Swinburne University of Technology

School of Software and Electrical Engineering

ASSIGNMENT AND PROJECT COVER SHEET

Subject Code: SWE30003 Unit Title: Software Architectures and Design

Assignment number and title: 2, Object Design Due date: 11:59pm, 5th May 2024

Tutorial Day and time: Wednesday 1PM Project Group: 9

Tutor: Dr. Le Minh Duc

To be completed as this is a group assignment
We declare that this is a group assignment and that no part of this submission has been copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part been written for us by another person.

ID Number	Name	Signature
104850254	Anh-Quan Nguyen	Quan
104850021	Minh-Hieu Tran	Hieu
104311214	Duc-Tam Nguyen	Tam

Marker's comments:

Total Mark:_____

Extension certification:

This assignment has been given an extension and is now due on _____

Signature of Convener:_____

Swinburne University of Technology

School of Software and Electrical Engineering

SWE30003: Software Architectures and Design

Assignment 2: Object Design

Project Group: 9
Tutor: Le Minh Duc

Contents

Executive summary

This document presents a detailed object-oriented design for the Long Chau Pharmacy Management System (LC-PMS). Long Chau Pharmacy is a leading pharmacy chain in Vietnam facing operational challenges due to disconnected and manual processes that hinder its growth and impact customer satisfaction. The current system suffers from inefficiencies in order processing, inventory management, and prescription handling, leading to slow service and frequent product unavailability.

The proposed solution in this report outlines a robust, scalable, and centralized electronic system designed to address these critical pain points. By leveraging Responsibility-Driven Design, this document specifies the system's core components, including a full set of candidate classes, their responsibilities, and their collaborations. It further details the application of established design patterns and heuristics to ensure a high-quality, maintainable, and efficient architecture. This design serves as the blueprint for the subsequent development phases, aiming to enhance operational efficiency, improve customer experience, and support Long Chau's strategic vision.

1 Introduction

This Object Design Document provides a comprehensive software design for the Long Chau Pharmacy Management System (LC-PMS). The system is being developed for Long Chau Pharmacy, a subsidiary of FPT Retail, which operates over 2,000 stores across Vietnam. This document is the direct successor to the Software Requirements Specification (SRS) from Assignment 1 and translates those requirements into a concrete architectural design. It will serve as a foundational guide for developers, project managers, and stakeholders during the implementation and testing phases of the project.

1.1 Outlook of solution

The proposed design is for a centralized, cloud-based application accessible through web browsers and dedicated mobile apps for customers, and a secure web portal for internal staff. The solution architecture assumes it will be built upon a central database that synchronizes data in real-time across all pharmacy branches. This approach ensures data consistency for critical entities such as inventory, orders, and customer records, directly addressing the core problem of fragmented systems. The design is modular, allowing for future scalability and integration with third-party services as outlined in the project scope.

1.2 Definitions, acronyms, and abbreviations

Term	Abbreviation/Definition
LC-PMS	Long Chau Pharmacy Management System
CRUD	Create, Read, Update, Delete
SRS	Software Requirements Specification
CRC	Class-Responsibility-Collaborator
UML	Unified Modeling Language
Heuristics	Guidelines or techniques used to guide system design efficiently
Bootstrap	The process of initializing system components and relationships
Branch	A physical retail location of Long Chau Pharmacy.
Order	A customer's request to purchase one or more products, placed online, via app, or in-store
Prescription	An official instruction from a qualified healthcare professional authorizing a pharmacist to dispense a specific medicinal product
Inventory	The complete list of goods stocked by Long Chau Pharmacy, including quantities and locations
Cashier	A staff member primarily responsible for processing customer payments, handling transactions, and issuing receipts at the point of sale
Pharmacist	A healthcare professional, licensed to dispense and validate prescriptions, offers drug information and patient counseling
Warehouse personnel	A staff member that manages stock in central storage facilities, encompassing receiving, storing, and dispatching products to branches

Table 1: Definitions, Acronyms, and Abbreviations

2 Problem analysis

The design presented in this document is derived directly from the problem analysis conducted in the requirements phase (Assignment 1). The central challenge at Long Chau Pharmacy is the operational friction caused by disconnected, manual processes. Key pain points identified include poor inventory visibility leading to stockouts, inefficient and slow prescription validation workflows, and an inconsistent customer experience lacking real-time updates. Our design confronts these issues by proposing a single, unified system that automates workflows and provides a consistent, real-time data source for all actors involved in the pharmacy's operations.

2.1 Simplifications

To ensure the initial design is focused and manageable, several simplifications have been made based on the project's scope and assumptions. These choices reduce complexity without compromising the core functional requirements.

- **Uniform product model:** The system uses a single `Product` class to represent all items for sale, including prescription drugs, over-the-counter medicine, and personal care items. The distinction between these categories is handled by data attributes (e.g., a `requiresPrescription` flag) rather than a complex inheritance hierarchy. This simplifies the model and makes adding new product types easier in the future.

