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 №7
Технології розроблення програмного забезпечення
ШАБЛОНИ «MEDIATOR», «FACADE»,
«BRIDGE», «TEMPLATE METHOD»
Особиста Бухгалтерія

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

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: шаблони «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD»
Мета: ознайомитись з теоретичними відомостями, розробити частину функціоналу системи з використанням одного із шаблонів проектування.

Теоретичні відомості

Принципи проєктування відіграють ключову роль у створенні якісного програмного забезпечення. Принцип Don't Repeat Yourself (DRY) закликає уникати повторень у коді, оскільки це спрощує його читання, зменшує ймовірність появи помилок і полегшує внесення змін. Код без повторень є компактнішим і зрозумілішим, а помилки виправляються ефективніше, адже їх не потрібно шукати в різних місцях. Інший принцип – Keep it Simple, Stupid! (KISS) – наголошує на важливості простоти. Системи, побудовані з невеликих простих компонентів, працюють надійніше та легше піддаються обслуговуванню, ніж складні монолітні структури. Такий підхід сприяє створенню зрозумілого та функціонального коду, який легко сприймати і підтримувати.

Принцип You Only Load It Once! (YOLO) наголошує на тому, що ініціалізаційні змінні варто завантажувати один раз під час запуску програми. Це дозволяє уникнути зайвих операцій зчитування і прискорити роботу системи. Закон Парето, відомий як правило 80/20, акцентує на тому, що більшість результатів можна досягти, зосередившись на невеликій частині зусиль. Наприклад, 80% помилок у програмі можна виправити, усунувши лише 20% багів. Нарешті, принцип You Ain’t Gonna Need It (YAGNI) закликає відмовлятися від зайвої функціональності, яка може ніколи не знадобитися. Це зменшує складність системи, знижує витрати часу і ресурсів.

Шаблони проєктування допомагають організувати структуру програми. Шаблон "MEDIATOR" використовується для координації взаємодії між компонентами через окремий об’єкт, що дозволяє зменшити їх взаємозалежність. Це схоже на роботу диспетчера в аеропорту, який координує літаки, щоб уникнути хаосу. Водночас, надмірна кількість логіки в посереднику може ускладнити його підтримку.

Шаблон "FACADE" пропонує створення єдиного інтерфейсу для доступу до складної підсистеми. Він спрощує використання компонентів і приховує їхню складність. Наприклад, співробітник служби підтримки магазину виступає фасадом для клієнта, спрощуючи взаємодію з внутрішніми системами. Втім, фасад може стати надто громіздким, якщо в ньому зосереджується занадто багато функцій.

Шаблон "BRIDGE" дозволяє відокремлювати інтерфейс від його реалізації, що спрощує розширення системи. Наприклад, можна мати різні типи фігур і кольорів без створення безлічі підкласів для їх комбінацій. Проте додаткові класи, які вводяться в процесі реалізації, можуть ускладнити код.

Шаблон "TEMPLATE METHOD" дозволяє визначити загальний алгоритм у базовому класі, залишаючи реалізацію окремих кроків підкласам. Це схоже на будівництво типового будинку, де основні етапи стандартні, але є можливість

додати унікальні елементи. Хоча шаблон полегшує повторне використання коду, він також може ускладнити підтримку, якщо алгоритм стає занадто деталізованим.

Хід роботи

У цій лабораторній роботі було реалізовано функціонал збору статистики транзакцій у файли різного формату за допомогою патерну «Template Method». Це дозволило стандартизувати процес генерації звітів, забезпечуючи чітку структуру та зменшуючи дублювання коду. Основна перевага такого підходу полягає у визначенні загального алгоритму в базовому класі та делегуванні реалізації окремих кроків підкласам. Це спростило додавання нових форматів звітів і зробило код більш зрозумілим.

Патерн «Bridge» не був використаний, оскільки його основна мета — відокремлення інтерфейсу від реалізації для забезпечення гнучкості та розширюваності системи — не була критичною в контексті цієї задачі. У нашому випадку немає необхідності в одночасній зміні різних аспектів, наприклад, типів інтерфейсів і способів реалізації, як це потрібно у «Bridge». До того ж, використання цього патерну могло б ускладнити структуру через додавання додаткових абстракцій, що не виправдано для відносно прямолінійного процесу генерації звітів.

