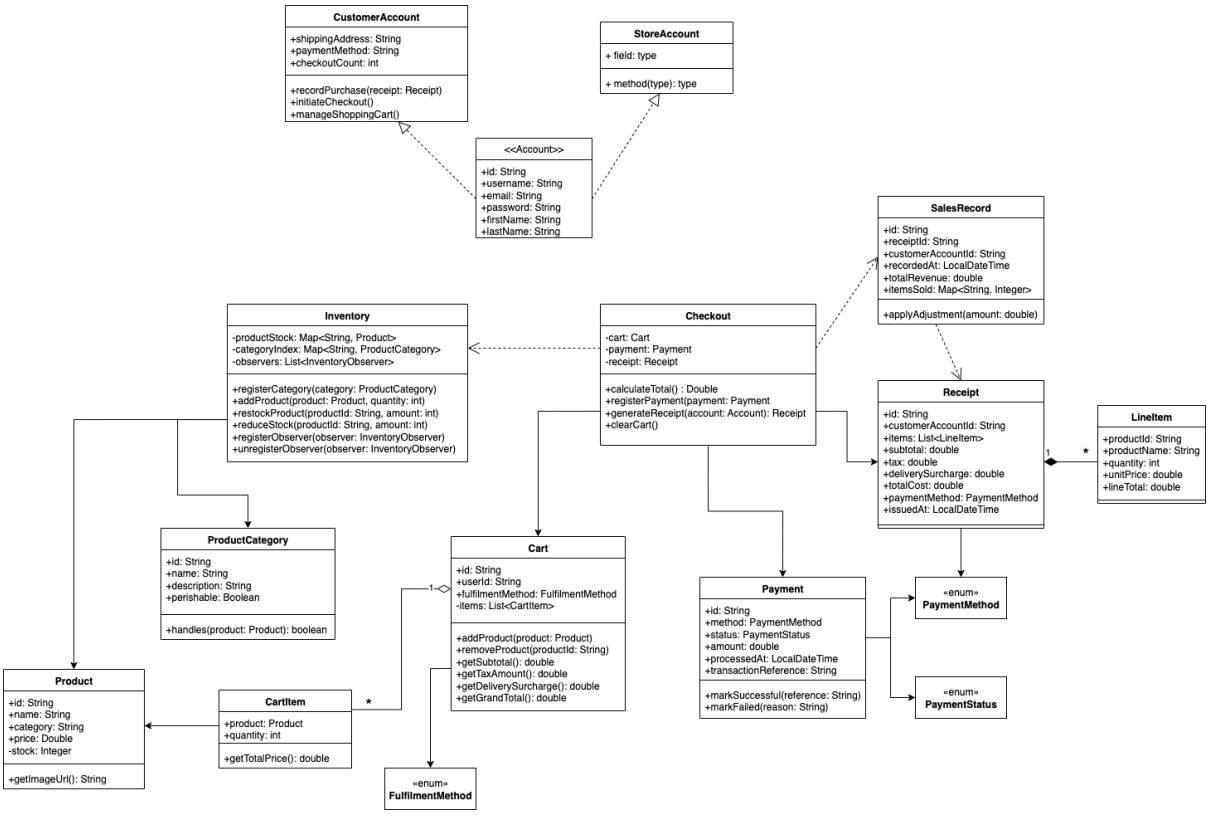
Object Design
Implementation & Reflection

Executive Summary

The Online Convenience Store system's final object design implementation and reflection are shown in this report. The project's goal was to turn the design from the Object Design document into a fully functional, object oriented system that lets management and employees handle inventory, goods, and sales data while letting customers explore products, manage carts, finish purchases, and produce receipts.
Java, JavaScript, HTML/CSS, and JSON for lightweight data handling were used in the implementation of the final system. For maintainability and a clear division of responsibilities, the implementation follows the MVC layered architectural patterns. The setup of user accounts, product browsing, shopping cart management, checkout, receipt production, and administrative reporting were all satisfied. With no actual financial transactions, the payment system was simulated with an assumed-successful confirmation. User tasks 1 through 7 were successfully implemented into the current system.

(I) Detailed Design, with Explanation and Justification of Changes and Non-Changes to Assignment 2 Design

1.1 Updated Class Diagram



Note: classes which were implemented as to adhere to a particular pattern (such as the Factory Method pattern) were omitted from the diagram. Classes which are not a part of the Model layer were also omitted for posterity.

Inventory is the singleton aggregate that owns stock state. It keeps a list of InventoryObservers. Observers implement InventoryObserver so they can react to those events. Inventory stores Product instances keyed by id.
A Product carries the category name, and ProductCategory checks if a product belongs to it. Categories are descriptors; products are the real stock items.
Cart (renamed from ShoppingCart) is the shopping basket root. It is composed of many CartItems. Each CartItem references the Product being purchased and tracks its quantity and line total. The cart aggregates price calculations using the items. A cart tracks whether the order is for pickup or delivery using FulfilmentMethod, rather than an attribute as was stated in the Object Design doc.
Account is the abstract base class for users. CustomerAccount adds checkout-facing info, while StoreAccount adds management metadata. Both subclasses inherit identifying fields from Account.
Checkout orchestrates a single transaction using the Builder pattern. It reads totals from a Cart, registers a Payment, builds a Receipt, and - through its services - triggers stock

updates in Inventory and logs the transaction in SalesRecord, then clears the cart on success.
Payment captures the tender record. It stores the selected PaymentMethod, current PaymentStatus, and has helpers for errors. Status and method are simple enums.
Receipt is the proof of purchase. It references the customer account id, stores the payment method used, totals, and a list of LineItems that summarise each product sold.
SalesRecord is built from a Receipt to support analytics - as opposed to the asynchronous design in the Object Design doc. It records totals, items sold, and the customer account id.
The service implementing sales records also observes Inventory so it can track stock changes and maintain reports.

1.2 Description and Justification of Detailed Design

The updated design adheres to the original system requirements and design goals relatively closely, with some modifications made to ground the design in pragmatic development choices.

Core entities, such as Account, Inventory, and Checkout remain lowly coupled from the infrastructure code. Controllers and services control persistence and orchestrate domain interactions, maintaining cohesiveness and testability. This is demonstrated in the extensive unit tests which were implemented for both domain model and critical service classes. The design satisfies the requirement for maintainable object design with clear responsibility boundaries, further exemplifying the Responsibility-Driven Design outlook of the development process.
The updated design was also developed with extensibility in mind - the builder and factory method patterns were successfully integrated into the solution.

The primary reasons for the updated structure were to manage the trade-offs between quality attributes, as well as keeping the main constraint - time - in mind. This was done to achieve encapsulation, dynamic reusability of code, and cohesion in the final implementation, while managing the satisfaction of the intent of Assignment 2's requirements.
Some examples of why the design shown was structured in such a way include:

- Adherence to patterns
    - e.g. Keeping Inventory as a singleton with an observer list. This was done to encapsulate the stock map, preventing random mutation and allowing us to plug in observers without changing the code of the initial Inventory class.
- Hierarchical intent
    - e.g. Account defining identity and credentials. Customer/StoreAccount add their own behaviour, this was done to maximise reusability of the code, maintaining cohesion.
- Services and repositories
    - e.g. Services wrap orchestration *and* persistence. The domain objects stay agnostic to persistence, while services collaborate with each other and reuse repositories. This was done to maximise testability, as we used Mockito to mock repositories.

    - e.g. Repositories were designed to expose narrow interfaces. Services and controllers should *not* reach into MongoDB APIs, which promoted separation of concerns and encapsulation of infrastructure concerns in the architecture.

1.3 Changes and Non-Changes (Class Level)

Classes that were removed:

- CashDetails
- CardDetails
  These classes were removed due to the lack of a need to implement further details in terms of PaymentMethod.

Classes that were added:

- PaymentStatus
- FulfilmentMethod
- LineItem (nested in Receipt)
- CartItem* (an extension of Product)
- Observer classes (InventoryEvent, InventoryObserver)
  Added to easily capture inventory state and management thereof.
- Factory classes (AccountFactory, AccountType)
  Added at the model level to adhere to the objective of implementing the Factory pattern.

- *Service classes*
  Added to orchestrate persistence, checkout, and observer updates. Not directly specified in Assignment 2; was not thought of at the time.
- *Controller classes*
  Added to expose REST endpoints aligned to the new services.
- *Repository classes*
  Added to interact with MongoDB (for persistence).

Classes that were modified (significantly or otherwise):

- Cart (previously ShoppingCart)
- Inventory
- Account (and its children, Customer/StoreAccount)
- Payment and PaymentMethod* (still mock)
- ProductCategory

CartItem was left unchanged as it was already a small object that paired a Product with a quantity to a Cart, with few other behaviours.
The AccountType enum was kept unchanged to promote extensibility, as new types of accounts can easily be added later with a single enum entry.
Product was also left mostly unchanged, as the description in Assignment 2 clearly defined its responsibilities and behaviours - as was needed for this implementation.
Checkout was left unchanged - its description in Assignment 2 captured all it needed.

All other classes either gained new responsibilities - such as Cart, or the Account children - or had to collaborate with the proposed and then introduced patterns or concepts, such as payments, receipts and observers.

1.4 Changes and Non-Changes (Responsibilities and Collaborators)

Account - The class was previously just a DTO; it now encapsulates shared identity/credential state, with services persisting it. It remains abstract so only concrete subclasses can be instantiated, clarifying ownership of behaviour.

CustomerAccount - now tracks checkout counts, receipt history, and timestamps (storeCustomerProfile, recordPurchase, etc.), fulfilling its obligation outlined in the Object Design doc to manage customer-specific purchase data.

StoreAccount - adds metrics counters (inventoryUpdates, reportsGenerated, etc.) to reflect managerial responsibilities, clarifying how store-side actions are tracked without overloading the base class.

AccountFactory - added for builder/factory, creates role-specific instances, and wires in extra fields (shipping address, access level) and remains the single entry point for account construction, reinforcing consistency.

AccountType - added as an enum class; it is the list of roles used by services and factories.

Cart - refined into the shopping aggregate: tracks fulfilment method, computes subtotal/tax/delivery/grand total, and exposes high-level mutations (addProduct, clear). This removes scattered price logic and keeps cart invariants inside one class.

CartItem - added as a minimal value object pairing Product with quantity and price math; unchanged because it already satisfied its single responsibility.

Checkout - remains the transactional coordinator. It relies on cart totals, registers a Payment, builds a Receipt, and knows when a cart can be cleared - matching the design document's sequence.

Inventory - strengthened to a Singleton aggregate with category registration and observer notification. Stock changes create InventoryEvents so dependent services stay in sync.

InventoryObserver & **InventoryEvent** - new helper types that implement the Observer pattern (Inventory now pushes structured change notifications).

Product - largely similar, but now keeps stock private (managed by Inventory) and exposes helper methods (e.g. getImageUrl). This change clarifies the boundary: Inventory controls quantities, Product is the presentation view.

ProductCategory - remains the domain class that captures category metadata, but extended with certain properties (description/perishable) and provides a handles() helper so both Inventory and accounts can reason about categories consistently.

FulfilmentMethod - new enum; formalises delivery vs pickup so tax/surcharge policy is explicit and type-safe throughout cart/checkout.

Payment, **PaymentMethod**, **PaymentStatus** - replaced the mock payment handling. Payment encapsulates status transitions, records timestamps/reference IDs, and stores the tender type using enums, giving a proper domain representation of payments. Easily extensible for full payment processing capabilities.

Receipt - added to fulfil the Client's requirement for proof of purchase. Contains totals, payment method, customer id, issue timestamp, and embedded LineItems; the builder centralises calculations so receipts are always consistent.

SalesRecord - Captures analytics data derived from receipts; fromReceipt() builds a record used for reporting, and the map of itemsSold supports best-seller queries. Also hooks into inventory observer logic through the service layer.

Controllers -> Services

REST controllers now defer to the new services. The controllers in the Object Design doc were largely left unspecified for flexibility in the development process, and they now coordinate request/response around the business services.

AccountService -> AccountRepository/AccountFactory

Accounts are no longer ephemeral. The service now looks up existing accounts by id/email, persists them via AccountRepository, and still creates instances through the builder/factory. These collaborations enable guest-to-customer transitions and reuse existing accounts for checkout.

CartService -> AccountService

Whenever a cart is mutated the interaction is recorded on the owning account. That cross-service dependency is new and keeps customer metrics up to date.

CheckoutService -> AccountService, CartService, ReceiptService, PaymentService, InventoryService, SalesRecordService

The checkout orchestration now depends on these services to retrieve the cart, process payments, issue receipts, decrement inventory, and persist sales records. This collaboration didn't exist before; it replaces the earlier isolated checkout logic.

Receipt and SalesRecord

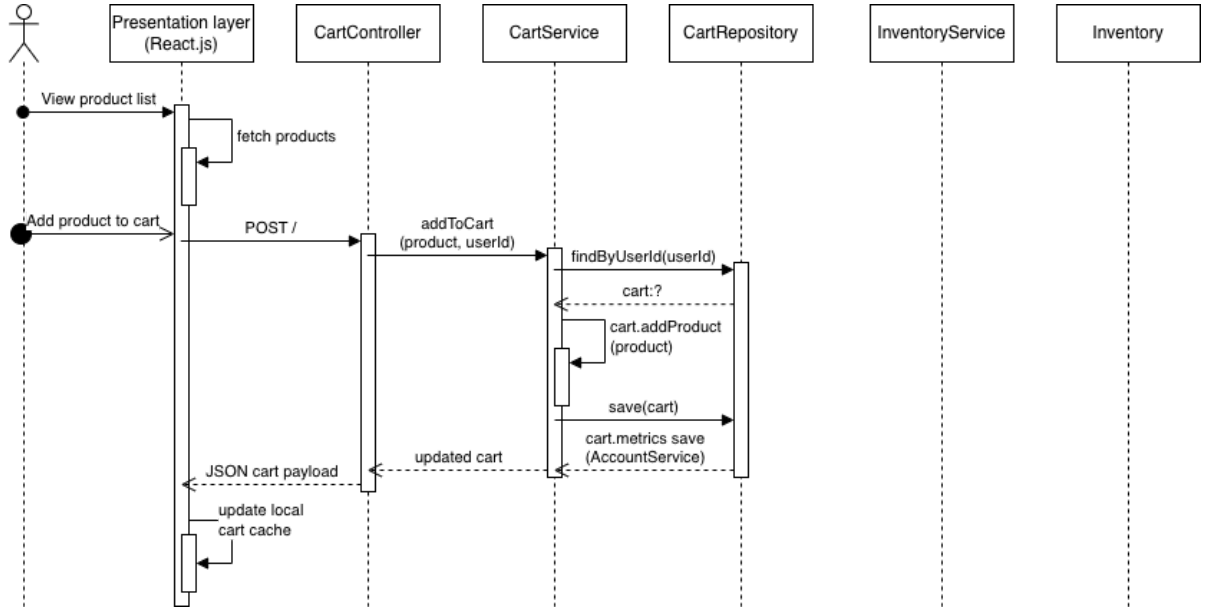
ReceiptService and SalesRecordService collaborate with the repositories to persist transactional data, and SalesRecordService also registers as an InventoryObserver so it stays aware of stock changes. This observer relationship is a new one.

Overall, the model layer moved from DTOs and placeholders to cohesive domain aggregates with explicit responsibilities: carts control order state, checkout coordinates the pipeline, payments/receipts document the transaction, inventory observes stock, and accounts hold persistent customer/store information.

1.5 Justification of Changes and Non-Changes (Dynamic Aspects)

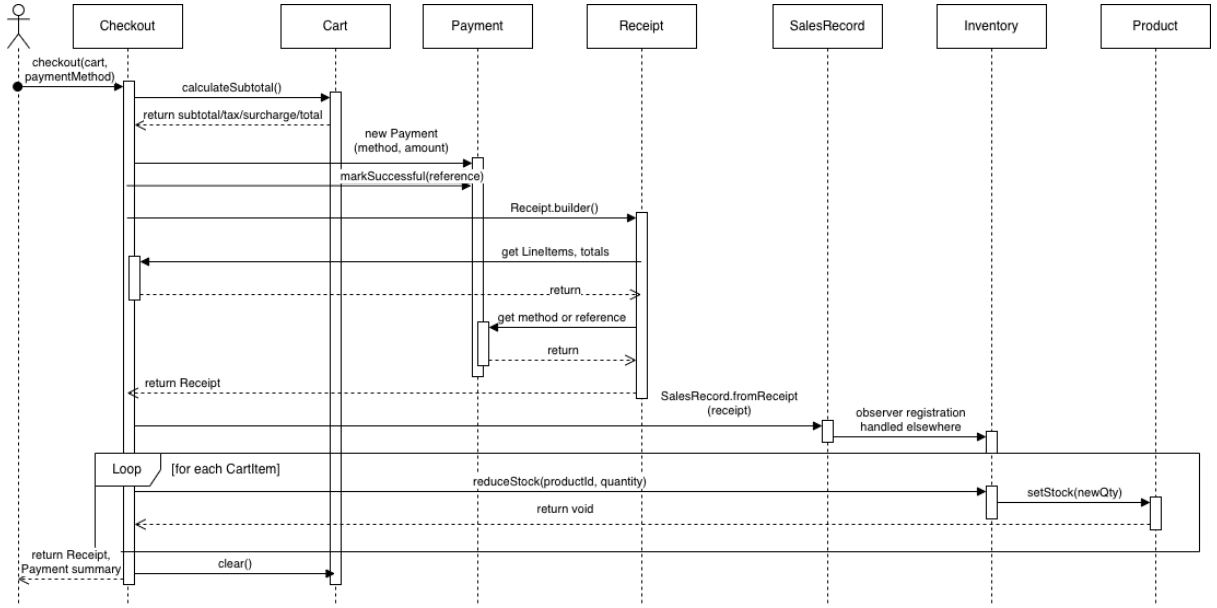
Updated sequence diagrams for related sections from Object Design Document

- **§7.1 Give customers the ability to browse the store catalogue online**
- **§7.3 Management of shopping cart**



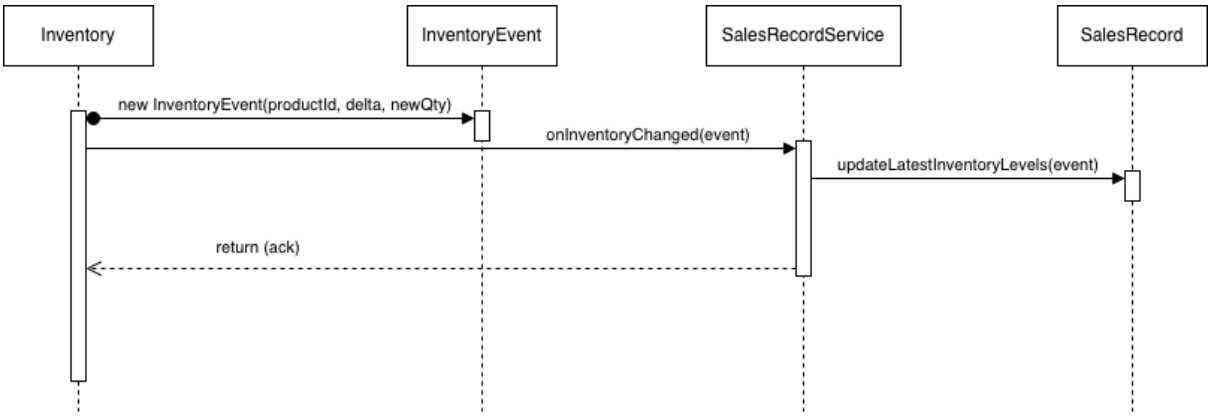
The Object Design doc showed Checkout talking directly to Payment, Receipt, Inventory, and SalesRecord. These steps are still executed in the same order, but each is now handed off to a dedicated service. This typa change makes the design more cohesive and persistent while keeping the overall flow intact, as well as keeping separation of concerns for RDD. The original diagram assumed a customer account was already present. The implementation explicitly resolves or creates the account via AccountService before checkout. The Object Design doc implicitly tied the cart to the customer. A cartUserId was added so that guest carts persist before an account exists. During checkout, the system falls back to the account id when available. If not, a new account id is generated for view only in the DB (it's still treated as a unique id). This satisfies real UX requirements without changing the rest of the sequence.

