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

«ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»»

Виконала
студентка групи ІА–22:
Фоменко Альона

Перевірив:
Мягкий Михайло Юрійович

Київ 2024

Зміст

Тема: ШАБЛОНИ «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»

Мета: Ознайомитися з короткими теоретичними відомостями. Реалізувати частину функціоналу робочої програми у вигляді класів та їх взаємодій для досягнення конкретних функціональних можливостей. Застосування одного з розглянутих шаблонів при реалізації програми

Теоретичні відомості

«Анти-шаблони» проектування Анти-патерни (anti-patterns), також відомі як пастки (pitfalls) - це класи найбільш часто впроваджуваних поганих рішень проблем. Вони вивчаються, як категорія, в разі коли їх хочуть уникнути в майбутньому, і деякі їхні окремі випадки можуть бути розпізнані при вивченні непрацюючих систем. Термін походить з інформатики, від авторів «Банди чотирьох» книги «Шаблони проектування», яка заклала приклади практики хорошого програмування. Автори назвали ці хороші методи «шаблонами проектування», і протилежними їм є «анти-патерни». Частиною хоршої практики програмування є уникнення анти-патернів.

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. Це дозволить в програмі звертатися до загального інтерфейсу, а не приводити різні сценарії роботи залежно від способу реалізації бібліотеки. Адаптери також називаються "wrappers" (обгортками).

Проблема:

Уявіть, що ви пишете додаток для біржі. Ваша програма спочатку завантажує біржові котирування з декількох джерел в XML, а потім малює гарні графіки. У якийсь момент ви вирішуєте покращити програму, застосувавши сторонню бібліотеку аналітики. Але от біда — бібліотека підтримує тільки формат даних JSON, несумісний із вашим додатком. Ви могли б переписати цю бібліотеку, щоб вона підтримувала формат XML, але, поперше, це може порушити роботу наявного коду, який уже залежить від бібліотеки, по-друге, у вас може просто не бути доступу до її вихідного коду.

Рішення:

Ви можете створити адаптер. Це об’єкт-перекладач, який трансформує інтерфейс або дані одного об’єкта таким чином, щоб він став зрозумілим іншому об’єкту. Адаптер загортає один з об’єктів так, що інший об’єкт навіть не підозрює про існування першого. Наприклад, об’єкт, що працює в метричній системі вимірювання, можна «обгорнути» адаптером, який буде конвертувати дані у фути. Таким чином, для програми біржових котирувань ви могли б створити клас XML_To_JSON_Adapter, який би обгортав об’єкт того чи іншого класу бібліотеки аналітики. Ваш код посилав би адаптеру запити у форматі XML, а адаптер спочатку б транслював вхідні дані у формат JSON, а потім передавав їх методам загорнутого об’єкта аналітики

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, 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).

Проблема:

Візьмемо процес побудови відповіді на запит web- сервера. Побудова складається з наступних частин: додавання стандартних заголовків (дата/час, ім'я сервера, інш.), код статусу (після пошуку відповідної сторінки на сервері), заголовки відповіді (тип вмісту, інш.), утримуване, інше.

Рішення:

Кожен з цих етапів може бути абстрагований в окремий метод будівельника. Це дасть наступні вигоди: - Гнучкіший контроль над процесом створення сторінки; - Незалежність від внутрішніх змін - наприклад, зміна назви сервера не сильно порушить процес побудови відповіді;

Шаблон «Command»

Призначення патерну:

Шаблон "command" (команда) перетворить звичайний виклик методу в клас. 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. Наприклад, команда

вставки символу може запам'ятовувати символ, і при виклику відміни викликати відповідну функцію витирання символу. Можна також визначити параметр "застосовності" команди (наприклад, на картинці писати не можна) - і використати цей аттрибут для засвічування піктограми в меню. Такий підхід до команд дозволяє побудувати дуже гнучку систему команд, що настроюється. У більшості додатків це буде зайвим (використовується спрощений варіант), проте життєво важливий в додатках з великою кількістю команд (редактори).

Проблема:

Уявіть, що ви працюєте над програмою текстового редактора. Якраз підійшов час розробки панелі керування. Ви створили клас гарних кнопок і хочете використовувати його для всіх кнопок програми, починаючи з панелі керування та закінчуючи звичайними кнопками в діалогах. Усі ці кнопки, хоч і виглядають схоже, але виконують різні команди. Виникає запитання: куди розмістити код обробників кліків по цих кнопках? Найпростіше рішення — це створити підкласи для кожної кнопки та перевизначити в них методи дії для різних завдань.

Рішення:

Хороші програми зазвичай структурують у вигляді шарів. Найпоширеніший приклад — це шари користувацького інтерфейсу та бізнес-логіки. Перший лише малює гарне зображення для користувача, але коли потрібно зробити щось важливе, інтерфейс користувача «просить» шар бізнес-логіки зайнятися цим. У дійсності це виглядає так: один з об’єктів інтерфейсу користувача викликає метод одного з об’єктів бізнес-логіки, передаючи до нього якісь параметри. Патерн пропонує більше не надсилати такі виклики безпосередньо. Замість цього кожен виклик, що відрізняється від інших, слід звернути у власний клас з єдиним методом, який і здійснюватиме виклик. Такий зветься командою.

Шаблон «Chain of Responsibility»

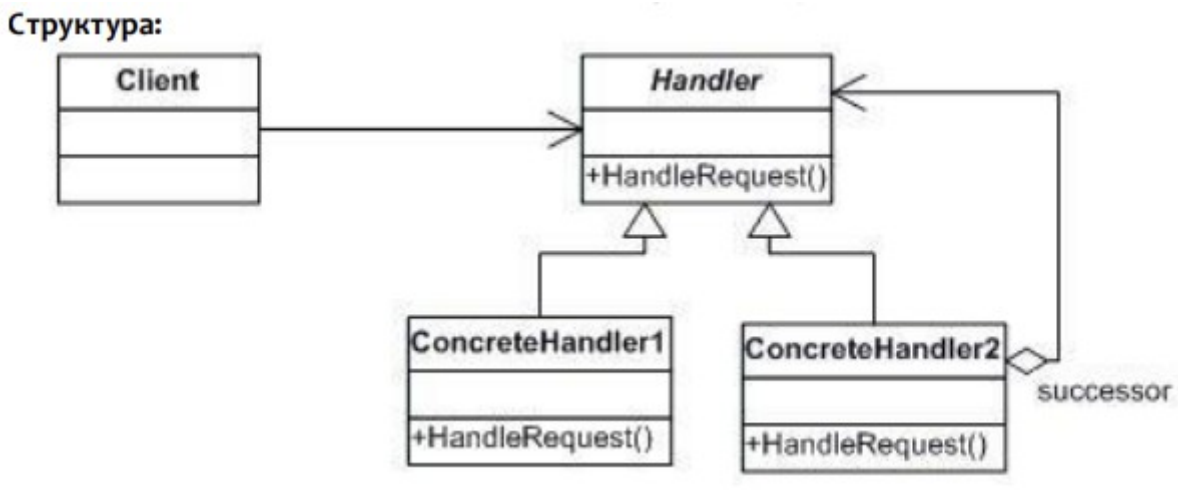

Рис. 1: Структара шаблону Chain of responsibility

Призначення патерну:

Шаблон "chain of responsibility" (ланцюг 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.

Проблема:

Уявіть, що ви робите систему прийому онлайн-замовлень. Ви хочете обмежити до неї доступ так, щоб тільки авторизовані користувачі могли створювати замовлення. Крім того, певні користувачі, які володіють правами адміністратора, повинні мати повний доступ до замовлень. Ви швидко збагнули, що ці перевірки потрібно виконувати послідовно. Адже користувача можна спробувати «залогувати» у систему, якщо його запит містить логін і пароль. Але, якщо така спроба не вдалась, то перевіряти розширені права доступу просто немає сенсу. З кожною новою «фічою» код перевірок, що виглядав як величезний клубок умовних операторів, все більше і більше «розбухав». При зміні одного правила доводилося змінювати код усіх інших перевірок.

Рішення:

Як і багато інших поведінкових патернів, ланцюжок обов’язків базується на тому, щоб перетворити окремі поведінки на об’єкти. У нашому випадку кожна перевірка переїде до окремого класу з одним методом виконання. Дані запиту, що перевіряється, передаватимуться до методу як аргументи. А тепер справді важливий етап. Патерн пропонує зв’язати всі об’єкти обробників в один ланцюжок. Кожен обробник міститиме посилання на наступного обробника в ланцюзі. Таким чином, після отримання запиту обробник зможе не тільки опрацювати його самостійно, але й передати обробку наступному об’єкту в ланцюжку.

Шаблон «Prototype»

Призначення патерну:

Шаблон "prototype" (прототип) використовується для створення об'єктів за "шаблоном" (чи "кресленню", "ескізу") шляхом копіювання шаблонного об'єкту. Для цього визначається метод "клонувати" в об'єктах цього класу. Цей шаблон зручно використати, коли заздалегідь відомо як виглядатиме кінцевий об'єкт (мінімізується кількість змін до об'єкту шляхом створення шаблону), а також для видалення необхідності створення об'єкту - створення відбувається за рахунок клонування, і зухвалій програмі абсолютно немає необхідності знати, як створювати об'єкт. Також, це дозволяє маніпулювати об'єктами під час виконання програми шляхом настроювання відповідних шаблонів; значно

зменшується ієрархія спадкоємства (оскільки в іншому випадку це були б не шаблони, а вкладені класи, що наслідують).

