Software Design Specification
for
Nardo's Inventory System

Version 1.0

Prepared by

GROUP: Mon_3-5_G02

Melissa Williams	**620150166**	**melissawilliams046@gmail.com**
Alonzo Brown	**620173600**	**ajanidbrown96@gmail.com**
Elizabeth Wright	**620170287**	**liz.wright5665@gmail.com**
Jevaughn Johnson	**620173043**	**johnsonjevaughn63@gmail.com**
Madison Sujanani	**620183857**	**madisonsujanani@gmail.com**

Course Instructor: **Dr. Ricardo Anderson**
Course: **COMP2140 – Introduction to Software Engineering**
Studio Facilitator: **Mr. Eyton Ferguson**
Date: **November 22, 2025**

TABLE of CONTENTS

1.0 Project Overview

Nardo's One Stop Shop, located at the UWI Mona Campus, currently manages its inventory using handwritten notes and memory, which often results in missed updates, inaccurate counts, and delayed restocking. Because the shop sells fast-moving items like snacks and drinks, even minor errors can create shortages or confusion during busy periods. Our project aims to replace this manual approach with a reliable, straightforward inventory system that organises products digitally. The intended users are the owner and trusted employees who are responsible for tracking stock throughout the week. This digital transition will help the business operate more smoothly and reduce avoidable mistakes.

The system will support core inventory operations, including adding, updating, and deactivating product records, automatically adjusting stock after sales, and generating alerts when items fall below a set threshold. Users will also be able to search for or filter products by name, category, supplier, or stock level to speed up daily decision-making. The software will log all stock changes, giving the owner a clear history of additions and removals for accountability. These features replace the manual book system with faster, more organised workflows that are easier to maintain. The goal is to ensure that staff can see accurate, real-time information whenever needed.

The system's design is guided by several functional and non-functional requirements from the SRS, including real-time updates, secure user authentication, quick response times, and dependable operation during extended business hours. The application will run on desktop computers within the shop's LAN using a browser interface, supported by a PHP backend and MySQL database. Security measures such as encrypted passwords, role-based access controls, and input validation will help protect inventory data. Performance expectations, like processing updates within a few seconds, also shape how the system is structured. These constraints support a clear, maintainable architecture that is easily understood and used by the shop's staff.

2.0 Architectural Design

2.1 General Constraints

The hardware and software requirements limit the Inventory Management System: it must run on standard desktop or laptop computers with 4GB of RAM, a 1.8 GHz CPU, and 500 MB of free storage. It must operate on Windows 10 or later and rely on Chrome or Edge as the main user interface. These constraints restrict processing power and memory, which may impact performance as data increases. Users require basic computer literacy, requiring a simple and user-friendly interface.

The system has specific technology constraints, requiring a PHP backend and MySQL database, which limit framework options. All operations occur over Nardo's LAN using TCP/IP protocols, making access and data synchronization reliant on network stability. It must also integrate with basic peripherals like keyboards, mice, and optional barcode scanners, necessitating a lightweight, browser-based design compatible with existing hardware.

