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 «Теорія розробки програмного забезпечення»

Тема: «Патерни проектування»

Виконав:
Студент групи ІА-34
Цап Андрій

Перевірив:
Мягкий Михайло Юрійович

Київ – 2025

Зміст

Вступ

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

У межах розробленої системи управління проєктами був виконаний аналіз функціональних підсистем, зокрема модуля управління задачами. Цей модуль містить різні бізнес-правила, які застосовуються під час створення, редагування чи оновлення задач. Враховуючи необхідність послідовної перевірки даних та можливість подальшого розширення логіки, найбільш доцільним для інтеграції патерном став **Chain of Responsibility**.

Теоретичні відомості

Шаблон «Adapter» - призначення патерну: Шаблон "Adapter" (Адаптер)

використовується для адаптації інтерфейсу одного об'єкту до іншого. 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.

Переваги та недоліки:

+ Відокремлює інтерфейс або код перетворення даних від основної бізнеслогіки.

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

- Умовним недоліком можна назвати збільшення кількості класів

Шаблон «Builder» - призначення патерну: Шаблон «Builder»

(Будівельник) використовується для відділення процесу створення об'єкту від його представлення. Це доречно у випадках, коли об'єкт має складний процес створення (наприклад, Webсторінка як елемент повної відповіді web- сервера) або коли об'єкт повинен мати декілька різних форм створення.

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.

Шаблон «Command» - призначення патерну: Шаблон "command" (команда) перетворить звичайний виклик методу в клас. Таким чином дії в системі стають повноправними об'єктами.

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.

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.

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.

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.

Переваги та недоліки:

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

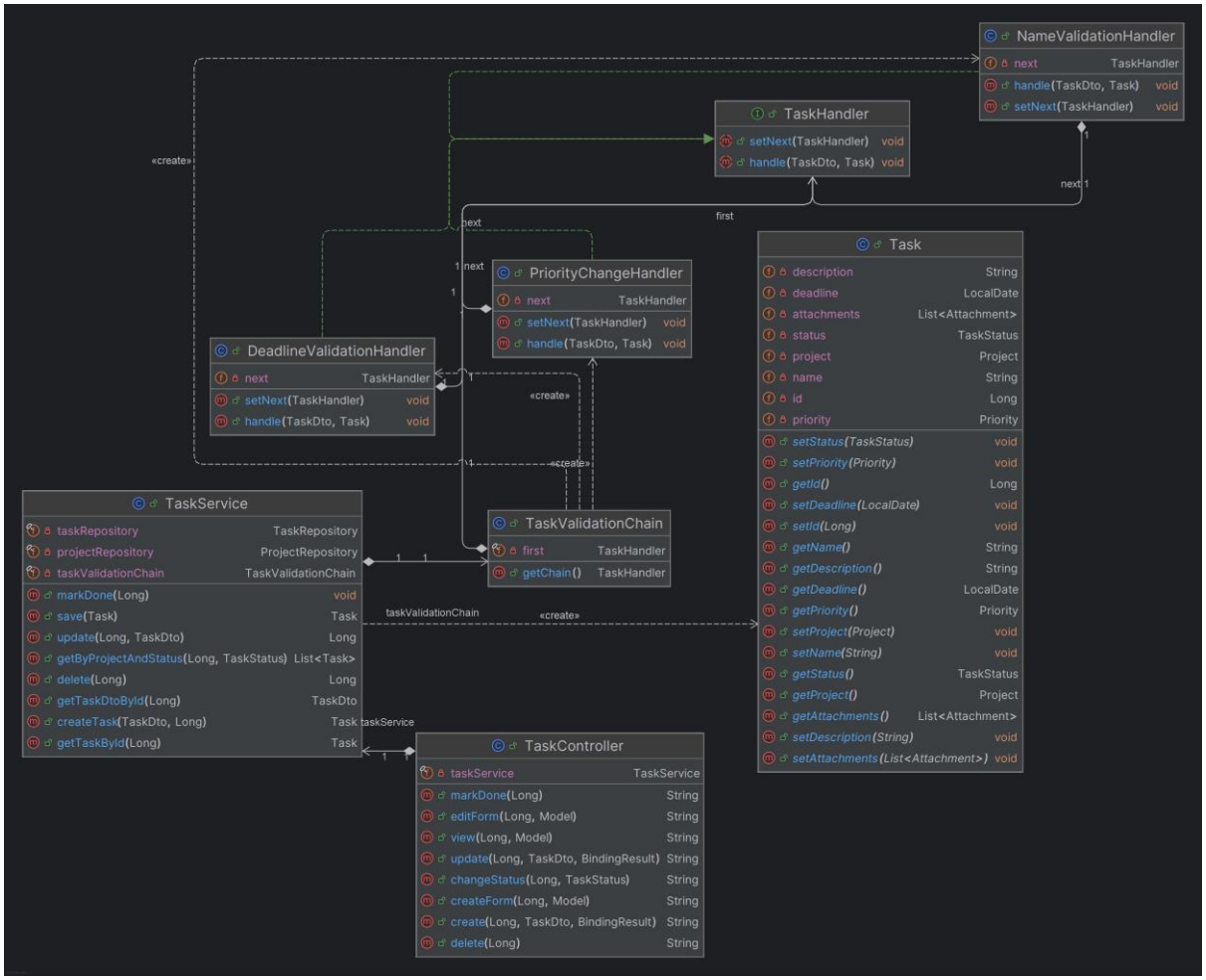
+ Різні варіації об’єктів можна отримувати за рахунок клонування, а не розширення ієрархії класів.

