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

«Патерни проектування»

Виконала:
студентка групи ІА-32
Чайковська С. В.

Перевірив:
Мягкий М. Ю.

Київ 2025

Зміст

Тема: Патерни проектування.

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

1 Теоретичні відомості

Шаблон "Adapter" (Адаптер)

Призначення: Адаптер використовується для приведення інтерфейсу одного об’єкта до іншого, забезпечуючи сумісність між різними системами.

Проблема: Існує потреба адаптувати різні формати даних (наприклад, XML до JSON) без зміни вихідного коду.

Рішення: Створення адаптера, який «перекладає» інтерфейси одного об'єкта для іншого (наприклад, XML_To_JSON_Adapter).

Переваги: Приховує складність взаємодії різних інтерфейсів від користувача. Недоліки: Ускладнює код через додаткові класи.

Шаблон "Builder" (Будівельник)

Призначення: Шаблон відділяє процес створення об'єкта від його представлення, що зручно для складних або багатоформатних об'єктів. Проблема: Ускладнене створення об'єкта, наприклад, формування відповіді web-сервера з кількох частин (заголовки, статуси, вміст).

Рішення: Кожен етап створення об'єкта абстрагується в окремий метод будівельника, що дозволяє контролювати процес побудови.

Переваги: Забезпечує гнучкість та незалежність від внутрішніх змін у процесі створення.

Недоліки: Клієнт залежить від конкретних класів будівельників, що може обмежити можливості.

Шаблон "Command" (Команда)

Призначення: Перетворює виклик методу в об'єкт-команду, що дозволяє гнучко керувати діями (додавати, скасовувати, комбінувати команди).

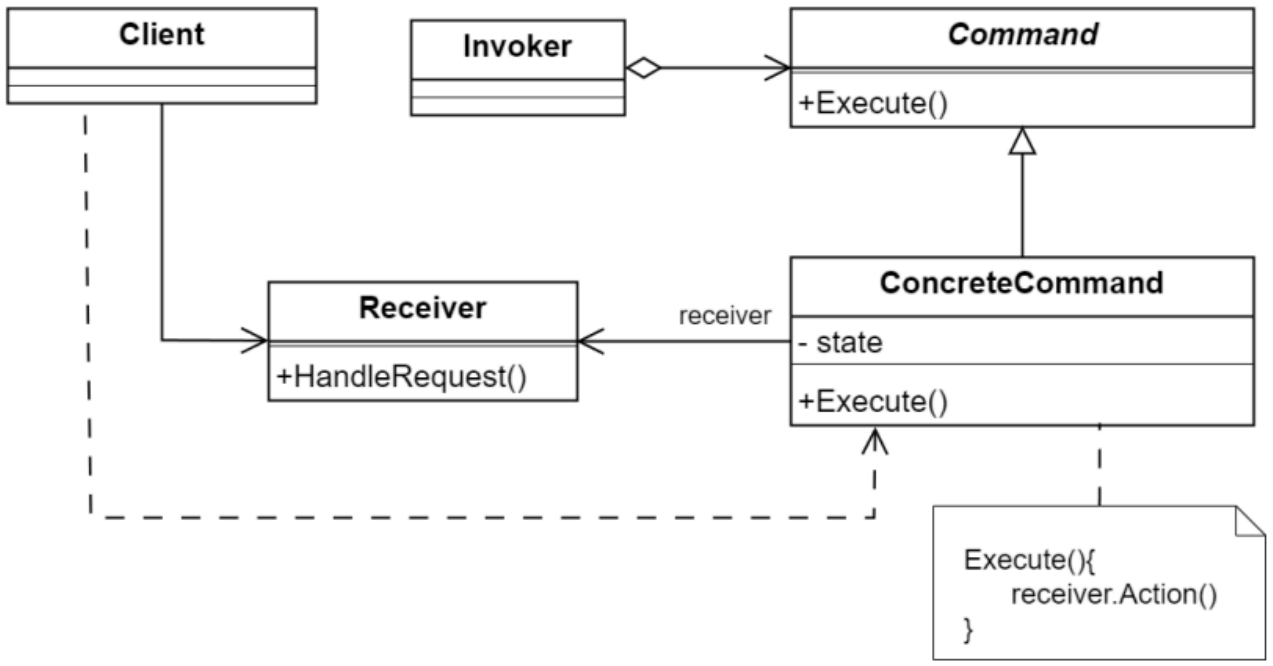


Рис 1. – Структура патерну «Command»

Проблема: Як організувати обробку кліків у текстовому редакторі без дублювання коду для різних кнопок.

Рішення: Створити окремий клас-команду для кожної дії, який буде викликати методи бізнес-логіки через об'єкт інтерфейсу.

Переваги:

- 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.

Недоліки: Ускладнює код додатковими класами.

Шаблон "Chain of Responsibility"

Призначення: Ланцюг відповідальності дозволяє організувати обробку запиту, передаючи його послідовно через ланцюг об'єктів, поки не буде знайдено обробник, здатний виконати запит. Це зручно для побудови гнучкої

системи обробників, коли важливо зменшити залежність клієнта від обробників і структурувати обробку.

Проблема: Припустимо, потрібно реалізувати систему обробки онлайн-замовлень, де доступ мають тільки авторизовані користувачі, а адміністратори мають додаткові права. Ці перевірки слід виконувати послідовно. Однак при додаванні нових перевірок код стає перевантаженим умовними операторами, що ускладнює підтримку.

Рішення: Створити для кожної перевірки окремий клас з методом, який виконує потрібні дії. Далі всі об'єкти-обробники об'єднуються в ланцюг, де кожен обробник має посилання на наступного. Запит передається першому обробнику ланцюга, який або самостійно обробляє його, або передає далі. Якщо обробник не може виконати дію, запит продовжує передаватися далі по ланцюгу, аж поки не знайдеться відповідний обробник або ланцюг закінчиться.

Переваги:

- Зменшує залежність між клієнтом і обробниками.
- Відповідає принципу єдиного обов’язку.
- Підтримує принцип відкритості/закритості.

Недоліки: Запит може залишитися без обробки, якщо жоден обробник його не обробить.

Шаблон "Prototype"

Призначення: Шаблон "Prototype" використовується для створення об'єктів шляхом копіювання існуючого шаблонного об'єкта. Це дозволяє спрощувати процес створення об'єктів, коли їх структура заздалегідь відома, та уникати прямого створення нових екземплярів.

Проблема: Якщо потрібно скопіювати об’єкт, звичайний спосіб — створити новий об'єкт і вручну копіювати його поля. Але деякі частини об’єкта можуть бути приватними, що ускладнює доступ до них і порушує інкапсуляцію.

Рішення: Шаблон "Prototype" доручає самим об’єктам реалізовувати метод clone(), який повертає їх копію. Це дозволяє створювати нові об'єкти без

прив'язки до їхніх конкретних класів, залишаючи логіку копіювання всередині класу.

Переваги:

- Дозволяє клонувати об’єкти без прив'язки до конкретного класу.
- Зменшує дублювання коду ініціалізації.
- Прискорює процес створення об'єктів.
- Є альтернативою підкласам при створенні складних об’єктів.

Недоліки: Складність клонування об'єктів, що містять посилання на інші об'єкти.

2 Тема

3. **Текстовий редактор** (strategy, command, observer, template method, flyweight, SOA)

Текстовий редактор повинен вміти розпізнавати текстові файли в будь-якій кодуванні, мати розширені функції редагування: макроси, сніппети, підказки, закладки, перехід на рядок / сторінку, підсвічування синтаксису (для однієї мови програмування або розмітки на розсуд студента).

3 Аргументація вибору патерну «Command»

Патерн «Command» було обрано для реалізації операцій з документами (відкриття, збереження, редагування, завантаження) з метою забезпечення гнучкого та масштабованого керування діями користувача. Його застосування дає можливість інкапсулювати кожну дію в окремий об’єкт-команду, що спрощує керування запитами та дозволяє додавати нові операції без зміни існуючої логіки контролера або сервісу.

Основні аргументи вибору патерну:

1. *Розділення відповідальностей.* Контролер не виконує бізнес-логіку напряму, а створює об’єкт команди та передає його інвокеру. Це забезпечує слабке зв’язування між рівнями системи.

2. *Масштабованість і розширюваність.* Додавання нової операції відбувається шляхом створення нової команди, без зміни існуючих класів.

3. *Єдиний інтерфейс виконання дій.* Усі команди реалізують однаковий метод execute(), що дозволяє однаково обробляти різні типи дій — відкриття, збереження, редагування тощо.

4. *Інкапсуляція запиту.* Кожна команда містить усю необхідну інформацію для виконання дії, що робить систему більш стійкою до змін у внутрішній логіці сервісів.

Таким чином, використання патерну «Command» дозволяє побудувати гнучку, розширювану та легко підтримувану архітектуру, у якій кожна операція з документом є незалежною, повторно використовуваною та контрольованою через єдиний механізм виконання.

4 Реалізація частини функціоналу робочої програми

Структура проекту зображена на Рисунку 1:

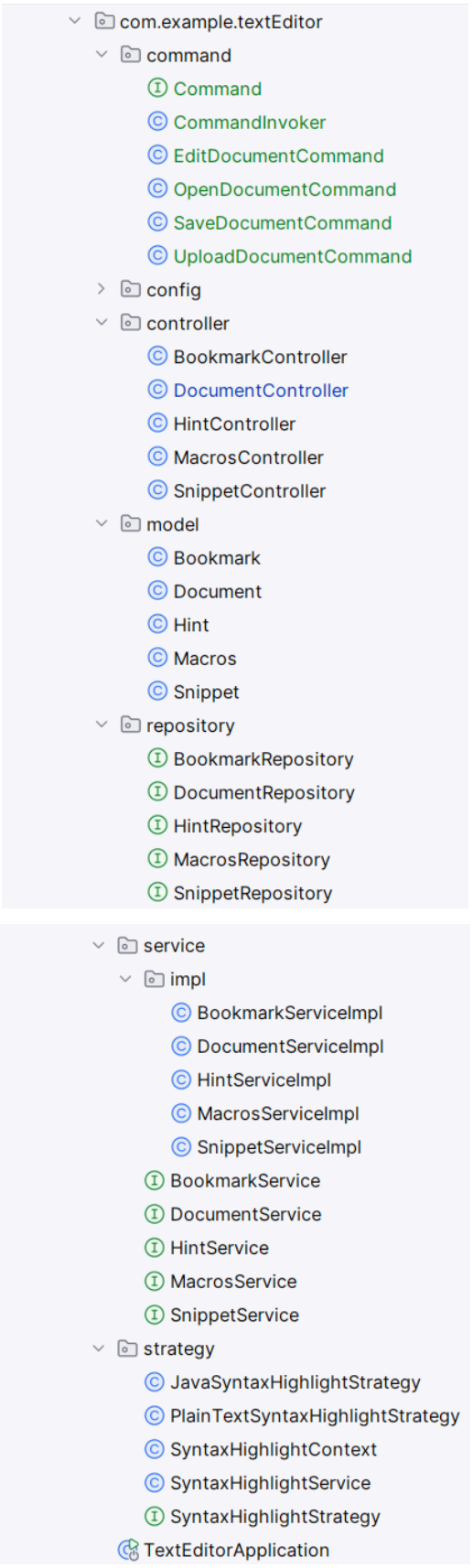


Рис. 2 – Структура проекту з використанням патерну «Command»

Структура проєкту TextEditor:

Архітектура побудована за принципом багаторівневого розподілу, що забезпечує чітке розділення відповідальностей:

- *command* – реалізація патерну «Command» для організації дій із документами через інкапсуляцію запитів у вигляді об’єктів. Забезпечує гнучке керування командами, підтримку історії дій. Основні класи:
  - *Command* – інтерфейс, що визначає метод *execute(),* який реалізується конкретними командами.
  - *OpenDocumentCommand, SaveDocumentCommand, UploadDocumentCommand, EditDocumentCommand* – конкретні команди, які інкапсулюють дії з відкриття, збереження, завантаження та редагування документів відповідно.
  - *CommandInvoker* – клас, що виконує команди, зберігає їх в історії та може реалізовувати скасування дій.
  - *DocumentServiceImpl* виступає отримувачем (receiver), який безпосередньо реалізує бізнес-логіку команд.
  - *DocumentController* є клієнтом, який створює конкретні команди та передає їх інвокеру для виконання.
- *controller* – містить контролери, які приймають запити від користувача, викликають бізнес-логіку та повертають відповіді на UI: *DocumentController, BookmarkController, HintController, MacrosController, SnippetController* – керують операціями з відповідними сутностями (документи, закладки, підказки, макроси, фрагменти коду).
- *model* – включає моделі даних (сутності), що відображають таблиці бази даних: *Document, Bookmark, Hint, Macros, Snippet*. Кожна з моделей описує структуру даних, які зберігаються у БД.

