SWE30003

Software Architectures and Design

Assignment 2 – Object Design

Swinburne University of Technology | Semester 2 – 2025

SWE30003 – Software Architectures and Design

Assignment 2 – Group 6

Team members

Dat Cao - 104497198

Nam Nguyen - 104836937

Hoang Nguyen - 104527639

Thuan Huynh – 105218471

Table of Contents

I. Introduction

1. Executive Summary

This project extends the object design of an online convenience store system that enables customers to browse products, manage a shopping cart, place orders and complete payments securely.

The design focus on modularity, scalability and ease maintenance, by establishing and approaching grounded in key design heuristics and design patterns.

The system models essential business entities with clear relationships representing real-world interactions. Inheritance captures roles specialization; composition ensures proper lifecycle ownership and aggregation allows independent persistence of records.

2. Domain Vocabulary

Users: the system actor with credentials and roles (can be customer or admin).

Customers: people who browse products, manage a cart, place orders and make payments.

Admins: people who manage catalogue content, stock levels, orders and invoices.

Account: the customer's profile and preferences linked to their user identity.

Catalogue: an organized collection of products exposed to customers for browsing and searching.

Product: sellable items with attributes, for instance, name, price, description etc.

SKU (Stock Keeping Unit): unique identifier for a specific product variant.

Inventory item: stock record for a product.

Backorder: order state where demand exceeds current stock.

Order item: line entry referencing a product, unit price, quantity and payment.

Shipment: package and delivery details (carrier, method, tracking).

Carrier: third-party delivery service.

Receipt: proof of successful payment issued to the customer.

Invoice: formal bill of an order (amount due, payments applied, balance).

Authorization: provider approve/holds funds for a specified amount.

Payment Method: strategy interface for executing payments such as DigitalWallet, PayPal, BankDebit, etc.

Tax: jurisdictional charges applied to order totals.

Audit Log: immutable record of key actions (stock changes, order updates, payments).

II. Problem Analysis

1. Assumptions

a) Scope and Actors

⇒ The system is a single online convenience store with two user roles: Customer and Admin, which both specialize User.

⇒ Every Customer has exactly one Account with profile, addresses and preferences. Guest checkout is out of scope, as documented in Simplifications below.

⇒ A customer holds one active ShoppingCart at a time.

b) Catalogue and Products

⇒ Products are discrete on sale items identified by SKU. A Product may appear in one or more catalogues, however, catalogue membership is maintained by the system, where Admins operate.

⇒ Product pricing is tax inclusive or exclusive according to a single store configuration. Price is store per SKU, no dynamic pricing.

c) Inventory

⇒ InventoryItem tracks current stock for a Product at a single store or warehouse.

⇒ Stock is an integer quantity, and updates are atomic to present overselling.

⇒ Backorders is a known domain term, but carts and checkout prevent adding items beyond available stock, therefore on automatic backorder creation.

d) Cart, Orders and Items

⇒ Order is a snapshot of the cart at checkout time and composes its OrderItem. If and Order is removed, then its OrderItems will be removed as well.

⇒ An OrderItem captures Product, unit price at time of purchase, quantity and the line total. Totals are recomputed from OderItems plus tax and shipping.

⇒ A Shipment can cover one or many OrderItems, using ine carrier and one method per shipment.

e) Billing, Invoice and Payments

⇒ An invoice is created when an Order is placed or at admin discretion and records amounts due. It can accept payments.

⇒ Payment success reduces the Invoice balance. After success, a Receipt is created. Therefore, Payment → Receipt is a composition.

⇒ The supported PaymentMethod are DigitalWallet, BankDebit and PayPal, each provides an abstraction to authorize and capture funds.

⇒ Partial payments and multiple payments per invoice are also allowed, as refunds/chargebacks exist conceptually but are administratively handled, as documented in Simplifications below.

f) Security and Compliance

⇒ Users authenticate with email and password. Role-based authorization gates admin functions.

⇒ Audit logging records sensitive actions such as stock adjustments, order state changes and payment outcomes. Logs are immutable once written.

g) Reliability and Operations

⇒ Business riles execute transactionally per use case. For instance, placing orders commits cart snapshot, stock decrement, invoice creation as an atomic unit.

⇒ System clock is authoritative for timestamps.

2. Simplifications

a) Modelling Simplifications

⇒ Only one active cart per customer; no wish lists or saved carts.

⇒ One carrier/method per Shipment. Rate calculation is simplified and out of scope for design details.

⇒ Email delivery is modelled as a single service call.

⇒ Inventory reservation occurs at checkout only, no long-lived holds while browsing.

⇒ Error handling is abstracted as domain outcomes (e.g. Payment authorized or failed). Exception hierarchies and retry policies are not modelled.

⇒ State machines for Order, Payment and Shipment are represented via status fields and transition, not full formal state charts.

b) Out of Scope

⇒ No guest checkout as each purchase requires Account.

⇒ No backorders/pre-orders checkout blocks when stock in insufficient.

⇒ Payments are minimal (DigitalWallet/BankDebit/PayPal only). No refunds, chargebacks workflows.

⇒ Fulfilment simplified with basic shipping rates, no multi-carrier optimization or complex split-shipment logic.

⇒ Customer service flows such as returns, exchanges and cancellations after shipments are not included.

3. Evidence of analysis

a) New Classes Implementation and Update

DigitalWallet, BankDebit and PayPal implementation – This implementation represents real-world payment methods and also provides a modular and easy to extend architecture. Each subclass will have its own method of authorization and behavior.

Database class implementation – this new class will provides a database responsible for information storage and execute SQL query from other classes.

Catalogue class implementation – This class will help the us to organise products into different categories (Price, Purpose,..) and groups such as weekly sale, holiday sales,…This implementation will not only improve the user experience but also improve the cohesion.

b) Added generalisation relationship

With User – Customer, Admin and PaymentMethod, implementing polymorphism concept to these classes to support multiple types of user identity, payment options. These change followed the principles of Stategy Pattern for future technical expansion without huge code modification.

c) Customer – Order relationship

Order changed from composition to association relationship.

This change will captures the weak relationship between the customer and orders to support history record and business purpose. Customers can place many orders, but the orders should not be removed if the customer account is deleted.

d) Admin – Invoice relationship

Invoice from aggregation to association relationship.

Because admin users (staff) do not own invoices, this model will represent the right job between them, admin users have the ability to access to the invoices storage and manage them. This change will remove the unnecessary coupling between admin role and invoice data.