- **Online-only operations:** The design assumes stable internet connectivity at all branches and therefore does not include functionality for an "offline mode." This significantly simplifies the design by removing the need for complex data synchronization, local caching with conflict resolution, and other mechanisms required to handle network interruptions.
- **Abstracted payment processing:** The internal design does not handle the low-level details of communicating with bank APIs or payment gateways. This complexity is abstracted away by the Strategy design pattern, where a `Payment` class delegates the task to a `IPayment` interface. This simplifies the core business logic and makes the system independent of any specific payment provider.
- **Integration with third-party services:** It is stated in the requirements that the system can be integrated with third-party services (e.g. logistic partner, digital health platforms), however this will be excluded from the design.
- **Administrative tasks:** The management of the master product catalog (adding new products, retiring old ones) and the setup of new branch locations are administrative tasks handled outside the scope of the daily operational workflows presented in this design.

2.2 Assumptions

The following assumptions establish the specific business rules, technical constraints, and operational scope for the design of the Long Chau Pharmacy Management System (LC-PMS). These rules serve as a definitive guide for the system's object-oriented design.

A1 The system manages the sale of pharmaceutical products, health supplements, medical devices, and personal care items.

A2 Each customer order is treated as a single, unique transaction. It must be associated with exactly one customer and one final payment and must generate a single, corresponding receipt.

A3 A customer must have a registered account to purchase prescription medication. Over-the-counter products may be purchased by unregistered customers (in-store).

A4 A digital prescription must be validated by a licensed pharmacist through the system before the associated medication can be sold or dispensed.

A5 A validated prescription is considered "used" after its associated order is completed and cannot be reused for subsequent orders.

A6 An order may contain multiple products, including a mix of prescription and non-prescription items.

A7 The system must support multiple payment methods (e.g. cash, credit card, e-wallet), but an order must be paid for in full via a single transaction using one of these methods. Split payments are not supported.

A8 All branches are assumed to have reliable internet connectivity to the central system to support real-time inventory synchronization and order processing.

A9 Staff members are assumed to possess basic digital literacy and will be trained to use the LC-PMS as part of their operational duties.

A10 Customer and staff data are considered permanent records. Once created, these records cannot be hard-deleted from the system, only marked as 'inactive' or 'archived' for data integrity and auditing purposes.

A11 The system is designed to handle a peak load of approximately 200 customer transactions per branch per day.

A12 It is assumed that the LC-PMS operates within Vietnam, and complies with Vietnamese data protection, healthcare and tax regulations.

3 Candidate classes

The classes for the LC-PMS were identified by applying the principles of Responsibility-Driven Design to the requirements specified in Assignment 1. The following list is categorized to clarify the role of each class within the overall architecture.

3.1 Candidate class list

- **Domain entities:**
  - `Customer`
  - `Staff` (and its subclasses: `Pharmacist`, `Cashier`, BranchManager, WarehousePersonnel.)
  - `Branch`
  - `Product`
  - `Inventory`
  - `Prescription`
  - `Order`
  - `OrderItem`
  - `Payment`
  - `Receipt`
  - `Notification`
- **External service interfaces:**
  - `IPayment`
  - `IDelivery`
  - `INotifications`
  - `IReport`

3.2 UML class diagram

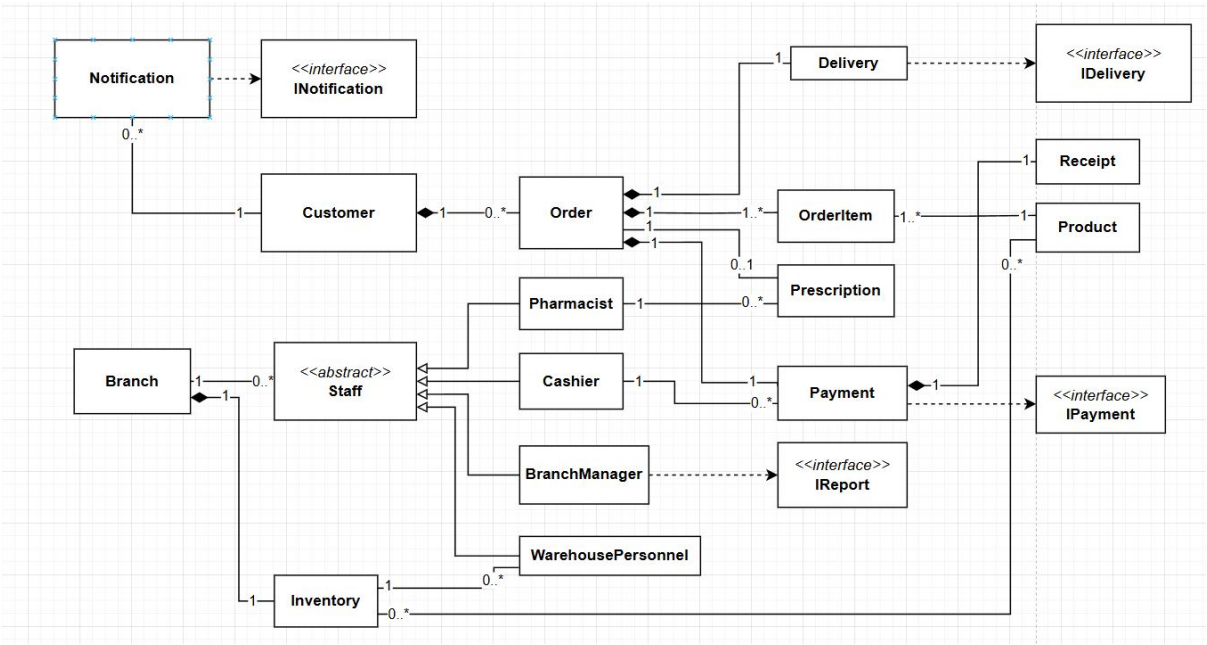

Figure 1: UML class diagram for LC-PMS.

