

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 IA-34  
Цап Андрій

**Перевірив:**  
Мягкий Михайло Юрійович

Київ – 2025

## Зміст

Зміст .....	1
Вступ .....	2
Теоретичні відомості .....	3
Шаблон «Adapter» .....	4
Переваги та недоліки:.....	4
Шаблон «Builder» .....	4
Переваги та недоліки:.....	4
Шаблон «Command» .....	5
Переваги та недоліки:.....	5
Шаблон «Chain of Responsibility» .....	5
Переваги та недоліки:.....	5
Шаблон «Prototype» .....	6
Переваги та недоліки:.....	6
Хід роботи.....	6
Діаграма класів для реалізації шаблону Chain of Responsibility:.....	7
Переваги використання .....	9
Повне відокремлення перевірок від бізнес-логіки .....	10
Гнучкість у зміні та розширенні валідацій .....	10
Заміна порядку перевірок без модифікації логіки.....	10
Недоліки використання .....	10
Збільшення кількості класів .....	10
Складніше зрозуміти для новачка.....	10
Вихідні коди реалізації шаблону Chain of Responsibility .....	11
Висновки.....	16
Контрольні питання.....	17

## 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»**

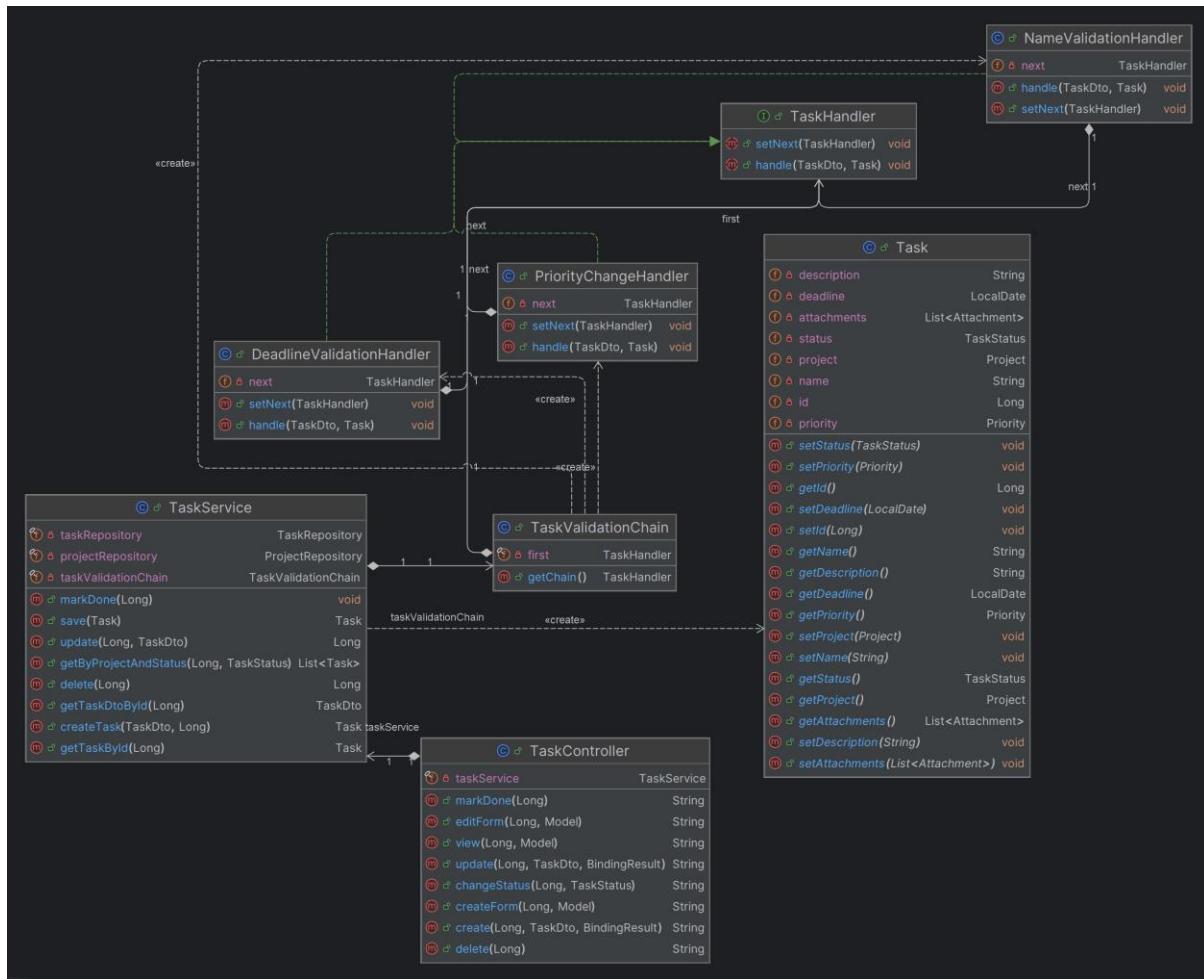
(Прототип) використовується для створення об'єктів за «шаблоном» (чи «кресленням», «ескізом») шляхом копіювання шаблонного об'єкту, який називається прототипом. Для цього визначається метод «клонувати» в об'єктах цього класу.

