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

**Технології розроблення програмного забезпечення**

ШАБЛОНИ «Abstract Factory»,
«Factory Method», «Memento»,
«Observer», «Decorator»
Особиста Бухгалтерія

Виконала:
Студентка групи ІА-22
Лахоцька С. О.

Перевірив:
Мягкий М. Ю

Київ 2024

# Зміст

Теоретичні відомості ..... 3

Хід роботи ..... 4

Висновки ..... 4

Вихідний код ..... 6

**Тема**: шаблони «Abstract Factory», «Factory Method», «Memento»,«Observer», «Decorator»

**Мета**: ознайомитись з теоретичними відомостями, розробити частину функціоналу системи з використанням одного із шаблонів проектування.

## Теоретичні відомості

Принципи SOLID є основоположними правилами об’єктно-орієнтованого програмування (ООП) і розробки програмного забезпечення. Вони були запропоновані Робертом Мартіном (Robert C. Martin) для покращення гнучкості, масштабованості та підтримуваності коду. Кожен принцип має свою специфіку і допомагає уникнути типових помилок у проектуванні.

1. Принцип єдиного обов'язку (Single Responsibility Principle - SRP)

Кожен клас повинен мати тільки одну причину для зміни.

Це означає, що клас повинен відповідати лише за одну конкретну функціональність або частину бізнес-логіки. Якщо клас виконує декілька завдань, він стає складним для підтримки і тестування, оскільки зміни в одній його частині можуть вплинути на інші.

2. Принцип відкритості/закритості (Open/Closed Principle - OCP)

Код повинен бути відкритим для розширення, але закритим для зміни.

Це означає, що нова функціональність має додаватися через розширення існуючого коду, а не шляхом його модифікації.

3. Принцип підстановки Ліскoв (Liskov Substitution Principle - LSP)

Об’єкти похідного класу повинні бути замінними на об’єкти базового класу без порушення коректності програми.

Це означає, що підкласи повинні поводитися так само, як і батьківські класи, і не змінювати їхню поведінку.

4. Принцип поділу інтерфейсу (Interface Segregation Principle - ISP)

Клієнти не повинні залежати від інтерфейсів, які вони не використовують.

Це означає, що великий інтерфейс слід розбивати на кілька дрібних, які містять лише необхідні методи для конкретного клієнта.

5. Принцип інверсії залежностей (Dependency Inversion Principle - DIP)

Модулі високого рівня не повинні залежати від модулів низького рівня. Обидва повинні залежати від абстракцій. Абстракції не повинні залежати від деталей. Деталі мають залежати від абстракцій. Цей принцип рекомендує використовувати інтерфейси або абстрактні класи для зв'язку між компонентами програми, щоб уникнути жорсткої залежності.

Шаблон «Abstract Factory» надає механізм для створення груп взаємопов'язаних об'єктів без зазначення їх конкретних класів. Він дозволяє розробникам створювати сімейства об'єктів, забезпечуючи узгодженість та спрощуючи заміну продуктів в майбутньому.

Шаблон «Factory Method» використовується для делегування створення об'єктів підкласам. Це забезпечує більшу гнучкість і розширюваність системи, оскільки підкласи можуть самостійно визначати, які об'єкти створювати.

Шаблон «Memento» орієнтований на збереження та відновлення стану об'єктів без порушення інкапсуляції. Його застосування корисне в програмах, де необхідно реалізувати функціональність скасування дій або історії змін.

Шаблон «Observer» організовує механізм спостереження, що дозволяє одному об'єкту (суб'єкту) повідомляти кілька інших об'єктів (спостерігачів) про зміни в його стані. Цей шаблон підтримує слабку залежність між об'єктами, що спрощує їхню взаємодію.

Шаблон «Decorator» надає можливість додавати нові функціональні можливості об'єктам динамічно. Його перевагою є можливість створювати ієрархію об'єктів із різними рівнями функціональності без необхідності створювати велику кількість підкласів.

## Хід роботи

У цій лабораторній роботі було прийнято рішення використати патерн «Factory Method» для створення різних типів транзакцій. Це дозволило реалізувати чіткий механізм розподілу відповідальності між класами та спростити додавання нових типів транзакцій у майбутньому. Кожна фабрика відповідає за створення конкретного типу транзакції, що забезпечує простоту та передбачуваність коду.

Патерн «Decorator», хоча й ефективний для динамічного додавання нових функцій до об'єктів, не був використаний з кількох причин. Основна причина полягає в тому, що функціонал роботи з транзакціями не передбачає необхідності в модифікації вже створених об'єктів. У цьому проєкті транзакції є статичними після створення, і їх логіка не змінюється в процесі виконання програми.

Крім того, використання «Decorator» створило б додаткову складність в реалізації, адже потрібно було б розробляти обгортки для кожного можливого випадку. Це ускладнило б архітектуру і зробило її менш зрозумілою для підтримки.