- *repository* – шар доступу до даних. Містить інтерфейси для роботи з БД: *DocumentRepository, BookmarkRepository, HintRepository, MacrosRepository, SnippetRepository*. Завдяки Spring Data JPA забезпечується збереження, пошук та вибірка даних без написання SQL-запитів вручну.
- *service* – шар бізнес-логіки. Інтерфейси: *DocumentService, BookmarkService, HintService, MacrosService, SnippetService.*
  Реалізації (impl): *DocumentServiceImpl, BookmarkServiceImpl, HintServiceImpl, MacrosServiceImpl, SnippetServiceImpl.* У цьому шарі виконується логіка обробки даних, яка відділяє контролери від репозиторіїв.
- *strategy* – реалізація патерну «Strategy» для підсвічування синтаксису. Містить інтерфейс, контекст і конкретні стратегії:
  - *SyntaxHighlightStrategy* – інтерфейс, що визначає метод *highlight(String text)* для різних алгоритмів підсвічування.
  - *JavaSyntaxHighlightStrategy* – конкретна стратегія для підсвічування синтаксису мови Java за допомогою регулярних виразів.
  - *PlainTextSyntaxHighlightStrategy* – стратегія, яка залишає текст без змін (використовується для випадків без підсвічування).
  - *SyntaxHighlightContext* – контекст, який зберігає поточну стратегію та викликає її методи.
  - *SyntaxHighlightService* – сервіс, який визначає, яку саме стратегію застосувати до тексту залежно від розширення файлу.
- *TextEditorApplication* – головний клас, що запускає Spring Boot застосунок.

Така структура проєкту дає змогу легко підтримувати й розширювати систему, оскільки контролери, бізнес-логіка та доступ до даних розділені по різних шарах.

5 Реалізація патерну «Command»

1. ***Інтерфейс Command*** представлений на Рисунку 3:

```
1   package com.example.textEditor.command;
2
3   public interface Command {  11 usages  4 implementations  new *
4       void execute();  1 usage  4 implementations  new *
5   }
```

Рис. 3 – Інтерфейс Command

Інтерфейс *Command*, наведений на Рис. 3, визначає спільний метод *execute()*, який має реалізовуватися в кожній конкретній команді. Він задає єдиний спосіб виконання дій для всіх команд, що описують операції в системі (створення, збереження, видалення або оновлення документа). Завдяки цьому кожна команда інкапсулює виклик операції над об’єктом-отримувачем (*receiver*), таким як *DocumentServiceImpl*, і може виконуватися незалежно від конкретної реалізації цієї дії.

2. ***Клас CommandInvoker*** представлений на Рисунку 4:

```
1   package com.example.textEditor.command;
2
3   import java.util.Stack;
4
5   public class CommandInvoker {  2 usages  new *
6       private Stack<Command> history = new Stack<>();  3 usages
7
8   @   public void executeCommand(Command command) {  5 usages  new *
9           command.execute();
10          history.push(command);
11      }
```

Рис. 4 – Клас SyntaxHighlightContext

Клас *CommandInvoker*, наведений на Рис. 4, виконує роль ініціатора (*Invoker*) у структурі патерна Command. Він зберігає історію виконаних команд у стеку (*history*) та відповідає за виклик їхнього методу *execute()*. Після виконання команда додається до історії. *CommandInvoker* лише керує процесом виконання, делегуючи виклик конкретному об’єкту-команді, який у свою чергу взаємодіє з відповідним receiver (*DocumentServiceImpl*).

3. ***Клас EditDocumentCommand*** представлений на Рисунку 5:

```java
package com.example.textEditor.command;

import com.example.textEditor.model.Document;
import com.example.textEditor.service.DocumentService;

public class EditDocumentCommand implements Command {  1 usage  new *
    private DocumentService documentService;  2 usages
    private Document document;  3 usages
    private String newContent;  2 usages

    public EditDocumentCommand(DocumentService documentService, Document document, String newContent) {
        this.documentService = documentService;
        this.document = document;
        this.newContent = newContent;
    }

    @Override  1 usage  new *
    public void execute() {
        document.setContent(newContent);
        documentService.update(document);
    }
}
```

Рис. 5 – Клас EditDocumentCommand

Клас *EditDocumentCommand*, який наведено на Рис. 5, реалізує команду редагування документа. Він інкапсулює зміну вмісту документа (*document.setContent(newContent)*) і виклик операції оновлення в об'єкті-отримувачі (*DocumentServiceImpl*). Завдяки цьому логіка редагування ізольована всередині команди, а виклик можна виконати через *CommandInvoker* незалежно від того, як саме працює *DocumentServiceImpl*.

4. ***Клас OpenDocumentCommand*** представлений на Рисунку 6:

```java
package com.example.textEditor.command;

import com.example.textEditor.model.Document;
import com.example.textEditor.service.DocumentService;

public class OpenDocumentCommand implements Command {  1 usage  new *
    private DocumentService documentService;  3 usages
    private int documentId;  2 usages

    public OpenDocumentCommand(DocumentService documentService, int documentId) {
        this.documentService = documentService;
        this.documentId = documentId;
    }

    @Override  1 usage  new *
    public void execute() {
        try {
            Document doc = documentService.getById(documentId);
```

```
19                System.out.println("Document opened: " + doc.getFilename());
20            } catch (Exception e) {
21                System.out.println("Document not found. Creating new...");
22                Document newDoc = new Document();
23                newDoc.setFilename("newfile.txt");
24                newDoc.setContent("");
25                documentService.create(newDoc);
26            }
27        }
28    }
```

Рис. 6 – Клас OpenDocumentCommand

Клас *OpenDocumentCommand*, який наведено на Рис. 6, реалізує команду відкриття документа. Він отримує документ за *id* через *DocumentServiceImpl*. Якщо документ не знайдено, команда створює новий документ і зберігає його. Команда інкапсулює весь процес відкриття документа, включно з перевіркою існування, і дозволяє викликати його через *CommandInvoker* незалежно від того, як реалізований сервіс.