+ 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.

Хід роботи

Діаграма класів для реалізації шаблону Chain of Responsibility:



1. Task — доменна модель задачі

Task є основною сутністю модуля управління задачами. Містить ключові поля: назву, опис, дедлайн, пріоритет, статус, а також прив’язку до проєкту.

Саме над Task відбуваються всі операції редагування, зміни статусів, пріоритетів та перевірок бізнес-логіки.

Цей клас використовується як базовий об’єкт, який передається в ланцюг обробників для валідації.

2. TaskHandler — базовий елемент ланцюга

TaskHandler є абстрактним класом, який визначає структуру для всіх обробників у ланцюгу.
Він містить методи:

- **setNext()** — встановлення наступного елемента ланцюга;
- **handle()** — метод, що виконує перевірку та передає керування далі.

Саме TaskHandler дозволяє побудувати ланцюг, де кожен обробник відповідає за окрему дію або перевірку.

3. NameValidationHandler — обробник перевірки назви

Цей обробник відповідає за базову перевірку даних задачі:
чи коректна назва, чи не порожнє поле, чи не складається воно з пробілів.
Якщо дані некоректні — обробник зупиняє ланцюг і викликає помилку.
Якщо все добре — передає керування наступному хендлеру.

4. DeadlineValidationHandler — перевірка дедлайну

Цей обробник відповідає за логіку, пов’язану з датою завершення задачі:

- дедлайн не може бути в минулому,
- дедлайн повинен відповідати загальним правилам планування.

При успішній перевірці обробник передає керування далі.

5. PriorityChangeHandler — обробник зміни пріоритету

Цей елемент реалізує логіку контролю змін пріоритетів задачі.
У проєкт додано обмеження: пріоритет не можна знижувати, що дозволяє захистити систему від небажаних змін і підтримувати ієрархію задач.

Поведінка обробника:

- якщо новий пріоритет нижчий за старий — викликається помилка;
- якщо пріоритет однаковий або збільшується — ланцюг продовжує роботу.

6. TaskValidationChain — збирач ланцюга

Цей клас створює та конфігурує увесь ланцюг обробників.
Він визначає порядок:

1. NameValidationHandler
2. DeadlineValidationHandler
3. PriorityChangeHandler

Його завдання — надати сервісу готовий об’єкт ланцюга.

