Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 2

Discipline: Techniques and Mechanisms of Software Design

Elaborated: FAF-222, Grosu Renat

Checked: asist. univ., Furdui Alexandru

Chișinău 2024

Topic: Structural DPs

Task: Implement atleast 3 structural design patterns in your project.

Objectives:

- Study and understand the Structural Design Patterns.
- Choose a domain, define its main classes/models/entities and choose the appropriate instantiation mechanisms.
- Implement 3 Structural Design Patterns in the project.

THEORY

Structural design patterns have an important role in software engineering since they focus on the organization and relationships between different components of a system. Structural patterns help with the integration and composition of classes and objects to form larger and more flexible structures. They also enable developers to create systems that are easier to understand and maintain and that are also adaptable to future changes.

Below is a description of some key structural design patterns:

1. **Composite Pattern**
   The Composite pattern is designed to represent hierarchies. It allows a group of objects to be treated as a single object which makes it easier to work with recursive structures such as trees especially.
   - **Key Idea:** Users can interact with both individual objects and groups (composites) without needing to distinguish between them.
   - **Use Cases:** This pattern is commonly used in file systems (where directories contain files or other directories)
2. **Decorator Pattern**
   The Decorator pattern is used for dynamically extending the functionality of an object without modifying its original code. Instead of using inheritance, it relies on composition to "wrap" an object and add new behaviors or attributes to it.
   - **Key Idea:** Objects can be enhanced or modified by chaining decorators around them. This ensures the original class remains open for extension but closed for modification, adhering to the Open/Closed Principle.
   - **Use Cases:** Adding functionality like logging, validation, or discounts to existing objects.
3. **Adapter Pattern**
   The Adapter pattern solves the issue of incompatibility between interfaces. It acts like a real life adapter, allowing classes with incompatible interfaces to work together by converting one interface into another that the client expects.

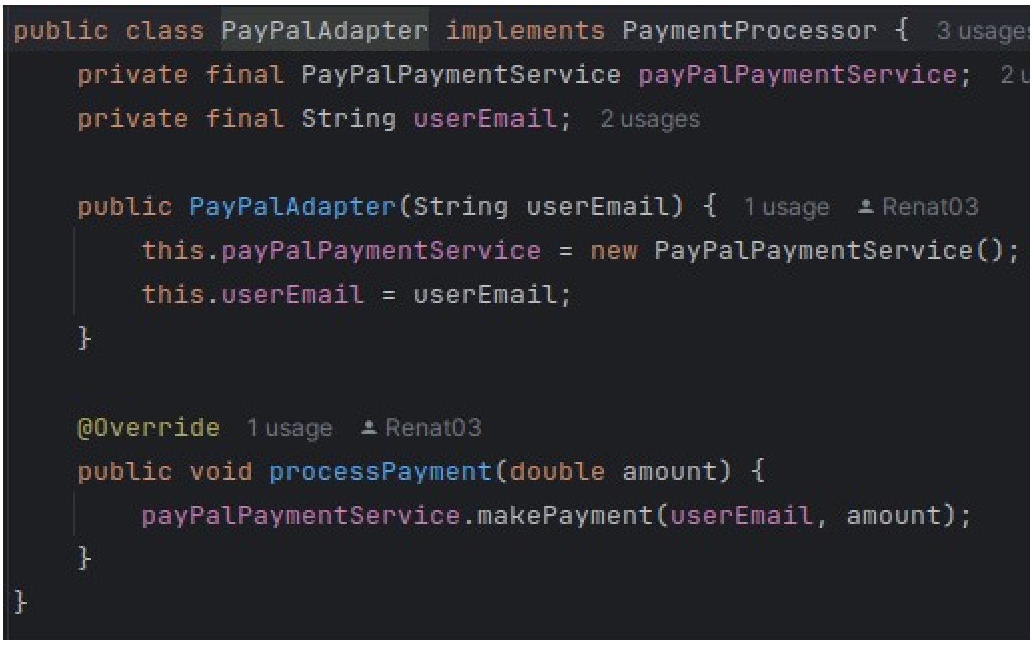
   - **Key Idea:** The adapter "adapts" the interface of one class to match the requirements of another class.
   - **Use Cases:** Integrating third-party APIs into an application.
4. **Flyweight Pattern**
   The Flyweight pattern is designed to minimize memory usage by sharing as much data as possible between similar objects. It is particularly useful in systems where many instances of a class are needed, but only a small portion of their state is unique.
   - **Key Idea:** Separate intrinsic (shared) state from extrinsic (unique) state and store the shared state in order to reduce the overhead made by object creation.
   - **Use Cases:** Systems involving graphical objects (like text rendering, where each character is stored as a shared glyph) or game development (like sharing visual data for entities with identical appearances).
5. **Bridge Pattern**
   The Bridge pattern lets you split an abstraction from its implementation, which allows the two to vary and be developed independently.
   - **Key Idea:** Separate the interface (abstraction) from its implementation, which makes it possible to modify either without affecting the other.
   - **Use Cases:** Device drivers, where the hardware (implementation) and the software interface (abstraction) evolve separately.
6. **Facade Pattern**
   The Facade pattern provides a simplified interface to a complex subsystem, which in turn makes it easier for clients to interact with it.
   - **Key Idea:** Instead of showing the complexity of multiple components, a single facade class offers a simple interface.
   - **Use Cases:** Libraries with complex APIs or systems with multiple subsystems.
7. **Proxy Pattern**
   The Proxy pattern acts as an intermediary for another object, used to control access, add additional functionality, cache and log requests and others.
   - **Key Idea:** The proxy object acts as the real object and handles operations instead of it.
   - **Use Cases:** Virtual proxies for lazy initialization, protection proxies for access control, remote proxies for local execution of a remote service etc.

