Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №8

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Шаблони «Composite», «Flyweight», «Interpreter», «Visitor»»

Варіант №15

Виконав:
студент групи ІА-23
Лядський Д.С.

Перевірив:
Мягкий М. Ю.

Київ 2024

Зміст

Тема

Шаблони «Composite», «Flyweight», «Interpreter», «Visitor»

Мета

Метою лабораторної роботи є вивчення та застосування шаблонів проектування Composite, Flyweight, Interpreter та Visitor. В рамках роботи студенти ознайомлюються з особливостями цих шаблонів, їхньою структурою, призначенням та реалізацією в програмуванні. Завдання полягає у розробці програмних рішень з використанням зазначених шаблонів для вирішення конкретних задач, таких як побудова складних деревоподібних структур, оптимізація пам'яті при роботі з великими об'ємами даних, інтерпретація складних виразів та реалізація різних операцій над об'єктами без зміни їх класів.

Короткі теоретичні відомості

Шаблон «Active Record» (Активний запис)

Призначення:

Цей шаблон поєднує в одному об’єкті і дані, і логіку взаємодії з базою даних. Об'єкт є своєрідною обгорткою для одного рядка з бази даних або уявлення. У ньому зберігаються властивості, що відповідають стовпчикам бази даних, а також методи, які дозволяють зчитувати, змінювати, видаляти та зберігати дані.

Особливості шаблону:

Найпростіший для розуміння та використання.

Вся логіка доступу до даних знаходиться в об’єкті-сутності.

Підходить для додатків із відносно простою логікою взаємодії з БД.

Недоліки шаблону:

Якщо кількість запитів і логіка ускладнюється, клас, що використовує шаблон Active Record, стає перевантаженим.

Логіка взаємодії з БД змішується з бізнес-логікою, що може призвести до поганої масштабованості.

Приклад використання:

Active Record використовується у фреймворках на кшталт Ruby on Rails та деяких ORM (Object Relational Mapping), які автоматично відображають рядки таблиць на об'єкти класів.

Шаблон «Table Data Gateway» (Шлюз до даних)

Призначення:

У цьому шаблоні взаємодія з базою даних зосереджена в окремих класах. Для кожної таблиці БД створюється відповідний клас, який містить усі запити (SELECT, INSERT, UPDATE, DELETE). Це дозволяє відокремити логіку доступу до даних від бізнес-логіки програми.

Особливості шаблону:

Запити до бази даних зосереджені в одному місці, що робить код легшим для тестування та супроводу.

Один клас обробляє всі запити для певної таблиці.

Часто реалізується через патерн Repository, коли бізнес-логіка працює з об'єктами репозиторію замість напряму взаємодії з БД.

Недоліки шаблону:

Код доступу до БД може дублюватися для різних таблиць, тому зазвичай він абстрагується в базовий клас.

Залежність від SQL-запитів може ускладнити підтримку в разі зміни схеми БД.

Приклад використання:

У Spring Data JPA клас репозиторію є прикладом Table Data Gateway. Він ізолює операції над БД і дозволяє розробнику зосередитися на бізнес-логіці.

Шаблон «Data Mapping» (Відображення даних)

Призначення:

Шаблон вирішує проблему перетворення даних між об'єктами додатка і реляційною базою даних. Замість того щоб об'єкт напряму взаємодіяв з таблицею БД, маппер відповідає за перенесення даних між ними.

Особливості шаблону:

Маппер знає про структуру таблиці бази даних (колонки) і об'єкта програми (поля).

Логіка взаємодії з БД і бізнес-логіка чітко розділені.

Гнучкість у роботі з різними джерелами даних.

Недоліки шаблону:

Наявність додаткових класів маперів може ускладнювати систему.

Потрібно ретельно підтримувати узгодженість між моделлю даних і базою даних.

Приклад використання:

ORM-бібліотеки, такі як Hibernate або MyBatis, використовують патерн Data Mapper для відображення об'єктів у реляційні таблиці.

Шаблон «Composite» (Компонувальник)

Призначення:

Цей шаблон дозволяє створювати дерево об'єктів для представлення ієрархічної структури «частина-ціле». Усі об'єкти у дереві обробляються уніфіковано, незалежно від того, чи це листовий об’єкт, чи складний (композитний) об'єкт.

