Assignment 2 - Object Design

Topic: Online Electronics Store (AWE Electronics - Online)

Executive Summary

The store owners of AWE Electronics have expressed an interest in expanding their operations online. Their current system is operating a local store situated at Glenferrie Road, limiting their reach to nearby suburbs encompassing the area. For this reason, the store owners of AWE Electronics have contacted Swinsoft Consulting to assess and develop the specifications required for a new Online Electronics Store system that will extend their reach to customers Australia-wide.

This report contains a deep analysis of the background problem which was used to develop a system design that describes what the new system will contain. The proposed solution that this report entails will feature an online platform system built on top of a database for the storage of information. This report also contains candidate class specification, a discussion into design quality and patterns, an overview of the bootstrap process as well as verification use cases for the design.

Table of Contents

Introduction

This Object Design Document presents the findings of the problem analysis conducted to identify the necessary classes, interfaces and guidelines that are required to build the system. Specifically, this document is intended to provide a comprehensive reference for developers, managers and stakeholders involved in the system's design and architecture, while also providing a means of exchanging ideas and future testing.

This structured approach ensures a comprehensive understanding of the system's design, facilitating effective development and future scalability.

Outlook of solution

The proposed solution is based on the assumption that it will be built on top of a database, keeping track of various details.

When the platform is hosted and running, there will also be database connection initiated. As each object will have actions that require a connection to the database, this connection will be instantiated, with the details of the connection passed in.

Trade-offs and object design

- **Naming Convention**

Described in **Documentation, and guidelines for interface**

- **Boundary Cases**

For any text entry field inside the User Interface (UI), there should be a fixed allowed length to create a boundary.

- **Invalid Username or Password Exception**

When asked, the user, customer or owner, enters the user name and password into the text field and selects "Sign In". If any of the details are incorrect, an Invalid Username or Password Exception will be thrown by the system.

- **Invalid Data Exception**

This is thrown whenever there data type entered does not match with what is expected inside the field.

Documentation, and guidelines for interface

Identifier	Rules	Examples

Classes	● Class names are to be simple but meaningful.
	● Names are to be capitalised as shown in the examples.
		class OnlineStore
		class Product

Interfaces	Similar to classes rules: ● Interface names are to be simple yet meaningful. ● Names are also to be capitalised.	Interface Payment
Variables	● Variable names will start with a lowercase letter. ● A capital letter will be used to start the next word if required. ● These names are to be simple and meaningful.	Int productCount

Table 1. *Documentation, and guidelines for interface*

Definitions, acronyms, and abbreviation

Online Electronic Store	**OES**
Sales Information System	**SIS**
Database	**DB**
Workflow	**WF**
Data Model	**DM**
User Interface	**UI**

Table 2. *Definitions, acronyms, and abbreviation*

Problem Analysis

The Software Requirements Specification document details a requirement analysis which describes the functional and quality requirements, with these being the guide to build an effective solution. Specifically, the analysis has provided a list of functionality that the platform needs to perform to adequately meet the specifications for various use cases.

The key functionalities of the platform include:

- Record details for:
  - Customer
  - Owner
  - Product
  - Order
- Display products that are currently available to be bought.
- Add items to cart.
- List items in cart.
- Amend details for:
  - Customer
  - Owner
  - Product
  - Delivery Information
- Process payment status
- Create Sales Report
- Create an Invoice

Assumptions

- There are no classifications over the products sold in the OES and they will all be considered electronic products in general, organized in alphabetical order.
- All products will have a unique product ID, a name and a description
- For a customer to make a purchase, they must provide their contact (name, address, phone number), delivery (Only within Australia) and payment (Australian bank only, safety guaranteed, no installments) details.
- Each customer will have a unique ID, name, and phone number (Must be an Australian number)
- Once an order has been made, it cannot be cancelled and the information of the order is sent to both the customer who makes the order and the owner. The customer then has 24 hours to fill in and change the delivery address and the additional delivery information before being able to only do that for the additional delivery information.
- There are two types of account which are the Owner Account and the Customer Account. The Customer Account is one-only and set up from the beginning.

- There is no restriction over the number of customer accounts that can be made within the store.
- There will be a daily one hour maintanence period of time, at late night, when the owner can make changes to products without having to worry about having any customer looking for products at that time.
- The sales report is simply the list of products sold and the total value of the products according to the price they are sold in a certain period of time.
- The OES is ready for service as soon as it is completed.

Simplifications

Due to the nature of this project, there were a few simplifications made to accommodate for an easier implementation of the platform, while still maintaining the key functionalities that it is required to have.