7. TaskService — сервіс бізнес-логіки задач

TaskService викликає весь ланцюг перед збереженням або оновленням задачі:

- отримує TaskDto
- завантажує існуючу задачу
- викликає **taskValidationChain.getChain().handle(dto, task)**
- і тільки після успішної валідації оновлює дані

Таким чином сервіс не містить жодних умовних перевірок — вони винесені в обробники.

8. TaskController — контролер представлення

Контролер отримує дані від UI, передає їх у бізнес-логіку та працює з TaskService.
Опосередковано він користується шаблоном Chain of Responsibility.

Переваги використання

Повне відокремлення перевірок від бізнес-логіки

TaskService не займається:

- перевіркою дедлайну
- перевіркою пріоритету
- перевіркою назви

Усе це винесено в окремі класи.

Гнучкість у зміні та розширенні валідацій

Щоб додати нову перевірку:

- створюється новий клас-хендлер
- додається в TaskValidationChain

Жодних змін у TaskService — відкритість до розширення, закритість до модифікації.

Заміна порядку перевірок без модифікації логіки

Можна легко змінити:

- порядок
- логіку
- кількість елементів

Це робиться виключно в TaskValidationChain.

Недоліки використання

Збільшення кількості класів

Для кожної перевірки створюється окремий клас.

Складніше зрозуміти для новачка

Коли є ланцюг з 4–5 хендлерів, поки не відкриєш діаграму, важко зрозуміти послідовність викликів.

Вихідні коди реалізації шаблону Chain of Responsibility

TaskValidationChain.java

```java
package org.example.projectmanagementsoftware.taskChain.config;

import org.example.projectmanagementsoftware.taskChain.handler.DeadlineValidationHandler;
import org.example.projectmanagementsoftware.taskChain.handler.NameValidationHandler;
import org.example.projectmanagementsoftware.taskChain.handler.PriorityChangeHandler;
import org.example.projectmanagementsoftware.taskChain.handler.TaskHandler;
import org.springframework.stereotype.Component;

@Component 2 usages new *
public class TaskValidationChain {

    private final TaskHandler first; 2 usages

    public TaskValidationChain() { new *

        TaskHandler name = new NameValidationHandler ();
        TaskHandler deadline = new DeadlineValidationHandler ();
        TaskHandler status = new PriorityChangeHandler ();

        name.setNext(deadline);
        deadline.setNext(status);

        this.first = name;
    }

    public TaskHandler getChain() { 2 usages new *
        return first;
    }
}
```

