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

REPORT

Laboratory work No. 1

Discipline: Techniques and Mechanisms of Software Design

Elaborated: FAF-222, Grosu Renat

Checked: asist. univ., Furdui Alexandru

Chișinău 2024

Topic: Creational DPs

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

THEORY

Creational design patterns are foundational principles in software engineering that provide solutions for object creation, aiming to increase flexibility and manage complexity in software systems. These patterns allow developers to handle various instantiation mechanisms without making the code overly dependent on specific classes, thus making applications easier to adapt, extend, and maintain. Creational patterns are especially valuable in scenarios where objects need to be created in different ways based on context, configurations, or constraints. Below is an overview of five core creational design patterns, each addressing different aspects of object instantiation.

1. Singleton Pattern

The Singleton pattern ensures that a class has only one instance and provides a global access point to that instance. This is particularly useful in situations where a single instance is required to coordinate actions across the system, such as logging, configuration management, or managing connections.

- **Purpose**: To restrict the instantiation of a class to a single instance and provide a global point of access to it.
- **Benefits**: Reduces memory usage by avoiding multiple instances of the same class and ensures consistent access to the single instance.
- **Example**: A configuration manager in an application, where multiple parts of the application require access to the same configuration settings.

Implementation Overview: In Singleton, the constructor is private, and the class provides a static method that returns the single instance. In multi-threaded applications, Singleton needs to be thread-safe to prevent concurrent access issues.

2. Builder Pattern

The Builder pattern simplifies object creation by separating the construction of a complex object from its representation. Instead of requiring multiple constructors or complex factory methods, the Builder pattern allows step-by-step construction of objects, particularly when the object has many optional attributes.

- **Purpose**: To construct a complex object incrementally, allowing for greater control over the construction process.
- **Benefits**: Improves readability and usability by enabling a clear and flexible way to create objects with different configurations without a bloated constructor.

- **Example**: An `Order` class where different properties like customer name, address, product type, and quantity can be set conditionally based on user input.

Implementation Overview: The Builder pattern defines a separate `Builder` class that provides methods to configure and build an object. Once configuration is complete, a `build()` method instantiates the final object.

3. Factory Method Pattern

The Factory Method pattern defines an interface for creating objects but allows subclasses to decide which class to instantiate. This promotes loose coupling by enabling clients to work with an abstract interface, while the concrete instantiation is handled by a subclass.

- **Purpose**: To create objects without specifying the exact class of object that will be created.
- **Benefits**: Increases flexibility and scalability by isolating the object creation process, making it easier to introduce new types without modifying existing code.
- **Example**: An application that generates different types of orders (e.g., electronics, furniture) where the exact type is determined at runtime.

Implementation Overview: In Factory Method, an abstract class defines a method for creating an object, but it's the subclass that provides the specific instantiation. This allows the creation of different types of objects based on context.

4. Abstract Factory Pattern

The Abstract Factory pattern provides an interface for creating families of related objects without specifying their concrete classes. It allows the client to create products that are designed to work together, promoting consistency and decoupling object creation from the main logic.

- **Purpose**: To provide an interface for creating a group of related or dependent objects without specifying the concrete classes.
- **Benefits**: Enhances flexibility by grouping related objects under a single factory, facilitating the replacement or extension of product families without modifying client code.
- **Example**: A factory for creating orders that may include both products (e.g., electronics, furniture) and associated delivery methods or packaging types.

Implementation Overview: The Abstract Factory pattern involves creating an interface with methods to create each type of product. Concrete factory classes implement this interface and instantiate specific product classes.

5. Prototype Pattern

The Prototype pattern allows objects to be created by copying an existing object rather than instantiating a new one. This is useful when the cost of creating a new object from scratch is more significant than creating a copy of an existing one.

- **Purpose**: To create a new object by cloning an existing object, leveraging its state as a prototype for new instances.
- **Benefits**: Reduces the cost of instantiating new objects and can simplify the creation process for objects with complex configurations.
- **Example**: A character in a game or an object in an application that can be cloned with minor changes, thus preserving much of the original configuration.

