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

«Патерни проектування»

Виконав
студент групи ІА-34:
Сльота Максим

Перевірив:
Мягкий М. Ю.

Київ - 2025

Тема: Патерни проектування.

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

Вихідні дані:

17. **System activity monitor** (iterator, command, abstract factory, bridge, visitor, SOA)

Монітор активності системи повинен зберігати і запам'ятовувати статистику використовуваних компонентів системи, включаючи навантаження на процесор, обсяг займаної оперативної пам'яті, натискання клавіш на клавіатурі, дії миші (переміщення, натискання), відкриття вікон і зміна вікон; будувати звіти про використання комп'ютера за різними критеріями (% часу перебування у веб-браузері, середнє навантаження на процесор по годинах, середній час роботи комп'ютера по днях і т.д.); правильно поводитися з «простоюванням» системи – відсутністю користувача.

Теоретичні відомості:

Шаблон «Adapter» - 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.

Шаблон «Builder» - використовується для відділення процесу створення об'єкту від його представлення. Це доречно у випадках, коли об'єкт має складний процес створення (наприклад, Web-сторінка як елемент повної відповіді web- сервера) або коли об'єкт повинен мати декілька різних форм створення (наприклад, при конвертації тексту з формату у формат).

Шаблон «Command» - перетворить звичайний виклик методу в клас [6]. Таким чином дії в системі стають повноправними об'єктами. Це зручно в наступних випадках: коли потрібна розвинена система команд – відомо, що команди будуть добавлятися; коли потрібна гнучка система команд – коли з'являється необхідність додавати командам

можливість відміни, логування і інш.; Коли потрібна можливість складання ланцюжків команд або виклику команд в певний час. Об'єкт команда сама по собі не виконує ніяких фактичних дій окрім перенаправлення запиту одержувачеві (тобто команди все ж виконуються одержувачем), однак ці об'єкти можуть зберігати дані для підтримки додаткових функцій відміни, логування і інш.

Шаблон «Chain of Responsibility» - призначення патерну 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.

Шаблон «Prototype» - 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.

Хід роботи:

1. Діаграма класів, яка представляє використання шаблону Command

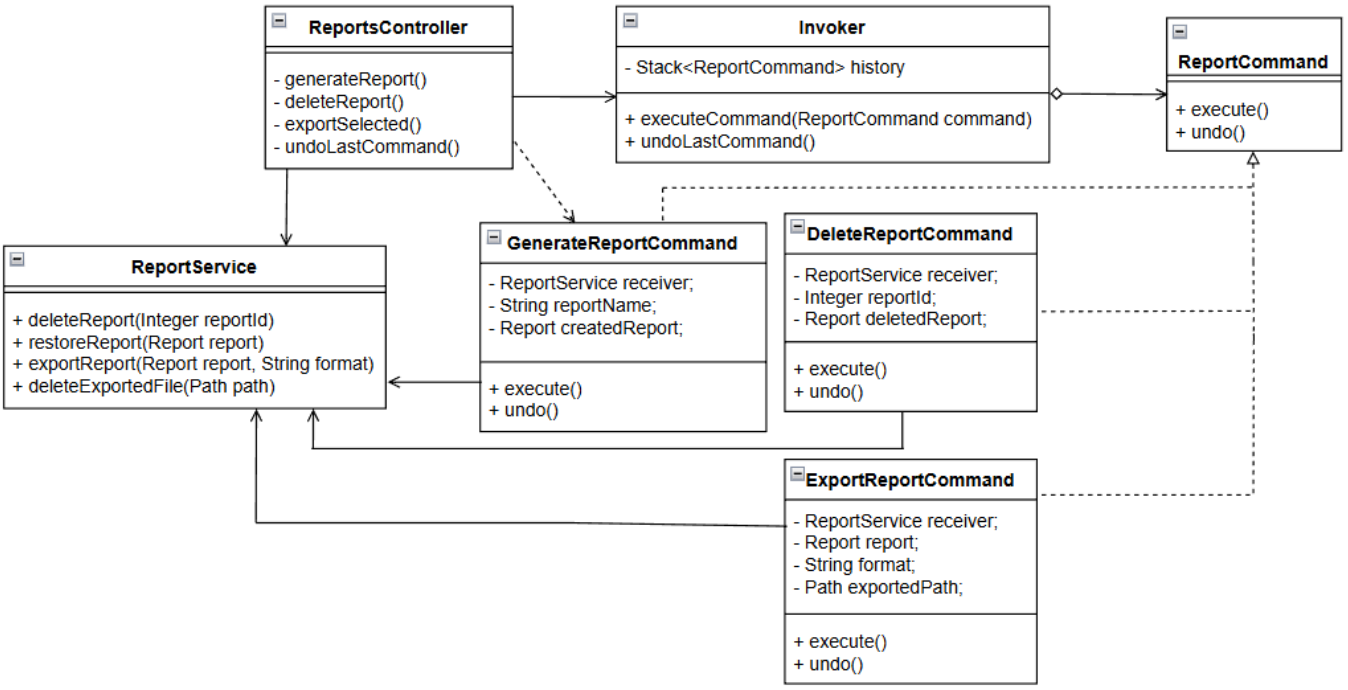


Рис 1. Діаграма класів

У проєкті System Activity Monitor шаблон «Команда» використано для вирішення проблеми залежності між інтерфейсом користувача та бізнес-логікою, а також для забезпечення можливості скасовувати дії користувача (Undo). Без застосування цього патерну усі операції зі створення, видалення чи експорту звітів довелося б виконувати безпосередньо в контролері. Такий підхід робив би контролер перевантаженим та

сильно прив’язаним до конкретних методів сервісного шару, ускладнюючи супровід, тестування та розширення системи.

Шаблон «Команда» дозволяє інкапсулювати кожну операцію у вигляді окремого об’єкта-команди. Зовнішній вигляд команди визначає інтерфейс ReportCommand, який встановлює єдиний спосіб взаємодії через методи execute() і undo(). Завдяки цьому виконання дії стає незалежним від того, де вона була ініційована, а самі команди можна легко зберігати, запускати повторно або відміняти.

Проблема прямого виклику бізнес-логіки в контролері вирішується за рахунок того, що ReportsController тепер лише ініціює дію: збирає параметри й створює відповідний об’єкт-команду. Він не виконує операції та не знає деталей їх реалізації. Команда передається в Invoker, який відповідає за її запуск і веде історію виконаних команд, дозволяючи реалізувати механізм Undo.

Реальне виконання операції відбувається у ReportService, який виступає об’єктом-отримувачем. Саме він містить бізнес-логіку створення, видалення, експорту чи відновлення звітів. Конкретні команди, такі як GenerateReportCommand, DeleteReportCommand та ExportReportCommand, виступають проміжною ланкою: вони зберігають параметри, необхідні для виконання дії, і делегують роботу ReportService.

Доданий вихідний код системи:

ReportCommand.java (Command):

```
public interface ReportCommand {
   void execute();
   default void undo() {
      throw new UnsupportedOperationException("Undo не підтримується для цієї команди.");
   }
}
```

ReportCommand задає загальний контракт для всіх команд.
Кожна команда повинна вміти виконати дію (execute), а також опціонально скасувати її (undo).
Інтерфейс визначає єдину форму для всіх операцій, що дозволяє Invoker працювати з ними однаково.

GenerateReportCommand.java (Concrete command)

```
public class GenerateReportCommand implements ReportCommand {
   private final ReportService receiver;
   private final User user;
   private final String reportName;
   private final LocalDate start;
   private final LocalDate end;
```

```
    private Report createdReport;
    @Override
    public void execute() {
        createdReport = receiver.generateReport(user, reportName, start, end);
    }
    @Override
    public void undo() {
        receiver.deleteReport(createdReport.getId());
    }
}
```

GenerateReportCommand відповідає за створення нового звіту.

Команда зберігає всі параметри, необхідні для створення, й делегує роботу класу ReportService.

При виконанні - генерує звіт і запам’ятовує його, щоб у undo() мати можливість видалити створений звіт.

DeleteReportCommand.java (Concrete command)

```
public class DeleteReportCommand implements ReportCommand {
    private final ReportService receiver;
    private final Integer reportId;
    private Report deletedReport;
    @Override
    public void execute() {
        deletedReport = receiver.findById(reportId);
        receiver.deleteReport(reportId);
    }
    @Override
    public void undo() {
        receiver.restoreReport(deletedReport);
    }
}
```

DeleteReportCommand виконує видалення звіту.