Рис 1 - Код TaskValidationChain

TaskHandler.java

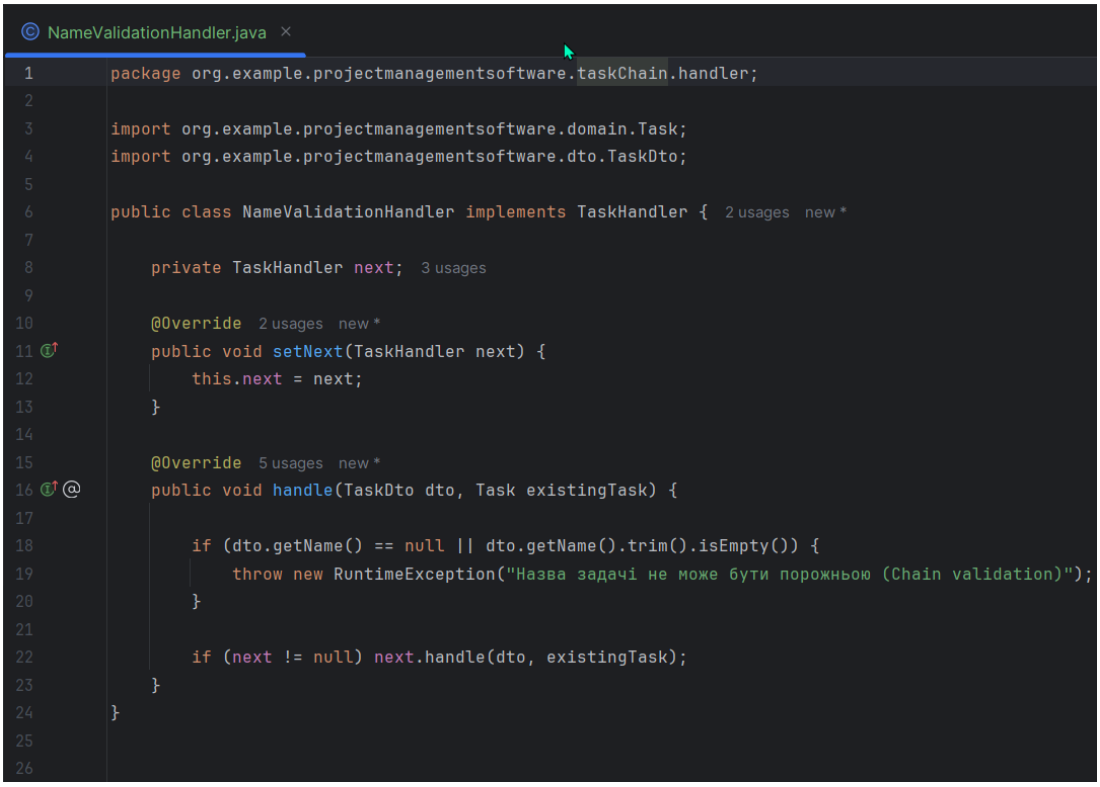
```java
package org.example.projectmanagementsoftware.taskChain.handler;

import org.example.projectmanagementsoftware.domain.Task;
import org.example.projectmanagementsoftware.dto.TaskDto;

public interface TaskHandler { 16 usages 3 implementations new *

    void setNext(TaskHandler next); 2 usages 3 implementations new *

    void handle(TaskDto dto, Task existingTask); 5 usages 3 implementations
}
```