Отже, «Template Method» став найкращим вибором, оскільки він дозволяє створювати стандартизовані процеси для різних форматів звітів без надмірного ускладнення структури коду, чого не вимагав «Bridge». У базовому класі ReportGenerator визначено структуру процесу генерації звіту: ініціалізація, додавання заголовків, запис транзакцій та завершення. Реалізації CsvReportGenerator і XlsReportGenerator забезпечують конкретні механізми роботи для кожного формату, наприклад, використання PrintWriter для CSV або бібліотеки Apache POI для XLS. Такий підхід спрощує додавання нових форматів звітів і зменшує дублювання коду.(Рис 1.1)

```java
public abstract class ReportGenerator {  1 usage  2 inheritors

    public final Resource generate(Account account) {  no usages
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

        try {
            startReport(outputStream);
            writeHeader(account, outputStream);
            writeTransactions(account.getTransactions(), outputStream);
            endReport(outputStream);
        } catch (Exception e) {
            throw new RuntimeException("Error generating report", e);
        }

        return new ByteArrayResource(outputStream.toByteArray());
    }

    protected abstract void startReport(ByteArrayOutputStream outputStream) throws Exception;  1 usage  2 implementations

    protected abstract void writeHeader(Account account, ByteArrayOutputStream outputStream) throws Exception;  1 usage  2 implementations

    protected abstract void writeTransactions(List<Transaction> transactions, ByteArrayOutputStream outputStream) throws Exception;

    protected abstract void endReport(ByteArrayOutputStream outputStream) throws Exception;  1 usage  2 implementations
}
```

Рисунок 1.1 – Код абстрактного класу ReportGenerator

Реалізація звітів за допомогою патерну Template Method дозволяє стандартизувати процес генерації звітів, забезпечуючи чітку структуру та мінімізуючи дублювання коду. Загальний алгоритм, визначений у базовому класі ReportGenerator, описує послідовність кроків: початок звіту, запис заголовків, обробка транзакцій та завершення. Це спрощує розуміння процесу та гарантує, що всі звіти дотримуються єдиного підходу. (Рис 1.2)

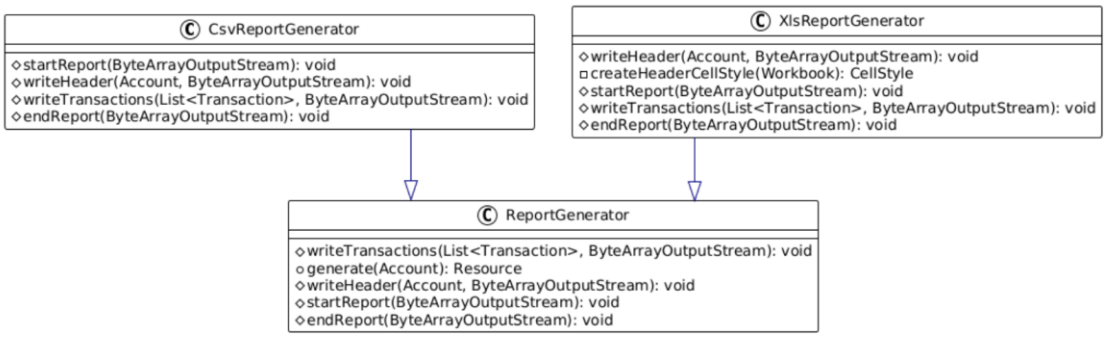

Рисунок 1.2. Діаграма класів генераторів звітів

Висновки

У цій лабораторній роботі було реалізовано функціонал збору статистики транзакцій у файли різного формату за допомогою патерну Template Method. Було розглянуто шаблони Mediator, Facade та Bridge. Знання цих патернів може бути використано для організації взаємодії між компонентами системи, спрощення доступу до складних підсистем, відокремлення інтерфейсу від реалізації та створення загальних алгоритмів у базових класах.