Загалом, «Factory Method» краще відповідав вимогам цього завдання, адже він забезпечує гнучкість і розширюваність у створенні об'єктів, без необхідності додавання ієрархій об'єктів, які притаманні «Decorator».
Було реалізовано інтерфейс TransactionFactory та його наслідників, які створюють транзакції поповнення рахунку та транзакції зняття коштів з рахунку. В наслідниках реалізований метод інтерфейсу createTransaction. Кожна реалізація створює різний тип транзакції. (Рис 1.1)

```java
@Component
public class WithdrawalTransactionFactory implements TransactionFactory {

    @Override 1 usage
    public Transaction createTransaction(String name, BigDecimal amount, String description, Category category) {
        return new Transaction.Builder()
                .name(name)
                .amount(amount)
                .description(description)
                .category(category)
                .transactionType(TransactionTypes.WITHDRAWAL)
                .build();
    }
}
```

Рисунок 1.1 – Приклад реалізації інтерфейсу TransactionFactory

Оскільки в коді ми часто створюємо різноманітні транзакції, цей патерн проектування стане в нагоді під час створення транзакцій в залежності від їхнього типу. Наприклад, використання готових фабрик у сервісі TransactionService (Рис 1.2)

```java
public Transaction createTransaction(Long accountId, String name, String description, BigDecimal amount,  1 usage
                                      TransactionTypes type, Category category) {

    TransactionFactory factory = switch (type) {
        case DEPOSIT -> new DepositTransactionFactory();
        case WITHDRAWAL -> new WithdrawalTransactionFactory();
    };

    Transaction transaction = factory.createTransaction(name, amount, description, category);

    Account account = accountRepository.findById(accountId).orElseThrow(() -> new IllegalArgumentException("Account not found"));
    account.addTransaction(transaction);
    accountRepository.save(account);

    return transactionRepository.save(transaction);
}
```

Рисунок 1.2. Використання фабрик під час створення транзакції

**Висновки**

У цій лабораторній роботі було реалізовано функціонал створення різноманітних транзакцій за допомогою шаблону проектування Factory Method. Було досліджено патерни проектування «Abstract Factory», «Memento», «Observer» та «Decorator», використання яких в подальшому допоможе ефективно та якісно розробляти будь-які системи.

# Вихідний код

```
package org.example.accounting.api.service.factory;

import org.example.accounting.api.model.Category;
import org.example.accounting.api.model.Transaction;

import java.math.BigDecimal;

public interface TransactionFactory {
    Transaction createTransaction(String name, BigDecimal amount, String
description, Category category);
}
```

```
package org.example.accounting.api.service.factory;

import org.example.accounting.api.model.Category;
import org.example.accounting.api.model.Transaction;
import org.example.accounting.api.model.TransactionTypes;
import org.springframework.stereotype.Component;

import java.math.BigDecimal;

@Component
public class DepositTransactionFactory implements TransactionFactory {
    @Override
    public Transaction createTransaction(String name, BigDecimal amount, String
description, Category category) {
        return new Transaction.Builder()
                .name(name)
                .amount(amount)
                .description(description)
                .category(category)
                .transactionType(TransactionTypes.DEPOSIT)
                .build();
    }
}
```

```
package org.example.accounting.api.service.factory;

import org.example.accounting.api.model.Category;
import org.example.accounting.api.model.Transaction;
import org.example.accounting.api.model.TransactionTypes;
import org.springframework.stereotype.Component;

import java.math.BigDecimal;

@Component
public class WithdrawalTransactionFactory implements TransactionFactory {

    @Override
    public Transaction createTransaction(String name, BigDecimal amount, String
description, Category category) {
        return new Transaction.Builder()
                .name(name)
                .amount(amount)
                .description(description)
                .category(category)
                .transactionType(TransactionTypes.WITHDRAWAL)
```

```
                .build();
    }
}
```

```
package org.example.accounting.api.service;

import org.example.accounting.api.model.Account;
import org.example.accounting.api.model.Category;
import org.example.accounting.api.model.Transaction;
import org.example.accounting.api.model.TransactionTypes;
import org.example.accounting.api.repository.AccountRepository;
import org.example.accounting.api.repository.TransactionRepository;
import org.example.accounting.api.service.factory.DepositTransactionFactory;
import org.example.accounting.api.service.factory.TransactionFactory;
import org.example.accounting.api.service.factory.WithdrawalTransactionFactory;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;

@Service
public class TransactionService {
    private final TransactionRepository transactionRepository;
    private final AccountRepository accountRepository;

    public TransactionService(TransactionRepository transactionRepository,
AccountRepository accountRepository) {
        this.transactionRepository = transactionRepository;
        this.accountRepository = accountRepository;
    }

    public Transaction createTransaction(Long accountId, String name, String
description, BigDecimal amount,
                                         TransactionTypes type, Category
category) {

        TransactionFactory factory = switch (type) {
            case DEPOSIT -> new DepositTransactionFactory();
            case WITHDRAWAL -> new WithdrawalTransactionFactory();
        };

        Transaction transaction = factory.createTransaction(name, amount,
description, category);

        Account account = accountRepository.findById(accountId).orElseThrow(() -
> new IllegalArgumentException("Account not found"));
        account.addTransaction(transaction);
        accountRepository.save(account);

        return transactionRepository.save(transaction);
    }
}
```