Проблема:

У вас є об’єкт, який потрібно скопіювати. Як це зробити? Потрібно створити порожній об’єкт того самого класу, а потім по черзі копіювати значення всіх полів зі старого об’єкта до нового. Чудово! Проте є нюанс. Не кожен об’єкт вдасться скопіювати у такий спосіб, адже частина його стану може бути приватною, а значить — недоступною для решти коду програми.

Рішення:

Патерн доручає процес копіювання самим об’єктам, які треба скопіювати. Він вводить загальний інтерфейс для всіх об’єктів, що підтримують клонування. Це дозволяє копіювати об’єкти, не прив’язуючись до їхніх конкретних класів. Зазвичай такий інтерфейс має всього один метод — clone.

Хід роботи

Тема 11: Web crawler (proxy, chain of responsibility, memento, template method, composite, p2p)

Веб-сканер повинен вміти розпізнавати структуру сторінок сайту, переходити за посиланнями, збирати необхідну інформацію про зазначений термін, видаляти не семантичні одиниці (рекламу, об'єкти javascript і т.д.), зберігати знайдені дані у вигляді структурованого набору html файлів вести статистику відвіданих сайтів і метадані.

1. Реалізувати не менше 3-х класів відповідно до обраної теми.

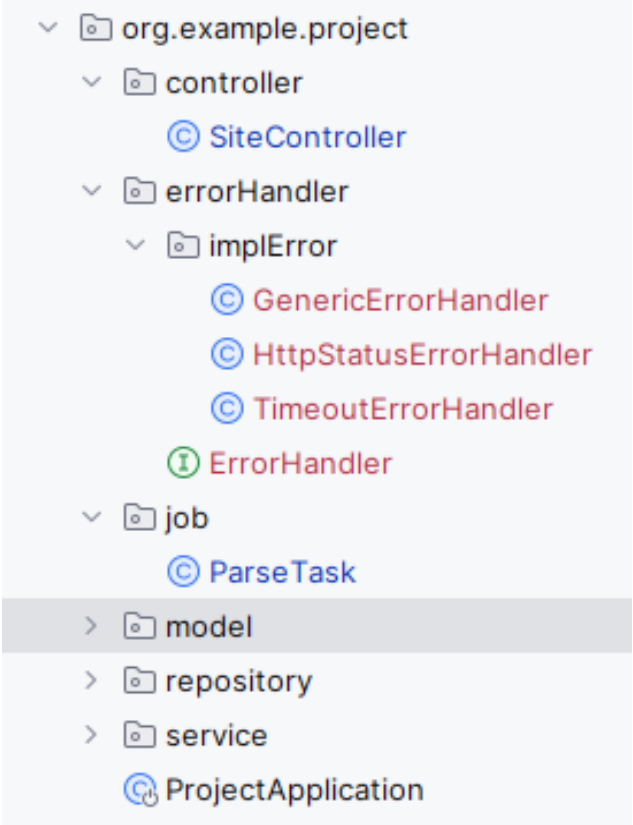

Рис. 2: Структура проекту

У ході лабораторної роботи була реалізована система кравлера, де основним завданням було використання патерну Chain of Responsibility.

2. Реалізація шаблону Chain of Responsibility

Паттерн Chain of Responsibility дозволяє легко відокремити обробники подій від місця їх виникнення і дати можливість гнучкіше настроювати обробку . У нашому прикладі цей патерн застосовується для виявлення помилок у доступі до сайту під час кравленгу. Патерн Chain of Responsibility допомагає зробити код гнучкішим і розширюваним, оскільки всі дії організовані як команди, що можна виконати чи скасувати.

1. Інтерфейс ErrorHandler

```
1  package org.example.project.errorHandler;
2
3
4
   10 usages  3 implementations
5  public interface ErrorHandler {
       2 usages  3 implementations
6      void handleError(Exception e);
7  }
```

Рис. 3: Інтерфейс ErrorHandler

- Оголошує метод handleError(Exception e), який кожен конкретний обробник помилок реалізує відповідно до свого призначення.
- Дозволяє створювати різні реалізації обробників, що будуть спеціалізуватися на обробці певних винятків.