- Information for the Customer and the Owner Account will be treated the same way. They are all User Accounts with the difference only in their functionality.
- Because this is a student project and not planned to be implemented in reality, the data verification check will be in the most simple state which is "if it is the right data then you can proceed". For example, when you make the payment, if the bank account number is filled as numbers only then it's all good.

Design Justification

The system is designed with the assumption that it uses a database in the background. The classes defined in this design act as interfaces to that database, allowing them to be reused by other applications without requiring direct knowledge of how the database works.

The structure avoids storing any duplicate data across the classes, which ensures the system follows the third normal form in database design.

Each class is responsible for managing a specific set of data (which will be saved in the database), while other classes are assigned the task of querying that data to generate relevant information based on their own responsibilities.

Discarded Class List

Class Name	Description
CartItem	Represents an individual item added to the shopping cart, including quantity and linked product information
Discount	Represents discount offers or promotional codes that can be applied to shopping carts or orders.
ReceiptGenerator	Responsible for creating digital receipts once an order has been successfully completed.
Review	Allows customers to leave feedback and ratings for purchased products.
NotificationManager	Sends notifications to customers regarding order status updates, promotional offers, and system messages.
InventoryManager	Manages stock levels of products, ensuring accurate tracking and updates after purchases.

Table 3. *Discarded Class List*

The above classes have been removed for the sake of simplification.

Candidate Class List

A preliminary collection of potential classes for the Online Electronics Store has been established in accordance with Responsibility-Driven Design principles. Each proposed class embodies a crucial concept or responsibility pertinent to the system's domain, prioritizing high cohesion, low coupling, and distinct abstraction. This initial class framework serves as the basis for the forthcoming detailed design and implementation stages.

The subsequent table presents the preliminary candidate classes along with the rationale for their selection:

Class Name	Description
OnlineStore	Central coordinator that manages customer interactions, product catalog browsing, order processing, and payment handling.
ProductCatalog	Stores and manages the list of products available for sale, including searching and filtering capabilities
Product	Represents an individual product with associated information such as name, category, and price
ShoppingCart	Temporary holding area where customers can add products before proceeding to checkout.
Customer	Represents the customer using the platform who can make orders for products.
Order	Represents a finalized transaction initiated by a customer, containing details of purchased products, quantities, and total cost.
Delivery	Manages the delivery information to fulfill orders, including the delivery address and the delivery additional information.
Authentication	Oversees login, registration, and authentication of customers to ensure secure access to the system.
Owner	Represents the owner using the platform who can manage the product catalog, the orders and view the sales report.
User	Represents the user using the platform which can be the customer or the owner.
Payment	Manages the processing of payments.
SaleReport	Conducts sales reports based on the orders having been made which can be viewed by the owner.
Database	Allows execution of arbitrary SQL. It knows nothing of the business logic.This call will either be provided by a third party library or be a thin wrapper around the external library.

Table 4. *Candidate Class List*

The selected candidate classes have been chosen to ensure a clear delineation of responsibilities and to closely mirror real-world functions while still having some space for simplification.

Essential business operations, including product browsing, order management, and payment processing are each encapsulated within separate classes to improve modularity and ease of maintenance.

Careful consideration was given to distinguishing between entity classes (such as Product, Customer, and Order) and controller classes (like OnlineStore, Payment, and Authentication) to guarantee an appropriate separation of behavior and data.

This foundational selection establishes a solid basis for developing a flexible, scalable, and resilient system capable of supporting future growth and enhancements in functionality.