5. ***Клас SaveDocumentCommand*** представлений на Рисунку 7:

```
1   package com.example.textEditor.command;
2
3   import com.example.textEditor.model.Document;
4   import com.example.textEditor.service.DocumentService;
5
6   public class SaveDocumentCommand implements Command {  2 usages  new *
7       private DocumentService documentService;  2 usages
8       private Document document;  2 usages
9
10      public SaveDocumentCommand(DocumentService documentService, Document document) {
11          this.documentService = documentService;
12          this.document = document;
13      }
14
15      @Override  1 usage  new *
16      public void execute() {
17          documentService.create(document);
18      }
19  }
```

Рис. 7 – Сервіс SaveDocumentCommand

Клас *SaveDocumentCommand*, який наведено на Рис. 7, реалізує команду збереження документа. Він інкапсулює операцію створення документа через *DocumentServiceImpl*. Використовуючи команду, можна зберегти документ без

прямого звернення до сервісу у зовнішньому коді. Це забезпечує гнучкість і дозволяє вести історію дій через *CommandInvoker*.

6. ***Клас UploadDocumentCommand*** представлений на Рисунку 8:

```java
package com.example.textEditor.command;

import com.example.textEditor.model.Document;
import com.example.textEditor.service.DocumentService;

public class UploadDocumentCommand implements Command {  1 usage  new *

    private final DocumentService documentService;  2 usages
    private final Document document;  2 usages

    public UploadDocumentCommand(DocumentService documentService, Document document) {
        this.documentService = documentService;
        this.document = document;
    }

    @Override  1 usage  new *
    public void execute() {
        documentService.create(document);
    }
}
```

Рис. 8 – Контролер UploadDocumentCommand

Клас *UploadDocumentCommand*, який наведено на Рис. 8, реалізує команду завантаження документа. Він приймає готовий документ і виконує його створення в базі через *DocumentServiceImpl*. Команда ізольовує логіку завантаження і роботу з сервісом, що дозволяє додавати цю операцію в історію команд і повторно виконувати її при необхідності.

7. ***Контролер DocumentController*** представлений на Рисунку 9:

```java
@RestController  Chaikovska Sofiia *
@RequestMapping("/file")
public class DocumentController {

    private final DocumentService documentService;  12 usages
    private final SyntaxHighlightService syntaxHighlightService;  2 usages
    private final CommandInvoker invoker = new CommandInvoker();  5 usages

    public DocumentController(DocumentService documentService, SyntaxHighlightService syntaxHighlightService) {
        this.documentService = documentService;
        this.syntaxHighlightService = syntaxHighlightService;
    }
```

```java

    @PostMapping("/save") new *
    public ResponseEntity<Document> save(@RequestBody Document document) {
        try {
            Document saved = documentService.create(document);

            Command command = new SaveDocumentCommand(documentService, saved);
            invoker.executeCommand(command);

            return ResponseEntity.ok(saved);
        } catch (Exception e) {
            e.printStackTrace();
            return ResponseEntity.internalServerError().build();
        }
    }

    @GetMapping("/open/{id}") new *
    public ResponseEntity<Document> open(@PathVariable int id) {
        try {
            Command command = new OpenDocumentCommand(documentService, id);
            invoker.executeCommand(command);

            Document doc = documentService.getById(id);
            return ResponseEntity.ok(doc);
        } catch (Exception e) {
            Document newDoc = new Document();
            newDoc.setFilename("newfile.txt");
            newDoc.setContent("");
            Document saved = documentService.create(newDoc);
```

```java

            Command command = new SaveDocumentCommand(documentService, saved);
            invoker.executeCommand(command);

            return ResponseEntity.ok(saved);
        }
    }

    @PutMapping("/edit/{id}") new *
    public ResponseEntity<Document> edit(@PathVariable int id, @RequestBody String content) {
        try {
            Document doc = documentService.getById(id);

            Command command = new EditDocumentCommand(documentService, doc, content);
            invoker.executeCommand(command);

            return ResponseEntity.ok(doc);
        } catch (Exception e) {
            return ResponseEntity.internalServerError().build();
        }
    }
```

```java
    @PostMapping("/upload") Chaikovska Sofiia *
    public ResponseEntity<Document> upload(@RequestParam("file") MultipartFile file) {
        try {
            String content = new String(file.getBytes(), java.nio.charset.StandardCharsets.UTF_8);
            String filename = file.getOriginalFilename();
            if (filename == null || filename.isBlank()) {
                return ResponseEntity.badRequest().build();
            }

            Document doc = new Document();
            doc.setFilename(filename);
            doc.setContent(content);

            Document saved = documentService.create(doc);

            Command command = new UploadDocumentCommand(documentService, saved);
            invoker.executeCommand(command);
```

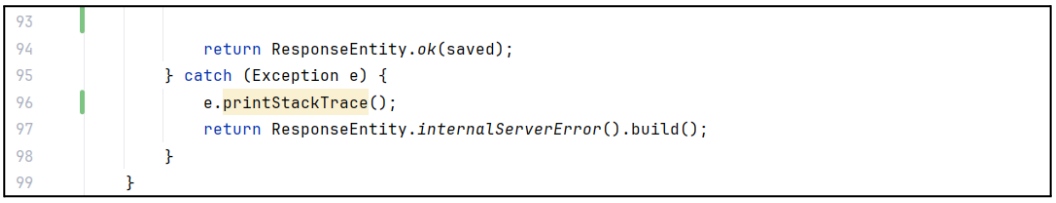


Рис. 9 – Контролер DocumentController

Контролер *DocumentController*, наведений на Рис. 9, виконує роль клієнта у структурі патерна Command. Він отримує HTTP-запити від користувача, створює відповідні команди, передає їм об'єкт-отримувач (*DocumentServiceImpl*) і виконує ці команди через *CommandInvoker*.

- Метод *save()* — створює команду для збереження документа і виконує її через CommandInvoker.
- Метод *open()* — створює команду для відкриття документа і виконує її через CommandInvoker.
- Метод *edit()* — створює команду для редагування документа і виконує її через CommandInvoker.
- Метод *upload()* — створює команду для завантаження документа і виконує її через CommandInvoker.

Контролер координує виконання команд, делегуючи всю бізнес-логіку об'єкту-отримувачу (*DocumentServiceImpl*), що дозволяє зберігати історію операцій і повторно виконувати команди при потребі.