- **§7.2 Generate receipts for customers**



Here, the original “ShoppingCart” is split into Cart + CartItem and logic was added for fulfilment/tax/delivery. This keeps all pricing rules inside the aggregate and supports both pickup and delivery, but behaviour is unchanged: customers still add/remove items and the cart computes totals as illustrated in the above 2 diagrams. The design doc treated payment classes as mocks; now they are consolidated into Payment plus enums, but sequence-wise Checkout still records a payment method, marks success, and passes the record into the receipt.
The flow mirrors the Object Design doc; determine cart contents, process payment, generate receipt, update inventory/sales, and clear the cart. Each step was pushed into dedicated services to improve cohesion and persistence. The dynamics stayed the same, but collaboration is more explicit.

- **§7.4 Allow management to view basic sales statistics**



The Object Design doc had Checkout decrementing Inventory. We still do it, and in addition Inventory now emits events to observers (e.g. the analytics service). This is an implementation enhancement that keeps the original behaviour while enabling real-time updates.

(II) Discussion of Assignment 2 Design

2.1 Good Aspects of Assignment 2 Design

Assignment 2's initial design offered a solid structural basis for implementation. Each class had distinct, well defined roles and collaborators thanks to the application of our Responsibility Driven Design (RDD). Scalability and reusability were encouraged by the use of design patterns like Factory, Observer, and Singleton. The system was straightforward to expand due to its low coupling across layers and high cohesiveness across related components (such as Product, Inventory, and Checkout). The team was able to implement the system effectively and with little uncertainty because of the documentation's clear explanations of workflows and presumptions.

2.2 Missing Aspects from Original Design

In the Object Design doc, some of the functionality outlined in the requirements were either not fully developed or just partially documented.

- Task 1’s email verification and SSO (Single Sign On) components were conceptually created but not translated into class level or dynamic designs
- Neither the class design nor the sequence diagrams included the address validation and pickup scheduling logic from Task 3
- The high-level architecture was not considered in the original design. This
- The way UI elements interacted with backend processes had a few minor gaps that needed to be fixed during writing

To gain complete functioning during implementation, these missing components had to be carefully reinterpreted.

2.3 Flawed Aspects of Original Design

The original design was good overall, however there were few things that could have been done better:

- High-level architectural considerations were not made in the original design. Initially, this was thought to be an issue, however after choosing a framework (Java, SpringBoot) for the backend, and the abundance of information regarding patterns for that framework, the choices were easy to make - MVC + Layered architecture.
- Certain tasks were duplicated in other classes (eg, overlapping data validation in Checkout and Receipt)
- There was some ambiguity during integration because the data flow between the front end and the back end wasn't well defined
- Some classes, like Payment, were too abstract to implement directly without rethinking about their logic
- There were conceptual descriptions of error management and validation procedures but no implementation on the specifics

During development, these problems were resolved by creating more transparent data handling channels, streamlining dependencies, and improving class responsibilities.

2.4 Level of Interpretation Required

Particularly for data storage and UI synchronisation, a moderate reinterpretation was required. We developed the missing controller functions for checkout validation and implemented local JSON data simulation for persistence.

Payments were listed as “mock”, but left implementation open due to lack of CRC or sequence diagram exploration. Rather than fake classes for each method related to Payment, a single payment entity was created with enums to record PaymentMethod and PaymentStatus.

Controllers in the Object Design doc interacted directly with model classes. In the implementation services were inserted for persistence, cross-cutting metrics, and transaction boundaries. That doesn’t change the behaviour but adds necessary glue absent from the original diagrams.

(III) Lessons Learnt

3.1 Key Lessons from Assignment 2 to 3

- **Object Oriented Design**: Early usage of encapsulation and abstraction reduced complexity later in the project, making it simpler to expand and maintain the system.
- **Design Patterns**: The Factory and Observer patterns, which enabled effective object generation and real time updates between modules, enhanced code scalability and readability, and there were many resources online which closely aligned with our use case. This helped immensely in the development process.
- **Error Handling**: Consistent error responses across components were ensured by centralised exception handling. This was a crucial step in both the design and development process, as it also facilitated debugging and decreased redundancy.
- **Teamwork**: Regular pull requests and GitHub commits, which offered transparency, version control, and easy progress monitoring, improved teamwork.
- **Future Improvement**: If database logic had been prototyped sooner, there might have been less rework. In subsequent revisions, we would incorporate a lightweight ORM or database API early in the development process and schedule testing for dynamic interactions earlier.

If it were to happen again, we would schedule testing for dynamic interactions sooner and implement a lightweight ORM or database API early.

(IV) Architecture Style(s)

4.1 Architecture Overview

The finished implementation uses a **Layered, MVC** architecture, clearly separating the following:

- **Model Layer**: The model consists of Java entity classes like Product, Inventory, and SalesRecord. These manage the application's data state, validation, and business logic.
- **Controller Layer**: CartController and CheckoutController are examples of Java and JavaScript classes that handle user input and application logic. They provide a bridge between the backend logic and the frontend.
- **View Layer**: React is used to build the frontend, with HTML, CSS, and interactive JavaScript components supporting it. When users interact with products, carts, and checkout features, React swiftly refreshes the DOM, handles state changes, and dynamically renders the user experience.

The system's structure, readability, and flexibility are all improved by this distinct division of responsibilities, which guarantees that each layer serves a distinct function.

4.2 Rationale for Chosen Architecture

The system's primary objectives of scalability, modularity, and maintainability are supported by this architectural style, which is why it was selected:

- **Scalability**: It is possible to update or enlarge each layer separately without impacting the others. For instance, without altering the React-based frontend, new product categories or more payment methods may be introduced to the Model and Controller levels.
- **Modularity**: Different components of the system can be built and tested independently thanks to the distinct split between Model, View, and Controller. Java controllers regulate data flow and application logic, while React components handle presentation logic.
- **Maintainability**: Future improvements, including UI redesigns or backend modifications, can be implemented with little disturbance because the architecture isolates responsibilities. Because each module has a distinct job, debugging and improvements are made easy.
- **Reusability**: It is possible to reuse controller methods, Java entities, and React components across several pages or even future applications (such an admin dashboard or mobile version).
- **Performance and Consistency**: React effectively updates only those UI elements that change, enhancing efficiency and preserving consistency. In the meanwhile, UI changes have no effect on the backend logic or data access.

In general, a clean, scalable, and maintainable system that satisfies both functional and non functional needs is guaranteed by the layered MVC structure with React as the View layer.

4.3 Implementation Reflection

The implementation successfully illustrates how the layered MVC architecture functions in real world scenarios. The user interface is managed by the React based View layer, which automatically renders elements like checkout forms, shopping carts, and product lists. Every time the data in the Model layer changes, its state management function makes sure that UI updates happen smoothly. Built in Java and JavaScript, the Controller layer controls communication between the backend logic and the React components, verifying inputs and handling user actions like completing checkout or adding items to a cart. The Repository layer employs JSON files to store data and ensure consistency between sessions, while the Model layer specifies the data structures and business logic for entities like Product, Inventory, and CustomerAccount.

This architecture increased development efficiency and decreased merge conflicts by enabling various team members to work independently on the frontend, backend, and logic levels. For real time updates like live cart totals and stock availability, the interaction between React and the controllers worked especially well. Overall, the implementation demonstrated that the layered MVC method with React offers a solid basis for long term maintainability (through extensive documentation), scalability (through pattern implementations), and modularity (through the decoupled design).

(V) Implementation: Source Code Quality

All source code can be seen in the GitHub repository:
https://github.com/j-armstrong535/Online-Convenience-Store

5.1 Coding Standards and Style

The coding conventions outlined in *Object Design – §1C* were consistently followed throughout the development of the website. All Java, JavaScript and React components adhere to standard ES6+ syntax and naming conventions, such as using camelCase for variables and functions, and PascalCase for React component names. Proper indentation (two spaces per level) and consistent bracket placement were maintained for readability and maintainability. Inline comments were used to clarify key logic sections, while block comments described component purpose and functionality. Additionally, CSS class names follow a clear, descriptive format that reflects the associated UI element, ensuring consistency between structure and style.

5.2 Code Structure and Readability

The application codebase was designed with modularity and readability in mind. Each feature of the website (e.g., Navbar, Product Display, Cart Management, and Authentication) is encapsulated within separate React components and organized into logical directories such as 'components/, pages/, and services/'.

Reusable functions - such as fetchCartItems, readCachedCart - were placed in a shared '*services*' module to promote code reuse and reduce duplication. Logical separation of concerns was achieved - UI logic resides in components, while data operations and network requests are handled by service modules.

Basic exception handling was implemented using try/catch blocks within asynchronous functions to manage potential API errors gracefully. Informative console logs and controlled error displays improve debuggability during development. In addition, concise JSX formatting and meaningful variable names enhance code readability and make the structure self-documenting.

5.3 Object-Oriented Principles Applied

The backend of the system was implemented using **Java Spring Boot**, which inherently supports object-oriented programming (OOP) concepts through its modular and layered architecture. The directory structure (controller, service, repository, model) demonstrates strong adherence to encapsulation, inheritance, polymorphism, and cohesion.

Encapsulation:
Encapsulation is evident across the entire backend architecture. The model package contains entity classes such as Account, Cart, Product, Payment, and SalesRecord, each encapsulating their attributes (e.g., id, name, price) and behaviors through getter and setter methods. Access modifiers (e.g., private, protected) are used to restrict direct data access, ensuring that data can only be manipulated through defined interfaces. The service layer (e.g., InventoryService, CheckoutService) further encapsulates business logic by providing dedicated methods that interact with repositories while hiding implementation details from controllers.

Inheritance:
Inheritance is applied through class hierarchies in the model package. For instance, CustomerAccount and StoreAccount extend from the abstract base class Account, inheriting shared fields and methods such as authentication credentials, balance, or account type. This reduces code duplication and improves maintainability. Similarly, the AccountFactory class implements a **Factory design pattern**, leveraging inheritance and abstraction to create specific account types dynamically.

Polymorphism:
Polymorphism is used to promote flexibility in object behavior. Abstract classes and interfaces (such as InventoryObserver and FulfilmentMethod) allow multiple implementations depending on context. For example, different fulfilment strategies (e.g., pickup vs. delivery) can be executed through a common interface. Method overriding also occurs where subclasses redefine parent methods (e.g., CustomerAccount and StoreAccount providing specific implementations for inherited methods). Additionally, Spring's dependency injection framework enhances polymorphism by allowing controllers to reference service interfaces rather than concrete implementations, enabling easier testing and code substitution.

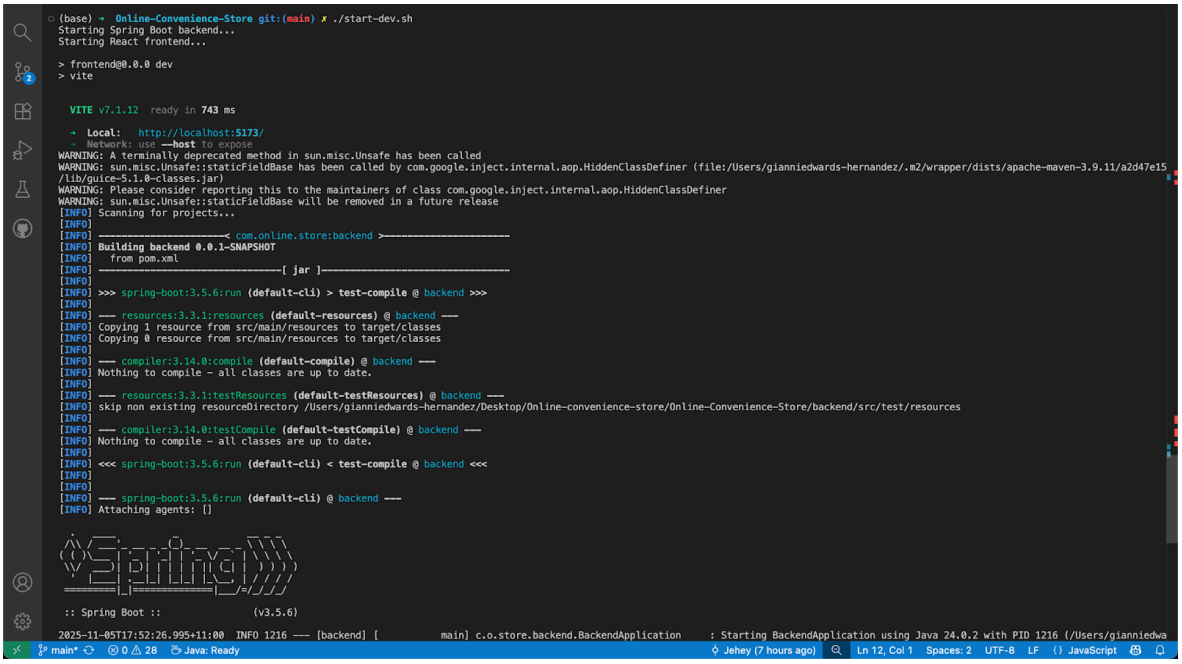
Cohesion:
The backend architecture demonstrates high cohesion by assigning clear and specific responsibilities to each layer:

- The **controller** classes (e.g., ProductController, CartController, UserController) handle HTTP requests and route them to the appropriate service methods.
- The **service** classes contain the business logic, performing operations like product updates, cart management, and transaction processing.

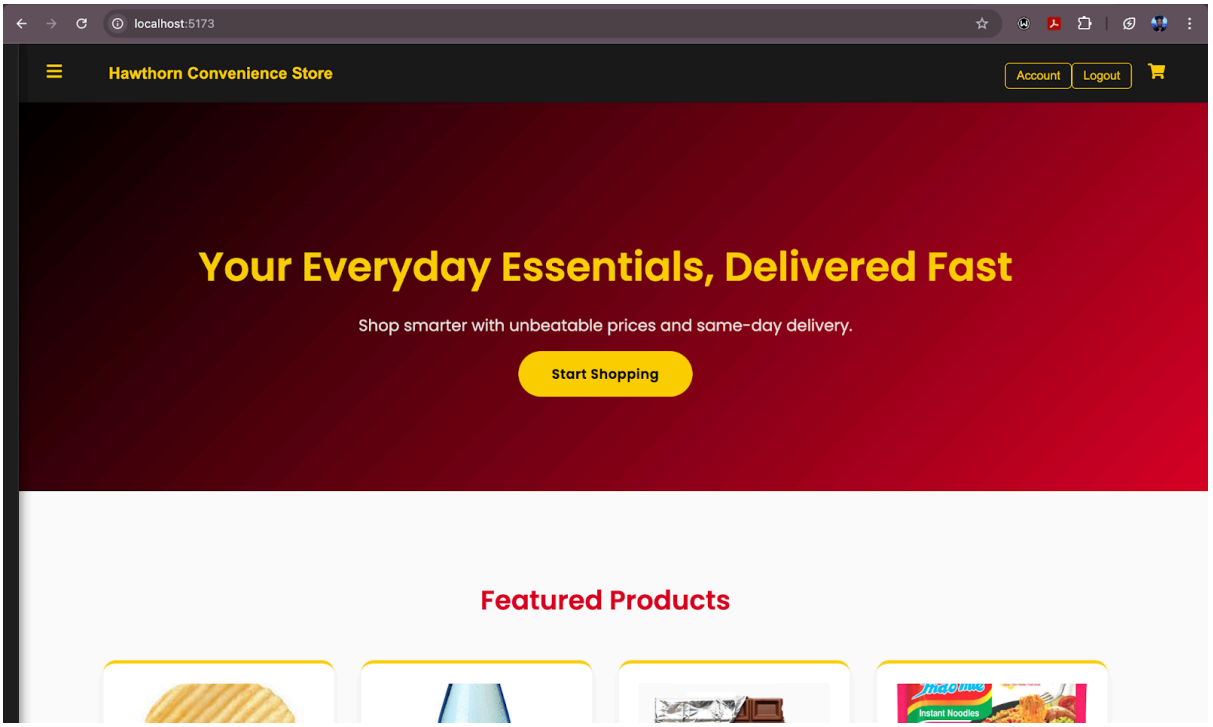
- The **repository** interfaces extend Spring Data JPA repositories to handle database persistence cleanly, maintaining separation from business logic.
- The **model** classes strictly represent data entities, maintaining a focused responsibility.

