HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

School of Information and communications technology

Software Requirement Specification

Version 1.3

AIMS Project

Subject: ITSS Software Development

Group 07:

No	Name	Student ID
1	Trịnh Thị Thuỳ Dương	20226034
2	Nguyễn Thị Thu Huyền	20220073
3	Hoàng Khải Mạnh	20225984
4	Đoàn Thị Thu Quyên	20226063
5	Dương Phương Thảo	20226001

Hanoi, May 11^{st} *2025*

Hanoi, <May, 2025>

<All notations inside the angle bracket are not part of this document, for its purpose is for extra instruction. When using this document, please erase all these notations and/or replace them with corresponding content as instructed>

<This document, written by Prof. NGUYEN Thi Thu Trang, is used as a case study for student with related courses. Any modifications and/or utilization without the consent of the author is strictly forbidden>

Table of Contents

List of Figures

No table of figures entries found.

List of Tables

1 Introduction

<The following subsections of the Software Design Document (SDD) document should provide an overview of the entire SDD.>

1.1 Objective

<Identify the purpose of this SDD and its intended audience. In this subsection, describe the purpose of the SDD and specify the intended audience for the SDD>

1.2 Scope

The software product to be developed is **AIMS (An Internet Media Store)**, a digital platform that enables users to browse, purchase, and download a wide range of media content, including applications, music, videos, and e-books. AIMS serves as a centralized marketplace designed to deliver a seamless and secure media consumption experience across web and mobile platforms. It can handle up to 1,000 customers at once without slowing down much and works for 300 hours without crashing. If something goes wrong, it gets back to normal within 1 hour. It responds in 2 seconds normally or 5 seconds during busy times.

AIMS will allow users to create accounts, search and filter media content, manage their purchases, and access their media library anytime. The platform will also support content uploads and management for verified publishers. Its primary goals include enhancing media accessibility, streamlining digital content distribution, and supporting user-friendly interactions.

AIMS will not handle content creation or moderation beyond basic automated checks; these responsibilities lie with the content providers and external policies. This overview is consistent with the system's Software Requirements Specification (SRS) and outlines the scope, benefits, and intended use of the application without detailing individual requirements.

1.3 Glossary

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 rules that allows different software applications to communicate with each order	VNPay API	Includes online shopping, digital payments, and logistics
3	E-Commerce (Electronic Commerce)	The buying and selling of goods and services over the internet	Amazon, Shopee	Includes online shopping, digital payment, and logistics

4	VAT (Value Added Tax	Value-added tax, a consumption tax added to goods and services.	10% VAT in Vietnam	Applied at each stage of production and distribution
5	Rush Order	An order that requires expedited processing and delivery.	Express shipping, Same-day delivery.	May involve additional fees for faster service.
6	Payment Gateway	A service that authorizes and processes online payments securely.	VNPay	Ensures secure transactions between customers and merchants.

1.4 References

Centers for Medicare & Medicaid Services. (n.d.). Test Case Specification. Retrieved from Centers for Medicare & Medicaid Services: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/TestCaseSpecification.docx

Centers for Medicare & Medicaid Services. (n.d.). Test Plan. Retrieved from Centers for Medicare & Medicaid Services: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/TestPlan.docx

guru99. (n.d.). Download Sample Test Case Template: Example Excel, Word Formats. Retrieved from guru99: https://www.guru99.com/download-sample-test-case-template-with-explanation-of-important-fields.html

2 Overall Description

<This section describes the principles and strategies to be used as guidelines when designing and implementing the system.>

2.1 General Overview

2.1.1. System Context and Design Philosophy

The design of AIMS (An Internet Media Store) centers around delivering a robust and scalable e-commerce platform for physical media products, tailored to meet both business requirements and user expectations. At its core, the system is built to be reliable, responsive, and easy to use — even under demanding conditions. Key design objectives include:

- **High Performance**: The system is expected to handle user requests swiftly, ensuring responsive interactions even during peak traffic periods.
- **Scalability**: AIMS aims to support up to 1,000 concurrent users, providing a seamless experience regardless of user volume.
- **Reliability**: The system should remain operational for at least 300 hours continuously and be able to recover within 1 hour following any downtime.
- **Security**: Strong security practices are integrated, including encrypted data storage and secure user authentication, to protect user privacy and transaction safety.

