OBJECT DESIGN

SWE30003 – Software Architecture and Design – Assignment 2

Due Date: 12/10/2025

Team Members:

103992935 - Seth Kalantzis

104577757 - Kaine Price

104443353 - Luke Magnuson

103824847 - Raida Zabin

Contents

Executive Summary

Your Local Shop is a Hawthorn-based convenience store specialising in everyday essentials and speciality products not commonly available in mainstream supermarkets. The company is undergoing a digital transformation – from a purely physical retail model to a hybrid online-physical approach to expand their customer base beyond the local neighbourhood to city-wide and potentially national markets.

This report presents a comprehensive object-oriented design solution for the online convenience store system identified in Assignment 1. The proposed design employs …

The design incorporates established design patterns …

Key quality attributes of performance, usability, reliability and security are addressed through appropriate class responsibilities and collaborations.

1. Introduction

This Object Design Document presents the high-level object-oriented design for **Your Local Shop's** online convenience store system. The document follows Responsibility Driven Design approach to identify classes, their relationships, and collaborative responsibilities required to fulfill the functional requirements established in Assignment 1.

1.1. Purpose and Scope

The purpose of this document is to:

- Define a comprehensive object-oriented design solution using responsibility-driven design principles for Your Local Shop's online convenience store system.
- Provide detailed class specifications with clear responsibilities and collaborations through CRC cards.
- Demonstrate the application of appropriate design patterns and heuristics to ensure maintainability and adherence to object-oriented design principles.
- Establish the foundation for detailed design and implementation to be utilised in Assignment 3.

The scope of this design includes customer account management, product catalogue browsing and management, shopping cart functionality, order processing, payment handling, invoice generation, and sales reporting capabilities.

1.2. Outlook of Solution

The proposed solution will be a web-based, layered architecture system that aligns directly with the business and functional requirements outlined in the Software Requirement Specification (SRS). This architecture separates the presentation, business logic, and data access layers, ensuring clear modular boundaries and maintainable code. The presentation layer will manage user interfaces for both customers and administrators, implemented through responsive web technologies to support access from standard browsers. The business logic layer encapsulates core operations such as catalogue management, order processing, and payment handling, while the data access layer ensures reliable persistence and retrieval of customer, product, and transaction information through a secure relational database.

The design choice satisfies the SRS requirement for potential scalability and maintainability, enabling future enhancements such as integrating with third-party payment or logistics service. Using a layered approach ensures that updates in one layer do not impact user interface components, reducing coupling and facilitating parallel development. This design architecture places emphasis on the separation of concerns and modularity for futureproofing of the system should the client wish to expand their operations, as well as ensuring greater security through separation of user-facing interfaces, executable logic, and raw data.

The object-oriented structure of the solution further strengthens this architecture. Classes such as *Customer, Order, Product,* and *Invoice* encapsulate single business abstractions with defined responsibilities supporting

the SRS quality attributes of reliability, security, and performance. For instance, transaction integrity is maintained through cohesive collaboration between *Order* and *Payment* classes, while data validation and exception handling ensure reliability during concurrent user operations. This object-oriented alignment with the domain model defined in the SRS ensures the system's design directly reflects real-world business operations and user tasks.

Finally, the solution incorporates responsibility-driven design principles and applies relevant design patterns such as Model-View-Controller (MVC) and Singleton (for services like authentication). This not only supports the SRS performance goals such as responsive catalogue browsing and order processing but also improves testability and extensibility.

1.3. Trade-offs and Object Design

1.3.1. Naming Convention

All classes, methods, and variables follow consistent naming conventions as outlined in Section 1.4 to ensure code readability and maintainability across the development team.

1.3.2. Boundary Cases

The system will handle boundary/edge cases including:

- Empty shopping carts during checkout attempts – the system will prompt users to add items before proceeding.
- Out-of-stock items during order processing – real-time stock validation will prevent orders for unavailable products.
- Concurrent cart updates by the same user across multiple sessions – the most recent session takes precedence.
- Product price changes while items are in a user's cart – prices are locked upon adding to cart and recalculated if the session duration limit expires.

1.3.3. Authentication and Authorisation Exceptions

When users attempt to access restricted functionality, the AuthenticationService will throw appropriate exceptions including:

- InvalidCredentialsException – thrown when login credentials are incorrect.
- UnauthorisedAccessException – thrown when authenticated users attempt to access functionality beyond their role permissions.
- SessionExpiredException – thrown when a user's session has timed out and requires re-authentication.

1.3.4. Data Validation Exceptions

The system enforces data integrity through validation exceptions including:

- InvalidDataFormatException – thrown when input data does not match the expected formats (email addresses, or invalid phone numbers)
- OutOfStockException – thrown when attempting to order quantities exceeding available stock.
- InvalidQuantityException – thrown when order quantities are negative or exceed reasonable limits.

1.4. Documentation and Guidelines for Interface

Identifier	Rules	Examples
Classes	Descriptive names using PascalCase representing business concepts	Customer, ProductCatalogue, ShoppingCart
Interfaces	PascalCase with descriptive business terminology	

Methods	camelCase describing actions or queries	addToCart( ), calculateTotal( ), getOrderStatus( )
Variables	camelCase with meaningful business context	productID, totalAmount, orderDate

1.5. Definitions, Acronyms, and Abbreviations

Term	Definition	Use in Design
CRC	The Class-Responsibility-Collaboration card is a tool in Object Oriented design to capture information about a class regarding its name, responsibilities, collaborations and inheritance.	Used to document every candidate class and ensure each has a single clear responsibility.
GST	Goods and Services Tax. A 10% taxation on the sale of goods or services.	Implemented in the *Invoice* and *Product* classes to determine order totals and financial reports.
SKU	Stock Keeping Unit. This is an alphanumeric identifier assigned to each product for tracking inventory.	Used in the *Product* class to act as an identifier.
AUD	Australian Dollar. The official currency used in the system. The scope of the project does not cover international currencies.	All monetary values are represented in AUD and stored with decimal precision to avoid rounding errors.
UML	Unified Modelling Language. This is a standard visual notation for representing software structure and behaviour	Used for class and sequence diagrams to illustrate relationships and workflow.
OO	Object-Oriented. The programming and design paradigm that acts as the focus for this unit, based on encapsulating data and behaviour within interacting objects.	The entire system applies OO principles across all core classes. This system applies Object-Oriented Design and will utilize Object-Oriented Programming for implementation.
API	Application Programming Interface. This is a defined set of operations that allows different software components to communicate.	External services such as payment gateways, email providers and external logistics handling would be handled with API calls.

2. Problem Analysis

The Software Requirements Specification identified key functional requirements that must be addressed through appropriate object-oriented design. These requirements reflect the core business operations and user interactions necessary for the system to support Your Local Shop's transition to a digital platform and include the following:

- Record and manage entity details
  - Customer profiles and delivery information
  - Product specifications and availability
  - Catalogue product options and categories
  - Administrative accounts and permissions

- Modify existing records
    - Updates to customer contact or delivery details
    - Adjustments to product pricing, stock levels, and availability
    - Amendments to catalogue metadata and product categorisation
    - Role-based administrative changes
- Track financial and transactional status
    - Payment status per order
    - Invoice generation
    - Order cancellation
- Support reporting and analytics
    - Generation of sales reports across configurable timeframes
    - Identification of top-selling products and low-stock
    - Aggregated revenue and order volume metrics for business insights
- Enable customer-facing operations
    - Browsing and searching catalogue
    - Managing shopping cart contents across sessions
    - Checkout and payment processing
    - Order tracking and history

These functional requirements form the foundation for the system's class structure and behavioural modelling. Each requirement has been mapped to specific candidate classes and responsibilities, ensuring high cohesion and traceability throughout the design. The object-oriented approach allows for modularity, scalability, and future extensibility which is critical for supporting Your Local Shop's evolving business needs.