INTRODUCTION

In this project, I continued my order processing system and I implemented the Decorator, Adapter, and Composite structural design patterns inside it. The Decorator pattern dynamically adds functionality to the order processor, the Adapter pattern allows the system to process payments with the PayPal third-party API, and the Composite pattern manages nested product bundles.

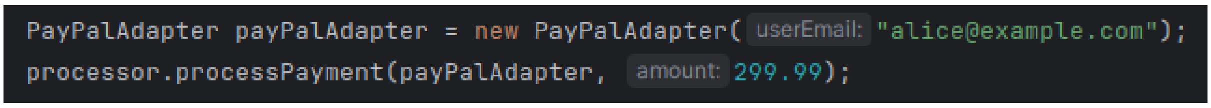
ADAPTER IMPLEMENTATION

The Adapter pattern was used to integrate an external payment service (PayPal) into the existing order processing system. The Adapter (PayPalAdapter), acts as a bridge between the PaymentProcessor interface and PayPalPaymentService, which allows the system to use PayPal for processing payments without modifying the external service's code, even though they have different interfaces.



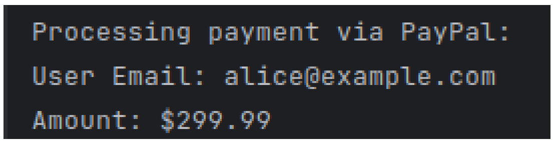
```java
public class PayPalAdapter implements PaymentProcessor {  3 usage
    private final PayPalPaymentService payPalPaymentService;  2 u
    private final String userEmail;  2 usages

    public PayPalAdapter(String userEmail) {  1 usage  Renat03
        this.payPalPaymentService = new PayPalPaymentService();
        this.userEmail = userEmail;
    }

    @Override  1 usage  Renat03
    public void processPayment(double amount) {
        payPalPaymentService.makePayment(userEmail, amount);
    }
}
```

The adapter implements the PaymentProcessor interface and uses an instance of PayPalPaymentService to process payments.



```java
PayPalAdapter payPalAdapter = new PayPalAdapter( userEmail: "alice@example.com");
processor.processPayment(payPalAdapter,  amount: 299.99);
```

Here is the result, which shows that Paypal was successfully integrated into our payment processor.

