**同济大学计算机系**

**软件开发方法作业二报告**

****

**学 号 2251745**

**姓 名 张宇**

**专 业 计算机科学与技术**

**授课老师 王小平老师**

**完成日期 2024年6月14日**

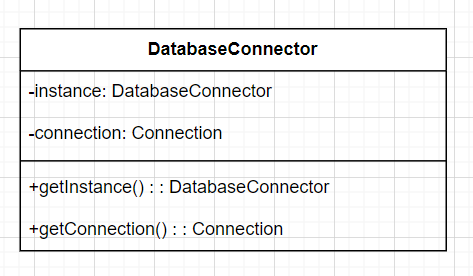
1. 创建型模式：单例模式 (Singleton Pattern)
2. 应用问题描述

在电子商城系统中，数据库连接器应作为单例存在，确保系统中只有一个数据库连接实例，以避免资源浪费和潜在的多线程问题。

1. 适用设计模式分析

单例模式确保一个类只有一个实例，并提供一个访问它的全局访问点。对于需要共享资源的场景，如数据库连接，单例模式是理想的选择。

1. UML类图



1. 源程序

public class DatabaseConnector {

    private static DatabaseConnector instance;

    private Connection connection;

    private DatabaseConnector() {

        // Initialize the database connection

        try {

            Class.forName("com.mysql.cj.jdbc.Driver");

            this.connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/ecommerce", "root", "password");

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

    public static DatabaseConnector getInstance() {

        if (instance == null) {

            synchronized (DatabaseConnector.class) {

                if (instance == null) {

                    instance = new DatabaseConnector();

                }

            }

        }

        return instance;

    }

    public Connection getConnection() {

        return connection;

    }

}

public class SingletonPatternDemo {

    public static void main(String[] args) {

        DatabaseConnector connector1 = DatabaseConnector.getInstance();

        DatabaseConnector connector2 = DatabaseConnector.getInstance();

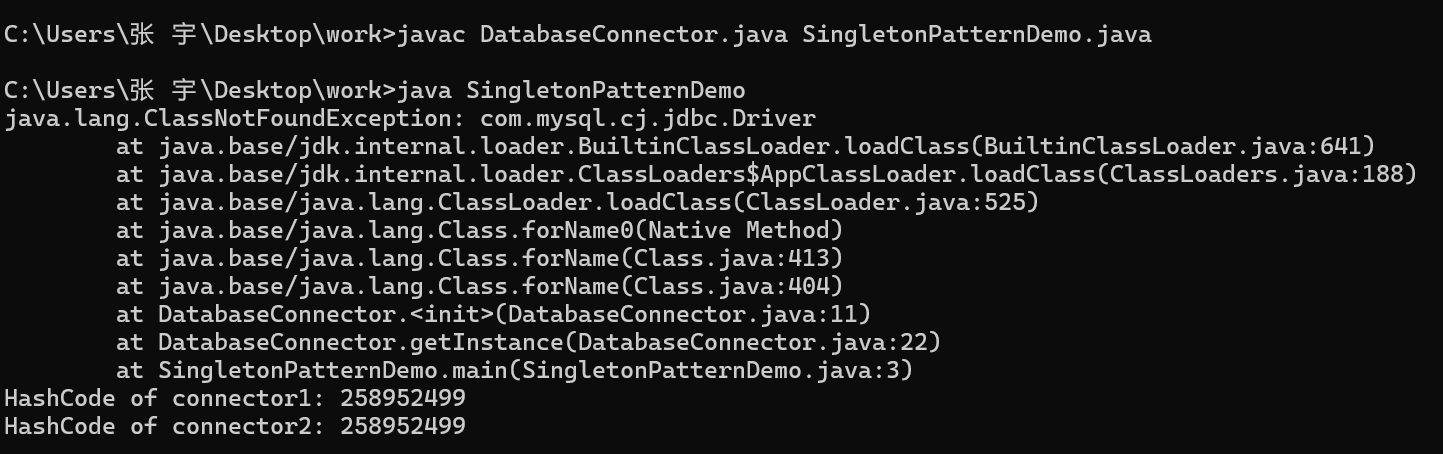
        System.out.println("HashCode of connector1: " + connector1.hashCode());

        System.out.println("HashCode of connector2: " + connector2.hashCode());

    }

}

1. 实验输出结果及其解释



两个连接器实例的哈希码相同，表明只有一个实例存在，验证了单例模式的有效性。

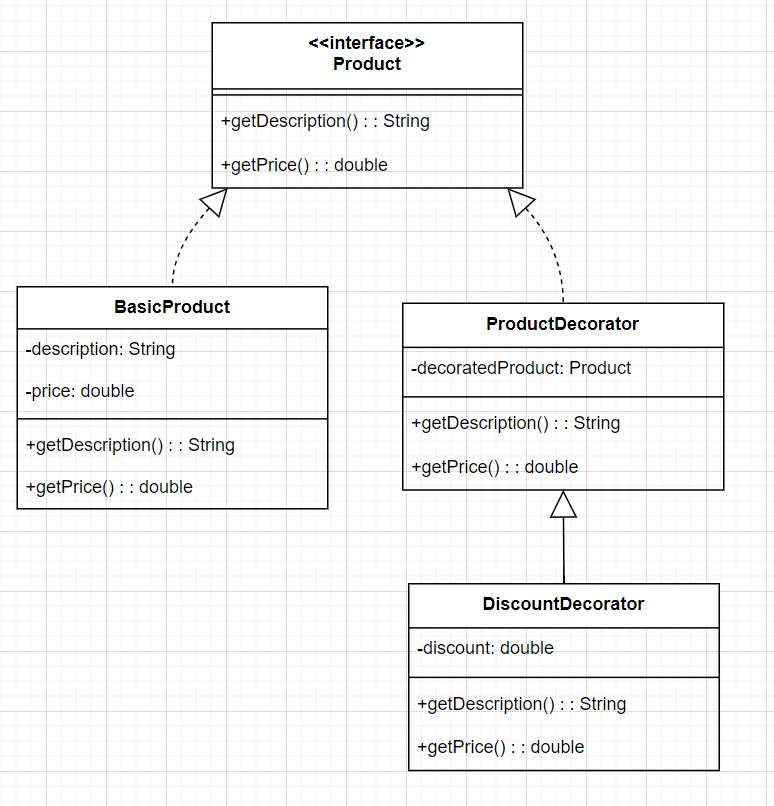
1. 结构型模式：装饰者模式 (Decorator Pattern)
2. 应用问题描述

在电子商城系统中，为商品添加不同的特性，例如折扣、促销等。

1. 适用设计模式分析

装饰者模式允许向一个对象动态添加职责，是继承的灵活替代方案。适用于需要扩展类功能但又不希望使用子类化的场景。

1. UML类图



1. 源程序

interface Product {

    String getDescription();

    double getPrice();

}

class BasicProduct implements Product {

    private String description;

    private double price;

    public BasicProduct(String description, double price) {

        this.description = description;

        this.price = price;

    }

    public String getDescription() {

        return description;

    }

    public double getPrice() {

        return price;

    }

}

abstract class ProductDecorator implements Product {

    protected Product decoratedProduct;

    public ProductDecorator(Product decoratedProduct) {

        this.decoratedProduct = decoratedProduct;

    }

    public String getDescription() {

        return decoratedProduct.getDescription();

    }

    public double getPrice() {

        return decoratedProduct.getPrice();

    }

}

class DiscountDecorator extends ProductDecorator {

    private double discount;

    public DiscountDecorator(Product decoratedProduct, double discount) {

        super(decoratedProduct);

        this.discount = discount;

    }

    public String getDescription() {

        return decoratedProduct.getDescription() + " with discount";

    }

    public double getPrice() {

        return decoratedProduct.getPrice() \* (1 - discount);

    }

}

public class DecoratorPatternDemo {

    public static void main(String[] args) {

        Product basicProduct = new BasicProduct("Basic Product", 100.0);

        Product discountedProduct = new DiscountDecorator(basicProduct, 0.1);

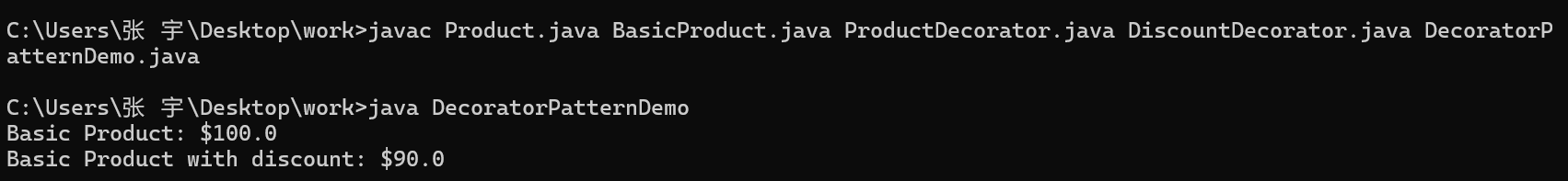
        System.out.println(basicProduct.getDescription() + ": $" + basicProduct.getPrice());