Implementation Overview: In Prototype, a class implements a method to return a copy of itself. The `clone()` method can be overridden to implement deep or shallow copies, depending on the use case.

INTRODUCTION

In this project, I developed a sample order processing system to demonstrate the practical application of several creational design patterns: Singleton, Builder, Factory Method, and Abstract Factory. The system supports the creation and management of different types of orders, such as electronics and furniture, and allows for custom configurations of orders, along with a centralized order processor.

The **Singleton** pattern is used to ensure a single instance of the order processor, allowing centralized management of all orders. The **Builder** pattern provides a flexible way to configure and build orders with specific details like customer information, product type, and quantity. The **Factory Method** and **Abstract Factory** patterns are used to encapsulate the creation of different types of orders, enabling the system to be easily extended to support new order types without altering existing code.

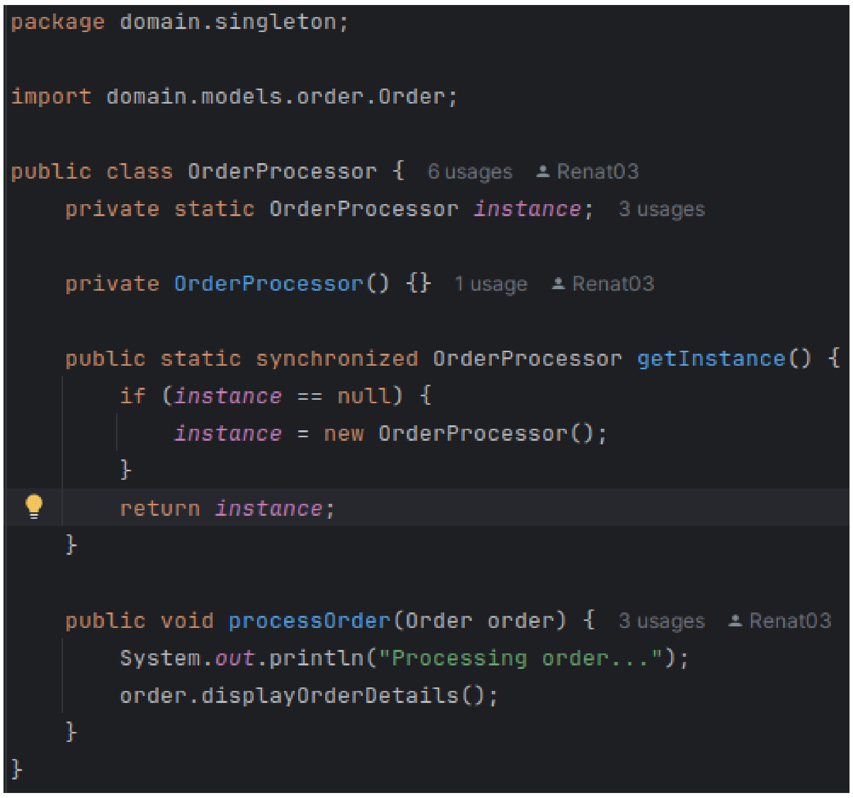
This approach ensures the system is modular, scalable, and adaptable, making it easier to add new features and maintain over time.

SINGLETON IMPLEMENTATION

The OrderProcessor class exemplifies the Singleton design pattern while adhering to the Single Responsibility Principle (SRP). Its primary responsibility is to manage the processing of customer orders. The class is designed to focus solely on order-related functionalities without intertwining with other components, ensuring clarity and maintainability.

- **Singleton Implementation:** The `OrderProcessor` class ensures that only one instance exists throughout the application. This design choice guarantees consistent order processing and avoids the complications that can arise from multiple instances managing orders simultaneously.