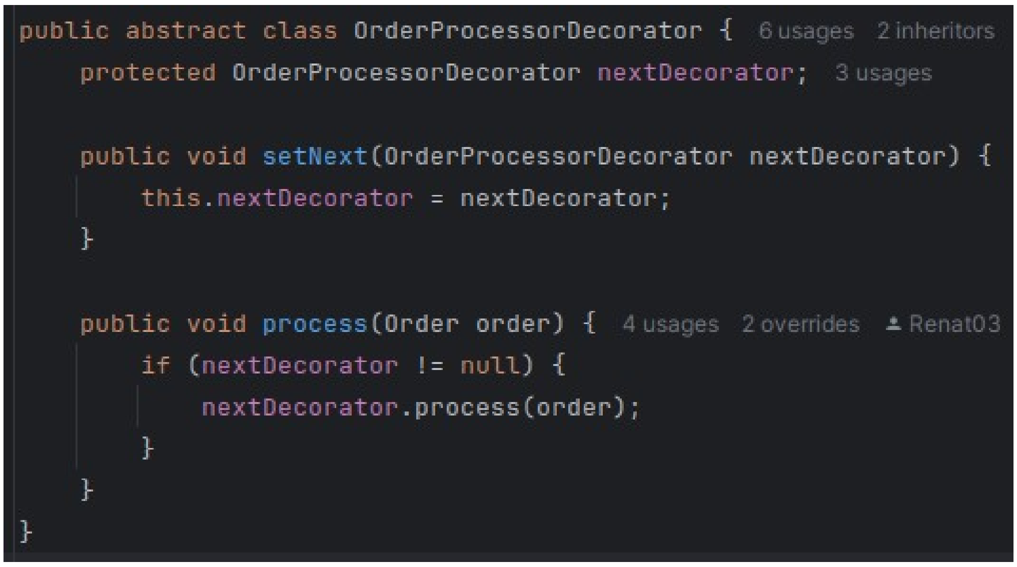


```
Processing payment via PayPal:
User Email: alice@example.com
Amount: $299.99
```

DECORATOR IMPLEMENTATION

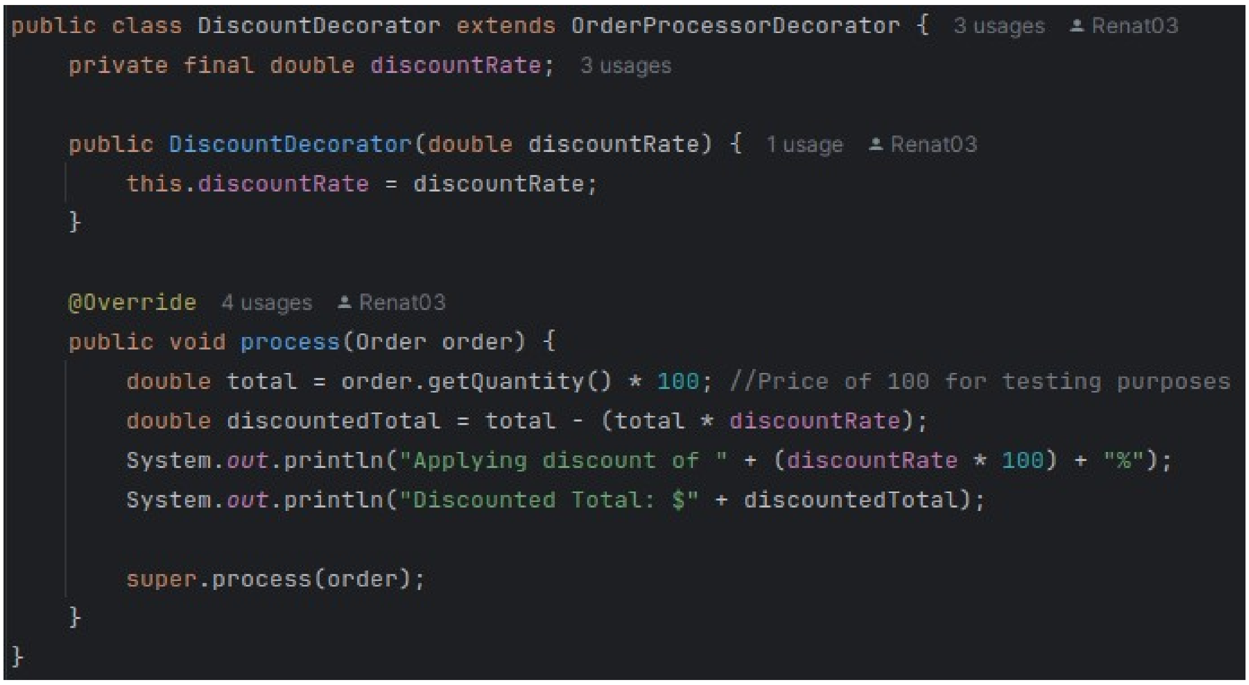
The Decorator pattern was used to extend the functionality of the order processing system dynamically. It allows additional behaviour to be added to the order processor without modifying its existing code, which is in accordance to the Open/Closed Principle.

