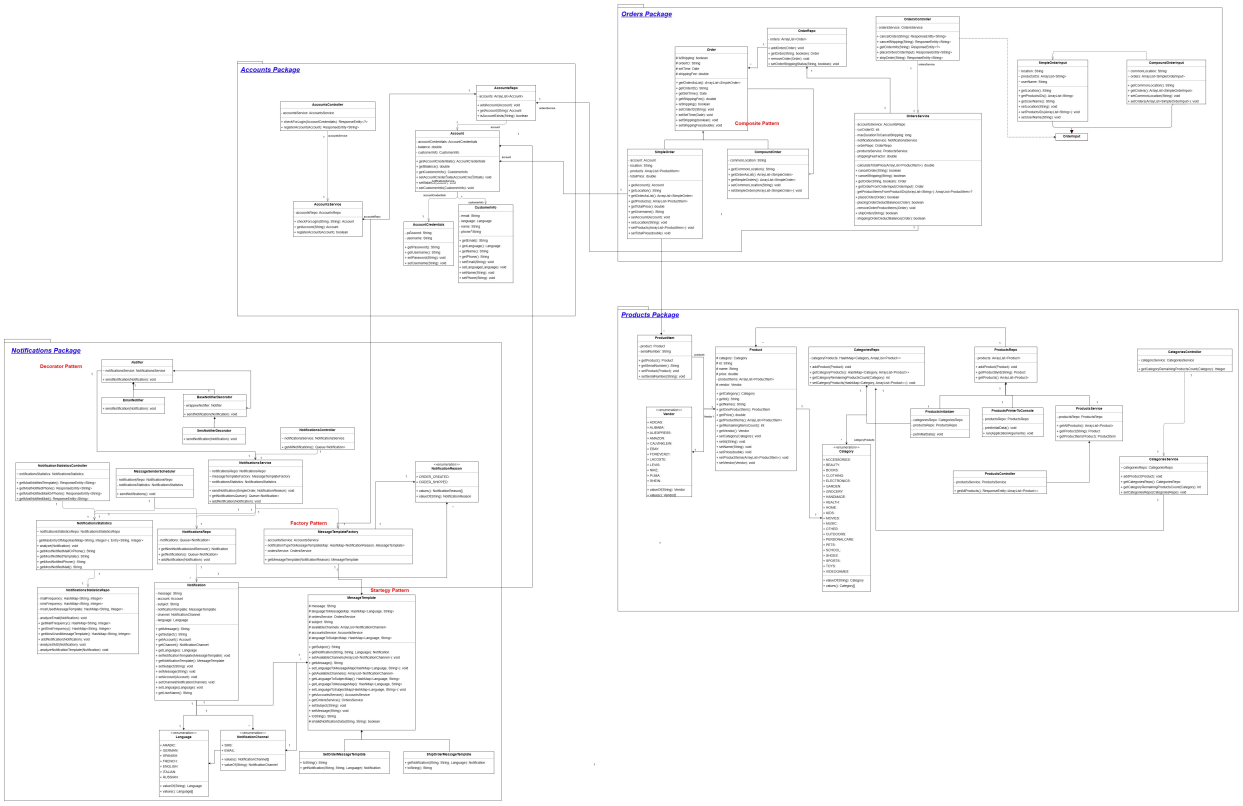
Final SDS

Class diagram design



Class diagram Explanation

Composite Pattern:

The Composite Pattern is a structural design pattern that allows clients to treat individual objects and compositions of objects uniformly. In this pattern, a single interface is used to represent both individual objects (leaves) and compositions of objects (composites). This is achieved through a common base class or interface.

Participating Classes:

1. **Order (Component):**
   - The common interface or base class that declares the operations that can be performed on both simple and compound orders.
2. **SimpleOrder (Leaf):**
   - Inherits from Order and represents a simple order. It contains the specific details and behavior of a standalone order.
3. **CompoundOrder (Composite):**
   - Inherits from Order and represents a compound order. It can contain a collection of simple orders and/or other compound orders. It implements the common operations defined in the Order interface.