- **Private Constructor:** By using a private constructor, the class prevents external instantiation, enforcing a single point of access to the order processing logic.
- **Synchronized getInstance Method:** The `getInstance` method is synchronized to ensure thread safety when accessing the singleton instance. This allows for safe usage in multi-threaded environments, where multiple threads may attempt to retrieve the instance concurrently.
- **Process Order Method:** The `processOrder` method is responsible for handling the details of an order. It takes an `Order` object as input, processes it, and displays relevant order information, keeping the focus on the order management aspect without delving into payment or menu management.

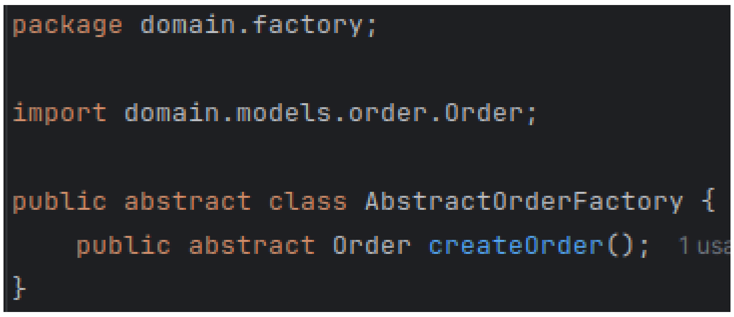


```
package domain.singleton;

import domain.models.order.Order;

public class OrderProcessor {  6 usages  Renat03
    private static OrderProcessor instance;  3 usages

    private OrderProcessor() {}  1 usage  Renat03

    public static synchronized OrderProcessor getInstance() {
        if (instance == null) {
            instance = new OrderProcessor();
        }
        return instance;
    }

    public void processOrder(Order order) {  3 usages  Renat03
        System.out.println("Processing order...");
        order.displayOrderDetails();
    }
}
```

FACTORY IMPLEMENTATION

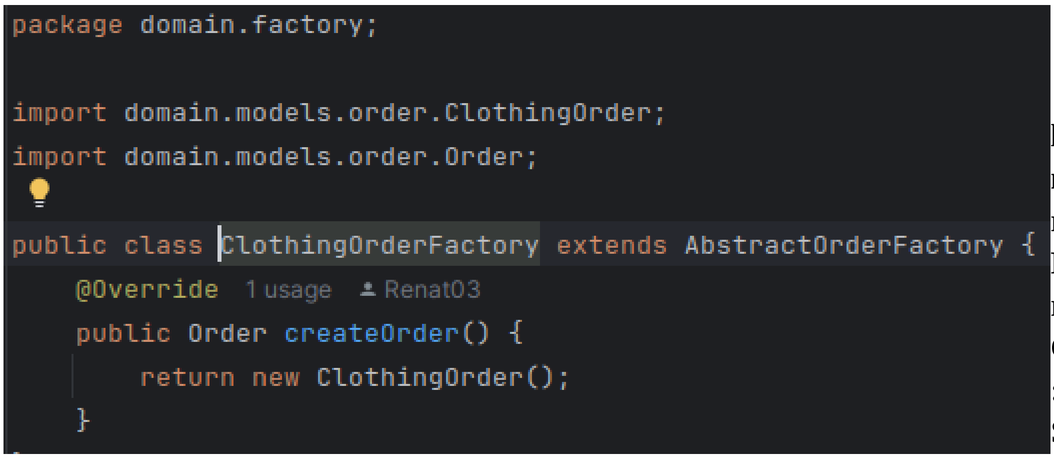
The Factory design pattern is utilized in the project to encapsulate the instantiation logic for different types of orders. This approach promotes loose coupling and adheres to the Open/Closed Principle, allowing the system to be easily extended with new order types without modifying existing code.

- **AbstractOrderFactory Class:** This abstract class serves as the blueprint for creating different types of orders. It defines a method, createOrder(), which must be implemented by any concrete factory class that extends it. By using an abstract class, the design ensures that all specific order factories adhere to a common interface.



