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 №5

Технології розробки програмного забезпечення

ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»

Виконала:
студентка групи ІА-24
Орловська О. В.
Перевірив:
Мягкий М. Ю.

Київ 2024

Зміст

Тема: ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»

Короткі теоретичні відомості

Adapter (Адаптер) – це структурний шаблон, який дозволяє об'єктам з різними інтерфейсами працювати разом. Адаптер перетворює інтерфейс одного класу на інтерфейс, який очікує інший клас. Це дозволяє інтегрувати різні компоненти в систему, не змінюючи їх внутрішню реалізацію. Адаптери часто використовуються для сумісності з існуючими бібліотеками чи старими системами.

Builder (Будівельник) – це створювальний шаблон, який розділяє процес створення складних об'єктів на кілька етапів. Це дозволяє створювати різні варіанти об'єкта за допомогою однакових кроків, але з різними параметрами. Шаблон Builder корисний для побудови об'єктів, які мають багато складових, і дозволяє керувати їх створенням поетапно.

Command (Команда) – це поведінковий шаблон, який інкапсулює запит як об'єкт, що дозволяє параметризувати методи виклику, ставити їх у чергу або записувати для подальшого виконання. Шаблон Command дозволяє розділяти запити та їх виконання, а також легко реалізовувати операції скасування (undo) чи повторення (redo). Це дозволяє зручно працювати з виконанням команд у додатках.

Chain of Responsibility (Цепочка відповідальності) – це поведінковий шаблон, який дозволяє передавати запит по ланцюгу обробників. Кожен обробник може або обробити запит, або передати його наступному обробнику в ланцюгу. Це дає змогу динамічно змінювати порядок обробки запитів, а також зменшує зв'язність між клієнтом і обробниками.

Prototype (Прототип) – це створювальний шаблон, який дозволяє копіювати існуючі об'єкти замість їх створення з нуля. За допомогою шаблону Prototype об'єкт може створити нові екземпляри на основі себе, зменшуючи накладні витрати на створення складних об'єктів. Цей шаблон корисний, коли потрібно створювати багато подібних об'єктів, але зі змінами, які важко досягти через звичайне конструювання.

Хід роботи

***Система для колективних покупок*(proxy, builder, decorator, façade, 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 Builder:

Шаблон проектування **Builder** (Будівельник) є одним з найбільш корисних у розробці програмного забезпечення, коли потрібно створити складні об'єкти з різними варіантами налаштувань або компонентів. Його основне завдання — спростити процес створення об'єкта, розділяючи процес побудови на окремі етапи.

Основні принципи шаблону Builder:

1. **Розділення побудови та представлення**:
    - Шаблон Builder дозволяє відокремити процес створення об'єкта від його представлення. Це означає, що одна й та сама будівельна логіка може створювати різні варіанти об'єкта без зміни самого класу або його структури.
2. **Крокове створення об'єкта**:
    - Builder дозволяє будувати об'єкти крок за кроком, даючи можливість створювати складні об'єкти з різними варіантами налаштувань на кожному етапі.
3. **Абстракція створення об'єкта**:
    - Клієнт працює з **Director** (керівник), який керує процесом побудови об'єкта, а безпосереднє створення об'єкта здійснюється через **Builder**.

Клієнт не взаємодіє з внутрішніми деталями створення об'єкта, це робить Builder.

4. **Чистота коду**:
    - Використання шаблону Builder дозволяє зберігати код чистим, не перевантажуючи його великою кількістю конструкцій для конфігурації об'єктів, а також дає можливість легко додавати нові варіанти об'єктів без зміни існуючої логіки.

Реалізація шаблону проєктування для майбутньої системи

Шаблон Builder (Будівельник) є поведінковим шаблоном проєктування, який дозволяє створювати складні об'єкти крок за кроком. Кожен етап побудови об'єкта може бути реалізований через окремий метод, що дозволяє динамічно налаштовувати різні частини об'єкта без необхідності змінювати його структуру. Це дає змогу створювати об'єкти з різними конфігураціями, зменшуючи залежність між клієнтом та конкретними етапами побудови. Шаблон Builder дозволяє розділити процес створення та представлення об'єкта, що забезпечує більшу гнучкість і масштабованість у розробці складних об'єктів.

```
23 usages
@Entity
@Table(name = "items")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Item {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;
    private double price;
    private double quantity;

```

Рис. 1 – Код Item

```
27      public static class Builder {
            3 usages
28          private Long id;
            3 usages
29          private String name;
            3 usages
30          private Category category;
            3 usages
31          private double price;
            3 usages
32          private double quantity;
33
            no usages
34          public Builder id(Long id) {
35              this.id = id;
36              log.info("Setting id: {}", id);
37              return this;
38          }
39
            no usages
40          public Builder name(String name) {
41              this.name = name;
42              log.info("Setting name: {}", name);
43              return this;
44          }
45
            no usages
46          public Builder category(Category category) {
47              this.category = category;
48              log.info("Setting category: {}", category);
49              return this;
50          }
51
```

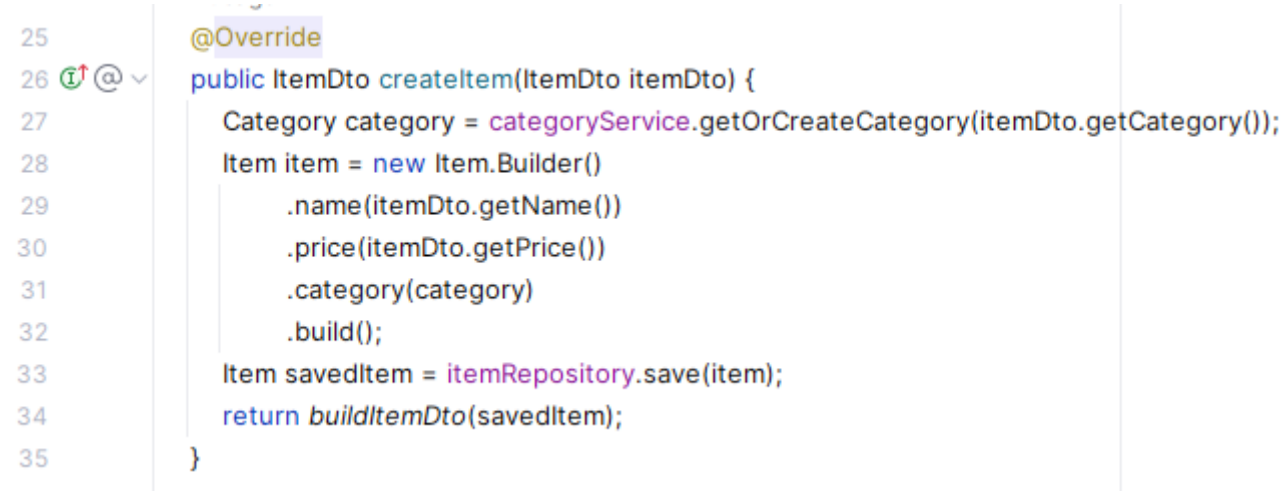
Рис. 2.1 – Код Builder

```
        no usages
52      public Builder price(double price) {
53          this.price = price;
54          log.info("Setting price: {}", price);
55          return this;
56      }
57
        no usages
58      public Builder quantity(double quantity) {
59          this.quantity = quantity;
60          log.info("Setting quantity: {}", quantity);
61          return this;
62      }
63
64      public Item build() {
65          log.info("Building Item with id: {}, name: {}, category: {}, price: {}, quantity: {}",
66                  id, name, category, price, quantity);
67          Item item = new Item();
68          item.id = this.id;
69          item.name = this.name;
70          item.category = this.category;
71          item.price = this.price;
72          item.quantity = this.quantity;
73          return item;
74      }
75  }
76  }
```

Рис. 2.2 – Код Builder

Паттерн Builder використовується для спрощення створення об'єктів класу Item.
Статичний клас Builder: Визначає всі поля, які є в класі Item, і методи для їх встановлення.
Методи встановлення значень: Кожен метод встановлення значення повертає об'єкт Builder, що дозволяє ланцюжковий виклик методів.
Метод build(): Створює і повертає новий об'єкт Item з встановленими значеннями.
Цей підхід дозволяє створювати об'єкти з багатьма параметрами, зберігаючи код чистим і зрозумілим.