Justification:

- **Uniform Treatment:**
  - The Composite Pattern ensures that both simple and compound orders can be treated uniformly. Clients can work with an order hierarchy without needing to distinguish between individual orders and compositions of orders.

- **Common Interface:**
  - Having a common interface (Order) ensures that both simple and compound orders follow the same set of rules, making it easier to add new order types in the future without affecting existing code.

Decorator Pattern:

The Decorator Pattern is a structural design pattern that allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class. It involves a set of classes where each class adds its own functionality to the base class, forming a chain of decorators.

Participating Classes:

1. **Notifier (Component Interface):**
   - Acts as the interface defining the operations for sending notifications. This could be an abstract class or interface that `EmailNotifier` and decorator classes implement.
2. **EmailNotifier (Concrete Component):**
   - Implements the base functionality of sending email notifications. This is the core class that other decorators will enhance.
3. **BaseNotifierDecorator (Decorator Abstract Class):**
   - Serves as the base class for all decorators. It contains a reference to a `Notifier` object and implements the same interface as the `Notifier`.
4. **SmsNotifierDecorator (Concrete Decorator):**
   - Extends `BaseNotifierDecorator` and adds functionality to send SMS notifications. It wraps around a `Notifier` object (can be an `EmailNotifier` or another decorator).

Justification:

- **Dynamic Behavior Extension:**
  - The Decorator Pattern allows for the dynamic addition of new behaviors to objects at runtime. For instance, you can have an `EmailNotifier` as the core notifier and add features like SMS notifications dynamically by using decorators.
- **Single Responsibility Principle:**
  - Each decorator has a single responsibility, making it easier to understand, maintain, and extend the functionality without altering the original class.
- **Open/Closed Principle:**
  - The pattern follows the open/closed principle, allowing for new functionality to be added without modifying existing code.
- **Code Reusability:**
  - Decorators can be reused in different combinations, promoting reusability of individual behaviours.

Factory Pattern:

The Factory Pattern is a creational design pattern that provides an interface for creating instances of a class, but allows subclasses to alter the type of instances that will be created. It involves an interface or abstract class for creating objects and concrete classes that implement the interface.

Participating Classes:

1. **MessageTemplate (Product):**
   - Represents the interface or abstract class for the message templates. It defines the common interface for all message templates.
2. **SetOrderMessageTemplate and ShipOrderMessageTemplate (Concrete Products):**
   - Extend the `MessageTemplate` abstract class. Each class represents a specific type of message template.
3. **MessageTemplateFactory (Creator):**
   - Defines the method for creating message templates. It can be an interface or an abstract class. Concrete subclasses, such as `SetOrderMessageTemplateFactory` and `ShipOrderMessageTemplateFactory`, implement this abstract class to create specific instances.

Justification:

- **Separation of Concerns:**
  - The Factory Pattern separates the responsibility of creating objects from the client code. It allows clients to create objects without knowing the specifics of their instantiation.
- **Extensibility:**
  - The pattern is extensible, making it easy to add new types of message templates in the future by creating new concrete classes that implement the `MessageTemplate` interface.
- **Consistency:**
  - By using a factory, you ensure that the creation of message templates follows a consistent process, promoting maintainability and reducing code duplication.

Strategy Pattern:

The Strategy Pattern is a behavioural design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows

the client to choose the appropriate algorithm at runtime.

Participating Classes:

1. **MessageTemplate (Context):**
   - An abstract class that defines the context for the algorithm. it's the `getNotification` method that serves as the interface for various notification strategies.
2. **SetOrderMessageTemplate and ShipOrderMessageTemplate (Concrete Strategies):**
   - Concrete classes that implement the `MessageTemplate` abstract by providing specific implementations for the `getNotification` method. Each class represents a different strategy for generating notifications.