2.1. Assumptions

ID	Assumption
A1	The system operates with a single physical store location with no immediate multi-location requirements.
A2	All products have unique identifiers (SKU or product ID) that remain consistent across their lifecycle in the system.
A3	Customers must provide valid email addresses, phone numbers, and delivery addresses to complete purchases.
A4	New customers are assigned a unique customer ID upon successful account creation and email verification.
A5	All transactions, customer accounts, products, and orders will be stored electronically in a centralised database.
A6	The system will only support English language content and Australian currency (AUD).
A7	Payment processing will be handled through a third-party payment gateway that is PCI-DSS compliant, eliminating the need for the system to store complete credit card details.
A8	All financial documents (invoices) must comply with Australian Tax Office requirements including GST calculations and 5-year retention periods. Invoices serve as both pre-payment bills and post-payment receipts.
A9	Delivery will only occur during regular business hours and only to Australian addresses.
A10	Specialty products and everyday essentials are treated identically within the system; categorisation is managed through Product attributes (category field) rather than a separate Categories class.
A11	Guest users can browse the catalogue and add items to cart but must create an account to complete checkout.
A12	Shopping cart contents for registered users persist across sessions indefinitely until checkout or manual removal; guest carts are session-based and expire when the browser session ends.
A13	Once an order is placed and payment confirmed, order details (products, prices) become immutable for audit and legal compliance purposes.
A14	Product prices may change over time, but historical orders retain the original purchase prices.
A15	Stock quantities are tracked in real-time, and the system prevents overselling by validating availability during checkout.
A16	A single order contains one delivery address and one payment transaction, though the order may contain multiple products.

A17	Refunds are processed through the original payment method and payment gateway, not directly by the system.
A18	Sales reports aggregate data from completed (paid) orders only; pending or cancelled orders are excluded from sales statistics.
A19	The system resides in a secure hosting environment with appropriate firewalls, SSL/TLS encryption, and access controls as specified in security requirements.
A20	Administrative users (Admin class) have role-based permissions, with store owners having elevated privileges for catalogue management and reporting.
A21	Products can be marked as unavailable or removed from catalogue without deletion from the database, preserving historical order integrity.
A22	Email notifications (order confirmations, shipping updates) are sent asynchronously and delivery failures do not block primary workflows.
A23	The system does not handle loyalty programs, promotional discounts, or customer reviews in the initial implementation.
A24	Integration with third-party marketplace platforms (eBay, Facebook Marketplace) is out of scope.
A25	The system does not sell age-restricted items (alcohol, tobacco, pharmaceuticals) that would require additional compliance verification.
A26	The system assumes a stable internet connection for all users; offline functionality is not supported in the initial implementation.
A27	All users (customers and administrators) access the system through standard web browsers (Chrome, Firefox, Safari, Edge) without requiring specialised software installation.
A28	The system uses a layered architecture pattern separating presentation, business logic, and data access layers as outlined in Solution Approach 3 from Assignment 1.
A29	Database backup and disaster recovery procedures are managed by the hosting infrastructure provider and are outside the scope of the application design.
A30	Product images are stored externally (e.g., CDN or cloud storage) and referenced by URL; the system does not handle image file uploads or storage directly.
A31	Shopping cart state for registered users persists in the database; guest cart state is maintained in browser session storage only.
A32	Order fulfillment and shipping logistics are managed manually by store staff; the system provides order management interfaces but does not automate warehouse operations.
A33	Sales reports are generated on-demand rather than pre-computed; report generation time is acceptable up to 30 seconds for large date ranges.
A34	User authentication sessions expire after 30 minutes of inactivity as per security requirements outlined in Assignment 1 Section 7.4.
A35	The system supports a maximum of 10,000 products in the catalogue without performance degradation, aligning with the scale of a single convenience store operation.
A36	Email notifications are triggered by system events, but actual email delivery is handled by a third-party email service provider (e.g., SendGrid, AWS SES).
A37	All monetary values are stored and calculated using decimal precision to avoid rounding errors in financial transactions and tax calculations.
A38	Customer passwords must meet minimum complexity requirements: at least 8 characters, including uppercase, lowercase, and numeric characters.
A39	The system logs all critical operations (orders, payments, admin actions) for audit purposes with logs retained for a minimum of 12 months.
A40	Product stock levels are updated synchronously during checkout to prevent race conditions; concurrent order processing is handled through database transaction locking.

2.2. Simplifications

Based on the assumptions and scope constraints, the following simplifications have been made:

Payment Processing

The initial implementation simulates payment transactions rather than integrating with actual payment gateways. The ‘Payment’ class records transaction details but does not process real financial transactions.

Email Notifications

Rather than implementing full email delivery infrastructure, the system logs notification events that would trigger emails in a production environment.

Delivery Management

Physical delivery logistics (courier assignment, tracking, delivery scheduling) are managed externally. The 'Order' class only tracks delivery address and order status.

Product Categories

Instead of a separate 'Category' class, product categorisation is handled as attributes within the 'Product' class.

Cart and Order Items

Rather than separate 'CartItem' and 'OrderItem' classes, these are represented as structured data elements within arrays/collections managed by 'ShoppingCart' and 'Order' respectively.

Inventory Management

A dedicated 'Inventory' class is not required as stock levels are managed as attributes of 'Product', with 'ProductCatalogue' coordinating stock queries.

2.3. Design Justification

The design employs a Responsibility-Driven Design (RDD) approach where each class encapsulates a single key abstraction with well-defined responsibilities and clear collaborations. This approach was chosen to:

Maintain High Cohesion

Each class has a focused purpose aligned with a single business or task concept e.g. Customer manages customer data and behaviour.

Minimise Coupling

Classes interact through defined interfaces rather than direct access to internal state, facilitating independent testing and any future modifications.

Support Extensibility

The design accommodates future enhancements including payment gateway integration, without requiring fundamental restructuring.

Ensure Traceability

The class structure maps directly to the business requirements identified in Assignment 1, ensuring all functional requirements are addressed.

The use of an abstract 'User' base class provides flexibility for adding additional user types in the future while consolidating common authentication and profile management responsibilities. The separation of 'ProductCatalogue' form 'Product' enables a centralised product management system and search functionalities while keeping individual product responsibilities focused.

2.4. Discarded Class List

The following candidate classes were considered but ultimately discarded:

Inventory

Stock management is handled as an attribute of 'Product' with coordination through 'ProductCatalogue' rather than requiring a separate class.

Supplier

Supplier information is not required for the initial online storefront focused on customer-facing operations and deemed out of scope.

StoreStaff/StoreOwner

These roles are consolidated into the ‘Admin’ class with role-based permissions rather than separate class hierarchies.

Receipt

Invoices serve as both pre-payment billing documents and post-payment receipts, eliminating the need for a separate ‘Receipt’ class.

PaymentMethod

Payment method details (card, account) are captured as attributes within the ‘Payment’ class rather than requiring a separate subclass for the initial implementation.

ShippingLabel

Shipping labels are generated externally, and the ‘Order’ class only maintains delivery address information.

DeliveryAddress

Address information is stored as attributes of ‘Customer’ and ‘Order’ rather than requiring a separate class.

SearchEngine

Product search functionality is implemented as methods within ‘ProductCatalogue’ rather than a separate control class.

OrderItem/CartItem

These are represented as structured data within collections rather than full-fledged classes, simplifying the design while maintaining necessary functionality.