UML Diagram

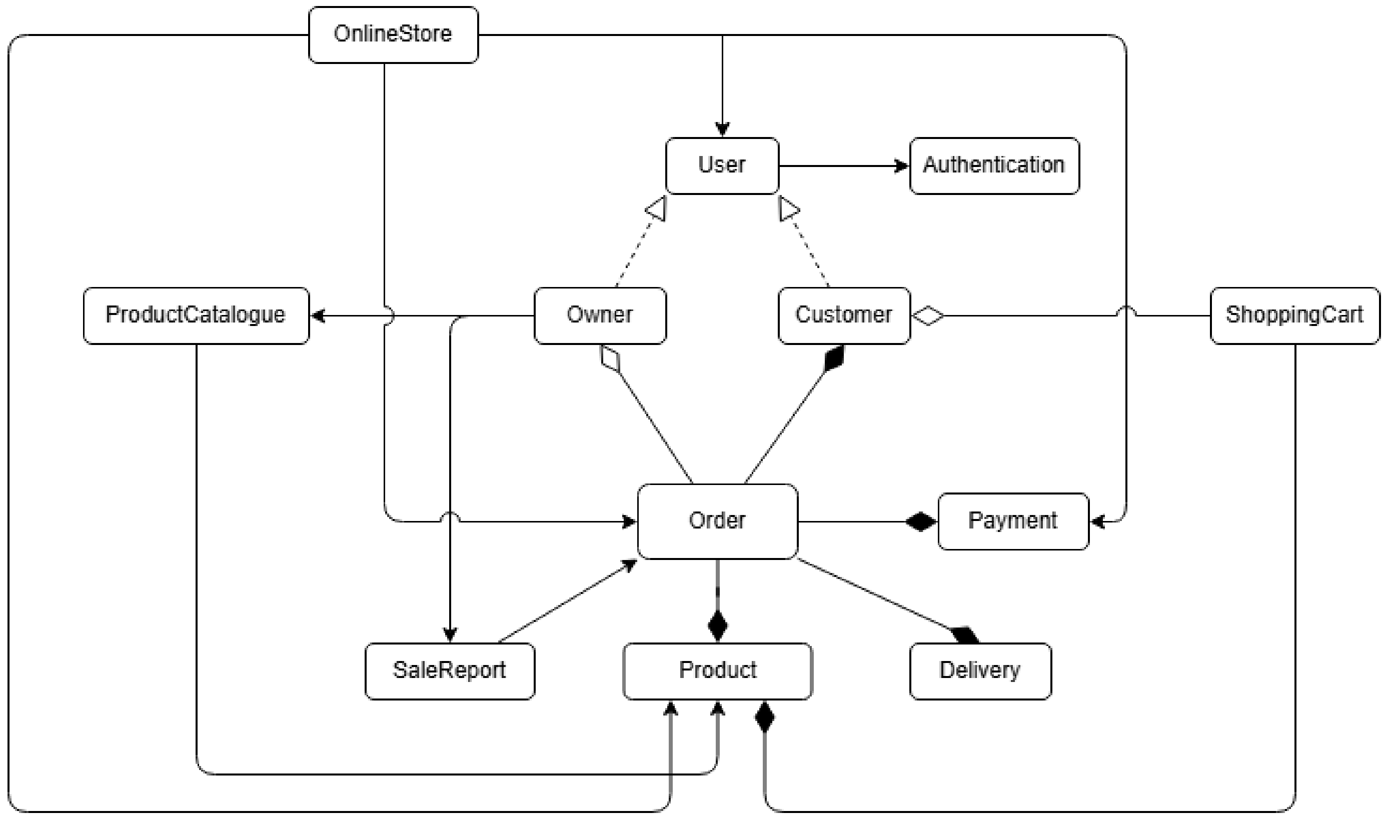

Figure 1. *UML Diagram*

Following a detailed problem analysis, several classes were identified based on the principle of separation of concerns for the development of the SIS. Among them is an essential **OnlineStore** class that initiates the system at runtime. Depending on the user's selected task, this class leads to the creation of one or more relevant objects. The system is built on the premise that an **Order** cannot be created without the presence of a **Customer**, **Product**, **Payment**, and **Delivery** — these are the core elements required for any transaction to proceed.

CRC Cards

Class Name: User **Super Class:** -	
A user knows the commonalities of a real person. However, it is not a guarantee that every instance of a user is a unique real person.	
Responsibilities	**Collaborators**
● Knows name, address and phone number	N/A

Class Name: Customer **Super Class:** User	
A Customer is a unique reference to a Customer, including information that the OES wishes to know about that Customer. *The Customer class is responsible for managing information directly related to themselves.*	
Responsibilities	**Collaborators**
● Knows ID	N/A
● View the products listed in the shopping cart by this customer	ShoppingCart, Product
● View the order status and additional details of this customer	Order
● Find a list of orders of this customer	Order

Class Name: Owner **Super Class:** User	
A Owner is a unique reference to a Owner. *The Owner class is responsible for managing the products catalogue, orders and sales reports.*	
Responsibilities	**Collaborators**
● Knows ID	N/A
● Modify the products shown in the product catalogue	Product, ProductCatalogue
● View and update an order of a Customer	Order, Customer
● View sales reports	SaleReport

Class Name: Order **Super Class: -**	
The Order class is responsible for knowing and managing information about a single Order.	
Responsibilities	**Collaborators**
● Knows ID	N/A
● List the contents in a Order (e.g products)	Product
● Record order details (products, delivery information, payment details)	Customer, Payment, Delivery, Product
● Record a sale when completed	SaleReport

Class Name: Payment **Super Class: -**	
The Payment class is responsible for knowing payment details and handling them of a Customer's order.	
Responsibilities	**Collaborators**
● Knows payment details	Order
● Knows amount due	Order
● Knows payment method	Order

Class Name: Delivery **Super Class: -**	
The Delivery class is responsible for knowing the delivery information of a customer's order.	
Responsibilities	**Collaborators**
● Knows delivery information (e.g address, additional delivery details)	Order