III. Candidate Classes

1. Overview

a) *Candidate Class List*

- User
    - Customer
    - Admin
- Account
- ShoppingCart
- OrderItem
- Shipment
- Product
- Catalogue
- InventoryItem
- Order
- Invoice
- Payment
- Receipt
- PaymentMethod
    - DigitalWallet
    - BankDebit
    - PayPal

b) *UML Class Diagram*

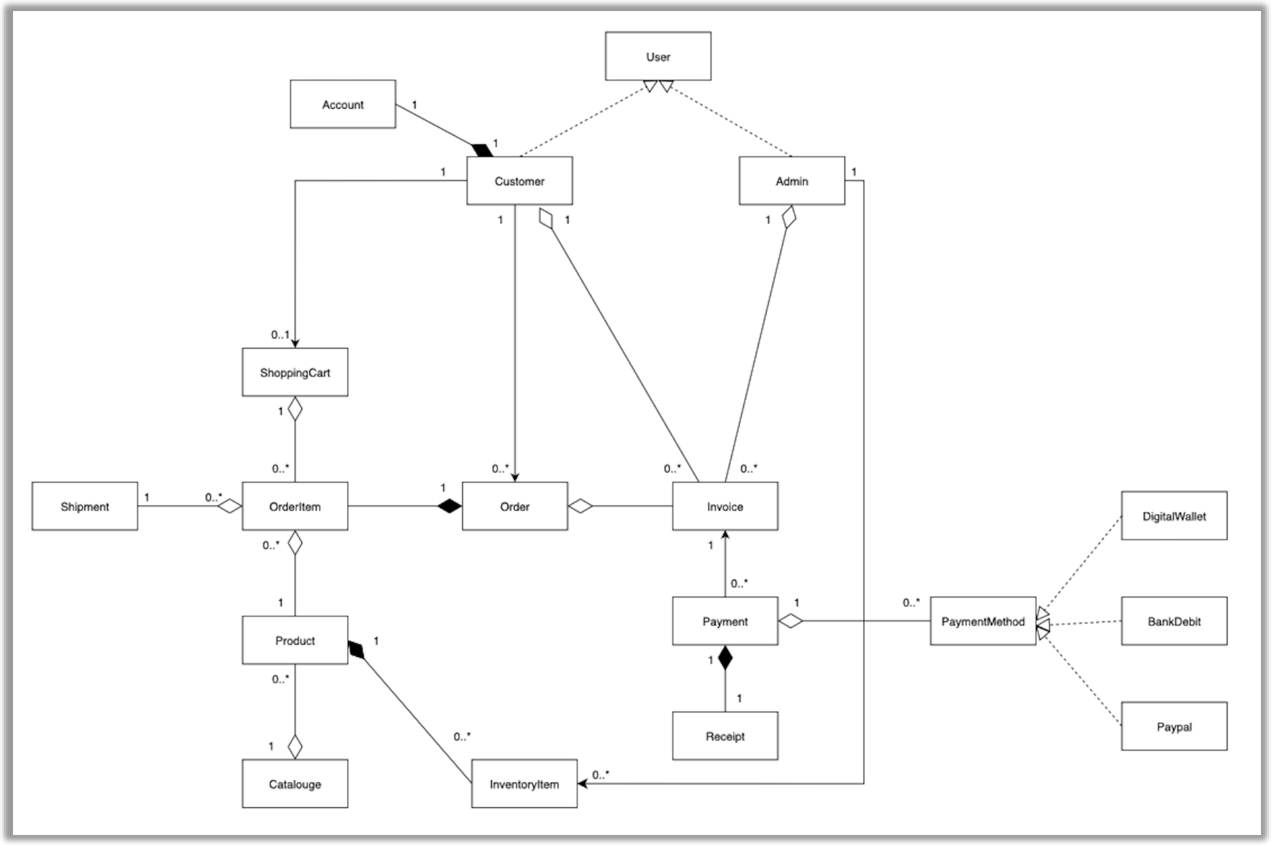

Figure 1 - UML Class Diagram illustration

c) *Design/Class Justification*

This system is designed to model an online convenience store that support users to navigating, ordering and making payment in a modular way and easy for technical expand.

Every classes in the UML diagram represent different business entity. Moreover, the relationships between them illustrate real-world dependencies, their interaction with each others such as ownership, lifecycle especially the customers, orders and payments.

By implementing the inheritance structure between *User* class and *Customer*, *Admin* classes, this help the system captures and define common indentity of the system users and support it in generate different authentication methods. *Customer* extends *User* to represent the buyers who have the ability and access to the UI to browse the products, filter the catalogue, place orders and make payments. *Admin* extends *User* to represent the staff who have access to make changes in the stock level or viewing the invoices.

This structure is also applied to *PaymentMethod*, enabling different concrete payment option (*DigitalWallet*, *PayPal*, *BankDebit*,..) to generate distinct authorisation ways.

In this system, composition relationships appear in many areas, the *Account* exists only as part of the customer, if there is no customer which means that theres is no account is created. The *Order* class is composed from the *OrderItem* that will represent the quantity and price of the products at a purchase time, if an order is cancelled or deleted from the customer, all of its item are removed. This relationship is same as *Product - InventoryItem*, *Payment – Receipt*, the stock records will be removed if there is no products available while the receipt is only generated if a payment is processed successfully.

Aggregation is used to express weaker relationships. Invoices are issued to customers but still get stored in the system support accounting needs, this is same as the *Admin* as well. A shopping cart temporarily contains *OrderItem* before checkout or the *Catalogue* aggregates *Product* so it allow products to appear within multiple catalogues without any strict dependency.

Association relationships describe interactions without ownership. A customer is linked to one active *ShoppingCart* and may places many orders. The *Payment* class can update the balance in the *Invoice*. *Admin* can access to the *InventoryItem* to manage the level of stock.