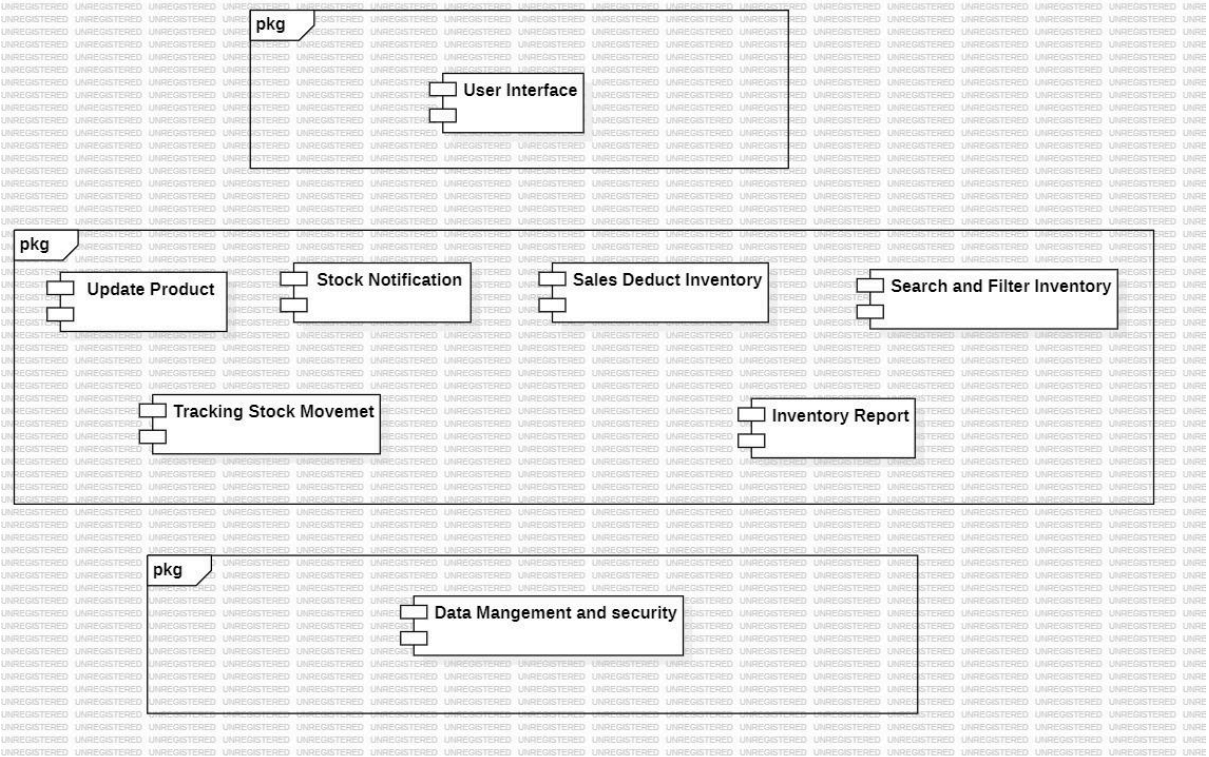
Security and performance requirements influence the system design by mandating encrypted passwords, strict role-based access, and validated inputs, adding necessary authentication and data-handling protocols.The Inventory Management System must process transactions within 3 seconds and maintain 99% uptime, requiring optimised database queries and a reliable LAN. Daily automated backups and audit logging will ensure data integrity and accountability. These requirements will create a fast, secure architecture suited for Nardo's small-scale network and allow for future scalability.

2.2 Alternatives Considered

The Model–View–Controller (MVC) architecture was evaluated. MVC allows data to change independently of its representation and supports good maintainability and reusability by separating the application into Models, Views, and Controllers. However, this model is less suited to Nardo's Inventory System because it introduces unnecessary complexity, especially given the system's simple data model and straightforward interactions. The additional layers introduced by MVC would increase code overhead without providing meaningful benefits for this small-scale project.

The Repository Architecture was also considered. This pattern manages all data in a central repository that system components access, rather than communicating directly with each other. While this structure works well for systems that require shared data across many subsystems, it creates a single point of failure: any issue in the repository affects the entire system. Additionally, distributing or scaling the repository across multiple devices is difficult and does not align with the limited LAN environment in which the system will operate.

2.3 System Architecture Diagram- Layered Architecture



2.3.1 Architectural Description

The diagram represents a three-layered architecture for the Inventory Management System. These layers consist of the Presentation Layer, Business Layer, and Persistence Layer. Each layer contains specific components that work together to meet the system's functional and non-functional requirements. The layers interact in a top-down manner, where users interact with the Presentation Layer, which forwards requests to the Business Layer, which in turn reads and writes data through the Persistence Layer.

The Presentation Layer contains the User Interface, accessed through a web browser on the shop's computer. This layer is the first point of contact between the employee and the system. It provides forms, tables, and visual elements that assist staff in performing tasks such as updating stock and viewing inventory summaries. The Presentation Layer ensures ease of use and supports usability requirements through a clean and intuitive interface. It only communicates with the Business Layer and does not contain business logic or direct database access.

The Business Layer implements the core business logic and coordinates all inventory operations between the user interface and the database. It is composed of the following logical components:

- **Update Product Component** – manages creation, editing, deactivation, and validation of product records to keep inventory data accurate.
- **Low Stock Alert Component** – monitors product quantities and triggers alerts when stock falls below the configured threshold.

- **Sales Deduct Inventory Component** – automatically reduces stock levels when a sale is recorded, preventing manual errors and maintaining real-time accuracy. **Search and Filter Component** – locates products by name, category, supplier, or stock level, reducing time spent manually searching inventory.
- **Track Stock Movement Component** – logs every addition, removal, or adjustment to stock, including user and timestamp metadata, to support accountability and auditing.
- **Summary Table / Reporting Component** – generates summary views of inventory and stock movements, enabling users to review trends and closing-time changes quickly.
- **Authentication & Authorization Component** – enforces login, encrypted passwords, and role-based access (Owner vs. Staff) to protect system data.