Justification:

- **Flexibility and Extensibility:**
  - The Strategy Pattern allows you to define a family of algorithms, encapsulate each algorithm, and make them interchangeable. This flexibility is beneficial when you have different ways of generating notifications, and you want to switch between them easily or add new ones without modifying the client code.
- **Encapsulation:**
  - Each notification strategy is encapsulated within its own class, promoting a clean and modular design. This encapsulation makes it easier to understand, maintain, and extend the system.
- **Separation of Concerns:**
  - The Strategy Pattern separates the algorithm from the client, providing a clear distinction between the context (MessageTemplate) and the algorithms (SetOrderMessageTemplate, ShipOrderMessageTemplate). This separation simplifies the code and promotes better organization.

Singleton Pattern (Default use by Spring Boot):

The Singleton Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance.

Participating Classes:

1. **Component Classes annotated with `@Component`:**
   - These are likely your business or utility classes that are managed by the Spring boot. By using the Singleton Pattern, you ensure that the Spring boot creates and manages only one instance of each component class, promoting reusability and efficient resource utilization.
2. **Service Classes annotated with `@Service`:**
   - These classes typically represent your business services. By applying the Singleton Pattern, you guarantee that there is a single instance of each service class within the Spring Pot, ensuring consistent behaviour.
3. **Repository Classes annotated with `@Repository`:**
   - Repository classes often handle data access logic. Using the Singleton Pattern here ensures that you have a single, shared instance of each repository class. This can be beneficial for managing Database.
4. **Controller Classes annotated with `@RestController`:**
   - If you are using the `@RestController` annotation for your API controllers, applying the Singleton Pattern ensures that there is only one instance of each controller class. This can be useful for maintaining a consistent state across requests.

Justification:

- **Resource Efficiency:**
  - The Singleton Pattern helps in optimizing resource usage by ensuring that a single instance of a class is reused, reducing the overhead of creating multiple instances.
- **Consistent State:**
  - For classes involved in handling business logic, services, or data access, having a single instance ensures a consistent state and behaviour throughout the application.
- **Global Access:**
  - Singleton classes can be globally accessed within the application, facilitating ease of use and maintenance.

Requirements Exposure as Web Service API

Part 1: Exposed Postman Collection

⇒ **Check/Import PostmanCollection.json**

Part 2: API to Requirement Mapping

	Request Type	Http request	Happy Scenario Description	Sad Scenario Description	Request Body	Requriments
Register	Post	**/accounts/register**	"**Returns the created Account object".** We've ensured that your chosen username is unique, your initial balance is a positive number, and all the necessary information has been accurately entered.	**"Invalid account input"** It appears that the username you've chosen already exists, or the entered balance is in the negative.	Check Usage Scenario **[Register]**	A customer should be able to create an account and put a specific balance using that account.
Login	Get	**/accounts/login**	**"Returns the matching Account object"** We've confirmed that you entered username right and ensured the corresponding password is correct.	**"Invalid username or password"** The entered username doesn't match the one you previously chose, or the password provided is incorrect.	Check Usage Scenario **[Login]**	User Should be able to Login.
Products	Get	**/products/all**	**"Return list of Information of all Products"** get a detailed overview of our entire products.			A list of all the products currently available for purchase should be returned.
Category Items	Get	**/products/categories/get_products_count/{CategoryType}** **such as: CategoryType = BOOKS**	**"Return Number of items in the Category"** an overview of the number of items available within the chosen category.			a count of the remaining parts from each category should be available.
Place Simple order	Post	**/orders/place**	**"Return the created order"** We've linked your order to the correct username, ensuring the availability of the selected product and its corresponding ID and we return a complete overview of your order with some detailed information.	**"Invalid order input"** It appears that the entered username doesn't match yours, also double-check product IDs, and quantities to make sure everything is correct.	Check Usage Scenario **[Place Simple order]**	a customer can place a simple order, where such an order would include a single product or several products.
Place Compound order	Post	**/orders/place**	**"Return the created order"** We've linked your order to the correct username,	**"Invalid order input"** It appears that the entered username doesn't match	Check Usage Scenario **[Place Compound Order]**	A customer can make a compound order, where that order can include various orders