Class Name: Product **Super Class: -**	
The Product class is responsible for knowing information about a single Product.	
Responsibilities	**Collaborators**
● Knows ID	N/A
● Records product details (name and description)	Owner

Class Name: ProductCatalogue **Super Class: -**	
The ProductCatalogue class is responsible for searching, listing and filtering all available products.	
Responsibilities	**Collaborators**
● List available products	Product
● Search available products	Product
● Filters available products (e.g type)	Product

Class Name: SaleReport **Super Class: -**	
The SaleReport class is responsible for recording a completed order, and generating sale reports upon request by the Owner.	
Responsibilities	**Collaborators**
● Record sales details (product sold, quantity, and amount paid)	Order
● Generate sales reports	Owner

Class Name: ShoppingCart **Super Class: -**	
The ShoppingCart class is responsible for recording a list of selected products.	
Responsibilities	**Collaborators**
● Record selected products	Product
● View selected products	Customer, Product

Class Name: Authentication **Super Class: -**	
The Authentication class is responsible for determining access and registration of user accounts.	
Responsibilities	**Collaborators**
● Controls account access	User
● Register new Customer account	User, Customer

Class Name: OnlineStore **Super Class: -**	
The OnlineStore class is the overall control class. It is responsible for user interaction, controlling and the initialisation of other classes.	
Responsibilities	**Collaborators**
● Interacts with the user	N/A (GUI which is not included in this design document).
● Initialises other classes	All classes with the exception of this class.
● Controls how other classes act	N/A

Class Name: Database **Super Class: -**	
The database class is responsible for the execution of arbitrary SQL, which consequently knows the connection details to a database.	
Responsibilities	**Collaborators**
● Knows DB connection details	N/A
● Executes SQL queries	N/A

Design Heuristics

- A class should capture one and only one key abstraction.
- All data unique to a class should be kept hidden inside of that class.
- All data inside a base class should be declared private and not accessed by any other class.
- Employ consistency in language, diagrams, and actions.
- Generated errors will be plain in language and proactively suggest a solution.
- A derived class should not be capable of overriding a base class method that returns a null operation.
- Any ‘superclass’ or a god class will not be created.
- Base classes should be declared abstract.
- Only base classes should be Abstract classes.

Design Patterns

Creational Patterns

Factory Method Design Pattern

This design pattern is specifically aimed at the creation of different types of users, which would be separated by different roles and responsibilities. Through careful analysis of the current problem, it has been identified that there is a requirement of two subclasses. However, as the business expands, there is a high likelihood that additional user objects (IT Support, managers) will be required to manage the platform. The advantage of using such a design pattern is its scalability and flexibility in relation to this context.

Implementation of this design pattern will be done by instead of its default constructor call, there will be a set factory interface method that will do this job instead.

Singleton Method Design Pattern

This design pattern has a unique quirk of ensuring a class has only one instance and provides a method to access it globally. More specifically, this design pattern will be aimed at the database class, being the most applicable implementation.

Implementation of this design pattern will be done by making the database’s default constructor as private (private constructor), barring it from external instance creation attempts.

Behavioural Patterns

State Method Design Pattern

This design pattern is specifically aimed at dealing with the different kinds of states that the Order class has. In an online order, it typically will have several states such as pending, processing, and completed. Given these states, the behaviour of each state is also different in terms of what they become responsible for, making this design pattern a decent applicable implementation.

The implementation of this design pattern will be done through creating the different order states as separate classes (e.g Pending class). By doing so, it allows for a separation of concerns, and each class has a specific responsibility in relation to the state the order is in.

Structural Patterns

Model View Controller (MVC) Design Pattern

This design pattern emphasises the separation of concerns to allow for an easier time maintaining and scaling the application in question. Specifically, it states that an application should consist of three distinct components; model, view, and the controller.
In the context of this project:

- The model is an object in the platform that holds data.
    - Each product object carries unique information about the product.
- The view is the presentation of these models to the user displaying information that is meaningful.
    - The product catalogue provides a view of these products (model) to the user.
- The controller are the actions that are made available to the user based on the view.
    - Actions available to the user after viewing the available products shown in the product catalogue.

Bootstrap Process

The OnlineStore platform follows a staged initialisation process to set up system components in a coherent and controlled sequence. The process begins with the instantiation of essential service-like classes, followed by the dynamic creation of user- and transaction-specific objects as required.

System Initialisation:

1. Main instantiates the OnlineStore class for the interface that interacts with the user.
2. Main initialises the Database class for database connection.
3. Main instantiates the Authentication class to start the registration/logging in process for the two types of users (owner and customer).
4. Main instantiates the ProductCatalogue class to start the product listing process.