2. CRC Cards

a) User

Class Name: User **Super Class: -**	
Description: provides the common identity, authentication with authorization and audit handle for all actors. Enables Customer + Admin specialization without duplicating security logic.	
Responsibilites	**Collaborators**
• Knows identity and credentials such as user ID, email and password hash.	N/A
• Knows role and permissions used for access control.	N/A
• Authenticates sign in and maintains session state.	AuthService (built-in class)
• Authorizes actions by checking role/permission.	AuthService (built-in class)
• Exposes audit identity for logging as who did what and when.	AuditLog (built-in class)

b) Customer

Class Name: Customer **Super Class: User**	
Description: represents shoppers that own an Account and Cart, place Orders, receives Invoices/Shipments, and makes Payments. Centralizes buyer's lifecycle.	
Responsibilites	**Collaborators**
• Links to exactly one Account (profile, addresses, preferences).	Account
• Browses catalogue and product details.	Catalogue, Product
• Manages one active ShoppingCart (add, update and remove items).	ShoppingCart, OrderItem
• Places Orders from the cart and reviews order history.	Order
• Views Invoices and initiatives Payments for outstanding balance.	Invoice, Payment, PaymentMethod
• Receives notifications (order, invoice, shipment updates).	DocumentPrinter (built-in class)

c) Admin

Class Name: Admin **Super Class: User**	
Description: models' staff with elevated privileges to curate the Catalogue, manage Inventory, oversee Orders, Invoices, Payments, Shipments. Separated control from customers.	
Responsibilites	**Collaborators**
• Manages catalogue content (create, update and deactivate products)	Catalogue, Product
• Oversees inventory levels and adjustments.	InventoryItem, Product
• Reviews Orders and verifies fulfilment/shipment status.	Order, Shipment
• Issues and oversees Invoices and reconciles Payments.	Invoice, Payment
• Audits sensitive actions and generates reports.	AuditLog (built-in class)

d) Account

Class Name: Account **Super Class: -**	
Description: encapsulates mutable customer profile data such as addresses, saved payments needed at checkout/invoices. Composed with Customer, create or remove with owner.	
Responsibilites	**Collaborators**
• Stores customer profile (name, contact), preferences.	Customer
• Maintains saved addresses (shipping/billing defaults)	Customer
• Holds customer communication preferences (email or SMS)	Customer
• Optionally stores references to saved payment options (tokens and aliases)	PaymentMethod
• Exposes profile data to checkout and invoice generation.	Order, Invoice

e) ShoppingCart

Class Name: ShoppingCart **Super Class: -**	
Description: holds pre-checkout state with items, quantities and variants that are separate from Order. Centralizes validation and price, tax and discount via services. Trigger inventory reservations.	
Responsibilites	**Collaborators**
• Add, update, remove items and validate options.	Product, OrderItem
• Persist and merge carts for users and guests.	Customer, Account
• Calculate totals such as items, tax, shipping and discounts.	PricingService (built-in class)
• Validate stock and cap quantities.	InventoryService (built-in class)
• Prepare cart snapshot for checkouts.	Order

f) OrderItem

Class Name: OrderItem **Super Class: -**	
Description: Purchase entry within an Order that captures the Product reference, unit price at time of purchase, quantity, and line total. Supports validation stock inventory and participates in reservation/commit workflows.	
Responsibilites	**Collaborators**
• Record product details (ID/SKU), unit price at purchase time, quantity, and calculate the line total.	Product
• Validate requested quantity and ensure it does not exceed available stock before checkout or order confirmation.	InventoryItem
• Manage inventory reservations during checkout and release them if the order is cancelled or times out.	InventoryItem, ShoppingCart
• Provide product attributes (weight and dimensions) to assist shipment cost and packaging calculation.	Product, Shipment
• Associate with exactly one Order (composition relationship).	Order

g) Shipment

Class Name: Shipment **Super Class: -**	
Description: Represents packaging and delivery of one or many OrderItems using single carrier and a method per shipment; tracks addresses, costs, and delivery status with a tracking number.	
Responsibilites	**Collaborators**
• Groups OrderItem into a shipment, ensure single carrier + method per shipment.	OrderItem, Order
• Stores shipping address, carrier, method, rate, ETA, and tracking number to indicate the delivery status (e.g., Pending, Packed, Shipped, Delivered, Failed Delivery).	Customer, Carrier
• Calculate shipping cost (using weight/dimensions from Product) and contribute to order totals.	OrderItem, Product, PricingService (built-in class)
• Notify the customers with shipment updates and tracking details.	Customer, NotificationService (built-in class)
• On ship/deliver events, trigger inventory commit/finalization for included items.	Inventory, OrderItem

h) Product

Class Name: Product **Super Class: -**	
Description: captures the sellable item's descriptive data such as SKU, name, pricing and options, independent of stock and transactions.	
Responsibilites	**Collaborators**
• Represent item belongings, for instance, name, price, description, availability.	Catalogue
• Knows price and tax class, provides price for a chosen variant.	PricingService, Tax (both built-in classes)
• Provides packaging/shipping metadata (weight, dimensions)	Shipment
• Exposes active or deactivated status for catalogue and browsing.	Catalogue
• Supplies data to create OrderItem (unit price, product reference)	ShoppingCart, OrderItem
• No stock management, reads availability directly from inventory.	InventoryItem
• Supports admin makes edits to product metadata.	Admin

i) Catalogue

Class Name: Catalogue **Super Class: -**	
Description: aggregates products to power browse, search, filter, sort while keeping Product lean; reflects InventoryItem availability without tight coupling; supports admin and multiple collections/categories.	
Responsibilites	**Collaborators**
• Display product listings, search, filter, sort.	Product, Category
• Provide product details and "Add to cart"	ShoppingCart
• Show availability and paginate results.	InventoryItem
• Support typo-tolerant searching	SearchEngine (built-in class)

j) InventoryItem

Class Name: InventoryItem **Super Class: -**	
Description: keeps stock state separate from Product, manages reserve to prevent overselling; composed with Product; supports admin adjustments, scales multi-location/batch inventory.	
Responsibilites	**Collaborators**
• Knows its Product (SKU/variant) and stock figures: on-hand, reserved, available, reorder level.	Product
• Exposes availability status for browsing/checkout (in-stock, low, out-of-stock, backorder).	Catalogue, Product
• Reserves stock for a Cart/OrderItem and releases on timeout/cancel.	ShoppingCart, OrderItem
• Commits (deducts) reserved stock when the Order is confirmed, packed or shipped.	Order, Shipment
• Supports Admin adjustments (stock-value, returns, replenishment).	Admin
• Notifies Catalogue, UI to refresh displayed stock levels after changes.	Catalogue, UI Controller (built-in class)