These components collectively ensure that the Inventory Management System operates efficiently and supports all required business functions.

The Persistence Layer of the architecture is responsible for storing all data that is processed and retrieved by the Business Layer. This layer includes the MySQL database, where product records, stock levels, transaction logs, user accounts, and audit information are securely stored. It ensures data integrity, supports concurrent access for multiple users, and enables daily backups as outlined in the SRS. All reads and writes from the Business Layer are routed through this layer using secure, validated queries.

The three layers work together in a coordinated flow. The Presentation Layer gathers input from the user and sends it to the Business Layer. The Business Layer then executes functions such as updating stock quantities, detecting low stock levels, and deducting inventory after sales. It communicates with the Persistence Layer to store and update data, ensuring accurate and real-time reflection of inventory. This structure ensures functional completeness and addresses non-functional requirements such as security (via strict login protocols, encryption, and controlled access), data integrity, maintainability, and usability through clear separation of responsibilities and consistent information flow across layers.

2.3.2 Architecture Justification

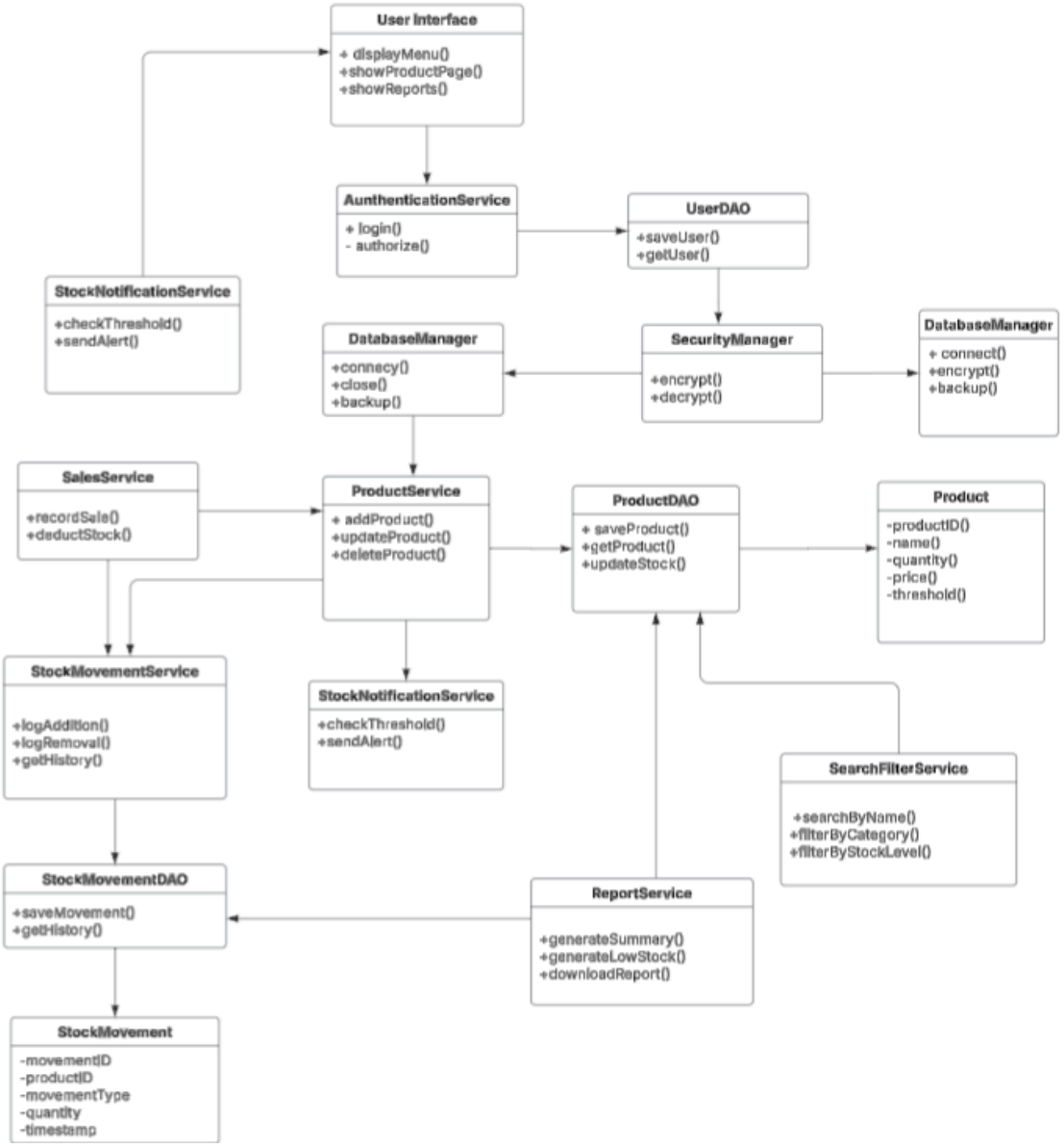
The layered architecture was chosen because it provides a clear, logical separation of responsibilities, which is essential for an inventory management system. By organising the system into the Presentation Layer, Business Layer, and Persistence Layer, each part is allowed to evolve independently. This separation supports strong maintainability, as updates to the user interface, business rules, or database structure can be made in isolation and tested cleanly. The architecture also considers Nardo's operational constraints, including limited hardware resources, a small number of concurrent users, and the need for a simple browser-based system that staff can easily learn. Unlike the more complex models, such as MVC or Repository, the layered approach offers the right balance of structure and simplicity, making it an efficient and appropriate solution for Nardo's One Stop Shop.