2.1.2. System and Software Architecture

- **Client Application**: A modern, web-based interface built using technologies like React, HTML, CSS, and JavaScript. It enables users to browse products, manage shopping carts, and complete purchases with ease.
- **Server Application**: The server-side logic is implemented using the SpringBoot framework, which handles business operations, database management, and integration with third-party services.
- **Database**: A relational database serves as the backbone for storing user accounts, product listings, order details, and transaction history.
- **External Integration**: The system integrates with **VNPay**, a third-party payment gateway, to securely process credit card transactions.

2.2 Assumptions/Constraints/Risks

2.2.1 Assumptions

<Describe any assumptions or dependencies regarding the system, software and its use. These may concern such issues as: related software or hardware, operating systems, end-user characteristics, and possible and/or probable changes in functionality>

- **Platform**: Users will interact with AIMS via up-to-date web browsers such as Chrome, Firefox, Safari, or Edge.

- **User Capabilities**: It is assumed that users have basic computer literacy and stable internet access.

- **Third-party Services**: VNPay's APIs are assumed to be stable and consistently available.

- **Scope**: The system will exclusively handle physical media items; digital goods and subscription services are not included.

2.2.2 Constraints

<Describe any global limitations or constraints that have a significant impact on the design of the system's hardware, software and/or communications, and describe the associated impact. Such constraints may be imposed by any of the following (the list is not exhaustive):

· *Hardware or software environment*

· *End-user environment*

· *Availability or volatility of resources*

· *Standards compliance*

· *Interoperability requirements*

· *Interface/protocol requirements*

· *Licensing requirements*

· *Data repository and distribution requirements*

· *Security requirements (or other such regulations)*

· *Memory or other capacity limitations*

· *Performance requirements*

· *Network communications*

· *Verification and validation requirements (testing)*

· *Other means of addressing quality goals*

· *Other requirements described in the Requirements Document >*

- **Hardware Environment**: The AIMS system must run smoothly on standard consumer computers without requiring high-performance hardware, which necessitates performance optimization and limited resource usage.

- **User Environment**: The user interface must be simple and intuitive for non-technical users, influencing the overall UI/UX design approach.

- **Resource Availability**: The system must support up to 1,000 concurrent users and operate continuously for 300 hours, requiring efficient resource management and quick recovery mechanisms.

- **Performance Requirements**: The application must respond within 2 seconds under normal load and within 5 seconds during peak times, affecting logic design and data handling strategies.

- **Network Communications**: Communication between frontend and backend must be stable and low-latency to support real-time feedback, requiring optimized APIs and asynchronous processing.

- **Standards Compliance & Licensing**: Integration with VNPay requires strict adherence to their API standards, which impacts the modular design of the payment system.

- **Interoperability & Protocol Requirements**: The system must interoperate with external APIs using standard protocols like HTTPS and JSON, affecting how external communications are structured and validated.

- **Security Requirements**: Security of user and payment information is critical, necessitating role-based access control, input validation, and data encryption.

- **Data Repository Constraints**: The system must store detailed logs of products, transactions, and user actions, impacting database schema design and logging mechanisms.

- **Capacity Limitations**: Business rules such as deleting a maximum of 30 products at once or updating prices only twice per day must be strictly enforced through system logic.

- **Verification and Validation Requirements**: All input must be validated with user-friendly error feedback, increasing the need for input checking, error handling, and testing coverage.

- **Scalability and Future Expansion**: The system should be scalable to support new product types and payment methods, requiring modular architecture, SOLID principle adherence, and appropriate design patterns.

