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»”

Виконала:
Студентка групи ІА-24
Сіденко Д.Д.

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 функціонала робочої програми у вигляді класів і їх взаємодій для досягнення конкретних функціональних можливостей.
3. Застосування одного з даних шаблонів при реалізації програми

Зміст

Хід роботи:

Крок 1: Короткі теоретичні відомості

Abstract Factory — це шаблон проектування, який дозволяє створювати інтерфейс для створення групи взаємопов'язаних або залежних об'єктів без вказівки їх конкретних класів. Він дозволяє клієнтському коду використовувати абстракцію замість конкретних класів, що дає можливість замінювати реалізації без змін у клієнтському коді. Наприклад, у програмі, яка працює з різними стилями інтерфейсів для Windows і Mac, абстрактна фабрика дозволяє створювати відповідні елементи інтерфейсу для кожної операційної системи.

Factory Method є шаблоном, який визначає інтерфейс для створення об'єктів, але дозволяє підкласам змінювати тип створюваного об'єкта. Це дозволяє інкапсулювати процес створення об'єктів у конкретних підкласах, що дає змогу керувати їх створенням без зміни основного коду. Наприклад, в системі транспортних засобів фабрика може створювати різні види транспорту, такі як автомобілі чи вантажівки, через конкретні фабрики, що реалізують метод створення об'єкта.

Memento — це шаблон, який дозволяє зберігати стан об'єкта таким чином, щоб пізніше можна було відновити цей стан без порушення інкапсуляції. Цей шаблон корисний у ситуаціях, коли потрібно зберігати історію змін об'єкта та дозволяти відновлення попереднього стану, як у випадку функції скасування в редакторах або відновлення попередніх налаштувань. Наприклад, у текстовому редакторі можна зберігати мемуари для кожного змінення тексту та відновлювати попередні варіанти.

Observer дозволяє встановити залежність між об'єктами типу «один до багатьох», так що коли один об'єкт змінюється, всі його залежні об'єкти отримують повідомлення про зміни і автоматично оновлюються. Цей шаблон використовується в системах, де багато об'єктів повинні реагувати на зміни стану іншого об'єкта, наприклад, у чатах, соціальних мережах або системах сповіщень, де користувачі отримують оновлення про нові події.

Decorator є шаблоном, що дозволяє динамічно додавати нову поведінку об'єктам, обгортаючи їх у нові об'єкти. Завдяки цьому шаблону можна додавати нові функціональності до об'єктів без зміни їхнього коду. Наприклад, у графічних програмах, де є об'єкти типу «Shape», можна додавати додаткові характеристики, такі як колір або товщина лінії, застосовуючи різні декоратори, що змінюють вигляд або властивості об'єкта, не змінюючи його основну структуру.

Крок 2: Реалізація шаблону “Decorator”

Шаблон Decorator є структурним патерном проектування, який дозволяє динамічно додавати поведінку об'єкту, не змінюючи його структуру. Він дає можливість додавати нову функціональність об'єкту під час виконання програми.

Для реалізації шаблону Decorator для валідації даних транзакцій було виконано наступні кроки:

Для початку визначаємо інтерфейс TransactionValidator, який буде реалізовувати базову валідацію.

```
package com.budget.budgetmate.validation;

public interface TransactionValidator {  1 usage  2 implementations  new *
    void validate();  2 usages  2 implementations  new *
}
```

Рис.1 - код інтерфейсу TransactionValidator

Створюємо базовий валідатор, який виконуватиме основну валідацію (наприклад, перевірка на порожній опис і неправильну суму).

```
package com.budget.budgetmate.validation;

import com.budget.budgetmate.dto.request.TransactionDTORequest;

public class BasicTransactionValidator implements TransactionValidator {  3 usages  1 inheritor  new *
    protected final TransactionDTORequest transactionDTORequest;  9 usages

    public BasicTransactionValidator(TransactionDTORequest transactionDTORequest) {  1 usage  new *
        this.transactionDTORequest = transactionDTORequest;
    }

    @Override  2 usages  1 override  new *
    public void validate() {
        if (transactionDTORequest.getDescription() == null || transactionDTORequest.getDescription().trim().isEmpty()) {
            throw new IllegalArgumentException("Опис транзакції не може бути порожнім.");
        }
        if (transactionDTORequest.getAmount() == null || transactionDTORequest.getAmount() <= 0) {
            throw new IllegalArgumentException("Сума транзакції повинна бути більшою за нуль.");
        }
    }
}
```

Рис.2 - код базового валідатору

Декоратор для додавання додаткових перевірок до основного валідатора. Наприклад, перевірка на періодичні транзакції, які мають кінцеву дату, яка не може бути раніше дати транзакції.

```
package com.budget.budgetmate.validation;

import com.budget.budgetmate.dto.request.TransactionDTORequest;

public class PeriodicTransactionValidator extends BasicTransactionValidator {  2 usages  new *
    public PeriodicTransactionValidator(TransactionDTORequest transactionDTORequest) {  1 usage  new *
        super(transactionDTORequest);
    }

    @Override  2 usages  new *
    public void validate() {
        super.validate();
        if (transactionDTORequest.isPeriodic() && transactionDTORequest.getEndDate() != null
                && transactionDTORequest.getTransactionDate().isAfter(transactionDTORequest.getEndDate())) {
            throw new IllegalArgumentException("Дата транзакції перевищує кінцеву дату.");
        }
    }
}
```

Рис.3 - код декоратору

Для створення транзакції, ми можемо по черзі додавати валідацію до основного валідатора. Наприклад, спочатку виконуємо базову валідацію, а потім додаємо перевірку для періодичних транзакцій і тд.

Таким чином, реалізація шаблону Decorator дозволяє додавати нові правила валідації до транзакцій без зміни основної логіки та структури класів, що робить систему гнучкішою і дає змогу легко масштабуватись.

Проєкт можна переглянути у репозиторії:
https://github.com/papiroskada/budget-mate

Висновок: у рамках виконання лабораторної роботи щодо проектування системи "Особиста бухгалтерія" було здійснено реалізацію шаблону проектування Decorator для створення валідації транзакцій. Для цього спочатку було проаналізовано функціональні вимоги системи, а потім створено необхідні класи та інтерфейси. Також завдяки даній лабораторній роботі я була ознайомлена з такими шаблонами проектування, як: «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator».