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 розроблення програмного забезпечення
ШАБЛОНИ «ADAPTER», «BUILDER»,
«COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»
Особиста Бухгалтерія

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

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: шаблони «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»
Мета: ознайомитись з теоретичними відомостями, розробити частину функціоналу системи з використанням одного із шаблонів проектування.

Теоретичні відомості

Шаблон Adapter використовується для узгодження інтерфейсів класів, які спочатку не були призначені для спільної роботи. Його головна мета полягає у створенні своєрідного мосту між класами без необхідності змінювати їхній код. Це дозволяє перетворювати інтерфейс одного класу в інший, очікуваний клієнтським кодом. Такий підхід особливо корисний, коли потрібно інтегрувати в систему сторонній код або бібліотеки з несумісними інтерфейсами.

Шаблон Builder допомагає спростити створення складних об'єктів, розділяючи цей процес на послідовні кроки. Його використання особливо виправдане, якщо об'єкт має багато конфігурацій. У цьому шаблоні відокремлюється логіка побудови об'єкта від його кінцевої структури, що сприяє гнучкості та розширюваності коду. Builder дозволяє легко створювати об'єкти з різними параметрами, зберігаючи при цьому їхню узгодженість.

Шаблон Command інкапсулює запити або операції як окремі об'єкти. Це дозволяє зберігати їх для подальшого виконання, передавати між об'єктами або навіть підтримувати операції скасування й повторення. Command використовується в системах, де необхідно організувати гнучке управління діями, наприклад, в системах з графічним інтерфейсом або у складних бізнес-логіках.

Шаблон Chain of Responsibility створює ланцюжок об'єктів, через який передається запит до тих пір, поки один із об'єктів не обробить його. Це дозволяє уникнути жорсткої прив'язки відправника запиту до конкретного одержувача. Кожен елемент ланцюжка відповідає за свою частину обробки, а якщо він не може виконати запит, то передає його далі. Такий підхід забезпечує гнучкість і полегшує розширення системи.

Шаблон Prototype дозволяє створювати нові об'єкти шляхом копіювання існуючих. Це особливо ефективно в ситуаціях, коли створення об'єкта з нуля є ресурсозатратним або складним. Прототипи використовуються для створення складних об'єктів, де важливо зберігати їхню початкову конфігурацію або стан.

Загалом, використання цих шаблонів у проєктуванні програмного забезпечення сприяє підвищенню якості архітектури, спрощує підтримку коду, робить систему більш гнучкою та зрозумілою для розробників. Їх вивчення та правильне застосування є невід’ємною частиною професійного зростання кожного програміста.

Хід роботи

Для виконання даної лабораторної роботи було прийнято рішення реалізувати функціонал створення транзакцій за допомогою патерну Builder.

У цій лабораторній роботі було обрано патерн «Builder», оскільки він дозволяє створювати об'єкти з великою кількістю параметрів у кілька етапів. Це особливо зручно, коли необхідно забезпечити гнучкість та узгодженість під час конфігурування об'єкта. У класі Transaction реалізація «Builder» забезпечила інтуїтивний спосіб задання параметрів через методи, що дозволило уникнути складності з перевантаженими конструкторами. Це спростило код і зробило його читабельним та зручним для розробників.

Патерн «Prototype», хоча й має свої переваги, зокрема швидке створення копій об'єктів, не був використаний у цьому випадку. Причина в тому, що завдання не передбачає частого створення ідентичних об'єктів на основі вже існуючих. Основна мета була в побудові об'єкта з різними конфігураціями, а не в копіюванні готового. Використання «Prototype» могло б бути виправданим, якби створення об'єктів було ресурсозатратним і потребувало швидкої генерації на основі існуючих шаблонів. Однак у цьому проєкті кожен об'єкт транзакції створюється з унікальними параметрами, які задаються динамічно.

Отже, «Builder» став оптимальним вибором, адже він забезпечує саме ту функціональність, яка потрібна для створення об'єктів у цій системі.

Було реалізовано клас Transaction із підтримкою патерну проектування Builder, який забезпечує зручний спосіб створення об'єктів із різними параметрами. Основна ідея цього підходу полягає у винесенні логіки побудови об'єкта в окремий статичний клас Builder.