k) Order

Class Name: Order **Super Class: -**	
Justification: Represents the confirmed customer purchase at checkout. Cooperate with OrderItem to track the stock, do invoice creation, and payment steps. Tracks pricing/tax totals.	
Responsibilites	**Collaborators**
• Cooperates OrderItem instances, then delete them when the order is cancelled/removed.	OrderItem
• Captures cart snapshot (items, unit prices at purchase time, quantities).	ShoppingCart, OrderItem
• Computes order totals from items + tax + shipping, exposes grand total and balances.	OrderItem, Tax, Shipment, PricingService (built-in class)
• Holds stock commitment on confirmation; release stock on cancellation.	InventoryItem
• Create an Invoice on placement and apply Payment outcomes to reduce balance.	Invoice, Payment
• Manage order status transitions (Placed => Shipped => Delivered)	Shipment
• Provides a unified interface for handling Invoice, Payment, and Shipment processes.	Invoice, Payment, Shipment, Customer

l) Invoice

Class Name: Invoice **Super Class: -**	
Description: Records the sale history, remaining balance of the payment and can be sent to customers and can be viewed by admin.	
Responsibilites	**Collaborators**
• Knows PaymentID.	Payment
• Knows payment amount and date of purchase.	Date
• Knows Customer, Admin, Product.	Customer, Admin, Product
• Determine the remaining balance of order (take the sum of payment).	Payment
• Knows the amount of money that Customer needs to pay.	
• Can create Invoice after order has been placed.	Invoice
• Can send invoice email.	DocumentManagement (built-in class)

m) Payment

Class Name: Payment **Super Class: -**	
Description: The payment class represents a single purchase of one or many products.	
Responsibilites	**Collaborators**
• Knows amount paid	Customer
• Creates Receipt after success payment	Receipt
• Update Invoice balance after success payment	Invoice
• Manages its ownership of Receipt	Receipt

n) Receipt

Class Name: Receipt **Super Class: -**	
Description: Confirms a successful payment with multiple details (amount, time, etc.), can be emailed to customers to serve as a complete transaction.	
Responsibilites	**Collaborators**
• Knows payment details and amount	Payment
• Can store its information in detailed format	
• Can send receipt email	DocumentManagement (built-in class)

o) PaymentMethod

Class Name: PaymentMethod (Abstract) **Super Class: -**	
Description: Serve as an abstract class for capturing different payment methods and authorizing them.	
Responsibilites	**Collaborators**
• Can withdraw an amount of money	

p) DigitalWallet

Class Name: DigitalWallet **Super Class: PaymentMethod**	
Description: represents the transaction via a digital wallet.	
Responsibilites	**Collaborators**
• Knows information of wallet provider.	
• Can perform money transaction from a digital wallet.	

q) BankDebit

Class Name: BankDebit **Super Class: PaymentMethod**	
Description: Represents the direct debit from a customer's bank account.	
Responsibilites	**Collaborators**
• Knows masked bannk account.	
• Can deduct money from bank account.	

r) PayPal

Class Name: PayPal **Super Class: PaymentMethod**	
Description: Represents the authorisation of PayPal and the money transtraction from PayPal.	
Responsibilites	**Collaborators**
• Knows information of PayPal account.	
• Can deduct money from PayPal wallets.	

s) Database

Class Name: Database **Super Class: -**	
Description: Allows data retrieve from other classes and centralise data access.	
Responsibilites	**Collaborators**
• Knows connection information.	
• Execute SQL query.	

VI. Design Quality

1. Design Heuristics

H 5.1: In heritance should be used only to model a specilization hierachy.

H 7.1: Choose the containment relationship.

H 4.1: Minimize the number of classes with which another class collaborates.

H 4.3: Minimize the amount of collaboration.

H 3.2: Do not create god classes/objects.

H 2.8: A class should capture one and only one key abstraction.

H 2.9: Keep related data and behaviour in one place.

2. Design Patterns

a) Creational Patterns

Factory Method Design Pattern

Factory method design pattern is often used in frame work to create a range of differents classes but have a some common abilities and responsibilites. Using Factory Method can reduce the coupling to the concrete class who is the creator, increase the future extension without modifying too much in code base.

Justification:

In our system, Receipt/Invoice, we have a lot of payment method (Card, Cheque,.) and can be expanded in the future (PayPal,..). the Payment structure follows a factory design model, createPaymentMethod() can create a appropriate PaymentMethod at runtime.

Moreover, Shipment can have a lot of departments to carry the job out. A method createShipment() can generates different choices of carriers.

This is also applied to the system user concept: Customer and Admin (Sale Person) are child class of User (Person), this can make ways for future implementation like Contractor or Manager.

<u>**Singleton Method**</u>

Singleton pattern is used for classes that only require a single instance which can give universally access to it.

Justification:

In our system, we define Database to play the role of universal conncetion and store all the information of products.

b) Behaviour Patterns

<u>**Strategy Design Pattern**</u>

Definition:

Define a family of algorithms and make them interchangeable.

Implementation:

This design pattern is applied at payment implementation from the motivation of multiple payment behaviours exist but all of them have a common goal. Our design uses PaymentMethod as abstraction which will provide an interface "withdraw(amount)" can support future concrete implementations like CardDetails or ChequeDetails.

<u>**Observer**</u>

Definition:

Define a one-to-many dependency so that when the state of one object chanes, all the objects that have the dependency are notified automatically.

Implementation:

When admin update the number of stocks in InventoryItem or after a successful Order/Payment event, InventoryItem will notify the Catalouge and UI controller to update the stock level.

c) Structural Patterns

<u>Facade Pattern</u>

Definition:

This pattern provides a simple interface to a complicated system.

Implementation:

Order plays the role of a facade that will collaborate with Invoice, Payment and Shipment during the checkout process. The controller classes or UI controller will only interact with Order which acts as a facade instead of individual components.

<u>Composite</u>

Definition:

Allows us to treate individual objects and compositions equally

Implementation:

Order (whole) and OrderItem (parts) has composition relationship. The OrderItem knows its product and the amount of the products, while Order takes the totals and manage the life cycle of the items in OrderItem.

IV. Bootstrap Process

1. Initialization

At startup, the system performs several initialization tasks to prepare runtime dependencies and ensure a consistent operational state:

1. Database Connection Setup: The Database singleton class establishes a universal connection instance shared across all modules. Connection credentials and parameters are loaded from the configuration file to avoid hard-coding and to ensure secure access.

2. Catalogue Loading: The Catalogue class retrieves the current product listings from the database, creating in-memory instances of Product objects that include name, price, SKU, stock status, and category references.

3. Inventory Synchronization: Each Product instance queries its associated InventoryItem to refresh stock levels, ensuring accurate availability before user interaction.