Shipment

Shipment tracking is managed externally; ‘Order’ maintains only basic delivery status information.

Category

Product categorisation is handled through ‘Product’ attributes rather than a separate class structure.

EmailService

Email notifications are simulated through logging in the initial implementation, with notification triggers managed within relevant domain classes.

3. Candidate Classes

3.1. Candidate Class List

The final class set derived from problem domain analysis includes:

User Management:

- User (abstract base class)
- Customer (extends user)
- Admin (extends user)

Product and Catalogue Management:

- Product
- ProductCatalogue

Order Management:

- Order
- ShoppingCart

Payment/Financial Documentation:

- Payment
- Invoice

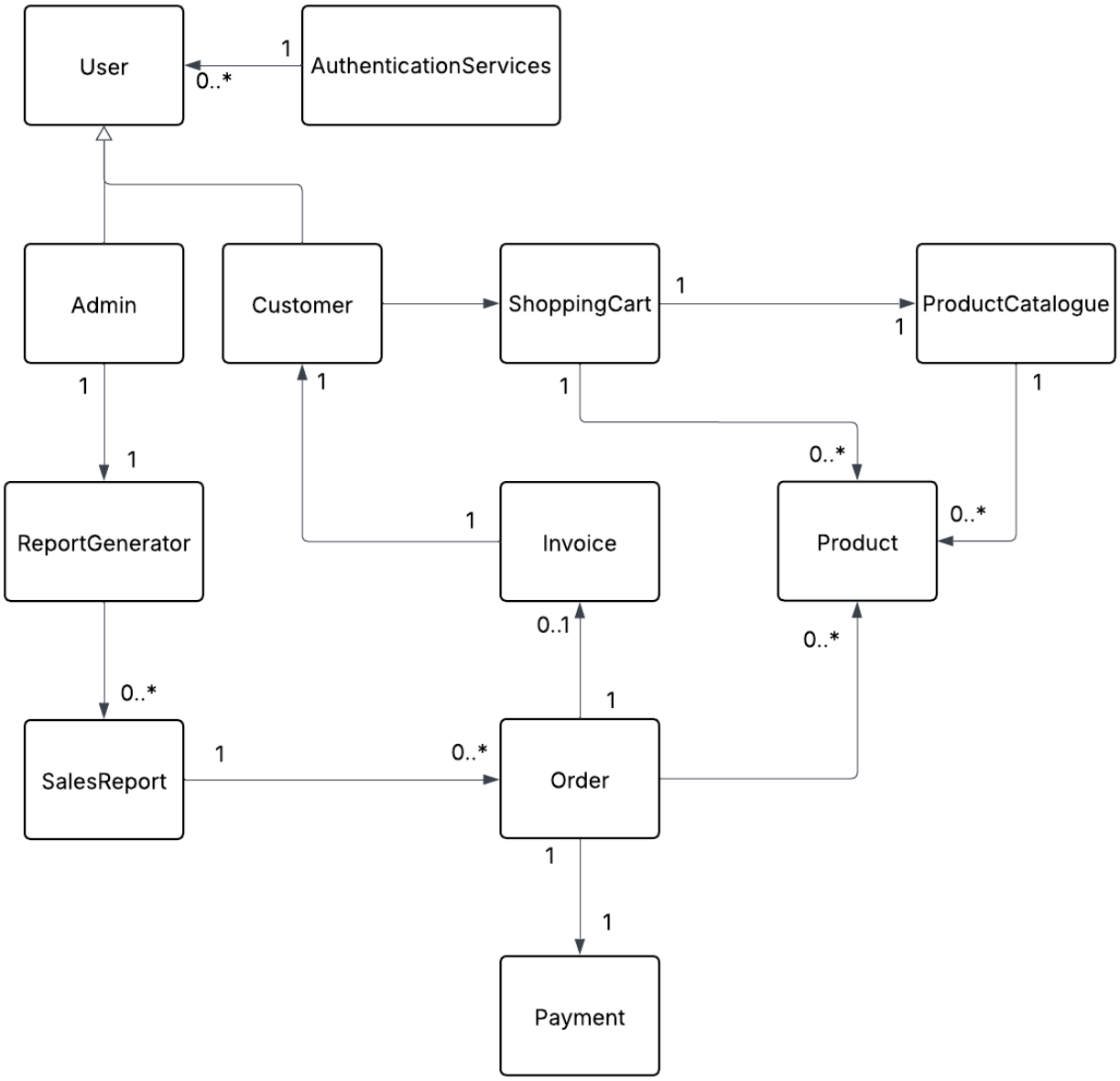
Reporting/Analytics:

- SalesReport
- ReportGenerator

SupportingClasses:

- AuthenticationService

3.2. UML Diagram



3.3. CRC Cards

3.3.1. User

Class Name: User **Super Class:** N/A	
The base user class other user types inherit from. Dictates the authentication requirements and principals for users (Customers and Admins) interacting with the system (setting credentials, user status, roles, etc.) Provides the core security- and profile-related behaviour that subclasses reuse.	
Responsibilities	**Collaborators**
Knows userID, role, account status, creation date and last logged in.	N/A
Knows user name, email, phone number, and password.	N/A
Can log in and start a session.	AuthenticationService
Can log out and invalidate / end session.	AuthenticationService
Can change password.	AuthenticationService
Can update profile email and contact info.	N/A
Can check if user has permission to perform actions or view pages using RBAC.	AuthenticationService

3.3.2. Customer

Class Name: CustomerUser **Super Class:** User	
Represents end-user customers who browse products, manage shopping carts, place orders, and track purchases. Extends ‘User’ with customer-specific functionality including order history, delivery addresses, and cart management.	
Responsibilities	**Collaborators**
Attributes: customerId, deliveryAddress, phoneNumber, orderHistory[], cartId	N/A
browseProducts() to view available products	ProductCatalgoue
addToCart(productId, quantity) to add products to shopping cart	ShoppingCart, ProductCatalogue
veiwCart() retrieves current shopping cart	ShoppingCart
checkout(): Order which initiates order creation from cart	ShoppingCart, Order
makePayment(orderId, paymentDetails) to process payment for order	Order, Payment
viewOrders(): Order[] to retrieve customer’s order history	Order
getOrderStatus(orderId) to check status of specific order	Order
updateDeliveryAddress(address) to update default delivery address	N/A
getRole(): string will return “Customer”	N/A

3.3.3. Admin

Class Name: AdminUser **Super Class:** User

User class utilised by store owner / staff role performing tasks related to managing the catalogue, order fulfilment, and generating/viewing sales reports.	
Responsibilities	**Collaborators**
Knows adminID and Admin permissions.	AuthenticationService
Knows last administrative action for auditing.	N/A
Can access the administrative dashboard to complete admin actions.	AuthenticationService
Can create, edit, publish, and remove products.	Product, ProductCatalogue
Can adjust stock levels and availability	ProductCatalogue
Can review orders and update order status.	Order
Can generate and resend invoices	Invoice, Order
Can produce sales reports for a period and export results	SalesReport, ReportGenerator
Can view audit logs of events and admin actions	N/A

3.3.4. Product

Class Name: Product **Super Class:** -	
Represents a single product item available within the catalogue and for purchase. Encapsulates all product-specific information including pricing, stock levels, and categorisation. Core business entity in the domain model.	
Responsibilities	**Collaborators**
Attributes: productId (SKU), name, description, price, stockQuantity, category, imageURL, isAvailable	N/A
getDetails() returns complete product information	N/A
checkAvailability(quantity) validates if requested quantity is in stock	N/A
updateStock(quantity) to adjust stock level either increase/decrease	N/A
updatePrice(newPrice) to change product price	N/A
setAvailability(available) to mark product as available or unavailable	N/A
getCategory() to return the products category	N/A
calculateGST() to calculate GST component of price (10%)	N/A