8. ***Інтерфейс DocumentService*** представлений на Рисунку 10:

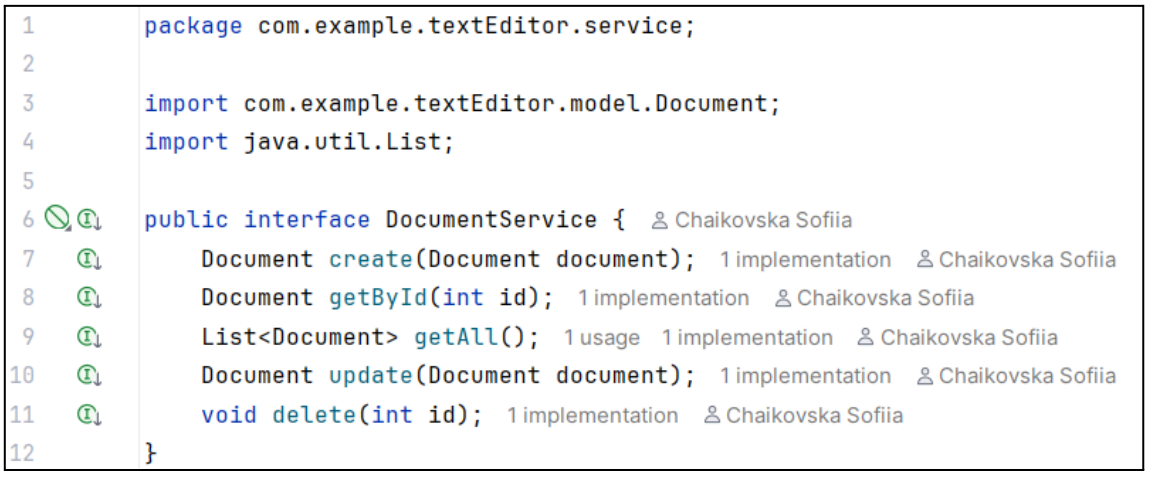


Рис. 10 – Інтерфейс DocumentService

Інтерфейс *DocumentService*, який наведено на Рис. 10, визначає основні операції для роботи з документами: створення, отримання за id, отримання всіх документів, оновлення та видалення. Він описує набір дій, які можуть бути виконані над об’єктом *Document*, не вдаючись до деталей реалізації.

9. ***Клас DocumentServiceImpl*** представлений на Рисунку 11:

```java
package com.example.textEditor.service.impl;

import com.example.textEditor.model.Document;
import com.example.textEditor.repository.DocumentRepository;
import com.example.textEditor.service.DocumentService;
import org.springframework.stereotype.Service;
import com.example.textEditor.strategy.SyntaxHighlightService;

import java.sql.Timestamp;
import java.util.List;

@Service  Chaikovska Sofiia
public class DocumentServiceImpl implements DocumentService {

    private final DocumentRepository documentRepository;  6 usages
    private final SyntaxHighlightService syntaxHighlightService;  3 usages

    public DocumentServiceImpl(DocumentRepository documentRepository,  Chaikovska Sofiia
                               SyntaxHighlightService syntaxHighlightService) {
        this.documentRepository = documentRepository;
        this.syntaxHighlightService = syntaxHighlightService;
    }

    @Override  Chaikovska Sofiia
    public Document create(Document document) {
        Timestamp now = new Timestamp(System.currentTimeMillis());
        document.setCreatedAt(now);
        document.setUpdatedAt(now);

        if (document.getExtension() == null || document.getExtension().isBlank()) {
            detectExtensionByContent(document);
        } else {
            String filename = document.getFilename();
            if (filename == null || filename.isBlank()) {
                filename = "newfile." + document.getExtension();
            } else if (!filename.contains(".")) {
                filename = filename + "." + document.getExtension();
            }
            document.setFilename(filename);
        }

        String highlighted = syntaxHighlightService.highlight(document);
        document.setHighlightedContent(highlighted);

        return documentRepository.save(document);

    }

```

```java
    @Override  Chaikovska Sofiia
    public Document getById(int id) {
        return documentRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Document not found"));

    }

```

```java
    @Override  1 usage  Chaikovska Sofiia
    public List<Document> getAll() {
        return documentRepository.findAll();
    }

    @Override  Chaikovska Sofiia
    public Document update(Document document) {
        document.setUpdatedAt(new Timestamp(System.currentTimeMillis()));

        detectExtensionByContent(document);

        String highlighted = syntaxHighlightService.highlight(document);
        document.setHighlightedContent(highlighted);

        return documentRepository.save(document);
    }
```

Рис. 11 – Клас DocumentServiceImpl

Клас *DocumentServiceImpl* реалізує інтерфейс *DocumentService* і виступає отримувачем (*receiver*) у патерні Command. Він інкапсулює всю бізнес-логіку роботи з документами. Усі команди (*SaveDocumentCommand, EditDocumentCommand, OpenDocumentCommand, UploadDocumentCommand*) взаємодіють із цим класом для виконання своїх дій.

Основні методи класу:

- *create(Document document)* — створює новий документ у базі даних. Під час створення автоматично встановлює час створення й оновлення, визначає розширення файлу, генерує ім’я за потреби та виконує підсвічування синтаксису перед збереженням.
- *getById(int id)* — отримує документ за його ідентифікатором, викидаючи виняток, якщо документ не знайдено.
- *getAll()* — повертає список усіх документів, збережених у базі.
- *update(Document document)* — оновлює існуючий документ, оновлює дату зміни, повторно визначає розширення та підсвічує текст перед збереженням.

6 Діаграма класів для патерну «Command»

Діаграма класів для патерну «Command» зображена на Рисунку 12:

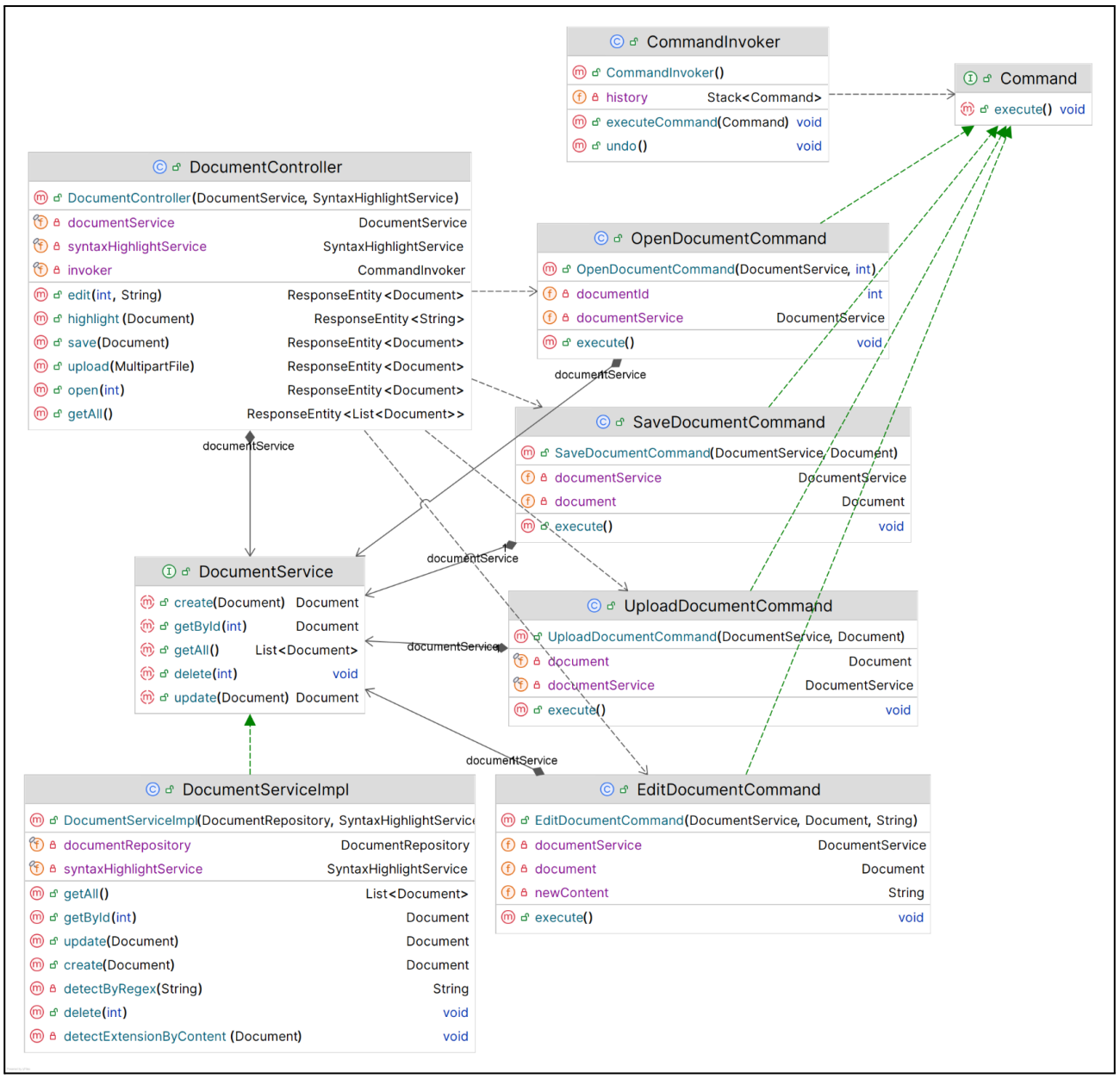


Рис. 12 – Діаграма класів для патерну «Command»

Діаграма класів для патерну «Command»:

Інтерфейс:

- *Command* – визначає загальний контракт для всіх команд. Містить метод *execute(),* який реалізують конкретні команди. Це дає змогу інкапсулювати дію як об’єкт, відокремивши відправника команди від виконавця.

Конкретні команди:

1. *OpenDocumentCommand* – реалізує відкриття документа. У методі *execute()* викликає відповідний метод сервісу для завантаження документа за ідентифікатором.
2. *SaveDocumentCommand* – реалізує збереження документа. Метод *execute()* зберігає документ через сервіс.
3. *UploadDocumentCommand* – реалізує завантаження нового документа у систему. Метод *execute()* викликає логіку завантаження файлу в сервісі.
4. *EditDocumentCommand* – реалізує редагування існуючого документа. Метод *execute()* оновлює вміст документа через сервіс.

Invoker:

- *CommandInvoker* – відповідає за виконання команд і управління історією операцій. Містить стек *history* для зберігання виконаних команд. Метод *executeCommand(Command)* – викликає *execute()* у команди та додає її в історію. *Invoker* не знає деталей реалізації команд — він просто викликає метод *execute()*, що забезпечує низьке зв’язування.

Receiver:

- *DocumentService* – інтерфейс, який містить методи, які реалізують реальні дії над документами: *create(), getById(), getAll(), delete(), update()*. Це логічна частина, що виконує запити, викликані командами.
- *DocumentServiceImpl* – конкретна реалізація інтерфейсу *DocumentService*. Використовує *DocumentRepository* для доступу до бази даних. Реалізує набір CRUD-операцій.

Клієнт (Client):

- *DocumentController* – клієнт, який створює та ініціює команди. Методи контролера (*open, save, upload, edit, getAll*) створюють відповідні команди і передають їх в *CommandInvoker* для виконання.

Така архітектура реалізує чисте розділення відповідальностей:

- Контролер лише формує команди й передає їх Invoker.
- Команди інкапсулюють логіку виконання певної дії.
- Сервіс реалізує бізнес-операції над документами.
- Invoker управляє викликами та історією.

Посилання на GitHub: https://github.com/chaikovsska/textEditor

7 Висновки

У ході виконання лабораторної роботи було реалізовано патерн проєктування «Command» для текстового редактора, що забезпечив зручне керування діями користувача, такими як створення, редагування, видалення або відміна змін у документах. Завдяки цьому патерну кожна дія була інкапсульована в окремий об’єкт-команду, що дозволило відокремити ініціатора дії від її виконавця та реалізувати можливість скасування або повтору операцій. Використання «Command» підвищило гнучкість і масштабованість системи, спростило додавання нових функцій без зміни основної логіки програми. Реалізація цього патерну зробила архітектуру застосунку більш модульною, підтримуваною та придатною до розширення в майбутньому.