3.3 Justification of design choices

Domain entities justification These classes represent the core concepts of the pharmacy domain. Following RDD, they are designed as "smart" objects, responsible for knowing their own data and managing their own state and fundamental behaviors.

- `Customer`**:** These classes model the primary actors. Separating them (rather than using a generic `User`) provides a clear distinction between external clients and internal employees, each with vastly different responsibilities, data, and permissions.
- **The staff hierarchy:** An abstract `Staff` class is used as a base to establish a clear "is-kind-of" relationship for all employee types. This is justified because it allows for shared attributes (like employee ID and assigned branch) to be defined once. The concrete subclasses are justified by their unique, specialized responsibilities:
  - `Pharmacist`**:** Justified by the need to encapsulate the legally distinct and critical responsibility of prescription validation.
  - `Cashier`**:** Justified by its focused role in handling payments.
  - `BranchManager`**:** Justified by its unique responsibility to oversee a branch and initiate high-level actions like report generation.
  - `WarehousePersonnel`**:** Justified by its specialized role in managing the physical stock within an inventory and coordinates the delivery.
- `Order` **and** `OrderItem`**:** An `Order` encapsulates a single customer transaction. The introduction of the `OrderItem` class is a crucial design choice. It properly models the many-to-many relationship between `Order` and `Product`, allowing an order to contain multiple products with specific quantities for each, which is a more robust and scalable design.

- `Payment` **and** `Receipt`**:** These classes cleanly separate the financial transaction from the order itself. `Payment` is responsible for the complex act of processing payment via different strategies, while `Receipt` is the resulting proof of that act. This separation keeps the `Order` class from being bloated with financial details.
- `Branch`**:** This class acts as the structural anchor for location-specific operations. As a container for its own `Inventory` and assigned `Staff`, it is fundamental to managing a multi-location business and enabling features like branch-specific reporting and stock checking.
- `Product`**:** This class is the digital representation of an item in the pharmacy's catalog. By encapsulating all product-specific data (name, price, prescription requirement), it stores all necessary information in one place, helping keep sales and inventory operations consistent.
- `Inventory`**:** This class is given the critical and focused responsibility of managing stock levels for products on a per-branch basis. Separating this from the `Product` class is essential for tracking real-time availability and preventing stockouts, a core business requirement.
- `Prescription`**:** This class is needed because medical prescriptions have a unique process and legal importance. It keeps track of key details and status changes (like pending, validated, rejected, spent), separating this complex part from the normal order process.
- `Notification`**:** This class encapsulates the concept of a message to be sent. This separates the content and recipient of a message from the mechanism used to send it (which is handled by the `INotification` interface), leading to a more flexible notification system.

External service interfaces justification To ensure the system is extensible, interfaces are used to define contracts for interacting with external or interchangeable components.

- `IPayment`**:** This interface defines a contract for how payments are processed. This allows the system to easily support new payment methods (e.g., a new e-wallet) in the future by simply creating a new class that implements this interface, without changing the `Payment` or `Order` classes.
- `IDelivery`**:** This interface encapsulates the strategy for fulfilling customer orders via different channels, such as in-store pickup or in-house courier. It allows seamless integration with new delivery partners or custom routing logic, improving modularity in the `Delivery` class.
- `INotification`**:** This interface defines how messages are sent through various channels like SMS, email, or in-app alerts. By implementing different notification strategies (e.g., `EmailNotification`, `SMSNotification`), the system can flexibly adapt to customer preferences or expand to support future communication platforms.
- `IReport`**:** This interface defines how reports are generated, formatted, and exported (e.g., PDF, CSV). It separates complex data querying logic from presentation, allowing new reporting types to be added (e.g. demand forecasting, audit reports) without altering the `BranchManager` or `Branch` classes.

Justification for discarded classes During the object design process, several candidate classes were initially identified based on system requirements and use case responsibilities. However, not all of these classes were retained in the final design. Below here is the rationale for

discarding specific classes that were deemed unnecessary, redundant, or misaligned with the principles of object-oriented design. Each decision was made to improve clarity, cohesion, and maintainability of the system architecture.

- `System`: This class was overly generic and acted as a catch-all coordinator, leading to high coupling and low cohesion. Its responsibilities overlapped with more specialized domain classes (e.g., `WarehousePersonnel`, `BranchManager`), making it redundant. Responsibilities were better handled by delegating to domain-specific classes to support responsibility-driven design.
- `UIController`: This class focused on interface concerns rather than core business logic, which did not align with object-oriented design goals. Including `UIController` would have introduced an unnecessary layer not represented in the domain model.

4 CRC cards

This section details the candidate classes for the LC-PMS using the Class-Responsibility-Collaborator (CRC) card methodology. Each card outlines the class's core purpose, its primary responsibilities, and, crucially, the specific collaborators required to fulfill each individual responsibility, reflecting the final design.

4.1 `Customer` CRC Card

Class Name: `Customer`
Superclass: –

Description: An object representing an individual who interacts with Long Chau Pharmacy to purchase products or use its services. It acts as the primary actor for most customer-facing interactions.

