



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 розробки програмного забезпечення

ШАБЛони «Abstract Factory», «Factory Method», «Memento»,
«Observer», «Decorator»

Виконала:
студенка групи ІА-24
Орловська А. В.
Перевірив:
Мягкий М. Ю.

Зміст

Короткі теоретичні відомості.....	3
Хід роботи.....	4
Реалізація шаблону проєктування для майбутньої системи	4
Зображення структури шаблону	8
Посилання на репозиторій.....	8
Висновок	8

Тема: ШАБЛОНИ «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»

Короткі теоретичні відомості

Abstract Factory (Абстрактна фабрика) – це створювальний шаблон, який надає інтерфейс для створення сімейства взаємопов'язаних об'єктів без вказівки їх конкретних класів. Абстрактна фабрика дозволяє створювати продукти, які належать до певної сім'ї, не залежачи від того, яка саме реалізація буде обрана. Цей шаблон корисний, коли потрібно забезпечити взаємодію різних продуктів, але без залежності від їх конкретних реалізацій.

Factory Method (Фабричний метод) – це створювальний шаблон, який визначає інтерфейс для створення об'єктів, але дозволяє підкласам змінювати тип створюваного об'єкта. Відмінність від шаблону **Abstract Factory** полягає в тому, що **Factory Method** забезпечує створення одиничного продукту, а не всієї сім'ї продуктів. Цей шаблон корисний, коли потрібно делегувати створення об'єкта підкласам і не прив'язуватися до конкретних класів об'єктів.

Memento (Мементо) – це поведінковий шаблон, який дозволяє зберігати та відновлювати попередній стан об'єкта без порушення інкапсуляції. Мементо використовується, коли потрібно зберегти стан об'єкта для подальшого відновлення, наприклад, для реалізації функції "відкату" (undo). Шаблон складається з трьох основних компонентів: **Originator** (об'єкт, чий стан зберігається), **Memento** (об'єкт, що зберігає стан) та **Caretaker** (об'єкт, який зберігає мементо).

Observer (Спостерігач) – це поведінковий шаблон, який визначає залежність типу "один до багатьох" між об'єктами, де зміни в одному об'єкті автоматично сповіщають усіх залежних від нього об'єктів. Це дозволяє створювати події або реакції на зміни, не знаючи точно, які об'єкти повинні реагувати. Шаблон **Observer** часто використовується для реалізації систем

подій або підписки.

Decorator (Декоратор) – це структурний шаблон, який дозволяє динамічно додавати нові функціональні можливості об'єктам, не змінюючи їх структуру. Декоратор надає можливість обгорнути об'єкт у новий клас, який додає або змінює його поведінку. Це дає змогу гнучко модифікувати функціональність об'єкта без створення численних підкласів. Шаблон **Decorator** часто застосовується для розширення функціональності елементів у графічних інтерфейсах або для реалізації патернів, що включають додаткові можливості для об'єктів.

Хід роботи

Система для колективних покупок(proxy, builder, decorator, facade, composite). Система дозволяє створити список групи для колективної покупки, список що потрібно купити з орієнтовною вартістю кожної позиції та орієнтовною загальною вартістю, запланувати хто що буде купляти. Щоб користувач міг відмітити що він купив, за яку суму, з можливістю прикріпити чек. Система дозволяє користувачу вести списки бажаних для нього покупок, з можливістю позначати списки, які будуть доступні для друзів (як списки, що можна подарувати користувачеві). Система дозволяє добавляти інших користувачів в друзі.

Основні принципи та застосування Decorator

Декоратор — це структурний шаблон проектування, який дозволяє динамічно додавати нову функціональність об'єктам без зміни їхнього коду. Його ключова ідея полягає в обгортанні об'єкта в інші об'єкти (декоратори), кожен з яких додає певну поведінку або властивість.

Цей підхід забезпечує прозорість для клієнта: декоратори реалізують той самий інтерфейс, що й об'єкти, які вони декорують. Таким чином, клієнтський код може працювати з об'єктом, не знаючи, чи є він "базовим" або декорованим.

Основна перевага декоратора полягає в його гнучкості, адже він дозволяє створювати багаторівневі композиції, де кожен рівень додає нову функціональність. Це досягається за допомогою композиції, а не наслідування, що значно спрощує модифікацію й розширення системи.

Завдяки декоратору можна додавати, змінювати або видаляти поведінку об'єкта під час виконання програми, комбінуючи різні декоратори в потрібному порядку.