3.3.5. ProductCatalogue

Class Name: ProductCatalogue **Super Class:** -	
Manages the complete collection of products available in the store. Provides search, filter, and stock coordination functionalities. Acts as the central point for product-related queries and operations.	
Responsibilities	**Collaborators**
Attributes: products[], lastUpdated	N/A
getAllProducts() to retrieve all available products	Product
searchProducts(keyword) to search products by name or description	Product
filterByCategory(category) to return products in specific category	Product
getProductById(productId) to retrieve specific product details	Product
addProduct(product) to add a new product to catalogue	Product
updateProduct(product) to update existing product	Product

removeProduct(productId) to remove product from catalogue	Product
checkStockLevel(productId) to return current stock quantity	Product
getLowStockProducts(threshold): Product[] to identify products below stock threshold	Product
getProductsByPriceRange(min, max): Product[] to filter products by price range	Product

3.3.6. Order

Class Name: Order **Super Class:** -	
Represents an order placed by a Customer. Connects Customer, Products, and	
Responsibilities	**Collaborators**
Knows the unique ID number for the order	N/A
Knows the total amount paid for the order	Payment
Knows the Customer that placed the order	Customer
Knows when the order was placed	N/A
Recording order status (pending, fulfilled, delayed)	N/A
Track Products and amounts in order	ShoppingCart
Tracks payment status of order	N/A
Report to ReportGenerator based on Admin input	ReportGenerator, Invoice

3.3.7. ShoppingCart

Class Name: -ShoppingCart **Super Class:** -	
Description – Contains items temporarily a customer wishes to buy. Maintains item list and quantities. Prices are kept consistent and recalculates totals. Validates the availability, persists for users to sign-in and hands off to order/checkout	
Responsibilities	**Collaborators**
Contains cart items quantity, price, availability	Product, ProductCatalogue
Add product to cart; increases quantity if already present and validates stock	ProductCatalogue, Product
Update quantity / remove item; enforce min/max and stock limits	ProductCatalogue, Product
Show running totals (subtotal, estimated tax/shipping if applicable)	Product, (Invoice for tax rules if needed)
Persist cart for authenticated customer; expire guest carts after a period	Customer
Restore cart on login / device change	Customer
Refresh item availability & price when cart is revisited; warn on changes	ProductCatalogue, Product
Create Order with OrderItems from current cart at checkout	Order
Clear cart after successful order placement	Order
Handle bulk actions (empty cart, remove selected)	
Record simple audit entries for cart mutations (not sure)	

3.3.8. Payment

Class Name:	Payment
Super Class:	-
Records financial transaction details for order payment. Captures payment method, amount, transaction status, and timestamps. Simulates payment processing in initial implementation.	
Responsibilities	**Collaborators**
Attributes: paymentId, orderId, amount, paymentMethod, paymentStatus, transactionDate, paymentDetails, confirmationNumber	N/A
processPayment(orderId, amount, method, details) to create and process payment	Order
validatePaymentDetails(details) to validate payment information format	N/A
confirmPayment() to mark payment as completed	Order
failPayment(reason) to mark payment as failed	Order
getPaymentStatus() to return current payment status	N/A
getTransactionDetails() to return payment information	N/A
generateReceipt() to create payment receipt	Invoice
refundPayment(amount) to process a refund	Order

3.3.9. Invoice

Class Name:	Invoice
Super Class:	
Generates document itemising products, calculating taxes, and serving as both pre-payment and post-payment receipt. Ensures Australian Tax Office compliance including GST calculations and retention requirements	
Responsibilities	**Collaborators**
Attributes: invoiceId, orderId, customerId, invoiceDate, dueDate, lineItems[] {productName, quantity, unitPrice, gst, total}, subtotal, totalGST, grandTotal, invoiceStatus, businessDetails, customerDetails	N/A
generateInvoice(order, customer) to create an invoice from order	Order, Customer

3.3.10. SalesReport

Class Name:	SalesReport
Super Class:	-
Represents reports about sales over a defined period, including totals, trends, and breakdowns by product, category, or region.	
Responsibilities	**Collaborators**
Aggregate sales data for target time frame	Orders
Evaluate collected data for specific metrics	N/A
Generate report output in required format (CSV, PDF, xls)	N/A

3.3.11. ReportGenerator

Class Name:	ReportGenerator
Super Class:	-

Handles the creation of reports for various elements of the business. This is done by retrieving data from sales and inventory systems.	
Responsibilities	**Collaborators**
Handle report parameters	Admin
Evaluate data according to metrics	Order, Invoice
Output report	SalesReport

3.3.12. AuthenticationService

Class Name: AuthenticationService **Super Class:** -	
Handles the registration, authentication, session lifecycle, password management, and RBAC checks for customer and admin users. Implements the Singleton and Factory Method patterns to ensure one global instance and controlled creation of user subclasses.	
Responsibilities	**Collaborators**
Knows session times, password policies and role permissions	
Can validate user input to register users, creates either a customer or Admin object, and stores credentials.	CustomerUser, AdminUser, User
Can authenticate a user via credentials and starts sessions	User
Can issue session tokens with designated timeout times for reauthentication	User
Can validate sessions, checking session validity and expiry	User
Can log out user and end user session.	User
Can change a user's password, validates new password with policy and updates credentials	User
Can check a user's permission to allow proper access based on role	User, Admin
Can throw exceptions for invalid credentials, unauthorised access, or session expiry	
Can create audit logs for successful and failed auth attempts	

4. Design Quality

4.1. Design Heuristics

4.1.1. H1: A class should capture one and only one key abstraction.

Each class in our system represents a single, defined business concept. For example, the 'Product' class exclusively manages product-related attributes and behaviours (pricing, stock, availability), while 'Order' handles order management. We deliberately avoided creating a monolithic 'Store' class that would combine product management, order processing, and payment handling. This separation ensures each class has a focused responsibility and can evolve independently.

This heuristic prevents "god classes" and promotes maintainability. When Your Local Shop needs to modify how products are priced or categorised, changes are isolated to the 'Product' and 'ProductCatalogue' classes

without impacting payment or order processing logic. For example, if the store decides to implement dynamic pricing based on seasonal demands, the ‘Product’ class only requires modification, leaving ‘ShoppingCart’ and ‘Order’ unchanged.

4.1.2. H2: Keep all implementation details private.

Internal attributes and helper methods should be encapsulated following OO principles. Classes should interact through public-facing interfaces, allowing for greater maintainability and integrity.

4.1.3. H3: Minimise coupling between classes.

Interactions should occur through well-defined associations and service-class interfaces (for example, the AuthenticationService class mediates user logins). This reduces dependency and allows for independent modification without significantly altering or affecting other classes.

4.1.4. H4: Maximize cohesion within classes.

Each class groups tightly related data and operations that serve a single functional goal. For instance, *Order* maintains all the behaviour and data related to managing the order lifecycle, rather than dispersing these across multiple classes. Whilst the *Admin* class may be able to change the status of the order manually, if this is done through a public interface provided by the *Order* class, cohesion is not reduced. This improves both the readability and maintainability by keeping logic local to where the related class states reside.

4.1.5. H5: Favour composition over inheritance where practical.

Rather than building deep inheritance hierarchies, which can overcomplicate things and create too many dependencies, the design uses aggregation of classes and collaboration. For instance, the *ShoppingCart* uses composition with *Product* objects instead of subclassing it. Each cart contains a collection of *Product* instances and manages quantities, totals, and validation. If inheritance were used (e.g., *ShoppingCart* extends *Product*), the design would falsely imply that a cart is a product, creating unnecessary coupling.