Responsibilities	**Collaborators**
Knows its personal information (e.g., name, address)	–
Place an order with selected products, delivery method	`Order`, `IDelivery`
Selects preferred payment method at checkout	`IPayment`
Receives payment confirmation or receipt	`Payment`, `Receipt`
Uploads and manages prescriptions	`Prescription`
Views past order history	`Order`
Receives notifications from the system	`Notification`

Table 2: `Customer` CRC Card

4.2 `Order` CRC Card

Class Name: `Order`
Superclass: –

Description: Encapsulates all information for a single transaction. It serves as a central hub that links the customer, the items being purchased, and the payment status.

Responsibilities	**Collaborators**
Knows the customer who placed it	`Customer`
Maintains a list of its line items	`OrderItem`
Knows its fulfillment method and current status	–
Associates prescription if required by selected products (if any)	`Prescription`
Calculates its total cost by summing all order items	`OrderItem`
Applies selected delivery method to the order	`IDelivery`
Applies selected payment method and processes payment	`IPayment`, `Payment`
Handles payment result and updates order status accordingly	`Payment`
Triggers delivery process once payment succeeds	`Delivery`
Notifies customer about order updates	`Notification`, `INotification`

Table 3: `Order` CRC Card

4.3 `Prescription` CRC Card

Class Name: `Prescription`
Superclass: –

Description: A digital record of a medical prescription that is uploaded by a customer and must be validated by a licensed pharmacist before the associated medication can be sold or dispensed.

Responsibilities	**Collaborators**
Stores prescription details (e.g., medication, issue date, patient info)	–
Knows its validation status (e.g., pending, approved, rejected)	–
Allows pharmacists to validate or reject with reasons	`Pharmacist`
Records pharmacist ID and comments for audit trail	`Pharmacist`
Associated with a customer's order	`Order`
Notifies customer of validation result	`Notification`, `INotification`

Table 4: `Prescription` CRC Card

4.4 `Branch` CRC Card

Class Name: `Branch`
Superclass: –

Description: A physical retail location of Long Chau Pharmacy. Each branch has its own staff and manages its own local inventory.

Responsibilities	Collaborators
Knows its location details (e.g., address, ID)	–
Knows the staff members assigned to it	`Staff`
Manages its local product stock	`Inventory`

Table 5: `Branch` CRC Card

4.5 `Notification` CRC Card

Class Name: `Notification`
Superclass: –

Description: An automated message sent by the system to a customer or staff member. It provides timely updates on order status, prescription approvals, payment confirmations, and promotional campaigns.

Responsibilities	Collaborators
Knows its content and intended recipient	`Staff, Customer`
Knows its delivery channel (e.g., SMS, email, in-app) and status	–
Assigns appropriate delivery channel	`INotification`
Stores timestamp of when notification was sent	–

Table 6: `Notification` CRC Card

4.6 `OrderItem` CRC Card

Class Name: `OrderItem`
Superclass: –

Description: A class that links a `Product` with an `Order` and specifies the quantity purchased. It resolves the many-to-many relationship between orders and products.

Responsibilities	Collaborators
Knows the specific product being ordered	`Product`
Knows the quantity being ordered	–
Calculates its own subtotal (price x quantity)	`Product`

Table 7: `OrderItem` CRC Card

4.7 `Staff` (Abstract) CRC Card

Class Name: `Staff`
Superclass: –

Description: An abstract base class that models the common behaviors and information for all employees. It defines a common contract for specialized staff roles.

Responsibilities	Collaborators
Knows common employee information (e.g., ID, name)	–
Knows the branch to which it is assigned	`Branch`

Table 8: `Staff` CRC Card

4.8 `Pharmacist` CRC Card

Class Name: `Pharmacist`
Superclass: `Staff`

Description: A specialized staff member responsible for dispensing medications and ensuring prescriptions are clinically and legally appropriate.

Responsibilities	Collaborators
Validates the authenticity of a prescription	`Prescription`
Updates the status of a prescription (approved/rejected)	`Prescription`

Table 9: `Pharmacist` CRC Card

4.9 `Cashier` CRC Card

Class Name: `Cashier`
Superclass: `Staff`

Description: A staff member primarily responsible for handling customer payments, processing transactions, and issuing receipts at the point of sale.

Responsibilities	Collaborators
Initiates the payment process for an in-store order	`Payment, Order`
Processes cash transactions	`Payment`
Prints or hands over the receipt after payment	`Receipt`, `Payment`

Table 10: `Cashier` CRC Card

4.10 `BranchManager` CRC Card

Class Name: `BranchManager`
Superclass: `Staff`

Description: A staff member responsible for overseeing the daily operations, staff, and performance of a specific pharmacy branch.

Responsibilities	Collaborators
Initiates the generation of operational reports	`IReport`
Views operational reports (sales report, ...) for their assigned branch	`IReport, Branch, Order, Payment, Inventory`

Responsibilities	Collaborators

Table 11: `BranchManager` CRC Card

4.11 `WarehousePersonnel` CRC Card

Class Name: `WarehousePersonnel`
Superclass: `Staff`

Description: A staff member responsible for managing stock in central storage facilities, including receiving, storing, and dispatching products to branches.

Responsibilities	Collaborators
Updates stock levels for incoming shipments	`Inventory, Product`
Prepare stock for dispatch and notify delivery team	`Delivery, IDelivery`
Coordinates the transfer of stock between branches	`Inventory`

Table 12: `WarehousePersonnel` CRC Card

4.12 `Product` CRC Card

Class Name: `Product`
Superclass: –

Description: Represents any item offered for sale by the pharmacy. It is responsible for knowing its own details and current availability.

Responsibilities	Collaborators
Knows its intrinsic details (e.g. name, price)	–
Determines its current stock level at a given branch	`Inventory`

Table 13: `Product` CRC Card

4.13 `Payment` CRC Card

Class Name: `Payment`
Superclass: –

Description: A class dedicated to handling the financial transaction for an order, using the Strategy pattern for flexibility.

Responsibilities	Collaborators
Knows the total amount to be charged	`Order`
Processes the financial transaction by delegating to a strategy	`IPayment`
Confirms successful transaction to the order	`Order`
Generates a transaction receipt	`Receipt`

Responsibilities	Collaborators
Handles failed transaction retry	`IPayment`

Table 14: `Payment` CRC Card

4.14 `Receipt` CRC Card

Class Name: `Receipt`
Superclass: –

Description: Represents the digital confirmation of a completed transaction.

Responsibilities	Collaborators
Knows the details of the transaction (items, prices, date)	`Order`, `Payment`
Generate a unique receipt ID for traceability	–
Provides receipt data in required format, such as: for viewing, printing, or for national tax reporting	–

Table 15: `Receipt` CRC Card

4.15 `Inventory` CRC Card

Class Name: `Inventory`
Superclass: –

Description: Manages the stock levels of all products for a specific branch.

Responsibilities	Collaborators
Knows the stock quantity for a specific product	`Product`
Increases/decreases stock count for a product	`Product`
Knows the branch it belongs to	`Branch`

Table 16: `Inventory` CRC Card

4.16 `Delivery` CRC Card

Class Name: `Delivery`
Superclass: –

Description: Handles the end-to-end delivery or pickup process for a customer order. It tracks delivery progress, assigns delivery staff, and integrates with real-time notification and tracking features.

Responsibilities	Collaborators
Knows the list of products, delivery method (e.g. home delivery, pickup)	`Order`
Stores delivery address	`Customer`
Updates delivery status in real time	`Order`, `Notification`

Responsibilities	Collaborators

Table 17: `Delivery` CRC Card

4.17 `IPayment` (Interface) CRC Card

Class Name: `IPayment`
Superclass: –

Description: An interface that defines a common contract for all payment processing algorithms, enabling the Strategy Pattern.

Responsibilities	Collaborators
Defines a contract for executing a payment transaction	–
Generate or return a transaction ID	`Payment`
Return payment result (success/failure)	`Payment`, `Order`

Table 18: `IPayment` CRC Card

4.18 `IDelivery` (Interface) CRC Card

Class Name: `IDelivery`
Superclass: –

Description: An interface for handling different order fulfillment methods such as delivery or in-store pickup.

Responsibilities	Collaborators
Define delivery logic	`Delivery`
Return delivery status	`Delivery`

Table 19: `IDelivery` CRC Card

4.19 `INotification` (Interface) CRC Card

Class Name: `INotification`
Superclass: –

Description: Defines a contract for sending notifications through various delivery channels.

Responsibilities	Collaborators
Define method for sending a notification message, spefic to the delivery channel	`Notification`
Define method for return status of delivery (sent, failed, pending)	`Notification`

Table 20: `INotification` CRC Card

4.20 `IReport` (Interface) CRC Card

Class Name: `IReport`
Superclass: –

Description: Defines a contract for generating reports such as sales, inventory, and demand analysis across branches.

Responsibilities	Collaborators
Define methods for filter and aggregate data based on report type	`Order`, `OrderItem`, `Payment`, `Inventory`, `Product`
Define methods for formatting report output	–

Table 21: `IReport` CRC Card

5 Quality of Design

The quality of this object-oriented design is measured by its adherence to fundamental software engineering principles that promote maintainability, robustness, and scalability. The primary goal is to achieve a system with **high cohesion** and **low coupling**. High cohesion ensures that classes like `Order` and `Payment` have a single, well-defined purpose. Low coupling, achieved through patterns like the Facade and Strategy, ensures that changes in one part of the system have minimal impact on others. This directly supports key quality attributes from Assignment 1: high maintainability, greater reliability, and the scalability required for Long Chau's growth.

5.1 Design heuristics

The following design heuristics were applied to guide the class design and responsibility assignment process, ensuring a logical and clean architecture.

ID	Heuristic name	System application
3.1	Distribute system intelligence horizontally.	The specialized `Staff` roles like `Pharmacist` and `Cashier` handle their own domain logic, rather than having one all-powerful controller making every decision.
3.2	Do not create god classes/objects.	The `Order` class delegates responsibilities to other classes like `Payment` and `Delivery`. It focuses on the contents of the order, preventing it from becoming a "god object" that handles every step of the transaction.
2.8	A class should capture one and only one key abstraction.	The design cleanly separates the distinct domain entities: `Order`, `Customer`, `Receipt`. Each class represents a single, focused abstraction.
2.9	Keep related data and behaviour in one place.	The `Inventory` class holds the stock quantity data and also contains the behavior to modify that data (e.g., 'increaseStock').

3.9	Do not turn an operation into a class.	The design correctly uses the exception to this rule for Strategies. The "pay" operation is abstracted by the `IPayment` interface, allowing different payment algorithms to be encapsulated in different classes.
4.1	Minimize the number of classes with which another class collaborates.	The `Product` class has minimal collaborations. It only interacts with `OrderItem` and `Inventory`. It has no knowledge of customers, payments, or staff, reducing its coupling.
4.3	Minimize the amount of collaboration.	The `BranchManager` initiates a report via a single dependency on the `IReport` interface. This represents one high-level request, not a complex series of interactions.
5.12	Avoid explicit case analysis; prefer polymorphism.	The `Payment` class's use of the `IPayment` interface is a prime example. It does not need an 'if/else' block for different payment types; it simply uses the polymorphic behavior of whichever class implements the interface.
5.16	Illegal to override a base method with a no-op in derived class.	Responsibilities are placed only in the relevant subclasses. For instance, the responsibility to validate a prescription belongs only to the `Pharmacist`, not the base `Staff` class, so a `Cashier` does not need to implement a "do nothing" version.
5.5	Keep inheritance hierarchies shallow.	The `Staff` inheritance tree is only one level deep. This keeps the class structure simple, understandable, and easy to maintain.
5.8	Factor commonality high in the inheritance hierarchy.	The association to the `Branch` class is defined once in the abstract `Staff` class, so this common relationship does not have to be repeated in every concrete staff role.
5.6	All abstract classes must be base classes.	The system's only abstract class, `Staff`, serves as the base for the staff role hierarchy.
5.7	All base classes should be abstract classes.	The primary base class in the design, `Staff`, is defined as abstract. This prevents the illogical creation of a generic, non-specialized staff member.
5.1	Inheritance should model specialization hierarchy.	The `Staff` hierarchy correctly models an "is-a-kind-of" relationship. A `Pharmacist` is a specialized type of `Staff` with distinct responsibilities.
5.2	Derived classes know their base class, but not vice versa.	The `Staff` base class has no dependencies pointing back to its subclasses like `Pharmacist` or `Cashier`, ensuring the base class remains generic and decoupled from its derivatives.

Table 22: Application of design heuristics in the LC-PMS

5.2 Design patterns

To address recurring design problems and enhance the system's quality attributes, several well-established design patterns have been incorporated.

Strategy The requirement to support multiple payment methods is addressed using the Strategy pattern. The `Payment` class holds a reference to an `IPayment` interface. Concrete classes (`CreditCardStrategy`, `EWalletStrategy`) implement this interface. This allows new payment methods to be added with no impact on the `Payment` class, making the system highly extensible.

Observer When the state of a prescription or order changes (e.g. a prescription is validated or an order is ready for pickup or delivery), the `Notification` class listens for these events and responds accordingly. This application of the Observer pattern ensures a clear separation between business logic and notification handling, enhancing modularity and maintainability.

6 Bootstrap process

This section illustrates the initialization of classes in LC-PMS system. The bootstrap process follows a logical sequence to initialize core components and prepare the system for operational use.

It is important to clarify the roles of certain participants in the sequence diagrams. The `System` actor is not a domain class but represents the application's entry point or runtime environment; it is the "prime mover" that starts the system and therefore does not have a CRC card.

The process can be summarized as follows:

- The `System` initiates the creation of a new `Branch`.
- Upon branch creation, the `Branch` immediately instantiates its associated `Inventory`.
- The `System` then creates the primary staff members for the branch:
  - `Branch Manager`
  - `Pharmacist`
  - `Cashier`
  - `Warehouse Personnel`
- The `System` proceeds to create `Product` records and adds them to the `Inventory`.
- A new `Customer` is registered by the `System` during the onboarding process.
- After registration, the `Customer` creates (uploads) a `Prescription`, which is validated by the `Pharmacist`.
- If the prescription is successfully validated:
  - A `Notification` is created to inform the customer.
  - The `Customer` will create an `Order`, which includes one or more `OrderItems`.
  - The `Order` creates a `Payment`, and upon success, a `Receipt` is created.
  - Then, `Delivery` is created.
  - Additional `Notifications` are sent to confirm order and delivery details.

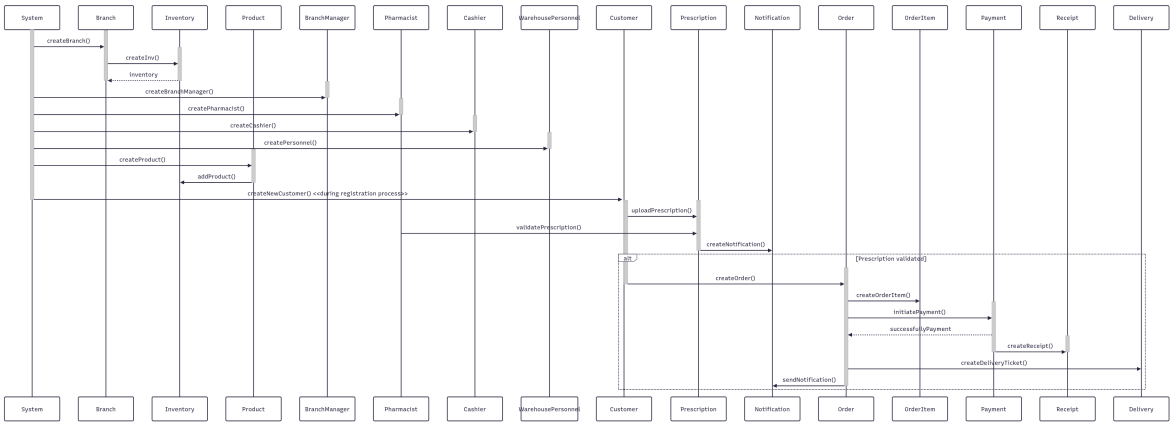

Figure 2: Sequence diagram of the bootstrap process.

