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 розробки програмного забезпечення

“ Патерни проектування.”

“Download manager”

Виконав студент групи ІА–33:
Ященко К.А.

Київ 2025

Тема: Патерни проєктування

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

Зміст

Завдання

- Ознайомитись з короткими теоретичними відомостями.
- Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
- Реалізувати один з розглянутих шаблонів за обраною темою.
- Реалізувати не менше 3-х класів відповідно до обраної теми.
- Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

Теоретичні відомості

Адаптер (Adapter)
Патерн Адаптер — це, по суті, "перехідник" або "перекладач". Його головне завдання — змусити два об'єкти працювати разом, навіть якщо їхні інтерфейси абсолютно несумісні. Уявіть, що у вас є ноутбук з сучасним портом USB-C, але ваша улюблена стара мишка має штекер USB-A. Ви не можете підключити їх напряму. Адаптер стає посередником, який "перекладає" сигнали з

одного інтерфейсу на інший, дозволяючи їм працювати разом. У програмуванні це виглядає так само: у вас є новий клас, який очікує отримати об'єкт з методом doWork(), але у вас є стара бібліотека, де схожий клас має метод executeLegacyTask(). Ви створюєте клас-Адаптер, який реалізує потрібний інтерфейс doWork(), але всередині себе він просто викликає метод

executeLegacyTask() старого об'єкта. Таким чином, ви "адаптуєте" старий код до нового, не змінюючи сам старий код, що особливо корисно при роботі зі сторонніми бібліотеками.

Будівельник (Builder)
Патерн Будівельник вирішує проблему створення складних об'єктів. Уявіть, що вам потрібно створити об'єкт "Комп'ютер". У нього може бути 10-15 характеристик: процесор, відеокарта, об'єм пам'яті, тип диска, блок живлення, материнська плата тощо. Створювати такий об'єкт через один гігантський конструктор (new Computer("i9", "RTX4090", 32, ...)), де половина параметрів може бути не потрібна (наприклад, немає дискретної відеокарти), — це просто жахіття. Будівельник пропонує інший підхід: він дозволяє "збирати" об'єкт крок за кроком. Ви створюєте окремий об'єкт-Будівельник і даєте йому послідовні команди: builder.setCPU("i9"), builder.setRAM(32), builder.setGPU("RTX4090"). Кожен з цих методів повертає самого себе, дозволяючи створювати ланцюжки викликів. Коли ви налаштували все, що хотіли, ви викликаєте фінальний метод build(), і Будівельник віддає вам готовий, сконструйований об'єкт "Комп'ютер". Це робить код набагато чистішим і дозволяє створювати різні конфігурації об'єктів, використовуючи той самий процес збірки.

Команда (Command)
Патерн Команда — це геніальний спосіб перетворити дію або запит на окремий об'єкт. Замість того, щоб один об'єкт напряму викликав метод іншого (editor.pasteText()), ви створюєте об'єкт-Команду, наприклад PasteCommand. Цей об'єкт "знає" все, що потрібно для виконання дії (наприклад, посилання на редактор і текст, який треба вставити). Найкраща аналогія — замовлення в

ресторані. Ви (клієнт) не йдете на кухню і не кажете кухарю, що робити. Ви даєте офіціанту "Команду" (замовлення на папірці). Офіціант ставить його в чергу. Кухар бере команди з черги і виконує їх. Це дає неймовірні переваги: ви можете ставити команди в чергу, відкладати їх виконання, зберігати історію команд і, найголовніше, — реалізувати функцію "Скасувати/Повторити" (Undo/Redo). Адже якщо кожна команда знає, як себе виконати (execute()), вона так само може знати, як скасувати свою дію (undo()).