Now that the services are instantiated, the bootstrapping process for the other classes would potentially be observed as:

1. The Authentication class initialises the User class (and its subclasses, depending on the credentials or registration).
2. The ProductCatalogue class will read in information from the database and create instances of the Product class to display products.
3. The Customer class will then instantiate the ShoppingCart class, as the customer adds in products (which have been instanced prior by the ProductCatalogue class) to their shopping cart.
4. The ShoppingCart class will (upon check-out) instantiate the Order class to proceed with the purchasing process.
5. The Order class will then instantiate the Payment and Delivery class, providing the payment and delivery details for the order, respectively.
6. As there is now a ‘sale’ that has happened, the Owner class will instantiate the SaleReport class, taking in the sale data and generating a report for analytical viewing.

This process demonstrates a delayed, on-demand instantiation model for most classes, ensuring resource efficiency while maintaining responsiveness. The bootstrapping pattern supports a modular, scalable design with a clear separation of responsibilities.

Verification

The proposed object design has been verified by simulating various use cases as shown below

Customer purchases a product

Scenario Description:
This scenario assumes that the customer has already logged into their account and is searching for a product to purchase of their choosing, in the OES.

Sequence of Events:

1. The customer initiates the purchasing process by browsing the ProductCatalogue which returns products.
2. Upon finding a suitable product, the customer will then add the product into their ShoppingCart.
3. When the customer decides to check-out, the 'Order' class will be responsible for recording the payment and delivery details of the customer.
4. Once all the details have been gathered, a query will be sent to the Database to record the order details and sale.

Sequence Diagram:

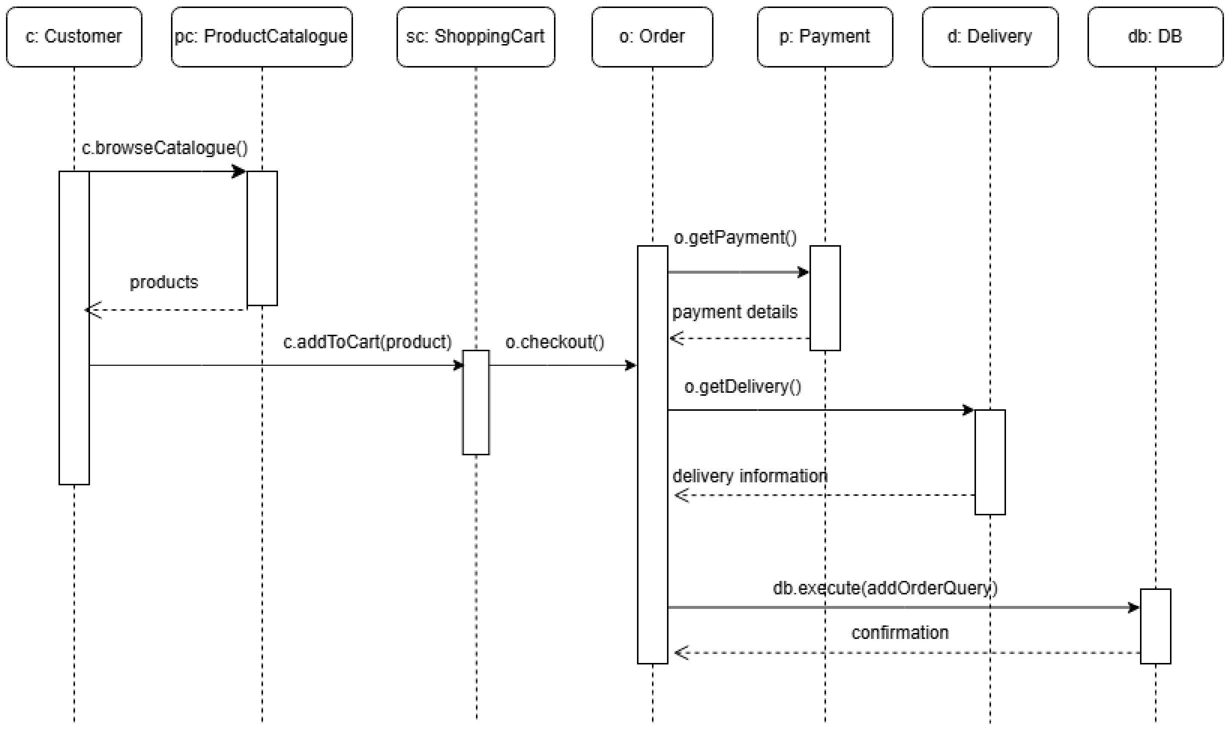

Figure 2. *UML Sequence Diagram representing a customer purchasing a product*

Design Validation:

This example demonstrates the modular representation of how the Customer, ProductCatalogue, ShoppingCart, Order, Payment, Delivery, and DB classes coordinate. It shows how the purchasing process is distributed across clearly defined components that promote separation of responsibilities. The ProductCatalogue serves as the interface for product discovery, while the ShoppingCart temporarily holds the customer's selected items. The Order class orchestrates the finalization of the purchase by interacting with the Payment and Delivery classes to gather all necessary transactional and logistical details. Once compiled, the order is persisted via the DB class, which handles the execution of the database query to record the sale. This design ensures that each component encapsulates its specific role, providing a robust and extensible system for managing customer purchases within the online environment.