In this implementation, decorators were used to log additional information and perform extra validations when processing orders.



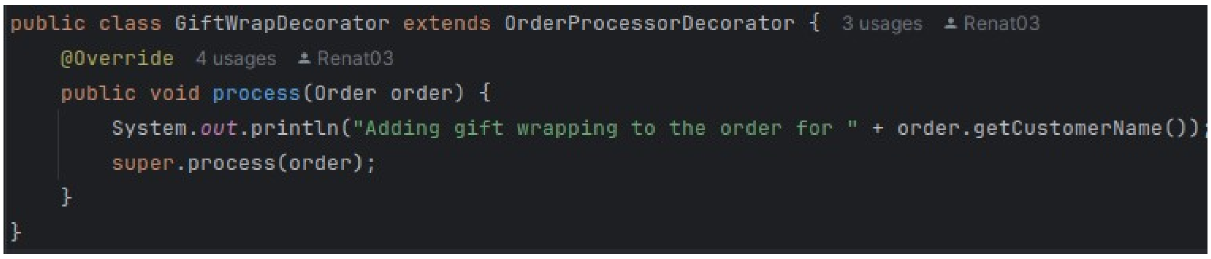
```
public abstract class OrderProcessorDecorator {  6 usages  2 inheritors
    protected OrderProcessorDecorator nextDecorator;  3 usages

    public void setNext(OrderProcessorDecorator nextDecorator) {
        this.nextDecorator = nextDecorator;
    }

    public void process(Order order) {  4 usages  2 overrides  Renat03
        if (nextDecorator != null) {
            nextDecorator.process(order);
        }
    }
}
```

This is
the base decorator class that defines the structure for all concrete decorators. It contains a reference to the next decorator in the chain (nextDecorator) as well as 2 methods: a method called setNext which assigns the next decorator in the chain, and a method called process which calls the process method of the next decorator to make sure all of the decorators in the chain are executed in a sequence.



```
public class DiscountDecorator extends OrderProcessorDecorator {  3 usages  Renat03
    private final double discountRate;  3 usages

    public DiscountDecorator(double discountRate) {  1 usage  Renat03
        this.discountRate = discountRate;
    }

    @Override  4 usages  Renat03
    public void process(Order order) {
        double total = order.getQuantity() * 100; //Price of 100 for testing purposes
        double discountedTotal = total - (total * discountRate);
        System.out.println("Applying discount of " + (discountRate * 100) + "%");
        System.out.println("Discounted Total: $" + discountedTotal);

        super.process(order);
    }
}
```

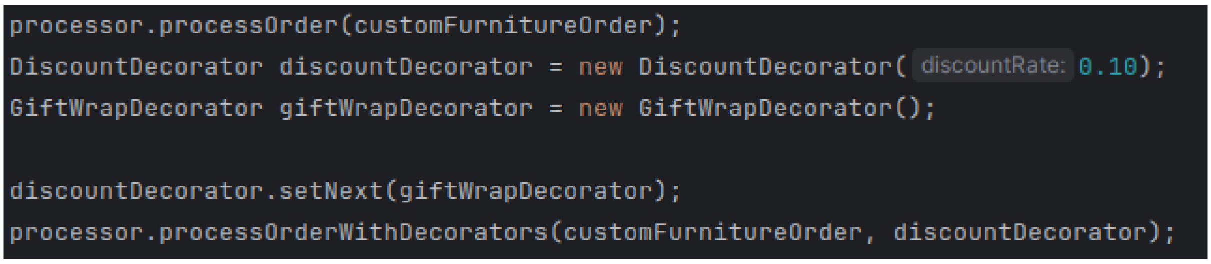
The DiscountDecorator is my implementation of a concrete decorator, which applies a discount to the order based on a specified discount rate. It calculates the total price, applies the discount, and prints the discounted total.



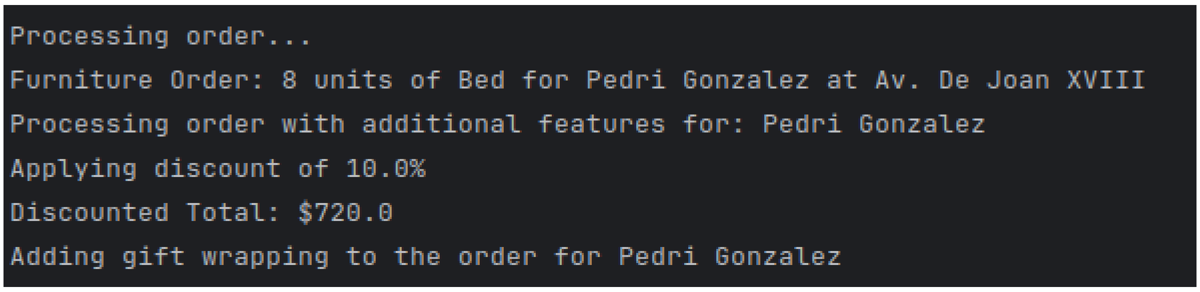
```
public class GiftWrapDecorator extends OrderProcessorDecorator {  3 usages  Renat03
    @Override  4 usages  Renat03
    public void process(Order order) {
        System.out.println("Adding gift wrapping to the order for " + order.getCustomerName());
        super.process(order);
    }
}
```

The GiftWrapDecorator is my other implementation of a concrete decorator, which adds a message that indicates that the order is being gift-wrapped.

Each of these decorators are linked in a chain, and the process method of a decorator calls the process method of the next decorator, thus allowing multiple behaviours to be applied to the same order without altering the base functionality.



```
processor.processOrder(customFurnitureOrder);
DiscountDecorator discountDecorator = new DiscountDecorator( discountRate: 0.10);
GiftWrapDecorator giftWrapDecorator = new GiftWrapDecorator();

discountDecorator.setNext(giftWrapDecorator);
processor.processOrderWithDecorators(customFurnitureOrder, discountDecorator);
```

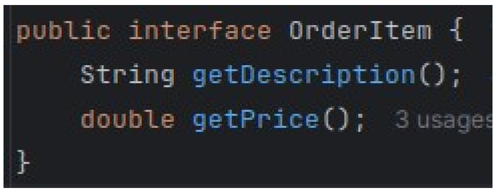


```
Processing order...
Furniture Order: 8 units of Bed for Pedri Gonzalez at Av. De Joan XVIII
Processing order with additional features for: Pedri Gonzalez
Applying discount of 10.0%
Discounted Total: $720.0
Adding gift wrapping to the order for Pedri Gonzalez
```

Here is the output of my decorator code.

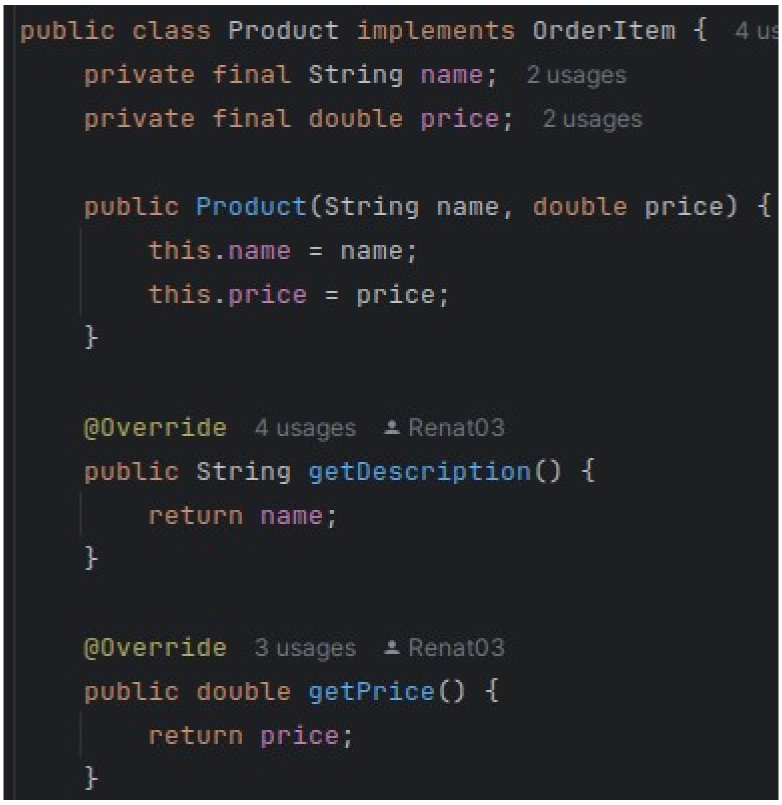
COMPOSITE IMPLEMENTATION

The Composite Pattern is implemented in this project to represent hierarchies of OrderItem objects, where individual products and bundles of them can be treated the same. This pattern is especially useful for managing orders with nested bundles.



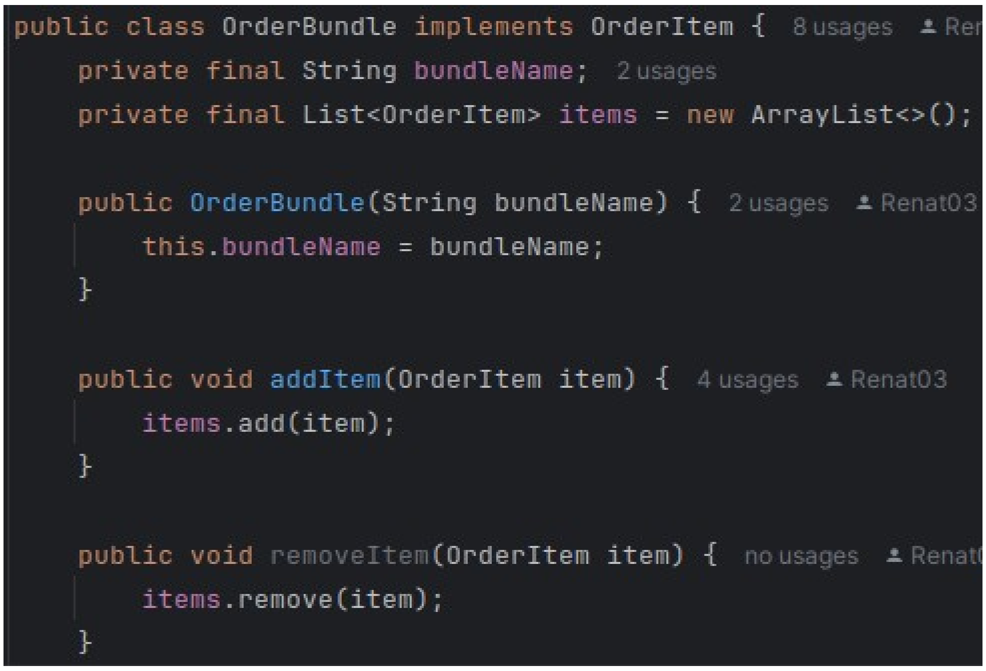
```
public interface OrderItem {
    String getDescription();
    double getPrice();  3 usages
}
```

The OrderItem interface defines the common functionality for both individual items (e.g., products) and composite items (e.g., bundles).