7 Verification

To verify the integrity and completeness of the design, this section illustrates how the system's classes collaborate to handle four non-trivial scenarios based on specific domain logic. Each scenario is illustrated with a UML sequence diagram, followed by a description of the interactions. This process validates that the assigned responsibilities are correct and sufficient to meet the system's functional requirements.

7.1 Scenario 1: Customer places order with prescription

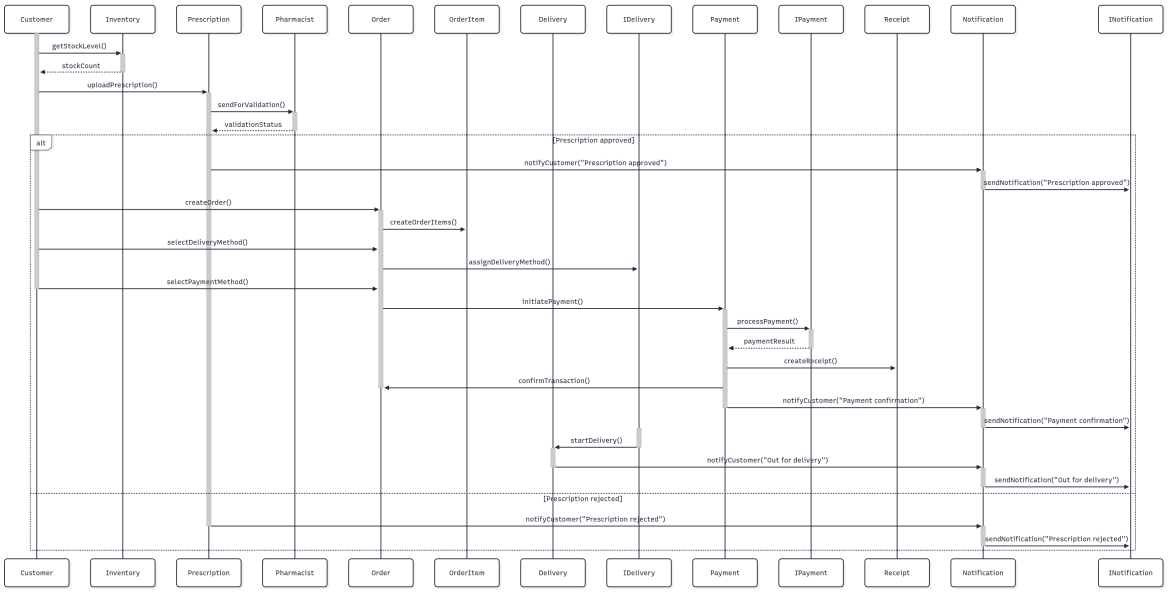

Figure 3: Sequence diagram for an online order with prescription validation.

The interaction begins with the `Customer` querying the `Inventory` to check the stock level of a product. Once the stock is confirmed, the Customer uploads a `Prescription`, which is then sent to the `Pharmacist` for validation. If the prescription is approved, a notification is sent to the `Customer` via the `Notification` component using the `INotification` interface.

The `Customer` then creates an `Order`, which involves generating corresponding `OrderItems`, selecting a delivery method (handled through the `IDelivery` interface), and choosing a payment method. The `Payment` component is triggered to process the transaction through the `IPayment` interface. Upon successful payment, a `Receipt` is generated, and the transaction is confirmed. Notifications are again sent to inform the `Customer` of the payment confirmation. Following this, the delivery process begins as the `IDelivery` interface initiates the `Delivery` component, which then notifies the `Customer` that the order is out for delivery. At each key stage: prescription approval, payment confirmation, and delivery status, notifications are dispatched via the `INotification` interface, ensuring the `Customer` remains informed throughout. If the prescription is rejected, a rejection notification is issued, and the order process is halted.

7.2 Scenario 2: Pharmacist validates a prescription

This scenario verifies the workflow for a specialized staff member, demonstrating how critical, role-specific responsibilities are delegated to the appropriate "expert" object to ensure compliance and security.

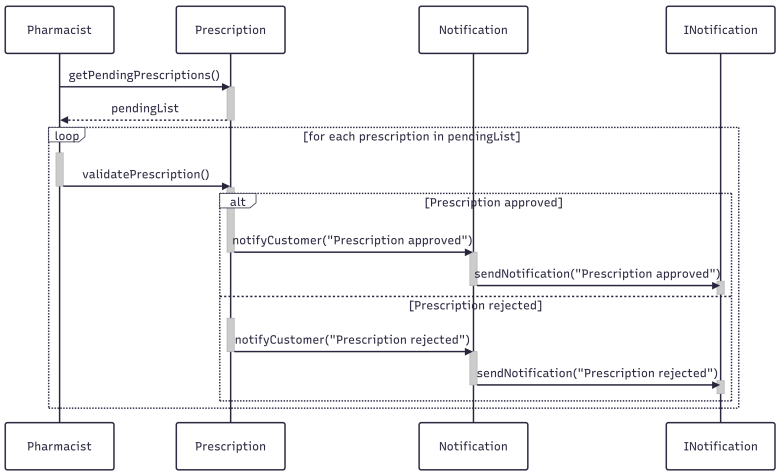

Figure 4: Sequence diagram for prescription validation.