#### **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 проект 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 одинаковий 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

Дляожної 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

```

1 package org.example.projectmanagementsoftware.taskChain.config;
2
3 import org.example.projectmanagementsoftware.taskChain.handler.DeadlineValidationHandler;
4 import org.example.projectmanagementsoftware.taskChain.handler.NameValidationHandler;
5 import org.example.projectmanagementsoftware.taskChain.handler.PriorityChangeHandler;
6 import org.example.projectmanagementsoftware.taskChain.handler.TaskHandler;
7 import org.springframework.stereotype.Component;
8
9 @Component 2 usages new *
10 public class TaskValidationChain {
11
12     private final TaskHandler first; 2 usages
13
14     public TaskValidationChain() { new *
15
16         TaskHandler name = new NameValidationHandler ();
17         TaskHandler deadline = new DeadlineValidationHandler ();
18         TaskHandler status = new PriorityChangeHandler ();
19
20         name.setNext(deadline);
21         deadline.setNext(status);
22
23         this.first = name;
24     }
25
26     public TaskHandler getChain() { 2 usages new *
27         return first;
28     }
29 }

```

Рис 1 - Код TaskValidationChain

```

1 package org.example.projectmanagementsoftware.taskChain.handler;
2
3 import org.example.projectmanagementsoftware.domain.Task;
4 import org.example.projectmanagementsoftware.dto.TaskDto;
5
6 public interface TaskHandler { 16 usages 3 implementations new *
7
8     void setNext(TaskHandler next); 2 usages 3 implementations new *
9
10    void handle(TaskDto dto, Task existingTask); 5 usages 3 implementations new *
11
12 }

```

Рис 2 - Код TaskHandler

```
© PriorityChangeHandler.java ×

1 package org.example.projectmanagementsoftware.pattern.taskChain.handler;
2
3 import org.example.projectmanagementsoftware.domain.Task;
4 import org.example.projectmanagementsoftware.domain.enums.Priority;
5 import org.example.projectmanagementsoftware.dto.TaskDto;
6 import org.example.projectmanagementsoftware.exception.TaskValidationException;
7
8 public class PriorityChangeHandler implements TaskHandler { 2 usages & Andrey *
9
10     private TaskHandler next; 3 usages
11
12     @Override 2 usages & Andrey
13     public void setNext(TaskHandler next) { this.next = next; }
14
15     @Override 5 usages & Andrey *
16     public void handle(TaskDto dto, Task existingTask) {
17
18         if (existingTask != null) {
19             Priority oldP = existingTask.getPriority();
20             Priority newP = dto.getPriority();
21
22             if (oldP != null && newP != null && newP.ordinal() < oldP.ordinal()) {
23                 throw new TaskValidationException (
24                     "Зниження пріоритету заборонено: " + oldP + " → " + newP
25                 );
26             }
27         }
28
29         if (next != null) next.handle(dto, existingTask);
30     }
31 }
32
33 }
```