Owner generates a sales report

Scenario Description:
Owner initiates to create sales-report in order to see finished orders, and implications for the selling of products of his shop. That allows the Owner to figure out the total income and best-selling product as well as have an ideal and clear sales performance from all of the unit data which stored on the whole system.

Sequence of Events:

1. The Owner generates the sales report by selecting "Generate Sales Report" option via the OnlineStore interface. They can add a period of time for the sales report.
2. SaleReport class will ask the Database for fully fleshed out Order records.
3. After it fetches the information, the SaleReport class aggregates the product info, the grand total units sold, and the grand total revenue within the given period of time.
4. The processed Analytic Report is delivered back to the Owner through the OnlineStore class for demonstration and acceptance.

Sequence Diagram:

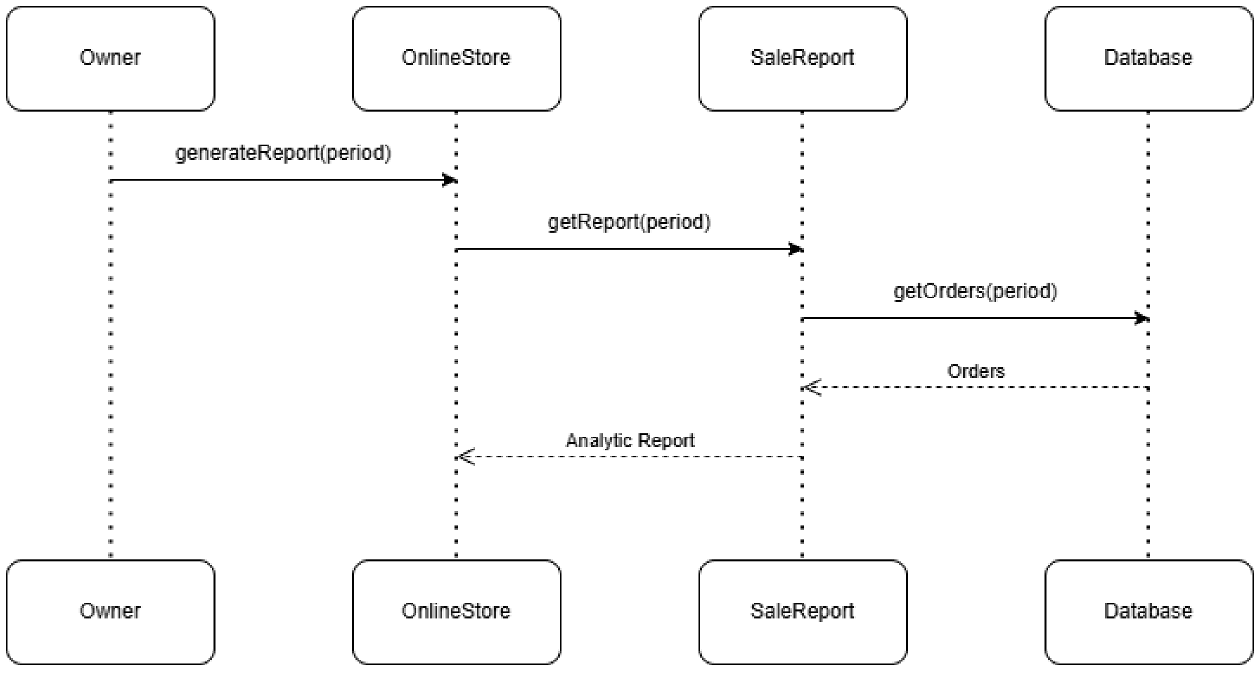

Figure 3. *UML Sequence Diagram representing the owner generating a sales report*

Design Validation:

This example demonstrates the modular representation of how the Owner, OnlineStore, SaleReport, and Database classes coordinate. It shows how the request to generate a sales report follows a clear and logical sequence that supports the principle of separation of concerns. The OnlineStore acts as a façade that bridges the Owner’s request to the system's internal reporting mechanism. The SaleReport class, which encapsulates the business logic, is responsible for retrieving relevant order data from the Database, processing it, and producing an analytic report. The Database class, meanwhile, functions as the persistent data store, supplying complete order records upon request. This design ensures a clean abstraction between user interaction, business analysis, and data persistence while maintaining the flexibility to adjust or expand individual components without disrupting the entire system.

Owner changes available products

Scenario Description:
Owner begins the process of modifying the list of products available for customers. It enables the Owner to maintain up-to-date product information by adding new products, updating existing details, or removing discontinued items from the Product Catalogue.

Sequence of Events:

1. The Owner activates the product customization by choosing the 'Edit Products' from the user interface of the 'OnlineStore' class.
2. The class ‘ProductManager’ that should receive the changes from the Owner and validate the details about the product, like its name, description and price.
3. The lists of batch updates are then validated and delegated upon the 'ProductCatalogue' class to mirror those modifications to the currently held product inventories.
4. Now that the product has been updated with user input, I inject the updated product back into the ‘Database’ class via the ‘store(product)’ operation, making absolutely sure that the changes are persisted and available on the next run of the program.
5. Upon completion, a confirmation is returned to the ‘Owner’, and the refreshed list of products is instantly accessible to customers navigating the store.

Sequence Diagram:

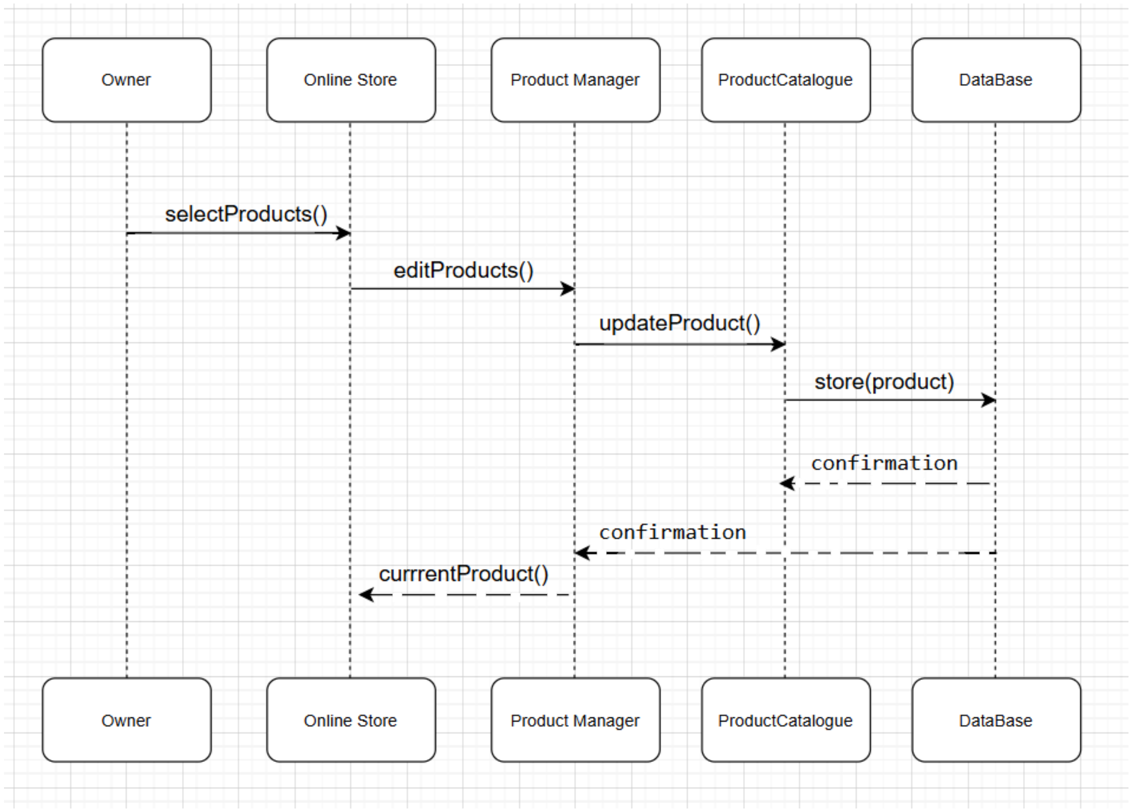

Figure 4. *UML Sequence Diagram representing the owner changing available products*

Design Validation:

This example demonstrates the modular representation of how the Owner, OnlineStore, ProductManager, ProductCatalogue, Database classes coordinate. It shows how modifications the Owner makes are propagated through a well-defined and logical program flow that enforces the separation of concerns. The ProductManager is my business logic layer which receives updates from the Owner and validates them prior to updating the productCatalogue. The Database class stores the modified product information after making changes, and assures data integrity and system health.