Рис 2 - Код TaskHandler

PriorityChangeHandler.java

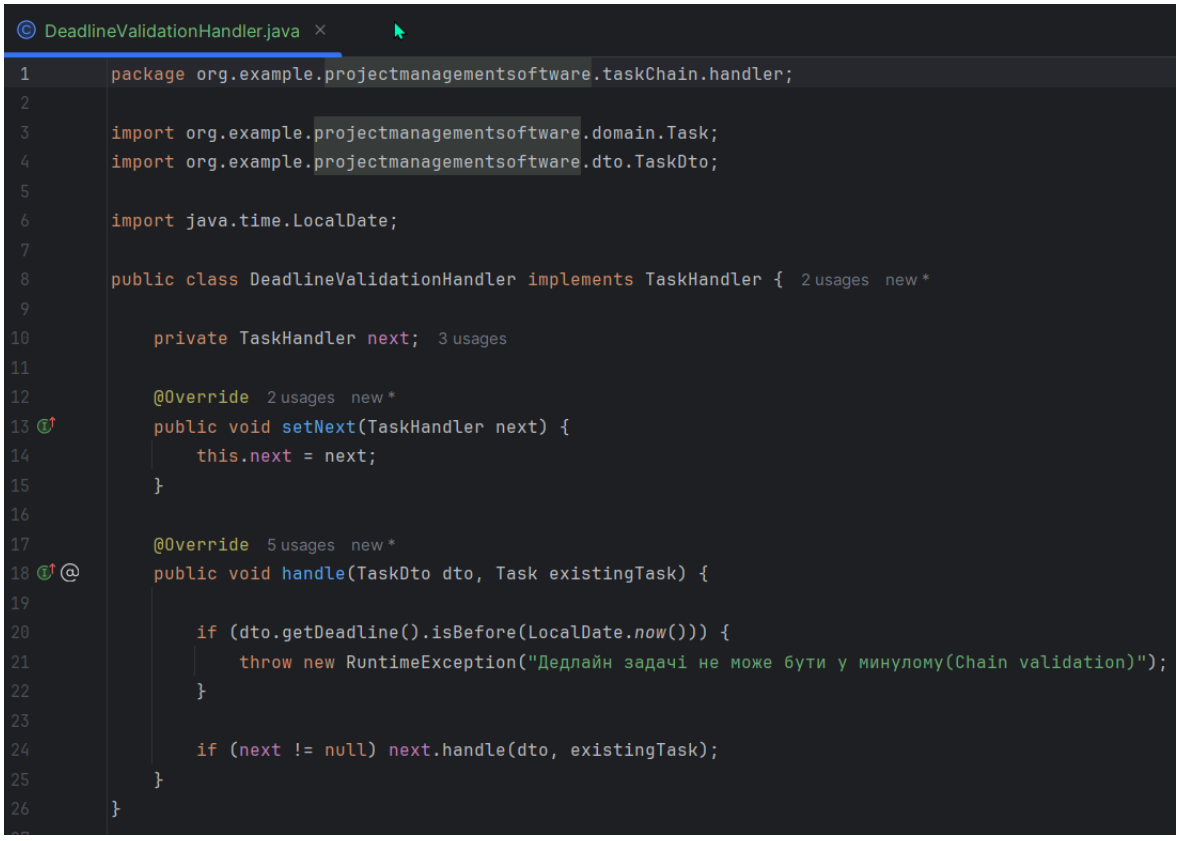
```java
package org.example.projectmanagementsoftware.taskChain.handler;

import org.example.projectmanagementsoftware.domain.Task;
import org.example.projectmanagementsoftware.domain.enums.Priority;
import org.example.projectmanagementsoftware.dto.TaskDto;

public class PriorityChangeHandler implements TaskHandler {  2 usages  new *

    private TaskHandler next;  3 usages

    @Override  2 usages  new *
    public void setNext(TaskHandler next) {
        this.next = next;
    }

    @Override  5 usages  new *
    public void handle(TaskDto dto, Task existingTask) {

        if (existingTask != null) {
            Priority oldP = existingTask.getPriority();
            Priority newP = dto.getPriority();

            if (newP.ordinal() < oldP.ordinal()) {
                throw new RuntimeException("Зниження пріоритету заборонено: "
                        + oldP + " → " + newP);
            }
        }

        if (next != null) next.handle(dto, existingTask);
    }
}
```

Рис 3 - Код PriorityChangeHandler



```java
package org.example.projectmanagementsoftware.taskChain.handler;

import org.example.projectmanagementsoftware.domain.Task;
import org.example.projectmanagementsoftware.dto.TaskDto;

public class NameValidationHandler implements TaskHandler {  2 usages  new *

    private TaskHandler next;  3 usages

    @Override  2 usages  new *
    public void setNext(TaskHandler next) {
        this.next = next;
    }

    @Override  5 usages  new *
    public void handle(TaskDto dto, Task existingTask) {

        if (dto.getName() == null || dto.getName().trim().isEmpty()) {
            throw new RuntimeException("Назва задачі не може бути порожньою (Chain validation)");
        }

        if (next != null) next.handle(dto, existingTask);
    }
}
```

Рис 4 - Код NameValidationHandler



```java
package org.example.projectmanagementsoftware.taskChain.handler;

import org.example.projectmanagementsoftware.domain.Task;
import org.example.projectmanagementsoftware.dto.TaskDto;

import java.time.LocalDate;

public class DeadlineValidationHandler implements TaskHandler {  2 usages  new *

    private TaskHandler next;  3 usages

    @Override  2 usages  new *
    public void setNext(TaskHandler next) {
        this.next = next;
    }

    @Override  5 usages  new *
    public void handle(TaskDto dto, Task existingTask) {

        if (dto.getDeadline().isBefore(LocalDate.now())) {
            throw new RuntimeException("Дедлайн задачі не може бути у минулому(Chain validation)");
        }

        if (next != null) next.handle(dto, existingTask);
    }
}
```