The `Pharmacist` begins by requesting a list of pending prescriptions from the `Prescription` component. Upon receiving the `pendingList`, the `Pharmacist` iterates through each item in the list and calls the validate prescription function on the `Prescription` object. Once a prescription is validated, the `Prescription` component initiates a notification to the customer by interacting with the `Notification` component. This component then utilizes the `INotification` interface to dispatch the message, informing the customer that their prescription has been approved/ declined. This process ensures clear delegation of responsibilities between components, allowing the `Pharmacist` to focus on validation, while the notification system handles communication with the customer.

7.3 Scenario 3: Branch manager generates a sales report

This scenario verifies the system's reporting capabilities, showing how a dedicated controller class is used to handle complex data aggregation tasks, thus keeping the core domain objects clean and focused.

The `BranchManager` initiates the process by requesting a sales report via the `IReport` interface, which is implemented by specific reporting strategies (in this case, sales report). The selected

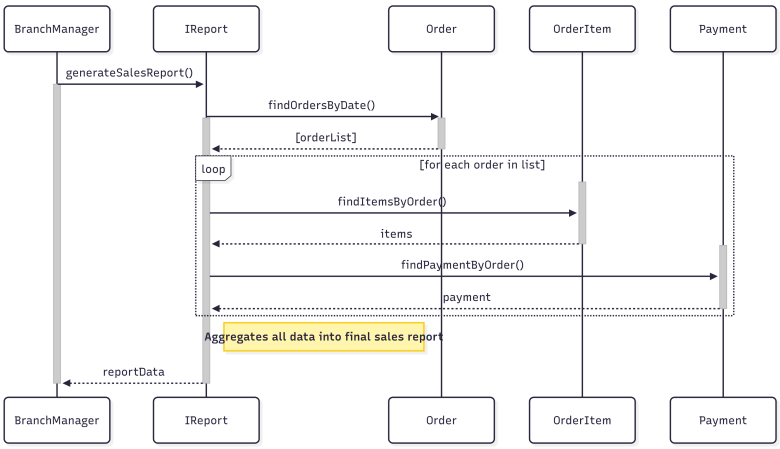

Figure 5: Sequence diagram for generating a sales report.

strategy implementation queries the `Order` component to retrieve a list of orders within a specified date range. After receiving the `orderList`, it iterates through each order to collect detailed data. For every order, the reporting strategy retrieves associated order items from the `OrderItem` component and corresponding payment details from the `Payment` component. Once all relevant data has been gathered, the strategy aggregates this information into a comprehensive sales report. Finally, the compiled `reportData` is returned to the `BranchManager`, enabling them to review sales performance.

7.4 Scenario 4: Staff manages inventory for an incoming shipment

This scenario verifies a critical backend operational task, demonstrating the principle of encapsulation and how the system maintains data integrity by ensuring that only the responsible object can modify its own state.

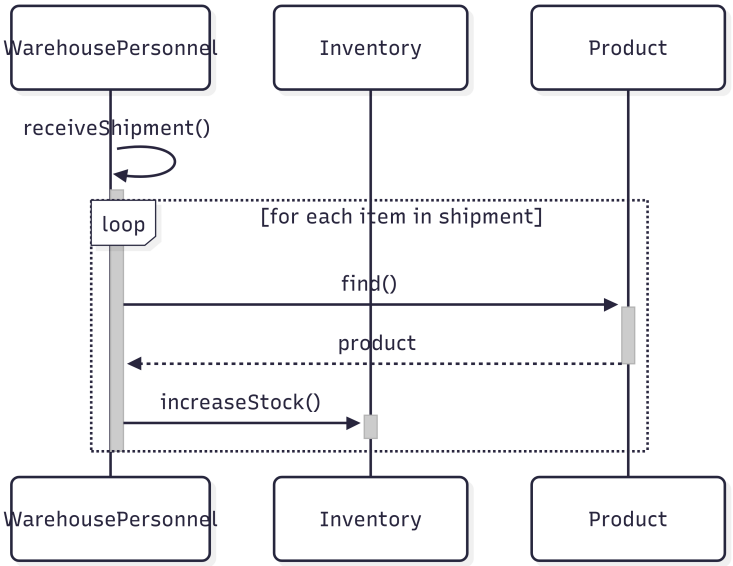

Figure 6: Sequence diagram for updating inventory.

The `WarehousePersonnel` is responsible for handling the inventory update process when a new

shipment arrives. For each item included in the shipment, the `WarehousePersonnel` interacts with the `Product` to retrieve the relevant product information. Once the product is identified and confirmed, the staff proceeds to increase the stock of that product on the `Inventory` class. This interaction ensures that all incoming products are correctly matched and that stock levels are accurately adjusted.

Appendix

A Assignment 1 Submission

This object design document is based on the specifications outlined in Assignment 1. For further details, please refer to the attached file.