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 №8
Технології розроблення програмного забезпечення
ШАБЛОНИ «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»
Особиста Бухгалтерія

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

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: шаблони «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»

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

Теоретичні відомості

Розробка програмного забезпечення для масштабних корпоративних додатків потребує використання спеціальних шаблонів роботи з базами даних. Одним із таких є шаблон "Active Record", який об'єднує дані та поведінку в одному об'єкті. Цей підхід передбачає, що кожен об'єкт виступає "обгорткою" для рядка бази даних, а логіка доступу до даних інтегрована в сам об'єкт. Хоча цей шаблон є простим і популярним, особливо в Ruby on Rails, зі збільшенням складності системи логіку запитів часто переносять в окремі об'єкти.

Інший підхід – шаблон "Table Data Gateway". У ньому взаємодія з базою даних виконується через окремі класи для кожного типу даних. Цей підхід забезпечує гнучкість і тестованість, розділяючи логіку запитів і збереження даних. Однак повторення коду для шлюзів часто вимагає абстракції через базові класи. Шаблон також відомий під назвою "репозиторій".

Шаблон "Data Mapping" зосереджується на проблемі перетворення об’єктів даних у формат реляційної бази. Відображення дозволяє гармонізувати типи даних і забезпечує взаємодію об’єктів з джерелом даних. Наприклад, маппер може відповідати за зіставлення властивостей об’єкта з колонками таблиці.

Шаблон "Composite" допомагає представляти об’єкти у вигляді дерева, що спрощує роботу з ієрархіями. Наприклад, форма в інтерфейсі може містити різні дочірні елементи, які можна обробляти як однорідні об’єкти. Це дозволяє легко масштабувати і додавати нові типи компонентів, хоча загальний дизайн класів може стати занадто складним.

Шаблон "Flyweight" вирішує проблему надмірного використання пам’яті за рахунок поділу об’єктів між різними частинами програми. Наприклад, у грі можна створити один об’єкт для кольору чи текстури, а зовнішні характеристики, як координати чи швидкість, винести в контекст. Це економить пам'ять, хоча може ускладнити код і збільшити витрати на обчислення.

Шаблон "Interpreter" забезпечує створення граматики та інтерпретатора для обраної мови, використовуючи деревоподібні структури. Це спрощує додавання нових правил і змін у граматиці, хоча для складних систем кількість класів може зрости до надмірної.

Шаблон "Visitor" дозволяє визначати операції для об’єктів без зміни їхньої структури. Це зручно для додавання нових операцій, наприклад, для експорту графів у XML. Проте кожен новий елемент ієрархії вимагає модифікації всіх існуючих відвідувачів, що може ускладнити підтримку.

Хід роботи

Для виконання даної лабораторної роботи було реалізовано патерн "FLYWEIGHT" для оптимізації роботи з часто повторюваними об'єктами. К транзакцій можуть багаторазово використовуватися у різних фінансових операціях. Цей підхід дозволяє значно скоротити використання пам’яті, зменшуючи дублювання об'єктів із однаковими властивостями, такими як назва категорії.

Для цього було створено клас CategoryFlyweight, що містить спільний стан (ім'я категорії), і фабрику Flyweight (CategoryFlyweightFactory), яка відповідає за створення та повторне використання таких об'єктів. Усі категорії транзакцій тепер отримують свої властивості через фабрику, що забезпечує централізоване управління екземплярами. Унікальні властивості транзакцій залишаються поза межами Flyweight і зберігаються у моделі транзакцій. (Рис 1.1)

```
public class CategoryFlyweight {  7 usages
    private final String name;  2 usages

    public CategoryFlyweight(String name) {  1 usage
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Рисунок 1.1 – Код класу CategoryFlyweight

Під час інтеграції в існуючу архітектуру, було оновлено модель Category для підтримки Flyweight і додано у сервіс транзакцій логіку роботи з фабрикою. (Рис 1.2)

```java
@Transactional  2 usages
public Transaction addTransaction(Transaction transaction, Long accountId) {
    var flyweight = categoryFactory.getFlyweight(transaction.getCategory().getName());
    transaction.getCategory().setFlyweight(flyweight);

    var account = accountRepository.findById(accountId).orElseThrow(IllegalArgumentException::new);
    var balance = switch (transaction.getTransactionType()) {
        case DEPOSIT -> account.getBalance().add(transaction.getAmount());
        case WITHDRAWAL -> account.getBalance().subtract(transaction.getAmount());
    };
    account.setBalance(balance);
    account.addTransaction(transaction);
    accountRepository.save(account);

    return transaction;
}
```

Рис 1.2. – Використання FlyweightFactory під час створення транзакції

Ця оптимізація знижує навантаження на пам’ять, особливо у системах із великим обсягом даних, таких як фінансові системи, де категорії транзакцій часто повторюються. Патерн "FLYWEIGHT" забезпечує баланс між продуктивністю й ефективністю використання ресурсів, що критично важливо для високонавантажених REST API застосунків.

Висновки

У цій лабораторній роботі було оптимізовано роботу з категоріями транзакцій за допомогою патерну FLYWEIGHT. Також було досліджено шаблони Composite, Interpreter та Visitor. Це дозволило зрозуміти, як ефективно організувати роботу з ієрархічними структурами, оптимізувати використання пам’яті, забезпечити гнучке визначення правил для роботи з даними та додавати нові операції без зміни існуючого коду.

Вихідний код