Рис 5 - Код DeadlineValidationHandler

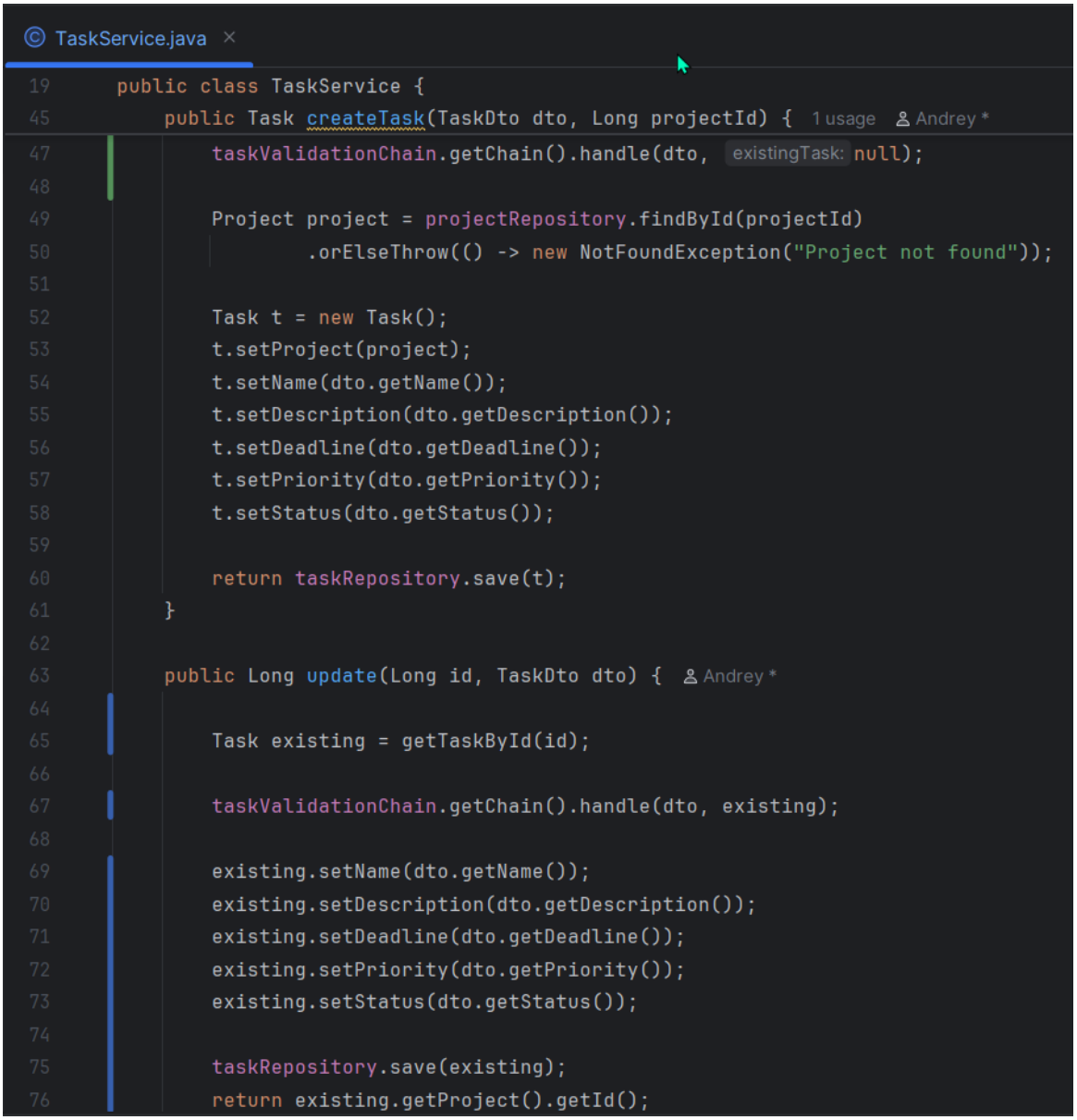


Рис 6 - Змінені методи з класу TaskService

Висновки

Під час виконання лабораторної роботи були розглянуті шаблони Adapter, Builder, Command, Chain of Responsibility та Prototype. Для інтеграції в існуючу архітектуру системи управління проєктами було обрано шаблон **Chain of Responsibility**, оскільки саме він найкраще відповідає потребам підсистеми задач.