2.2.3 Risks

<Describe any risks associated with the system design and proposed mitigation strategies.>

Risk	Description	Mitigation Strategy
Downtime	System outages may affect the shopping experience.	Employ redundancy and automatic failover systems.
Security Vulnerabilities	Risks of data breaches and unauthorized access.	Perform regular security audits, updates, and enforce best practices.
Scalability Limitations	Inability to support increasing traffic.	Optimize queries and backend logic, and implement load balancing.
Payment Integration Issues	Failures in VNPay integration could block payments.	Conduct continuous integration testing and maintain communication with VNPay support.
Performance Bottlenecks	Slow responses under heavy usage.	Conduct load testing, enable caching, and optimize backend processes.
Data Loss	Potential data corruption or loss due to bugs or hardware failure.	Schedule regular backups and implement integrity checks.

3 System Architecture and Architecture Design

<Briefly describe the architectural design steps>

3.1 Architectural Patterns

<Specify and briefly describe the chosen architectural patterns and the reasons why they were chosen>

3.2 Interaction Diagrams

3.3 Analysis Class Diagrams

3.4 Unified Analysis Class Diagram

3.5 Security Software Architecture

<Describe the software components and configuration supporting the security and privacy of the system. Specify the architecture for (1) authentication to validate user identity before allowing access to the system;(2) authorization of users to perform functional activity once logged into the system, (3) encryption protocol to support the business risks and the nature of information, and (4) logging and auditing design, if required.>

4 Detailed Design

4.1 User Interface Design

<Suppose that you design a Graphical User Interface (GUI)>

4.1.1 Screen Configuration Standardization

4.1.2 Screen Transition Diagrams

4.1.3 Screen Specifications

<Screen images should be included in the screen specifications>

4.2 Data Modeling

4.2.1 Conceptual Data Modeling

<E-R Diagram image and description of entities and relationships>

4.2.2 Database Design

4.2.2.1 Database Management System

<Specify what is the decision of Database Management System (DBMS) and give some description of the DBMS>

4.2.2.2 Database Diagram

<

- *Show the process to design database from E-R diagram*
- *Show the diagram of DB design*

>

4.2.2.3 Database Detail Design

<

Give a detail design of each element in the DB diagram. For instance, in a Relational DBMS, give a detail design for each Table and their constraints, illustrated in below table (PK: Primary Key, FK: Foreign Key).

Table 1. Example of table design

#	PK	FK	Column name	Data type	Default value	Mandatory	Description
1	x		ProductID				
2		x	CategoryID				

You may add indexing, trigger, view, etc.

Give the database script>

4.3 Non-Database Management System Files

<Provide the detailed description of all non-DBMS files if any and include a narrative description of the usage of each file that identifies if the file is used for input, output, or both, and if the file is a temporary file. Also provide an indication of which modules read and write the file and include file structures (refer to the data dictionary). As appropriate, the file structure information should include the following:

• *Record structures, record keys or indexes, and data elements referenced within the records*

• *Record length (fixed or maximum variable length) and blocking factors*

• *Access method (e.g., index sequential, virtual sequential, random access, etc.)*

• *Estimate of the file size or volume of data within the file, including overhead resulting from file access methods*

- *Definition of the update frequency of the file (If the file is part of an online transaction-based system, provide the estimated number of transactions per unit of time, and the statistical mean, mode, and distribution of those transactions.)*

- *Backup and recovery specifications>*

4.4 Class Design

4.4.1 General Class Diagram

<General class diagram which shows the whole class diagram of the software. This diagram may have packages, subsystems and classes. Classes in this diagram may not have all attributes and operations>

4.4.2 Class Diagrams

<Detail class diagram with full attributes and operations>

4.4.2.1 Class Diagram for Package A

4.4.2.2 Class Diagram for Subsystem B

…

4.4.3 Class Design

<Detail design for each class>

4.4.3.1 Class “SampleClass1”

<SampleClass1 class image in UML>

Table 2. Example of attribute design

#	*Name*	*Data type*	*Default value*	*Description*
1				
2				

Table 3. Example of operation design

#	*Name*	*Return type*	*Description (purpose)*
1			
2			

Parameter:

- x: Default value, description
- y: Default value, description

Exception:

- AException if …
- BException if …

Method

How to use parameters / attributes

Flowchart / activity diagram / sequence diagram if the method has a complex/special algorithm

State

State diagram if any

4.4.3.2 Class “SampleClass2”

…

5 Design Considerations

<Describe issues which need to be addressed or resolved before attempting to devise a complete design solution. Remember that, you have to refactor your source code to strictly follow the final design>

5.1 Goals and Guidelines

<Describe any goals, guidelines, principles, or priorities which dominate or embody the design of the system and its software.

Examples of such goals might be: an emphasis on speed versus memory use; or working, looking, or "feeling" like an existing product.

Guidelines include coding guidelines and conventions.

For each such goal or guideline, describe the reason for its desirability unless it is implicitly obvious.

Describe any design policies and/or tactics that do not have sweeping architectural implications (meaning they would not significantly affect the overall organization of the system and its high-level structures), but which nonetheless affect the details of the interface and/or implementation of various aspects of the system (e.g., choice of which specific product to use)>

5.2 Architectural Strategies

<Describe any design decisions and/or strategies that affect the overall organization of the system and its higher-level structures. These strategies should provide insight into the key abstractions and mechanisms used in the system architecture. Describe the reasoning employed for each decision and/or strategy (possibly referring to previously stated design goals and principles) and how any design goals or priorities were balanced or traded-off.

Examples of design decisions might concern (but are not limited to) things like the following:

- *Use of a particular type of product (programming language, database, library, commercial off-the-shelf (COTS) product, etc.)*
- *Reuse of existing software components to implement various parts/features of the system*
- *Future plans for extending or enhancing the software*
- *User interface paradigms (or system input and output models)*
- *Hardware and/or software interface paradigms*
- *Error detection and recovery*
- *Memory management policies*
- *External databases and/or data storage management and persistence*

- *Distributed data or control over a network*
- *Generalized approaches to control*
- *Concurrency and synchronization*
- *Communication mechanisms*
- *Management of other resources*

>

5.3 Design and Program Evaluation

<Evaluate your design and describe which levels of coupling and cohesion that your design is at. Give proofs for your assumptions. Explain if there is any special design or exceptions>

<You may show the previous design from which you made improvements to get better levels of coupling and cohesion. You should clarify how and why you did these improvements>

<Does your design follow the SOLID principles if there are new requirements/changing requirements in the future? Give proofs for your assumptions. Explain if there is any special design or exceptions>

<You may show the previous design from which you made improvements to get a better design, which follows SOLID principles in spite of additional requirements. You should clarify how and why you did these improvements>

5.3.1 Cohesion & SRP

Explain in detail for each class in the table.

Table 4. Cohesion & SRP of AIMS

#	*Class Name*	*PIC*	*Cohesion*	*SRP*	*Solution*
x	Order	TuấnNH	Temporal: m1(), m2()	Yes: 2 responsibilities: r1, r2	Separate m2() to the class XXX…
y	<class name>	<person in charge>	<which level, why, where>	<Yes/No, why, where>	<solution if necessary>

1	PlaceRushOrderService.java	Quyen_20226063	Medium – Deals with both address eligibility and product filtering; logic is somewhat cohesive but spans multiple steps of rush order processing.	No – Has 2 responsibilities: (1) checking address/product eligibility, and (2) building the response DTO.	Refactor suggestion: Split into two classes: • RushOrderEligibilityChecker (logic) • PlaceRushOrderService (orchestrator only).
2	PlaceRushOrderResponse	Quyen_20226063	High – Groups together rush-related results (products and message).	Yes – Stores response results. 1 responsibility	
3	Product	Quyen_20226063 Huyen_20220073	High – All fields and methods pertain to product identity and rush eligibility.	Yes – Only manages product data (ID, name, rush eligibility). 1 responsibility.	
4	DeliveryInfo DTO	Quyen_20226063	High – All fields and methods relate to representing a delivery address.	Yes – Models delivery address. 1 responsibility.	Remove or clarify the ambiguous 4-parameter constructor; it reduces clarity and cohesion.
5	RushAddressProperties	Quyen_20226063	High – Solely binds configuration	Yes – Manages external rush delivery address	

			values from properties.	configuration. 1 responsibility.	
6	AddressRushEligibility	Quyen_20226063	Medium–High – Checks eligibility based on address data.	Yes – Evaluates eligibility based on delivery info. 1 responsibility.	
7	ProductRushEligibility	Quyen_20226063	High – Only concerned with product-based eligibility.	Yes – Checks product rush eligibility. 1 responsibility.	
8	RushEligibility<T> (Interface)	Quyen_20226063	High – General contract for rush eligibility logic.	Yes – Declares a single method for checking eligibility. 1 responsibility.	
9	Order	Duong_20226034 Thao_20226001	Functional Cohesion – All methods and fields are related to the single responsibility: managing an order	No – Just a data class.	
10	PaymentTransaction	Duong_20226034	Functional Cohesion – All fields serve the single purpose of representing a payment transaction.	No – Just a data class.	If future business logic related to payment arises, consider moving those responsibilities into a separate service class like

					PaymentServi ce.
11	PayOrderSe rvice	Duong_202 26034	Communicatio nal Cohesion – Methods share common data and purpose (processing payments) but test logic reduces clarity of single-purpos e design	Yes – 1 responsibilities: 1. setCurrentOrderFor Test	-Extract test logic into a separate mock or test utility class - Use proper dependency injection (e.g. repository) to get Order and PaymentTrans action instead of using instance-level test data - Keep PayOrderServi ce focused only on real payment logic → move test-specific state out
12	VNPaySubs ystem	Duong_202 26034	Functional Cohesion – All fields and methods support the single purpose of integrating with VNPay payment system	No, Class handles only VNPay-specific operations	
13	CartService	Manh_2022 5984	Procedural: methods like getCartItems() , addToCart() handle different	**Yes**: 3 responsibilities — 1. manage cart logic	Extract to smaller services like InventoryValid ator, CartItemMapp

			sequential tasks but share no common goal	2. product validation 3. DTO mapping	er, and possibly CartOperation Handler
14	DVD, CD, Book	Manh_2022 5984	**Functional**: Holds fields related to products for each specific type only.	No – Just a data class.	
15	GlobalExcep tionHandler	Manh_2022 5984	**Logical Cohesion**: Many methods handle exceptions, but each method handles a different type of exception with different logic and structure	**Yes**: This class handles multiple exception types and also formats different response payloads (e.g., ApiException, Map<String, String>)	Split into specialized handlers: • Optionally group related exception handlers into separate handler classes if the number increases.
16	CartItem	Manh_2022 5984	**Functional**: Holds fields related to cart for only	No – Just a data class.	
17	AuthService	Manh_2022 5984	**Logical Cohesion**: – login() handles authentication and token generation. –register() handles validation, user creation,	**Yes**: – 3 responsibilities: 1. Authentication 2. Account creation 3. Token issuance	Extract: •TokenService for JWT logic •UserRegistrati onService for validation + user creation • Let AuthService focus on

			and post-registration login.		coordination and response construction.
18	ProductService	Huyen_20220073	**Logical Cohesion**: –getAllProducts(), getProductById(), searchProducts() retrieve product data. –createProduct(), updateProduct() handle creation & update of multiple product types. –convertToDTO() transforms models into DTOs.	**Yes**: 4 responsibilities — 1. CRUD product logic 2. validation 3. type handling 4. DTO mapping.	Extract: ProductValidator, DTOConverter, and delegate to specific services per category (BookService, CDService, etc.).
19	OrderService	Huyen_20220073	**Low to Medium** cohesion: This class handles multiple tasks such as creating orders, updating status, processing payments, managing cart items, generating invoices, etc. These	**No:** The class violates SRP by taking on multiple responsibilities: order creation, updating order status, processing payment, managing inventory, handling delivery, etc.	Split this class into smaller services (e.g., OrderCreationService, PaymentService, InventoryService) each focusing on a single responsibility.

			responsibilities are not tightly related.		
20	ProductOrderEntity	Thao_20226001	Cohesion: High – all fields/methods relate to product order	**Yes** - 1 responsibility- data storage	
21	PlaceOrderService	Thao_20226001	The methods in PlaceOrderService such as checkAddressForRushOrder(), checkRushOrder(), createOrder(), calculateTotalAmount(), and saveProductOrders() perform different tasks in the order placement workflow	Yes - 4 reponsibilities Order creation – saving order, delivery info, product orders. Validation – check rush order support and address. Type conversion – mapping from DTOs to entities (InvoiceDTO → Order, etc.). Error handling – try-catch, null checks.	

5.3.2 Coupling & Other SOLID

Explain in detail for each problem in the table. Each problem, you may provide a class diagram before and after the improvement.

Table 5. Coupling & other SOLID of AIMS

#	*Problem*	*PIC*	*Location*	*Solution*
	Control coupling	TuấnNH	Class A and class B	Create a new class C as a superclass for A and B…
	<Any bad coupling level or any SOLID violation>	<person in charge>	<list of class or you may provide a class diagram if necessary>	
1	Low Cohesion (doing multiple things: validation, classification, DTO building)	Quyen_20226063	PlaceRushOrderService	Split into separate classes: RushOrderEligibilityChecker to check rush support RushOrderResponseAssembler for building DTO
2	Control Coupling – logic depends on how RushEligibility<T> is implemented	Quyen_20226063	PlaceRushOrderService	Add a coordinator like RushPolicyEvaluator to delegate address and product checks in a more declarative way.
3	This class depends on whole Order and PaymentTransaction objects, even though only specific fields (e.g., order.getId(), order.getStatus()) are // used.	Duong_20226034	PayOrderService, Order and PaymentTransaction	Suggestion: In future, pass only necessary fields (e.g., orderId, totalAmount) to reduce coupling to Data level.

4	This class depends on the entire Order object, even though only specific fields (e.g., id, content) may be relevant for payment tracking.	Duong_20226034	PaymentTransaction and Order	In future, pass only necessary fields (e.g., orderId, totalAmount) to reduce coupling to Data level.
6	Low cohesion (logical)	Huyen_20220073	ProductService — methods handle unrelated subtasks (CRUD, mapping, validation)	Extract DTO mapping into ProductDTOMapper, validation into ProductValidator, and type-specific logic into BookService, CDService, etc.
7	Duplicate code	Huyen_20220073	ProductService — create/update/delete logic for each product type	Encapsulate logic for each type; reduce duplication by delegating to shared interfaces and per-type services.
8	Code duplication risk	Huyen_20220073	convertToDTO() in ProductService	Extract to ProductDTOMapper; optionally use per-type mappers for BookDTOMapper, CDDTOMapper, etc.
9	Open-closed principle risk	Huyen_20220073	ProductService — adding new product types requires editing multiple methods	Replace if-else category logic with polymorphism; use MediaProductFactory

				to construct domain entities dynamically.
10	Control Coupling	Huyen_20220073	`OrderService` depends on 7 repositories injected directly	Introduce **Facade Services** (e.g., `InventoryService`, `PaymentService`) to decouple `OrderService` from direct repository access..
11	SRP Violation	Huyen_20220073	Whole `OrderService` class	Split into multiple services: `OrderCreationService`, `DeliveryService`, `PaymentService`, `InvoiceService` etc.
12	Tight Coupling between DTO and Repository	Huyen_20220073	Method `convertToDTO()` inside `OrderService`	Move DTO mapping logic to a separate class (e.g., `OrderMapper`) to isolate DTO transformation logic.
13	Low Cohesion in createOrderFromCart()	Huyen_20220073	`createOrderFromCart()` method	Break down into multiple private methods or delegate to respective services.
14	Violation of Dependency	Huyen_20220073	Constructor-based injection of all	Program to interfaces; inject facade interfaces

	Inversion Principle (DIP)		repositories in `OrderService`	instead of concrete repository classes.
15	Potential ISP Violation (future case)	Huyen_20220073	All repository interfaces (if bloated)	Split repository interfaces if they grow too large with unrelated methods.
16	Control Coupling	Thao_20226001	PlaceOrderService (The method checkRushOrder(DeliveryProductDTO[], DeliveryInfo) passes both data and control logic to a sub-method (checkAddressForRushOrder), which increases control coupling between the execution flow and the input data.)	Separate the RushOrderService class to handle rush delivery condition checking, instead of passing DeliveryProductDTO[] and DeliveryInfo directly into the method.
17	SRP Violation – PlaceOrderService handles too many responsibilities (order creation, calculation, persistence)	Thao_20226001	PlaceOrderService	Split into multiple services: e.g. OrderCalculationService, OrderPersistenceService, RushOrderChecker, etc.
18	OCP violated: payment logic is hardcoded, not extendable (suggest strategy pattern)	Duong_20226034	PayOrderService	Suggestion: Extract a PaymentStrategy or PaymentProcessor interface to support multiple payment methods

19	DIP violated: depends directly on concrete classes; should rely on interfaces	Duong_20226034	PayOrderService	Suggestion: Depend on interfaces (e.g., IOrderRepository, IPaymentTransactionRepository) to decouple high-level business logic
20	DIP – Violated: This entity class depends directly on `Users` (another entity), which is acceptable for the data model layer. However, if business logic becomes more complex, consider using services for decision-making logic instead of embedding it here.	Duong_20226034	Order	Suggestion: Move logic such as `checkOrderStatus` or `changeRejectOrder` to a service class (e.g., `OrderStatusService`) to depend on abstractions and keep the model purely as a data holder.
21	DIP-Violated: Currently, this class depends directly on the `Order` entity through `@OneToOne`, which is normal in JPA for entity relationships. However, to respect DIP better in the service layer, avoid relying on full `PaymentTransaction` objects when	Duong_20226034	PaymentTransaction	Suggestion: Services can depend on abstraction (e.g., DTOs or interfaces) or pass only necessary fields.

	only parts (e.g., `content`, `datetime`) are needed.			
22	Multiple Responsibilities (SRP Violation)	Quyen_20226063	PlaceRushOrderService	Apply SRP by extracting each responsibility (eligibility logic, DTO building) into its own class.
23	**Tight coupling** to type-specific repositories (BookRepository, CDRepository, etc.)	Manh_20225984	ProductService	This is **content coupling**. Replace with a ProductHandler interface and inject handler implementations via **Factory Pattern**. Applies **DIP** (Dependency Inversion).
24	**Content coupling** due to convertToDTO() accessing subtype repositories directly	Manh_20225984	ProductService.convertToDTO	Apply **Open-Closed Principle (OCP)**: Extract per-type mappers (e.g., BookMapper) and apply **Strategy Pattern** for runtime dispatch based on product category.
25	**Control coupling**: GlobalExceptionHandler directly builds response structures	Manh_20225984	GlobalExceptionHandler	Move response construction to ExceptionResponseFactory. Resolves **SRP** violation and

				decouples construction logic.
26	**Content coupling** to product/user validation logic	Manh_20225984	CartService.addToCart	Break into InventoryService and UserValidator; inject via interface. Resolves **DIP** and **SRP** violations.
27	**Control coupling**: role-dependent logic may require conditionals for new roles	Manh_20225984	AuthService.register	Violates **OCP**. Use **Strategy Pattern** with RegistrationHandler per role type to support extensibility without modification.
28	**Content coupling**: all exception logic embedded in one class	Manh_20225984	GlobalExceptionHandler	Split into modular exception handler classes (e.g., OrderExceptionHandler). Use abstract base handler if shared formatting is needed. Resolves **SRP** and **OCP**.

5.4 Design Patterns

<Do you use any design patterns for your design? If yes, describe detailly why you use those design patterns? Describe in detail on the solutions and how to implement each design pattern>