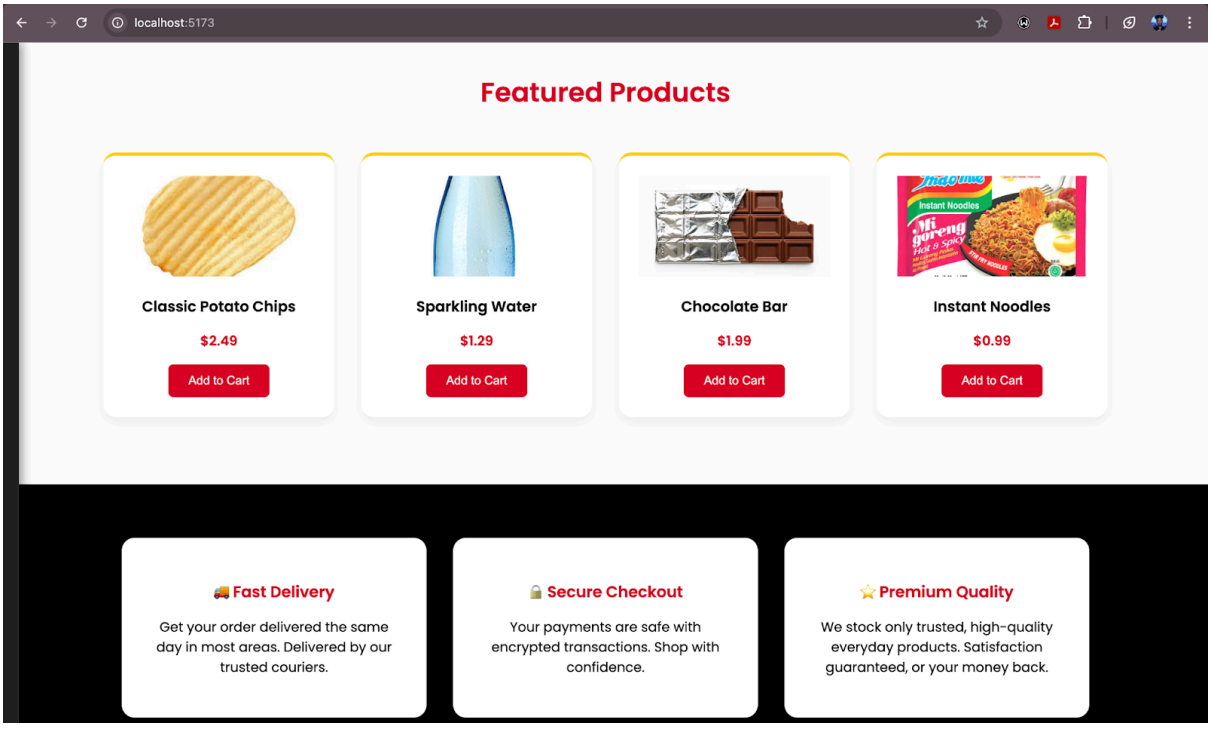
(VI) Implementation: Compilation and Execution Evidence

6.1 Evidence of Successful Compilation

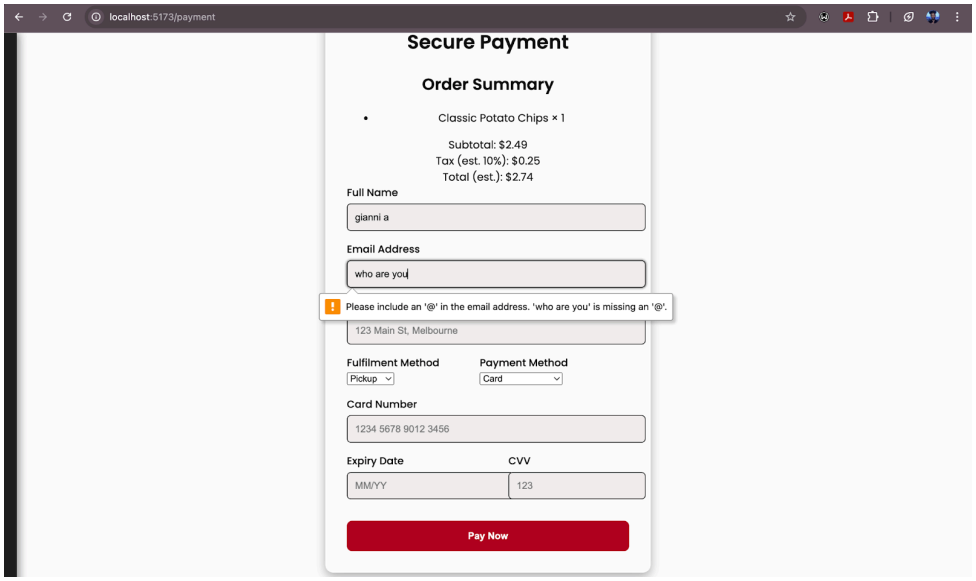


6.2 Home Screen Illustration


localhost:5173
Hawthorn Convenience Store
Account
Logout
Your Everyday Essentials, Delivered Fast
Shop smarter with unbeatable prices and same-day delivery.
Start Shopping
Featured Products


localhost:5173
Featured Products
Classic Potato Chips
$2.49
Add to Cart
Sparkling Water
$1.29
Add to Cart
Chocolate Bar
$1.99
Add to Cart
Instant Noodles
$0.99
Add to Cart
Fast Delivery
Get your order delivered the same day in most areas. Delivered by our trusted couriers.
Secure Checkout
Your payments are safe with encrypted transactions. Shop with confidence.
Premium Quality
We stock only trusted, high-quality everyday products. Satisfaction guaranteed, or your money back.

6.3 Data Input and Validation Process



Example of data validation during the Checkout process

6.4 Input Processing and Business Logic Execution

This is demonstrated in §6.7 below.

6.5 Sample Outputs

This is demonstrated in §6.7 below.

6.6 Exit and Test Screens

Tests, exit screens, confirmation screens and such are demonstrated below, and in §6.7

```
(base) ➜  backend git:(main) ✗ ./mvnw test
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 7.495 s -- in com.online.store.backend.BackendApplicationTests
[INFO] Running com.online.store.backend.model.SalesRecordTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.038 s -- in com.online.store.backend.model.SalesRecordTest
[INFO] Running com.online.store.backend.model.CheckoutTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.041 s -- in com.online.store.backend.model.CheckoutTest
[INFO] Running com.online.store.backend.model.InventoryTest
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.042 s -- in com.online.store.backend.model.InventoryTest
[INFO] Running com.online.store.backend.model.ReceiptBuilderTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.022 s -- in com.online.store.backend.model.ReceiptBuilderTest
[INFO] Running com.online.store.backend.model.PaymentTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.008 s -- in com.online.store.backend.model.PaymentTest
[INFO] Running com.online.store.backend.model.StoreAccountTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.008 s -- in com.online.store.backend.model.StoreAccountTest
[INFO] Running com.online.store.backend.model.AccountFactoryTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.032 s -- in com.online.store.backend.model.AccountFactoryTest
[INFO] Running com.online.store.backend.model.CustomerAccountTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.011 s -- in com.online.store.backend.model.CustomerAccountTest
[INFO] Running com.online.store.backend.model.CartItemTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.014 s -- in com.online.store.backend.model.CartItemTest
[INFO] Running com.online.store.backend.model.CartTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 s -- in com.online.store.backend.model.CartTest
[INFO] Running com.online.store.backend.service.ProductCategoryServiceTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.858 s -- in com.online.store.backend.service.ProductCategoryServiceTest
[INFO] Running com.online.store.backend.service.ProductServiceTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.215 s -- in com.online.store.backend.service.ProductServiceTest
[INFO] Running com.online.store.backend.service.CartServiceTest
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.307 s -- in com.online.store.backend.service.CartServiceTest
[INFO] Running com.online.store.backend.service.ReceiptServiceTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.206 s -- in com.online.store.backend.service.ReceiptServiceTest
[INFO] Running com.online.store.backend.service.SalesRecordServiceTest
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.223 s -- in com.online.store.backend.service.SalesRecordServiceTest
[INFO] Running com.online.store.backend.service.CheckoutServiceTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.267 s -- in com.online.store.backend.service.CheckoutServiceTest
[INFO] Running com.online.store.backend.service.PaymentServiceTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.126 s -- in com.online.store.backend.service.PaymentServiceTest
[INFO] Running com.online.store.backend.service.AccountServiceTest
[INFO] Tests run: 18, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.438 s -- in com.online.store.backend.service.AccountServiceTest
[INFO] Running com.online.store.backend.service.InventoryServiceTest
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.052 s -- in com.online.store.backend.service.InventoryServiceTest
[INFO] Running com.online.store.backend.service.AnalyticsServiceTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.036 s -- in com.online.store.backend.service.AnalyticsServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 104, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  14.856 s
[INFO] Finished at: 2025-11-07T12:56:42+11:00
[INFO] ------------------------------------------------------------------------
```

Console output of tests running

6.7 Multiple Scenario Demonstrations

Tasks 2-7, §6.4, §6.5, §6.6 demonstration: https://www.youtube.com/watch?v=T5blAZiYxUI
Task 1 was also completed successfully, however the video does not cover it, but the video does briefly show and speak on the different types of accounts.
If you want to clone the full app onto your PC, here is the GitHub link, and password for the MongoDB connection
mongodb+srv://StoreAdmin:MyStorePass1@cluster0.y0pfcm7.mongodb.net/convenience_store

Instructions for required dependencies and running a quick-start script can be found in the README.md file in the repository root.

References

- Airbnb. (2023) *Airbnb JavaScript Style Guide*. GitHub. Available at: https://github.com/airbnb/javascript.
- Bass, L., Clements, P. and Kazman, R. (2021) *Software Architecture in Practice*. 4th edn. Boston: Addison-Wesley Professional.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley.
- Mozilla Developer Network (MDN). (2025) *JavaScript Guide*. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide.
- React. (2025) *React Documentation – Components and Props*. Meta Open Source. Available at: https://react.dev/learn.
- W3Schools. (2025) *JavaScript Coding Conventions*. Available at: https://www.w3schools.com/js/js_conventions.asp.
- Pressman, R.S. and Maxim, B.R. (2020) *Software Engineering: A Practitioner's Approach*. 9th edn. New York: McGraw-Hill Education.
- Sommerville, I. (2020) *Software Engineering*. 10th edn. Harlow: Pearson Education.
- Baeldung. (2020, October 30). Www.baeldung.com. https://www.baeldung.com/.

Appendix: Assignment 2

Object Design Document

Executive Summary

The Online Convenience Store system was created to offer a digital platform that makes managing customer purchases, product details and sales records easier and more efficient. Traditional small scale retail stores currently confront a number of difficulties which include manual updating, fragmented data handling and inefficiencies in inventory control and order monitoring. This system offers an integrated, fully electronic solution in order to overcome these constraints. It guarantees that all inventory and purchase data is digitally kept, updated and retrieved, increasing accuracy, cutting down on redundancy and speeding up the entire shopping experience.

The object oriented design of the suggested system is described in this report, together with the architecture, class specifications and design patterns that serve as the framework for its execution. To accomplish modularity, scalability and maintainability, the design makes use of a Responsibility Driven Design (RDD) methodology which is backed by CRC cards, UML sequence diagrams and important design patterns including Factory, Observer and Singleton. In order to guarantee consistent and effective system behaviour, the document also covers design quality, system verification and the application of pattern based solutions. A strong, flexible and future ready framework that supports additional features and potential company expansion is provided by the final design.

1. Introduction

The high level Object Oriented Architecture for the Online Convenience Store system, as outlined in Assignment 1, is presented in this Object Design document. This design's objective is to act as a guide for the system's deployment in Assignment 3, guaranteeing a standardized framework that embodies the essential business features and quality standards demanded by the Customer.

The RDD approach is used in the suggested design, which aims to precisely outline each class's duties and interclass cooperation. Account administration, product browsing, cart updates, payment processing, receipt creation, inventory management, and administrative analytics are just a few of the crucial user functions that the system architecture facilitates. The design is platform agnostic and ignores lower-level issues like database design or user interface layout, instead emphasising the functions and interactions of the system's main components.

a. Outlook of solution

As suggested in Assignment 1, the system is meant to act as a hybrid solution that connects online and in-store procedures. Key domain entities have been modeled into object classes with well defined roles in order to achieve these goals. A UML diagram is used to illustrate these classes, which are displayed as CRC cards. In order to facilitate scalability, separation of concerns, and long-term maintainability, the system also incorporates design heuristics and patterns such as Factory, Observer, and Singleton.

b. Tradeoffs and object design

We made a number of choices during the design phase to keep the system straightforward, understandable, and simple to maintain. Some of the most important design trade-offs we took into account are listed below.

i. Naming conventions

This is demonstrated in 1C

ii. Boundary cases

All text entry fields, including emails, passwords, and usernames, are constrained by a set maximum character length within the User Interface (UI). This keeps input from overflowing and guarantees uniform validation throughout the system.

iii. Invalid username or password exception

Users must enter their email address and password when they want to log in. The system will raise an “InvalidCredentialsException” and notify the user to submit the right credentials again if they are entered incorrectly.

iv. Invalid data exception

When a user puts data in a field that does not have the appropriate format or type (such as letters in a field that only accepts numbers), this exception is triggered. The system gently manages the error and gives the user a useful message.

c. Documentation and guidelines for interface

Identifier	Rules	Examples
Classes	Class names need to be descriptive and capitalized in PascalCase. Every class needs to stand for a distinct responsibility.	CustomerAccount, PaymentHandler, ShoppingCart
Interfaces	Interface names should start with an uppercase letter, just like classes. Used while creating an implementation contract.	IPaymentProcessor, IDataExporter
Variables	The names of variables should be brief but significant. For clarity, if necessary, use lowercase words followed by uppercase ones.	currentUser, productList, cartTotal

d. Definitions, acronyms, and abbreviations

UI	User Interface
DB	Database
CRUD	Create, Read, Update, Delete
OTP	One Time Password
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation

2. Problem analysis

The Software Requirements Specification (SRS) contained an analysis of requirements, which identified both functional and quality needs for creating a suitable solution. The analysis outlined the main features the online program must have in order to satisfy the requirements of different use cases. The main features of the application are:

- Creating customer accounts
- Browsing the product catalogue
- Managing a shopping cart
- Checking customers out
- Generating proof-of-purchase
- Multiple payment methods
- Method of managing stock
- Generating basic charts and graphs of store statistics
- Giving management the ability to change product types

a. Assumptions

A1 Only physical products sold in the store will be available online.

A2 A product in this context refers to any item sold in the store (e.g. food, drinks, electronics, kitchenware, etc.).

A3 All products have a unique method of identification (SKU or barcode).

A4 All products have a name, category, description, manufacturer/supplier, and a standard price.

A5 A product may be purchased, but not returned.

A6 Each purchase involves exactly one receipt, one record of the sale (internally only), one customer, and may include multiple products.

A7 For a customer to make a purchase online, they must provide their name, delivery address, and payment details to be recorded by the system and optionally 3rd party payment processors.

A8 If a customer is new, they will be assigned a unique customer ID upon account creation.

A9 All employees, customers, products, transactions, and stock details will be registered in the system.

A10 All customers have a unique ID, name, address, and phone number.

A11 Once the details of a customer, salesperson, and product are registered, they will remain in the system for record-keeping and cannot be removed.

A12 The store sells approximately 100 products each day, up to 500 during promotional periods.

A13 A payment refers to a full payment during a checkout sequence.

A14 Upon each checkout sequence, a receipt is provided to the customer.

A15 The system resides in a secure environment so that customer payment information is protected and handled according to security standards.

b. Simplifications

- Information for all products will be stored in the same way in the database and treated the same in the application, regardless of category.
- The persistent data boundary may only be accessed via interacting with objects in the program. Direct access to the DB is not allowed.
- All customer records follow the same structure (name, address, phone number, customer ID) without variation.
- Discounts, promotions, and special offers are treated as simple price adjustments, rather than separate product types, and are denoted by a single property.

c. Design Justification

It is expected that the Online Convenience Store system is supported by a central database. The classes specified in this architecture serve as the main conduit between the database and the user facing features. This makes it possible for other applications to enhance or reuse the system without having direct knowledge of the database structure by guaranteeing that any class can access or edit data through a regulated and consistent interface. While

the Singleton Pattern is used to keep a single, restricted instance of crucial classes like database connectivity or inventory management, the Factory Pattern guarantees that object generation stays centralised and effective.

By assigning distinct tasks to every class, the design removed redundant data and achieves a degree of normalisation that is in line with the third normal form. Only the data required for each class's operation is kept up to date, accessing and processing related data is done through cooperation with other classes. This strategy promotes high cohesion and low coupling, which lowers maintenance complexity and enhances scalability. The systems responsiveness and integrity are strengthened by the Observer Patterns incorporation which also guarantees real time data synchronisation across connected elements like the inventory and the shopping cart.

d. Discarded class list

- ☐ Store

This class was discarded from the design as implementation of it, despite being tremendously useful from a domain-entity visualisation point of view, would lead to too many classes sharing similar responsibilities and properties. The responsibilities of a Store class, for example pricing of items and structures of product catalogues, are already owned by Product, ProductCategory, and Inventory. The responsibility-driven design approach would simply not support this kind class existing, especially as a god class, and instead supports distributing those responsibilities to the objects that have the information.

- ☐ Person

This class was not needed as it contributes no domain behaviour other than what the account types already encapsulate. It would either duplicate attributes across actors or become a place for data to stay. We are only keeping classes that actually do things.

- ☐ Employee

This class was discarded as any behaviour regarding staff members was already represented by StoreAccount.

- ☐ Customer

Similarly to Employee, the idea of representing a customer as a person in the domain adds no unique behaviours or responsibilities beyond what CustomerAccount already does/represents.

