення шаблону «Будівельник»?

Патерн «Будівельник» (Builder) призначений для поетапного створення складних об’єктів. Він відділяє процес побудови об’єкта від його внутрішньої структури, дозволяючи створювати різні варіації того самого продукту з використанням одного й того самого коду.

Будівельник корисний тоді, коли об’єкт має багато обов’язкових і необов’язкових параметрів або коли процес створення повинен виконуватись послідовно. Завдяки цьому забезпечується чистота коду, а конструктори не перевантажуються великою кількістю параметрів.

6. Нарисуйте структуру шаблону «Будівельник».



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

- Client створює екземпляр ConcreteBuilder і передає його Director.
- Director керує порядком виклику методів будівельника (алгоритмом побудови).
- Builder визначає інтерфейс для створення частин об’єкта.
- ConcreteBuilder реалізує цей інтерфейс, поступово створюючи

продукт.

- Product - кінцевий результат.

Клієнт може використовувати одного й того самого Director з різними ConcreteBuilder, отримуючи різні варіації об’єкта.

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

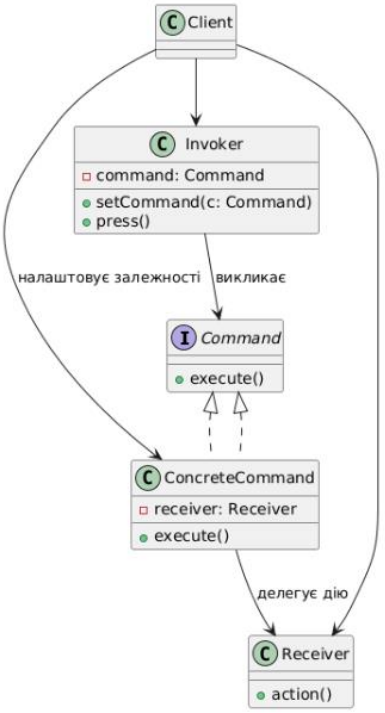
1. Коли потрібно створювати складні об’єкти з багатьма полями.
2. Коли конструктори класу мають занадто багато параметрів, і потрібно спростити їх виклик
3. Коли процес створення об’єкта має виконуватись у кілька кроків.
4. Коли потрібно мати можливість створювати різні варіанти одного продукту, змінюючи лише послідовність побудови або конкретного будівельника.

9. Яке призначення шаблону «Команда»?

Патерн «Команда» (Command) перетворює запити або дії користувача у самостійні об’єкти. Це дозволяє передавати дії як параметри, зберігати їх у чергах, скасовувати або повторювати.

Його головна мета - відокремити об’єкт, який ініціює дію (Invoker), від об’єкта, який її виконує (Receiver). Завдяки цьому можна легко додавати нові команди, не змінюючи код клієнта чи інтерфейсу користувача.

10. Нарисуйте структуру шаблону «Команда».



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

Клієнт створює об’єкт Receiver і команду (ConcreteCommand), зв’язує їх, потім передає команду Invoker. Коли користувач виконує дію (натискає кнопку, вибирає пункт меню), Invoker викликає метод execute(), який делегує виконання Receiver. Також команду можна зберегти для повторного виконання або для скасування (undo()).

12. Розкажіть як працює шаблон «Команда».

Патерн інкапсулює операції в об’єкти. Кожна команда знає, що потрібно зробити (Receiver) і як це зробити (метод).

Invoker викликає команди, не знаючи, як вони реалізовані. Це дозволяє легко додавати нові дії без зміни інтерфейсу користувача.

Наприклад, у текстовому редакторі кнопки «Скасувати», «Повторити» або «Вставити» можуть бути реалізовані як окремі команди. Команда «Вставити» зберігає вставлений текст і при натисканні «Скасувати» викликає метод undo(), який видаляє вставлений фрагмент.