Ланцюжок Обов'язків (Chain of Responsibility)
Цей патерн дозволяє уникнути жорсткої прив'язки відправника запиту до його одержувача. Ідея в тому, щоб дати можливість обробити запит кільком об'єктам. Ви вибудовуєте ці об'єкти в "ланцюжок". Коли надходить запит, він передається першому об'єкту в ланцюжку. Цей об'єкт дивиться на запит і вирішує: "Це моя робота? Можу я це обробити?". Якщо так, він його обробляє, і на цьому все. Якщо ні, він передає запит наступному об'єкту в ланцюжку, і так далі. Відправник запиту навіть не знає, хто саме в кінцевому підсумку обробить його запит. Це дуже схоже на роботу банкомата: ви просите 480 гривень. Перший обробник ("відповідальний" за 200-гривневі купюри) видає дві й передає залишок 80 гривень далі. Наступний (для 100) нічого не робить. Наступний (для 50) видає одну і передає 30 гривень. І так, доки запит не буде виконано повністю. Це часто використовується у веб-серверах для фільтрів: перевірка автентифікації, потім логування, потім кешування — кожен є ланкою в ланцюжку.

Прототип (Prototype)
Патерн Прототип вирішує проблему створення об'єктів, коли їх "будівництво" з нуля є дуже дорогим або складним процесом. Замість того, щоб щоразу створювати об'єкт через new, цей патерн пропонує створити новий об'єкт шляхом копіювання (клонування) вже існуючого. Уявіть, що у вас є складний об'єкт "НалаштуванняГри", на створення якого ви витратили 5 секунд, завантаживши дані з файлів та бази даних. Якщо вам потрібен точно такий же об'єкт, але з однією маленькою зміною, набагато "дешевше" створити повну

копію (клон) першого об'єкта за мілісекунди і змінити лише одне поле, ніж знову 5 секунд створювати його "з нуля". Ви створюєте один об'єкт-зразок, або "прототип", а всі наступні об'єкти такого типу отримуєте, просто викликаючи у прототипа метод clone(). Це особливо корисно, коли вам потрібно створити багато однакових або майже однакових об'єктів.

Тема проєкту

26. Download manager (iterator, command, observer, template method, composite, p2p)
Інструмент для скачування файлів з інтернету по протоколах http або https з можливістю продовження завантаження в зупиненому місці, розподілу швидкостей активним завантаженням, ведення статистики завантажень, інтеграції в основні браузери (firefox, opera, internet explorer, chrome).