- ☐ Supplier

Discarded since it was decided that the integration of a supplier was out of scope for the proposed system. Inventory adjustments would be triggered either by checkout or staff. Adding this class would increase the complexity of multiple tasks/subtasks.

- ☐ InvoiceRecord

Discarded because it duplicates the responsibilities of Receipt and SalesRecord. Also since Supplier was discarded, there was no reason to keep this class.

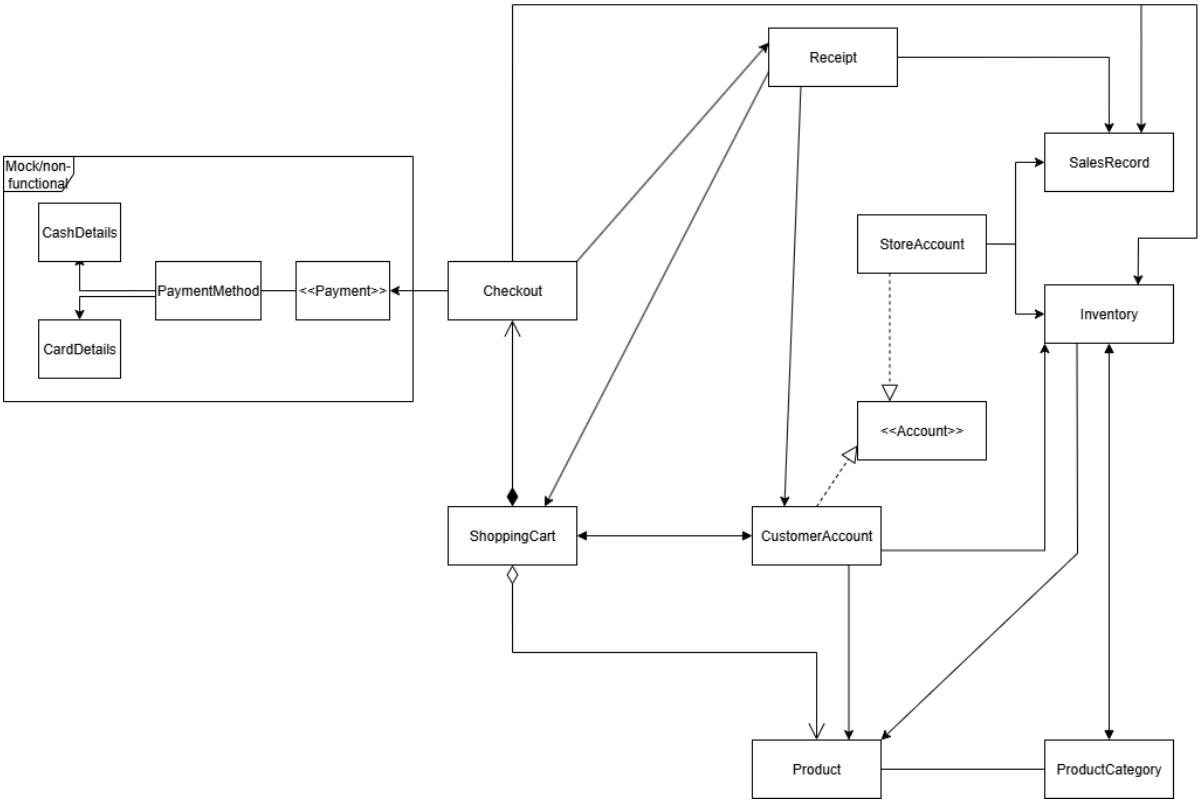
3. Candidate classes

a. Candidate class list

= Successful candidate
= Partially successful candidate (implemented for demonstrative purposes only)
= Unsuccessful candidate

- ☐ Store
- ☑ Product
- ☑ Inventory
- ☑ Receipt
- ☑ SalesRecord
- ☐ InvoiceRecord
- ☑ Account
  - ☑ CustomerAccount
  - ☑ StoreAccount
- ☑ ShoppingCart
- ☑ ProductCategory
- ☐ Person
  - ☐ Employee
  - ☐ Customer
  - ☐ Supplier
- ☑ Payment
- ☑ PaymentMethod
  - ☑ CashDetails
  - ☑ CardDetails
- ☑ Checkout

b. UML diagram



Following the problem analyses and reviews of the Requirements Specification, classes were identified via decomposition, with the solution having classes with high cohesion. The system is guided by RDD, assigning distinct responsibilities to domain classes and defining their collaborations explicitly. The system is entered via feature routes as opposed to a central menu - this was to prevent the introduction of a ‘god class’. Thin boundaries and control components handle navigation and request orchestration, while entity objects delineate business rules and invariants. This division keeps policy separate from navigation, maximises cohesion, and minimises coupling.

A CustomerAccount has 1 ShoppingCart. When the user proceeds to Checkout, and completes the checkout process, this process will create a Receipt, update Inventory and SalesRecord. 1 Product is associated with exactly 1 ProductCategory, and Inventory monitors levels of Product and ProductCategory.

For clarity in scope, the Payment area is a non-functional mockup used for illustrative purposes only, as no actual processing will be performed. Checkout simply records the selected method for the receipt. The Client may require this in the future, however this is out of scope for implementation currently.

Data persistence is treated as an infrastructural concern and is intentionally abstracted behind interfaces, so no class representing a database appears here.

c. CRC cards

i. Product

Class name: *Product* Superclass:	
A product represents a single item type in the store. A product is identified with a unique id, there cannot be more than a single product with a unique identifier.	
Responsibilities	**Collaborators**
Knows unique identifier, product name, price, use-by, batch no.	n/a
Knows product type (on special)	Inventory
Knows what kind of Product it is	ProductCategory

ii. Receipt

Class name: *Receipt* Superclass:	
Receipt serves as the official proof of purchase after a completed order.	
Responsibilities	**Collaborators**
Can generate a receipt after a successful checkout.	Checkout
Can retrieve purchase details from ShoppingCart and Payment.	Payment, ShoppingCart
Can calculate subtotal, delivery surcharge, tax, and total cost.	Payment, ShoppingCart
Can assign a unique receipt ID and issue date.	CustomerAccount
Can validate and confirm that payment was successful.	SalesRecord

iii. SalesRecord

Class name: *SalesRecord* Superclass:	
SalesRecord stores records of all completed sales for reporting, analytics, and accounting.	
Responsibilities	**Collaborators**
Knows the date and time of a transaction	n/a
Can create a sales entry based on details from Receipt	Receipt
Can store date, total revenue, items sold, payment details	ShoppingCart
Can report sales performance by product or category	n/a
Can handle adjustments when needed.	CustomerAccount
Can interface with Account classes for revenue tracking	StoreAccount

iv. Inventory

Class name: *Inventory* Superclass: *ProductCategory*	
Inventory keeps track of the stock levels of each product, grouped by ProductCategory, and identified by a unique identifier.	
Responsibilities	**Collaborators**
Knows what kind of Inventory it is (Meat, Fruit, Vegetable)	ProductCategory
Can add or remove a product	SalesRecord
Can determine product type	Product
Can determine how many - or the value of - products it has, and how many of a specific product it has	Product

v. ShoppingCart

Class name: *ShoppingCart* Superclass:	
ShoppingCart can contain Products and can begin the sales pipeline.	
Responsibilities	**Collaborators**
Knows the fulfilment method (delivery or pickup)	n/a
Knows which Customer it belongs to	CustomerAccount
Knows which products and how many of each product it contains	Product
Can determine the total price of itself	Product
Can determine the total price of a quantity of product	Product
Can determine the subtotal, delivery surcharge, tax, and total cost.	Product, Payment
Can initiate checkout process	Checkout

vi. ProductCategory

Class name: *ProductCategory* Superclass:	
A ProductCategory knows what kind of product it is	
Responsibilities	**Collaborators**
Knows what kind of product it is	n/a

vii. Account

Class name: *Account* Superclass:	
This is the system's basic account type, which is used as a base for more specific account types like CustomerAccount and StoreAccount.	
Responsibilities	**Collaborators**
Authenticate login details	Database
Store and manage basic account information (eg. Username, password)	n/a
Provide access control based on role type	StoreAccount, CustomerAccount

viii. CustomerAccount

Class name: *CustomerAccount* Superclass: Account	
An end user of the shop system is represented by a customer account. It interacts with the shopping, checkout and payment components and keeps track of customer specific information and past purchases	
Responsibilities	**Collaborators**
Store customer profile (account information)	Database
Access and manage ShoppingCart contents	ShoppingCart
View order history and receipts	Receipt, SalesRecord
Initiate checkout and payment process	Checkout, Payment
Update or delete account information	Database
Can view product / category information	Inventory

ix. StoreAccount

Class name: StoreAccount Superclass: Account	
System users having higher access, like store managers or administrators, are represented by StoreAccount, it gives users access to sales, stock and analytical features	
Responsibilities	**Collaborators**
View and update store inventory	Inventory
Generate reports and sales summarise	SalesRecord
Approve or reject customer returns	Receipt, SalesRecord
Manage staff and supplier accounts	Account, Supplier
Modify product categories and prices	Product, ProductCategory

x. Checkout

Class name: Checkout Superclass:	
Checkout coordinates the checkout workflow by assembling all components in a transaction.	
Responsibilities	**Collaborators**
Can determine the contents of a ShoppingCart	ShoppingCart
Can determine the record of a payment in a transaction	Payment
Can create and return a Receipt	Receipt
Can update Inventory and SalesRecord after completion	Inventory, SalesRecord

4. Design quality

a. Design heuristics

H1: Data unique to a class should be inaccessible by other classes
H2: A class should have a single purpose and responsibility
H3: Classes should have low coupling to promote modularity
H4: Abstract classes should always have their properties marked private
H5: A class should not supercede all others
H6: Naming conventions, diagrams, and logic should be consistent and unambiguous
H7: Errors should suggest a recommended pathway to a solution

5. Design patterns

a. Creational patterns

i. Factory Method

This design pattern is useful for creating a range of accounts for people who have common needs, roles and responsibility. This pattern is being employed because the Client has expressed an interest in expanding their operation, and although there are only 2 types of account subclasses at the moment - as is needed for our current design - this pattern allows for flexibility in the aspirations of the Client, who in the future may need to create all different types of accounts, which should extend the lifetime of this solution.
This pattern promotes encapsulation of account instances, and promotes extensibility.

ii. Builder Pattern

This pattern is good for creating aggregate objects such in a controlled way. Each transaction in this system involves several cohesive components that should be assembled in a similar way every time. The Builder Pattern separates the logic of the construction of transaction-related objects from their actual representations. In this way, checks can be executed to validate the construction and completeness of objects. This pattern improves the interpretability, flexibility, and maintainability of the solution, and especially the extensibility of the solution. For example, the construction of a receipt instance can be easily extended without modifying existing code to accept different payment methods, or accept multiple products, or promotions and sales. This pattern is also very testable, as unit tests for paths of construction can be observed without the entire checkout pipeline.

b. Behavioural patterns

i. Observer Pattern

The system will use the Observer Pattern to keep important parts like Inventory, ShoppingCart and SalesRecord consistent in real time. The Inventory class automatically alerts all dependent observers to change when something happens, such as when a product

is bought or restocked. This eliminates the need for direct component connectivity and enables the ShoppingCart and Store interface to rapidly update product availability. The design increases modularity and decreases dependencies while guaranteeing that every component of the system displays correct, current information. It works especially well in a retail setting where data is constantly changing and synchronizing between user carts, transaction records and stock levels is essential for efficient operation.

c. Structural patterns

i. Singleton Pattern

Throughout the course of the application's lifecycle, the Singleton Pattern guarantees that crucial system components like Inventory and DatabaseConnection have only one active instance. This keeps correct stock and transaction records across all modules and avoids data discrepancies. All actions will have access to the same data source thanks to the pattern's central point of control. It avoids redundant or conflicting updates, improving system integrity, resource management, and dependability. The Online Convenience Store's overall stability and effectiveness are enhanced by the Singleton Pattern, which facilitates reliable product tracking and database activities.

6. Bootstrap process

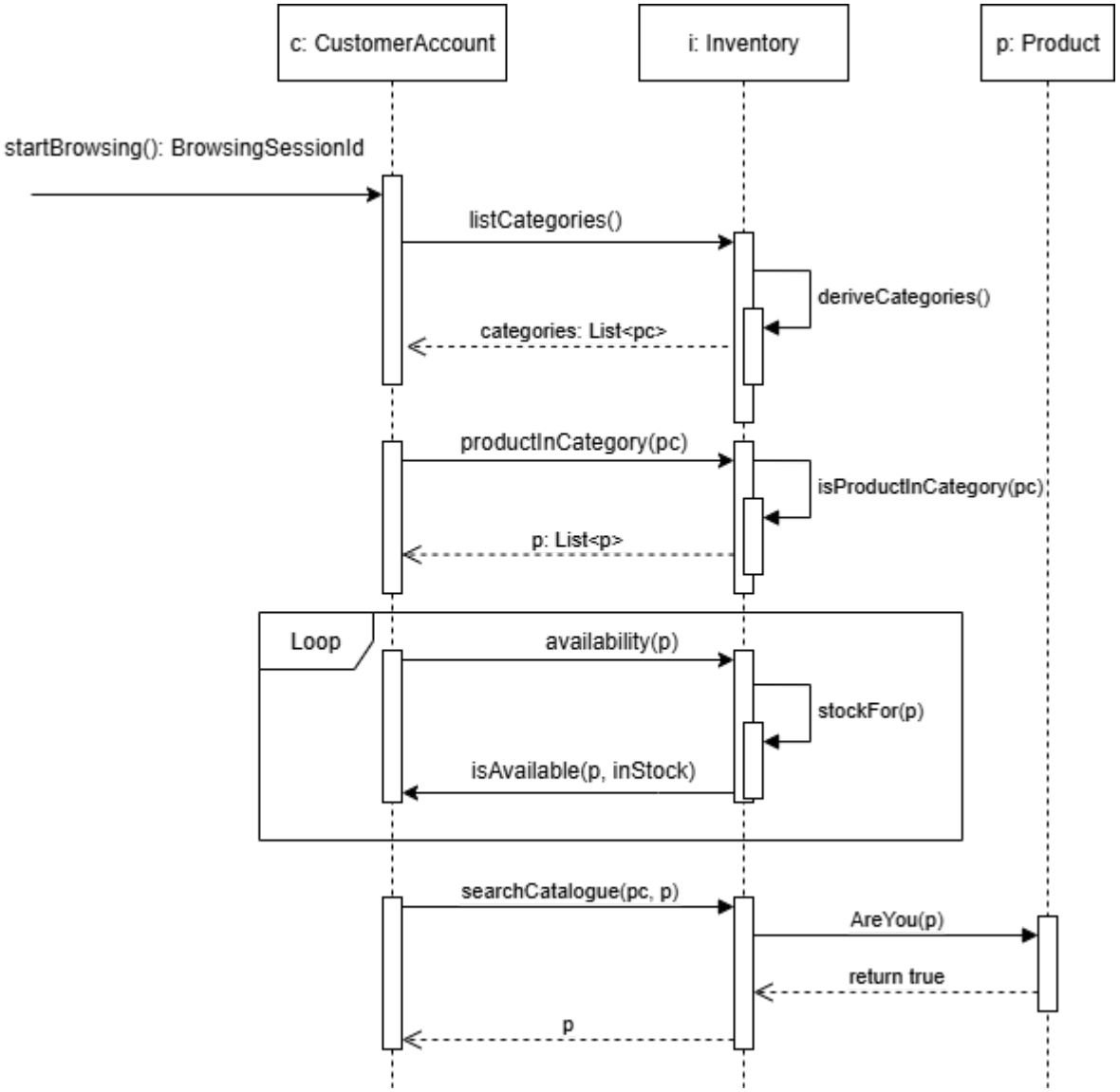
This bootstrapping process illustrates the pipeline of a customer buying goods.

1. Main creates an instance of the Checkout class
2. Main loads or creates an instance of CustomerAccount with an owned ShoppingCart instance
3. Checkout validates the ShoppingCart instance and uses it to add up the total of each Product
4. Checkout calculates any discounts against the total by checking each Product and subtracting the discounted price from the total
5. Checkout records the selected PaymentMethod on the Receipt (no stored payment information)
6. Checkout updates Inventory
7. Checkout creates, saves, forwards a Receipt, and creates a SalesRecord
8. CustomerAccount clears the ShoppingCart and returns the Receipt to the email of the customer

7. Verification

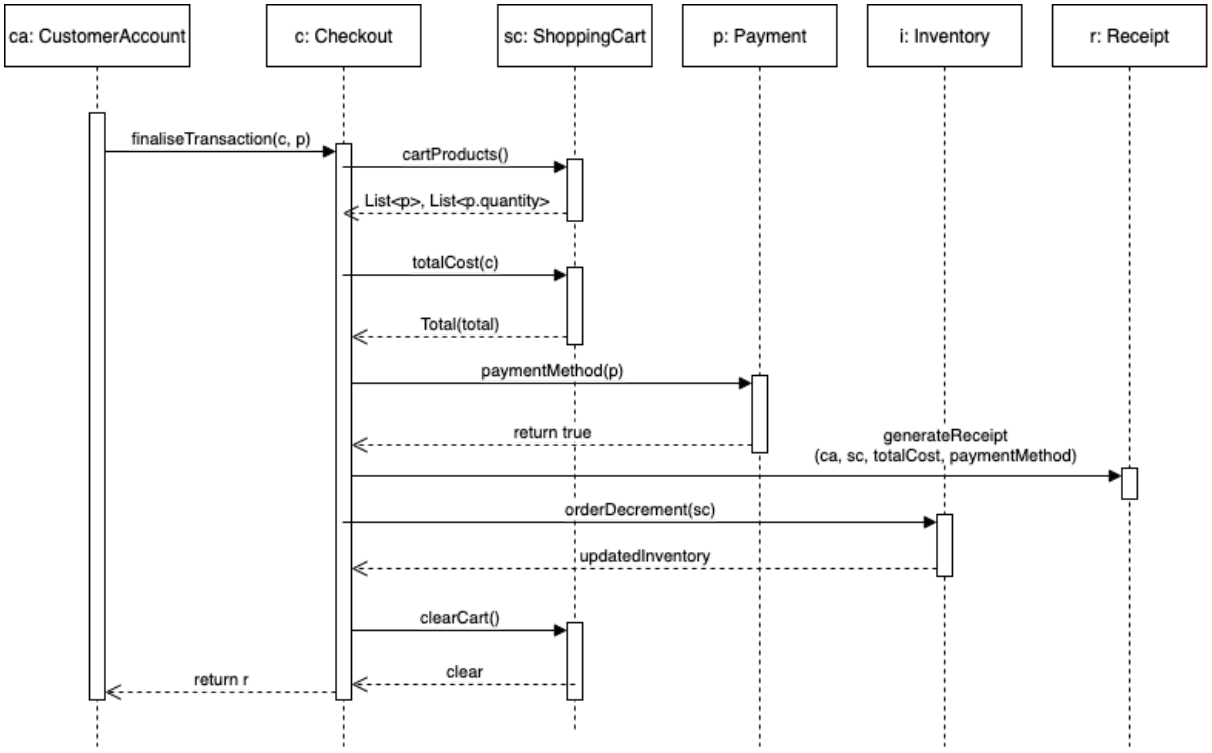
The tendered design has been verified with simulations of several use cases, as can be seen below.

