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 №6

із дисципліни *«Технології розроблення програмного забезпечення»*

Тема: ««Шаблони «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»»

Виконав:

студент групи ІА-23
Пожар Д. Ю.

Перевірив:

Мягкий М.Ю.

Київ 2024

Тема лабораторних робіт:

Варіант 27

Особиста бухгалтерія (state, prototype, decorator, bridge, flyweight, SOA)

Програма повинна бути наочним засобом для ведення особистих фінансів: витрат і прибутку; з можливістю встановлення періодичних витрат / прибутку (зарплата і орендна плата); введення сканованих чеків з відповідними статтями витрат; побудова статистики; експорт/імпорт в Excel, реляційні джерела даних; різні рахунки; ведення єдиного фонду на всі рахунки (всією сім'єю) - на особливі потреби (ремонт, автомобіль, відпустка); можливість введення вкладів / кредитів для контролю банківських рахунків (звірка нарахованих відсотків з необхідними і т.д.).

Завдання:

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми.

Зміст

Короткі теоретичні відомості 4

Реалізація шаблону “Decorator” 5

Висновок 9

Хід роботи

Крок 1. Короткі теоретичні відомості

Шаблон **Abstract Factory** дозволяє створювати сімейства взаємопов'язаних або взаємозалежних об'єктів без вказівки їх конкретних класів. Він забезпечує абстракцію над логікою створення, що робить систему гнучкішою при зміні або розширенні об'єктів. Головною перевагою є підтримка принципу інверсії залежностей, що спрощує заміну цілих груп об'єктів.

Шаблон **Factory Method** визначає інтерфейс для створення об'єктів, дозволяючи підкласам самостійно вирішувати, який клас створювати. Це забезпечує більш гнучку та модульну структуру, адже конкретна реалізація створення об'єктів відокремлена від основного коду, який використовує ці об'єкти. Factory Method дозволяє клієнту працювати лише з абстракціями, не залежачи від деталей реалізації.

Шаблон **Memento** використовується для збереження та відновлення стану об'єкта без порушення його інкапсуляції. Він особливо корисний у системах, де потрібно реалізувати функціональність "відміни" (undo) або "повтору" (redo). Основна ідея полягає у створенні знімків стану об'єкта, які зберігаються у спеціальних об'єктах-згадках (mementos), а зовнішній код працює лише через опосередковані методи.

Шаблон **Observer** дозволяє одному об'єкту (суб'єкту) повідомляти інші об'єкти (спостерігачів) про зміни у своєму стані. Це забезпечує динамічну взаємодію між об'єктами, зберігаючи низький рівень зв'язності. Observer використовується для створення систем, які реагують на події, наприклад, у графічних інтерфейсах чи реалізації патернів публікації та підписки.

Шаблон **Decorator** дозволяє динамічно додавати об'єктам нову поведінку або функціональність, зберігаючи інтерфейс об'єкта. Це досягається шляхом обгортання базового об'єкта в інші об'єкти-декоратори. Decorator сприяє гнучкому розширенню функціоналу без необхідності модифікувати існуючий код, підтримуючи принцип відкритості/закритості (Open/Closed Principle).

Крок 2. Реалізація шаблону "Decorator"

Для реалізації шаблону проєктування "Decorator" було вирішено створити валідатор для транзакцій (Transactions) в системі "Особиста бухгалтерія". Цей шаблон дозволяє створювати систему валідаторів, які динамічно додають додаткові перевірки до основної логіки.