Проблема:

Коли необхідно працювати зі складною структурою даних, що має вкладеність (наприклад, ієрархії товарів або структур командування).

Рішення:

Реалізується загальний інтерфейс для всіх об'єктів у дереві. Композитний об'єкт містить посилання на свої дочірні елементи і делегує їм операції.

Переваги:

Уніфікований підхід до обробки об'єктів на всіх рівнях ієрархії.

Полегшує роботу зі складними структурами даних.

Недоліки:

Загальний інтерфейс може стати занадто універсальним, що призводить до надлишкового коду.

Приклад використання:

Побудова UI-компонентів, де вкладені об'єкти (наприклад, кнопки, текстові поля) обробляються у складі батьківського об'єкта (форми).

Шаблон «Flyweight» (Легковаговик)

Призначення:

Flyweight дозволяє зменшити використання оперативної пам'яті шляхом поділу однакових об'єктів між різними частинами програми. Шаблон розділяє стан об'єкта на внутрішній (універсальний для багатьох об'єктів) і зовнішній (контекст застосування).

Проблема:

Велика кількість об'єктів із повторюваними даними призводить до перевитрати пам'яті.

Рішення:

Зберігання загальних даних у єдиному об'єкті (внутрішній стан), а контекст застосування передається ззовні.

Переваги:

Знижує використання пам’яті.

Підходить для роботи з великою кількістю дрібних об'єктів.

Недоліки:

Збільшує витрати на обчислення для управління контекстом.

Ускладнює код програми через поділ стану.

Приклад використання:

Реалізація графічних редакторів або ігор, де спрайти зображень є однаковими, але їх положення чи розмір відрізняються.

Шаблон «Interpreter» (Інтерпретатор)

Призначення:

Шаблон дозволяє створювати інтерпретатор для мови, що визначена у вигляді граматики. Пропозиції цієї мови подаються як абстрактне синтаксичне дерево, де кожен вузол інтерпретується відповідно до контексту.

Проблема:

Необхідно вирішити задачу інтерпретації складних виразів або пошуку за певною граматикою.

Рішення:

Розбиття мови на термінальні й нетермінальні вирази, кожен з яких реалізує операцію інтерпретації.

Переваги:

Гнучкість у розширенні граматики.

Спрощує обробку виразів із рекурсивною структурою.

Недоліки:

Велика кількість класів для кожного правила граматики.

Підходить для простих і невеликих мов.

Приклад використання:

Інтерпретатори регулярних виразів або обчислення математичних виразів у калькуляторах.

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

Обрана тема

15 E-mail клієнт (singleton, builder, decorator, template method, interpreter, SOA)

Поштовий клієнт повинен нагадувати функціонал поштових програм Mozilla Thunderbird, The Bat і т.д. Він повинен сприймати і коректно обробляти pop3/smtp/imap протоколи, мати функції автонастройки основних поштових провайдерів для України (gmail, ukr.net, i.ua), розділяти повідомлення на папки/категорії/важливість, зберігати чернетки незавершених повідомлень, прикріплювати і обробляти прикріплені файли.

Хід роботи

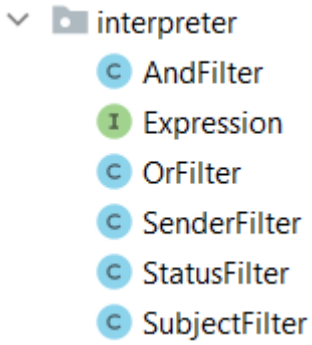

Рисунок 1. Структура шаблона Interpreter

Реалізація патерну Interpreter:

```
14 usages   5 implementations
public interface Expression {
    5 usages   5 implementations
    boolean interpret(EmailMessage email);
}
```

Рисунок 2. Інтерфейс Expression

Інтерфейс Expression: Це базовий інтерфейс для всіх фільтрів (виразів), що реалізують метод interpret(). Це є основною частиною патерну, оскільки кожен фільтр є інтерпретатором, який оцінює, чи відповідає певне повідомлення заданим умовам.

```
3 usages
public class SenderFilter implements Expression {
    2 usages
    private String sender;

    3 usages
    public SenderFilter(String sender) { this.sender = sender; }

    5 usages
    @Override
    public boolean interpret(EmailMessage emailMessage) {
        return emailMessage.getFrom() != null && emailMessage.getFrom().equalsIgnoreCase(sender);
    }
}
```