7.1 Give customers the ability to browse the store catalogue online



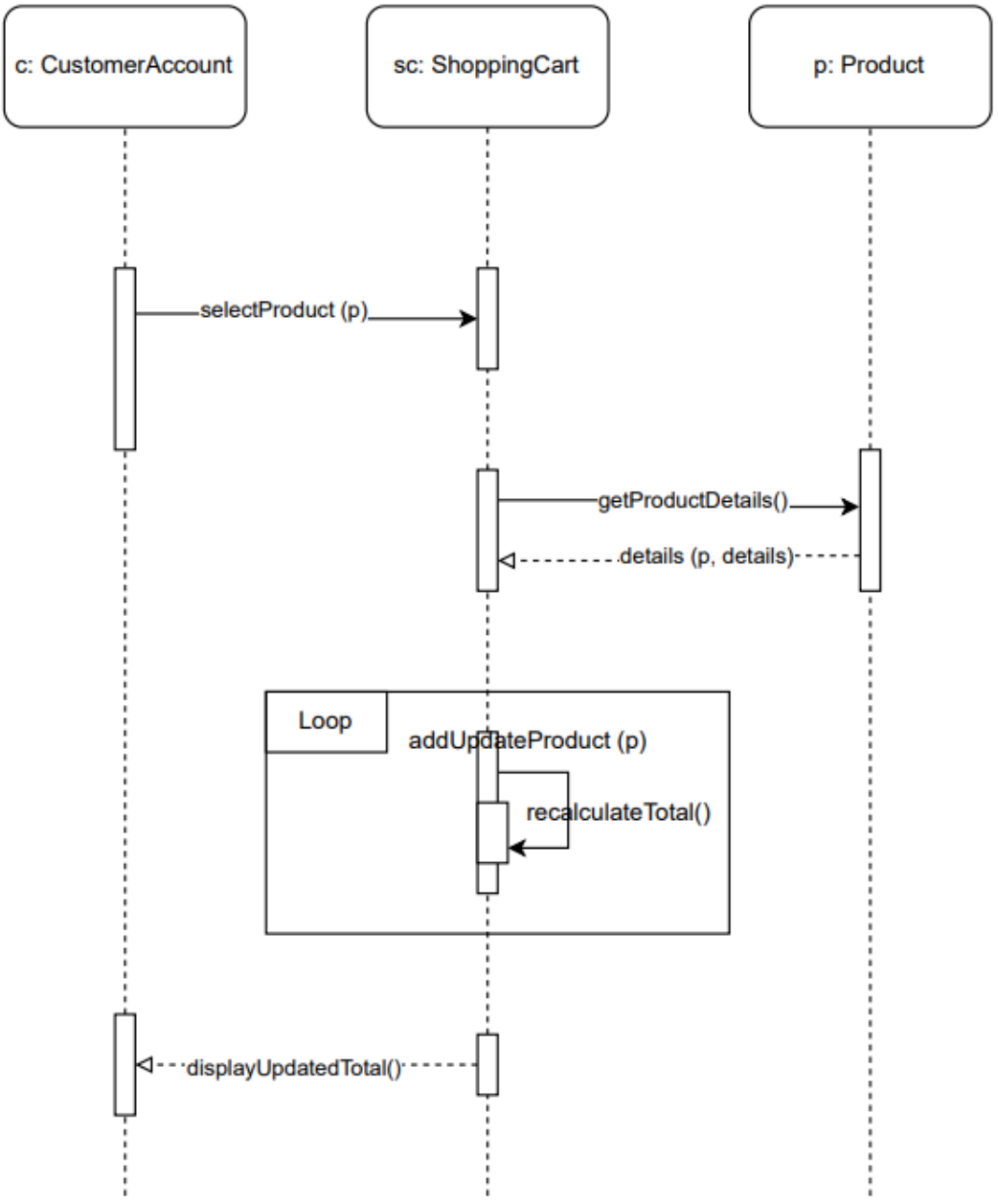
The interaction begins when a browsing session is started for CustomerAccount. The account first queries Inventory and receives a list of categories. It then requests the products in a category. For each product in that list, availability is resolved inside a Loop: CustomerAccount asks Inventory for availability(p), Inventory fetches the current stock count, returning a boolean result. This diagram then continues with searchCatalogue(pc, p), seeing if a product or category exists, and if it does, returns it. The subtask for viewing product details is not represented here because the details of a given product are already bundled in the List<p> when displaying the catalogue.

7.2 Generate receipts for customers



This sequence diagram illustrates the final checkout and receipt generation process in the online store system. The process begins when the CustomerAccount initiates finaliseTransaction(c, p) on the Checkout object. The Checkout retrieves the list of products and quantities from the ShoppingCart using cartProducts(), then requests the total purchase amount with totalCost(c) and receives the computed total. It proceeds to confirm payment by invoking paymentMethod(p) on the Payment object, which returns a success confirmation. After successful payment, Checkout calls generateReceipt(ca, sc, totalCost, paymentMethod) on the Receipt object to create and issue the official proof of purchase. The system then updates stock levels through the Inventory using orderDecrement(sc) and confirms the update with updatedInventory. Finally, the Checkout clears the shopping cart using clearCart() and returns the generated receipt to the CustomerAccount, completing the transaction.

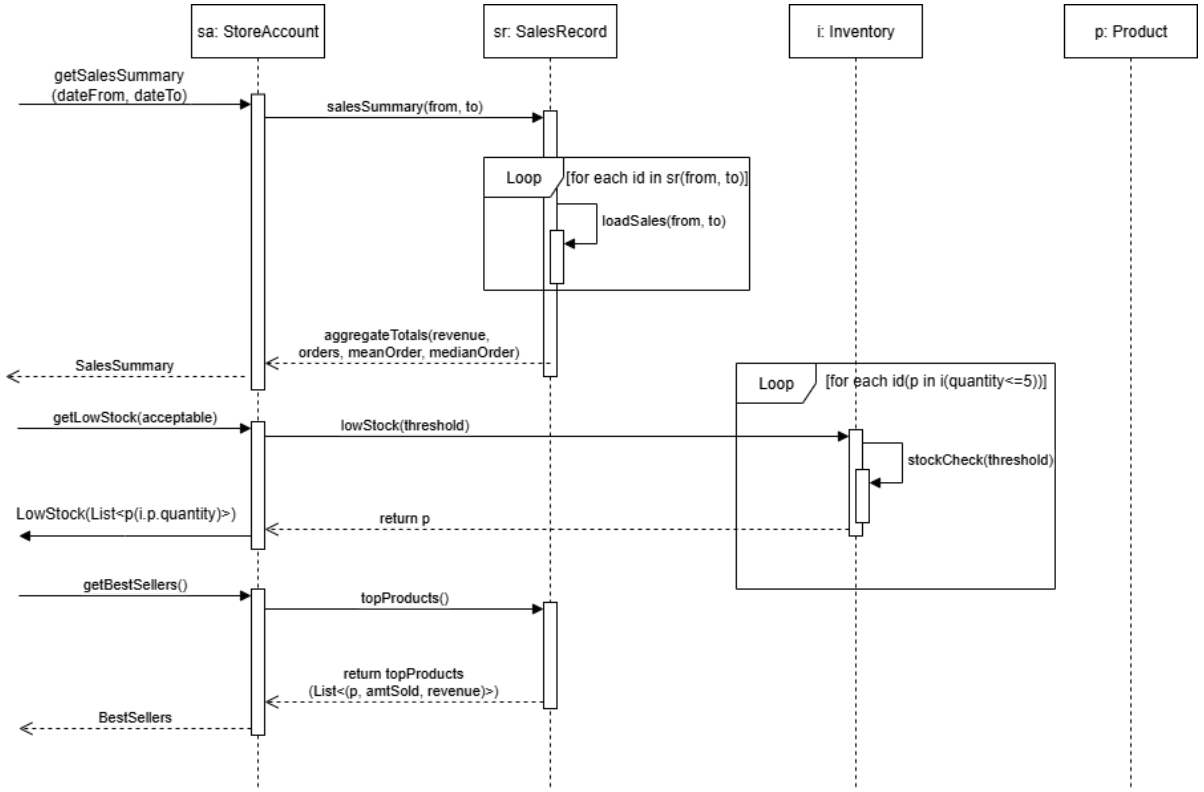
7.3 Management of shopping cart



This sequence starts when the customer initiates selectProduct(p) to start the process, the cart retrieves product details from the Product class using getProductDetails(). To maintain

accurate totals, the cart repeatedly runs recalculateTotal() and addUpdateProduct(p) for each product added or changed inside of the loop. Lastly the customer receives the displayUpdateTotal() from the cart.

7.4 Allow management to view basic sales statistics



To view some basic sales statistics, the manager executes 3 queries here. First, their StoreAccount asks the SalesRecord class to load sales within a range of time and aggregate the totals.
Then the manager decides they want to check products which are low in stock, and they define an acceptable threshold for which products will be marked as such (the value ‘5’ is just a placeholder). They are returned a list of products and their quantities.
Lastly, the manager decides they want to view their best selling products, and StoreAccount asks SalesRecord for a ranked list, which is returned. This sequence doesn’t need a loop, since the range is already pre-defined. The lifeline for Product is present here only if product details are resolved later, which they aren’t in these sequences.

8. References

Bass, L., Clements, P. and Kazman, R. (2021). Software Architecture in Practice, 4th Edition. Addison-Wesley Professional.

Bass, L., Clements, P. and Kazman, R. (2003). Software architecture in practice. Boston: Addison-Wesley.

Meyer, B. (1992). Applying ‘design by contract’. Computer, [online] 25(10), pp.40–51. Doi: https://doi.org/10.1109/2.161279

Appendix - Assignment 1

Swinsoft's Software Requirements Specification Document

Case Study: Online Convenience Store

1. Introduction

This Requirements Specification document is for a mid-sized business based in Hawthorn who wish to expand their business into the online world, enabling their customers to purchase products from the comfort of their home.
This document outlines the key features, tasks, functional requirements, and quality attributes for developing and integrating an online convenience store system. It will achieve this by identifying the main user tasks and demonstrating them via the Tasks and Support method of describing task descriptions.

2. Project Overview:

a. Domain vocabulary

Catalogue *- A searchable list of products that are available for purchase*
Shopping Cart *- A temporary collection of selected products for checkout*
Invoice/Receipt *- A document that confirms a transaction, including details of the product and the total cost*
Stock *- The amount of product that is available*
Product Type *- The category or classification of assigned to each product (snack, drink, ect)*
Customer *- A person that will make purchases online*
Store Manager/Employee *- A person that will be responsible for stock and sales monitoring*

b. Goals

- *Allow the customers to browse products and shop online efficiently*
- *Provide account creation and shopping cart functionality*
- *Support multiple payment methods and an automated receipt/invoice generator*
- *Enable staff to be able to change product categories and stock levels*
- *Allow management to be able to view basic statistics & logistics status of the products*

c. Assumptions

- *Customer has and uses their own internet to order on their selected device*
- *The Client has only 1 store currently, and doesn't plan on opening any more*
- *The Client has their own scalable product storage solution and WMS which they may wish to integrate into the proposed system at a later date*
- *The Client via Swinsoft Consulting will direct UI/UX design and aesthetic decisions*
- *The business currently has a hybrid paper-based and electronic system for managing stock in store only*
- *The Client sells on average 1000 products daily*
- *The Client has 3 types of staff:*
    - *Management*
    - *Store employee*
    - *Delivery driver/courier (transportation provided)*

d. Scope

Swinsoft Consulting has directed this group to assist in the requirements specification for a medium-sized grocery/convenience business in Hawthorn.
The client desires a system which can allow their business to expand into the ecommerce space. The system will allow customers to create accounts, be able to browse a product catalogue, manage their shopping cart, make payments and be able to receive invoices or receipts. For the workers of the store, the system will be able to support stock management, the ability to change product categories, as well as viewing basic sales statistics. The aim of this system is to be able to provide an easy and functional platform that both customers and workers will be able to use.

3. Problem domain

a. Pain points

Stock management
Users of the stock management system currently manage stock via a hybrid electronic and paper-based system, which is kept in-store. On occasion, a shipment might be missed, due to lack of automated notifications or reports. Users of the system are frequently frustrated due to the workarounds they need to constantly utilise.

Online infrastructure
Staff don't have a way of accessing the business online. Staff can only access the system digitally in-store, or via the paper-based system. The lack of an electronically centralised system limits the scalability of the business in terms of expandability, and also limits the visibility of sales trends and bottlenecks in logistics capabilities.

Systemic customer limitations
With no way to access the business online, the convenience of the customer is highly impacted, especially for the elderly or disabled. The system currently only provides physical invoices & receipts, limiting both customers and suppliers ability to track purchases and order fulfillments.

b. Domain entities

i. *Customer*
ii. *Employee*
iii. *Product(s)*
iv. *Invoice*
v. *Shipping company*

c. Actors

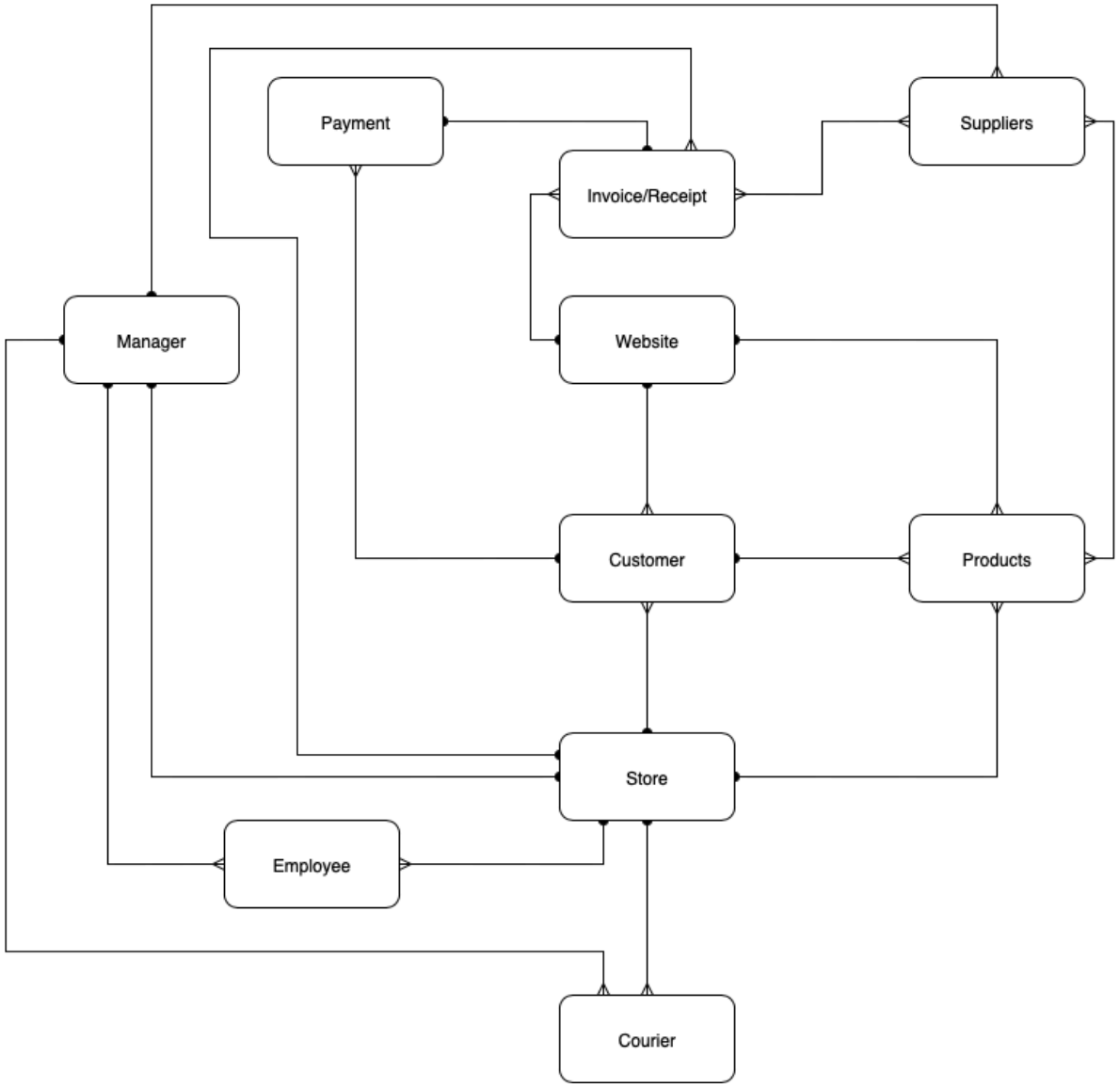
i. Store staff
    1. Employees & Couriers
    2. Management
ii. Customers
    1. Online Customers/Physical Customers
iii. Software Maintenance Team

d. Tasks

i. Create new customer accounts
ii. Ability to browse store catalogue
iii. Manage shopping cart
iv. Invoice/receipts
v. Payment handling
vi. Stocktake/logistics
vii. Product statistics
viii. Change product type