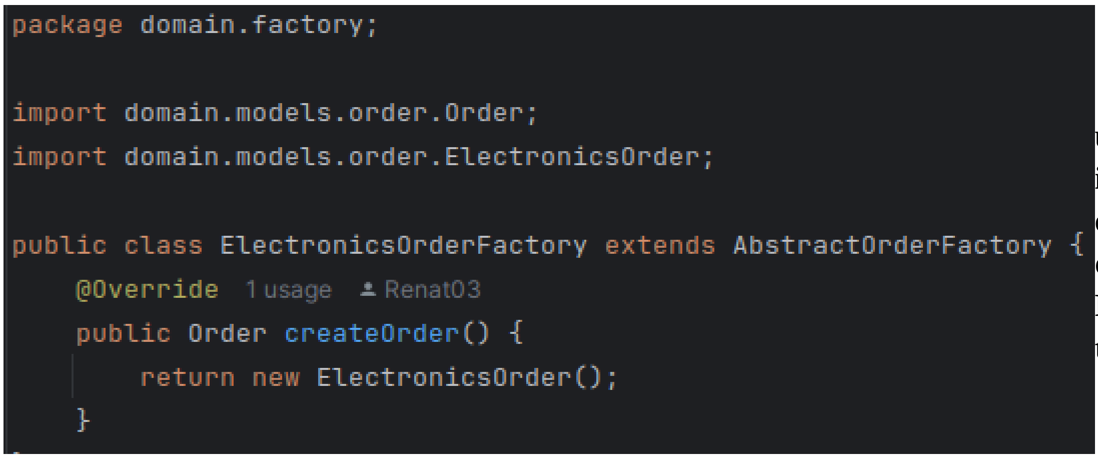
```
package domain.factory;

import domain.models.order.Order;

public abstract class AbstractOrderFactory {
    public abstract Order createOrder();  1 usa
}
```

- **ClothingOrderFactory Class:** This concrete factory extends the AbstractOrderFactory and implements the createOrder() method to instantiate ClothingOrder objects. By centralizing the creation logic, the system maintains a clear separation between order types and their instantiation.



```
package domain.factory;

import domain.models.order.ClothingOrder;
import domain.models.order.Order;

public class ClothingOrderFactory extends AbstractOrderFactory {
    @Override  1 usage  Renat03
    public Order createOrder() {
        return new ClothingOrder();
    }
```

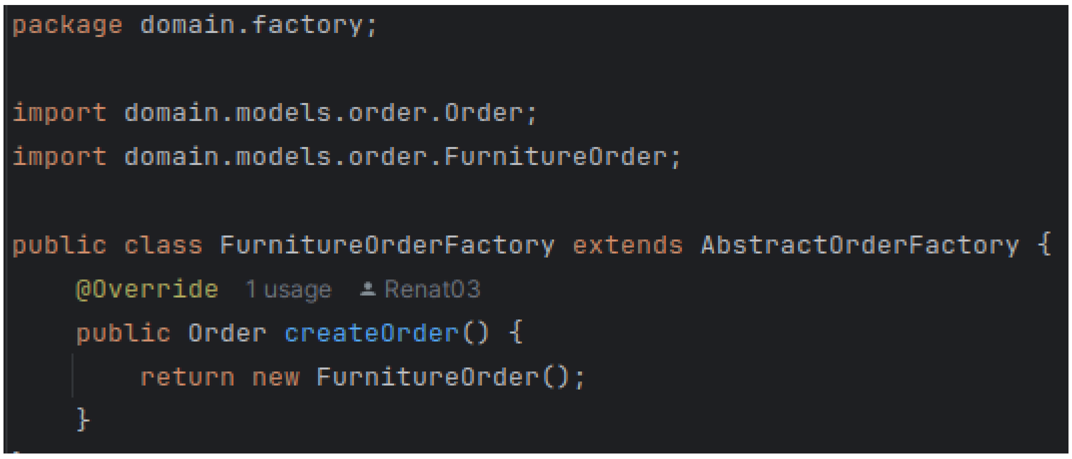
- **ElectronicsOrderFactory Class:** Similar to the ClothingOrderFactory, this class provides the implementation for creating ElectronicsOrder objects. By utilizing the factory pattern, the application can easily introduce new order types, ensuring scalability without impacting existing order processing logic.



```
package domain.factory;

import domain.models.order.Order;
import domain.models.order.ElectronicsOrder;

public class ElectronicsOrderFactory extends AbstractOrderFactory {
    @Override  1 usage  Renat03
    public Order createOrder() {
        return new ElectronicsOrder();
    }
```