4. User Session Initialization: When a user logs in, the system instantiates a User subclass (Customer or Admin) and binds it to the session context. If the user is a returning customer, a corresponding ShoppingCart is restored from previous data.

5. Payment Gateway Initialization: The system registers available PaymentMethod subclasses (Paypal, BankDebit, DigitalWallet) into a factory registery, allowing dynamic creation during checkout.

6. Logging and Error Handlers: Global exception handlers and transaction logs are configured to capture runtime events for verification and debugging.

This initialization sequence ensures that all persistent data (catalogues, inventories, and user accounts) and dynamic components (shopping carts, payments) are synchronised before the system begins normal operation.

2. Process

This bootstrapping process reflects a customer placing an order and the system fulfilling it.

- Main starts with the UI controller and connects to the database.
- Customer logs in or first registers; their ShoppingCart is loaded.
- In checkout, an Order is created from the cart snapshot.
- Order creates OrderItem for each product and calculates totals.
- InventoryItem tracks stock for all OrderItem.
- Invoice and Payment are created and processed.
- Upon successful payment, stock is committed, and Shipment is created.
- Shipment calculates cost, assigns tracking, and updates Order status.
- All data (Order, OrderItem, Invoice, Payment, Shipment) are saved.
- If payment fails or the order is cancelled, stock reservations are released.

