HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

School of Information and communications technology

Software Design Document

AIMS Project

Subject: ITSS Software Development

Group 13

Nguyễn Hữu Hoàng	20225972
Lê Ngọc Quang Hưng	20225975
Trần Quang Hưng	20226045
Tưởng Phi Tuấn	20226069
Trần Việt Anh	20226012

Hanoi, June 27th 2025

Table of Contents

1 Introduction

1.1 Objective

The purpose of this Software Design Document (SDD) is to provide a comprehensive description of the architecture, components, interfaces, and functionalities of the AIMS project, a desktop e-commerce application. This document is intended for software developers, project managers, and stakeholders involved in the development and maintenance of the AIMS software. It serves as a guide to understand the system's design and to ensure that all development efforts are aligned with the specified requirements and design principles.

1.2 Scope

The AIMS project (An Internet Media Store) aims to develop a desktop e-commerce web application that allows users to buy and sell physical media products such as books, CDs, LP records, and DVDs. The key functionalities of the software include product management, user account management, order processing, and purchasing media products. It also integrates with the VN Pay gateway for online payment.

1.2.1 Software Product(s) to be Produced

- **AIMS Web Application**: Front-end client for managing media products, user accounts, and customer orders..
- **AIMS Server Application**: Back-end handling business logic, database operations, and payment gateway (VNPay) integration.

1.2.2 Functional Capabilities

- **Product Management**: Product managers can add, edit, view, and delete products. Specific details for different media types must be provided, such as authors for books and artists for CDs.
- **User Management**: Administrators can create, update, and delete user accounts, assign roles, and reset passwords.
- **Order Processing and Purchasing**: Customers can browse products, add items to their cart, and complete purchases. The system will handle cart management, order validation, delivery information, and payment processing through VNPay. Customers can review their cart, update quantities, and receive notifications if product quantities are insufficient.

- **Notifications**: Email confirmation for order placement and status changes.

1.2.3 Application of the Software

The AIMS software is designed to operate continuously and efficiently, supporting up to 1,000 simultaneous users and providing a seamless experience for product managers, administrators, and customers. It aims to ensure high availability and performance, with a maximum response time of 2 seconds under normal conditions and 5 seconds during peak hours. Additionally, the software can resume normal operations within one hour after an incident, ensuring minimal downtime.

1.2.4 Relevant Benefits, Objectives, and Goals

- **Benefits**: Provide a user-friendly platform for managing and purchasing media products. Ensure reliable performance and quick recovery from failures.
- **Objectives**: Facilitate efficient product management, secure user account handling, and streamlined order processing and purchasing.
- **Goals**: Achieve high user satisfaction through intuitive design and robust functionality, support continuous operation with minimal downtime, and maintain compliance with security standards.

1.3 Glossary

<Listing and explaining the terms appearing in the software's profession and this document. Any assumption of the reader's prior knowledge or experience on the subject is ill advised>

No	***Term***	***Explanation***	***Example***	***Note***
1	token	A piece of data created by server, and contains the user's information, as well as a special token code that user can pass to the server with every method that supports authentication, instead of passing a username and password directly.	JSON Web Token (JWT)	Compact, URL-safe and usable especially in web browser single sign-on (SSO) context.
2	**API (Application Programming Interface)**	A set of protocols and tools for building software applications that can access the features or data of an existing system.	VNPay API	

No	*Term*	*Explanation*	*Example*	*Note*
3	**E-commerce (Electronic Commerce)**	The buying and selling of goods and services over the internet.		
4	**Payment Gateway**	A secure online service that authorizes payments made through electronic methods such as credit cards.	VNPay	
5	**Rush Order**	An order with expedited delivery, usually at an additional cost.		
6	**VAT (Value Added Tax)**	A consumption tax levied on the value added to a product or service at each stage of production and distribution.		

1.4 References

[1] Centers for Medicare & Medicaid Services, "System Design Document Template," [Online]. Available: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/SystemDesignDocument.docx.

2 Overall Description

2.1 General Overview

2.1.1 System Context and Design Approach

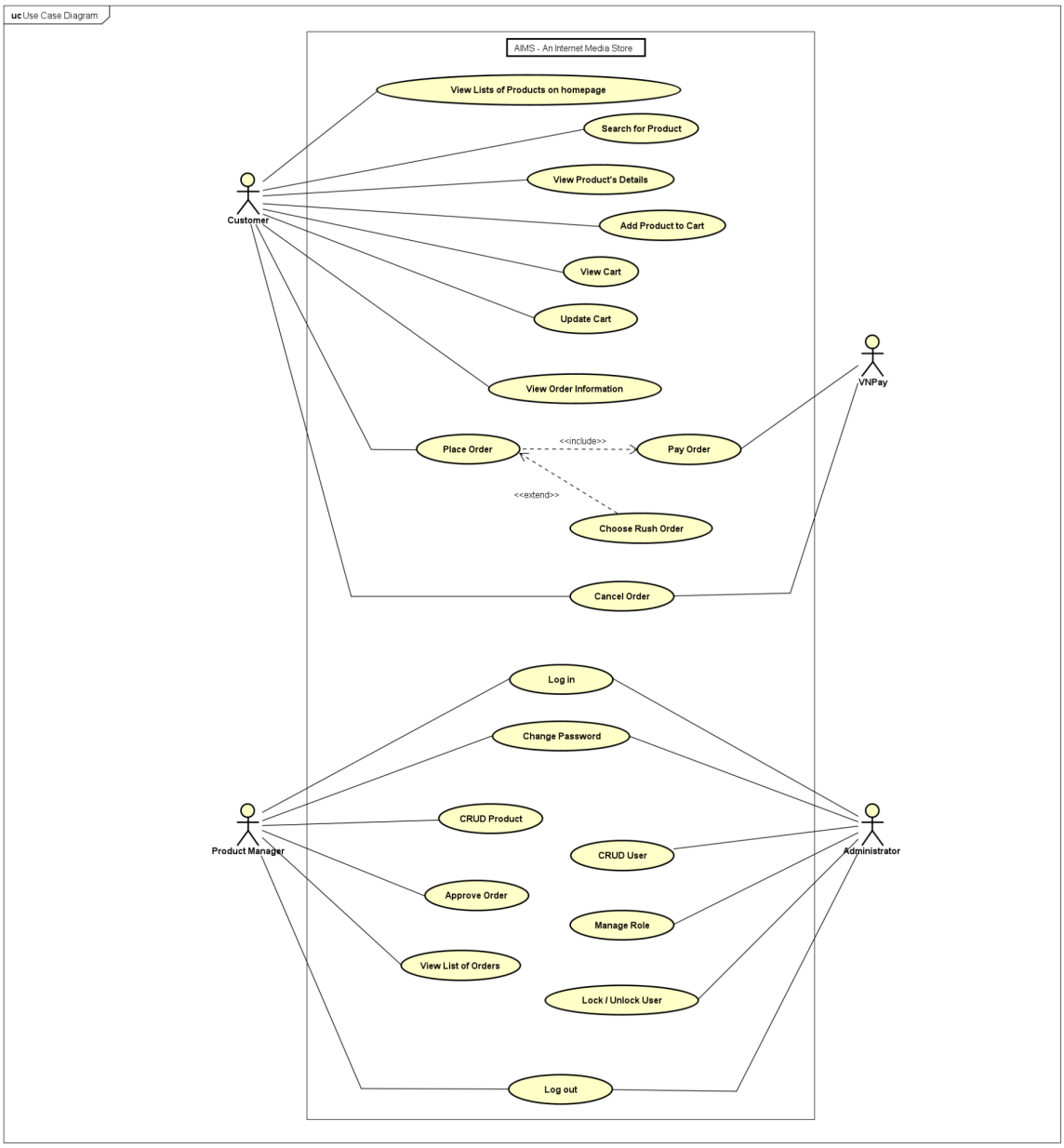
The basic design approach for AIMS is to create a robust, scalable, and user-friendly system that supports a high number of concurrent users and transactions while maintaining performance and reliability. The design goals include:

- **High Performance**: Ensuring quick response times even under peak loads.
- **Scalability**: Supporting up to 1,000 simultaneous users.
- **Reliability**: Operating continuously for 300 hours without failure and resuming normal operation within 1 hour after an incident.
- **Security**: Implementing strong security measures for data protection and user authentication.

2.1.2 System and Software Architectures

- **Client Application**: The web-based user interface developed using modern web technologies (e.g.,NestJs, React) to handle user interactions, product browsing, cart management, and checkout processes.
- **Server Application**: The server-side logic developed using SpringBoot framework to manage core business logic, database interactions, and integrations with external systems like VNPay for payment processing.
- **Database**: A relational database stores user data, product information, orders, and transaction history.
- **External Integration**: The system integrates with VNPay to handle credit card payments securely.

2.1.3 Use case Diagram



2.2 Assumptions/Constraints/Risks

2.2.1 Assumptions

- **Operating Environment**: The software will run on modern web browsers (e.g., Chrome, Firefox, Safari, Edge).
- **User Base**: Users will have basic computer literacy and internet access.
- **Related Systems**: Integration with VNPay is assumed to remain stable and reliable.
- **Functionality**: The scope includes only physical media products; no digital media or subscriptions are considered.

2.2.2 Constraints

- **Hardware Environment**: The system must function on standard web-enabled devices without requiring high-end specifications.
- **End-User Environment**: The interface must be intuitive and accessible to users with basic computer skills.
- **Resource Availability**: The system must manage resources efficiently to handle peak loads without significant performance degradation.
- **Performance Requirements**: Maintain response times under 2 seconds under normal conditions and 5 seconds during peak hours.
- **Network Communications**: Ensure reliable network communication between the front-end and back-end, with minimal latency.

2.2.3 Risks

- **System Downtime**: Risk of system downtime impacting user experience. Mitigation: Implement redundancy and failover mechanisms.
- **Security Breaches**: Potential vulnerabilities leading to data breaches. Mitigation: Regular security audits and updates.
- **Scalability Issues**: Difficulty in handling a large number of concurrent users. Mitigation: Optimize code and database queries, use load balancing.
- **Integration Failures**: Issues with VNPay integration affecting payment processing. Mitigation: Regularly test the integration and maintain close communication with VNPay support.
- **Performance Bottlenecks**: Slow response times under peak load conditions. Mitigation: Performance testing and optimization, use of caching mechanisms.
- **Data Loss**: Risk of data loss due to software bugs or hardware failures. Mitigation: Regular data backups and integrity checks.

3 System Architecture and Architecture Design

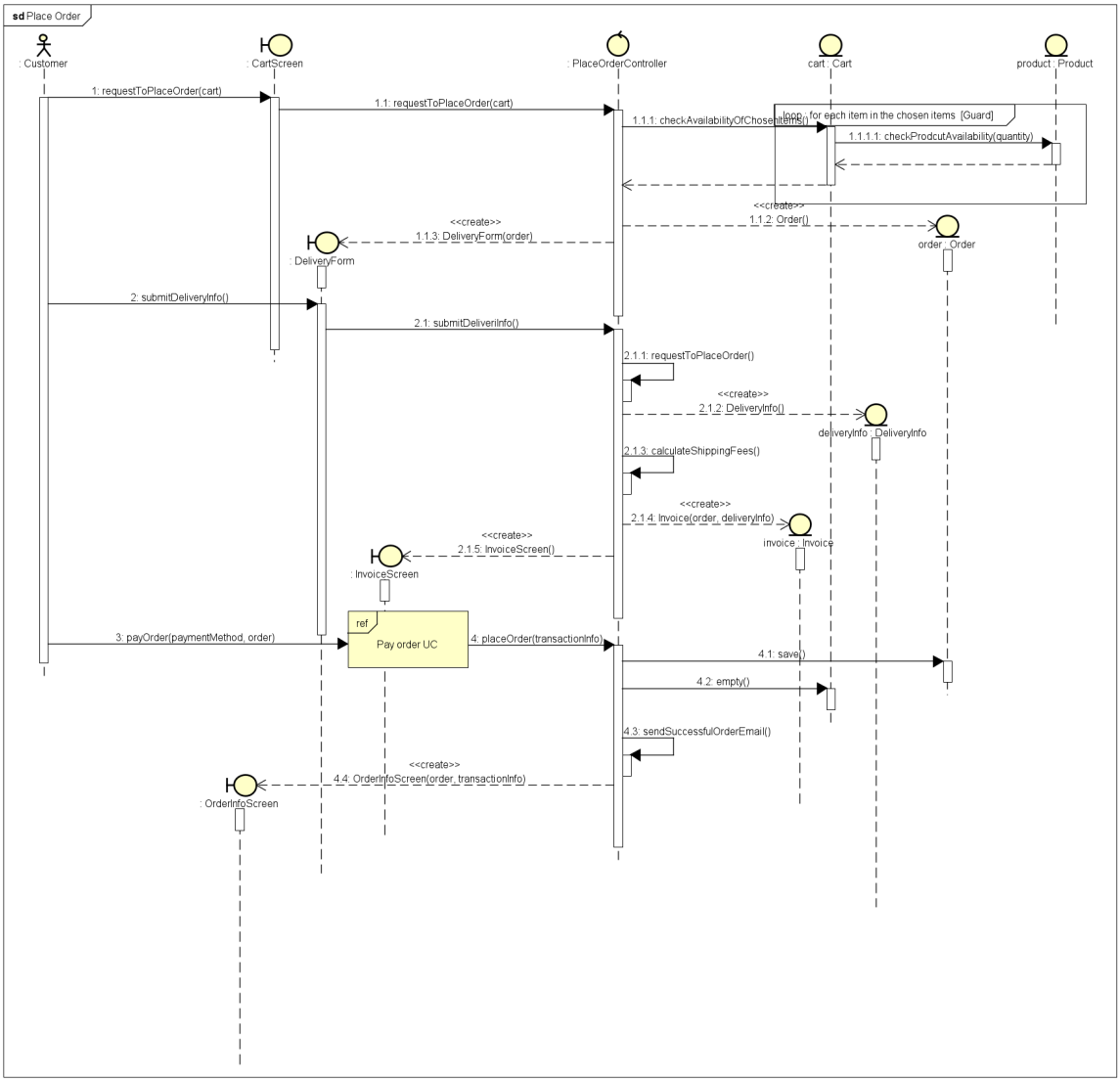
3.1 Architectural Patterns

Three-Tier Architecture:

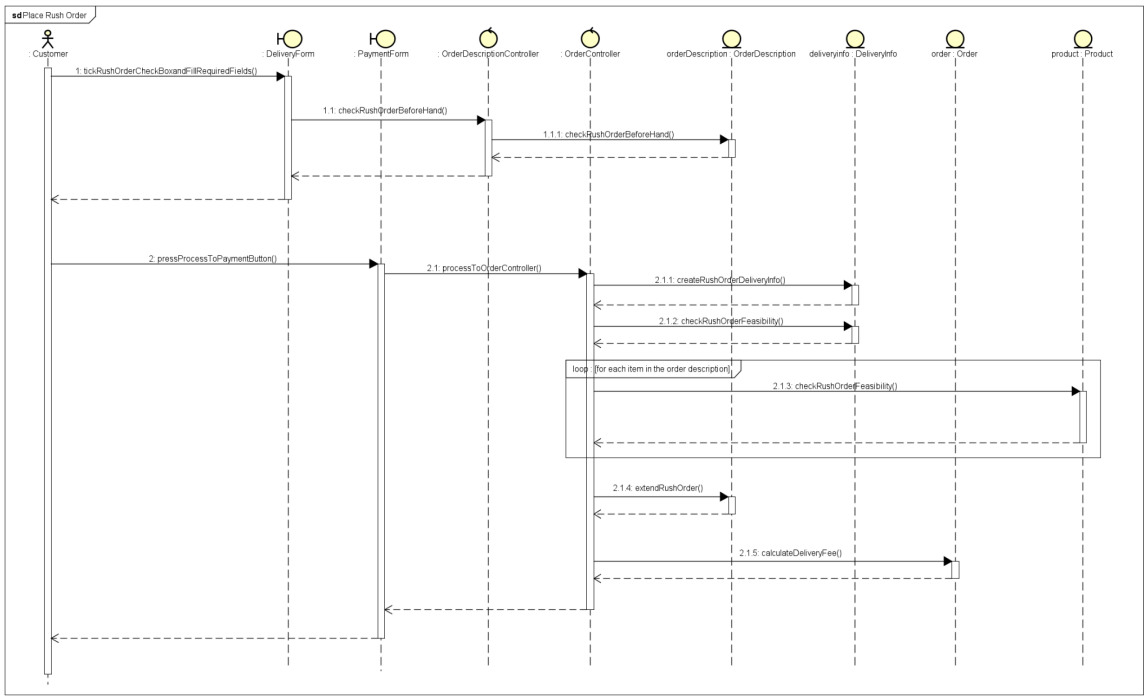
- **Description**: The Three-Tier Architecture divides the application into three logical layers: Presentation, Business Logic, and Data Access.
    - **Presentation Tier**: The frontend implemented using React, which handles the user interface and user interaction.
    - **Business Logic Tier**: The backend implemented using Spring Boot, which processes business logic and handles client requests.
    - **Data Access Tier**: The database layer that manages data storage, retrieval, and management.
- **Reasons for Choosing**:
    - **Separation of Concerns**: Each layer is responsible for a specific part of the application, making the system easier to develop, maintain, and scale.
    - **Scalability**: Each tier can be scaled independently based on the load and resource requirements.
    - **Maintainability**: Changes in one layer do not directly affect other layers, allowing for easier updates and maintenance.
    - **Flexibility**: Allows for the integration of different technologies in each layer (e.g., React for the frontend, Spring Boot for the backend, and a relational database).

3.2 Interaction Diagrams

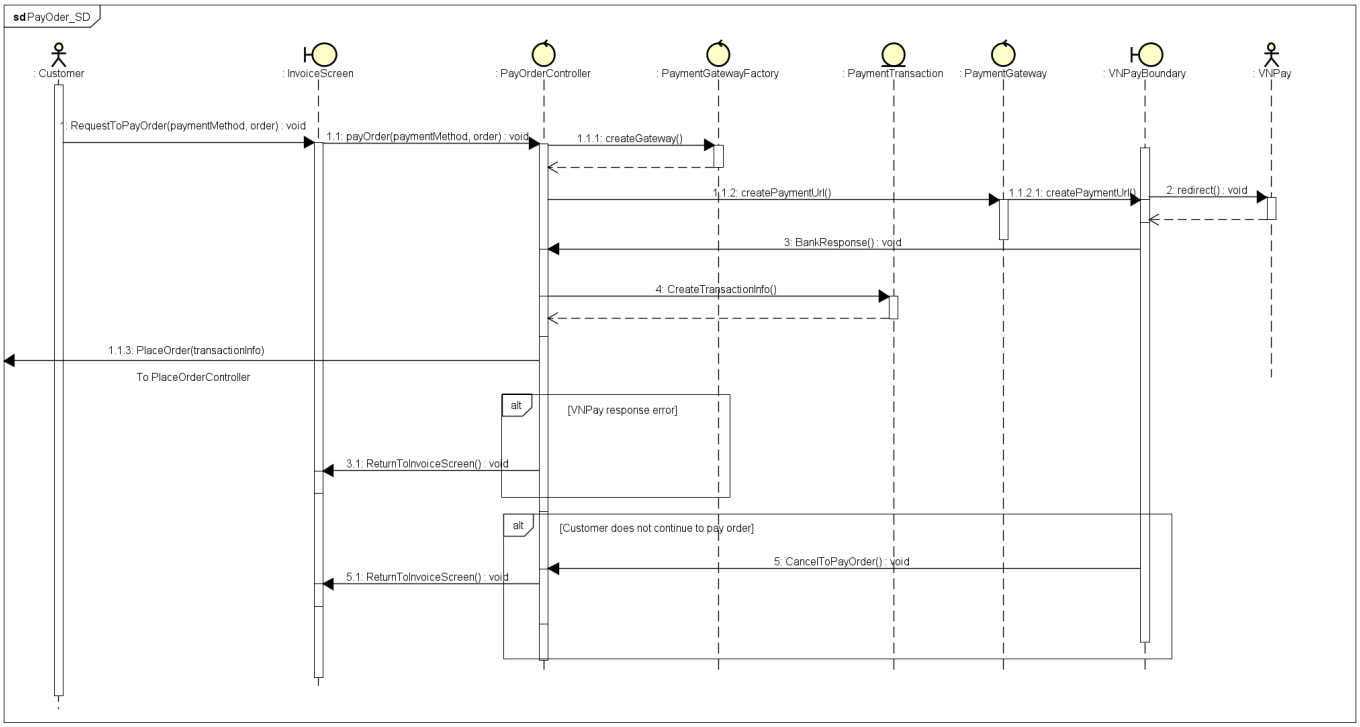
3.2.1 Sequence Diagram – Place Order



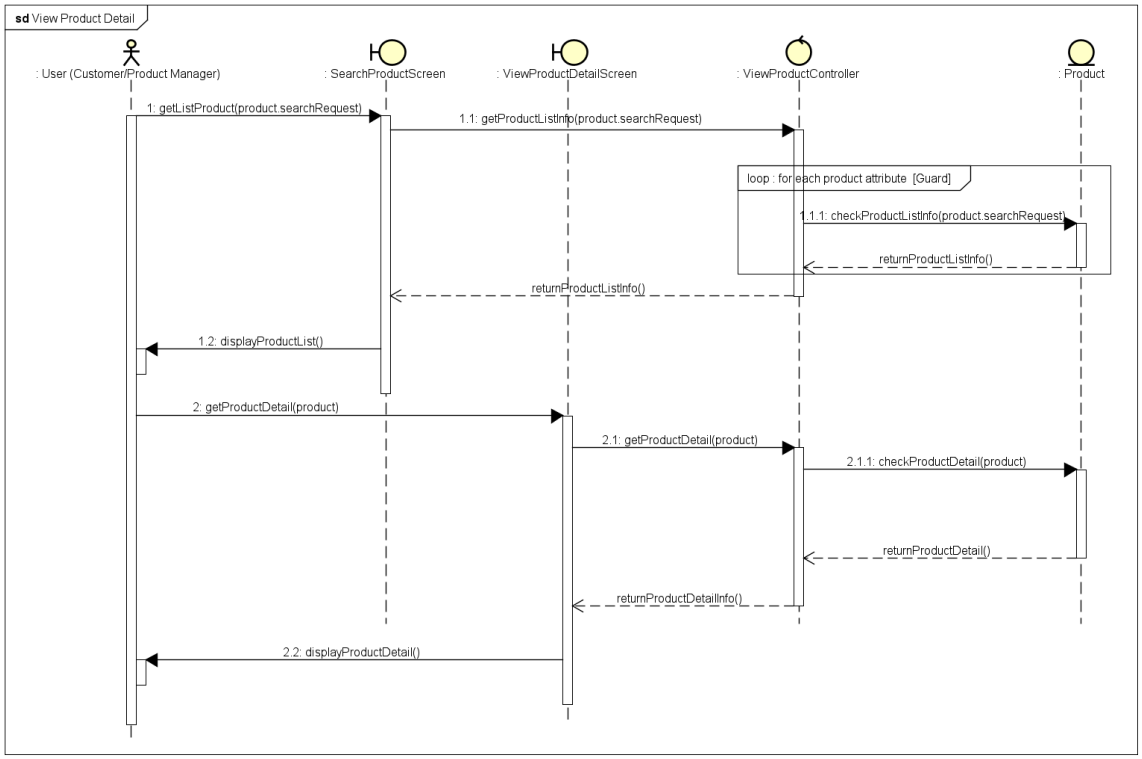
3.2.2 Sequence Diagram – Place Rush Order



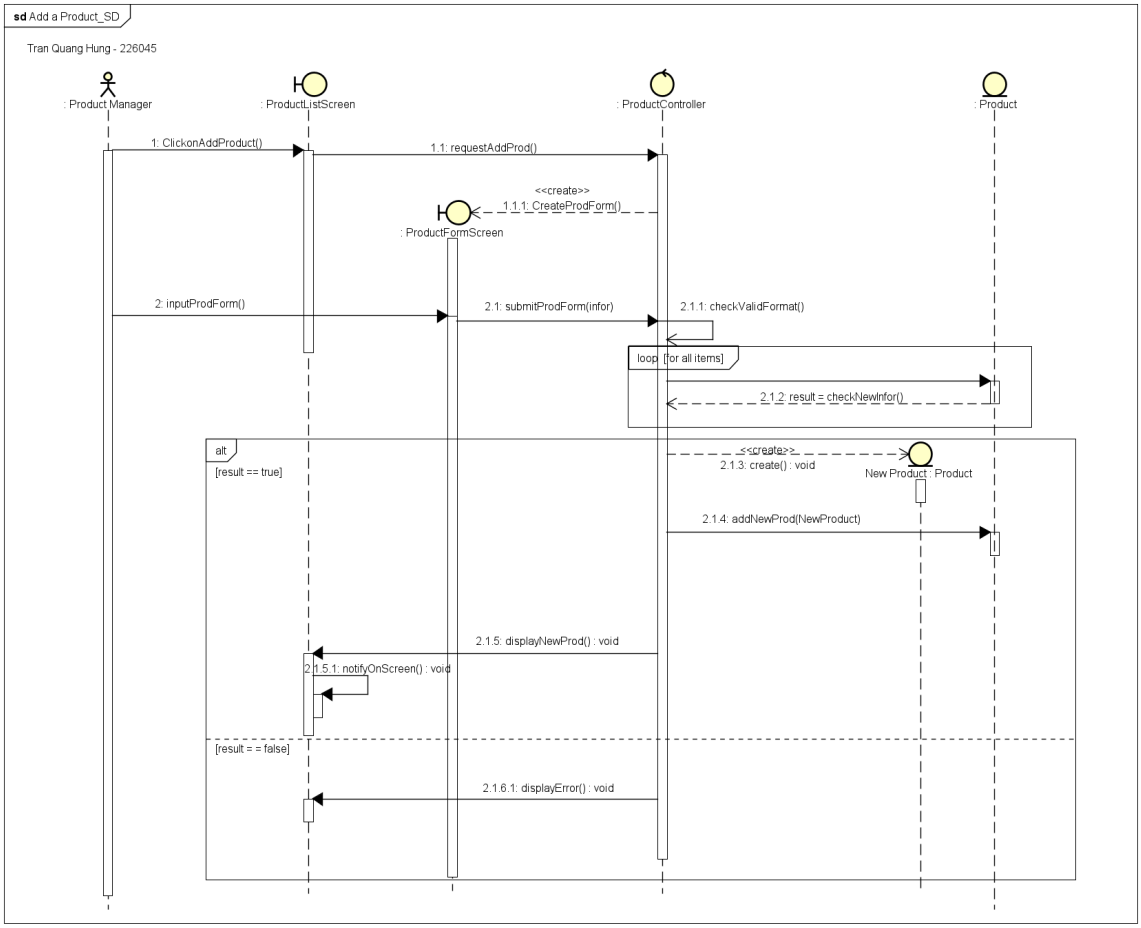
3.2.3 Sequence Diagram – Pay Order



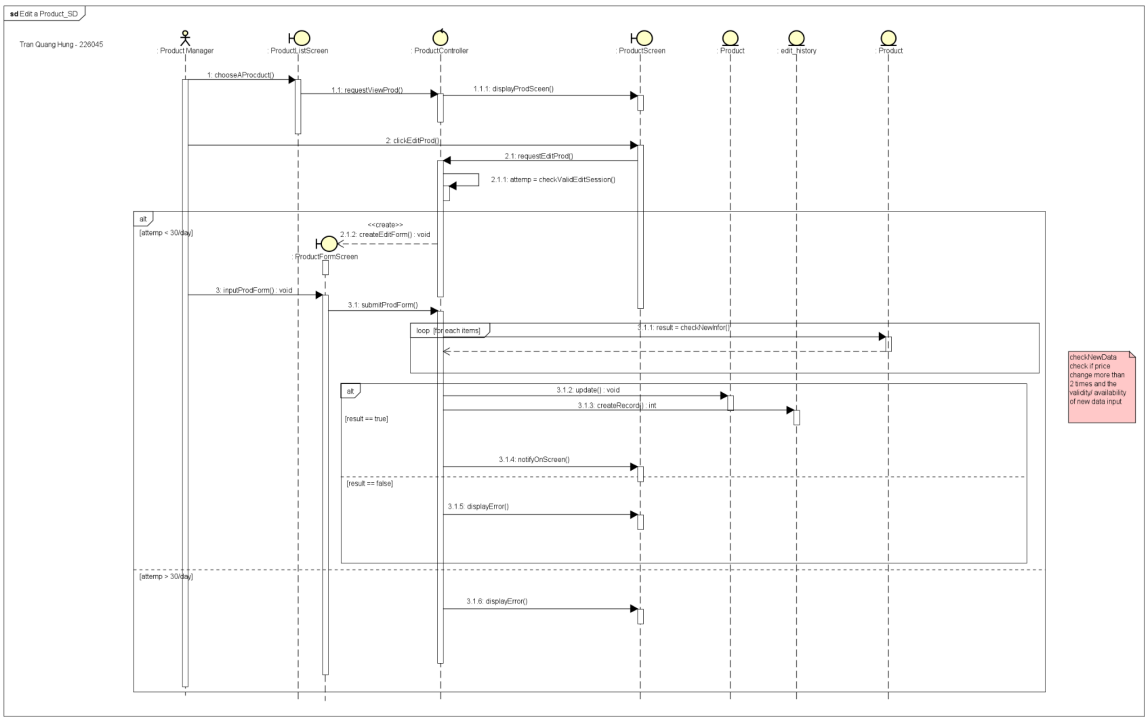
3.2.4 Sequence Diagram – View Product Detail



3.2.5 Sequence Diagram – Add/ Edit a Product



Sequence diagram for Add a Product

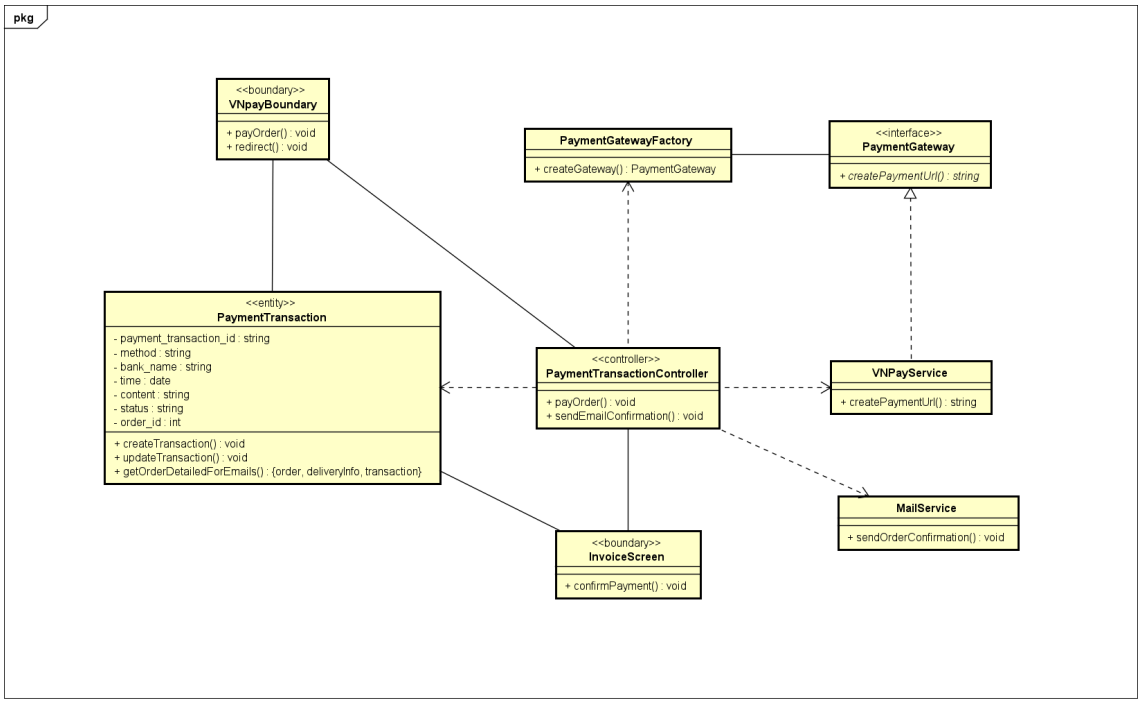


Sequence diagram for Edit a Product

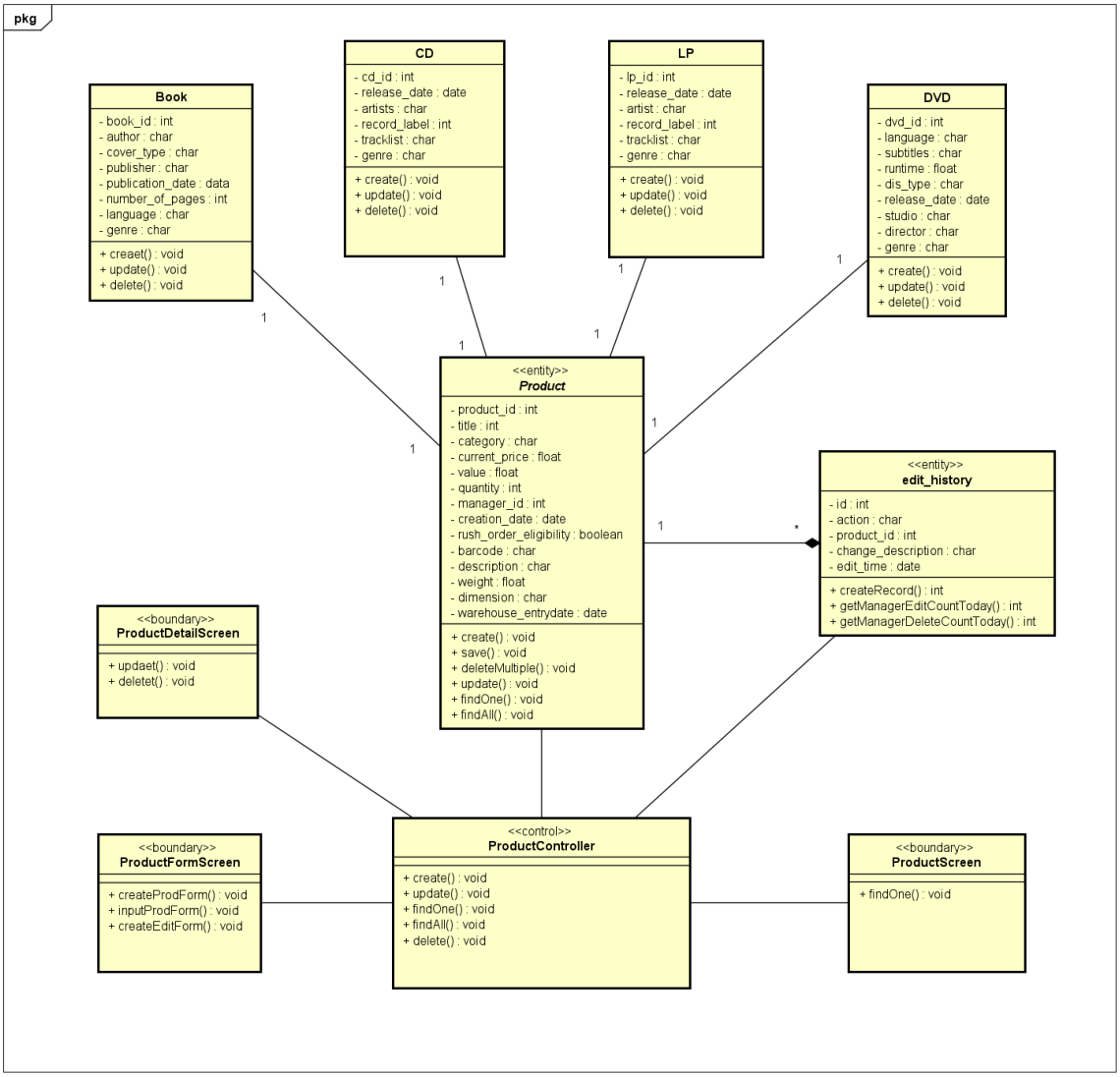
3.3 Analysis Class Diagrams

3.4 Unified Analysis Class Diagram

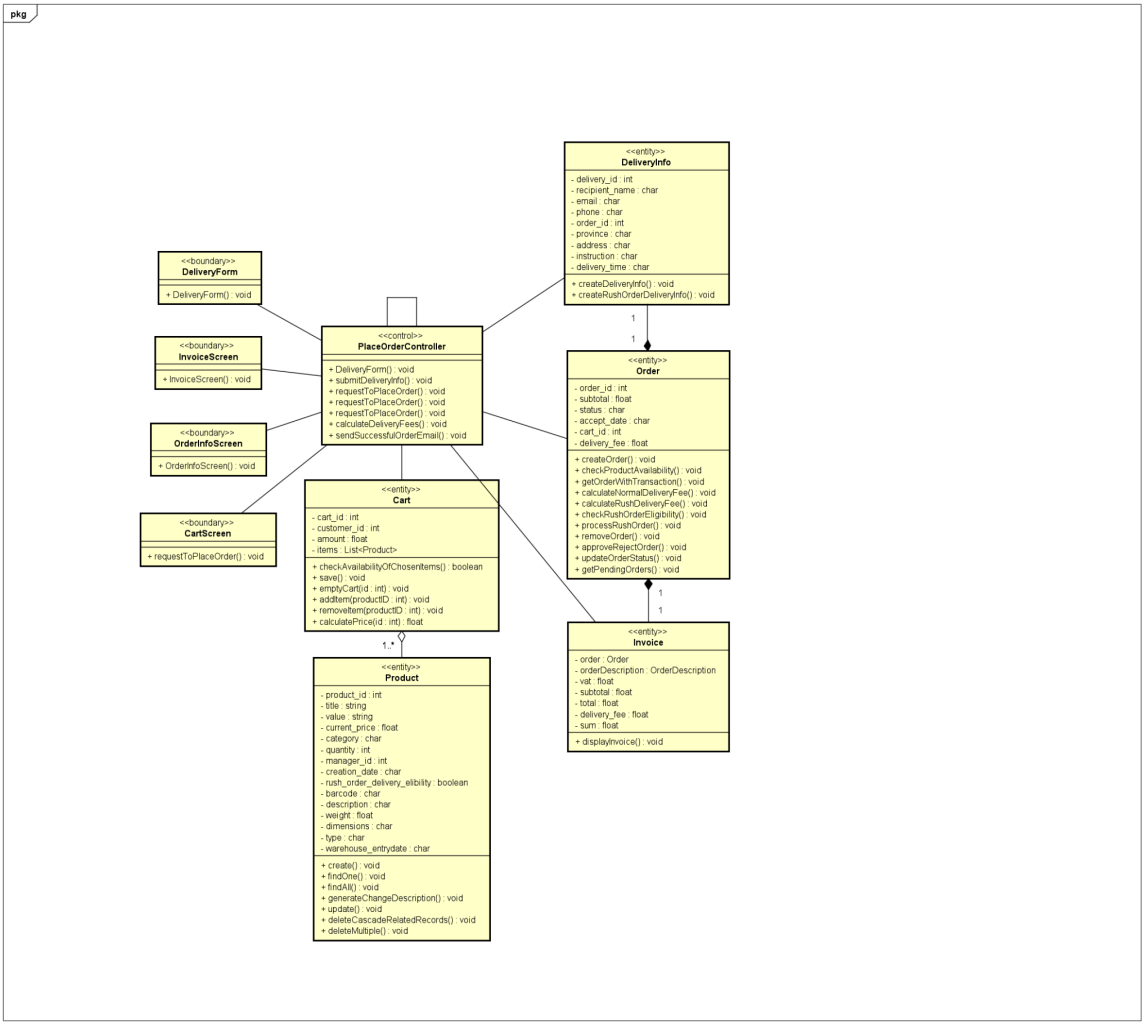
3.4.1 Class Diagram – Pay Order



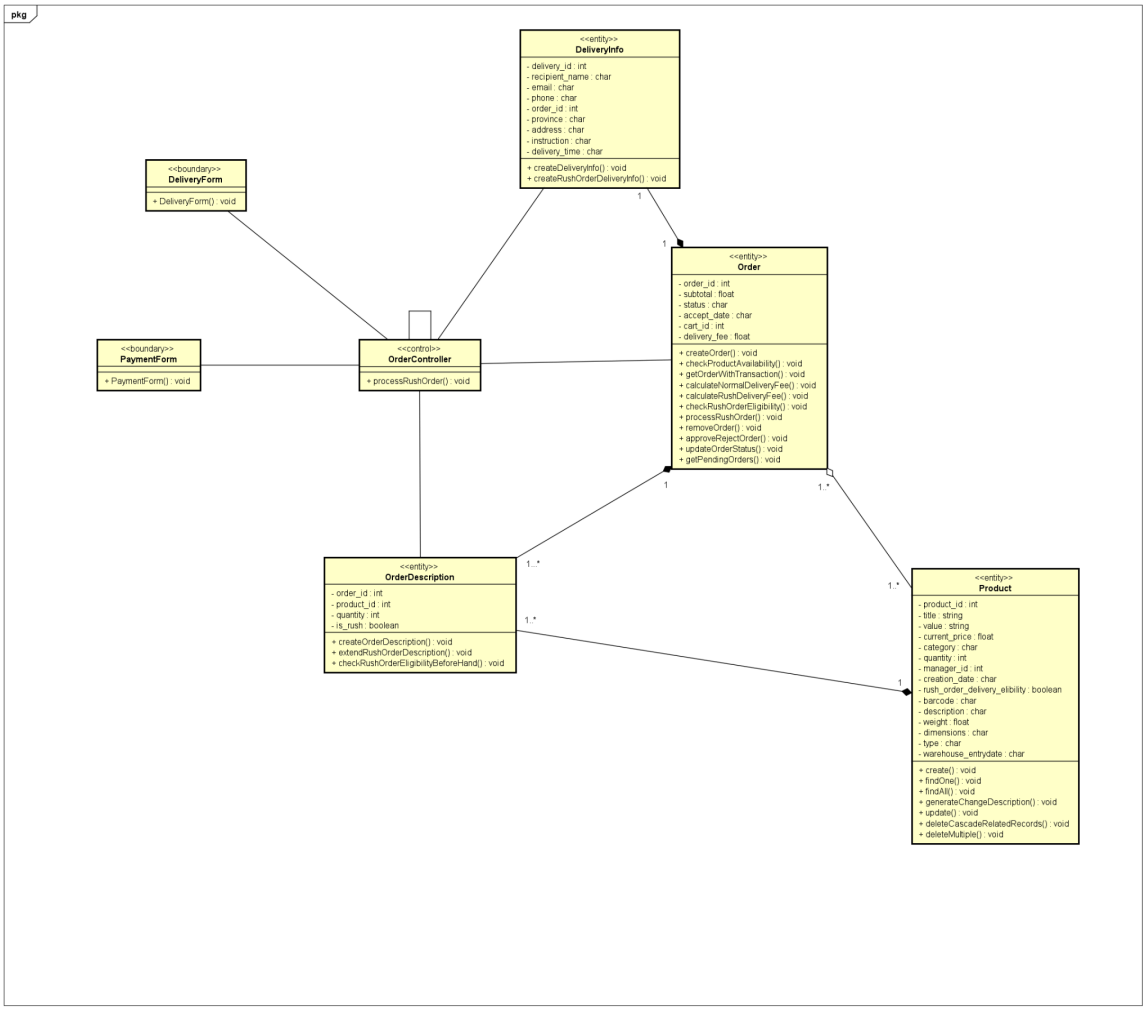
3.4.2 Class Diagram - Add/ Edit a Product and View Product Detail



3.4.3 Class Diagram - Place Order



3.4.4 Class Diagram - Place Rush Order



3.5 Security Software Architecture

The security architecture of the system encompasses several critical components and configurations to ensure the protection of data and user interactions:

1. **Authentication**:
    - **Configuration**: Supports username/password and JSON Web Tokens (JWT) authentication for stateless authentication.
    - **Purpose**: Validates user identities before granting access to system resources, ensuring only authenticated users can interact with protected endpoints.
2. **Authorization**:
    - **Component Description**: Utilizes NestJS Guards to implement robust authorization mechanisms within the application
    - **Configuration**: Defines roles (e.g., admin, user, product manager) and maps permissions to these roles within the application.

- **Purpose**: Controls user access to specific functionalities and data based on their assigned roles and permissions.

4 Detailed Design

4.1 User Interface Design

4.1.1 Screen Configuration Standardization

Layout and Navigation

1. **Header and Navbar**:
    - Contains the logo, search bar, and user menu.
    - Fixed at the top of the screen for consistent access.
2. **Main Content Area**:
    - **Max Width**: The maximum width of the content is 1560px, with some screens like product details limited to 1280px.
    - **Text Alignment**: Default text alignment is left.

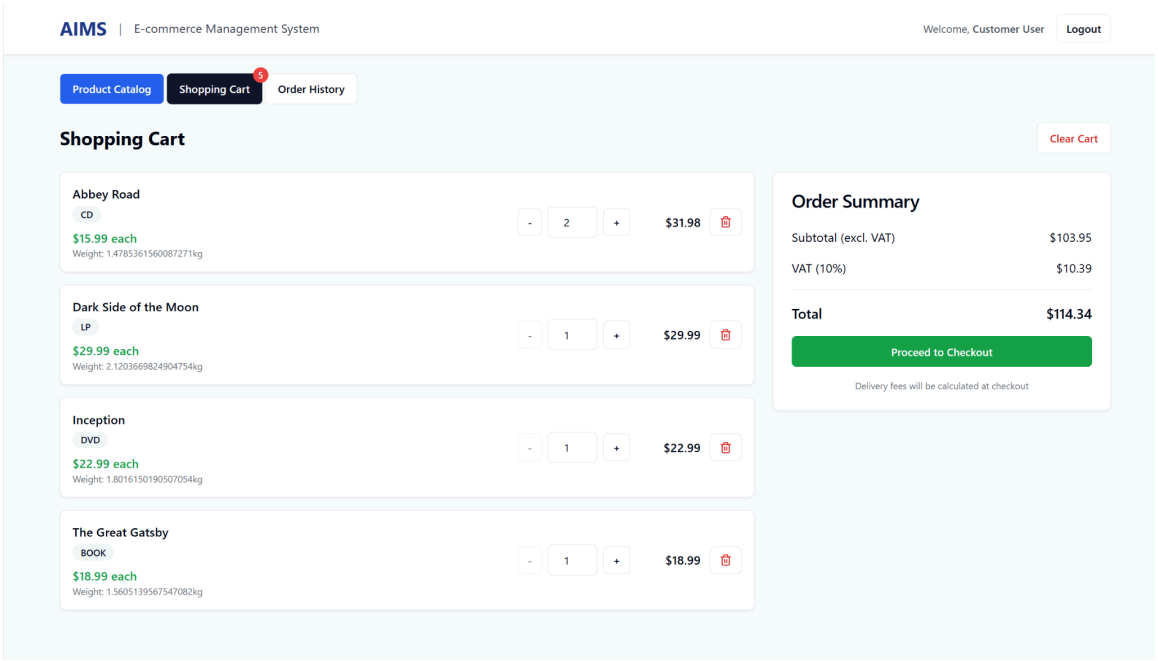
Design Standards

1. **Typography**:
    - **Font Family**: Inter
    - **Font Weights**: Normal text uses font weight 400 (normal), and headers use font weight 600 (semibold).
2. **Color Scheme**:
    - Ensure sufficient contrast ratios for readability and accessibility.
3. **Icons and Images**:
    - **Icons**: Use SVG format for consistency and scalability.
    - **Images**: Provide alt text for accessibility.
4. **Spacing and Alignment**:
    - **Padding and Margins**: Use consistent padding and margins around elements to maintain a clean layout.
    - **Grid System**: Implement a grid system to align content systematically.

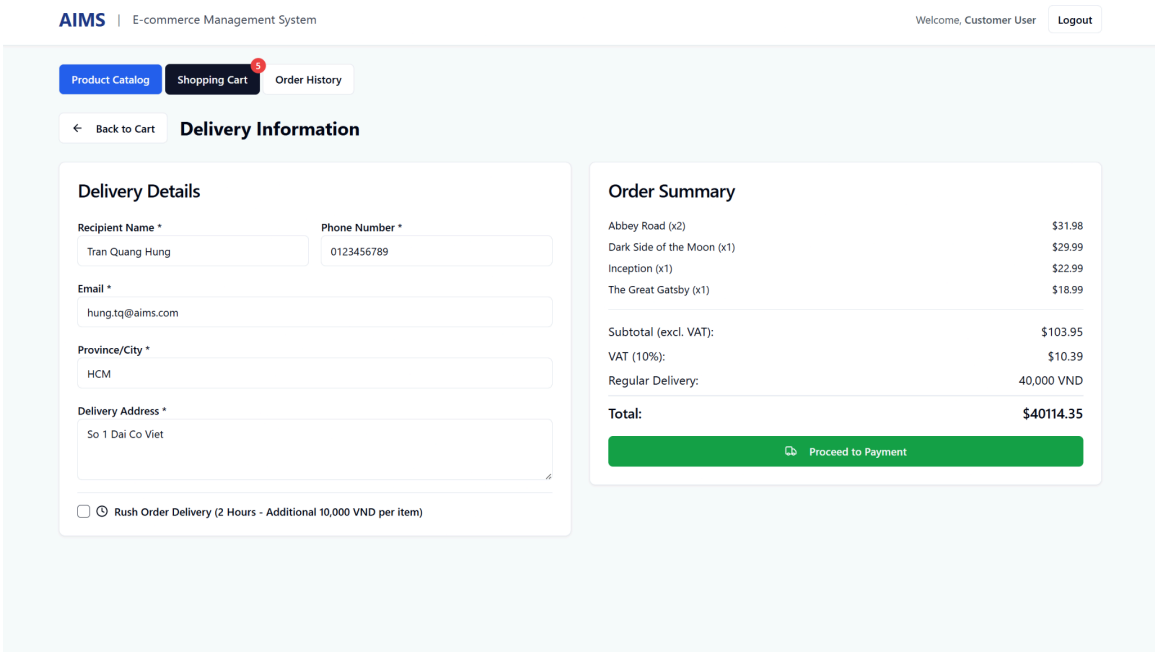
4.1.2 Screen Specifications

<Screen images should be included in the screen specifications>

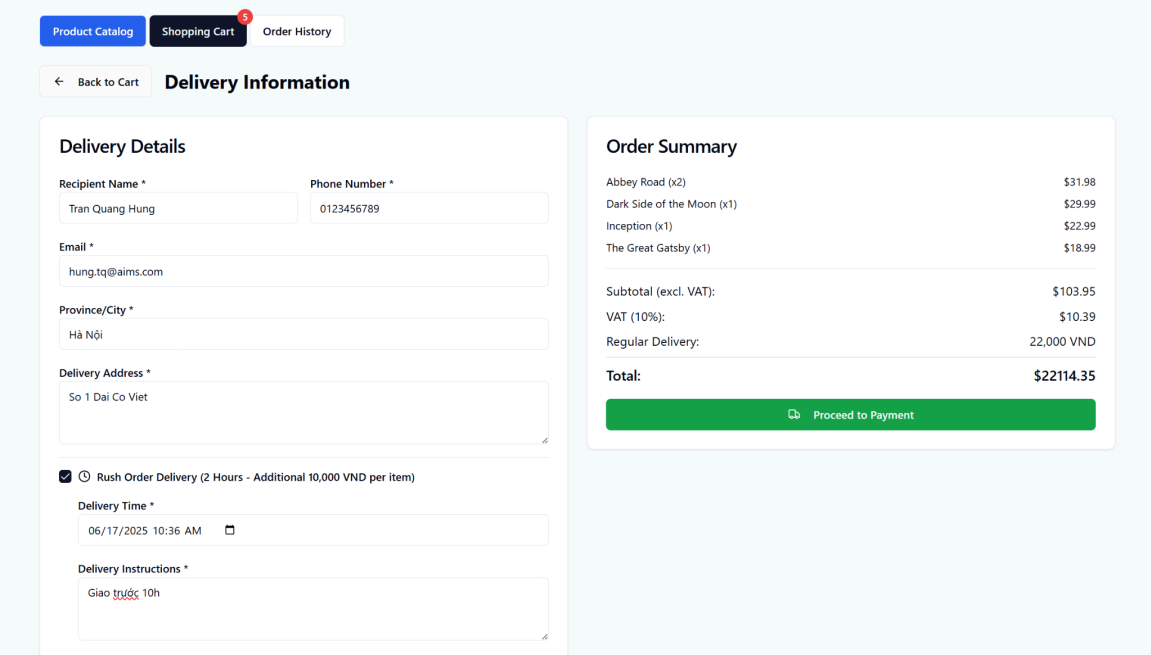
4.1.2.1 Cart Screen



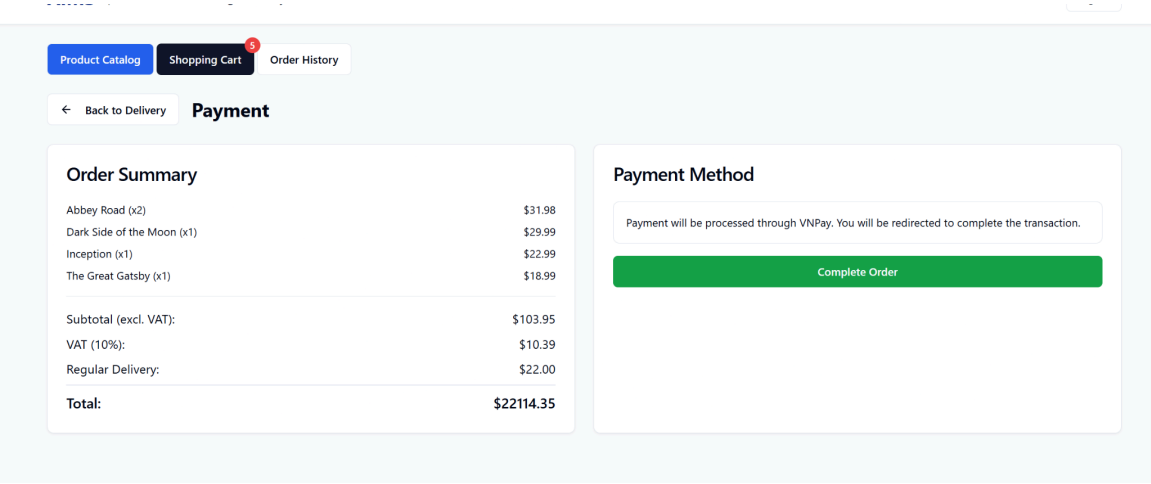
4.1.2.2 Delivery Form



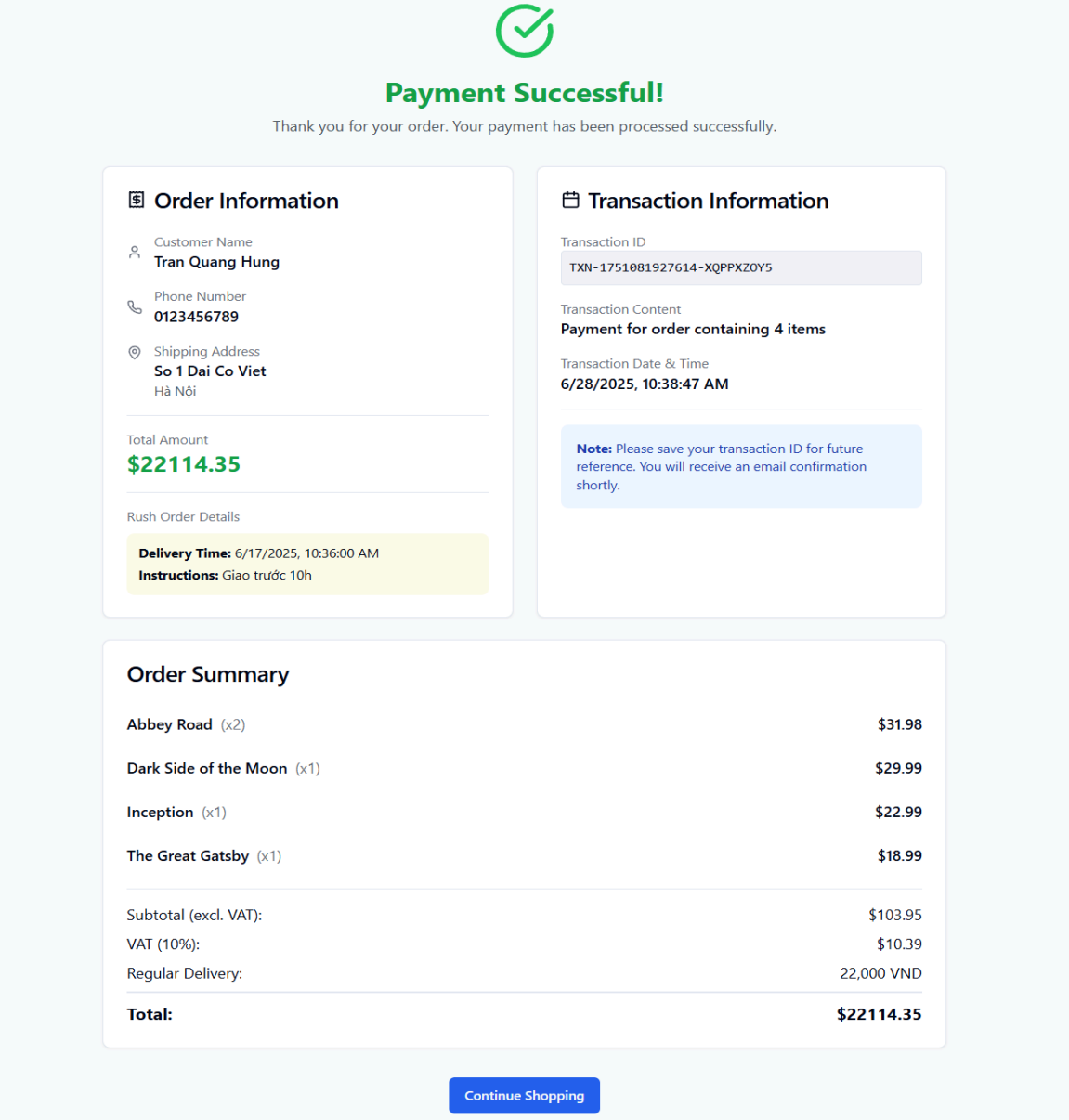
4.1.2.3 Delivery options with rush orders



4.1.2.4 Invoice



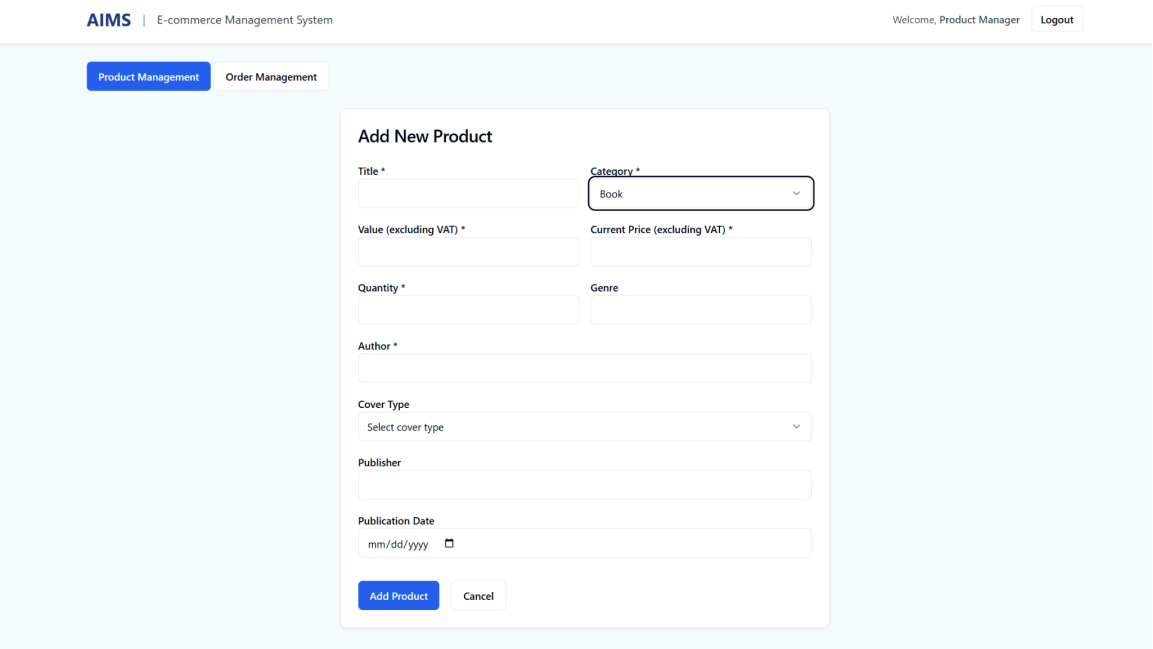
4.1.2.5 Success Payment



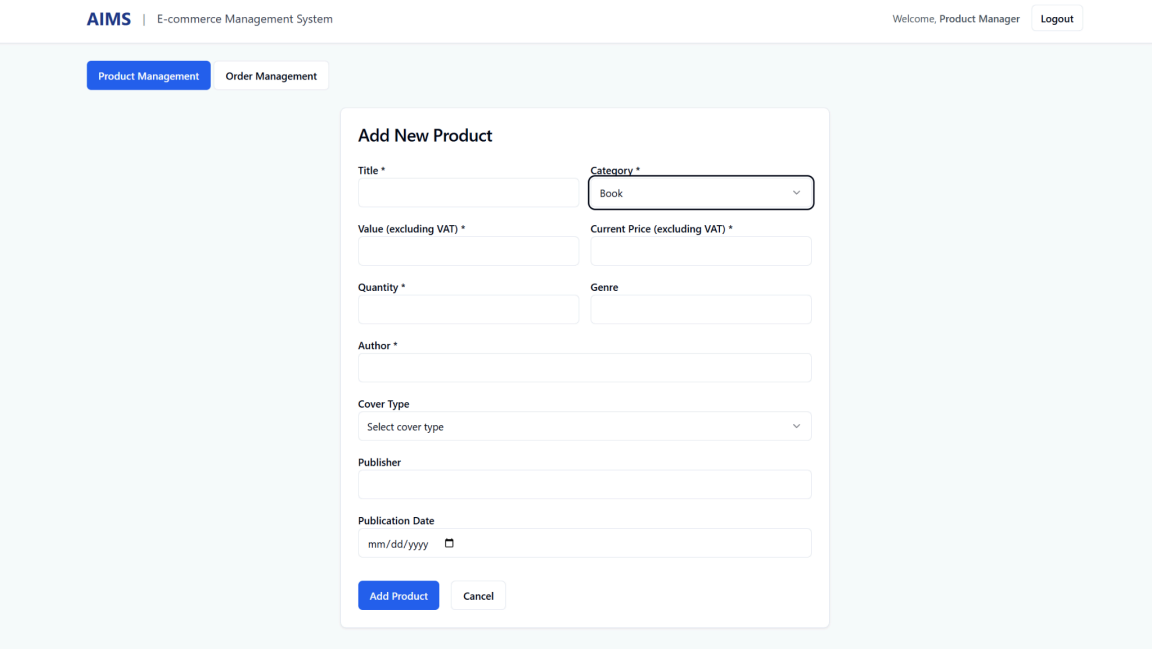
4.1.2.6 Fail Payment

4.1.4 Product Manager

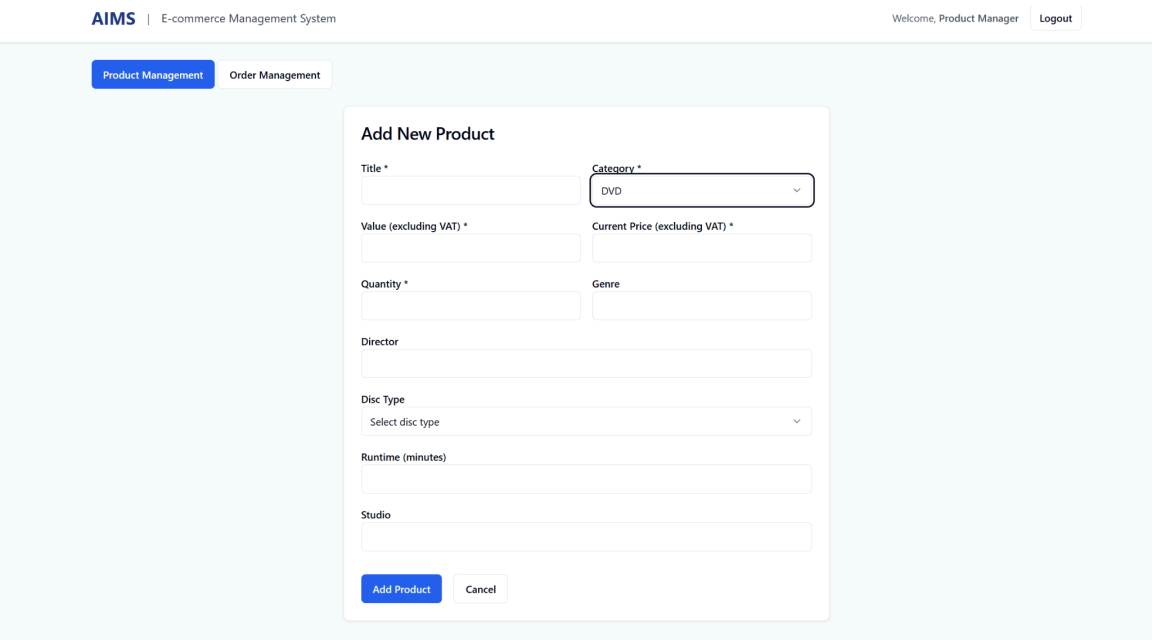
4.1.4.1 Add Book product



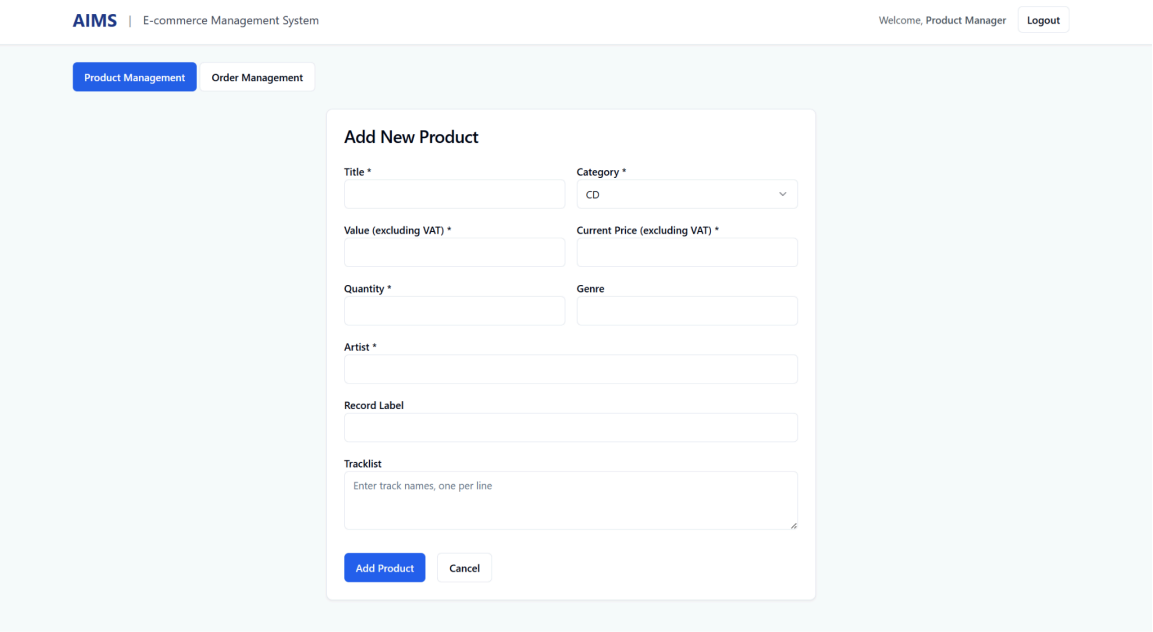
4.1.4.2 Add Lp product



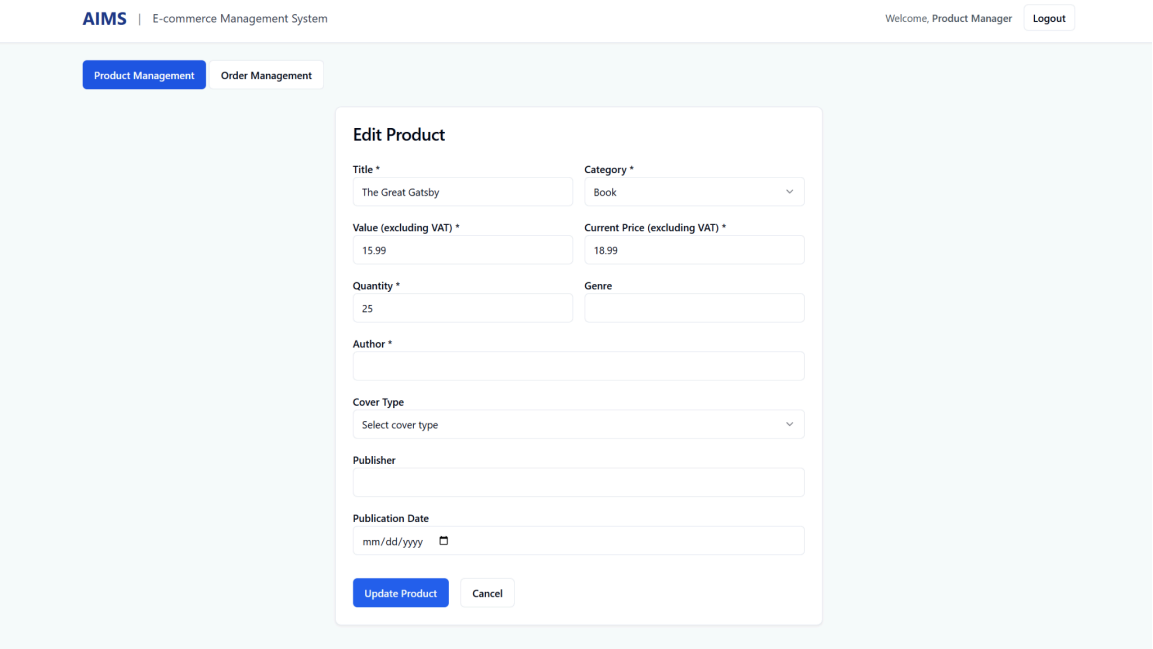
4.1.4.3 Add DvD product



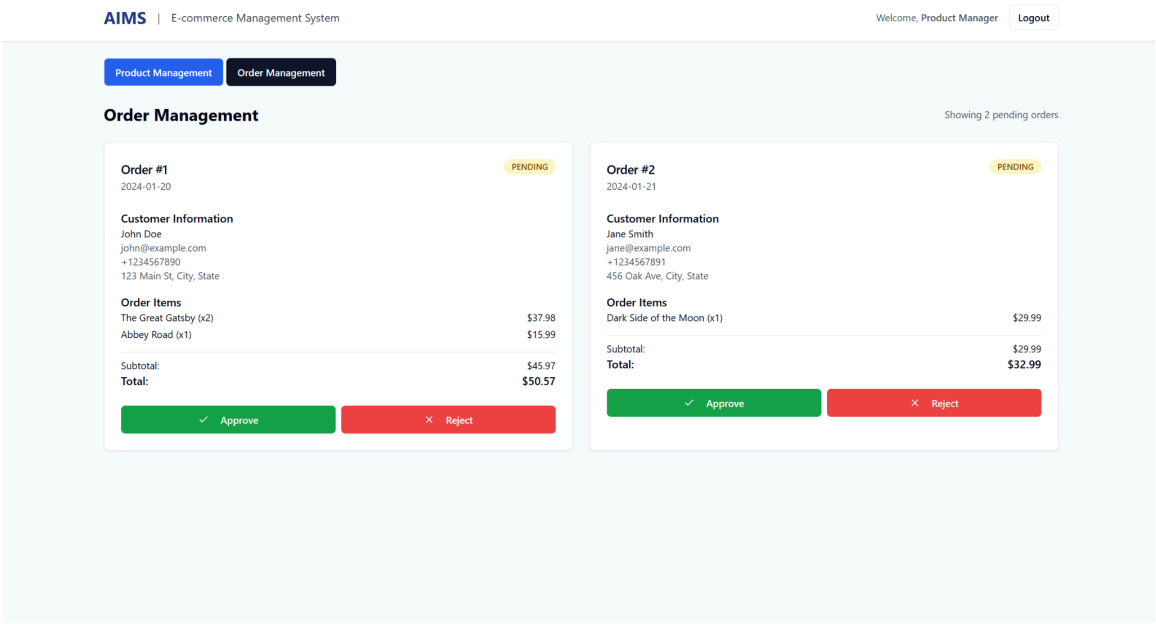
4.1.4.4 Add CD Product



4.1.4.5 Edit product

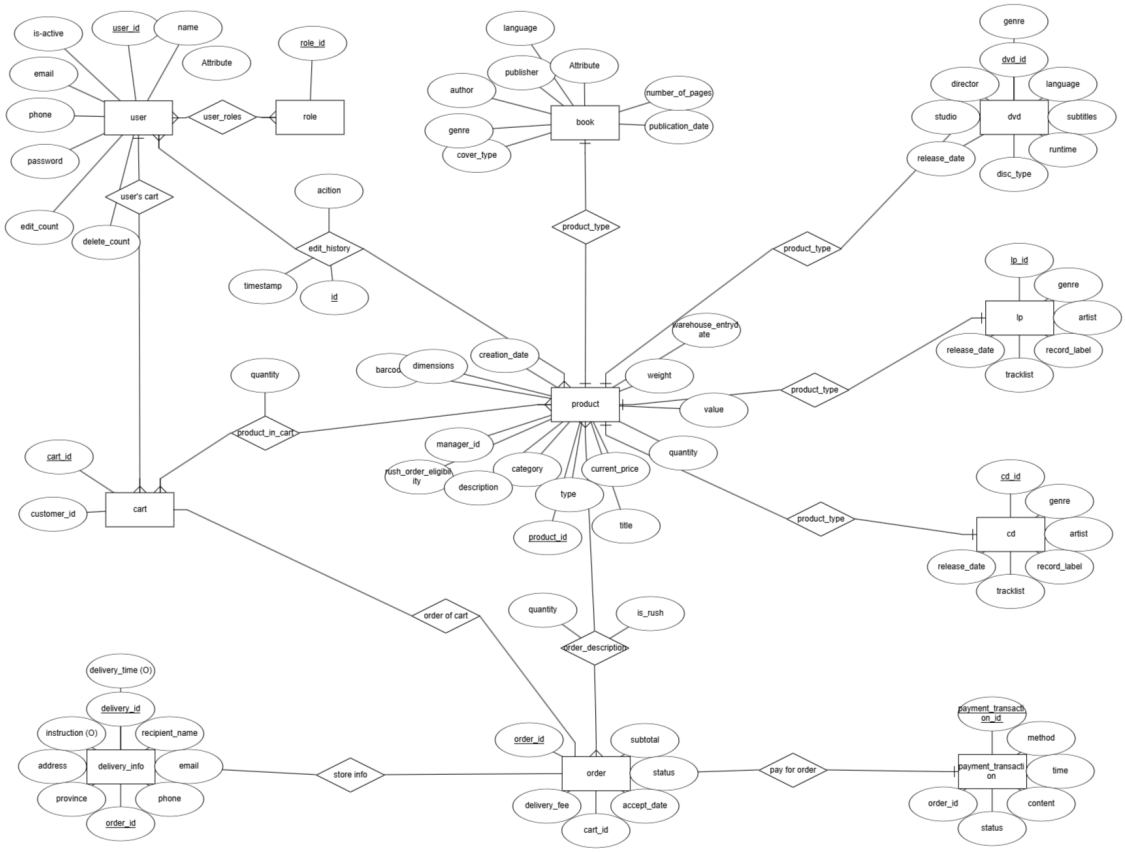


4.1.4.6 View Order


AIMS | E-commerce Management System
Welcome, Product Manager
Logout
Product Management
Order Management
Order Management
Showing 2 pending orders
Order #1
PENDING
2024-01-20
Customer Information
John Doe
john@example.com
+1234567890
123 Main St, City, State
Order Items
The Great Gatsby (x2) $37.98
Abbey Road (x1) $15.99
Subtotal: $45.97
Total: $50.57
Approve
Reject
Order #2
PENDING
2024-01-21
Customer Information
Jane Smith
jane@example.com
+1234567891
456 Oak Ave, City, State
Order Items
Dark Side of the Moon (x1) $29.99
Subtotal: $29.99
Total: $32.99
Approve
Reject

4.2 Data Modeling

4.2.1 Conceptual Data Modeling



4.2.2 Database Design

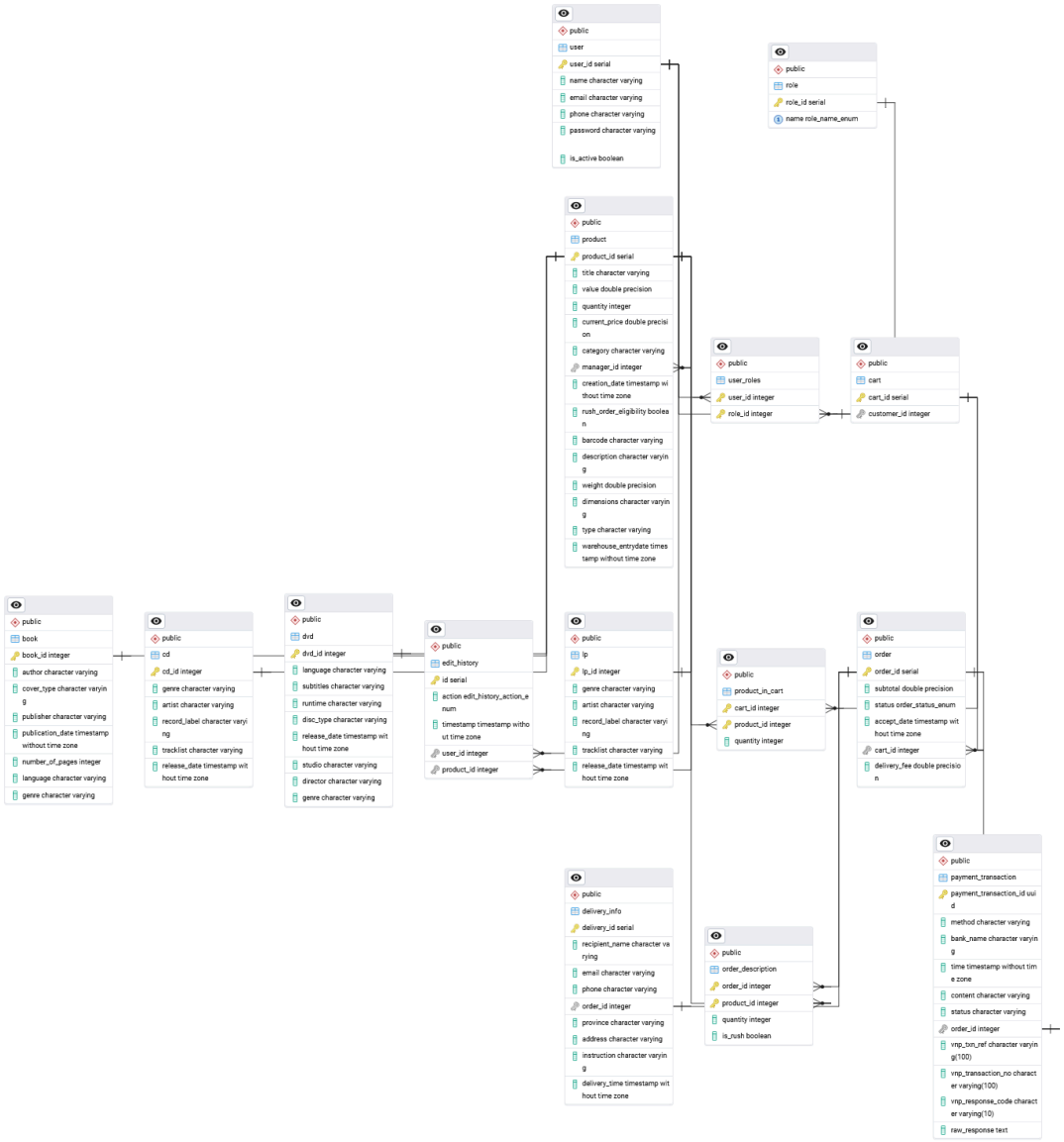
4.2.2.1 Database Management System

For the AIMS (An Internet Media Store) project, we have selected PostgreSQL as the Database Management System (DBMS).

PostgreSQL is a powerful open-source relational database management system known for its reliability, advanced features, and high performance. It supports large-scale data operations with efficient indexing, caching, and sophisticated query optimization, ensuring fast and accurate data retrieval. PostgreSQL offers robust security features such as role-based authentication, granular access controls, and data encryption, making it well-suited for managing sensitive e-commerce data. Its compatibility with multiple

platforms, support for complex data types, and an active developer community make PostgreSQL a scalable and flexible choice for the AIMS project.

4.2.2.2 Database Diagram



4.2.2.3 Database Detail Design

Table 1. Order

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		order_id	INT		Yes	Unique specifier of order

2			subtotal	FLOAT		No	Amount of money of all products in order, excluding VAT and delivery fee
3			status	ENUM	Waiting for Payment	Yes	Status of order
4			accept_date	DATE		No	Date product manager accept the order
5		x	cart_id	INT		Yes	The cart which customer choose products from to place order
6			delivery_fee	FLOAT		Yes	Delivery fee including normal and rush delivery

Table 2. Delivery_info

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		delivery_id	INT		Yes	Unique identifier of delivery_info
2			recipient_name	VARCHAR		Yes	
3			email	VARCHAR		Yes	Email that customer input to delivery_info
4			phone	VARCHAR		Yes	Phone number that customer input to delivery_info

5		x	order_id	INT		Yes	Order which this delivery_info belongs to
6			province	VARCHAR		Yes	Province that customer input to delivery_info. This helps considering products rushable or not
7			address	VARCHAR		Yes	Address that customer input to delivery_info
8			instruction	VARCHAR		No	Instruction that customer input to delivery_info
9			delivery_time	DATE		No	Delivery time that customer want to get the order

Table 3. Order_description

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x	x	order_id	INT		Yes	Identifier of order which this order_description belongs to
2	x	x	product_id	INT		Yes	Identifier of product which this order_description contains

3			quantity	INT		YES	Number of a unique product in the order
4			is_rush	BOOLEAN		YES	Indicate in case customer choose placing rush order, if this product is rushable, is_rush=true

Table 4. Products

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		product_id	INT		Yes	
2			title	VARCHAR		Yes	
3			quantity	TEXT		Yes	
4			current_price	DOUBLE		Yes	
5			category	ENUM		Yes	
6		x	manager_id	BOOLEAN		Yes	
7			creaetion_date	DATE		Yes	
8			rush_order_eligibility	BOOLEAN		Yes	
9			barcode	VARCHAR		Yes	
10			description	VARCHAR		Yes	
11			weight	VARCHAR		Yes	
12			dimension	VARCHAR		Yes	
13			type	VARCHAR		Yes	

14			warehouse_entryda te	DATE		Yes	

Table 5. CD

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		cd_id	INT		Yes	
2			genre	VARCHAR			
3			artist	VARCHAR			
4			record_label	VARCHAR			
5			tracklist	VARCHAR			
6			release_date	DATE			

Table 6. DVD

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		id	INT		Yes	
2			language	VARCHAR			
3			subtitles	VARCHAR			
4			runtime	VARCHAR			
5			disc_type	VARCHAR			
6			release_dat e	DATE			
7			studio	VARCHAR			
8			director	VARCHAR			
9			genre	VARCHAR			

Table 7. LP

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		lp_id	INT		Yes	
2			genre	VARCHAR			
3			artist	VARCHAR			
4			record_label	ENUM			
5			track_list	JSON			
6			release_date	DATE			

Table 8. Books

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		id	INT		Yes	
2			authors	VARCHAR			
3			cover-types	ENUM			
4			publisher	VARCHAR			
5			dublication_date	DATE			
6			number_of_pages	INT			
7			language	VARCHAR			
8			genre	VARCHAR			

Table 9. Edit_history

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		history_id	INT		Yes	

3		x	product_id	INT		Yes	
4			action	VARCHAR		Yes	
5			change_description	VARCHAR		Yes	

Table 10. Users

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		id	INT		Yes	
2			email	VARCHAR		Yes	
3			password	VARCHAR		Yes	
4			role	ENUM		Yes	
5			isBlocked	BOOLEAN	False	Yes	
6			edit_count	INT	0	Yes	
7			delete_count	INT	0	Yes	

Table 11. Cart

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		cart_id	INT		Yes	Unique identifier of a cart
2		x	customer_id	INT		Yes	Unique customer own this

							cart. Each customer has a unique cart

Table 12. Product_in_cart

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x	x	cart_id	INT		Yes	Cart which this product_in_cart belongs to
2	x	x	product_id	INT		Yes	Product which this product_in_cart contains
3			quantity	INT		Yes	Number of unique product in the cart

Table 13. Payment_transactions

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		payment_transaction_id	VARCHAR		Yes	
2			method	VARCHAR			
3			bank_name	VARCHAR			
4			time	TIMESTAMP			
5		x	order_id	INT		Yes	
6			content	VARCHAR			

7			status	VARCHAR			
8			order_id	INT			
9			vnp_txn_ref	VARCHAR			
10			vnp_transaction_no	VARCHAR			
11			vnp_response_code	VARCHAR			
12			raw_response	VARCHAR			

Table 14. User_role

#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1	x		user_id	VARCHAR		Yes	
2	x		role_id	VARCHAR		Yes	

Table 15. Role

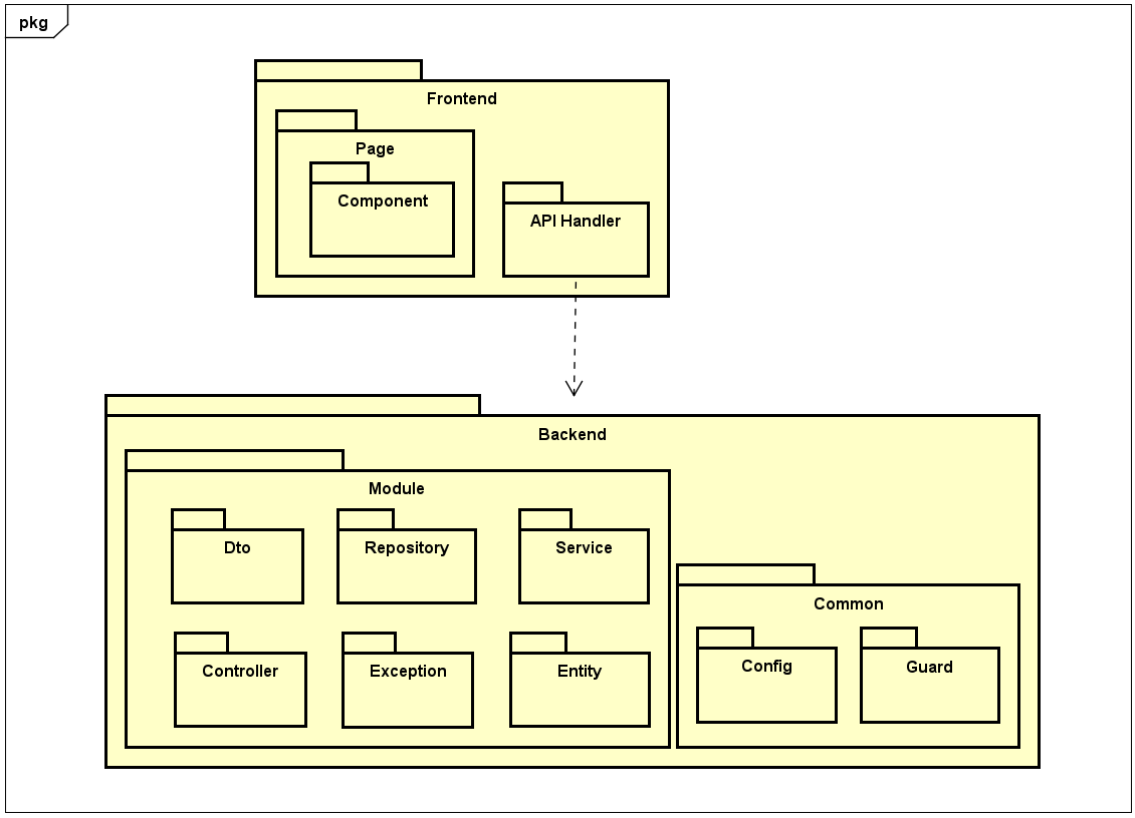
#	*PK*	*FK*	*Column name*	*Data type*	*Default value*	*Mandatory*	*Description*
1			name	VARCHAR		Yes	
2	x		role_id	VARCHAR		Yes	

4.3 Non-Database Management System Files

None

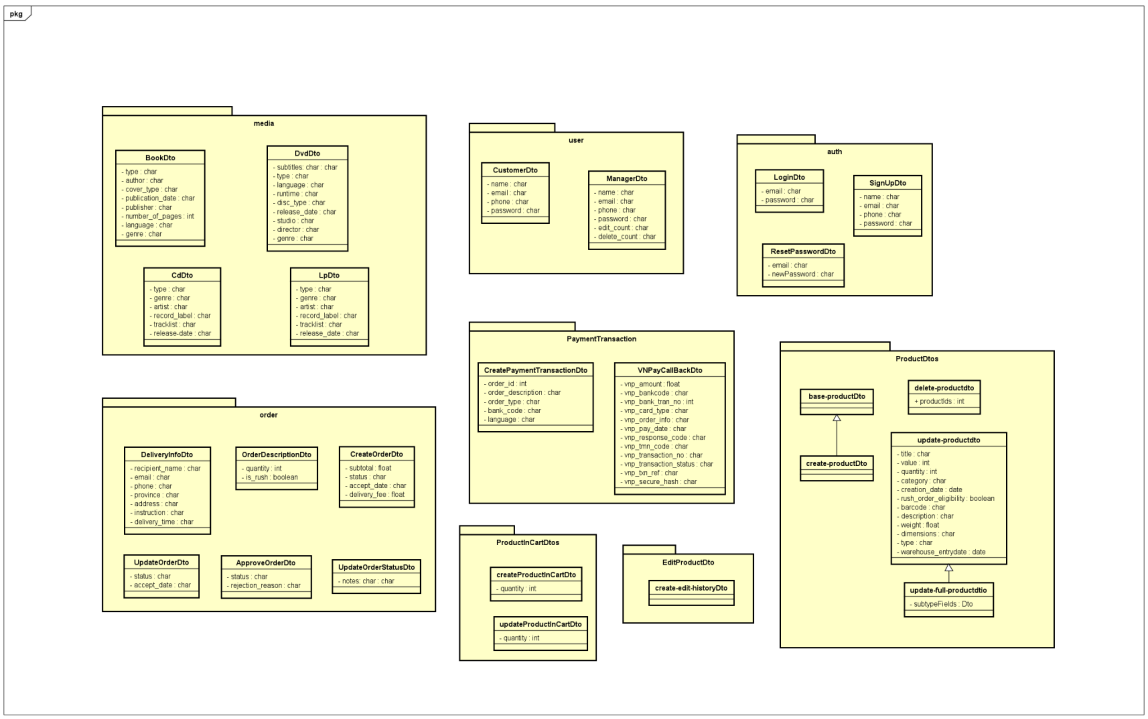
4.4 Class Design

4.4.1 General Class Diagram



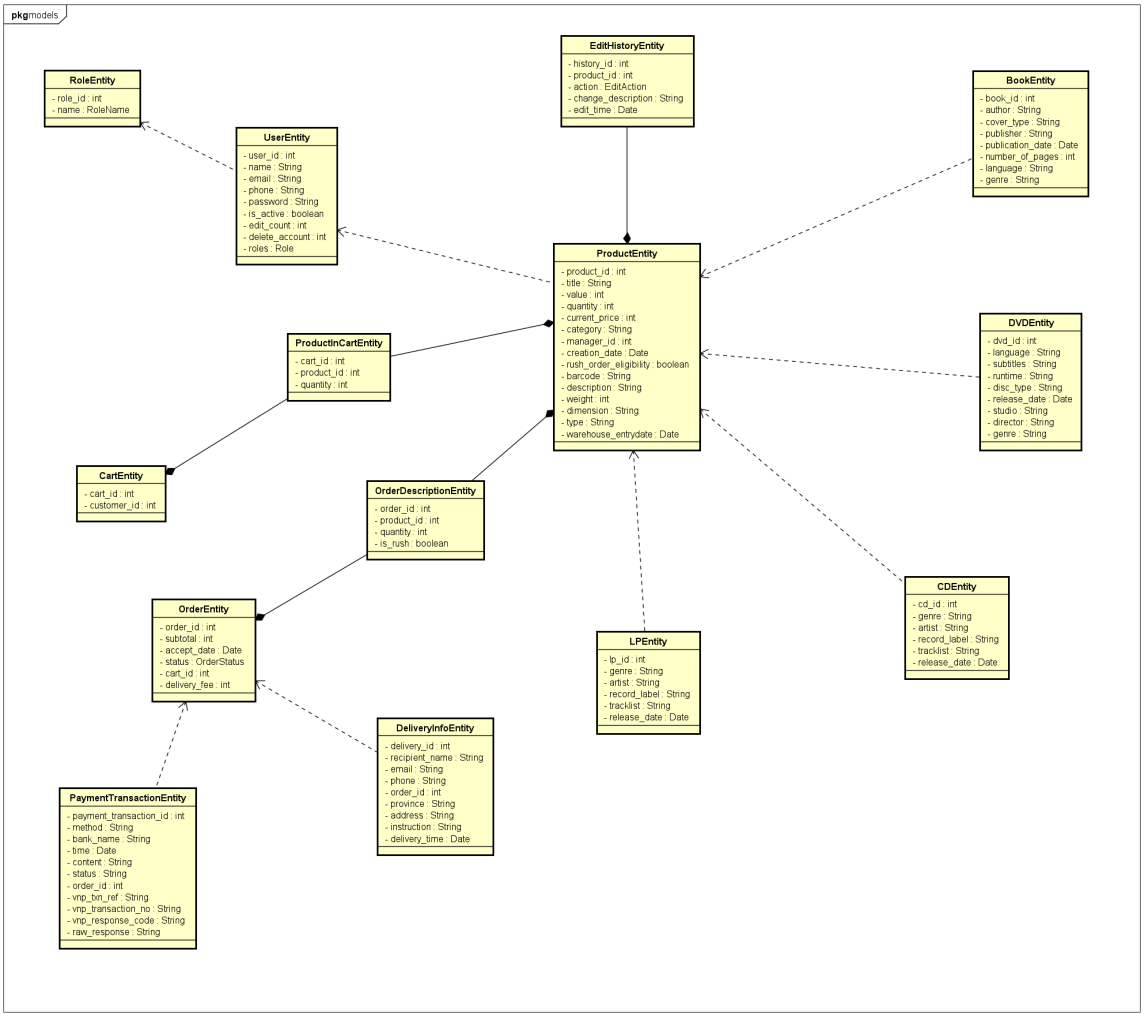
4.4.2 Class Diagrams

4.4.2.1 Class Diagram for Package Dto

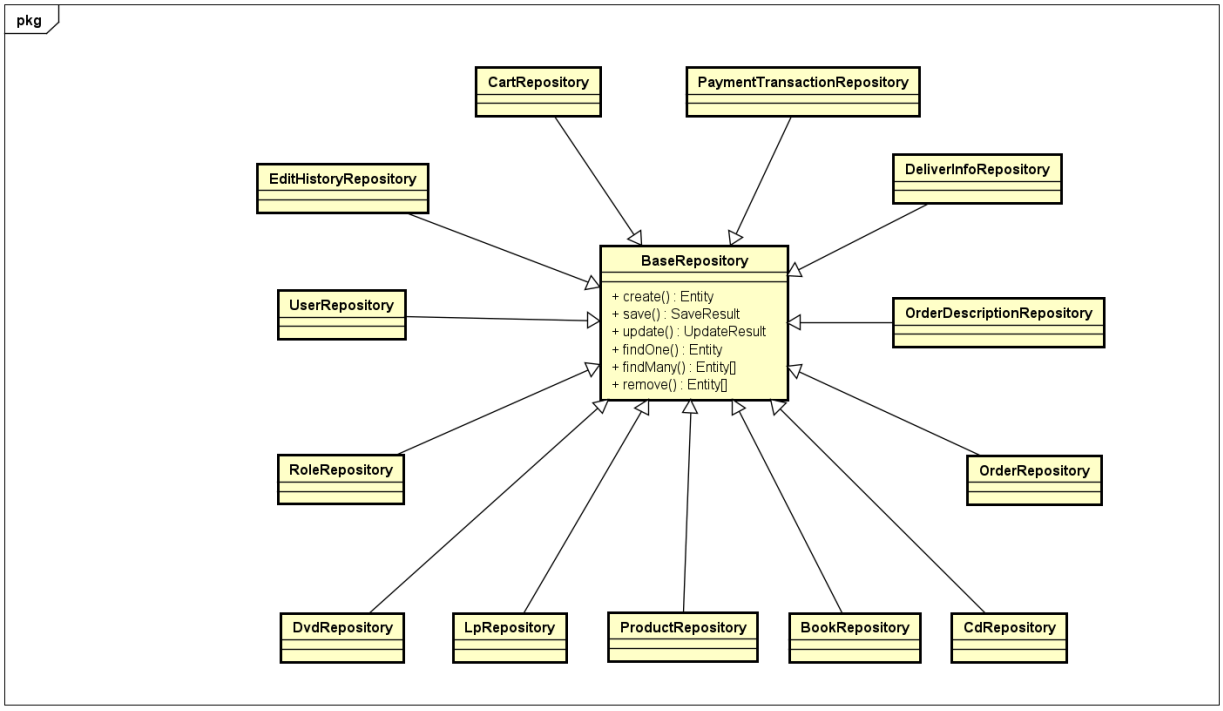


4.4.2.2 Class Diagram for Package Exceptions

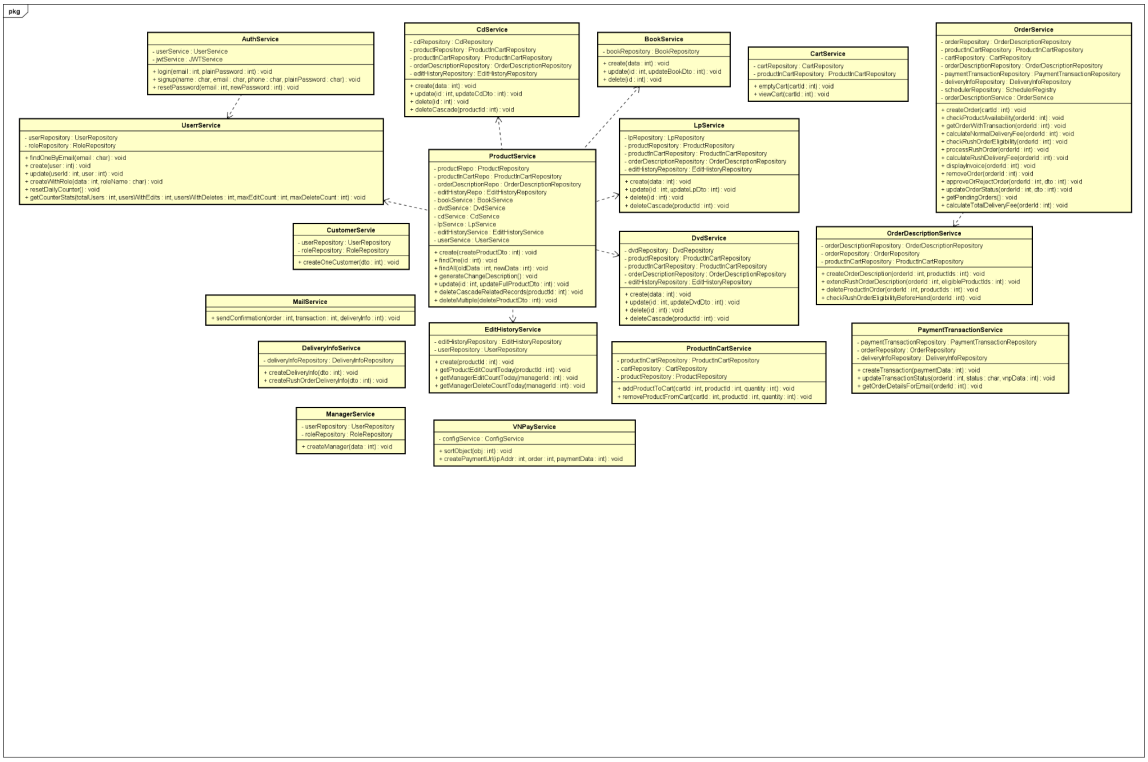
4.4.2.3 Class Diagram for Package Model



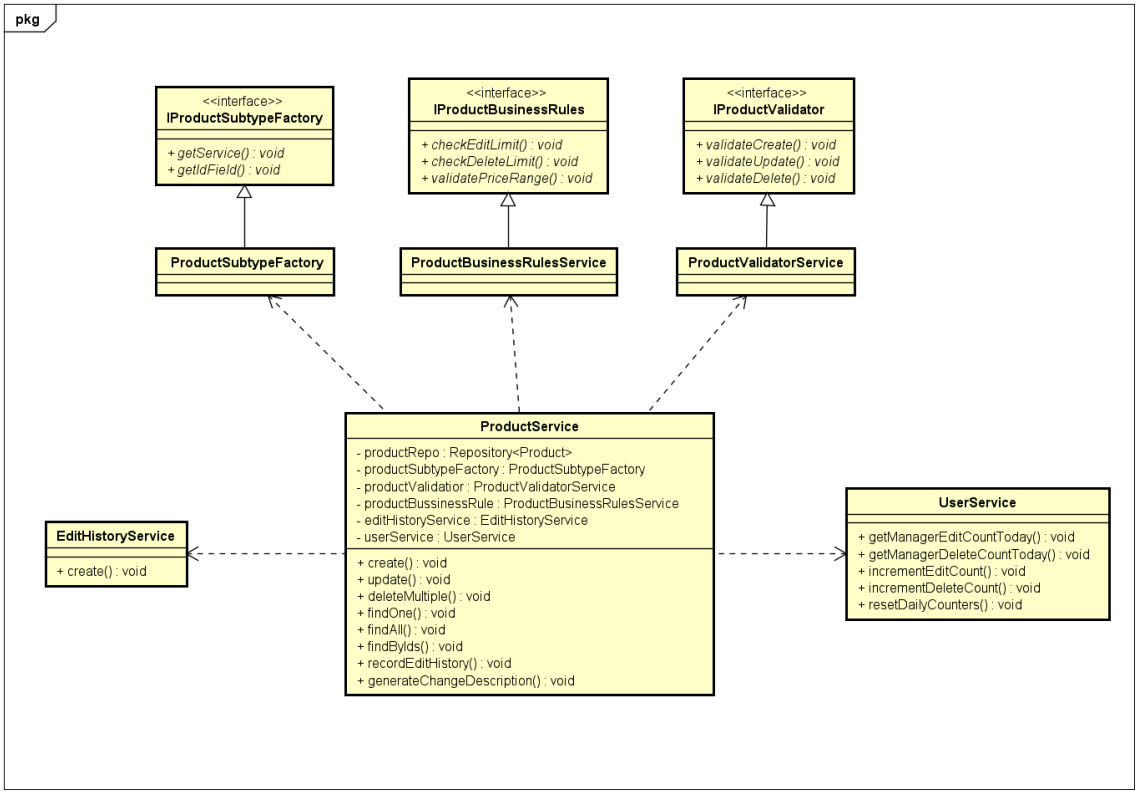
4.4.2.4 Class Diagram for Package Repositories



4.4.2.5 Class Diagram for Package Service

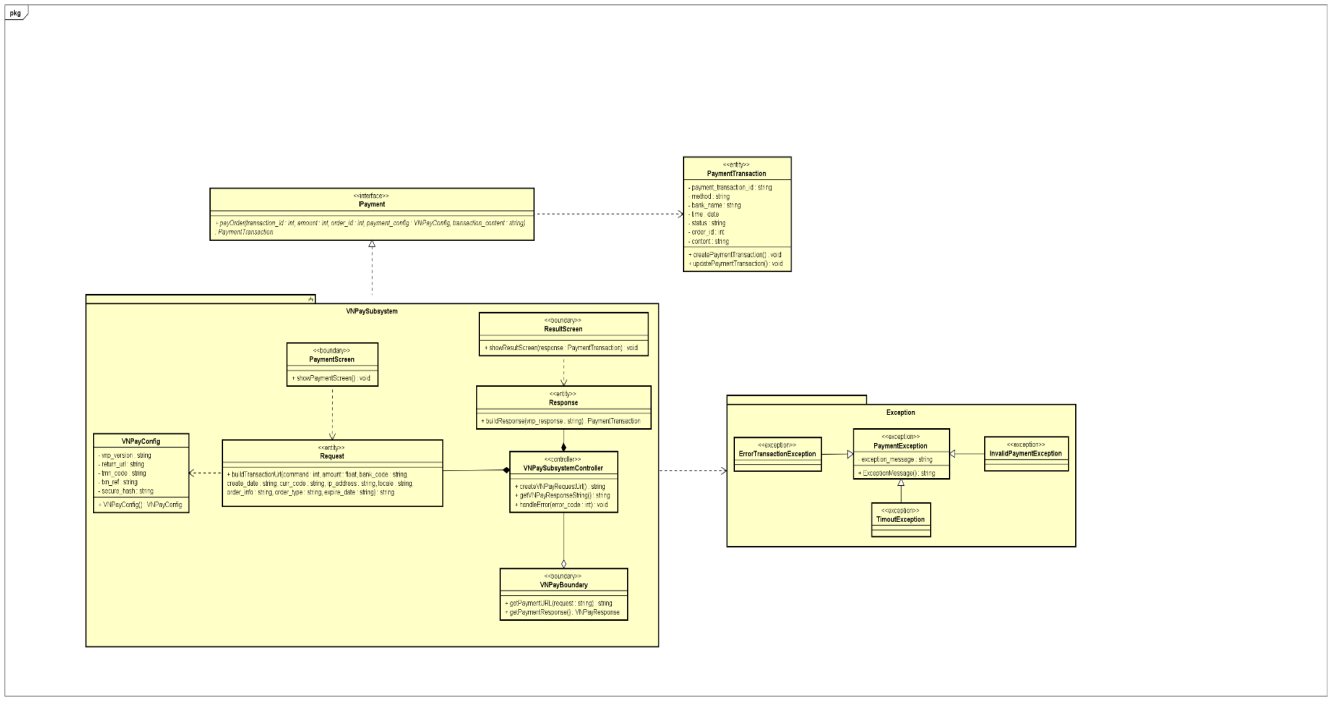


Overall Diagram



Detail Product Service

4.4.2.6 Class Diagram for Package VNPay Subsystem



4.4.3 Class Design

4.4.3.1 Class “IPayment”

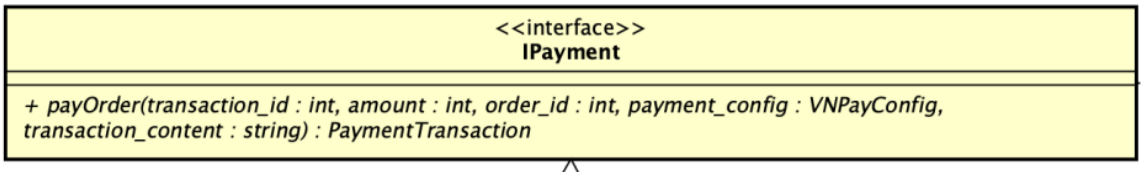


Table 1. Operation design for IPayment

#	*Name*	*Return type*	*Description (purpose)*
1	payOrder()	PaymentTransaction	Receives message from PayOrder usecase and makes request to a payment service such as VNPay and is expected to get the transaction detail back.

1. Parameter

- transaction_id: ID of the transaction in the system.

- amount: total amount of money for the transaction.

- order_id: ID of the order in system, currently serve a similar role transaction_id but can be extended to support payment with installment plan.

- payment_config: the config used to be passed to the payment service such as api version or secure hash.

- transaction_content: content of the order, expected to be a string of digits representing the transaction.

2. Exception

None

4.4.3.2 Class “Request”

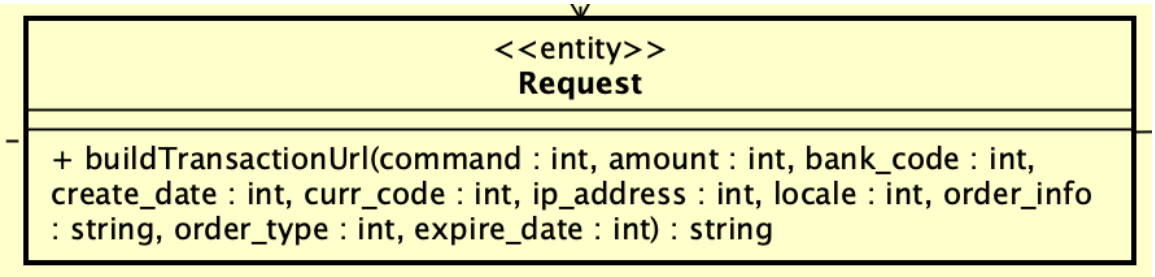


Table 2. Operation design for Request

#	*Name*	*Return type*	*Description (purpose)*
1	buildTransactionUrl()	string	Takes the reponse from VNPay and parses it to a standard response format to be used in the system.

1. Parameter

- command: string

API command code for payment transactions

- amount: float

Payment amount

- bank_code: string

Payment method code – bank or e-wallet code.

- create_date: string

Transaction creation time in format yyyy:MM:dd:HH:mm:ss

- curr_code: string

Currency code used for the transaction.

- ip_address: string

IP address of the customer performing the transaction.

- locale: string

Language for display interface.

- order_info: string

Description of the payment content

- order_type: string

Goods category code. Each product belongs to a category defined by VNPAY.

- expire_date: string

Payment expiration time with format yyyy:MM:dd:HH:mm:ss

2. Exception

None

3. Method

- buildTransactionUrl(command : int, amount : float, bank_code : string, create_date : string, curr_code : string, ip_address : string, locale : string, order_info : string, order_type : string, expire_date : string) : string

Gets the current config for VNPay request and builds the payment URL from VNPay based on the parameters and configs given.

4.4.3.3 Class "Response"

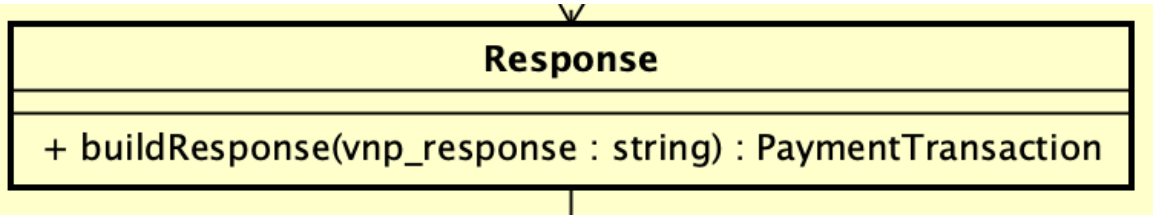


Table 3. Operation design for VNPaySubSystem

#	*Name*	*Return type*	*Description (purpose)*
1	BuildResponse	PaymentTransaction	Takes the reponse from VNPay and parses it to a standard response format to be used in the system.

4.4.3.4 Class "VNPaySubsystemController"

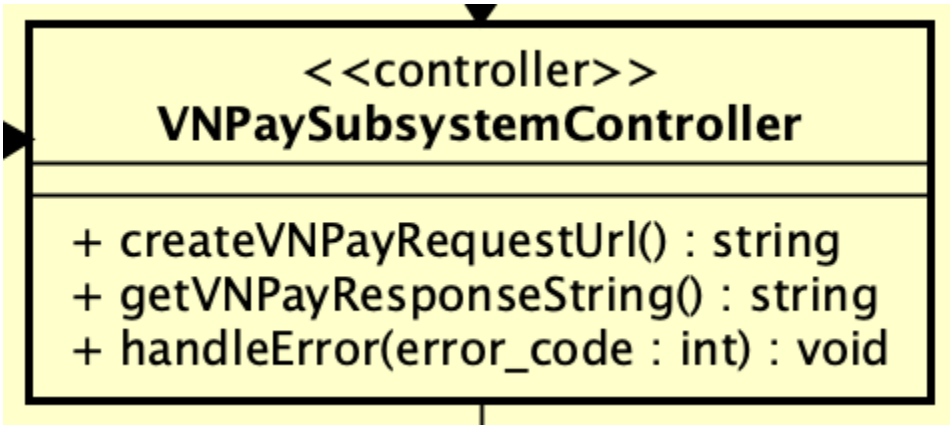


Table 4. Operation design for ProductService

#	Name	Return type	Description (purpose)
1	createVNPayRequestUrl	string	Returns a request string to pass to VNPay boundary.
2	getVNPayResponseString	string	Gets VNPay response and formats it to string format.
3	handleError	void	Calls the exception handler to handle exception and error when they happen.

1. Parameter

- error_code: int

An interger used to identify the error if any.

2. Exception

NonE

3. Method

- createVNPayRequestUrl(): string

Call the buildTransactionUrl operation from from Request entity to handle building the VNPay url string.

- getVNPayResponseString(): string

Call the getPaymentResponse operation from VNPayBoundary and parse the result into a string format to pass to Response entity.

- handleError(error_code: int): void

Specifically handle exceptions and errors

4.4.3.5 Class “VNpayConfig”

Table 4. Operation design for VNpayConfig

1. Table of Attributes

#	Name	Return type	Default Value	Description (purpose)
1	vnp_version	string	1.0	The API version that the merchant connects to to in the VNPAY system
2	return_url	string	null	Return URL to notify transaction results after the transaction session is ended
3	tmn_code	string	null	Merchant website code on VNPAY system to identifing the correct merchant, link their account, and process the transaction properly
4	txn_ref	string	null	Transaction reference code in the merchant system. This code is unique and used to distinguish orders sent to VNPA
5	secure_hash	string	null	This attribute is used to store a cryptographic hash of the configuration data to detect any unauthorized changes or tampering of the configuration data.

2. Table of Operations

#	Name	Return type	Description (purpose)
1	VNPayConfig	VNPayConfig	Getter for all the configuration values.

3. Parameter None

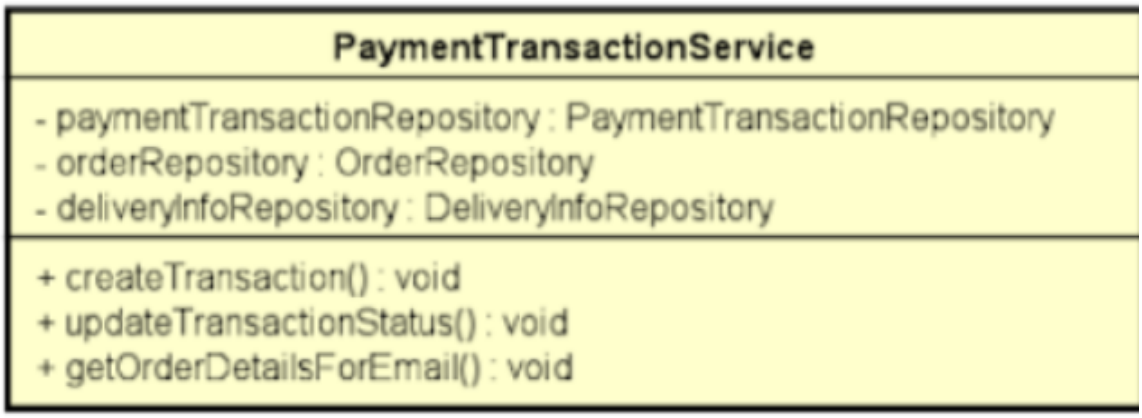
4. Exception None

4.4.3.6 Class diagram for “MailService”

Table of Operations

#	Name	Return type	Description (purpose)
1	sendConfirmation	void	Sending email with the delivery and transaction info to User who payment successful

4.4.3.7 Class diagram for “PaymentTransactionService”



1. Table of Attributes

#	Name	Return type	Default Value	Description (purpose)

1	paymentTransactionRepository	PaymentTransactionRepository	null	Manages the storage, retrieval, and update of payment transaction records, including creating new transactions and updating their statuses.
2	orderRepository	OrderRepository	null	Handles database operations related to customer orders, such as finding, updating, and retrieving order details.
3	deliveryInfoRepository	DeliveryInfoRepository	null	Responsible for accessing and managing delivery information associated with orders, including retrieving delivery details for a specific order.

2. Table of Operations

#	**Name**	**Return type**	**Description (purpose)**
1	createTransaction	void	create new transaction when place order
2	updateTransaction	void	Update status of transaction after order or cancel
3	getOrderDetailsforEmail	{deliveryinfo, order, transaction}	Get the information of order for notify user by email

3. Parameter

-Order_id: int

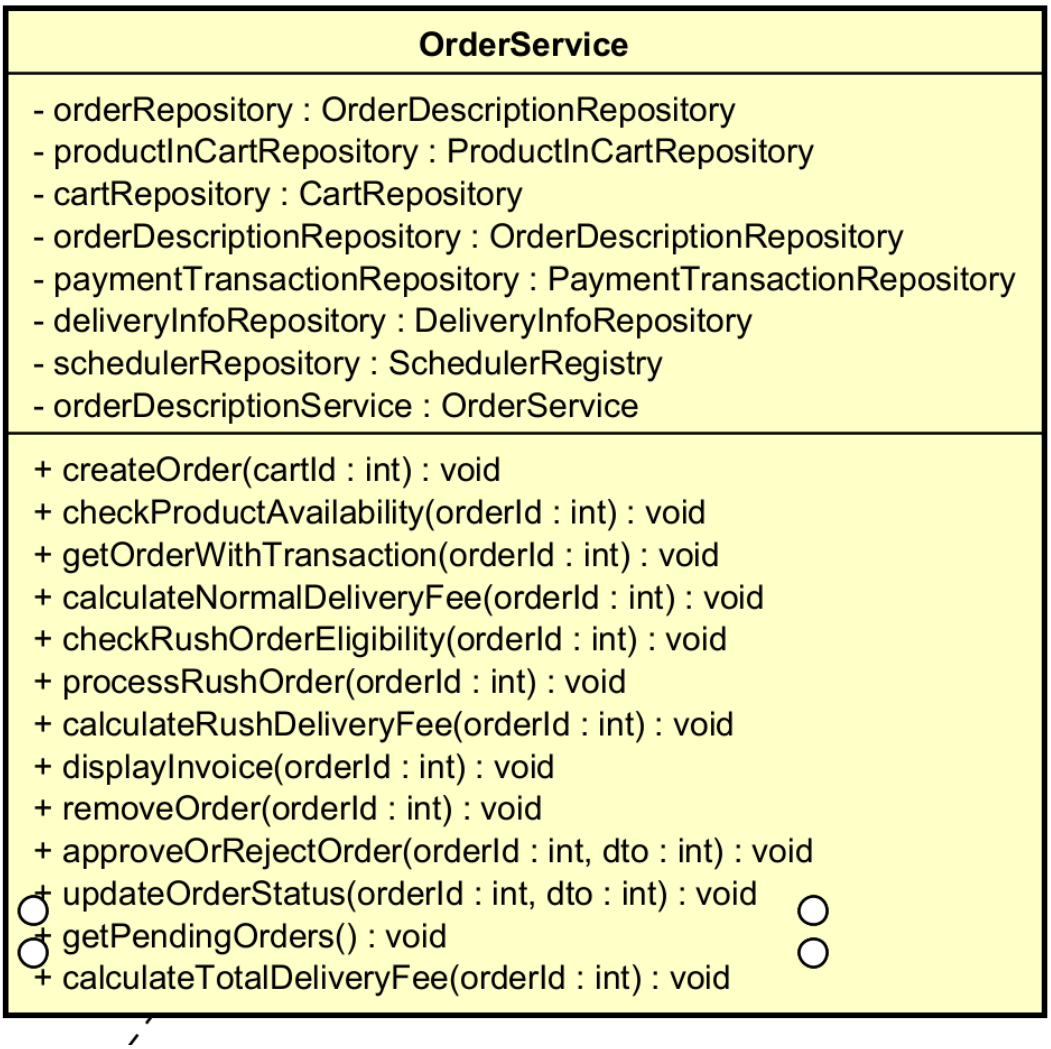
Tracking order_id to create new transaction, delivery_info

-paymenttransaction_id:

Tracking paymentTransaction_id for update status

4. Exception None

4.4.3.8 Class diagram for “OrderService”



1. Table of Attributes

#	Name	Type	Description (purpose)
1	orderRepository	OrderRepository	Handle database operations related order
2	productInCartRepository	ProductInCartRepository	Handle database operations related product in cart
3	cartRepository	CartRepository	Handle database operations related cart
4	orderDescriptionRepository	OrderDescriptionRepository	Handle database operations related order descriptions
5	paymentTransactionRepository	PaymentTransactionRepository	Handle database operations related payment transaction
6	deliveryInfoRepository	DeliveryInfoRepository	Handle database operations related delivery info
7	schedulerRepository	SchedulerRegistry	Manage and interact with crons, intervals, and timeouts
8	orderDescriptionService	OrderDescriptionService	Handle order services

2. Table of Operations

#	Name	Return type	Description (purpose)
1	createOrder	Order	Create an order with subtotal, accept_date and delivery_fee not filled yet. Attach each order with a timeout that order will be removed after 10 mins without payment
2	checkProduct Availability	void	Check if products quantity in database is sufficient for products in an order
3	getOrderWith Transaction	Order, PaymentTra nsaction	Return order with transaction to customer
4	calculateNorm alDeliveryFee	float	Calculate and return subtotal and normal delivery fee of an order
5	calculateRush DeliveryFee	float	Calculate and return subtotal and rush delivery fee of an order
6	displayInvoice	void	Return invoice information to customer
7	removeOrder	void	Remove order along with order descriptions belongs to it
8	checkRushOr derEligibility	void	Checks whether the given order qualifies for rush delivery
9	processRushO rder	void	Processes a rush order request for the specified order.

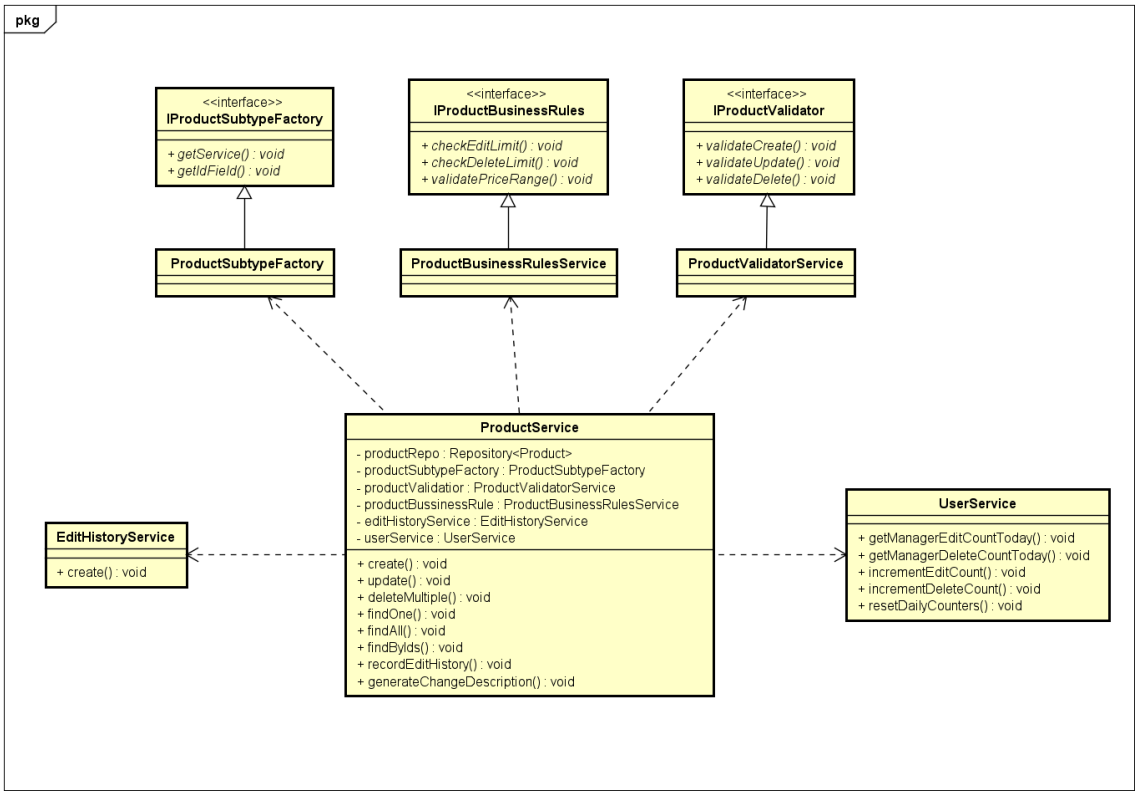
10	calculateTotal DeliveryFee	float	Calculate and return subtotal and normal delivery fee of an order in case just certain products are eligible for rush order delivery

3. Parameter

- cartId: unique cart ID
- orderId: unique order ID
- dto: include necessary fields of entity for specific use case, here is create order

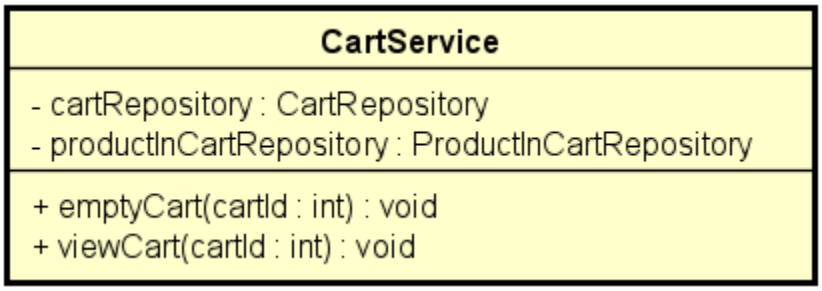
4. Exception None

4.4.3.9 Class “Product Service” for UC “Add/ Edit a Product”



#	Name of Class	Type	Description (purpose)
1	IProductSubtypeFactory	interface	Defines the contract for Factory pattern implementation. Ensures all factories must provide service creation and ID field retrieval
2	IProductBusinessRules	interface	Defines mandatory business rules. Ensures all implementations follow business requirements. Easy to modify business rules based on changing requirements
3	IProductValidator	inteface	Separates validation logic from business logic. Easy to change validation rules without affecting other components. Guarantees consistent validation across all implementations
4	ProductSubtypeFactory	Concrete class	Creates appropriate services based on product type (Book, CD, DVD, LP)
5	ProductValidatorService	Concrete class	Validates data integrity and business rules for inputs. (Prevents creation of duplicate products)
6	ProductBusinessRulesService	Concrete class	Implement AIMS logic rule for product manager permission/ price of products
7	EditHistory Service	Class	Take/ save information in user case “Edit a Product”
8	UserService	Class	Support Product Validation Services and user case “Edit a Product”

4.4.3.10 Class diagram for “CartService”



1. Table of Attributes

#	Name	Type	Description (purpose)
1	cartRepository	CartRepository	Handle database operations related to cart
2	productInCartRepository	ProductInCartRepository	Handle database operations related to product in cart

2. Table of Operations

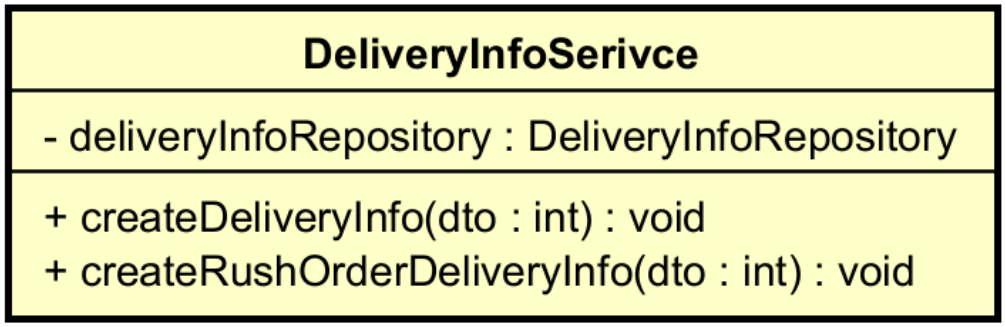
#	Name	Return type	Description (purpose)
1	emptyCart	void	Remove all products from cart
2	viewCart	void	View all products information from cart

3. Parameter

- cartId: unique cart ID

4. Exception None

4.4.3.11 Class diagram for “DeliveryInfoService”



1. Table of Attributes

#	Name	Type	Description (purpose)
1	deliveryInfoRepository	DeliveryInfoRepository	Handle database operations related to delivery info

2. Table of Operations

#	Name	Return type	Description (purpose)

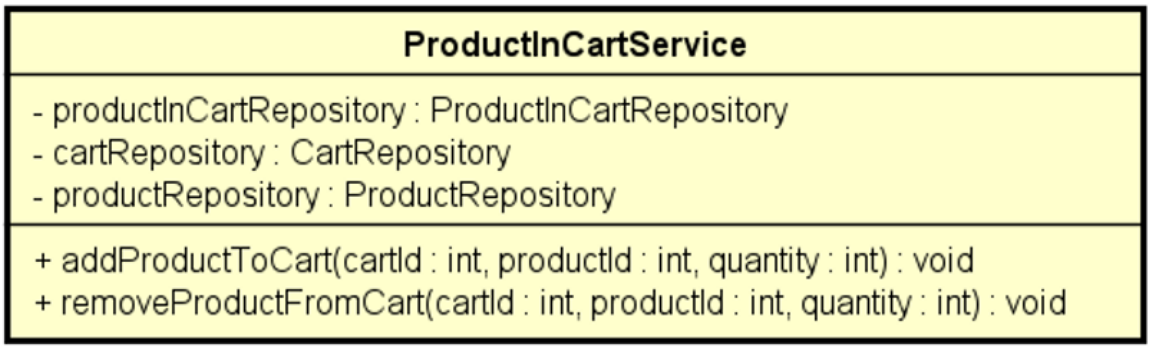
1	createDelivery Info	DeliveryInfo	Create delivery info belongs to an order; automatically calculate and update subtotal and normal delivery fee to order
2	createRushOrd erDeliveryInfo	DeliveryInfo	Create rush order delivery information

3. Parameter

- dto: include necessary fields of entity for specific use case, here is create delivery info

4. Exception None

4.4.3.12 Class diagram for “ProductInCartService”



1. Table of Attributes

#	Name	Type	Description (purpose)

1	productInCartRepository	ProductInCartRepository	Handle database operations related to product in cart
2	cartRepository	CartRepository	Handle database operations related to cart
3	productRepository	ProductRepository	Handle database operations related to product

2. Table of Operations

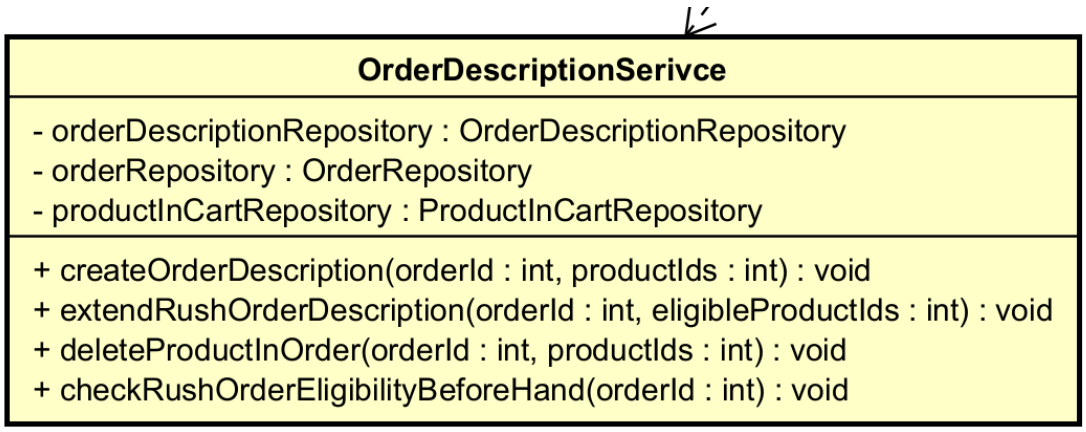
#	Name	Return type	Description (purpose)
1	addProductToCart	void	If product quantity in database is sufficient: if product not in cart yet, add new product to cart; else increase quantity of product in cart. If product quantity in database is insufficient: raise error
2	removeProductFromCart	void	If product exists in cart: if removing quantity >= product quantity in cart, remove product from cart; else just decrease product quantity from cart. If product not exist in cart: raise error

3. Parameter

- cartId: unique cart ID
- productId: unique product ID
- quantity: adding/removing product quantity

4. Exception None

4.4.3.13 Class diagram for “OrderDescriptionService”



1. Table of Attributes

#	Name	Type	Description (purpose)
1	orderDescription Repository	OrderDescription Repository	Handle database operations related to order description
2	orderRepository	OrderRepository	Handle database operations related to order
3	productInCartRe pository	ProductInCartRe pository	Handle database operations related to product in cart

2. Table of Operations

#	Name	Return type	Description (purpose)

1	createOrderDescription	void	Create an order description according to an order via order description
2	deleteProductInOrder	void	Remove a product from an order via order description
3	extendRushOrderDescription	void	Updates order descriptions by setting the is_rush attribute to true for all products that were successfully verified as eligible for rush delivery
4	checkRushOrderEligibilityBeforeHand	boolean	Check if any product is eligible for rush order assuming that the condition for delivery info is validated for rush order

3. Parameter

- orderId: unique order ID
- productIds: products ID that are added/removed to/from order

4. Exception None

4.4.3.14

Design Considerations

4.5 Goals and Guidelines

Key goals and guidelines shape the design of the system:

- **Consistency and Maintainability**: Adhere to strict coding standards and conventions to enhance code readability and facilitate future maintenance.
- **User-Centric Design**: Prioritize intuitive user interfaces that are easy to navigate, ensuring the application aligns with user expectations.
- **Scalability**: Design with scalability in mind to accommodate future growth and increased user demands.

4.6 Architectural Strategies

- **RESTful API Design**: Implement RESTful APIs using NestJS to enable seamless communication between the frontend and backend, ensuring flexibility and ease of integration with the ReactJS frontend.
- **Database Management**: Use **PostgreSQL** as the relational database management system (RDBMS) for structured data storage, ensuring data integrity and efficient querying capabilities.
- **Component-Based UI Development**: Employ ReactJS for frontend development, leveraging its component-based architecture to create reusable UI elements and improve development efficiency.
- **Object-Oriented Programming (OOP)**: Apply OOP principles in NestJS for backend development, promoting code organization, encapsulation, and code reuse through modules, services, and controllers.

4.7 Coupling and Cohesion

Our project leverages NestJS to achieve high cohesion and low coupling, which are essential qualities of maintainable and scalable software. High cohesion is reflected in the way NestJS modules, services, and controllers are organized—each class or module has a well-defined responsibility. For example, the PayOrderController is dedicated solely to handling payment-related operations, making the codebase easier to understand, test, and maintain. Low coupling is accomplished through the use of interfaces and NestJS's built-in dependency injection system. For instance, the PayOrderController and related services depend on the PaymentGateway interface rather than a specific payment gateway implementation. The actual payment strategy (such as VNPay) is selected at runtime by the PaymentGatewayFactory. This approach decouples the business logic from specific implementations, allowing for greater flexibility and making it easier to modify or extend payment methods in the future.

4.8 Design Principles

Our design adheres to the SOLID principles, which are fundamental to creating maintainable and scalable software.

Single Responsibility Principle (SRP): The project demonstrates strong adherence to SRP. Each module (e.g., product, order, cart, payment-transaction) is organized so that controllers handle HTTP requests, services encapsulate business logic, and entities represent data models. This clear separation of concerns makes the codebase easier to maintain and extend.

Open-Closed Principle (OCP): The use of interfaces (such as for payment gateways) and NestJS's modular structure allows for easy extension of functionality without modifying existing code. For example, new payment methods or delivery strategies can be added by implementing the appropriate interfaces, keeping the core logic untouched.

Liskov Substitution Principle (LSP): The project's reliance on interfaces and abstract classes ensures that new implementations (e.g., a new payment gateway service) can replace existing ones without breaking the system. This is especially evident in areas like payment processing, where different strategies can be swapped seamlessly

Interface Segregation Principle (ISP): Interfaces in the project are generally well-defined and focused, ensuring that classes only implement what they need. This reduces unnecessary dependencies and keeps modules clean and maintainable.

Dependency Inversion Principle (DIP): The project makes effective use of NestJS's dependency injection, allowing high-level modules to depend on abstractions rather than concrete implementations. This decouples components and facilitates easier testing and future modifications.

Overall, the project exhibits a strong application of the SOLID principles. The codebase is modular, extensible, and easy to maintain, with clear boundaries between responsibilities and a flexible architecture that supports future growth. While no project is perfect, and ongoing refactoring is always beneficial, the current structure provides a solid foundation for scalable and professional software development.

4.9 Design Patterns

4.9.1 Dependency Injection pattern:

In our application, we apply the **Dependency Injection (DI)** design pattern, which is a core feature of the NestJS framework. This pattern enables us to manage and supply class dependencies in a clean and structured way, promoting loose coupling and high cohesion across our codebase. With DI, we do not need to manually create instances of services or classes; instead,

NestJS's built-in Inversion of Control (IoC) container automatically resolves and injects the required dependencies at runtime. This approach makes our application more modular, testable, and easier to maintain. It also simplifies unit testing, as we can easily replace real implementations with mocks or stubs. By leveraging DI, we are able to scale our application more efficiently while keeping the codebase organized and aligned with SOLID principles.

4.9.2 Strategy Pattern

In the design of the current project, we have utilized the Strategy Pattern, which is a behavioral design pattern that allows selecting an algorithm at runtime. The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from the clients that use it. The reason for using the Strategy Pattern in this project is to provide a flexible way to select different algorithms or business logic at runtime. This is particularly The Strategy Pattern is applied in the payment-transaction module, where we might have different ways to process payments based on various gateways such as VNPay, PayPal, or others. The PaymentGatewayFactory service encapsulates this logic, allowing the system to select and use the appropriate payment strategy (e.g., VNPay) at runtime, making it easy to extend or modify payment methods without changing the core business logic.

We first define a PaymentGateway interface that declares methods for processing payments. This interface can be implemented by different payment gateway classes, such as VNPayService, each providing its own logic for handling payments.

Then, we create a concrete class VNPayService that implements the PaymentGateway interface. This class provides a specific implementation for processing payments through VNPay.

In the service or controller where payments are handled, we use the PaymentGateway interface to process payments. The specific implementation (e.g., VNPayService) is determined at runtime by the PaymentGatewayFactory, which selects the appropriate payment strategy based on the payment method requested.

By using the Strategy Pattern, we can easily switch between different payment gateway strategies (such as VNPay, PayPal, etc.) without modifying the core

payment processing logic. This makes our codebase more flexible, extensible, and easier to maintain.

In the Product module, the Strategy Pattern is used for both validation and business rules enforcement. Interfaces such as IProductValidator and IProductBusinessRules define contracts for validation and business logic, while concrete classes like ProductValidatorService and ProductBusinessRulesService implement these strategies.

The ProductService depends on interfaces (IProductValidator, IProductBusinessRules) rather than concrete implementations.

Different strategies for validation and business rules can be injected into the service, allowing for flexible and interchangeable logic. For example, validation logic for creating, updating, or deleting products is encapsulated in the ProductValidatorService, and can be swapped or extended as needed.

Benefits:

- Flexibility: Different validation or business rule strategies can be used without changing the main product logic.
- Testability: Strategies can be easily mocked or replaced for testing purposes.
- Separation of Concerns: Validation and business rules are separated from the core product logic, making the codebase cleaner and easier to maintain.

4.9.3 Factory Pattern

The Factory Pattern is a creational design pattern that provides an interface for creating objects, allowing the actual instantiation to be determined by subclasses or specific logic. This pattern promotes loose coupling by delegating the responsibility of object creation to a factory class or method, making it easier to introduce new types or change instantiation logic without modifying existing code.

In the context of the payment gateway module, the Factory Pattern is used to encapsulate the logic for selecting and instantiating the appropriate payment gateway (such as VNPay) at runtime, based on the payment method requested. The PaymentGatewayFactory class is responsible for returning the correct

payment gateway implementation, allowing the core payment processing logic to remain unchanged even as new gateways are added. This enhances the flexibility and maintainability of the application.

Moreover, in the Product module, the ProductSubtypeFactory class implements the Factory Pattern. Its main responsibility is to provide the correct service for handling different product subtypes (e.g., Book, CD, DVD, LP) based on the product type..

The ProductSubtypeFactory maintains a mapping between product types and their corresponding service classes.

When a product operation (create, update, etc.) is performed, the factory is used to retrieve the appropriate service for the product subtype. This allows the ProductService to work with any product subtype in a generic way, without hardcoding the logic for each subtype.

Benefits:

- Extensibility: New product types can be added easily by registering new services in the factory, without modifying existing business logic.
- Encapsulation: The creation logic for subtype services is centralized, making the codebase easier to maintain.
- Decoupling: The main product logic is decoupled from the specifics of each subtype, adhering to the Open/Closed Principle.