- **FurnitureOrderFactor**

y Class: This factory is responsible for creating instances of FurnitureOrder. Each factory class encapsulates the creation of a specific order type, promoting the Single Responsibility Principle by focusing solely on the instantiation logic for its respective order.

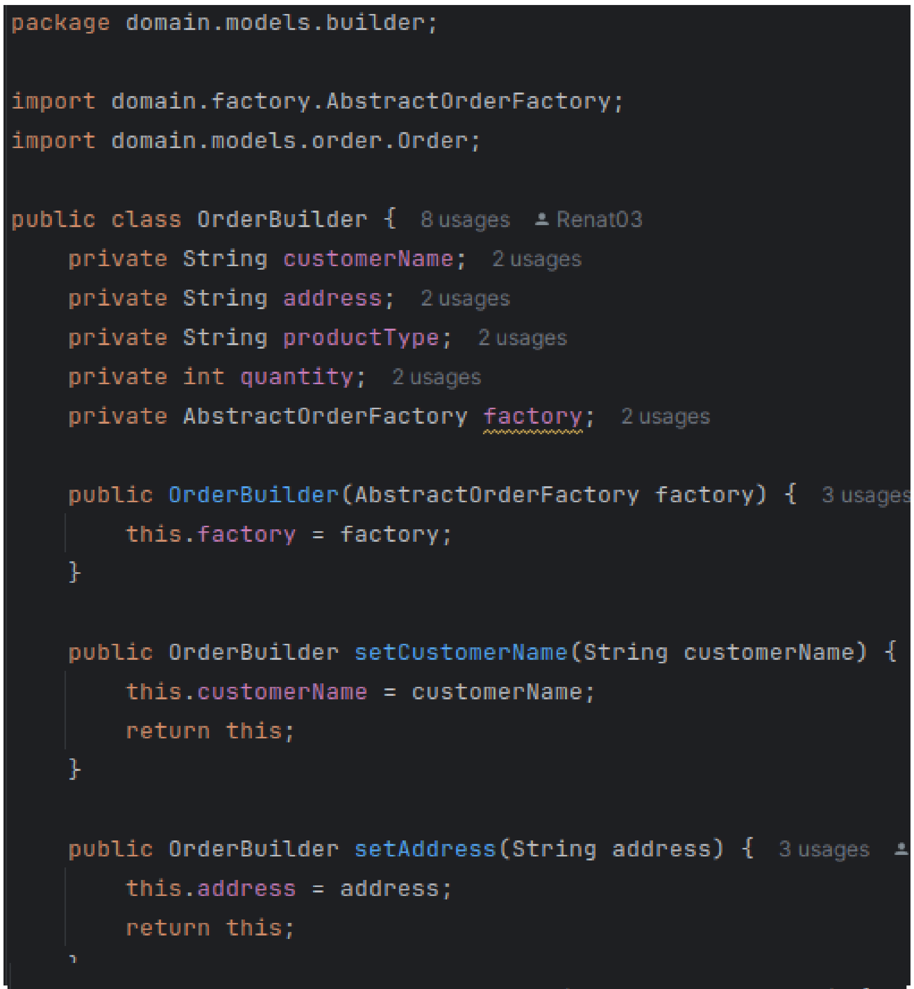


```
package domain.factory;

import domain.models.order.Order;
import domain.models.order.FurnitureOrder;

public class FurnitureOrderFactory extends AbstractOrderFactory {
    @Override  1 usage  Renat03
    public Order createOrder() {
        return new FurnitureOrder();
    }
```