3.0 Architecture Decomposition

3.1 Component Decomposition – Classes

Requirement ID	Architecture Component	Class Name	Description
Req.1.1 Req 1.3	Presentation Layer	UserInterface	This will provide a graphical interface through which the user will interact with the system.
Req 1.4	Presentation Layer	AunthenticationService	This will handle the user login, validation of login credential and ensure secure user authentication.
Req 1.1	Data Access Layer	UserDAO	Manages the database operation related to the user
Req 1.2	Business Layer	StockNotificationService	This is responsible for sending an alert when stock levels reach a defined threshold.
Req 1.1 Req 1.2 Req 1.3 Req 1.4 Req 1.5 Req 1.6	Data Access Layer	Database Manager	This manages all interaction with the system database.
Req 1.4	Business Layer	Security Manager	Manages security of the system this will ensure encryption and authorization checks.
Req 1.3 Req 1.2	Business Layer	SalesService	This is responsible for recording sales and updating inventory levels.
Req 1.1	Business Layer	ProductService	Manages all product related operations which includes and is not limited to adding new products, updating product detail.
	Persistent Layer	ProductDAO	Database for product information
Req 1.1	Business Layer	Product	Provides information about products being added,updated or deleted.
Req 1.3	Business Layer	StockMovementService	Tracks the update,deletion or addition of stock
Req 1.5	Persistent Layer	StockMovementDAO	Database that stores information regarding the movement of stock
Req 1.4	Business Layer	SearchFilterService	Provides the function to search and filter through products and sales data.
Req 1.6	Business Layer	ReportService	Generates a report of how much stock was added,deleted,

3.2 Structural Design – < UML Class Diagram



3.2.1 Design Notes

Relationships Between Classes

- UserInterface → AuthenticationService
    - The UserInterface class depends on AuthenticationService to verify login attempts and determine user access levels. This ensures that no business logic or database operations occur before proper authentication, aligning with security constraints.
- AuthenticationService → UserDAO - Data Access Object

    - AuthenticationService uses UserDAO to retrieve or validate stored user credentials from the database. This separation maintains security and prevents the UserInterface from directly handling sensitive data.

- UserDAO → SecurityManager
    - UserDAO interacts with SecurityManager to hash, encrypt, or validate password data before it is stored or compared. This satisfies the non-functional security requirement for encrypted credential handling.
- SecurityManager → DatabaseManager
    - SecurityManager depends on DatabaseManager to ensure that encrypted data is stored securely and that database connections are properly opened, closed, and managed. This relationship enforces centralized control of data storage and secure communication with the database.
- UserInterface → ProductService
    - Product operations (add, edit, delete, update stock) are initiated by the UserInterface, with the UserInterface requiring the ProductService to give those operations meaning. This prevents UI-level manipulation of database content and enforces proper validation.
- ProductService → ProductDAO
    - ProductService delegates all data storage and retrieval actions to ProductDAO. This follows the Data Access Object (DAO) pattern, which isolates SQL logic from business logic and makes the system maintainable and extensible.
- ProductService → StockNotificationService
    - After updating product quantities, ProductService triggers StockNotificationService to check stock levels and generate low-stock alerts. This supports real-time stock monitoring and ensures timely alerts for employees.
- StockNotificationService → UserInterface
    - When thresholds are reached, StockNotificationService sends a message back to the User Interface to display alerts. This two-way interaction implements the Stock Notification architectural requirement.
- SalesService → ProductService
    - When recording a sale, SalesService must deduct stock through ProductService to maintain consistency and maintain a single authority for stock updates.
- SalesService → StockMovementService
    - Each sale must be logged for accountability, so SalesService also interacts with StockMovementService to record the change.