Вихідний код

```
// file:
src/main/java/org/example/accounting/api/service/strategy/CsvReportGenerator.jav
a
package org.example.accounting.api.service.strategy;

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

import java.io.ByteArrayOutputStream;
import java.io.PrintWriter;
import java.util.List;

public class CsvReportGenerator extends ReportGenerator {

    @Override
    protected void startReport(ByteArrayOutputStream outputStream) {
    }

    @Override
    protected void writeHeader(Account account, ByteArrayOutputStream
outputStream) {
        try (PrintWriter writer = new PrintWriter(outputStream)) {
            String[] headers = {"ID", "Name", "Description", "Amount",
"Timestamp", "Account", "Category", "Transaction Type"};
            writer.println(String.join(",", headers));
            writer.flush();
        }
    }

    @Override
    protected void writeTransactions(List<Transaction> transactions,
ByteArrayOutputStream outputStream) {
        try (PrintWriter writer = new PrintWriter(outputStream)) {
            for (Transaction transaction : transactions) {
                String line = String.format(
                        "%s,%s,%s,%s,%s,%s,%s,%s",
                        transaction.getId() != null ? transaction.getId() : "",
                        transaction.getName() != null ? transaction.getName() :
"N/A",
                        transaction.getDescription() != null ?
transaction.getDescription() : "N/A",
                        transaction.getAmount() != null ?
transaction.getAmount().toString() : "0.00",
                        transaction.getTimestamp() != null ?
transaction.getTimestamp().toString() : "N/A",
                        transaction.getAccount() != null &&
transaction.getAccount().getId() != null ? transaction.getAccount().getId() :
"N/A",
                        transaction.getCategory() != null &&
transaction.getCategory().getId() != null ? transaction.getCategory().getId() :
"N/A",
                        transaction.getTransactionType() != null ?
transaction.getTransactionType().toString() : "N/A"
                );
                writer.println(line);
            }
            writer.flush();
        }
    }
```

```
    @Override
    protected void endReport(ByteArrayOutputStream outputStream) {
    }
}


// file:
src/main/java/org/example/accounting/api/service/strategy/ReportGenerator.java
package org.example.accounting.api.service.strategy;

import org.example.accounting.api.model.Account;
import org.example.accounting.api.model.Transaction;
import org.springframework.core.io.ByteArrayResource;
import org.springframework.core.io.Resource;

import java.io.ByteArrayOutputStream;
import java.util.List;

public abstract class ReportGenerator {

    public final Resource generate(Account account) {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

        try {
            startReport(outputStream);
            writeHeader(account, outputStream);
            writeTransactions(account.getTransactions(), outputStream);
            endReport(outputStream);
        } catch (Exception e) {
            throw new RuntimeException("Error generating report", e);
        }

        return new ByteArrayResource(outputStream.toByteArray());
    }

    protected abstract void startReport(ByteArrayOutputStream outputStream)
throws Exception;

    protected abstract void writeHeader(Account account, ByteArrayOutputStream
outputStream) throws Exception;

    protected abstract void writeTransactions(List<Transaction> transactions,
ByteArrayOutputStream outputStream) throws Exception;

    protected abstract void endReport(ByteArrayOutputStream outputStream) throws
Exception;
}


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

import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.ss.usermodel.*;
import org.example.accounting.api.model.Account;
import org.example.accounting.api.model.Transaction;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.List;

public class XlsReportGenerator extends ReportGenerator {
```

```
    @Override
    protected void startReport(ByteArrayOutputStream outputStream) {
    }

    @Override
    protected void writeHeader(Account account, ByteArrayOutputStream
outputStream) throws IOException {
        Workbook workbook = new HSSFWorkbook();
        Sheet sheet = workbook.createSheet("Transactions");

        Row headerRow = sheet.createRow(0);
        String[] headers = {"ID", "Name", "Description", "Amount", "Timestamp",
"Account", "Category", "Transaction Type"};

        for (int i = 0; i < headers.length; i++) {
            Cell cell = headerRow.createCell(i);
            cell.setCellValue(headers[i]);
            cell.setCellStyle(createHeaderCellStyle(workbook));
        }

        workbook.write(outputStream);
    }

    @Override
    protected void writeTransactions(List<Transaction> transactions,
ByteArrayOutputStream outputStream) throws IOException {
        Workbook workbook = new HSSFWorkbook();
        Sheet sheet = workbook.getSheetAt(0);

        int rowIdx = 1;
        for (Transaction transaction : transactions) {
            Row row = sheet.createRow(rowIdx++);

            row.createCell(0).setCellValue(transaction.getId() != null ?
transaction.getId() : 0);
            row.createCell(1).setCellValue(transaction.getName() != null ?
transaction.getName() : "");
            row.createCell(2).setCellValue(transaction.getDescription() != null
? transaction.getDescription() : "");
            row.createCell(3).setCellValue(transaction.getAmount() != null ?
transaction.getAmount().toString() : "0.00");
            row.createCell(4).setCellValue(transaction.getTimestamp() != null ?
transaction.getTimestamp().toString() : "");
            row.createCell(5).setCellValue(transaction.getAccount() != null ?
transaction.getAccount().getId() : 0);
            row.createCell(6).setCellValue(transaction.getCategory() != null ?
transaction.getCategory().getId() : 0);
            row.createCell(7).setCellValue(transaction.getTransactionType() !=
null ? transaction.getTransactionType().toString() : "");
        }

        workbook.write(outputStream);
    }

    @Override
    protected void endReport(ByteArrayOutputStream outputStream) {
    }

    private CellStyle createHeaderCellStyle(Workbook workbook) {
        CellStyle style = workbook.createCellStyle();
        Font font = workbook.createFont();
        font.setBold(true);
        style.setFont(font);
        return style;
  }
```