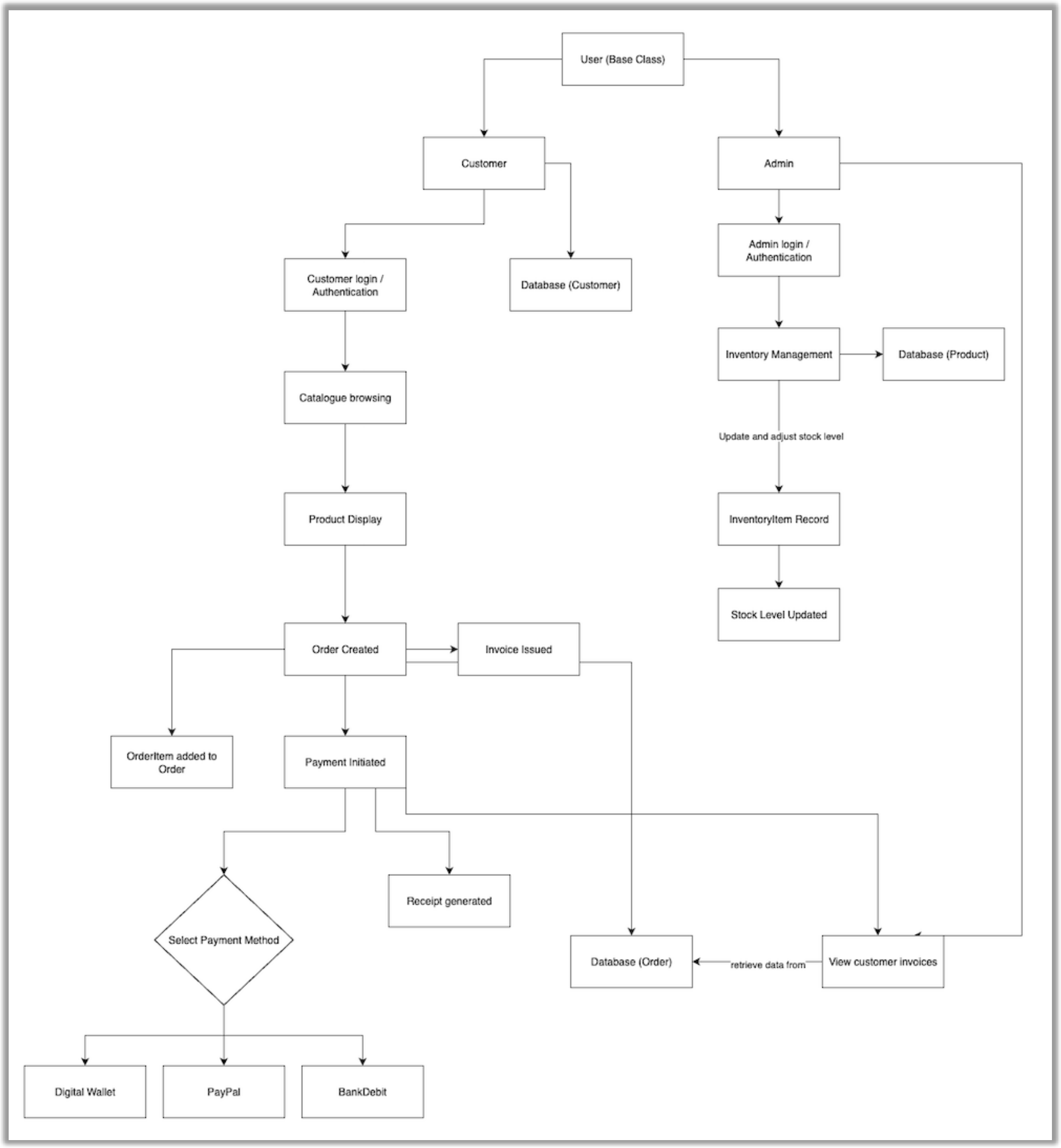

Figure 2 – image illustrating workflow diagram

V. Verification

1. Managing Shopping Cart

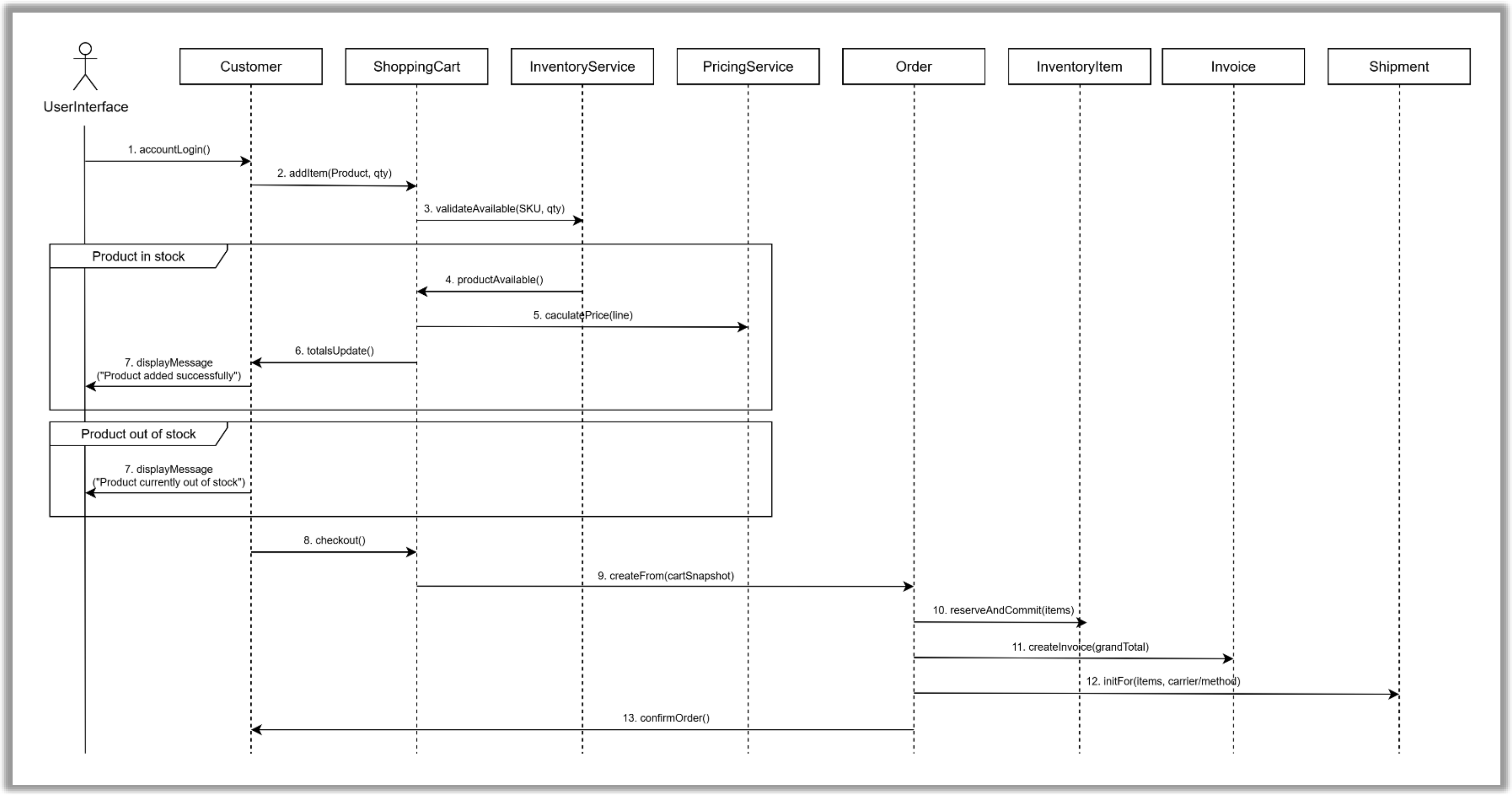

Figure 3 - Sequence Diagram of ShoppingCart Entity

This scenario shows how a customer adds products to the shopping cart and proceeds to checkout. The “ShoppingCart” validates stock via “InventoryService” and updates totals using “PricingService”. If the product is available, it is added successfully; otherwise, an out-of-stock message is shown. During checkout, the “Order” is created from the cart, stock is reserved, an Invoice is generated, and a Shipment is initialized to complete the purchase.

2. Browsing Catalogue

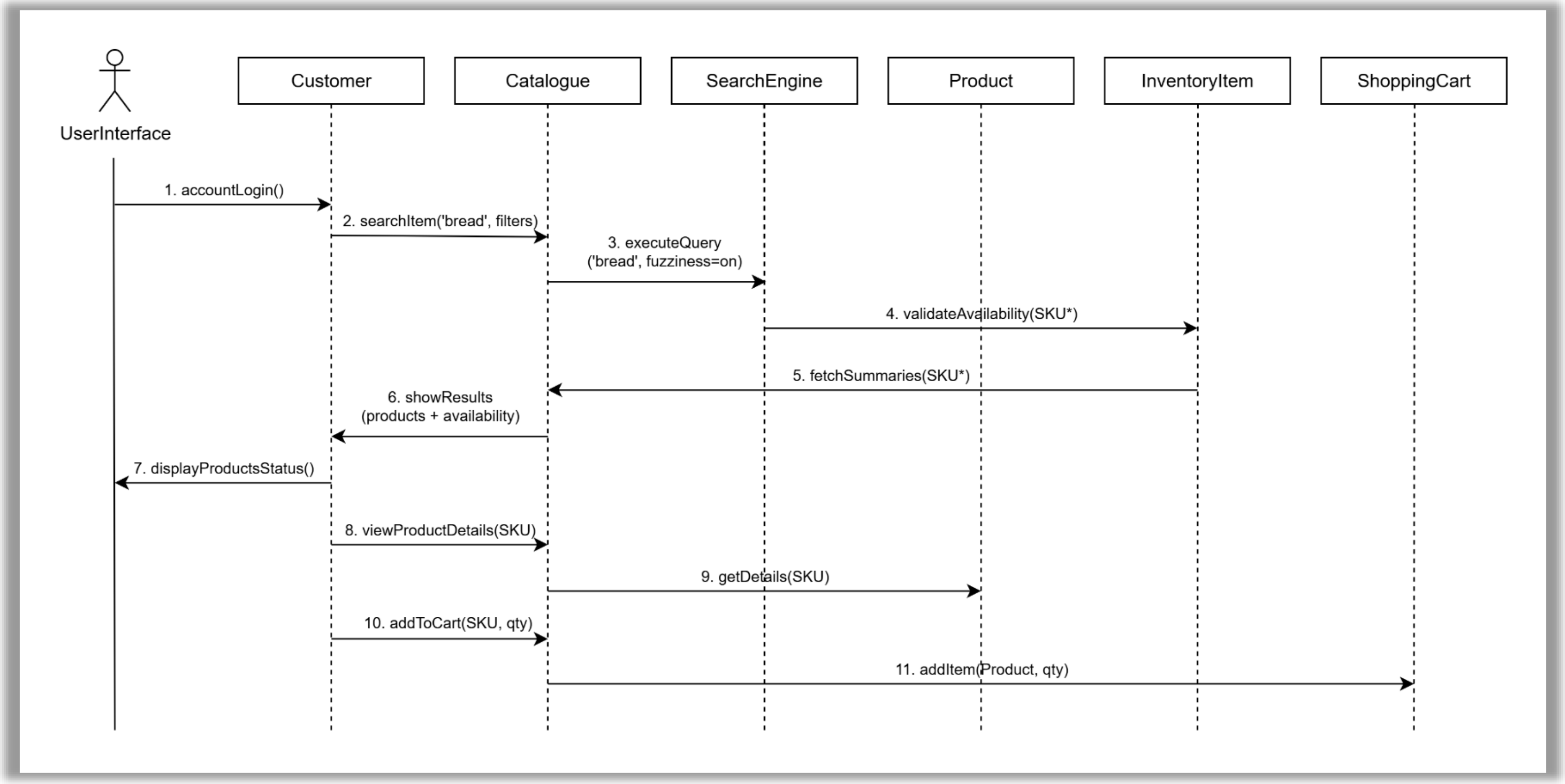
UserInterface
Customer
Catalogue
SearchEngine
Product
InventoryItem
ShoppingCart
1. accountLogin()
2. searchItem('bread', filters)
3. executeQuery ('bread', fuzziness=on)
4. validateAvailability(SKU*)
5. fetchSummaries(SKU*)
6. showResults (products + availability)
7. displayProductsStatus()
8. viewProductDetails(SKU)
9. getDetails(SKU)
10. addToCart(SKU, qty)
11. addItem(Product, qty)