        System.out.println(discountedProduct.getDescription() + ": $" + discountedProduct.getPrice());

    }

}

1. 实验输出结果及其解释



商品在应用折扣后，价格从100元降至90元，验证了装饰者模式的有效性。

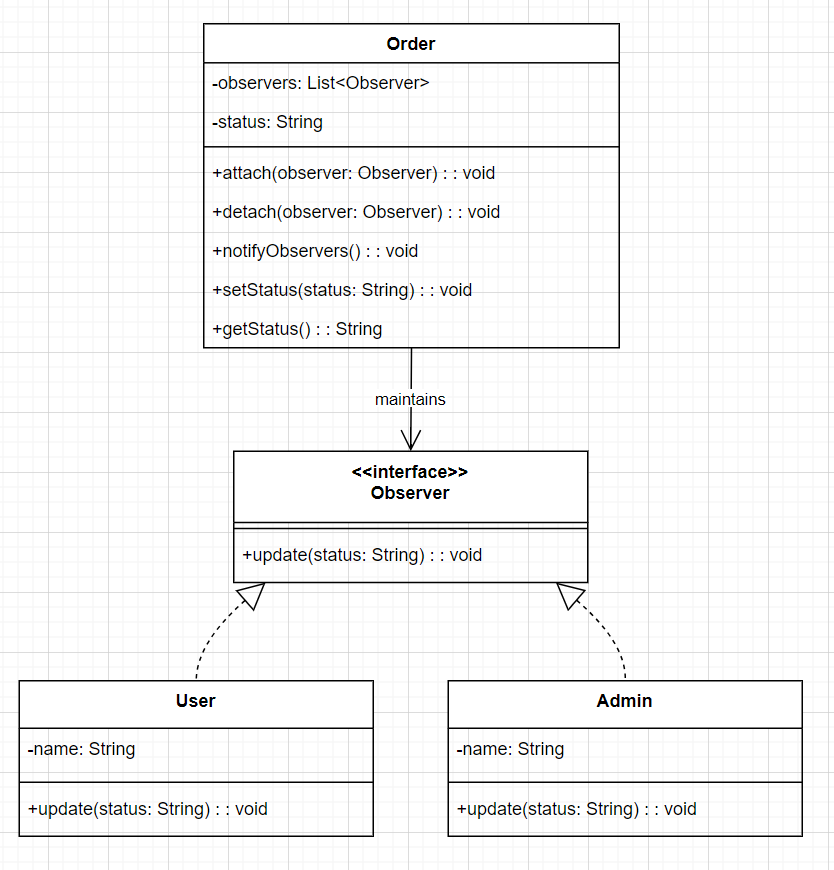
1. 行为型模式：观察者模式 (Observer Pattern)
2. 应用问题描述

在电子商城系统中，当订单状态变化时，用户和管理员都需要及时获取通知。

1. 适用设计模式分析

观察者模式定义对象间的一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。适用于事件触发及通知的场景。

1. UML类图



1. 源程序

import java.util.ArrayList;

import java.util.List;

interface Observer {

    void update(String status);

}

class Order {

    private List<Observer> observers = new ArrayList<>();

    private String status;

    public void attach(Observer observer) {

        observers.add(observer);

    }

    public void detach(Observer observer) {

        observers.remove(observer);

    }

    public void notifyObservers() {

        for (Observer observer : observers) {

            observer.update(status);

        }

    }

    public void setStatus(String status) {

        this.status = status;

        notifyObservers();

    }

    public String getStatus() {

        return status;

    }

}

class User implements Observer {

    private String name;

    public User(String name) {

        this.name = name;

    }

    public void update(String status) {

        System.out.println("User " + name + " notified. New Order Status: " + status);

    }

}

class Admin implements Observer {

    private String name;

    public Admin(String name) {

        this.name = name;

    }

    public void update(String status) {

        System.out.println("Admin " + name + " notified. New Order Status: " + status);

    }

}

public class ObserverPatternDemo {

    public static void main(String[] args) {

        Order order = new Order();

        User user1 = new User("Alice");

        User user2 = new User("Bob");

        Admin admin = new Admin("Admin1");

        order.attach(user1);

        order.attach(user2);

        order.attach(admin);

        order.setStatus("Shipped");

        order.detach(user2);

        order.setStatus("Delivered");

    }

}

1. 实验输出结果及其解释



订单状态变化时，所有观察者都收到了通知，验证了观察者模式的有效性。