Паттерн: Command
Посилання на GitHub: https://github.com/fl1ckyexe/trpz

Хід роботи

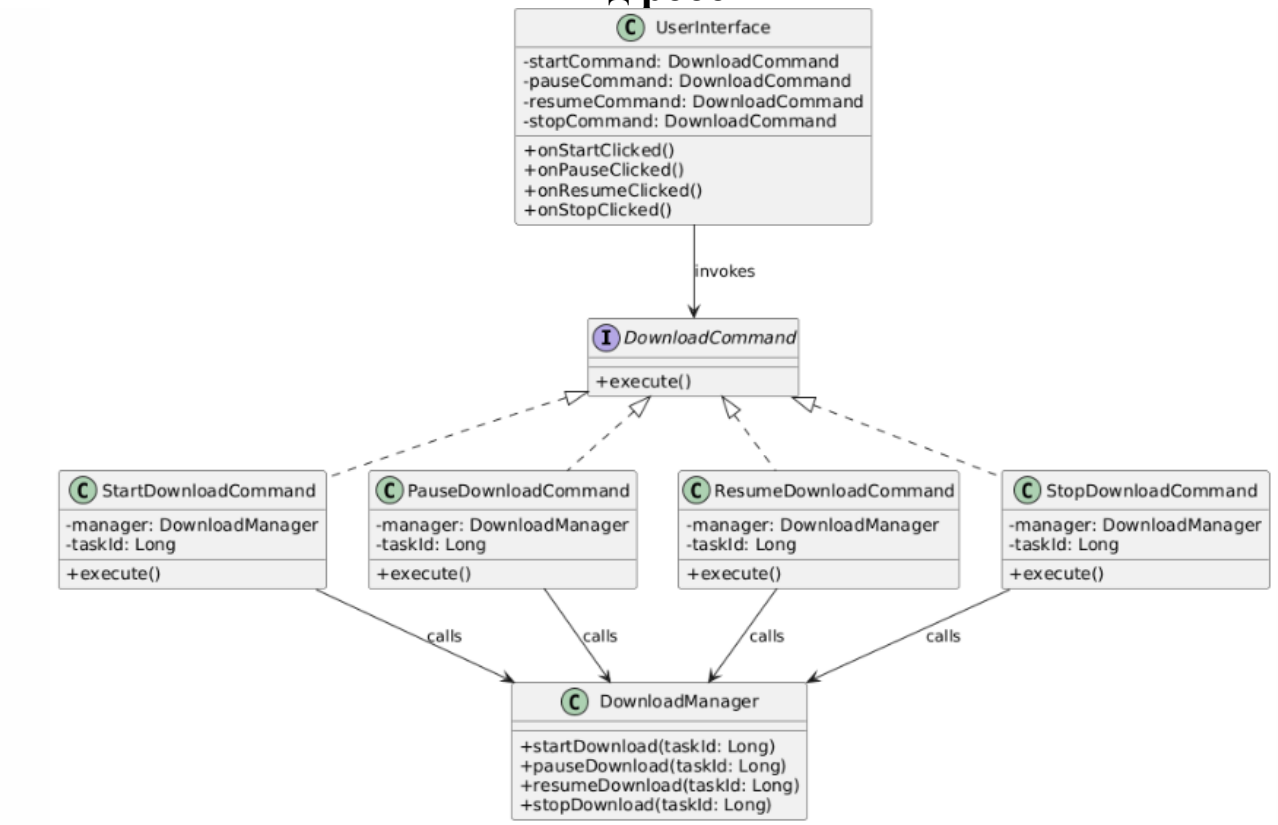


Рис. 1 – Діаграма класів для Command

Діаграма класів відображає використання шаблону проєктування **Command** у

системі десктопного менеджера завантажень файлів. Шаблон застосовується для інкапсуляції дій користувача у вигляді окремих об’єктів-команд, що дозволяє відокремити користувацький інтерфейс від логіки керування процесом завантаження.

Основні елементи діаграми

DownloadCommand — інтерфейс команди, який визначає спільний метод execute() для виконання дії.
StartDownloadCommand, **PauseDownloadCommand**, **ResumeDownloadCommand**, **StopDownloadCommand** — конкретні реалізації команд, що інкапсулюють відповідні дії керування завантаженням.
DownloadManager — клас-одержувач (Receiver), який містить реальну бізнес-логіку запуску, зупинки, паузи та відновлення завантажень.
UserInterface — клас-ініціатор (Invoker), що реагує на дії користувача та викликає відповідні команди.

Зв’язки та взаємодія між елементами

Інтерфейс **DownloadCommand** реалізується всіма конкретними командами. Кожна конкретна команда містить посилання на об’єкт **DownloadManager** та викликає відповідний метод керування завантаженням. Клас **UserInterface** зберігає посилання на об’єкти команд і ініціює їх виконання у відповідь на дії користувача, не взаємодіючи безпосередньо з **DownloadManager**. Таким чином, керування процесом завантаження здійснюється через команди, а не прямі виклики методів менеджера.

Переваги використання шаблону

Використання шаблону Command дозволяє зменшити зв’язність між користувацьким інтерфейсом та бізнес-логікою системи, спрощує додавання нових дій керування без зміни існуючого коду, а також створює основу для реалізації додаткових можливостей, таких як журналювання команд, відміна дій або їх повторне виконання.