Перед видаленням команда зберігає сам звіт, щоб мати можливість його відновити, якщо користувач викликає undo(). Уся реальна логіка видалення та відновлення виконується сервісом ReportService.

ExportReportCommand.java (Concrete command)

```
public class ExportReportCommand implements ReportCommand {
    private final ReportService receiver;
    private final Report report;
    private final String format;
    private Path exportedPath;
```

```
    @Override
    public void execute() {
        exportedPath = receiver.exportReport(report, format);
    }
    @Override
    public void undo() {
        receiver.deleteExportedFile(exportedPath);
    }
}
```

ExportReportCommand займається експортом звіту у різні формати (CSV, Excel, PDF).

Після експорту команда запам’ятовує шлях до створеного файлу.

Метод undo() дає змогу видалити експортований файл, якщо потрібно скасувати операцію.

ReportService.java (Receiver)

```
public class ReportService {
    public Report generateReport(User user, String reportName, LocalDate startDate, LocalDate endDate)
{… }
    public void deleteReport(Integer reportId) {…}
    public void restoreReport(Report report) {…}
    public Path exportReport(Report report, String format) {…}
    public void deleteExportedFile(Path path) {…}
}
```

ReportService - це отримувач, який містить усю бізнес-логіку роботи зі звітами.

Invoker.java

```
public class Invoker {
    private final Stack<ReportCommand> history = new Stack<>();
    public void executeCommand(ReportCommand command) {
        command.execute();
        history.push(command);
    }
    public void undoLastCommand() {
        ReportCommand last = history.pop();
        last.undo();
    }
}
```

Invoker не знає, що саме робить команда — він працює з нею абстрактно, через інтерфейс Command.

ReportsController.java (Client)

```
private final Invoker commandManager = new Invoker();
ReportCommand cmd = new GenerateReportCommand(
```

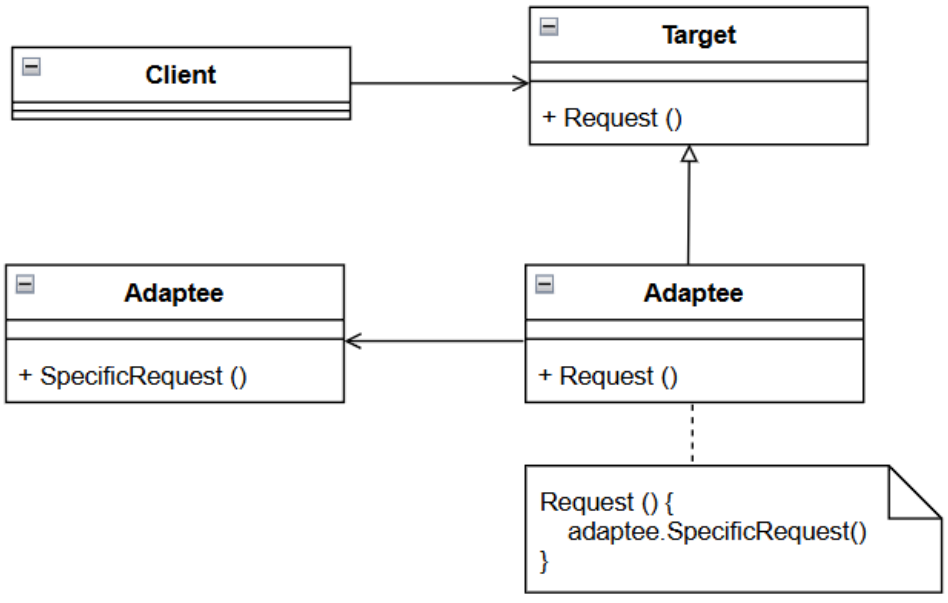
```
        reportService,
        user,
        reportNameField.getText(),
        start,
        end
);
commandManager.executeCommand(cmd);
ReportCommand cmd = new DeleteReportCommand(reportService, selected.getId());
commandManager.executeCommand(cmd);
ReportCommand cmd = new ExportReportCommand(reportService, selected, format);
commandManager.executeCommand(cmd);
commandManager.undoLastCommand();
```

ReportsController - це клієнт, який ініціює операції. Контролер не містить бізнес-логіки, а лише формує команди та передає їх об’єктам патерну.