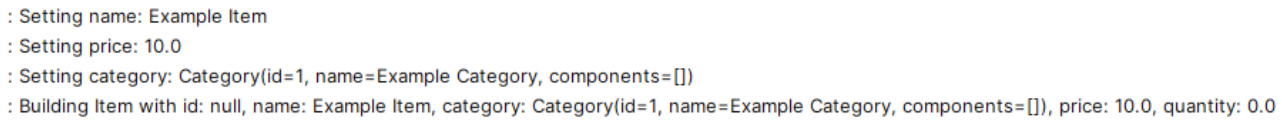
У цьому прикладі ми створюємо об'єкт **Item** за допомогою класу **Builder**. Використовуючи ланцюгові виклики методів, ми задаємо значення для полів **name**, **price** та **quantity**. В кінці викликаємо метод **build()**, щоб отримати готовий об'єкт **Item**. Це дозволяє створювати об'єкти з багатьма параметрами без потреби створювати громіздкі конструктори з великою кількістю параметрів.



```
25      @Override
26      public ItemDto createItem(ItemDto itemDto) {
27          Category category = categoryService.getOrCreateCategory(itemDto.getCategory());
28          Item item = new Item.Builder()
29                  .name(itemDto.getName())
30                  .price(itemDto.getPrice())
31                  .category(category)
32                  .build();
33          Item savedItem = itemRepository.save(item);
34          return buildItemDto(savedItem);
35      }
```

Рис. 3– Приклад використання Builder

У цьому прикладі метод createItem використовує паттерн Builder для створення об'єкта Item. Ось як це працює:
Отримання або створення категорії: Викликається метод getOrCreateCategory з сервісу categoryService, щоб отримати або створити об'єкт Category на основі даних з itemDto.
Створення об'єкта Item за допомогою Builder: Використовується статичний клас Builder для поетапного встановлення значень полів об'єкта Item. Викликаються методи name, price та category, щоб встановити відповідні значення з itemDto та об'єкта Category.
Збереження об'єкта Item в репозиторії: Викликається метод save з репозиторію itemRepository, щоб зберегти створений об'єкт Item в базі даних.
Повернення об'єкта ItemDto: Викликається метод buildItemDto, щоб перетворити збережений об'єкт Item в об'єкт ItemDto, який потім повертається як результат методу createItem.



```
: Setting name: Example Item
: Setting price: 10.0
: Setting category: Category(id=1, name=Example Category, components=[])
: Building Item with id: null, name: Example Item, category: Category(id=1, name=Example Category, components=[]), price: 10.0, quantity: 0.0
```

Рис. 4– Приклад роботи Builder

Зображення структури шаблону

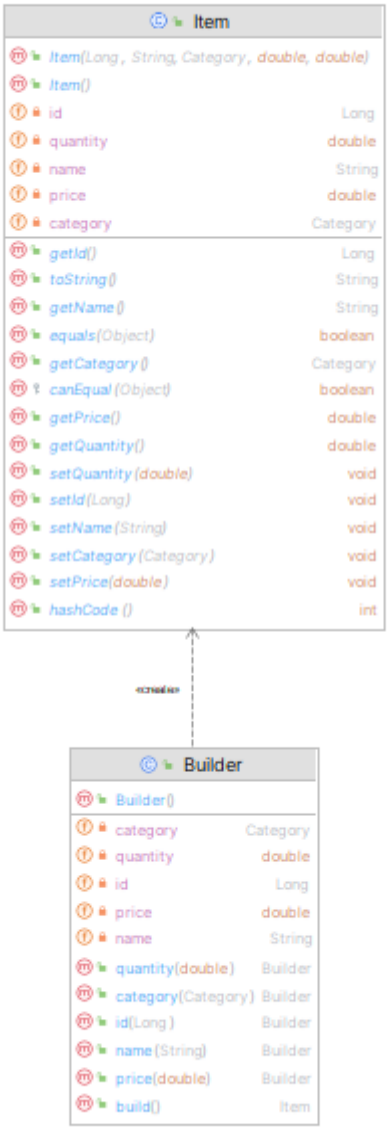


Рис. 5 – Структура шаблону

Посилання на репозиторій:

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

Висновок: У результаті реалізації шаблону Builder у вигляді класу Item.Builder було продемонстровано ефективність цього патерна для створення об'єктів з

багатьма параметрами. Клас Builder дозволяє поетапно встановлювати значення полів об'єкта Item, використовуючи методи для кожного поля, і в кінці викликати метод build(), який повертає готовий об'єкт. Це забезпечує гнучкість системи, спрощує створення об'єктів з різними параметрами та підтримку існуючого коду. Використання шаблону дозволило зменшити кількість умовних конструкцій, зробивши код більш структурованим і відповідним принципам SOLID, зокрема принципу єдиного обов'язку та принципу відкритості/закритості.