4.1.6. H6: An inheritance class should not override a base class method if it returns a null operation.

Abstract base classes (e.g., *User*) should not provide placeholder or “do-nothing” implementations for methods intended to be overridden. If a subclass needs to define a new behaviour, declare the parent method as abstract so the compiler will enforce implementation. This prevents silent logic errors from occurring and ensures that classes that inherit from a parent class have meaningful extensions upon their base abstraction.

4.1.7. H7: Clearly define class boundaries and interfaces.

Each class exposes a small, well-defined public interface and hides its internal data and helper logic. Other parts of the system must interact only through that interface. This keeps changes local (we can adjust internals without ripple effects), makes testing simpler, and prevents “reach-through” coupling where one class manipulates another’s internals.

In our design, the ShoppingCart provides cart-level operations (e.g., add, update quantity, checkout) without exposing its internal line-item structure; the Order class owns its lifecycle and is created from a cart snapshot, then changes its own status (e.g., to Paid) rather than allowing callers to set fields directly; the Payment component accepts an amount and order identifier and returns a result (status and reference) without revealing simulation details; and the ProductCatalogue offers read/search functions so pricing and availability are read through the catalogue or Product, not modified by external classes. These clear boundaries keep responsibilities obvious, reduce accidental dependencies, and allow us to evolve internals, such as how totals are calculated or how payment is simulated without rewriting callers.

4.1.8. H8: Design for testability

Classes have clear, single responsibilities with well-defined inputs and outputs, making them independently testable. Method outputs can be tested with mock objects without requiring a database connection. Each

method has a focused purpose that can be verified through unit tests. Dependencies are injected through method parameters rather than hard-coded, allowing test doubles to be substituted easily.

Testable design reduces defects and facilitates continuous integration practices outlined in Assignment 1's design requirements. Modifying 'Invoice' GST calculation to handle tax-exempt business customers can quickly verify correctness through isolated unit tests without deploying the entire system or processing payments. This supports the quality attribute requirements for reliability by enabling thorough testing before deployment.

4.1.9. H9: Abstract classes should serve as base types online

Abstract types (e.g. User) shouldn't be instantiated directly or used as placeholders. When an order or session is created, the system should always reference a specific, concrete subtype (either *Customer* or *Admin*). This prevents ambiguous object behaviour and guarantees that all actions occur through a define role, aligning with the system's real-world model where "generic users" do not operationally exist.

4.1.10. H10: Ensure traceability to requirements.

Every class should map directly to a requirement or use case gathered from the Software Requirements Specification. Traceability ensures that modifications to requirements can be tracked back to specific design components and verified through the provided CRC cards and sequential diagrams.

5. Design Patterns

5.1. Creational Patterns

5.1.1. Factory Method Pattern

The Factory Method pattern defines an interface for creating objects but allows subclasses to decide which class to instantiate. It delegates instantiation logic to subclasses, promoting loose coupling between the creator and concrete products.

The 'AuthenticationService' acts a factory for creating user instances during registration and login. When a user registers, the service determines whether to instantiate a 'Customer' or 'Admin' object based on registration parameters. Methods utilised examines the user details and invokes the appropriate constructor, returning a 'User' reference that can be either 'Customer' or 'Admin'.

The Factory Method provides controlled, centralised user creation that enforces security policies and ensures consistent initialisation. When Your Local Shop adds new user roles (e.g. Supplier, Manager, Warehouse etc.), only the factory method in 'AuthenticationService' requires modification. This supports extensibility requirements while maintaining encapsulation of instantiation logic.

5.1.2. Singleton Pattern

The Singleton Pattern is a creational design pattern that ensures a class has only one instance while providing a global access point to it. This is useful when exactly one object is needed to coordinate actions across the system, such as configuration managers, logging services, or database connection handling. By restricting direct construction through a private constructor and exposing a public static *getInstance()* method, the pattern prevents uncontrolled instantiation while maintaining lazy initialization when necessary.

In OOD, the Singleton encapsulates shared resources and centralised control, promoting consistency across modules that rely on a common state or configuration. For example, for Your Local Store, a *ReportGenerator* or *AuthenticationService* could use the Singleton pattern to ensure that all parts of the system reference the same service instance, maintaining uniform access control, or consistent report generation.

However, the pattern should be used sparingly, as it introduces a form of global state that can complicate testing and concurrency if not implemented carefully. Thread safety and lifecycle management must be explicitly handled to prevent multiple instances or race conditions, particularly in an environment such as a

multi-threaded web system. When properly implemented, the Singleton supports controlled resource management and predictable system behaviour.

5.2. Behavioural Patterns

5.2.1. Strategy Pattern

The Strategy pattern is planned for future implementation when payment method complexity changes. Currently, the 'Payment' handles minimal methods and complexity through a unified interface with 'paymentMethod' attribute distinguishing type. As Your Local Shop expands payment options (digital wallets, split payment, buy now pay later schemes), the Strategy pattern will encapsulate method-specific processing.

Different payment methods require different validation rules, security protocols, and integration points. Credit cards for example, require algorithm validation, CVV checks, PCI-DSS compliance, integration with payment gateways, fraud detection and authentication. Without the Strategy pattern, the 'Payment' class would contain large conditional statements that violate principles and make the class difficult to test and maintain.

This pattern will support Task 5 (Handling Payments) from Assignment 1 and security quality attributes in Section 7.4 if PCI-DSS compliance was later required.

5.2.2. Template Method Pattern

The abstract 'User' class implements the Template Method pattern for user authentication workflows. The 'authenticate' method defines the common authentication algorithm structure, while allowing 'Customer' and 'Admin' subclasses to customise specific steps.

The authentication algorithm needs to follow a consistent structure for all user types:

- Validate input
- Check account status
- Verify password
- Role-specific checks
- Create session

However, administrators require additional security measures that customers do not need. The Template Method pattern will allow reusing the common authentication logic while enabling role-specific customisation through a hook method.

This approach prevents code duplication as 'Customer' and 'Admin' would each implement complete authentication methods with similar code. It also ensures consistent application of security policies across all user types.

5.3. Structural Patterns

5.3.1. Façade Pattern

The 'ReportGenerator' class acts as a façade for the sales reporting subsystem. Generating sales reports involves coordinating multiple classes including 'Order', 'Product', 'Customer', 'SalesReport') and performing complex operations including:

- Querying orders from database with date filters
- Aggregating financial data
- Calculating metrics
- Identifying top-selling products
- Grouping sales by Product Category
- Comparing periods
- Formatting output in various formats

Report generation involves complex interactions across multiple subsystems that administrators should not need to understand. The Façade pattern provides a simple, intuitive interface that hides complexity and thus reduces coupling between 'Admin' and the reporting subsystem.

The façade also provides a place for cross over between concerns including caching, authorisation, logging, and error handling.

5.3.2. Model – View – Controller pattern

In this project, the Model-View-Controller pattern is used to separate the data carrying classes (Invoice, Order, Product, SalesReport, User and related classes) and view classes (ShoppingCart, ProductCatalogue) from the controller classes (ReportGenerator, AuthenticationServices). This pattern allows for a clear separation of classes when it comes to designing the customers website. The compartmentalisation that comes with this set up allows for future expansion options to be easily incorporated.

There may be an option to include the Model - View – ViewModel (MVVM) pattern in place of the Controller, at least when focusing on the website elements.

6. Bootstrap Process

The bootstrap process describes the initialisation sequence when Your Local Shop online store customers login and create their first purchase. This section demonstrates which classes are responsible for creating instances of other classes and the order in which these instantiations occur and demonstrate the collaborative responsibilities between these classes in our design.