Рисунок 3. Клас SenderFilter

```java
public class StatusFilter implements Expression {
    2 usages
    private EmailStatus status;

    2 usages
    public StatusFilter(EmailStatus status) { this.status = status; }

    5 usages
    @Override
    public boolean interpret(EmailMessage emailMessage) { return emailMessage.getEmailStatus() == status; }
}
```

Рисунок 4. Клас StatusFilter

```java
public class SubjectFilter implements Expression {
    2 usages
    private String keyword;

    2 usages
    public SubjectFilter(String keyword) { this.keyword = keyword; }

    5 usages
    @Override
    public boolean interpret(EmailMessage emailMessage) {
        return emailMessage.getSubject() != null && emailMessage.getSubject().contains(keyword);
    }
}
```

Рисунок 5. Клас SubjectFilter

Класи, такі як SenderFilter, SubjectFilter, StatusFilter реалізують інтерфейс Expression і визначають конкретні умови фільтрації для кожного з фільтрів. Ці фільтри перевіряють, чи відповідає повідомлення певному критерію (відправник, тема, статус).

```
1 usage
public class AndFilter implements Expression {
    2 usages
    private Expression filter1;
    2 usages
    private Expression filter2;

    1 usage
    public AndFilter(Expression filter1, Expression filter2) {
        this.filter1 = filter1;
        this.filter2 = filter2;
    }

    5 usages
    @Override
    public boolean interpret(EmailMessage emailMessage) {
        return filter1.interpret(emailMessage) && filter2.interpret(emailMessage);
    }
}
```

Рисунок 6. Клас AndFilter

```
public class OrFilter implements Expression {
    2 usages
    private Expression filter1;
    2 usages
    private Expression filter2;

    1 usage
    public OrFilter(Expression filter1, Expression filter2) {
        this.filter1 = filter1;
        this.filter2 = filter2;
    }

    5 usages
    @Override
    public boolean interpret(EmailMessage emailMessage) {
        return filter1.interpret(emailMessage) || filter2.interpret(emailMessage);
    }
}
```

Рисунок 7. Клас OrFilter

Класи AndFilter і OrFilter реалізують логічні операції, комбінуючи декілька фільтрів. Це теж є важливим аспектом патерну Interpreter, оскільки він дозволяє створювати складніші вирази, комбінуючи простіші вирази. Клас

AndFilter повертає true, якщо обидва фільтри дають позитивний результат, а клас OrFilter — якщо хоча б один з фільтрів дає позитивний результат.

```
19  public class EmailFilterService {
20      public List<EmailMessage> getMessageBySender(List<EmailMessage> messages, String email){
21          if (email == null || email.isEmpty()) return Collections.emptyList();
22          return filterMessages(messages, new SenderFilter(email));
23      }
24
25      public List<EmailMessage> getMessageBySubject(List<EmailMessage> messages, String keyWord){
26          if (keyWord == null) return Collections.emptyList();
27          return filterMessages(messages, new SubjectFilter(keyWord));
28      }
29
30      public List<EmailMessage> getMessageByStatus(List<EmailMessage> messages, EmailStatus status){
31          if (status == null) return Collections.emptyList();
32          return filterMessages(messages, new StatusFilter(status));
33      }
34      public List<EmailMessage> getMessageByStatusAndSender(List<EmailMessage> messages, EmailStatus status, String email){
35          if (status == null || email == null || email.isEmpty()) return Collections.emptyList();
36          return filterMessages(messages, new AndFilter(new StatusFilter(status), new SenderFilter(email)));
37      }
38      public List<EmailMessage> getMessageBySubjectOrSender(List<EmailMessage> messages, String keyWord, String email){
39          if (keyWord == null || email == null || email.isEmpty()) return Collections.emptyList();
40          return filterMessages(messages, new OrFilter(new SubjectFilter(keyWord), new SenderFilter(email)));
41      }
        5 usages
42      private List<EmailMessage> filterMessages(List<EmailMessage> messages, Expression expression) {
43          if (messages == null || messages.isEmpty()) return Collections.emptyList();
44          return messages.stream()
45                  .filter(expression::interpret)
46                  .collect(Collectors.toList());
47      }
```

Рисунок 8. Використання фільтрів в класі EmailFilterService

Методи, такі як getMessageBySender, getMessageBySubject, getMessageByStatus, дозволяють застосовувати фільтри до списку повідомлень. Вони використовують конкретні фільтри або комбінації фільтрів (через AndFilter або OrFilter), щоб вибрати повідомлення, що відповідають певним критеріям.