У модулі Task існує послідовність бізнес-правил: валідність статусу, коректність пріоритету, допустимість зміни даних тощо. Застосування Chain of Responsibility дозволило розподілити ці перевірки на окремі класи, об'єднані в єдиний ланцюг.

Контрольні запитання

1. Яке призначення шаблону «Адаптер»?

Щоб зробити два несумісні інтерфейси сумісними. Він «обгортає» сторонній клас і дозволяє використовувати його так, ніби він підтримує потрібний інтерфейс.

2. Нарисуйте структуру шаблону «Адаптер».

Клієнт → Target (інтерфейс) → Adapter → Adaptee

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

- Client працює тільки з Target (потрібний інтерфейс).
- Adaptee має «чужий» інтерфейс.
- Adapter реалізує Target і всередині викликає методи Adaptee, переводячи одні виклики в інші.

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

- Об’єктний: адаптер містить посилання на Adaptee (композиція). Гнучкіший.
- Класовий: адаптер наслідує і Target, і Adaptee. Працює тільки коли дозволене множинне наслідування.

5. Яке призначення шаблону «Будівельник»?

Створювати складні об’єкти поетапно, контрольовано й без гігантських конструкторів з 15 параметрами.

6. Нарисуйте структуру шаблону «Будівельник».

Director → Builder (інтерфейс) → Concrete Builders → Product

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

- Director знає порядок побудови.
- Builder задає абстрактні кроки.
- ConcreteBuilder реалізує кроки створення конкретного продукту.
- Product – результат, який збирається частинами.

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

- Коли об’єкт має багато параметрів.
- Коли побудова має бути поетапною.
- Коли є різні варіації одного продукту.
- Коли не хочеш створювати монстр-конструктор.

9. Яке призначення шаблону «Команда»?

Перетворити запит у окремий об'єкт, щоб його можна було зберігати, ставити в черги, відміняти, логуати і передавати між компонентами.

10. Нарисуйте структуру шаблону «Команда».

Client → Command (інтерфейс) → Concrete Commands → Receiver
Invoker стоїть збоку і викликає команду.

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

- Command – інтерфейс з методом `execute()`.
- ConcreteCommand – реалізація дії, викликає методи Receiver.
- Receiver – той, хто реально виконує роботу.
- Invoker – той, хто "запускає" команду.
- Client створює команду та прив’язує її до виконавця.

12. Розкажіть як працює шаблон «Команда».

- Client створює об’єкт команди і передає в нього Receiver.
- Invoker зберігає посилання на команду.
- Коли треба виконати дію, Invoker викликає `command.execute()`.
- Команда всередині викликає потрібні методи Receiver.
  Все. Ти від’єднуєш запит від того, хто реально його виконує.

13. Яке призначення шаблону «Прототип»?

Створювати нові об’єкти шляхом копіювання вже існуючих, а не через конструктор. Зручно, коли цей конструктор важкий, або об’єкти сильно налаштовані.

14. Нарисуйте структуру шаблону «Прототип».

Client → Prototype (інтерфейс) → Concrete Prototypes (з Clone)

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

- Prototype – інтерфейс з методом `clone()`.
- ConcretePrototype – реалізує клонування.
- Client – просить створити копію без участі конструктора.

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

- Обробка HTTP middleware (кожен шар або обробляє, або передає далі).
- Логування різних рівнів: info → warning → error → critical.
- Обробка подій у GUI: кнопка → контейнер → вікно → система.
- Валідація форми: перевірка полів одна за одною.
- Системи підтримки: оператор 1 рівня, потім 2, потім 3.