6.1. New Customer Registration and First Purchase Example

This workflow reflects a new customer creating an account, browsing products, and completing their first purchase.

Creation Sequence:

1. AuthenticationService receives a registration request
2. AuthenticationService creates a Customer instance (Factory Method)
    a. Customer stores account info in database
3. Customer browses catalogue by querying ProductCatalogue (singleton)
    a. No new objects created as uses existing Product instances
4. Customer adds items to cart
5. Customer Creates a ShoppingCart instance
    a. ShoppingCart stores cart items
    b. ShoppingCart validates availability with ProductCatalogue
6. Customer proceeds to checkout
7. ShoppingCart creates an Order instance
    a. Order captures immutable snapshot of items, prices, and delivery details
    b. Order stores itself in database
8. Order creates an Invoice instance
    a. Invoice calculates GST and totals
    b. Invoice stores itself in database
9. Customer submits payment information
10. Order creates a Payment instance
    a. Payment processes transaction
    b. Payment stores transaction details in database
11. Payment confirms successful transaction
12. Order updates status to CONFIRMED
13. Invoice updates status to PAID
14. ProductCatalogue updates stock level for purchased products.

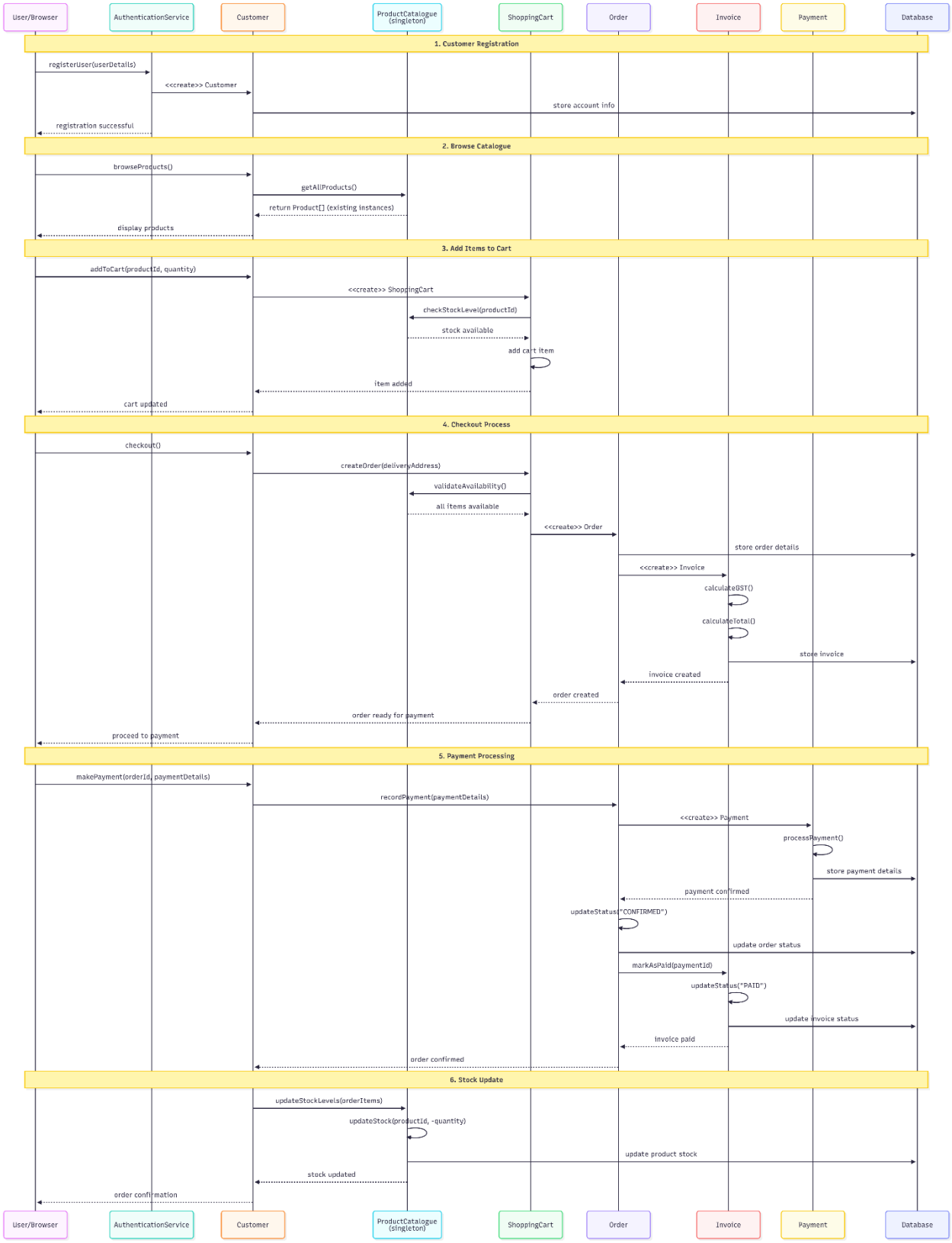


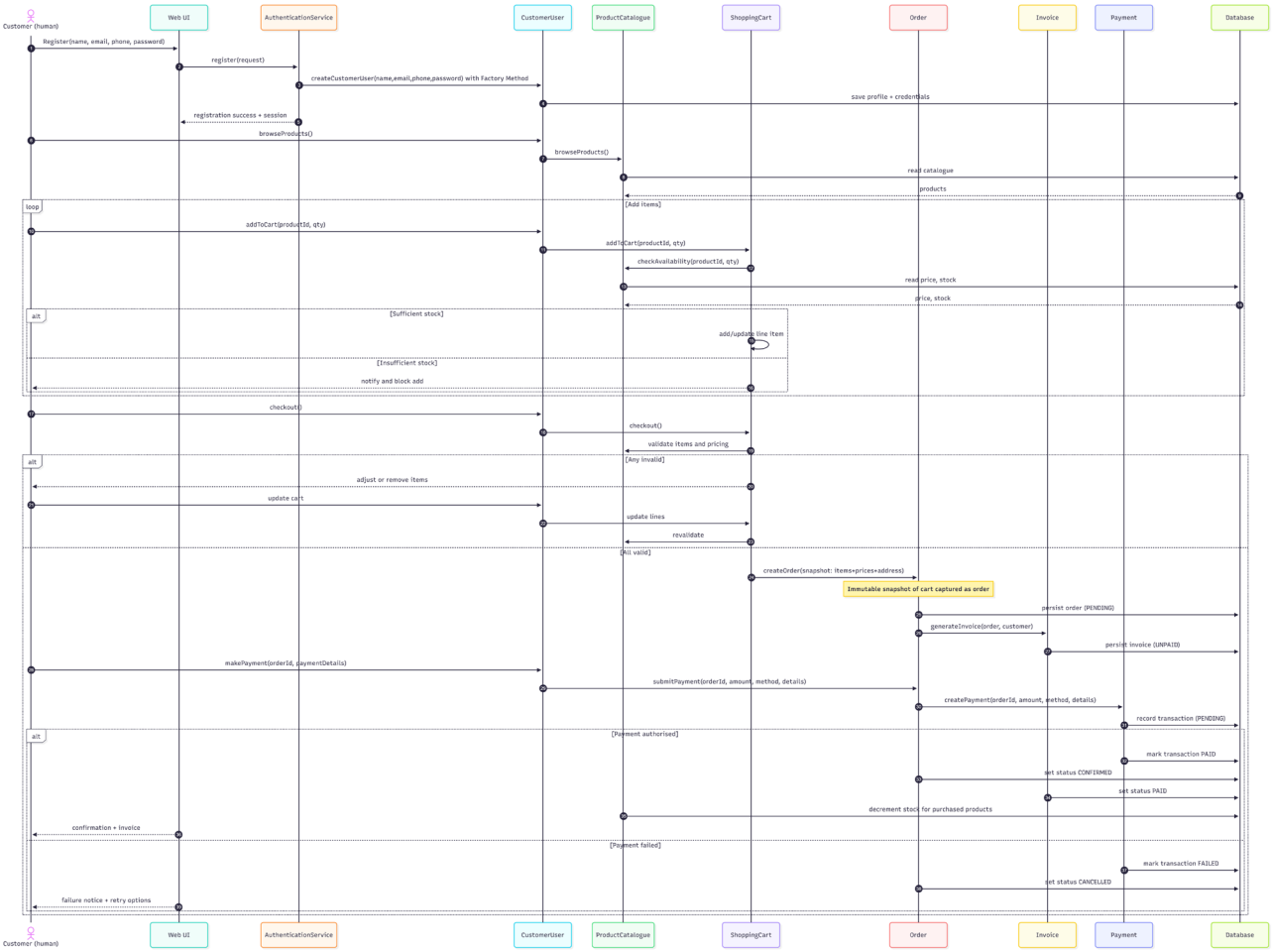
Figure 1 Bootstrap process for new customer account and first purchase