Частина коду програми з Command

```
package command;

public interface DownloadCommand {
   void execute();
}
```

```
package command;

import core.DownloadManager;
```

```
public class StartDownloadCommand implements DownloadCommand {

    private DownloadManager manager;
    private Long taskId;

    public StartDownloadCommand(DownloadManager manager, Long taskId) {
        this.manager = manager;
        this.taskId = taskId;
    }

    @Override
    public void execute() {
        manager.startDownload(taskId);
    }
}
```

```
package command;

import core.DownloadManager;

public class PauseDownloadCommand implements DownloadCommand {

    private DownloadManager manager;
    private Long taskId;

    public PauseDownloadCommand(DownloadManager manager, Long taskId) {
        this.manager = manager;
        this.taskId = taskId;
    }

    @Override
    public void execute() {
        manager.pauseDownload(taskId);
    }
}
```

```
package command;

import core.DownloadManager;

public class ResumeDownloadCommand implements DownloadCommand {

    private DownloadManager manager;
    private Long taskId;

    public ResumeDownloadCommand(DownloadManager manager, Long taskId) {
```

```
        this.manager = manager;
        this.taskId = taskId;
    }

    @Override
    public void execute() {
        manager.resumeDownload(taskId);
    }
}


package command;

import core.DownloadManager;

public class StopDownloadCommand implements DownloadCommand {

    private DownloadManager manager;
    private Long taskId;

    public StopDownloadCommand(DownloadManager manager, Long taskId) {
        this.manager = manager;
        this.taskId = taskId;
    }

    @Override
    public void execute() {
        manager.stopDownload(taskId);
    }
}


package core;

public class DownloadManager {

    public void startDownload(Long taskId) {
        System.out.println("Start download: task " + taskId);
        // запуск сегментного завантаження
    }

    public void pauseDownload(Long taskId) {
        System.out.println("Pause download: task " + taskId);
        // пауза завантаження
    }

    public void resumeDownload(Long taskId) {
        System.out.println("Resume download: task " + taskId);
        // відновлення завантаження
```

```
    }

    public void stopDownload(Long taskId) {
        System.out.println("Stop download: task " + taskId);
        // зупинка завантаження
    }
}



package ui;

import command.*;
import core.DownloadManager;

public class UserInterface {

    private DownloadCommand startCommand;
    private DownloadCommand pauseCommand;
    private DownloadCommand resumeCommand;
    private DownloadCommand stopCommand;

    public UserInterface(DownloadManager manager, Long taskId) {
        this.startCommand = new StartDownloadCommand(manager, taskId);
        this.pauseCommand = new PauseDownloadCommand(manager, taskId);
        this.resumeCommand = new ResumeDownloadCommand(manager, taskId);
        this.stopCommand = new StopDownloadCommand(manager, taskId);
    }

    public void clickStart() {
        startCommand.execute();
    }

    public void clickPause() {
        pauseCommand.execute();
    }

    public void clickResume() {
        resumeCommand.execute();
    }

    public void clickStop() {
        stopCommand.execute();
    }
}


package ui;
```

```
import core.DownloadManager;

public class Main {

    public static void main(String[] args) {

        DownloadManager manager = new DownloadManager();
        UserInterface ui = new UserInterface(manager, 1L);

        ui.clickStart();
        ui.clickPause();
        ui.clickResume();
        ui.clickStop();
    }
}
```

Опис програмного коду

Програмний код реалізує фрагмент десктопного менеджера завантажень файлів з використанням шаблону проєктування Command. Архітектура побудована таким чином, щоб відокремити користувацький інтерфейс від логіки керування процесом завантаження.

Інтерфейс DownloadCommand визначає єдиний метод execute(), який представляє дію користувача. Конкретні класи команд (StartDownloadCommand, PauseDownloadCommand, ResumeDownloadCommand, StopDownloadCommand) інкапсулюють відповідні операції керування завантаженням та містять посилання на об’єкт DownloadManager, який виконує реальні дії.

Клас DownloadManager реалізує бізнес-логіку запуску, зупинки, паузи та відновлення завантаження файлів. Він виступає у ролі одержувача команд і не залежить від конкретного способу виклику цих операцій.

Клас UserInterface виконує роль ініціатора команд. Він реагує на дії користувача та викликає відповідні об’єкти команд, не звертаючись безпосередньо до DownloadManager. Це забезпечує слабке зв’язування між інтерфейсом користувача та логікою системи.

Таким чином, програмний код демонструє практичне застосування шаблону Command для організації керування діями користувача у менеджері завантажень та створює основу для подальшого розширення функціональності системи.