Цей код є чітким прикладом використання патерну Interpreter для фільтрації електронної пошти, де кожен фільтр (як правило, об'єкт, що інтерпретує вираз) перевіряє відповідність певним умовам. Логічні операції (І та АБО) дозволяють комбінувати прості фільтри в більш складні вирази. У результаті цей патерн дає змогу створювати гнучку систему фільтрації, де правила можуть бути поєднані та змінювані.

```java
@    static List<EmailMessage> emailMessages() {
        // Створюємо список EmailMessage
        List<EmailMessage> emailMessages = new ArrayList<>();

        // Додаємо тестові повідомлення
        EmailMessage email1 = new EmailMessage();
        email1.setId("1");
        email1.setSubject("Робоча зустріч");
        email1.setFrom("manager@example.com");
        email1.setTo(List.of("employee1@example.com", "employee2@example.com"));
        email1.setSentDate(LocalDate.of( year: 2023,  month: 12,  dayOfMonth: 1));
        email1.setBody("Доброго дня! Нагадую про зустріч у понеділок о 10:00.");
        email1.setEmailStatus(EmailStatus.UNREAD); // Коректно встановлюємо статус
        email1.setAttachments(List.of()); // Немає вкладень

        EmailMessage email2 = new EmailMessage();
        email2.setId("2");
        email2.setSubject("Звіт за проектом");
        email2.setFrom("teamlead@example.com");
        email2.setTo(List.of( e1: "manager@example.com"));
        email2.setSentDate(LocalDate.of( year: 2023,  month: 12,  dayOfMonth: 13));
        email2.setBody("Добрий день! Надсилаю звіт про стан проекту.");
        email2.setEmailStatus(EmailStatus.UNREAD); // Встановлюємо статус
        email2.setAttachments(List.of()); // Немає вкладень

        EmailMessage email3 = new EmailMessage();
        email3.setId("3");
        email3.setSubject("Документи");
        email3.setFrom("hr@example.com");
        email3.setTo(List.of( e1: "new.employee@example.com"));
```

Рисунок 9. Створення списку повідомлень

```java
public static void testEmailAccount(){
    EmailAccountService emailAccountService = new EmailAccountService(new EmailAccountRepository(), new UserRepository());
//      try {
//          emailAccountService.addEmailAccount(new User(), new EmailAccount("qqwe35949@gmail.com", "Asdfgh321"));
//      } catch (SQLException e) {
//          throw new RuntimeException(e);
//      }
    EmailAccount emailAccount = new EmailAccount.EmailAccountBuilder( emailAddress: "qwertyeee@ukr.net",  password: "O4oiRBZT1bj5V8my").setAutoconfig(true).build();
    emailAccountService.authorizeEmail(emailAccount);

}
```

```
Main
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
[EmailMessage{id='1', subject='Робоча зустріч', from='manager@example.com', emailStatus=UNREAD}, EmailMessage{id='2', subject='Звіт за проектом', from='teamlead@example.com', e
[EmailMessage{id='1', subject='Робоча зустріч', from='manager@example.com', emailStatus=UNREAD}]
[EmailMessage{id='1', subject='Робоча зустріч', from='manager@example.com', emailStatus=UNREAD}]
```

Рисунок 10. Результат тестування

Висновок

У результаті виконання лабораторної роботи було успішно реалізовано патерн Interpreter для фільтрації електронної пошти, що дозволяє ефективно

комбінувати різні фільтри за допомогою логічних операторів І та АБО. Також були розглянуті принципи роботи з іншими патернами проектування, такими як Composite, Flyweight та Visitor, що дозволяє створювати програмні рішення, які мають високу гнучкість, зручність у підтримці та ефективність у виконанні.