7. Verification

SWIMLANE DIAGRAMS GO HERE:

7.1. New Customer Registration and First Purchase

This sequence diagram shows the process of a new customer registering with the website and completing their first purchase. It includes fragments to show alternative and optional pathways.



The following describes the various workflows that occur throughout this scenario:

Registration to Purchase: Authentication receives a signup request from the physical Customer actor. It validates input and creates a concrete CustomerUser via its factory method. Credentials and profile persist to the database for logging in later. Just the single AuthenticationService handles the centralisation of authentication, sessions, and RBAC concerns.

Browsing to Checkout: CustomerUser queries ProductCatalogue to view existing Product instances. Items are added to ShoppingCart, which validates availability and pricing through the catalogue. On checkout, the cart creates an Order that captures an immutable snapshot of items, prices, and delivery details from the CustomerUser, then persists itself. Order then creates an Invoice, which calculates GST and totals and stores itself. All writing instances target the database at cart to order, order to invoice, and finally payment, providing a full audio trail.

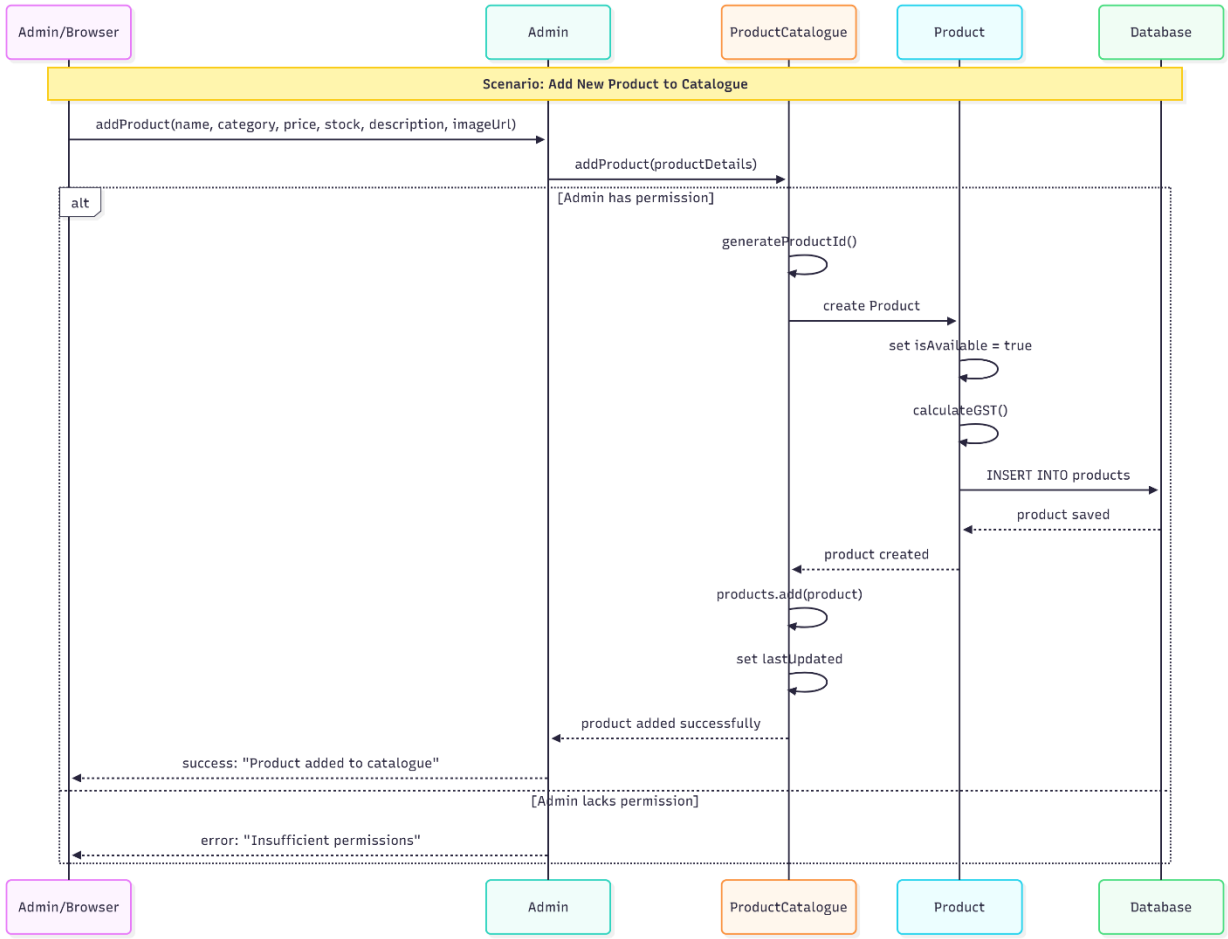
Payment and Completion: CustomerUser submits payment details. Order creates a Payment that processes the transaction and persists it. On authorisation, Payment is confirmed, Order is confirmed, Invoice is set to paid, and the catalogue updates product stock. On failure, Payment and Order record the failed status and the UI reports recovery or alternative options. Every state change writes to the database at the payment and order layers.

Fragments: The *loop* for Add Items repeats cart operations until the user stops. This captures iterative add / update patterns cleanly and accurately reflects the process of browsing that a customer would perform. The *alt* for Stock Check splits on sufficient vs. insufficient stock to enforce inventory constrains and signal

OutOfStock conditions. The *alt* for Cart Validation branches where any field is invalid to force the customer to correct any issues before an Order can be created. The *alt* for Payment Authorised branches into mutually exclusive outcomes based on a valid vs. invalid payment, performing actions based on the outcome such as throwing an exception, or proceeding with the order and updating the stock count.