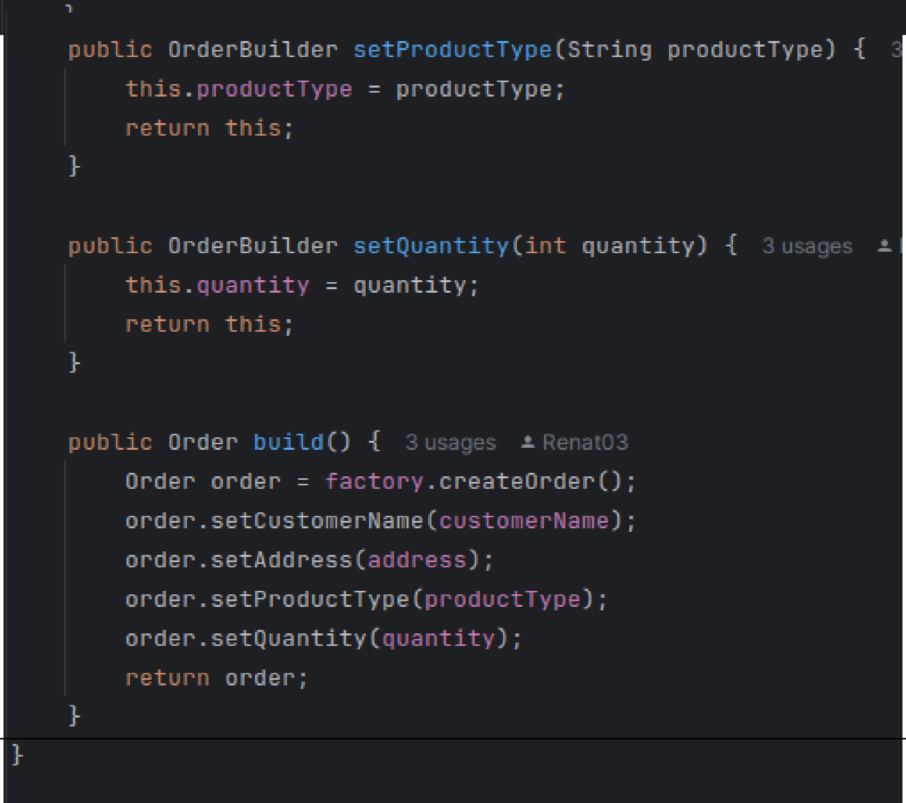
BUILDER IMPLEMENTATION

The Builder design pattern is utilized in the project to facilitate the construction of complex Order objects. This pattern promotes clarity and flexibility in object creation, allowing for step-by-step construction of Order instances while maintaining immutability and readability.

- **OrderBuilder Class:** The OrderBuilder class provides a fluent interface for constructing Order objects. It contains fields for essential order attributes, such as customerName, address, productType, and quantity, allowing for detailed customization during the order creation process.
    - **Constructor:** The builder accepts an instance of AbstractOrderFactory, enabling the dynamic creation of specific order types while decoupling the order construction logic from the actual order types.
    - **Setters:** The class provides setter methods (setCustomerName, setAddress, setProductType, setQuantity) that return the builder instance itself. This fluent interface allows method chaining, making it intuitive to configure the order's attributes.
- **Build Method:** The build() method orchestrates the construction of the Order object. It calls the factory's createOrder() method to instantiate the specific order type and then sets the order's attributes using the values provided via the builder's setters.