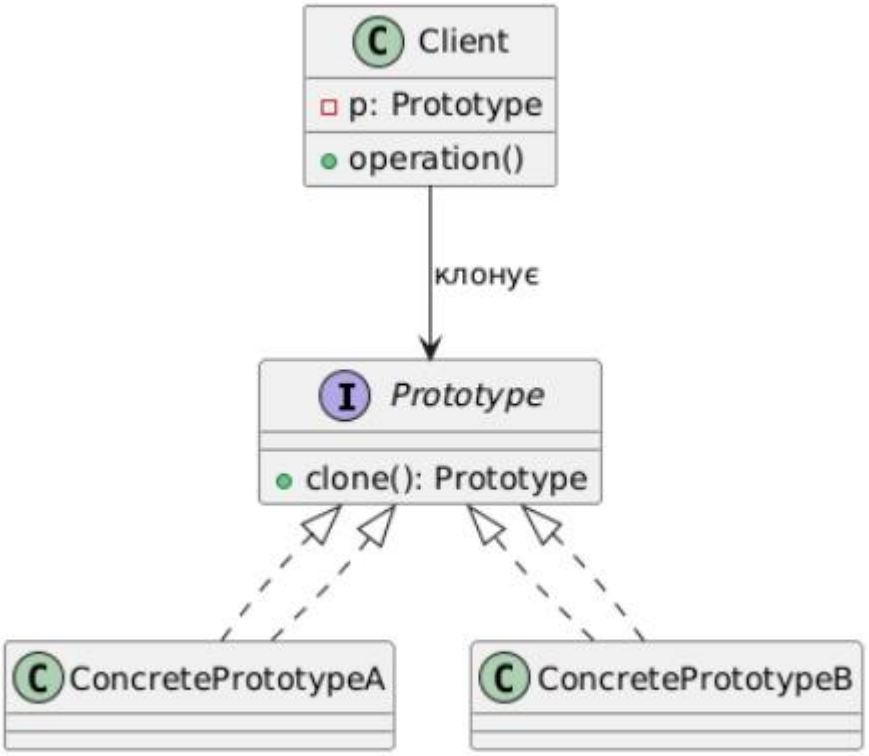
13. Яке призначення шаблону «Прототип»?

Патерн «Прототип» (Prototype) дає змогу створювати нові об’єкти шляхом клонування вже існуючих, а не через створення їх з нуля. Це особливо

корисно, коли створення об’єкта є складним або ресурсомістким процесом, або коли потрібно часто створювати однотипні екземпляри з незначними змінами.

Основна перевага - скорочення витрат часу на ініціалізацію та підвищення гнучкості системи, адже зразки можна динамічно налаштовувати під час виконання.

14. Нарисуйте структуру шаблону «Прототип».



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

Prototype - описує інтерфейс клонування.

ConcretePrototype - знає, як копіювати себе (глибоке або поверхневе копіювання).

PrototypeRegistry - керує колекцією шаблонів, з яких можна створювати копії.

Client - використовує зареєстрований прототип для створення нового об’єкта.

Взаємодія: Client запитує потрібний зразок у PrototypeRegistry, викликає

copy(), отримує новий екземпляр, який можна модифікувати незалежно від оригіналу.

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

Патерн «Ланцюжок відповідальності» (Chain of Responsibility) використовується, коли потрібно передати запит уздовж ланцюжка обробників, доки один із них не обробить його. Це дозволяє динамічно змінювати порядок обробки або додавати нових обробників без зміни коду клієнта.

Приклади:

- Інтерфейси користувача: обробка подій натискання миші - від елемента до контейнера (наприклад, текст → панель → форма).
- Веб-запити: HTTP-фільтри або middleware (автентифікація → авторизація → логування → кеш).
- Бізнес-процеси: погодження документів різними рівнями менеджерів.
- Обробка помилок: передача винятку від нижнього рівня до вищого, доки його хтось не обробить.
- Валідація даних: послідовна перевірка різними валідаторами (формат, обов’язковість, права доступу тощо).