Застосування:

- Додавання функціональності без зміни коду класу.
- Розширення можливостей об'єктів у динамічних системах.
- Інкапсуляція додаткової логіки (наприклад, логування, кешування, перевірка доступу).
- Приклад: у веб-застосунках декоратори можуть використовуватися для додавання автентифікації або обмеження доступу до певних ресурсів.

Реалізація шаблону проєктування для майбутньої системи

Decorator демонструє добре структурований підхід до розширення функціональності без зміни основного коду. Я створила інтерфейс `PurchasePlanDecorator`, який визначає метод `decorate`, що служить контрактом для всіх декораторів. Це дозволяє легко додавати нові декоратори, забезпечуючи дотримання принципів SOLID, особливо Open/Closed Principle. Клас `ImportantPurchasePlanDecorator` реалізує цей інтерфейс, додаючи логіку для встановлення прапорця важливості плану через метод `setImportant(true)`. У процесі виконання він також логує операції за допомогою бібліотеки `SLF4J`. Це дає змогу не лише модифікувати об'єкт, але й фіксувати дії для зручного відстеження.

Метод `createPurchasePlan` ілюструє практичне використання декоратора. Спочатку DTO перетворюється в сутність, після чого виконується перевірка на важливість плану. Якщо план є важливим, до нього застосовується декоратор, який додає відповідну модифікацію. Після цього план зберігається в базі даних, і результат повертається у вигляді DTO.

Такий підхід забезпечує гнучкість і чистоту коду. Ви ізолювали логіку декорування в окремих класах, що спрощує підтримку та розширення функціональності. Якщо виникне потреба в додаткових модифікаціях, їх можна реалізувати через нові декоратори, не змінюючи основний код.

Для подальшого вдосконалення можна реалізувати підтримку ланцюжка

декораторів, що дозволить застосовувати кілька декораторів до одного об'єкта. Це може бути особливо корисним у складних системах. Крім того, інтеграція з механізмом DI, наприклад, Spring, дозволить автоматизувати управління декораторами, роблячи систему ще більш масштабованою.


```
1 package org.example.collectivepurchases.services.purchaseplan;  
2  
3 import org.example.collectivepurchases.models.PurchasePlan;  
4  
5  
6 2 usages 1 implementation  
7 public interface PurchasePlanDecorator {  
8     2 usages 1 implementation  
9     PurchasePlan decorate(PurchasePlan purchasePlan);  
10 }
```

Рис. 1 – Код інтерфейсу PurchasePlanDecorator

```
1 package org.example.collectivepurchases.services.purchaseplan;  
2  
3 import lombok.extern.slf4j.Slf4j;  
4 import org.example.collectivepurchases.models.PurchasePlan;  
5 1 usage  
6 @Slf4j  
7 public class ImportantPurchasePlanDecorator implements PurchasePlanDecorator {  
8     2 usages  
9     @Override  
10    public PurchasePlan decorate(PurchasePlan purchasePlan) {  
11        purchasePlan.setImportant(true);  
12        log.info("Purchase plan is decorated");  
13        return purchasePlan;  
14    }  
15 }
```

Рис. 2 – Код класу ImportantPurchasePlanDecorator

```

28
29  @Override
30 public PurchasePlanDto createPurchasePlan(PurchasePlanDto purchasePlanDto) {
31     PurchasePlan purchasePlan = purchasePlanConverter.convertToEntity(purchasePlanDto);
32     if (purchasePlanDto.isImportant()) {
33         purchasePlan = importantPurchasePlanDecorator.decorate(purchasePlan);
34     }
35     PurchasePlan savedPlan = purchasePlanRepository.save(purchasePlan);
36     return purchasePlanConverter.buildPurchasePlanDto(savedPlan);
37 }