8 Відповіді на питання

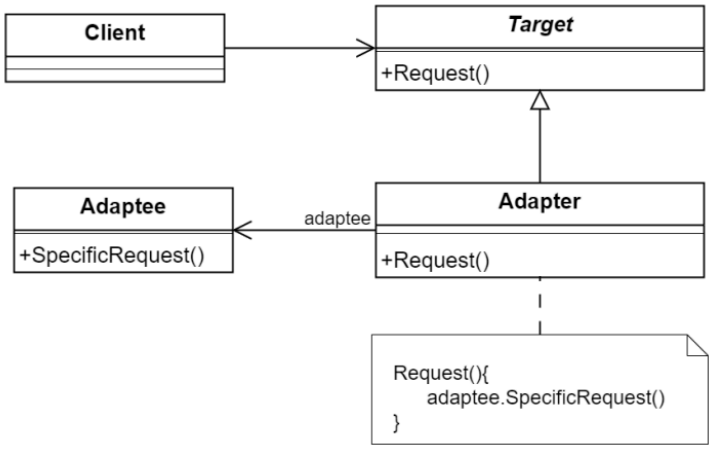
1. **Яке призначення шаблону «Адаптер»?**

   Шаблон «Адаптер» використовується для узгодження двох несумісних інтерфейсів. Його основне призначення полягає в тому, щоб дозволити об’єктам із різними інтерфейсами працювати разом, не змінюючи їхнього вихідного коду. «Адаптер» виступає посередником, який перетворює інтерфейс одного класу в інтерфейс, очікуваний іншим класом. Завдяки цьому можна повторно використовувати вже існуючі класи в нових системах без модифікації їх реалізації, що сприяє зниженню зв’язності та підвищенню гнучкості програми.

2. **Нарисуйте структуру шаблону «Адаптер».**

   Структура шаблону «Адаптер» складається з чотирьох основних елементів: клієнта (Client), цільового інтерфейсу (Target), адаптера (Adapter) та адаптованого класу (Adaptee). Клієнт звертається до об’єкта

через інтерфейс Target, який реалізується класом Adapter. Усередині адаптера міститься виклик методів класу Adaptee, що має інший інтерфейс.



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

   До шаблону «Адаптер» входять чотири класи. Клас Client надсилає запит через інтерфейс Target, який визначає загальний спосіб виклику функціоналу. Клас Adaptee містить специфічний інтерфейс або методи, несумісні з тим, що очікує клієнт. Адаптер (Adapter) реалізує інтерфейс Target і всередині себе містить екземпляр Adaptee, до якого делегує виклики.

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

   Реалізація шаблону «Адаптер» може бути двох типів: об’єктна та класова. Об’єктний адаптер використовує композицію — тобто містить у собі посилання на об’єкт класу Adaptee та викликає його методи. Такий підхід є більш гнучким, адже дозволяє адаптувати як окремий об’єкт, так і групу об’єктів динамічно. Класовий адаптер, навпаки, реалізується через множинне наслідування, коли Adapter одночасно наслідує як Target, так і Adaptee, перевизначаючи необхідні методи.

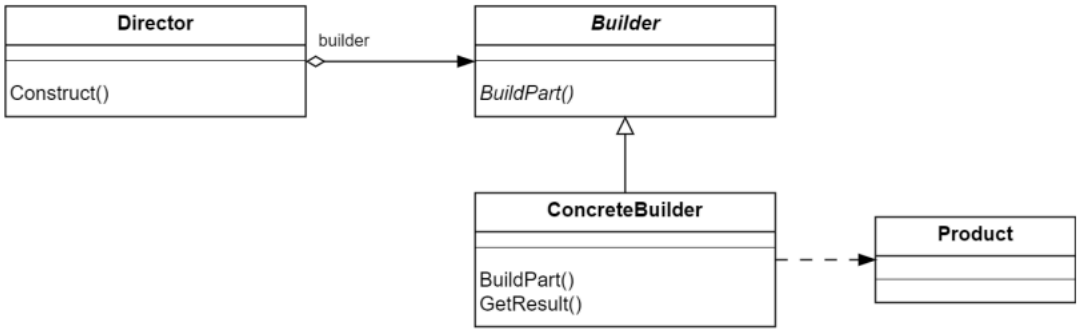
5. **Яке призначення шаблону «Будівельник»?**

   Шаблон «Будівельник» призначений для створення складних об’єктів крок за кроком, відокремлюючи процес побудови від його кінцевого

представлення. Це дозволяє використовувати один і той самий алгоритм побудови для створення різних варіантів об’єкта.

6. **Нарисуйте структуру шаблону «Будівельник».**

Структура шаблону «Будівельник» складається з чотирьох основних елементів: Director, Builder, ConcreteBuilder та Product. Director визначає порядок виклику методів будівельника для створення об’єкта. Інтерфейс Builder описує кроки створення частин продукту. ConcreteBuilder реалізує конкретні кроки побудови та створює об’єкт типу Product. У результаті взаємодії Director викликає методи ConcreteBuilder, який поступово формує готовий об’єкт Product і повертає його клієнту.



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

У шаблон входять класи Director, Builder, ConcreteBuilder і Product. Клас Director визначає послідовність кроків для побудови складного об’єкта. Інтерфейс Builder містить абстрактні методи, необхідні для створення частин продукту. ConcreteBuilder реалізує ці методи, створюючи конкретні частини, і формує кінцевий продукт. Після завершення побудови об’єкт Product передається клієнту. Director не залежить від конкретного типу будівельника — він працює лише з інтерфейсом, що дозволяє легко змінювати тип створюваних об’єктів.

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

Шаблон «Будівельник» варто застосовувати тоді, коли об’єкт має складну структуру або створюється в кілька етапів. Його використовують, коли потрібно створювати різні представлення одного і того ж об’єкта,

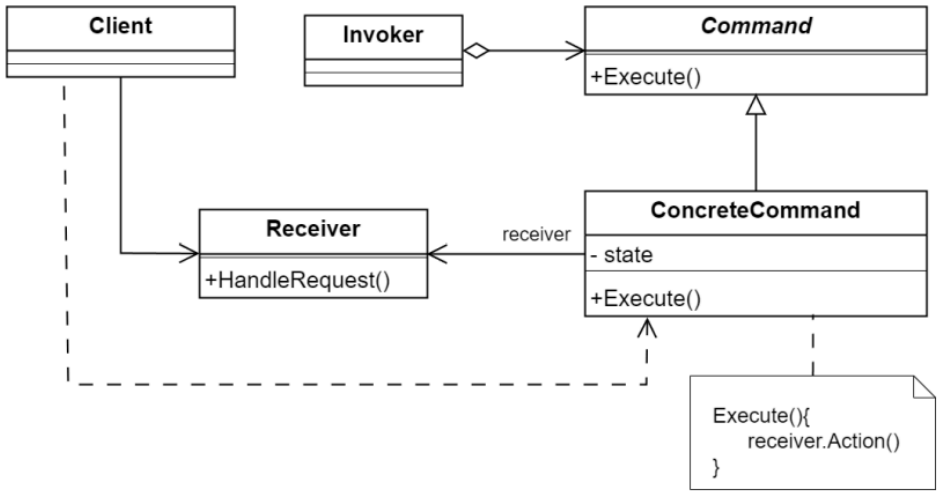
змінюючи лише частини або порядок побудови, а також тоді, коли процес створення об’єкта повинен бути незалежним від його складових.