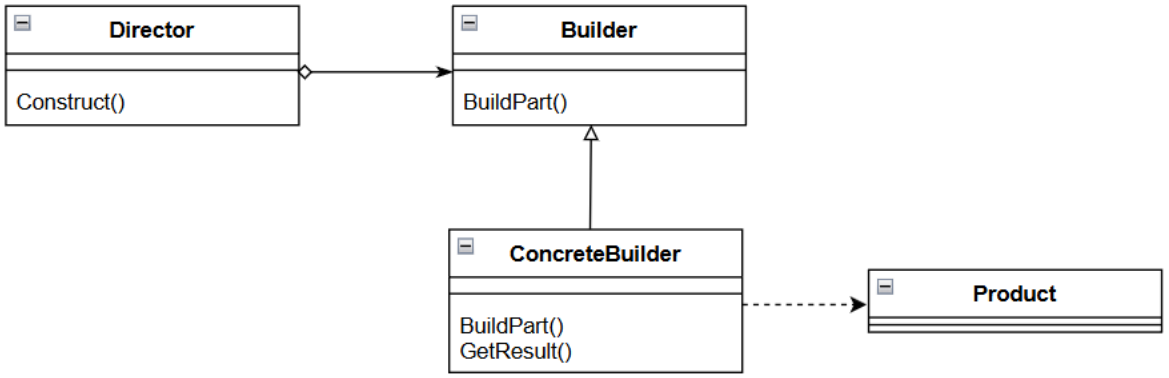
Висновок: У результаті реалізації шаблону «Command» у проєкті System Activity Monitor вдалося перетворити окремі операції над звітами в автономні об’єкти-команди. Використання команд зробило систему гнучкою, розширюваною та придатною до подальшого розвитку, оскільки додавання нових операцій тепер не потребує зміни існуючої архітектури.

Відповідь на контрольні питання:

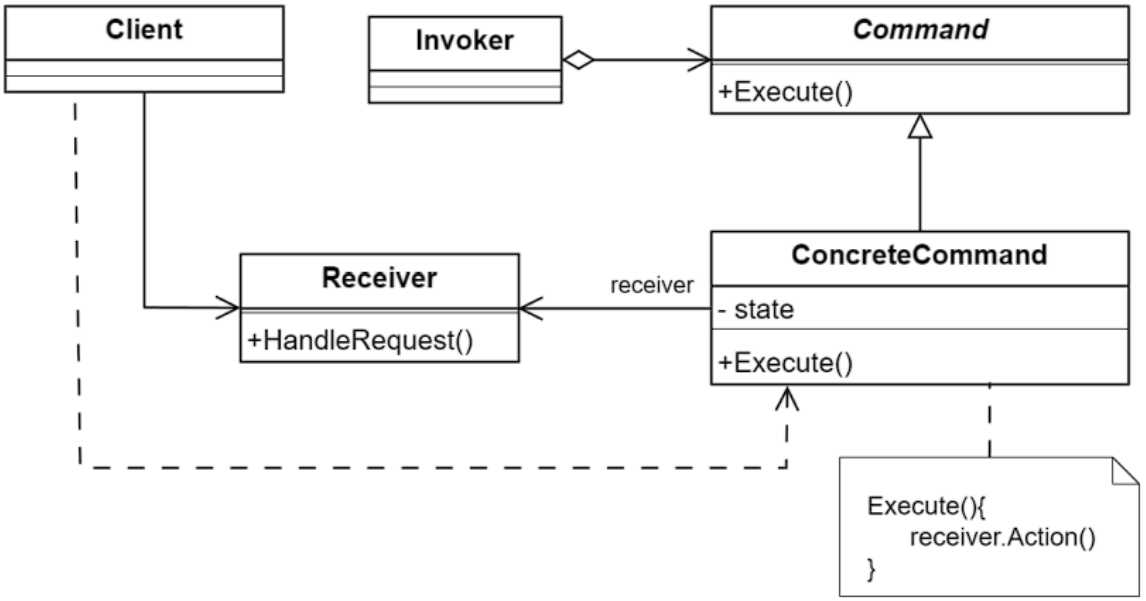
1. Призначення шаблону «Адаптер» у тому, щоб з’єднати два несумісні класи. Він дозволяє працювати з об’єктом, інтерфейс якого не підходить клієнту, перетворюючи його в той формат, який очікується.
2.