	Request Type	Http request	Happy Scenario Description	Sad Scenario Description	Request Body	Requriments
			ensuring the availability of the selected product and its corresponding ID and we show a complete overview of your order with some detailed information.	yours, also double-check product IDs, and quantities to make sure everything is correct.		headed to near-by locations, in addition to his own products, to lessen the shipping fees.
Order Info	Get	**/orders/get/{orderID}** **such as orderID = 1**	**"Return Order object"** get a detailed information of the products, total price, and any additional information associated with your purchase for a complete overview of your order.	**"Order not found"** We couldn't find your order. It seems like the provided order ID doesn't exist, or you might have forgotten to place an order.		You should be able to list all the details of both simple and compound orders.
Ship Order	PUT	**/orders/ship/{orderID}**	**"Returns the updated Order object."** *(Shipping property = true)* We've confirmed that your order ID exists, and your account balance is sufficient.	**"Order is already shipped" .** The order with the provided ID has already been shipped. **"Customer balance is not enough"** Your account balance is insufficient to complete shipping process. **"Order not found or is part of a compound order"** We couldn't find the order with the provided ID, or it might be part of a compound order.		After placing the order, the user can ship the order.
Cancel Order	Delete	**/orders/cancel/{orderID}**	Returns **"Order is cancelled successfully"** We've confirmed that the order was not shipped before processing the cancellation and Your Balance Has Been Refunded	**"Order is already shipped"** Your order has already been shipped. If you wish to cancel, you can utilize the 'Cancel Shipping' option. **"Order not found or is part of a compound order"** We couldn't find the order with the provided ID, or it might be part of a compound order.		Customers can cancel an order placement.

	Request Type	Http request	Happy Scenario Description	Sad Scenario Description	Request Body	Requriments
Cancel Shipping	Put	**/orders/cancel_shipping/{orderID}**	**“Returns the updated Order object.”** *(Shipping property = false)* Your Order Has Been Successfully Cancelled: After thorough verification, we confirmed that the order had already been shipped, and the cancellation was processed within the allowable time frame.	**“Order not found or is part of a compound order”** We couldn't find the order with the provided ID, or it might be part of a compound order. **"Order is not shipped yet”** Your order has not been shipped yet. If you intended to cancel the entire order, please use the 'Cancel Order' option. **"Order shipping time is expired”** Your order couldn't be cancelled. We only have 2 minutes for cancelling shipping and that time has passed.		Customers can cancel its shipping within a pre-configured automated duration.
Notifications	Get	**/notifications/get_in_queue**	**“Returns list of all messages to be sent”** Check out your messages, starting with the subject and detailed message body. Each message includes the chosen language, whether it's SMS or email, and is associated with your username.	**Note:** Each 1 minute a notification is popped from the queue, so if you found the queue empty or not having all required notifications, that means some of them were popped. You can create another order and quickly check the queue		For created notifications, you should implement a "notifications Queue", where you insert "notifications" that ARE TO BE SENT.
Notifications Statistics	Get	**/notifications/statistics/get_most_notified_mail**	**“Returns most notified mail”** Check out the email address that receives the most messages.	**"No notified mails found”** It seems there are no emails associated with notifications.		The system should provide live statistics on the most notified email-address/phone-number.
Notifications Statistics	Get	**/notifications/statistics/get_most_notified_phone**	**“Returns most notified phone”** Check out the phone number that receives the most messages.	**"No notified phone number found”** It seems there are no phone numbers associated with notifications.		The system should provide live statistics on the most notified email-address/phone-number.
Notifications Statistics	Get	**/notifications/statistics/get_most_notified_mail_or_phone**	**“Returns most used channel”** We will determine the most frequently used communication channel—	“**No notifications were sent”**		The system should provide live statistics on the most frequently used communication channel.

	Request Type	Http request	Happy Scenario Description	Sad Scenario Description	Request Body	Requriments
			whether it's SMS or email—and then display the most used email address or phone number.			
Notifications Statistics	Get	**/notifications/statistics/get_most_notified_template**	**“Returns most used template”** We will determine the most frequently used template it will be easy access of commonly chosen format for your messages.	“**No notifications were sent”**		The system should provide live statistics on the most frequently used notification template.

Github repository link

⇒ YoussefMorad1/Orders-and-Notifications-Module-Web-API