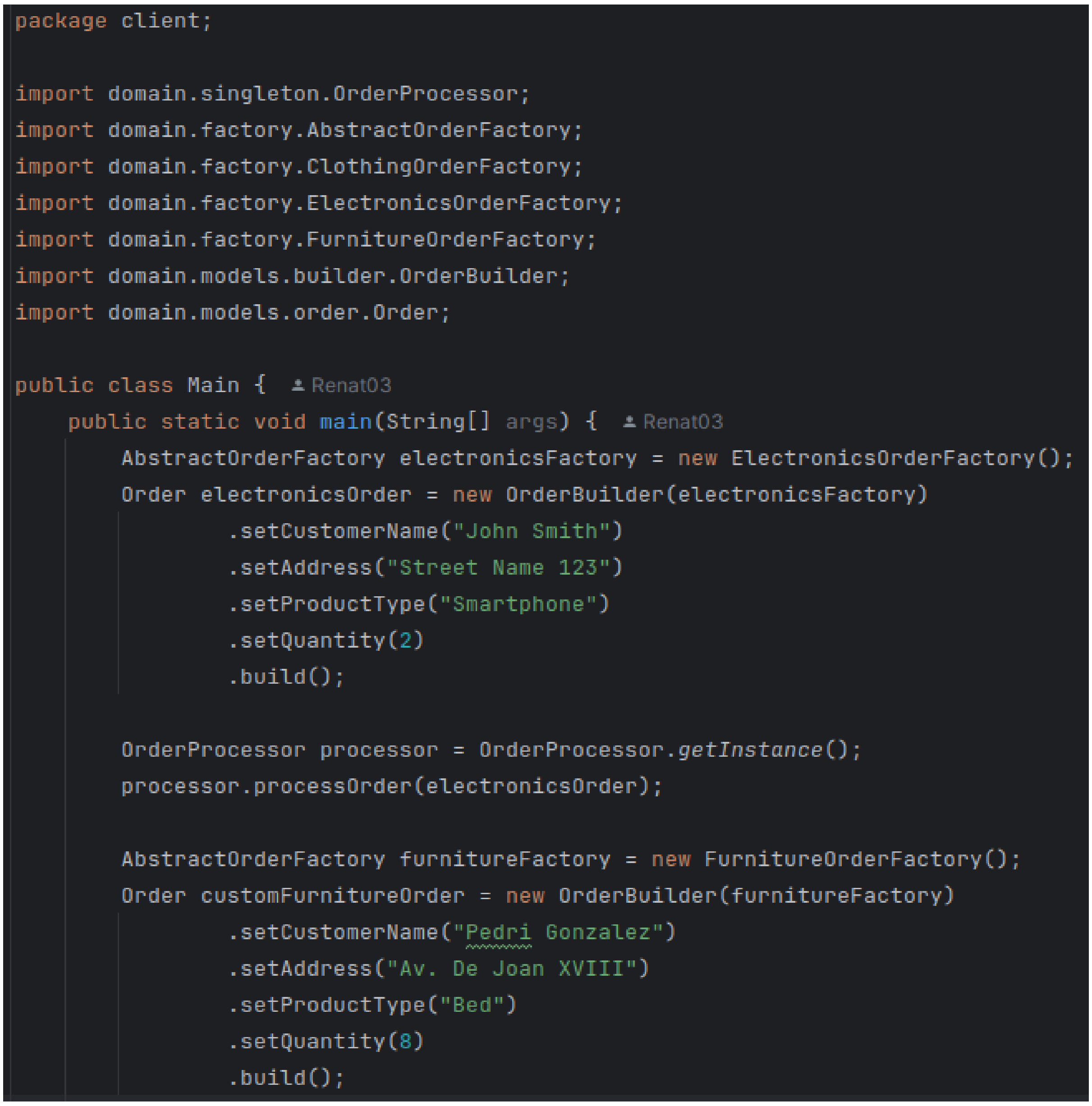


```
package domain.models.builder;

import domain.factory.AbstractOrderFactory;
import domain.models.order.Order;

public class OrderBuilder {  8 usages  Renat03
    private String customerName;  2 usages
    private String address;  2 usages
    private String productType;  2 usages
    private int quantity;  2 usages
    private AbstractOrderFactory factory;  2 usages

    public OrderBuilder(AbstractOrderFactory factory) {  3 usages
        this.factory = factory;
    }

    public OrderBuilder setCustomerName(String customerName) {
        this.customerName = customerName;
        return this;
    }

    public OrderBuilder setAddress(String address) {  3 usages
        this.address = address;
        return this;
    }
```