```

Рис. 3 – Приклад використання PurchasePlanDecorator

o.e.c.s.p.ImportantPurchasePlanDecorator : Purchase plan is decorated

Рис. 4 – Результат використання

Зображення структури шаблону

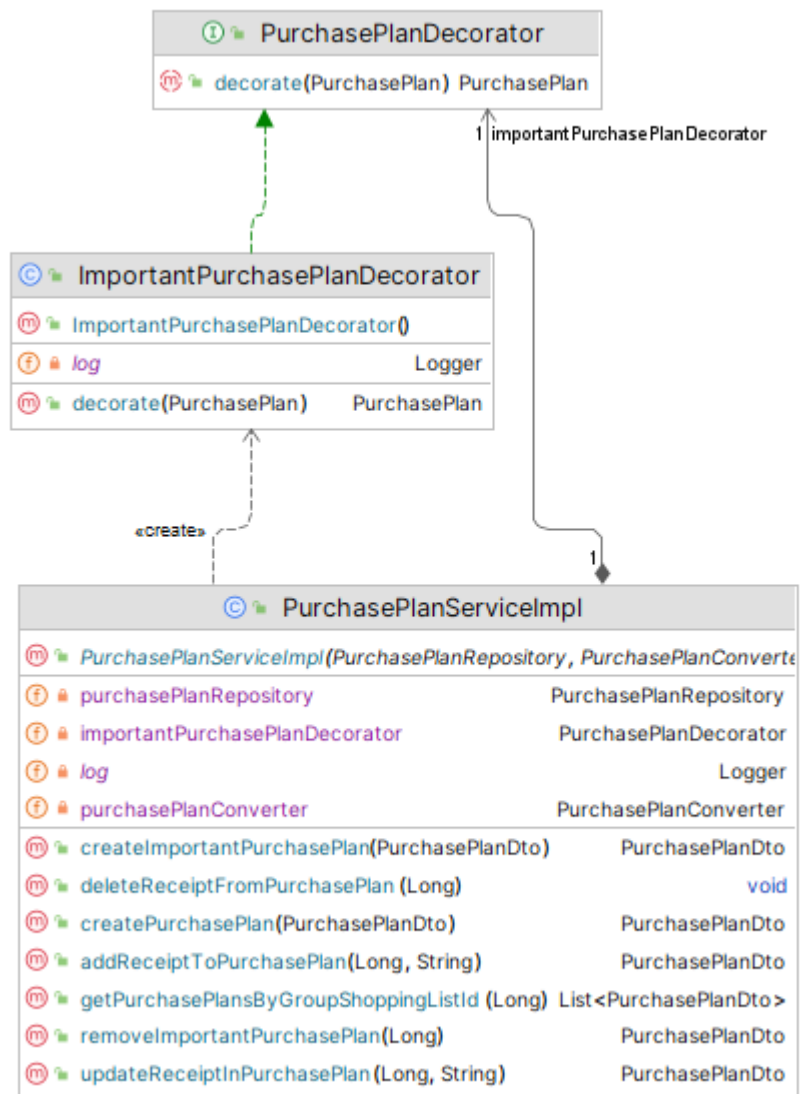


Рис. 5 – Структура шаблону

На представленій діаграмі реалізовано патерн **Декоратор**, який забезпечує можливість динамічного розширення функціональності об'єктів без зміни їхньої базової структури. Основу цього патерна складає клас `PurchasePlanDecorator`, який виступає абстрактним декоратором і визначає метод `decorate`, що дозволяє модифікувати або розширювати функціональність об'єктів типу `PurchasePlan`. Конкретна реалізація декоратора представлена класом `ImportantPurchasePlanDecorator`, який додає специфічну поведінку до

об'єкта, наприклад, позначення плану закупівель як важливого. Крім того, використання компонента `Logger` у цьому класі дозволяє вести запис виконаних дій, додаючи ще більше функціональності.

Усі ці елементи взаємодіють у рамках класу `PurchasePlanServiceImpl`, який використовує декоратор для обробки планів закупівель. Це дає можливість застосовувати додаткові модифікації до об'єктів `PurchasePlan` без змін у базовій реалізації інших компонентів системи. Таким чином, дана структура забезпечує гнучкість і розширюваність, дозволяючи легко додавати нові функції за допомогою патерна Декоратор, що відповідає принципам об'єктно-орієнтованого дизайну.

Посилання на репозиторій:

<https://github.com/annaorlovskaaa/collectivePurchases.git>

Висновок: Шаблони проєктування «Abstract Factory», «Factory Method», «Memento», «Observer» та «Decorator» є потужними інструментами, які забезпечують гнучкість і масштабованість програмного забезпечення. Їх використання дозволяє створювати структурований код, дотримуючись принципів SOLID і забезпечуючи легкість підтримки та розширення.

У випадку «Decorator», реалізація показала ефективний підхід до розширення функціональності об'єктів без внесення змін до основного коду. Використання інтерфейсу `PurchasePlanDecorator` та конкретних реалізацій, як-от `ImportantPurchasePlanDecorator`, забезпечує ізоляцію додаткової логіки. Це дозволяє гнучко модифікувати поведінку об'єктів і підтримувати принцип відкритості/закритості (ОСР). Крім того, застосування бібліотеки `SLF4J` для логування робить систему зручною для відстеження дій. Залучення декораторів у методі `createPurchasePlan` демонструє, як можна ефективно поєднувати декоратори з основним функціоналом.