Рис 3 - Код PriorityChangeHandler

```
⑥ NameValidationHandler.java ×

1 package org.example.projectmanagementsoftware.pattern.taskChain.handler;
2
3 import org.example.projectmanagementsoftware.domain.Task;
4 import org.example.projectmanagementsoftware.dto.TaskDto;
5 import org.example.projectmanagementsoftware.exception.TaskValidationException;
6
7 public class NameValidationHandler implements TaskHandler { 2 usages new *
8
9     private TaskHandler next; 3 usages
10
11    @Override 2 usages new *
12    public void setNext(TaskHandler next) {
13        this.next = next;
14    }
15
16    @Override 5 usages new *
17    public void handle(TaskDto dto, Task existingTask) {
18
19        if (dto.getName() == null || dto.getName().trim().isEmpty()) {
20            throw new TaskValidationException("Назва задачі не може бути порожньою");
21        }
22
23        if (next != null) next.handle(dto, existingTask);
24    }
25 }
```

Рис 4 - Код NameValidationHandler

```
© DeadlineValidationHandler.java ×

1 package org.example.projectmanagementsoftware.pattern.taskChain.handler;
2
3 import org.example.projectmanagementsoftware.domain.Task;
4 import org.example.projectmanagementsoftware.dto.TaskDto;
5 import org.example.projectmanagementsoftware.exception.TaskValidationException;
6
7 import java.time.LocalDate;
8
9 public class DeadlineValidationHandler implements TaskHandler { 2 usages new *
10
11     private TaskHandler next; 3 usages
12
13     @Override 2 usages new *
14     public void setNext(TaskHandler next) {
15         this.next = next;
16     }
17
18     @Override 5 usages new *
19     public void handle(TaskDto dto, Task existingTask) {
20
21         if (dto.getDeadline() != null && dto.getDeadline().isBefore(LocalDate.now())) {
22             throw new TaskValidationException("Дедлайн задачі не може бути у минулому");
23         }
24
25         if (next != null) next.handle(dto, existingTask);
26     }
27 }
```

Рис 5 - Код DeadlineValidationHandler

```
© TaskService.java ×

19  public class TaskService {
45      public Task createTask(TaskDto dto, Long projectId) { 1 usage & Andrey *
47          taskValidationChain.getChain().handle(dto, existingTask: null);
48
49          Project project = projectRepository.findById(projectId)
50              .orElseThrow(() -> new NotFoundException("Project not found"));
51
52          Task t = new Task();
53          t.setProject(project);
54          t.setName(dto.getName());
55          t.setDescription(dto.getDescription());
56          t.setDeadline(dto.getDeadline());
57          t.setPriority(dto.getPriority());
58          t.setStatus(dto.getStatus());
59
60          return taskRepository.save(t);
61      }
62
63      public Long update(Long id, TaskDto dto) { & Andrey *
64
65          Task existing = getTaskById(id);
66
67          taskValidationChain.getChain().handle(dto, existing);
68
69          existing.setName(dto.getName());
70          existing.setDescription(dto.getDescription());
71          existing.setDeadline(dto.getDeadline());
72          existing.setPriority(dto.getPriority());
73          existing.setStatus(dto.getStatus());
74
75          taskRepository.save(existing);
76          return existing.getProject().getId();
```

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). Гнучкий.
- Класовий: адаптер наслідує і Target, і Adaptee. Працює тільки коли дозволене множинне наслідування.

### 5. Яке призначення шаблону «Будівельник»?

Створювати складні об'єкти поетапно, контролювано й без гіантських 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, логувати 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.