2. Реалізація TimeoutErrorHandler

```
package org.example.project.errorHandler.implError;

import org.example.project.errorHandler.ErrorHandler;

import java.net.SocketTimeoutException;

no usages
public class TimeoutErrorHandler implements ErrorHandler {
    3 usages
    private ErrorHandler nextHandler;

    no usages
    public void setNextHandler(ErrorHandler nextHandler) { this.nextHandler = nextHandler; }

    1 usage
    @Override
    public void handleError(Exception e) {
        if (e instanceof SocketTimeoutException) {
            System.out.println("Handling timeout error: " + e.getMessage());
        } else if (nextHandler != null) {
            nextHandler.handleError(e);
        }
    }
}
```

Рис. 4: Класс TimeoutErrorHandler

- Цей клас обробляє помилки типу SocketTimeoutException.
- Використовує механізм "ланцюжка відповідальностей": якщо поточний обробник не може опрацювати помилку, він передає її далі через nextHandler.

3. Реалізація HttpStatusErrorHandler

```java
package org.example.project.errorHandler.implError;

import org.example.project.errorHandler.ErrorHandler;
import org.jsoup.HttpStatusException;

no usages
public class HttpStatusErrorHandler implements ErrorHandler {
    3 usages
    private ErrorHandler nextHandler;

    no usages
    public void setNextHandler(ErrorHandler nextHandler) { this.nextHandler = nextHandler; }

    2 usages
    @Override
    public void handleError(Exception e) {
        if (e instanceof HttpStatusException) {
            System.out.println("Handling HTTP status error: " + e.getMessage());
        } else if (nextHandler != null) {
            nextHandler.handleError(e);
        }
    }
}
```

Рис. 5: Клас HttpStatusErrorHandler

- Цей клас спеціалізується на обробці помилок HttpStatusException (наприклад, помилки HTTP-кодів 404, 500 тощо).
- Аналогічно, якщо помилка не належить до його сфери відповідальності, вона передається наступному обробнику.

4. Реалізація GenericErrorHandler

```java
package org.example.project.errorHandler.implError;

import org.example.project.errorHandler.ErrorHandler;

no usages
public class GenericErrorHandler implements ErrorHandler {

    no usages
    private ErrorHandler nextHandler;
    2 usages
    @Override
    public void handleError(Exception e) {
        System.out.println("Generic error: " + e.getMessage());
    }
}
```

Рис. 6: Клас GenericErrorHandler

- Загальний обробник, який опрацьовує будь-які помилки, що не були оброблені попередніми обробниками в ланцюжку.
- Завершальна точка в ланцюжку відповідальностей.

Проблема яку вирішує патерн "Chain of responsibility"

Гнучка обробка запитів

Ланцюжок відповідальностей дозволяє розподіляти обробку запитів між різними об'єктами в ланцюзі. Замість створення єдиного обробника для всіх типів запитів, цей патерн дозволяє делегувати завдання відповідному обробнику, що підтримує заданий тип запиту. Якщо поточний об'єкт не може обробити запит, він передає його наступному обробнику в ланцюгу.

Наприклад, у системі краулінгу (як у ParseTask) це дає можливість послідовно перевіряти помилки, такі як мережеві таймаути, проблеми з HTTP-статусами або інші загальні винятки. Завдяки цьому кожен обробник спеціалізується на своєму завданні, а загальний процес обробки залишається добре структурованим і масштабованим.

Зниження зв'язності компонентів

Замість жорсткого зв'язку між клієнтом (запитом) і всіма обробниками, кожен обробник в ланцюгу відповідає лише за свій обсяг роботи і має посилання на наступний обробник. Це зменшує залежність між різними частинами системи та полегшує їхнє тестування та розширення.

У ParseTask, наприклад, додавання нового типу обробки помилок (наприклад, для SSL-помилок) не потребує змін у вже існуючих обробниках. Достатньо створити новий обробник і додати його до ланцюга.

Переваги використання патерну «Chain of responsibility»

Гнучкість і масштабованість:

- Додавання нового обробника вимагає мінімальних змін у коді.
- Легко налаштовувати порядок обробників у ланцюгу.

Зменшення зв'язності:

- Клієнт не залежить від конкретних обробників, що спрощує підтримку і тестування.

Локалізація логіки:

- Кожен обробник відповідає тільки за свій тип запитів, що робить код більш зрозумілим і підтримуваним.

Висновок: У даній лабораторній роботі я реалізувала патерн проєктування "Ланцюжок відповідальностей" (Chain of Responsibility) для системи веб-краулера. Це дозволило організувати обробку помилок у вигляді послідовного ланцюга обробників, кожен з яких відповідає за свій тип винятків (наприклад, таймаути, HTTP-статуси або загальні помилки). Завдяки цьому патерну система отримала гнучкість у налаштуванні та масштабуванні: додавання нового типу обробки тепер можна здійснити без змін у наявному коді.