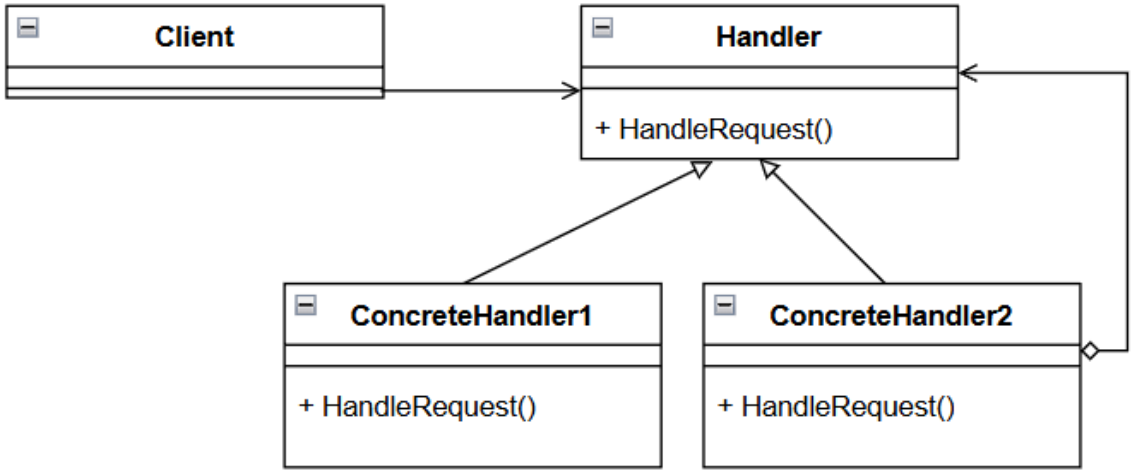
3. Різниця між адаптером на рівні об’єктів і на рівні класів полягає в способі з’єднання. Об’єктний адаптер використовує композицію, тобто зберігає всередині об’єкт і делегує йому виклики. Класовий адаптер будується через множинне наслідування, тому одночасно наслідує і потрібний інтерфейс, і сторонній клас.
4. Призначення шаблону «Будівельник» у тому, щоб створювати складні об’єкти поступово, крок за кроком. Він дозволяє розділити процес побудови від самого об’єкта і формувати різні його варіанти.
5. Структура «Будівельника» складається з керівника, який задає порядок виконання кроків, із самого інтерфейсу будівельника, конкретних будівельників та кінцевого продукту, який вони створюють.
6.



7. Будівельник застосовується тоді, коли об’єкт має багато параметрів, різні конфігурації або коли звичайний конструктор стає занадто громіздким. Він корисний для побудови складних, багатоступеневих або змінних об’єктів.
8. Призначення шаблону «Команда» в тому, щоб представити дію як окремий об’єкт. Це дозволяє передавати операції, ставити їх у чергу, зберігати історію, робити undo та відокремлювати ініціатора дії від того, хто її виконує.
9. Структура «Команди» складається з інтерфейсу команди з методом execute, конкретних команд, які реалізують цей метод, отримувача, який виконує реальні дії, invoker, який запускає команди, і клієнта, який створює та налаштовує їх.
10.



11. Працює шаблон просто: кожна дія загортається в об’єкт команди. Invoker не знає, як команда працює, він просто викликає її виконання. Команда всередині звертається до отримувача, який робить справжню роботу. Завдяки цьому можна легко додавати нові дії, зберігати історію та робити скасування.
12. Призначення шаблону «Прототип» у тому, щоб створювати нові об’єкти шляхом копіювання існуючих. Це корисно, коли створення об’єкта є дорогим, або коли потрібно копіювати складні об’єкти з певним початковим станом.
13. Структура «Прототипа» складається з інтерфейсу з методом clone, конкретних класів, які реалізують цей метод, і клієнта, який створює нові об’єкти не через конструктор, а через клонування.
14.



15. Приклади ланцюжка відповідальності можна побачити в логуванні, де повідомлення передаються по рівнях, поки один із них не обробить їх. У графічних інтерфейсах події миші піднімаються від дочірніх вікон до

батьківських. В системах авторизації запит проходить через кілька перевірок: роль, токен, права, сесія. Кожен обробник може або опрацювати запит, або передати його далі.