4. Data model:

a. Domain model



b. Entity description

Manager - A high level user that's responsible for overseeing store operations. Can edit product types, view sales and statistics, assign tasks to employees as well as manage suppliers and courier interactions

Employee - Regular store staff that oversee day to day operations such as stock management, order fulfilment and customer service. They can interact with the store and products as well as customers

Store - Represents a physical retail environment. It facilitates interactions between customers, employees and the product within the environment

Customer - An End User who can browse the website, adds items to a cart, places orders and makes payments. Customers are able to create accounts, manage their carts and receive invoices

Products - Goods offered for sale at the store, products have stock levels, categories, prices and also availability. They can interact with suppliers and are visible through the website

Website - The online platform that allows customers to browse products, manage accounts, place orders , and interact with the store digitally

Payment - Represents the way for customers to be able to pay for their orders. It can include card payments, 3rd party payments through apps like paypal. Triggers an automatic invoice generation

Invoice/Receipts -Shows proof of a transaction when the payment has been successful. Sent to customers through email and kept for reporting purposes. Accessed by customers, employees and manager

Suppliers - External vendors who provide inventory. Managers and employees interact with the suppliers to order more stock, receive goods and can update product information

Courier - External delivery services that are responsible with deliveries for shipped orders. Receives instructions from the store and can update tracking info for customers and managers to access

5. Task descriptions

a. Task 1

Task: Create new customer accounts	
Purpose:	Give email sign-up capability to customer. Customer accounts enable order history, favourite products, and faster checkout. Accounts are validated through email verification.
Trigger/precondition:	Customer wishes to create a new account. Customer doesn't already have an account. System is working and can send verification emails.
Frequency:	~10 new accounts daily, up to ~50 during promotional periods
Critical:	
Work area:	Store website and back-end database
Subtasks:	**Example solution:**
1. Enter sign-up details **Problem:** Customer mistypes email or chooses a weak password	System validates entered fields. System enforces password strength protocols. System displays error messages when validation fails.
2. Confirm identity **Problem:** System doesn't send a verification code/link	System generates a token. System sends an email containing a code or link for identity verification. System allows for resending the email and sets a 10 minute indirect expiry extension.
3. Complete account setup	System stores minimal customer information. Optional fields can be entered later. In case of abandonment, no information is retained.
Variants	
1a. Customer wants to use an SSO login 1b. Customer doesn't want an account	1a. System supports OAuth 2.0 SSO login methods (e.g. Google, Facebook) 1b. System supports guest checkouts

b. Task 2

Task: Give customers the ability to browse the store catalogue online	
Purpose:	Give customers online product browsing capabilities. Products inform customers of descriptions and pricing information.
Trigger/precondition:	Customer views store website on desktop or mobile. Catalogue is available and concordant with stock DB.
Frequency:	~200 visits per day, ~1000 during promotional events
Critical:	Browsing the store's catalogue is essential for engagement
Work area:	Store website
Subtasks:	**Example solution:**
1. Customer uses their preferred device and platform to request access to the landing page.	System responds with client-side information, including web-app elements, core navigation elements.
2. Customer is able to navigate product catalogue **Problem:** Customer can't find a category or has accessibility issues	System provides clickable menus and categories. System design philosophy is clear and unambiguous, with identifiable menus, and category icons.
3. Customer can search for a product	System provides a toolbar to search for products and categories. System autocompletes queries for increased operability.
4. Customer can view product details	System displays product descriptions, pricing, product availability, images, and similar products.
Variants	
1a. Customer accesses an "offline" (cached) version of the website **2a.** Customer wants to view promotional products	**1a.** System offers the capability for navigation of the website on a user's local network, but with reduced functionality. **2a.** System highlights promotional products in the catalogue. System also shows promotional products under a "Special Offers" section.

c. Task 3

Task: Give customers the ability to manage their shopping cart	
Purpose:	Let customers edit their intended purchases and minimise problems for checking out by keeping the selections consistent on different devices
Trigger/precondition:	Customer adds item from catalogue or a saved cart exits. Pricing and inventory of products are available
Frequency:	Used in most sessions from 60-80% of visitors with peak times rising during promotions
Critical:	Works in tune with stock count, price changes, cart updates/expiry, tax/shipping
Work area:	Store Website (front end), Cart service, Pricing/Tax/Shipping services
Subtasks:	**Example solution:**
1. Customer Adds/Removes/Updates product quantities **Problem:** Customer exceeds available stock or price changes	System validates with the live stocks, System reserves products for a short period of time. System notifies with price change, recalculates price total/tax
2. Customer chooses delivery type (pick up or delivery) **Problem:** Address incorrect or pick up time slot not available	System shall support address validation, and show available pick up times. System can calculate shipping fees/ETA, and can handle no-delivery postcodes
3.Customer reviews and checkout cart **Problem:** Cart lost on logout/session time out across devices	System keeps anonymous carts in local storage. System allows customer to upgrade to an account for when customer logs in, lets them utilise a “save for later” option.
Variants	
1a. Price changes while items in cart	**1a**. Flag the price difference before checkout and receive customers confirmation before checkout

d. Task 4

Task: Generate invoices/receipts for customers	
Purpose:	Give customers a clear record of what they bought, confirm their payment, help track money, and act as proof of purchase.
Trigger/precondition:	Customer has purchased a product
Frequency:	Used about 100% of the time when a purchase is made
Critical:	Customer makes multiple purchases
Work area:	Store website
Subtasks:	**Example solution:**
1. Generate Receipt/Invoice	System queries database to provide statistics and data on sales/purchases
2. Save Receipt **Problem**: Allocated Space is FULL	System allows for the receipt to be automatically saved for future reference, and can be seen by other authorised staff If space is full, system will auto-archive receipts older than 24months
Variants	
1a. Customer wants an e-receipt.	**1a.** System shall support email proof of purchase

e. Task 5

Task: Give customers the ability to pay for goods/services in multiple ways	
Purpose:	Allow customers to complete purchases conveniently and securely by offering different payment options.
Trigger/precondition:	Customer has added items to the cart and proceeds to checkout.
Frequency:	Occurs whenever a customer purchases items (very frequent task).
Critical:	Payments directly impact sales revenue and customer trust.
Work area:	Checkout page, backend payment processing, integration with third-party payment gateways.
Subtasks:	**Example solution:**
1. Display available payment options	System displays the available payment options (credit/debit card, PayPal, etc.)
2. Card Detail	System collects payment details from the customer securely
3. Payment Status	System processes the payment and gives feedback on if it was successful or not
4. Record Update	System update order status and generates receipt/invoice
Variants	
1a. Online Order but in-person payment	**1a.** System allows a customer to create an order and then come pick it up in the store

f. Task 6

Task: Create a system which manages stock levels and logistics	
Purpose:	Keeps inventory accurate and helps to fulfill order efficiently by preventing overselling
Trigger/precondition:	Stock received from a delivery. Customer adds/updates/cancels an order
Frequency:	Continuous for live time updates
Critical:	Inventory accuracy, pickup/packing errors, back orders
Work area:	Warehouse, inventory service
Subtasks:	**Example solution:**
1.Controlling stock on order events **Problem:**Multiple carts/orders oversell on limited stocks	System allows for short term reservation when customer adds product to the cart. System allows product releases when a cancel/timeout/return happens in the cart
2.Pick and Pack **Problem:** Product misplaced in warehouse, wrong product picked	System Create a list with a specific area for each product with a product code, System allows for suitable replacements, scan product label/code to verify product and quantity
3.Ship and Track **Problem:** Label for tracking misprinted or wrong delivery location	System auto creates labels, and adds tracking numbers for delivery confirmation. System supports split shipments with multi tracking.
Variants	
2a. Back orders/ pre orders	**2a.** System allows allocation from incoming stock with an estimated time, customer receives a separate service level

g. Task 7

Task: Allow management to view basic statistics for stores	
Purpose:	Allow management to access sales, stock, and logistics statistics to support business goals
Trigger/precondition:	Authorised person logs in with appropriate credentials. System is available and connected to the store DB.
Frequency:	~5 times daily
Critical:	Necessary for informing the business, but won't impact core business strategies
Work area:	Administrator dashboard
Subtasks:	**Example solution:**
1. Manager logs in to administrator dashboard **Problem:** Unauthorised person attempts to access administrator dashboard	System compares provided credentials with those in its internal DB. System approves or refuses access based on provided credentials.
2. Manager wants to see statistics about daily sales	System collects store transactions, revenue, mean and median purchase value. System shows these as tables in administrator dashboard.
3. Manager wants to view stock movement and stock levels	System uses store DB to see which products are low in stock and flags them as needing to be pulled from the warehouse. System shows best selling products, and stock turnover rates.
4. Manager needs to see order fulfilment and deliveries status	System pulls data from DB to allow for viewing of shipped, pending, or delayed orders. Courier/delivery-person performance is also displayed.
5. Manager wants to view trend analysis	System shows a comparison of the current week, month, and year with the previous ones. System highlights positive and negative growth.
Variants	
2a. Manager wants to view administrator dashboard without internet access	**2a.** System provides the option to download current/previous reports

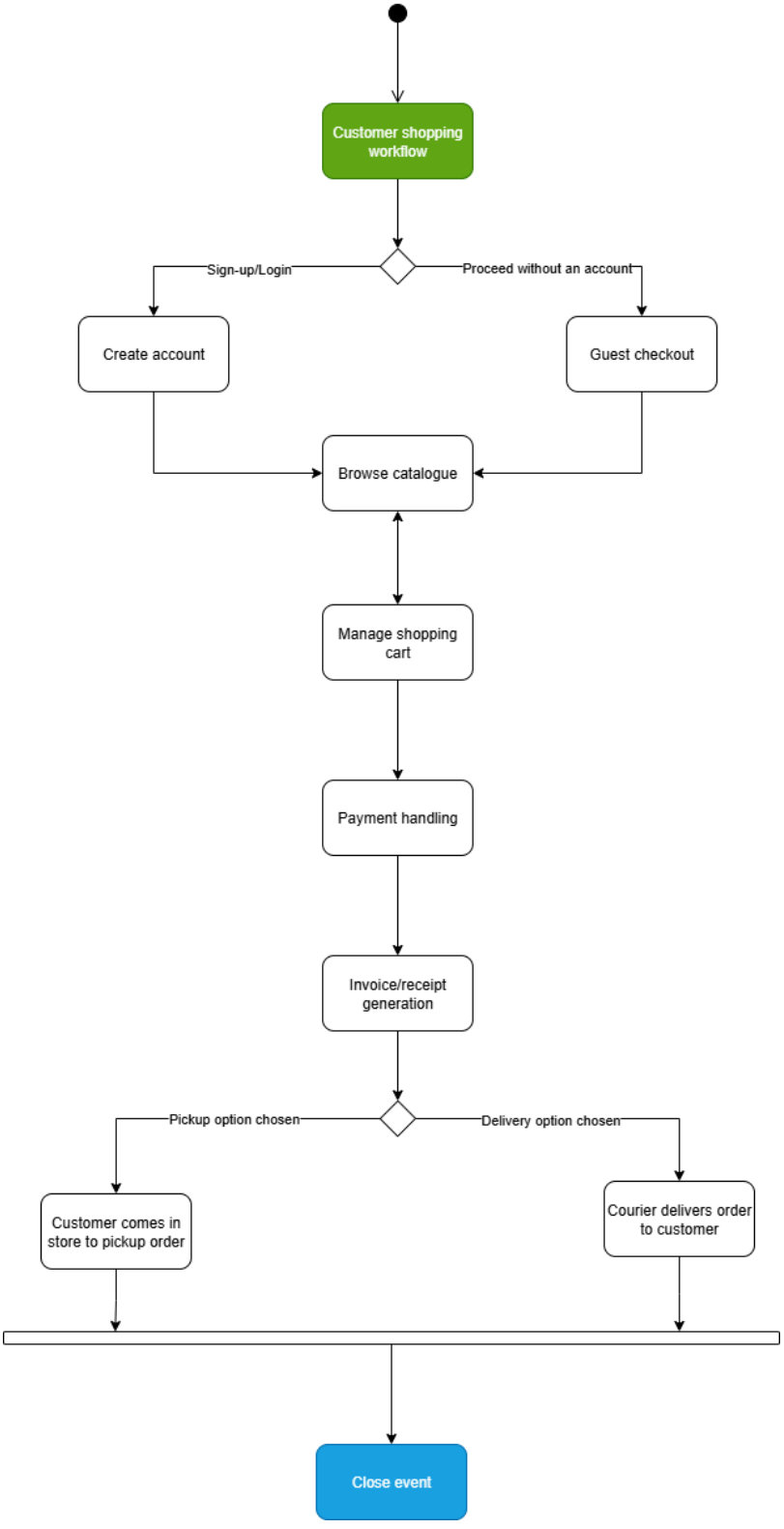
h. Task 8

Task: Allow management to change product/s type/s	
Purpose:	Makes the catalogue to adapt to change
Trigger/precondition:	Re-categorising a product
Frequency:	Whenever it is needed
Critical:	Impact on later listings/reports
Work area:	Managerial office
Subtasks:	**Example solution:**
1. Create/Edit product types **Problem:** Attribute model change could break existing products	System warns the user of any changes to the products. System shows a preview of what will change and block changes if it will break
2.Connects products to types **Problem:** Multiple updates could misclassify product numbers or time the system out	System validates changes before they go live. System shows a summary of what products have become affected
3.Publish and Update **Problem:** Search results show un-updated post change	System keeps logs of what was changed. System refreshes the search results and product filters.
Variants	
1a. Merge two product types	**1a.** System lets the manager pick which product to keep and what to merge. i.e. Move the products and update the category in a single go.

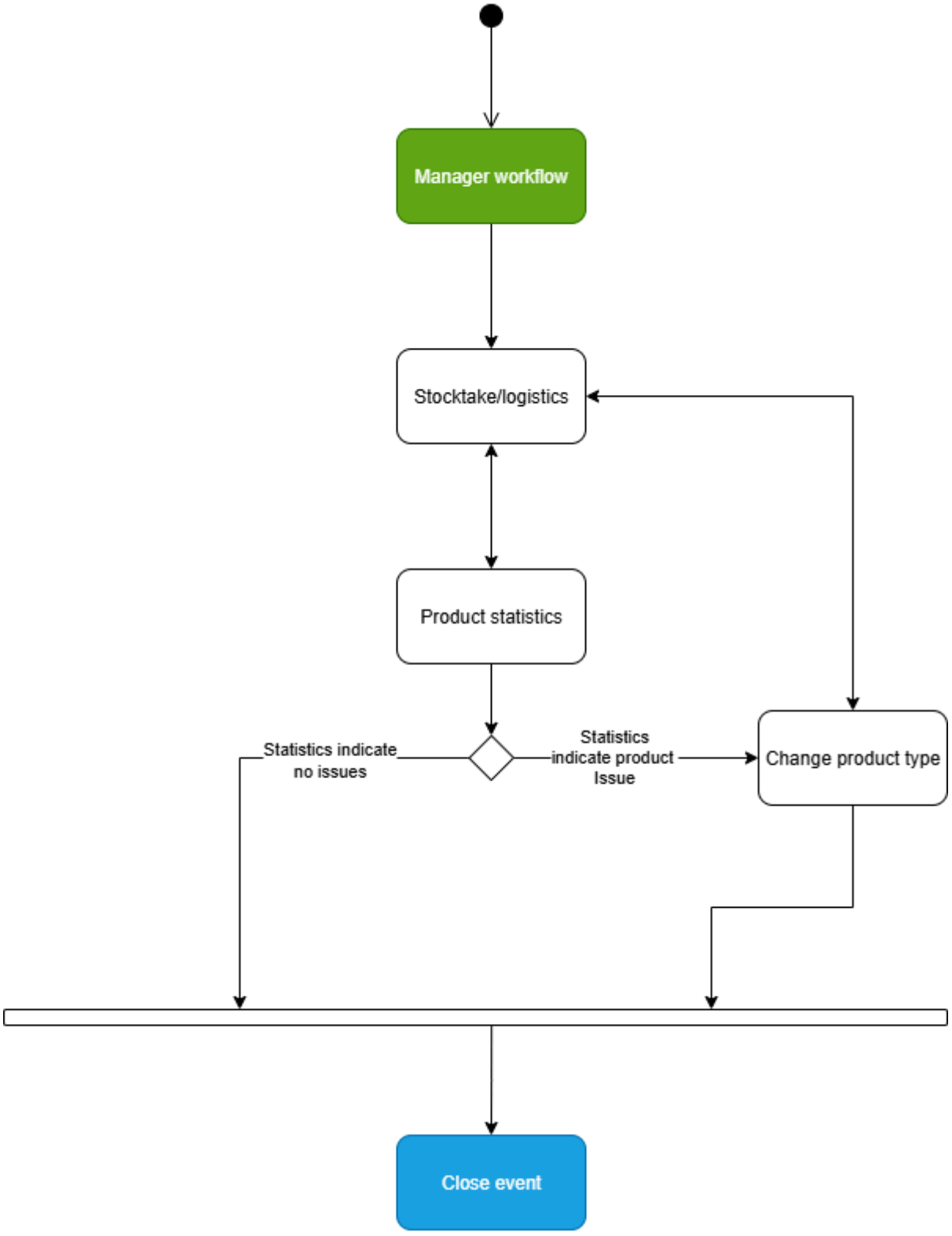
6. Workflow

a. **High level workflows**

Customer shops on the website:

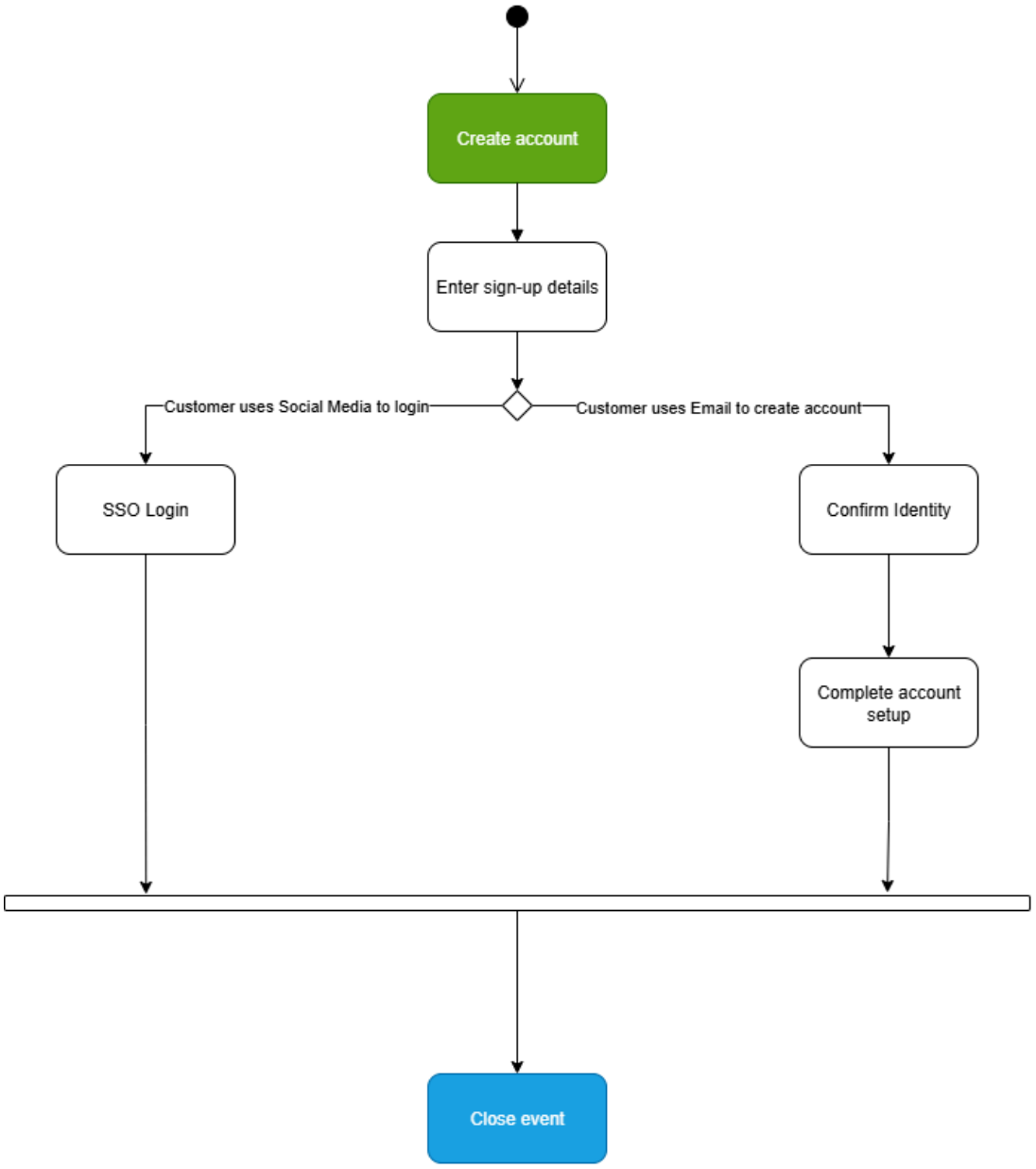


Management and operations:

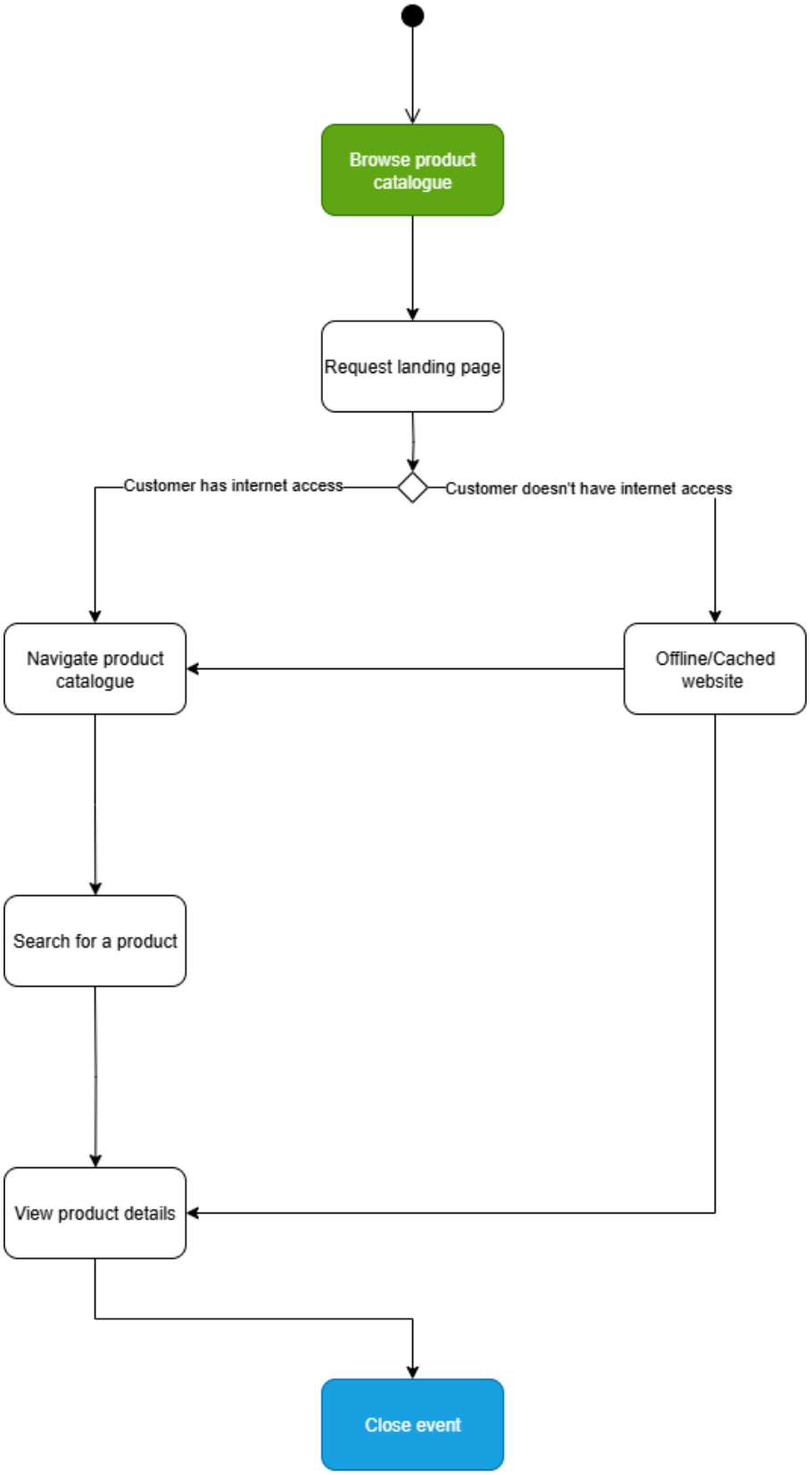


b. **Internal user task workflows**

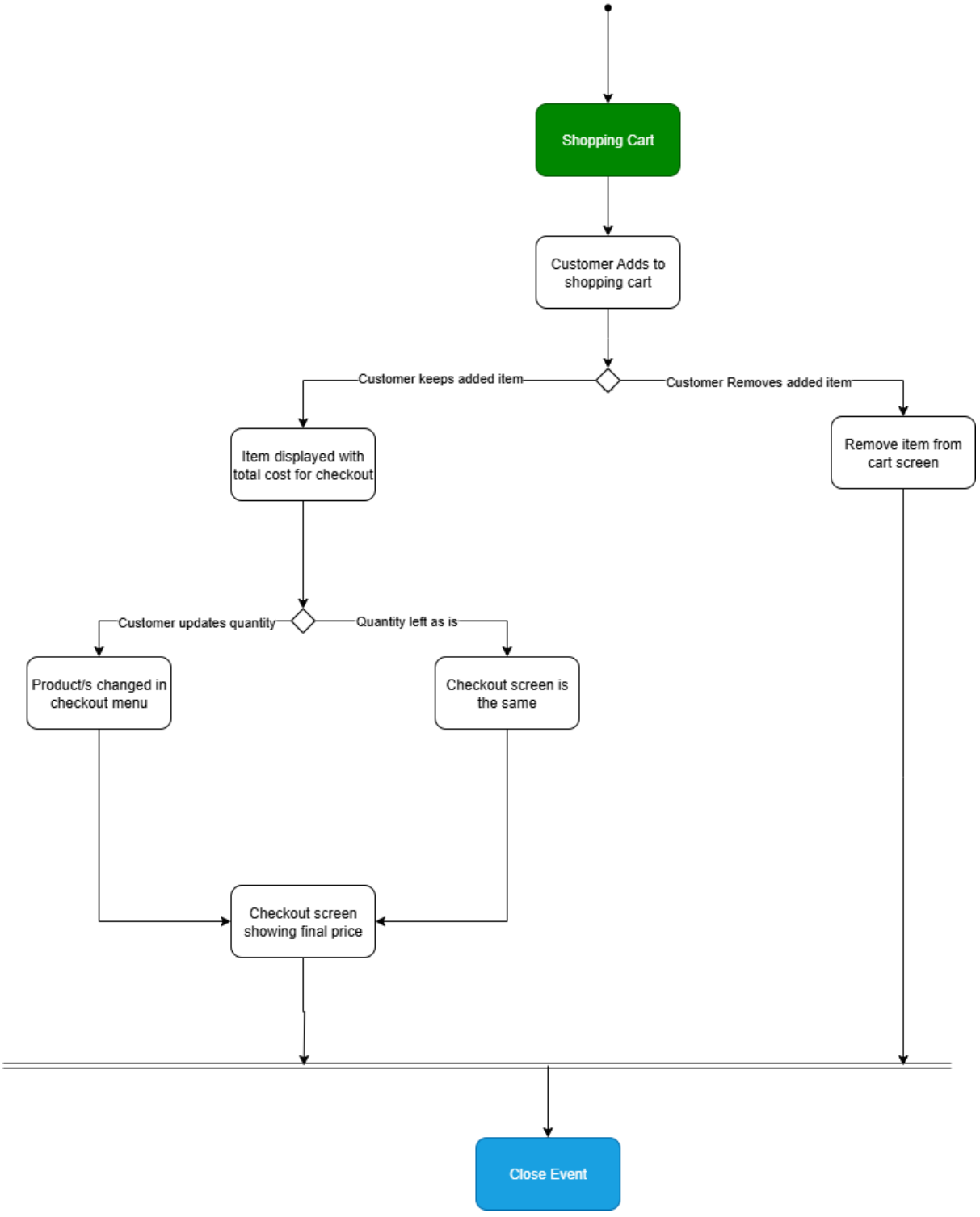
Task 1 - Create new customer accounts



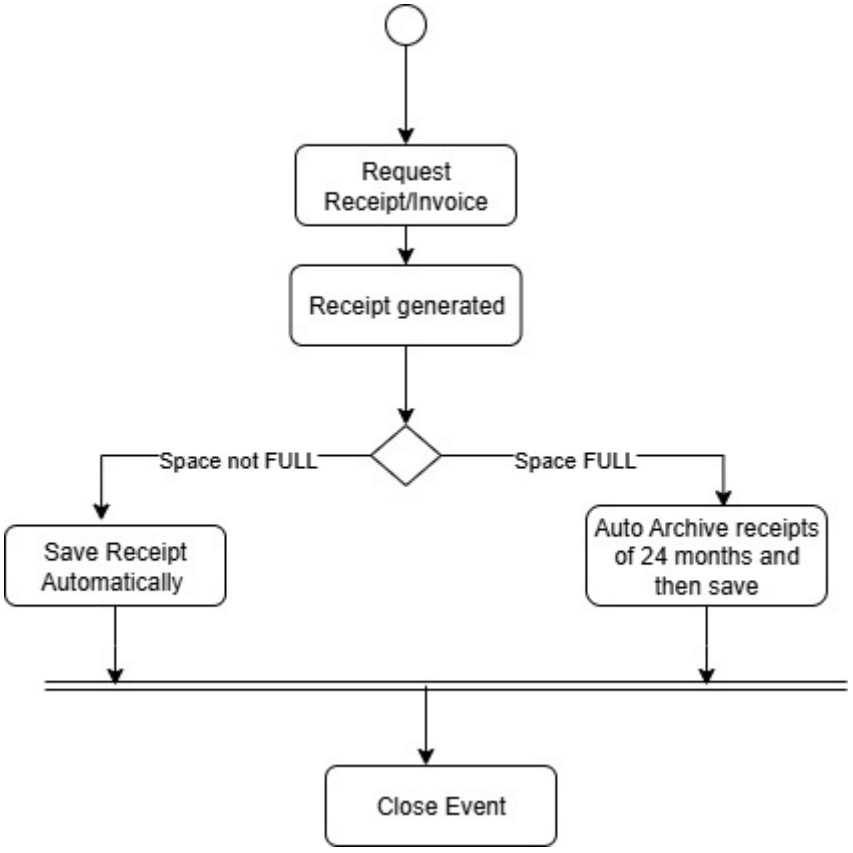
Task 2 - Create new customer accounts



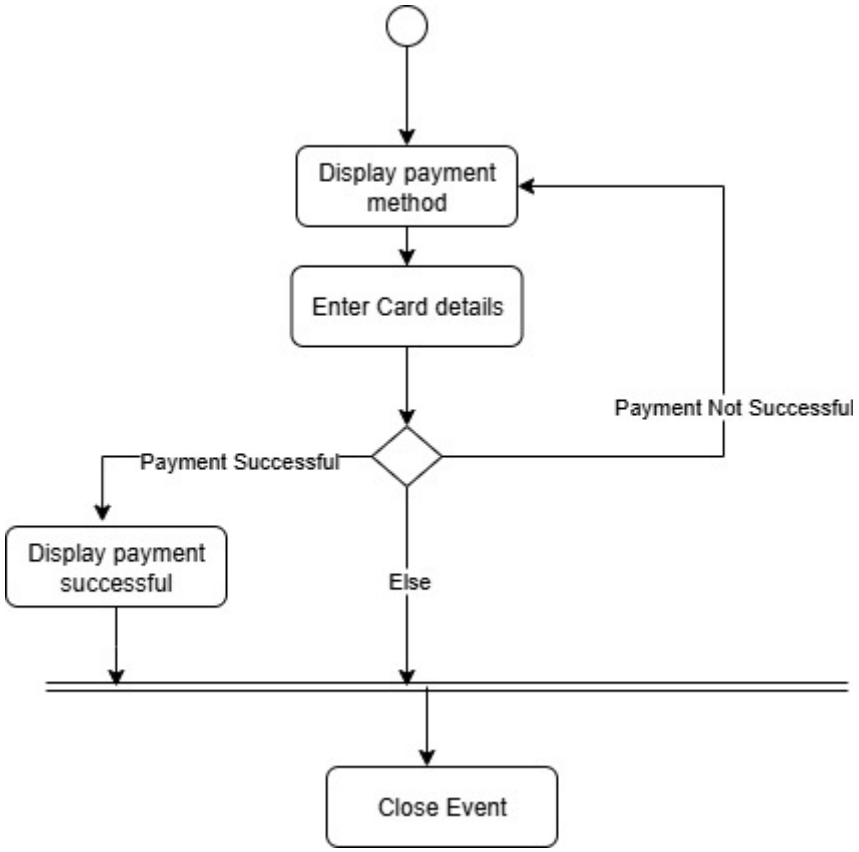
Task 3 - Updating Cart



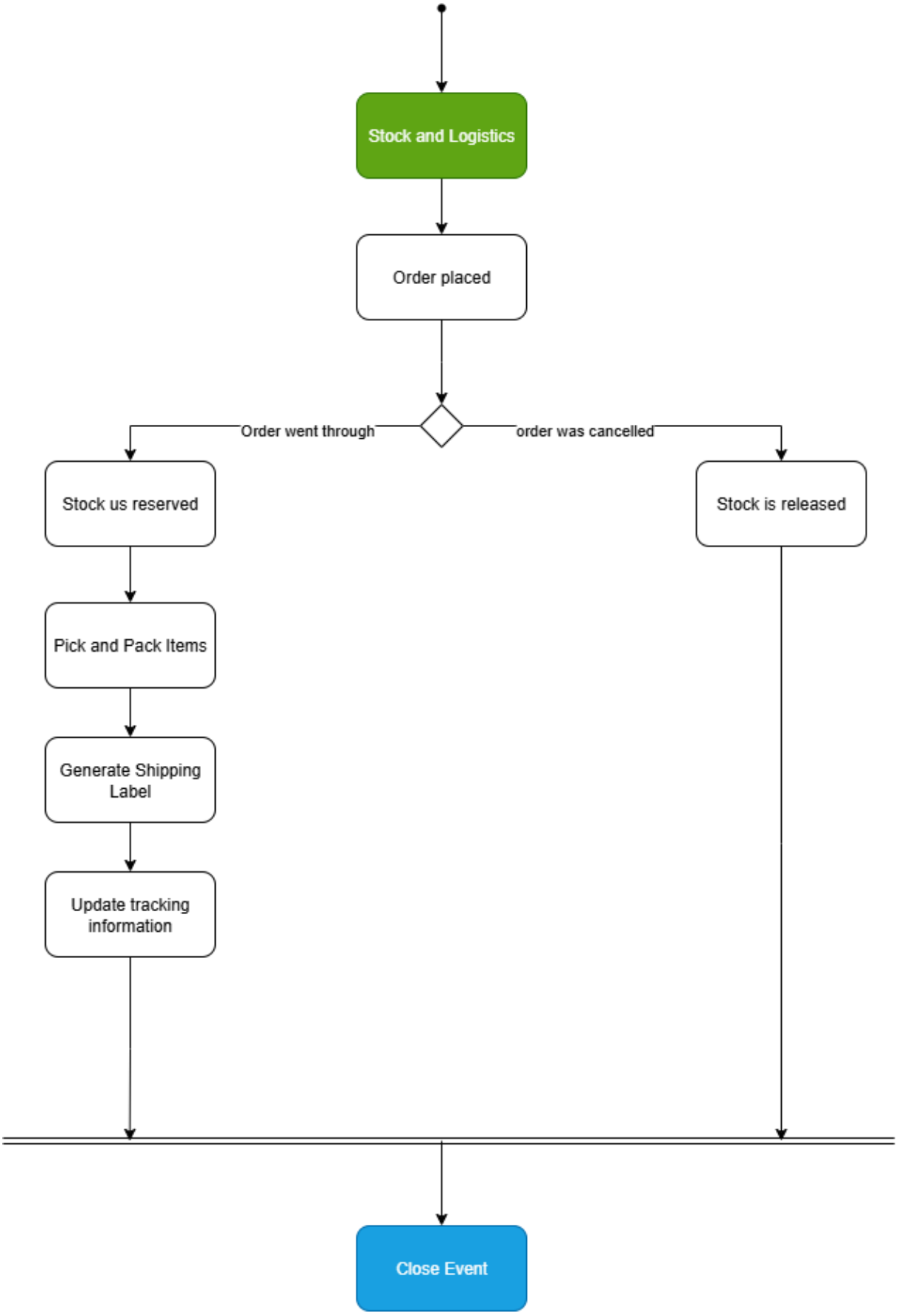
Task 4 - Generate Invoices/Receipts



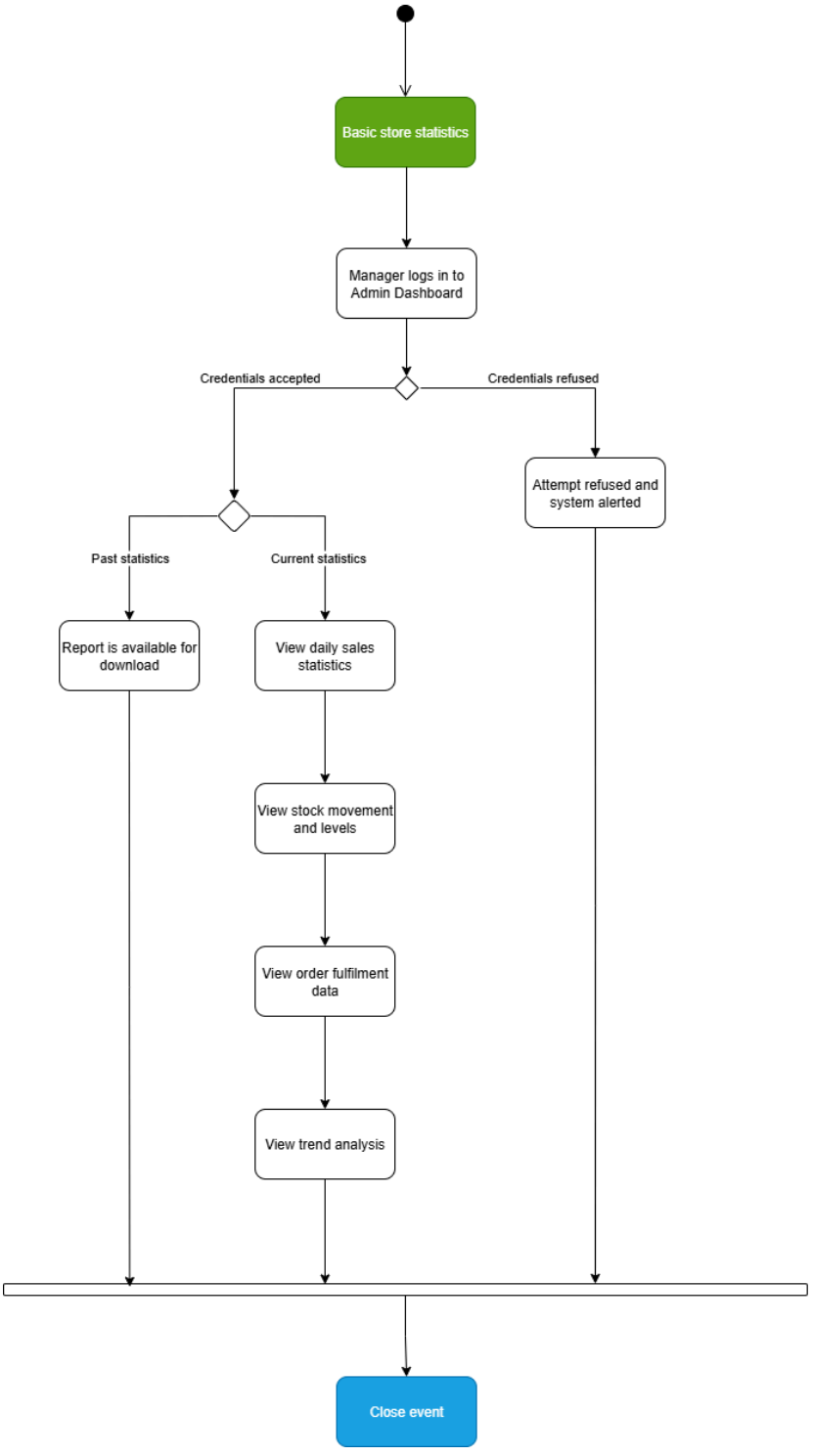
Task 5 - Give Customers Multiple Ways to Pay



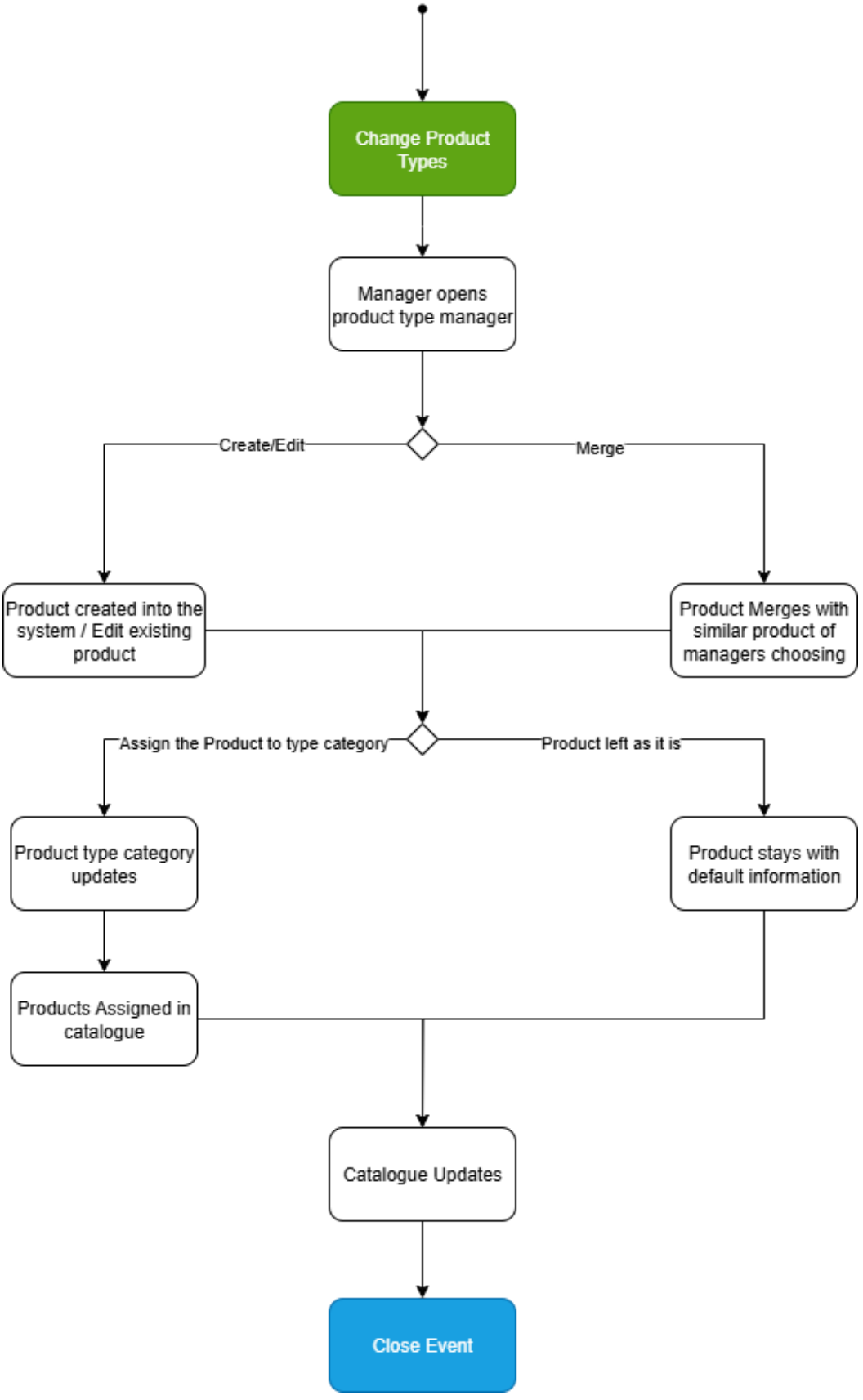
Task 6 - Stock and Logistics



Task 7 - View store statistics



Task 8 - Change Product Type



7. Quality attributes

Performance

The system must be highly responsive for the best user experience

- The product catalogue should be able to load within 2 seconds of access.
- The checkout flow should be under 5 steps before reaching the final checkout.
- Individual queries should be available and visible to the user within 1 second of request. An individual query can be something such as viewing your account details, or viewing the details of a product.
- Bulk queries should be available and visible to the user within 10 seconds of the request. A bulk query could be an invoice archive retrieval, or a report generation request.

Security

An essential and critical attribute given the system will likely store sensitive customer information, and the fact that the system may be integrated with third-party payment providers.

- Use of HTTPS/TLS encryption is needed for the communication between client and server.
- Clients passwords to be hash encrypted.
- Payments to be handled by PCI-DSS third parties
- Access control to sensitive systems should enforce role-based access.
- Suspicious activities should be logged.

Reliability & Availability

Likely the most critical quality attribute. Without these, business goals and thus the sales pipeline cannot be realised. The system should be available on a consistent basis, and under heavy load.

- System needs to be highly available, ~99.9% of the time.
- Cart should be able to keep cart data after session expiry or after downtime.
- Product data should be consistent across all navigable elements.

Usability & Portability

An easy to use UI is critical for optimal UX.

- Clear and engaging mobile/desktop UX.
- Guest checkouts are available, with account-based checkouts being the default
- UI design should be intuitive and usable on the main variety of browsers - Chrome, Firefox, Edge, Safari.

Maintainability & Scalability

The system ought to be adaptable to future client-elicited implementations (such as a Loyalty Program), while also having low cost of operation.

- Code should follow modular decoupling principles, allowing for maintenance and feature addition in specific components without full system outages. Maintenance tasks should require minimal downtime.

- System shall support up to 10,000 daily active users, alongside growths in the product range. More flexible scalability options will be discussed in Possible Solutions.
- Documentation should be written with future developers and managers/administrators in mind.
- Architecture will be OOD driven, with the implementation following OOP principles.

8. Other requirements

In addition to fulfilling and supporting all of the tasks and functions, the proposed system shall also be required to comply with a series of supplementary requirements. These additional requirements are intended to ensure that the system not only meets its core functional objectives but also adheres to broader standards of reliability, usability, security, and operational efficiency necessary for its successful implementation and long-term use.

a. Product level requirements

- The system shall store and manage all user-entered data effectively.
- Input fields must include appropriate validation checks.
- Validation must follow the internal standards and policies defined by SwinSoft.
- Provide functionality to print or export receipts, invoices, and other transaction-related documents.
- Comply with applicable data privacy and security regulations (e.g., GDPR, PCI DSS if payment data is stored).
- Maintain system performance to handle multiple users and transactions simultaneously without delays.

b. Design level requirements

In addition to the functional specifications, the system must also adhere to a set of design-level requirements that ensure consistency, usability, and compliance with organizational standards. The SwinSoft UI/UX design guidelines provide a foundation for key design decisions such as font selection, sizing, layout structures, and appropriate color contrasts. Beyond these, there are further requirements that must be observed to maintain a professional appearance, enhance user experience, and align with brand identity. Specifically, the system shall:

- Incorporate data analysis in an easy-to-use Administrator Dashboard or equivalent for accuracy in management decisions.
- Clearly display the Terms of Use, Privacy Policy, and other such documents while also providing users with the option to download each document in a convenient format.
- Show broad and specific product details, including but not limited to supplier information, product images, detailed product descriptions (likely taken from the supplier or manufacturer), and pricing breakdowns.

9. Validation of requirements

The requirements in this specification were able to be validated through task breakdowns, role based mapping, and using diagram models. Each functional task that was described in the Task and Support section in the marking criteria was defined by realistic triggers, expected workflows and to ensure a clarity of purpose.

In order to replicate real world usage and verify alignment with system goals, workflows were visualised using diagrams. Each domain entity's complete support for creation, read, update, and delete actions was guaranteed by the usage of a CRUD matrix. It was easier to confirm edge cases and usability when each task description included many versions and error conditions.

Additionally, quality attributes (such as performance, usability, and availability) that are important to the success of the system were mapped to non-functional criteria. Through this validation procedure, it was made sure that the suggested solution satisfies user and business requirements while being technically viable.

a. CRUD Completeness Check

Task/Entity	*Customer*	*Employee*	*Product*	*Invoice*	*Supplier*
Create an account	C R U D	CRUD			
Browse Catalogue	R	R	R U		
Manage Shopping Cart	C R U D	C R U D			
Generate Receipt	R	R		R U	R
Manage Payment Method	U R	U R			
Stock & Logistics	R	R U			R U
Store Stats	R	R		R	R
Change Product type	U	U	R U	U	U

10. Possible solutions:

Online system integrated with in-store system

This is the approach which is currently most popular among Australia's largest supermarkets and retailers. Management has the opportunity to integrate its in-store systems with the online ones, and has an easy and accessible way to monitor both remotely.

Similarly to the big supermarkets, the online system would provide a direct sales pipeline from the online portal to the store. The variant for Task 5 illustrates this well, where with no involvement from management, and small effort from employees, the customer can essentially serve themselves.
The online system being integrated with the in-store system provides a direct upgrade to only having one or the other. Consistency is critical for success in this area, and if successful, this kind of integrated system would allow all types of metrics to be measured and recorded for better business decisions, and pipelines to business goals and deliverables to be met more aptly.

The main issue with this type of integration, where entities such as stock levels, or order statuses update with a high level of consistency across platforms and locations, is the level of complexity required for implementation. This is especially true since the Client's business was not initially designed with an online presence in mind, and thus all of the existing in-store system's metrics for business goals and deliverables are heavily lacking in explicit data flow.
However the upfront cost would be more than more than worthwhile. The efforts which would be put towards the centralisation of systems and data would more than support the eventuality of future nation-wide scalability and further feature implementations.

Customer Oriented Self Service System

This is an alternative that focuses on creating a customer driven experience, where each user will be able to navigate independently through the website to find products, manage their shopping carts, and to complete their orders without having to worry about going through staff. The system will integrate users being able to create accounts, use guest accounts to checkout, and to be able to checkout securely.

Customers will be able to view the whole store catalogue online, and can search using filters or keywords, and then add products to the cart, similarly to all the major supermarket systems. The customers will be able to checkout using their credit card, PayPal, or other similar supported systems. Once payment has been successful, a receipt will be generated and sent to the customers email, or an email option to be given to guest checkouts. Users who are logged in will be able to view their order histories or be able to re-order recent purchases.

Store staff will be able to access the system to manage stock levels, fulfil orders , and to assist customers in store when it is needed. A separate admin system will be accessed by

managers to be able to view store statistics, update product inventory, and to track supplier deliveries.

The advantages of this system is that it will be able to help reduce the staff workload and will increase the customer independence, there will be a 24/7 access to the store catalogue and allows customers to purchase items at any time, and will be able to handle more customers and a larger product inventory. However there can be drawbacks to this system as there will need to be a better UX system since a bad interface can result in less sales, customers who aren't familiar with digital devices can find it hard to navigate online, and in some cases still requires some workload from staff or managers.

However this solution is still ideal for the modern retail environments with users who are more digitally confident, and where automation can be seen to improve more of the efficiency and performance.

Hybrid Marketplace Partnership System

This solution takes a different approach by leveraging existing e-commerce platforms or marketplaces (such as Amazon, eBay, or Shopify-based marketplaces) rather than building a fully independent system from scratch. Instead of relying solely on an in-store integrated system or a fully self-managed online shop, the business would partner with an established marketplace that already provides the technical infrastructure, payment processing, and customer management tools.

Through this method, the business can quickly establish an online presence with significantly reduced development and maintenance costs. Customers would be able to browse and purchase products through the chosen marketplace, taking advantage of its familiar interface, trusted reputation, and built-in traffic. The business would, in turn, manage its product listings, stock levels, and deliveries through a streamlined vendor dashboard provided by the marketplace, while still maintaining in-store systems for traditional sales.

One of the main benefits of this solution is the reduced technical complexity: instead of building and maintaining every component in-house, the business uses a proven and secure third-party platform. This reduces the need for heavy investment in IT infrastructure while allowing the store to reach a wider customer base, even beyond its immediate region. Customers benefit from a smooth and trusted checkout process, multiple payment options, and reliable order tracking.

However, the drawbacks of this approach include reduced control over the customer experience and branding, as the marketplace dictates many of the design and interaction elements. There may also be commission fees or listing charges imposed by the marketplace, reducing profit margins. Despite these limitations, this solution is particularly effective for small-to-medium businesses seeking to expand quickly into the online space without bearing the full weight of system development and long-term technical management.

12. Verification

All requirements that are defined in this specification are verifiable through functional testing, system performance monitoring and usability walkthroughs.

Each task represents clear user goals, the trigger conditions and the expected results. An example of this is using “Browse Catalogue”, as it can be tested by confirming filter and search behaviour. Another example can be the “checkout process”, which can be tested by completing an order and confirming that a receipt was generated.

Non functional requirements are also stated in measurable terms, an example is that a page load time needs to be under 2 seconds, the site needs to have an uptime of 99.9% and that there are password requirements when an account is created. These types of functions can be tested through system logs, load testing tools and UI automation.

Usability testing and customer input can be used to evaluate acceptability in situations where there may be uncertainty.