Figure 4: Browsing Catalogue's diagram

This scenario shows how customers search and view products in the catalogue. The "Catalogue" queries the "SearchEngine" to find matching products, including flexible text-based search handling. Then retrieves their stock from the "InventoryItem" with the available results; eventually, it shows the products' status to our customers. Next, when the customers view a product, the "Product" details are fetched, and if they choose to buy the product, the item will be added to the "ShoppingCart" with the chosen quantity.

3. Printing Receipt

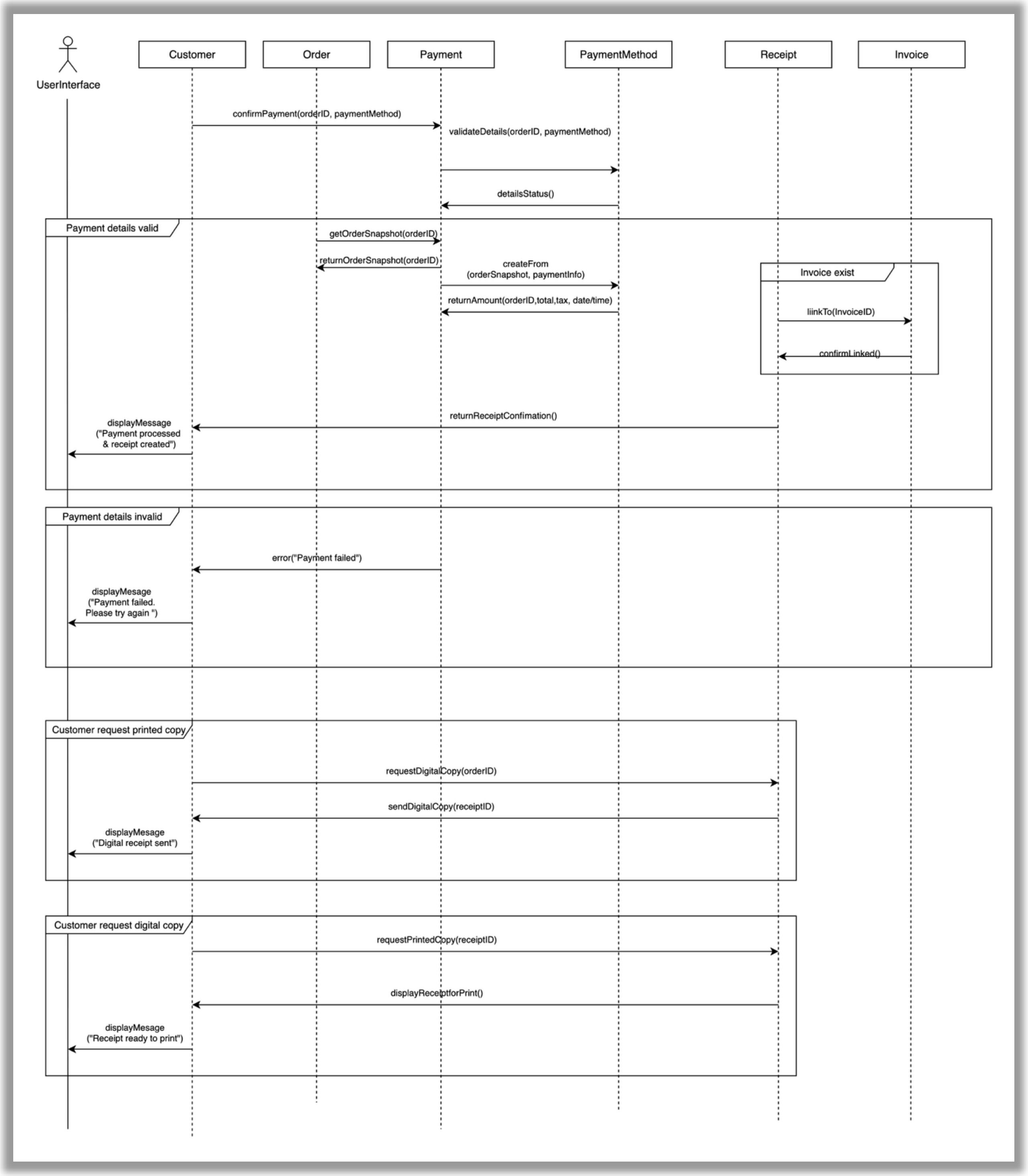
UserInterface
Customer
Order
Payment
PaymentMethod
Receipt
Invoice
confirmPayment(orderID, paymentMethod)
validateDetails(orderID, paymentMethod)
detailsStatus()
Payment details valid
getOrderSnapshot(orderID)
returnOrderSnapshot(orderID)
createFrom (orderSnapshot, paymentInfo)
returnAmount(orderID,total,tax, date/time)
Invoice exist
liinkTo(InvoiceID)
confirmLinked()
returnReceiptConfimation()
displayMessage ("Payment processed & receipt created")
Payment details invalid
error("Payment failed")
displayMesage ("Payment failed. Please try again ")
Customer request printed copy
requestDigitalCopy(orderID)
sendDigitalCopy(receiptID)
displayMesage ("Digital receipt sent")
Customer request digital copy
requestPrintedCopy(receiptID)
displayReceiptforPrint()
displayMesage ("Receipt ready to print")

Figure 5 - sequence diagram of generating receipt

Payment validates with PaymentMethod and processes transactions. On success, Payment gets Order snapshots, Receipts fetch amounts, computes total with taxes and confirms creation, which is linked to an invoice. If payment fails, an error is returned. The optional flows are that digital copies will be sent, or printed copy will be provided on request.