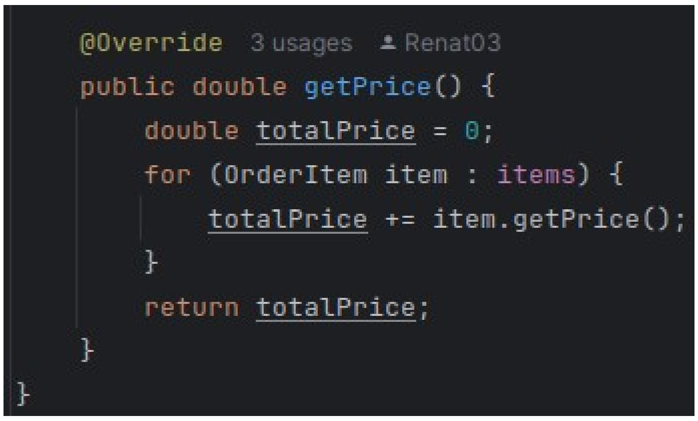
```
public class Product implements OrderItem {  4 us
    private final String name;  2 usages
    private final double price;  2 usages

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override  4 usages  Renat03
    public String getDescription() {
        return name;
    }

    @Override  3 usages  Renat03
    public double getPrice() {
        return price;
    }
```

The Product class represents the individual products of my order system. the non-composite elements in my structure.



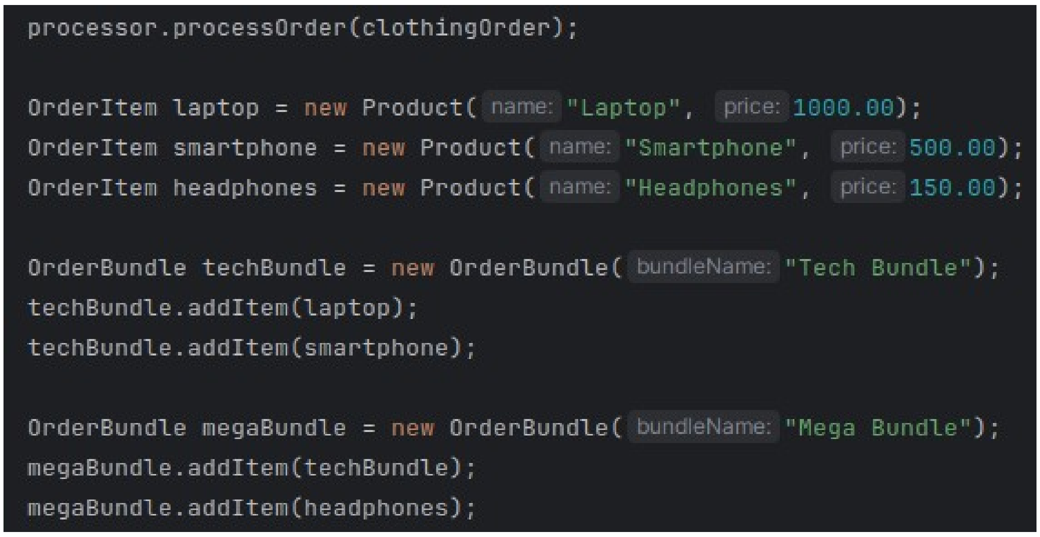
```
public class OrderBundle implements OrderItem {  8 usages  ≗ Rer
    private final String bundleName;  2 usages
    private final List<OrderItem> items = new ArrayList<>();

    public OrderBundle(String bundleName) {  2 usages  ≗ Renat03
        this.bundleName = bundleName;
    }

    public void addItem(OrderItem item) {  4 usages  ≗ Renat03
        items.add(item);
    }

    public void removeItem(OrderItem item) {  no usages  ≗ Renat(
        items.remove(item);
    }
```

The OrderBundle class represents composite items that can contain both individual products and other bundles. It maintains a list of OrderItem objects, and new items can be added and removed to it.