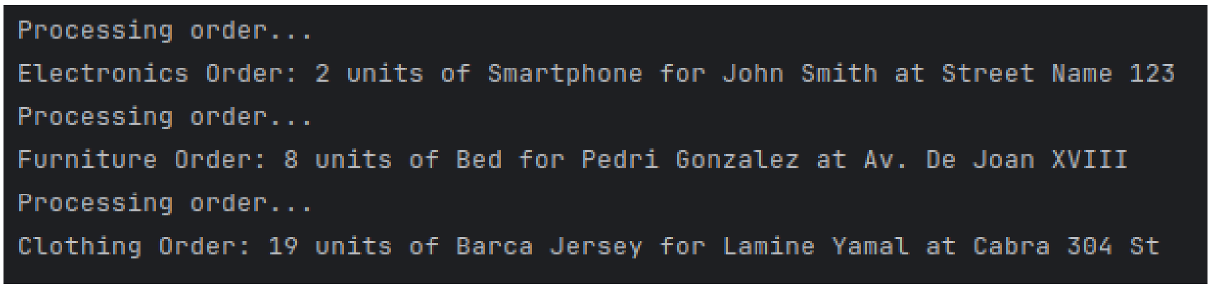


```
    public OrderBuilder setProductType(String productType) {  3
        this.productType = productType;
        return this;
    }

    public OrderBuilder setQuantity(int quantity) {  3 usages
        this.quantity = quantity;
        return this;
    }

    public Order build() {  3 usages  Renat03
        Order order = factory.createOrder();
        order.setCustomerName(customerName);
        order.setAddress(address);
        order.setProductType(productType);
        order.setQuantity(quantity);
        return order;
    }
}
```

RESULTS



```java
package client;

import domain.singleton.OrderProcessor;
import domain.factory.AbstractOrderFactory;
import domain.factory.ClothingOrderFactory;
import domain.factory.ElectronicsOrderFactory;
import domain.factory.FurnitureOrderFactory;
import domain.models.builder.OrderBuilder;
import domain.models.order.Order;

public class Main {  Renat03
    public static void main(String[] args) {  Renat03
        AbstractOrderFactory electronicsFactory = new ElectronicsOrderFactory();
        Order electronicsOrder = new OrderBuilder(electronicsFactory)
                .setCustomerName("John Smith")
                .setAddress("Street Name 123")
                .setProductType("Smartphone")
                .setQuantity(2)
                .build();

        OrderProcessor processor = OrderProcessor.getInstance();
        processor.processOrder(electronicsOrder);

        AbstractOrderFactory furnitureFactory = new FurnitureOrderFactory();
        Order customFurnitureOrder = new OrderBuilder(furnitureFactory)
                .setCustomerName("Pedri Gonzalez")
                .setAddress("Av. De Joan XVIII")
                .setProductType("Bed")
                .setQuantity(8)
                .build();
```



```
Processing order...
Electronics Order: 2 units of Smartphone for John Smith at Street Name 123
Processing order...
Furniture Order: 8 units of Bed for Pedri Gonzalez at Av. De Joan XVIII
Processing order...
Clothing Order: 19 units of Barca Jersey for Lamine Yamal at Cabra 304 St
```

CONCLUSION

By applying the Singleton and Factory design patterns in this project, the code achieves a high level of modularity and maintainability. The Singleton pattern ensures that critical components, such as order processing, are instantiated only once, promoting resource efficiency and consistency throughout the application. Meanwhile, the Factory pattern encapsulates the creation logic of various order types, allowing for easy extension and integration of new order categories without modifying existing code.

The Builder pattern further enhances the clarity of object construction, providing a flexible and intuitive way to create complex Order objects while maintaining readability and reducing the likelihood of errors.

Overall, the implementation of these design patterns contributes to a clean, scalable, and robust architecture for the restaurant menu management system, enabling seamless future enhancements and modifications while minimizing the risk of introducing bugs.