4. Creating Invoice

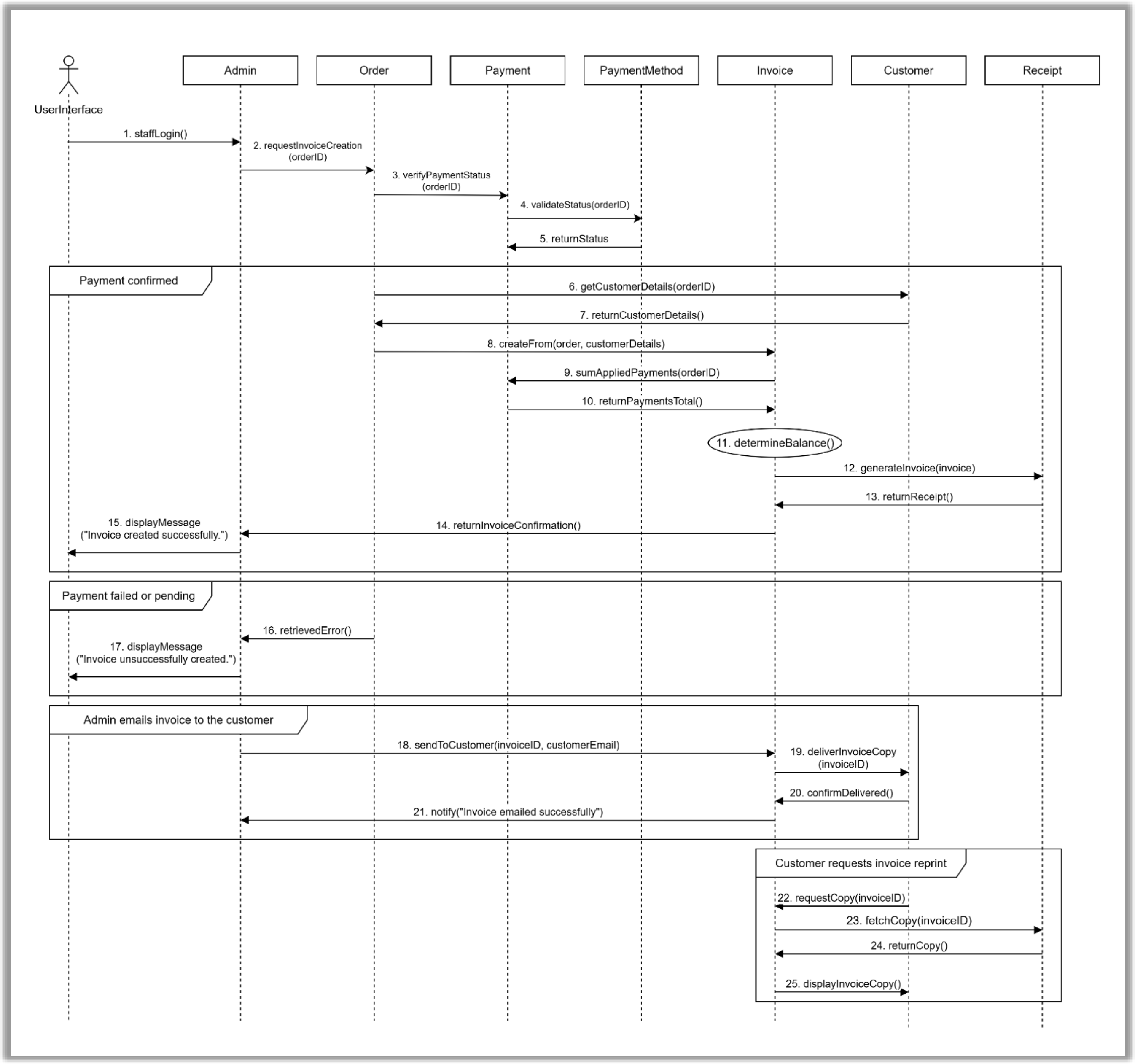
UserInterface
Admin
Order
Payment
PaymentMethod
Invoice
Customer
Receipt
1. staffLogin()
2. requestInvoiceCreation (orderID)
3. verifyPaymentStatus (orderID)
4. validateStatus(orderID)
5. returnStatus
Payment confirmed
6. getCustomerDetails(orderID)
7. returnCustomerDetails()
8. createFrom(order, customerDetails)
9. sumAppliedPayments(orderID)
10. returnPaymentsTotal()
11. determineBalance()
12. generateInvoice(invoice)
13. returnReceipt()
14. returnInvoiceConfirmation()
15. displayMessage ("Invoice created successfully.")
Payment failed or pending
16. retrievedError()
17. displayMessage ("Invoice unsuccessfully created.")
Admin emails invoice to the customer
18. sendToCustomer(invoiceID, customerEmail)
19. deliverInvoiceCopy (invoiceID)
20. confirmDelivered()
21. notify("Invoice emailed successfully")
Customer requests invoice reprint
22. requestCopy(invoiceID)
23. fetchCopy(invoiceID)
24. returnCopy()
25. displayInvoiceCopy()

Figure 6 - sequence diagram of creating invoice

This scenario is when an Admin requests invoice creation from Order which verifies payment with Payment, PaymentMethod, including alteratives and options. Invoice aggregates applied payments, computes total and remaining balance, linking to a Receipt. A confirmation is returned to Admin, otherwise an error is shown, and no invoice is created. The optional flows are emailing invoices to customers or customer invoice reprinting.

VI. References

Bass, L., Clements, P. & Kazman, R. 2013, *Software Architecture in Practice (3rd Edition)*, Addison-Wesley.

Sommerville, I. 2016, *Software Engineering (10th Edition)*, Pearson Education.

Larman, C. 2004, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*, Prentice Hall.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Swinburne University of Technology, *SWE30003 – Software Architecture and Design Assignment 2 Brief*, Semester 2 2025.

VII. Appendix

Swinburne University of Technology, *SWE30003 – Software Architecture and Design Assignment 1 Submission*, Semester 2 2025. (Link)