Висновок

У ході виконання даної лабораторної роботи було реалізовано шаблон проєктування Command у системі десктопного менеджера завантажень файлів. Шаблон було застосовано для інкапсуляції дій користувача з керування процесом завантаження у вигляді окремих об’єктів-команд.

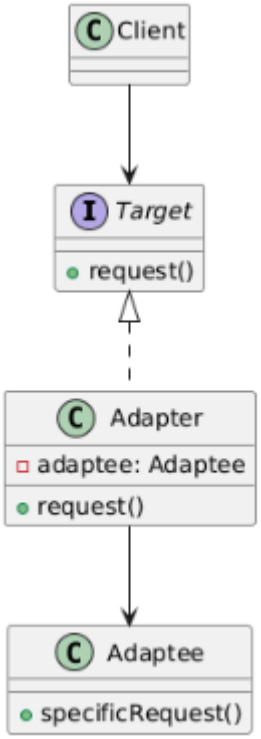
Було виділено інтерфейс команди та реалізовано конкретні команди для запуску, паузи, відновлення та зупинки завантаження. Користувацький інтерфейс взаємодіє з логікою завантаження виключно через об’єкти команд, не звертаючись безпосередньо до менеджера завантажень.

Застосування шаблону Command дозволило зменшити зв’язність між інтерфейсом користувача та бізнес-логікою, спростило розширення системи новими діями та створило основу для подальшої реалізації складніших механізмів керування завантаженнями.

Відповіді на контрольні питання

1. Яке призначення шаблону «Адаптер»?
Шаблон проєктування «Адаптер» призначений для узгодження інтерфейсів несумісних класів. Він дозволяє об’єктам з різними інтерфейсами працювати разом шляхом перетворення інтерфейсу одного класу у формат, очікуваний клієнтом.

2. Нарисуйте структуру шаблону «Адаптер».



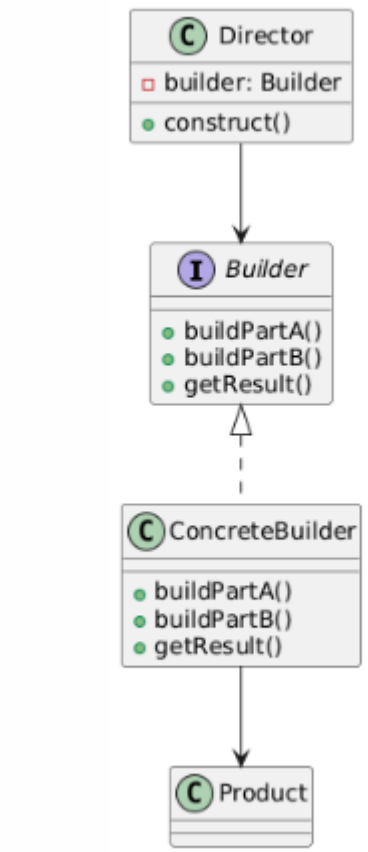
3. Які класи входять в шаблон «Адаптер», та яка між ними взаємодія?
У шаблоні «Адаптер» беруть участь класи Target, Adaptee, Adapter та Client. Клієнт працює з інтерфейсом Target. Adapter реалізує інтерфейс Target і всередині використовує об’єкт Adaptee, делегуючи йому виклики у відповідному форматі.

4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?
Адаптер на рівні об’єктів використовує композицію та містить посилання на об’єкт Adaptee, що забезпечує більшу гнучкість. Адаптер на рівні класів

використовує наслідування і напряму розширює Adaptee, але обмежується можливостями множинного наслідування.

5. Яке призначення шаблону «Будівельник»?
Шаблон «Будівельник» призначений для поетапного створення складних об’єктів, дозволяючи відокремити процес побудови об’єкта від його представлення.

6. Нарисуйте структуру шаблону «Будівельник».