9. Яке призначення шаблону «Команда»?

Шаблон «Команда» призначений для інкапсуляції запиту як об’єкта, що дозволяє відокремити клієнта, який ініціює дію, від об’єкта, який її виконує. Він забезпечує можливість зберігати історію дій, скасовувати виконані операції (undo), повторювати їх (redo) або виконувати команди у певній послідовності. Завдяки цьому патерну всі операції можна обробляти через єдиний інтерфейс, що робить систему більш гнучкою та розширюваною. Нові команди можна додавати без зміни існуючого коду, просто створюючи новий клас команди.

10. Нарисуйте структуру шаблону «Команда»

Структура шаблону «Команда» включає такі елементи: Client, Invoker, Command, ConcreteCommand і Receiver. Клієнт створює об’єкт команди (ConcreteCommand) і передає його виконавцю (Invoker). Команда реалізує інтерфейс Command, який містить метод execute(). Усередині команди міститься посилання на отримувача (Receiver), який виконує фактичну дію. Invoker викликає метод execute(), не знаючи деталей реалізації команди. У результаті клієнт може запускати різні дії через один інтерфейс, а Invoker може зберігати або повторювати команди.



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

До шаблону «Команда» входять такі класи: Command, ConcreteCommand, Receiver, Invoker і Client. Інтерфейс Command визначає метод execute(), який виконується всіма командами. ConcreteCommand реалізує цей інтерфейс і містить посилання на Receiver, який знає, як виконати дію. Invoker зберігає команду та викликає її виконання. Client створює команду, встановлює отримувача та передає команду інвокеру. Коли Invoker викликає execute(), команда делегує виконання Receiver.

12. Розкажіть як працює шаблон «Команда».

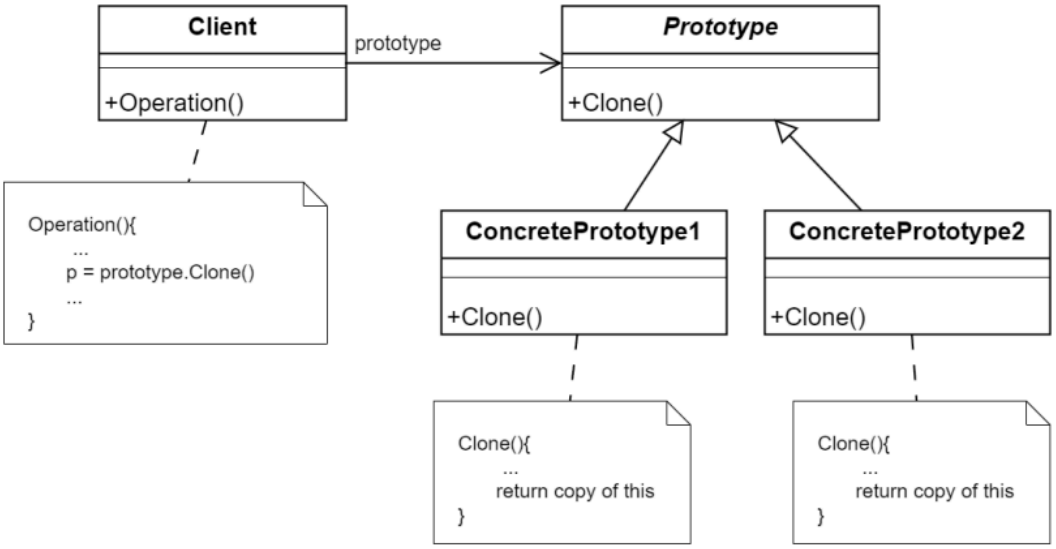
Клієнт створює об’єкт команди та задає, хто буде виконавцем дії (Receiver). Потім команда передається об’єкту Invoker, який відповідає за виклик методу execute() у потрібний момент. Коли Invoker викликає команду, вона звертається до свого отримувача (Receiver) і виконує відповідну дію. Таким чином, запит інкапсульований у вигляді об’єкта, який можна зберігати, виконувати пізніше або навіть скасувати.

13. Яке призначення шаблону «Прототип»?

Шаблон «Прототип» використовується для створення нових об’єктів шляхом копіювання існуючого зразка, замість створення їх із нуля через конструктор. Це дозволяє значно скоротити витрати часу, коли створення об’єкта є ресурсомістким або вимагає складної ініціалізації. Кожен об’єкт-прототип реалізує метод клонування, який повертає його копію.

14. Нарисуйте структуру шаблону «Прототип».

У структурі шаблону «Прототип» є три основні учасники: Prototype, ConcretePrototype і Client. Prototype визначає інтерфейс для клонування об’єктів за допомогою методу clone(). ConcretePrototype реалізує цей метод і створює точну копію самого себе. Client звертається до прототипу та створює новий об’єкт шляхом копіювання, а не виклику конструктора.



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

У шаблон входять класи Prototype, ConcretePrototype і Client. Prototype є базовим інтерфейсом або абстрактним класом із методом clone(). ConcretePrototype реалізує цей метод, виконуючи копіювання поточного об’єкта. Client використовує метод clone() для створення нових екземплярів без виклику конструктора. Взаємодія проста: клієнт викликає клонування на конкретному прототипі, отримує копію і працює з нею незалежно від оригіналу.

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

Шаблон «Ланцюжок відповідальності» часто застосовується для послідовної обробки запитів кількома об’єктами, де кожен може або обробити запит, або передати його далі по ланцюгу. Прикладами використання є обробка подій у графічних інтерфейсах користувача (наприклад, натискання кнопок, гарячі клавіші, меню), фільтрація HTTP-запитів у веб-застосунках, реалізація систем логування або перевірки доступу. Також цей шаблон широко застосовується в серверних middleware-ланцюгах (як у Spring Filters або Express.js), де кожен обробник виконує свою частину логіки та передає запит далі по ланцюгу, доки не буде знайдено відповідного виконавця.