```
// file: src/main/java/org/example/accounting/api/service/AccountService.java
package org.example.accounting.api.service;

import jakarta.transaction.Transactional;
import org.example.accounting.api.model.*;
import org.example.accounting.api.repository.AccountRepository;
import org.example.accounting.api.repository.PeriodicTransactionRepository;
import org.example.accounting.api.repository.TransactionRepository;
import org.example.accounting.api.repository.UserRepository;
import org.example.accounting.api.service.flyweight.CategoryFlyweightFactory;
import org.example.accounting.api.service.strategy.CsvReportGenerator;
import org.example.accounting.api.service.strategy.XlsReportGenerator;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class AccountService {


    private final AccountRepository accountRepository;
    private final TransactionRepository transactionRepository;
    private final PeriodicTransactionRepository periodicTransactionRepository;
    private final UserRepository userRepository;
    private final CategoryFlyweightFactory categoryFactory;


    public AccountService(AccountRepository accountRepository,
TransactionRepository transactionRepository, PeriodicTransactionRepository
periodicTransactionRepository, UserRepository userRepository,
CategoryFlyweightFactory categoryFactory) {
        this.accountRepository = accountRepository;
        this.transactionRepository = transactionRepository;
        this.periodicTransactionRepository = periodicTransactionRepository;
        this.userRepository = userRepository;
        this.categoryFactory = categoryFactory;
    }

    public Optional<Account> findById(Long id) {
        return accountRepository.findById(id);
    }

    public List<Account> findByUsername(String username) {
        return accountRepository.findByUserUsername(username);
    }

    public List<Account> findAll() {
        return accountRepository.findAll();
    }

    @Transactional
```

```
    public Account create(Account account, String username) {
        var user =
userRepository.findByUsername(username).orElseThrow(IllegalArgumentException::ne
w);
        user.addAccount(account);
        return accountRepository.save(account);
    }

    @Transactional
    public List<Transaction> getTransactions(Long accountId) {
        return
accountRepository.findById(accountId).map(Account::getTransactions).map(List::co
pyOf).orElseThrow(IllegalStateException::new);
    }

    @Transactional
    public Transaction addTransaction(Transaction transaction, Long accountId) {
        var flyweight =
categoryFactory.getFlyweight(transaction.getCategory().getName());
        transaction.getCategory().setFlyweight(flyweight);

        var account =
accountRepository.findById(accountId).orElseThrow(IllegalArgumentException::new)
;
        var balance = switch (transaction.getTransactionType()) {
            case DEPOSIT -> account.getBalance().add(transaction.getAmount());
            case WITHDRAWAL ->
account.getBalance().subtract(transaction.getAmount());
        };
        account.setBalance(balance);
        account.addTransaction(transaction);
        accountRepository.save(account);

        return transaction;
    }

    @Transactional
    public PeriodicTransaction addPeriodicTransaction(PeriodicTransaction
transaction, Long accountId) {
        var account =
accountRepository.findById(accountId).orElseThrow(IllegalArgumentException::new)
;
        account.addPeriodicTransaction(transaction);
        accountRepository.save(account);
        return transaction;
    }


    @Transactional
    public Resource exportStats(Long accountId, ExportType exportType) {
        var account =
accountRepository.findById(accountId).orElseThrow(IllegalArgumentException::new)
;
        var context = new ExportContext();
        switch (exportType) {
            case CSV -> context.setStrategy(new CsvReportGenerator());
            case XLS -> context.setStrategy(new XlsReportGenerator());
            default -> throw new IllegalArgumentException("Invalid export
type");
        }
        return context.exportStats(account);
    }

}
```

```
// file:
src/main/java/org/example/accounting/api/service/flyweight/CategoryFlyweightFact
ory.java
package org.example.accounting.api.service.flyweight;

import java.util.HashMap;
import java.util.Map;

public class CategoryFlyweightFactory {
    private final Map<String, CategoryFlyweight> flyweights = new HashMap<>();

    public CategoryFlyweight getFlyweight(String name) {
        if (!flyweights.containsKey(name)) {
            flyweights.put(name, new CategoryFlyweight(name));
        }
        return flyweights.get(name);
    }

    public int getFlyweightCount() {
        return flyweights.size();
    }
}


// file:
src/main/java/org/example/accounting/api/service/flyweight/CategoryFlyweight.jav
a
package org.example.accounting.api.service.flyweight;

public class CategoryFlyweight {
    private final String name;

    public CategoryFlyweight(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```