Шаблон Builder дозволяє поступово задавати необхідні параметри майбутнього об'єкта, забезпечуючи гнучкість і простоту конфігурації. У класі Transaction Builder містить набір методів, таких як name(), description(), amount(), що відповідають за заповнення окремих полів об'єкта. В кінці створення викликається метод build(), який повертає екземпляр Transaction із заданими параметрами. (Рис. 1.1)

```java
public static class Builder {  7 usages
    private String name;  2 usages
    private String description;  2 usages
    private BigDecimal amount;  2 usages
    private Category category;  2 usages
    private TransactionTypes transactionType;  2 usages

    public Builder name(String name) {  no usages
        this.name = name;
        return this;
    }

    public Builder description(String description) {  no usages
        this.description = description;
        return this;
    }

    public Builder amount(BigDecimal amount) {  no usages
        this.amount = amount;
        return this;
    }

    public Builder category(Category category) {  no usages
        this.category = category;
        return this;
    }

    public Builder transactionType(TransactionTypes transactionType) {  no usages
        this.transactionType = transactionType;
        return this;
    }

    public Transaction build() { return new Transaction( builder: this); }
}
```

Рисунок 1.1. Статичний клас Builder для створення об'єктів Transaction

Використання Builder дозволяє зручно конструювати екземпляри класу без необхідності створення численних перевантажених конструкторів, що підвищує читабельність коду. Наприклад, створення транзакції може виглядати наступним чином:

```java
@Transactional  1 usage
public void executeTransaction(PeriodicTransaction transaction) {
    var newTransaction = new Transaction.Builder()
            .name(transaction.getName())
            .amount(transaction.getAmount())
            .category(transaction.getCategory())
            .transactionType(transaction.getTransactionType())
            .description(transaction.getDescription())
            .build();
    accountService.addTransaction(newTransaction, transaction.getAccount().getId());
    transaction.setLastExecution(Instant.now());
    periodicTransactionRepository.save(transaction);
}
```

Рисунок 1.2 Використання Builder для створення транзакції

Висновки

У роботі було реалізовано шаблон Builder, який дозволив створювати об'єкти з багатьма параметрами у декілька етапів. Я розробила функціонал для генерації складних фінансових звітів, що вимагають конфігурації. Це допомогло зрозуміти, як можна розділити процес створення об'єкта на логічні кроки, забезпечуючи більшу гнучкість у налаштуванні системи. Також я усвідомила важливість збереження узгодженості створюваних об'єктів. Інші шаблони були розглянуті для теоретичного аналізу їхнього потенційного застосування

Вихідний код

Вихідний код, пов’язаний з реалізацією патерну Builder, що відповідає за створення транзакції.

Реалізація класу Transaction зі статичним класом Builder всередині

```
@Getter
@Setter
@Entity
@NoArgsConstructor
@Table(name = "transaction")
public class Transaction {
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  @Column(nullable = false)
  private Long id;

  @Column(name = "name")
  private String name;

  @Column(name = "description")
  private String description;

  @Column(name = "amount", precision = 19, scale = 2)
  private BigDecimal amount;

  @Column(name = "timestamp")
  private Instant timestamp = Instant.now();

  @ManyToOne
  @JoinColumn(name = "account id")
  private Account account;

  @ManyToOne
  @JoinColumn(name = "category_id")
  private Category category;

  @Enumerated(EnumType.STRING)
  @Column(name = "transaction_type")
  private TransactionTypes transactionType;

  @ManyToOne
  @JoinColumn(name = "user_id")
  private User user;

  private Transaction(Builder builder) {
    this.name = builder.name;
    this.description = builder.description;
    this.amount = builder.amount;
    this.category = builder.category;
    this.transactionType = builder.transactionType;
  }

  public static class Builder {
    private String name;
    private String description;
    private BigDecimal amount;
    private Category category;
    private TransactionTypes transactionType;

    public Builder name(String name) {
      this.name = name;
```

```
      return this;
    }

    public Builder description(String description) {
      this.description = description;
      return this;
    }

    public Builder amount(BigDecimal amount) {
      this.amount = amount;
      return this;
    }

    public Builder category(Category category) {
      this.category = category;
      return this;
    }

    public Builder transactionType(TransactionTypes transactionType) {
      this.transactionType = transactionType;
      return this;
    }

    public Transaction build() {
      return new Transaction(this);
    }
  }


}
```

Використання Buider під час створення транзакції:

```
@Transactional
public void executeTransaction(PeriodicTransaction transaction) {
    var newTransaction = new Transaction.Builder()
            .name(transaction.getName())
            .amount(transaction.getAmount())
            .category(transaction.getCategory())
            .transactionType(transaction.getTransactionType())
            .description(transaction.getDescription())
            .build();
    accountService.addTransaction(newTransaction,
transaction.getAccount().getId());
    transaction.setLastExecution(Instant.now());
    periodicTransactionRepository.save(transaction);
}
```