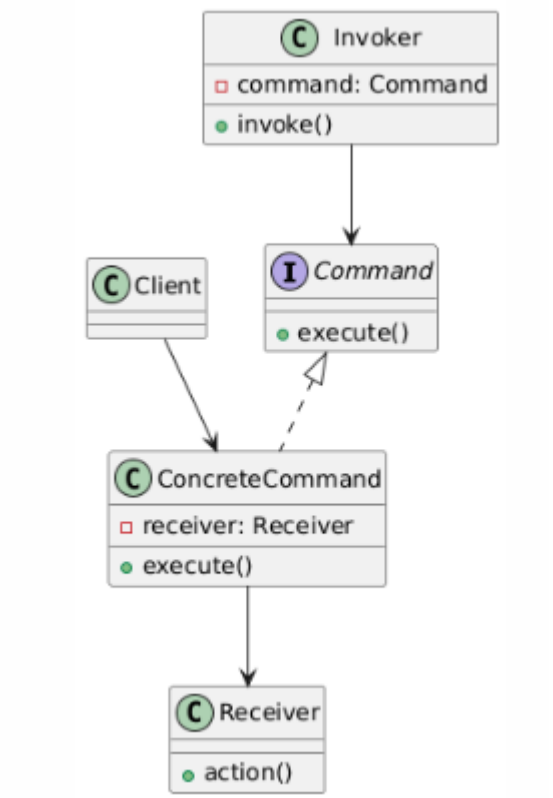


7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?
У шаблоні «Будівельник» використовуються класи Builder, ConcreteBuilder, Director та Product. Director керує послідовністю побудови об’єкта, викликаючи методи Builder. ConcreteBuilder реалізує побудову частин об’єкта та формує кінцевий Product.

8. У яких випадках варто застосовувати шаблон «Будівельник»?
Шаблон «Будівельник» доцільно застосовувати, коли об’єкт має складну структуру, багато необов’язкових параметрів або різні способи створення, а також коли необхідно відокремити алгоритм створення об’єкта від його кінцевого представлення.

9. Яке призначення шаблону «Команда»?
Шаблон «Команда» призначений для інкапсуляції запиту у вигляді об’єкта, що дозволяє параметризувати клієнтів різними запитами, підтримувати черги команд, журналювання та скасування операцій.

10. Нарисуйте структуру шаблону «Команда».

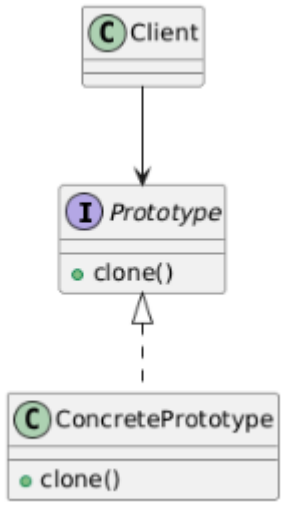


11. Які класи входять в шаблон «Команда», та яка між ними взаємодія?
У шаблоні «Команда» беруть участь Command, ConcreteCommand, Receiver, Invoker та Client. Client створює команду, Invoker викликає її виконання, ConcreteCommand інкапсулює запит і делегує виконання об’єкту Receiver.

12. Розкажіть як працює шаблон «Команда».
Клієнт створює об’єкт команди та передає його ініціатору. Ініціатор викликає метод execute, не знаючи деталей реалізації дії. Команда виконує запит, звертаючись до Receiver, який виконує фактичну операцію.

13. Яке призначення шаблону «Прототип»?
Шаблон «Прототип» призначений для створення нових об’єктів шляхом копіювання існуючих екземплярів, що дозволяє уникнути складного процесу ініціалізації.

14. Нарисуйте структуру шаблону «Прототип».



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

У шаблоні «Прототип» використовуються інтерфейс Prototype, який оголошує метод clone, та ConcretePrototype, який реалізує копіювання об’єкта. Client створює нові об’єкти шляхом виклику clone без використання оператора new.

16. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?
Прикладами використання шаблону «Ланцюжок відповідальності» є обробка подій у графічних інтерфейсах, фільтрація HTTP-запитів, системи логування з різними рівнями повідомлень, а також механізми авторизації та валідації даних.