Customer registers for an account

Scenario Description:
A new user begins the procedure of creating a customer account to gain access to the purchasing features of the Online Electronics Store.

Sequence of Events:

1. The user initiates the registration process by selecting the 'Register' option available in the user interface of the ‘OnlineStore’ class.
2. The ‘Authentication’ class is responsible for gathering and verifying the user's personal information, such as name, phone number, and address.
3. Upon successful validation, a new ‘User’ object is created and subsequently transformed into a ‘Customer’ instance.
4. The newly established account is saved using the ‘Database’ class, ensuring that the information is securely stored and uniquely identifiable.
5. Once registration is complete, the ‘Customer’ is redirected to the primary Product Catalog, allowing them to begin browsing and engaging with the system

Sequence Diagram:

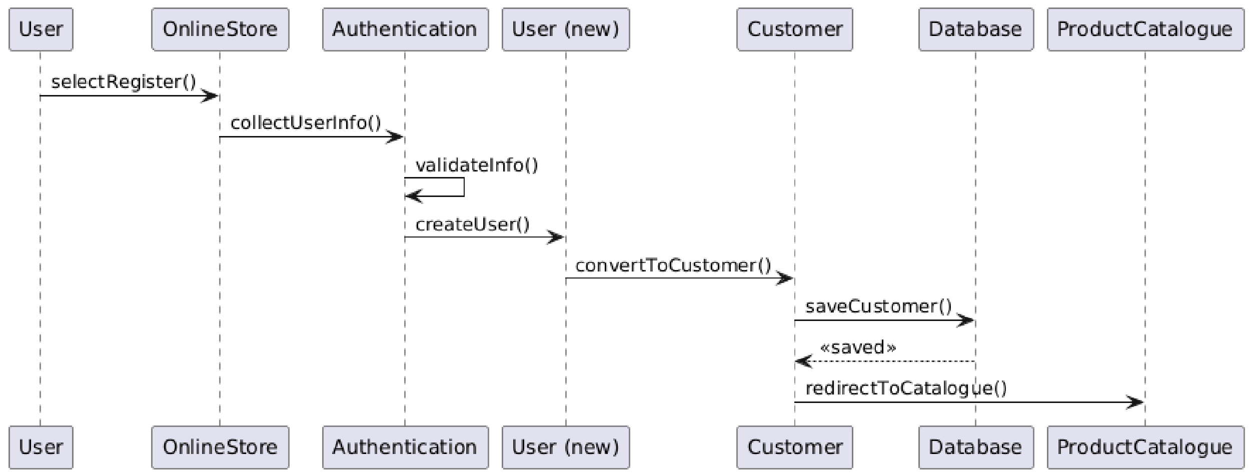

Figure 5. *Sequence diagram representing the registration process for a new customer*

Design Validation:
This scenario confirms the modular interaction among the Authentication, User, Customer, and Database classes. It demonstrates secure account creation, encapsulated user data management, and a seamless transition into the main shopping interface, thereby reinforcing the integrity and functional coherence of the registration process within the system’s overall architecture.

References

[1] V. Huston, "Object-Oriented Design Heuristics", Vincehuston.org, 2019. [Online]. Available: http://www.vincehuston.org/ood/oo_design_heuristics.html/. [Accessed: 23- April- 2025].

[2] "Design Patterns", Refactoring.guru, 2019. [Online]. Available: https://refactoring.guru/design-patterns/. [Accessed: 24- April- 2025].

[3] “Class-Responsibility-Collaboration Card”, GeeksforGeeks, 2024. [Online]. Available: https://www.geeksforgeeks.org/class-responsibility-collaboration-card/. [Accessed: 3-May-2025].

[4] “Software Design Patterns Tutorial”, GeeksforGeeks, 2025. [Online]. Available: https://www.geeksforgeeks.org/software-design-patterns/. [Accessed: 3-May-2025].

[5] “State Design Pattern”, GeeksforGeeks, 2025. [Online]. Available: https://www.geeksforgeeks.org/state-design-pattern/. [Accessed: 3-May-2025].

[6] “Factory Method Design Pattern”, GeeksforGeeks, 2024. [Online]. Available: https://www.geeksforgeeks.org/factory-method-for-designing-pattern/. [Accessed: 3-May-2025].

[7] “Singleton Method Design Pattern”, GeeksforGeeks, 2025. [Online]. Available: https://www.geeksforgeeks.org/singleton-design-pattern/. [Accessed: 3-May-2025].

[8] “MVC Design Pattern”, GeeksforGeeks, 2025. [Online]. Available: https://www.geeksforgeeks.org/mvc-design-pattern/. [Accessed: 3-May-2025].