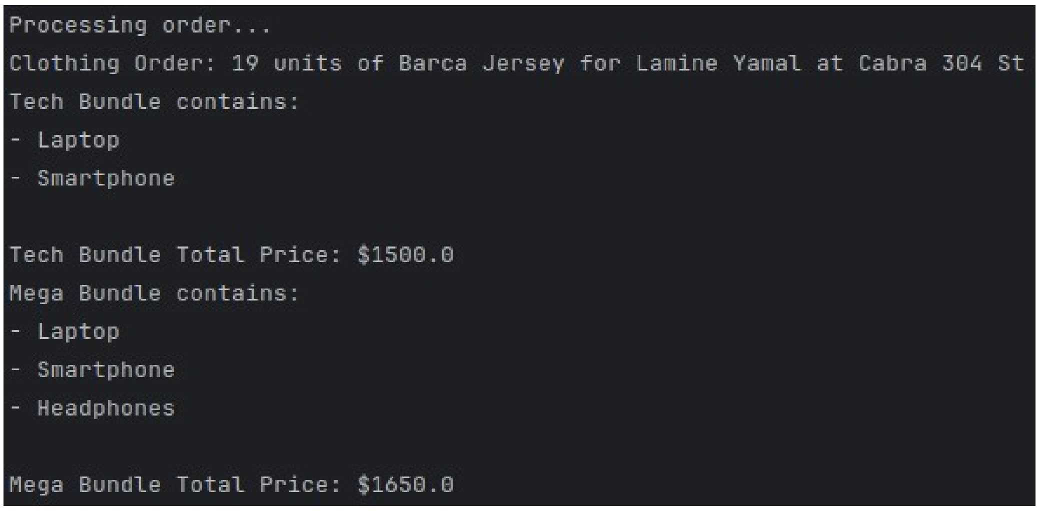


```
    @Override  3 usages  ≗ Renat03
    public double getPrice() {
        double totalPrice = 0;
        for (OrderItem item : items) {
            totalPrice += item.getPrice();
        }
        return totalPrice;
    }
}
```

Here is the getPrice method of my OrderBundle class which differs from the one from the simple Product class as it sums up the prices of all of the products in the bundle.



```
processor.processOrder(clothingOrder);

OrderItem laptop = new Product( name: "Laptop",  price: 1000.00);
OrderItem smartphone = new Product( name: "Smartphone",  price: 500.00);
OrderItem headphones = new Product( name: "Headphones",  price: 150.00);

OrderBundle techBundle = new OrderBundle( bundleName: "Tech Bundle");
techBundle.addItem(laptop);
techBundle.addItem(smartphone);

OrderBundle megaBundle = new OrderBundle( bundleName: "Mega Bundle");
megaBundle.addItem(techBundle);
megaBundle.addItem(headphones);
```



```
Processing order...
Clothing Order: 19 units of Barca Jersey for Lamine Yamal at Cabra 304 St
Tech Bundle contains:
- Laptop
- Smartphone

Tech Bundle Total Price: $1500.0
Mega Bundle contains:
- Laptop
- Smartphone
- Headphones

Mega Bundle Total Price: $1650.0
```

Here is the output of my composite implementation.

CONCLUSION

In this laboratory work, I implemented three structural design patterns: Decorator, Adapter, and Composite. These patterns teached me how to enhance system functionality, enable compatibility between incompatible interfaces, and manage complex hierarchical structures eficiently. By using these patterns, the project became more flexible, and easier to extend for future purposes. Overall, it was a valuable experience that strenghtenede my understanding of applying design principles in real projects.