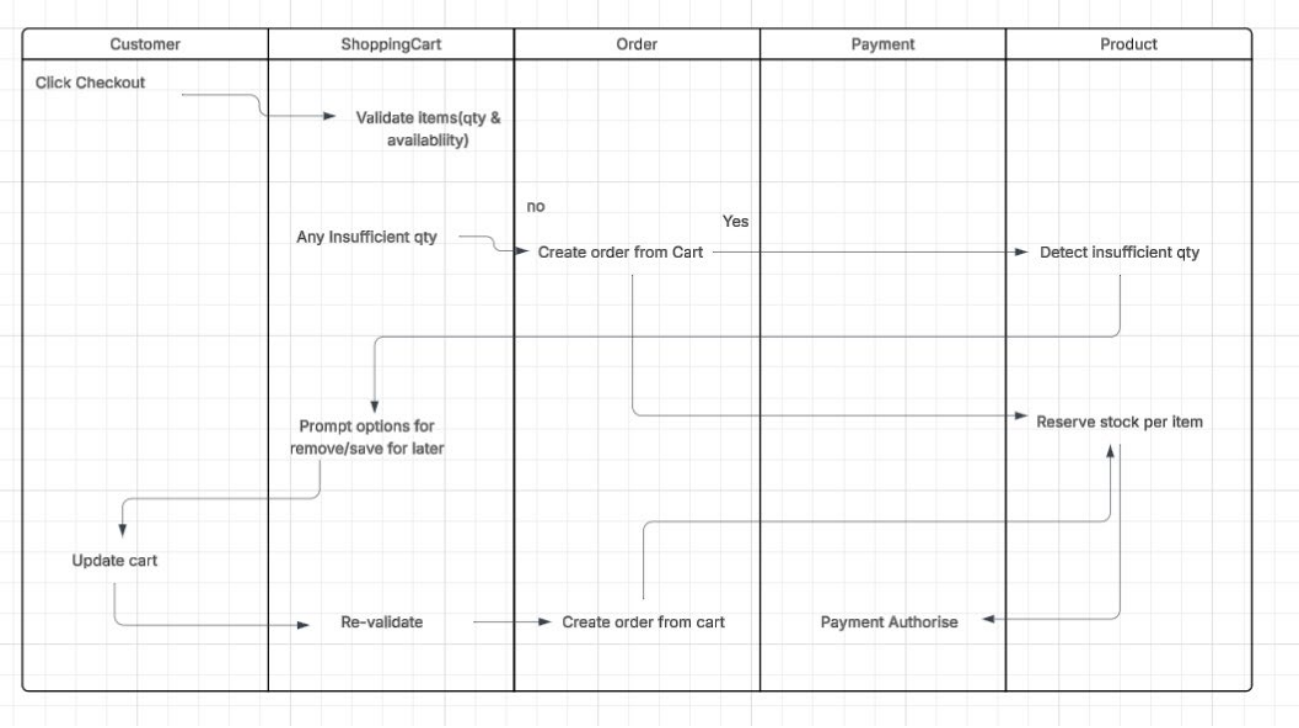
7.2. Product Catalogue Management

This situation showcases how a product is added to the catalogue.



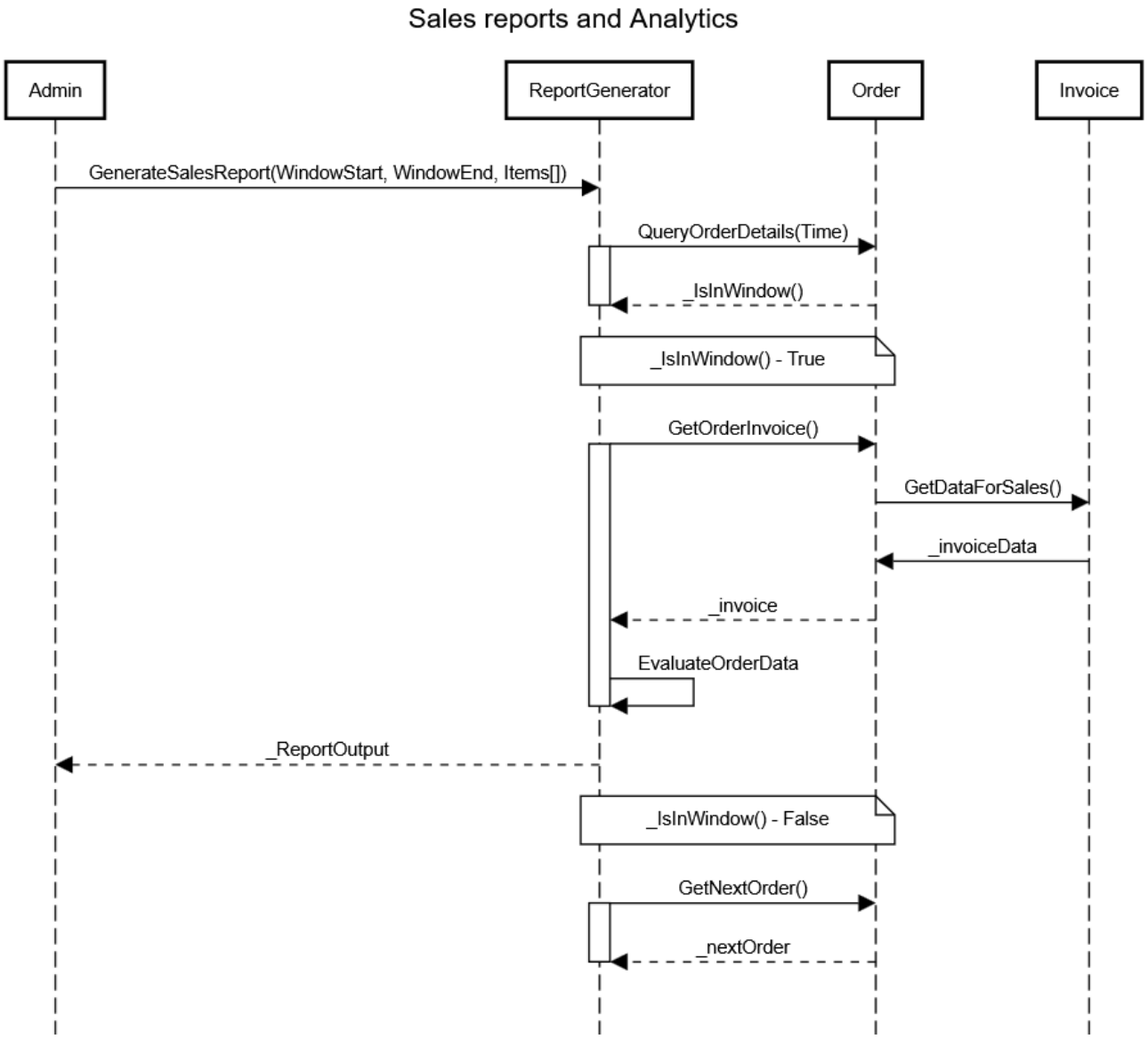
This sequence diagram shows an action within Product Catalogue management to illustrate the process of an admin adding a new product to the catalogue. It begins with the admin submitting product details through the interface, which are passed to the product catalogue system. If the admin has the necessary permissions, then the system generates a unique ‘productId’, creates the product object, calculates GST, and stores the product in the database. Once saved, the product is added to the active catalogue, and the admin receives confirmation. If the admin is lacking permissions, the system returns an error message notification instead displayed by the alt fragment.

7.3. Order Processing and Payment



This diagram shows the checkout flow. The customer Click Checkout, then ShoppingCart runs Validate items (qty & availability). If Any Insufficient qty, the cart Prompt options for remove/save for later; the customer Update cart, and the cart Re-validate. When there is no insufficiency, Order Create order from cart, Product Detect insufficient qty (final check) and Reserve stock per item, and Payment performs Payment Authorise. The flow confirms that orders are only created when quantities are acceptable, users get a clear recovery path when they aren't, stock is reserved before payment, and payment is authorised once.

7.4. Sales Reporting and Analytics



This sequence diagram shows the process to generate a sales report for a specific window. The admin provides a specific time window, and can specify items to be checked. The report generator queries all orders to check if they are in the window and collates the invoices of any orders that are within that window. It then evaluates the invoices based on metrics and returns the formatted report to the user.