TransactionValidator

Першим кроком є створення інтерфейсу TransactionValidator, який містить метод validate(CreateTransactionDto transactionDto). Це єдина точка взаємодії для всіх валідаторів, що дозволяє забезпечити їх взаємозамінність.

```
public interface TransactionValidator {
    3 usages  3 implementations  new *
    void validate(CreateTransactionDto transactionDto);
}
```

Рис 1. TransactionValidator

BasicTransactionValidator

Далі створюється базовий валідатор BasicTransactionValidator, який реалізує основну логіку перевірки транзакцій, таку як перевірка наявності суми, дати, ідентифікаторів рахунку та категорії. Цей валідатор є ядром системи, яке виконує базові перевірки, необхідні для всіх транзакцій.

```
@Component
public class BasicTransactionValidator implements TransactionValidator {
    3 usages  new *
    @Override
    public void validate(CreateTransactionDto transactionDto) {
        if (transactionDto.getAmount() == null || transactionDto.getAmount() <= 0) {
            throw new IllegalArgumentException("Amount must be greater than 0");
        }
        if (transactionDto.getDate() == null) {
            throw new IllegalArgumentException("Date is required");
        }
        if (transactionDto.getAccountId() == null) {
            throw new IllegalArgumentException("Account ID is required");
        }
        if (transactionDto.getCategoryId() == null) {
            throw new IllegalArgumentException("Category ID is required");
        }
    }
}
```

Рис 2. BasicTransactionValidator

Валідатори AccountExistenceValidator CategoryExistenceValidator

На наступному етапі додаються декоратори, такі як AccountExistenceValidator та CategoryExistenceValidator. Декоратори реалізують той самий інтерфейс TransactionValidator, але приймають у конструкторі інший об'єкт цього ж інтерфейсу, який називається "обгорнутим валідатором". У методі validate кожен декоратор викликає метод валідації обгорнутого валідатора, після чого додає власну специфічну перевірку. Наприклад, AccountExistenceValidator перевіряє, чи існує рахунок у базі даних, використовуючи AccountRepository. Якщо рахунок не знайдено, кидається помилка із відповідним повідомленням. Аналогічно, CategoryExistenceValidator перевіряє існування категорії у базі даних.

```java
@Component
public class AccountExistenceValidator implements TransactionValidator {
    2 usages
    private final TransactionValidator wrappedValidator;
    2 usages
    private final AccountRepository accountRepository;

    new *
    public AccountExistenceValidator(TransactionValidator wrappedValidator, AccountRepository accountRepository) {
        this.wrappedValidator = wrappedValidator;
        this.accountRepository = accountRepository;
    }

    3 usages  new *
    @Override
    public void validate(CreateTransactionDto transactionDto) {
        wrappedValidator.validate(transactionDto);

        if (!accountRepository.existsById(transactionDto.getAccountId())) {
            throw new IllegalArgumentException("Account does not exist");
        }
    }
}
```

Рис 3. Реалізація AccountExistenceValidator

```java
@Component
public class CategoryExistenceValidator implements TransactionValidator {

    2 usages
    private final TransactionValidator wrappedValidator;
    2 usages
    private final ExpenseCategoryRepository categoryRepository;

    new *
    public CategoryExistenceValidator(TransactionValidator wrappedValidator, ExpenseCategoryRepository categoryRepository) {
        this.wrappedValidator = wrappedValidator;
        this.categoryRepository = categoryRepository;
    }

    3 usages  new *
    @Override
    public void validate(CreateTransactionDto transactionDto) {
        wrappedValidator.validate(transactionDto);

        if (!categoryRepository.existsById(transactionDto.getCategoryId())) {
            throw new IllegalArgumentException("Category does not exist");
        }
    }
}
```

Рис 4. Реалізація CategoryExistenceValidator

[Посилання на репозиторій проєкту]

Висновок

У рамках лабораторної роботи було реалізовано функціонал для системи "Особиста бухгалтерія" з використанням шаблону "Decorator". Після ознайомлення з теоретичними відомостями про шаблони, було спроєктовано взаємодію класів для досягнення поставлених функціональних цілей.

Шаблон "Decorator" був застосований для валідації при створенні транзакцій. Було розроблено базовий валідатор, який забезпечує основні перевірки даних транзакції, а також декоратори, що додають додаткову логіку, наприклад, перевірку існування рахунків і категорій у базі даних. Усі валідатори об'єднано в ланцюжок за допомогою декораторів, що гарантує послідовне виконання перевірок.

Отриманий досвід під час виконання роботи показав, як шаблони проєктування можуть допомогти оптимізувати функціонал і підвищити гнучкість системи.