- StockMovementService → StockMovementDAO
    - StockMovementService uses StockMovementDAO to store logs of inventory changes (additions, removals, sales). This supports auditing and the requirement to track all stock movement events.
- SearchFilterService → ProductDAO
    - SearchService retrieves filtered or searched product data through ProductDAO. This separation ensures efficient queries without mixing UI logic or business rules directly with database code.

Assumptions Impacting the UML Class Diagram:

- MySQL is the database, therefore DAOs were required for structured database access.
- Users must be authenticated before accessing inventory features, requiring dedicated Authentication class
- Only authorized staff can manipulate products, so the system models user roles through Authorization logic.
- Stock changes must always be logged, leading to the creation of StockMovementService and StockMovementDAO.

Constraints Shaping the Design of the UML Class Diagram:

- Security Constraints
    - Sensitive data must be encrypted, justifying the inclusion of SecurityManager.
    - All data access must be verified through the various DAOs to satisfy integrity and secure connection handling.
- Maintainability and Modularity
    - UI is completely separated from business logic.
    - Business logic is separated from data access via the DAO layer.
      Each module handles a single responsibility (Single Responsibility Principle).
- Auditing and Traceability
    - StockMovement entity requires movementID, movementType, and timestamp to meet auditing requirements.
    - All stock-related actions must be logged in chronological order.
- Scalability
    - The system is designed so that new services (e.g., an email alert service) can be added without altering core functionality.

Why These Relationships Were Specified

- To enforce separation of concerns, improving maintainability.
- To implement security, ensuring authentication and encryption are handled correctly.
- To ensure traceability, allowing full audit trails for stock movement.
- To satisfy functional requirements, such as:
    - Stock deduction on sale
    - Low-stock alerts
    - Searching and filtering
    - Inventory reporting

Every arrow in the diagram represents a dependency aligned directly with a requirement in the architecture and SRS.

Test Plan
for
Nardo's Inventory System

Version 1.0

Prepared by

GROUP: Mon_3-5_G02

Melissa Williams	**620150166**	**melissawilliams046@gmail.com**
Alonzo Brown	**620173600**	**ajanidbrown96@gmail.com**
Elizabeth Wright	**620170287**	**liz.wright5665@gmail.com**
Jevaughn Johnson	**620173043**	**johnsonjevaughn63@gmail.com**
Madison Sujanani	**620183857**	**madisonsujanani@gmail.com**

Course Instructor: **Dr. Ricardo Anderson**
Course: **COMP2140 – Introduction to Software Engineering**
Studio Facilitator: **Mr. Eyton Ferguson**
Date: **December 3, 2025**

Introduction

This test plan outlines how the Nardo's Inventory System will be evaluated to ensure all major functional and non-functional requirements have been implemented correctly. Testing includes both valid and invalid scenarios to ensure proper responses.

Test Execution Process

Testing will be performed manually using the system's browser interface. Each test case includes its associated requirement, test data, expected and actual results, and a pass/fail indicator. A test passes when actual results match expected results.

Test Case Table

Test Case	Test Data	Assoc. Requirement	Expected Result	Actual Result	Pass/ Fail	Comment	Screenshot
Case 1: Valid Login	username: owner1, password: validPwd	Req 1.1 (Authentication)	Access granted, user redirected to dashboard				
Case 2: Invalid Login – Wrong Password	username: owner1, password: wrongPwd	Req 1.1	Access denied, error message displayed				
Case 3: Invalid Login – Empty Fields	username: " ", password: " "	Req 1.1	System prompts user to enter required data				

Test Case	Test Data	Assoc. Requirement	Expected Result	Actual Result	Pass/ Fail	Comment	Screenshot
Case 4: Add New Product with Valid Data	product: “Chips”, qty: 20, min: 5, category: Snacks	Req 1.1 (Update Product)	Product saved and appears in list				
Case 5: Add Product with Missing Required Field	product: name empty, qty: 10, min: 3	Req 1.1	System prevents saving and displays validation message				
Case 6: Prevent Duplicate Product	product: “Chips” already exists	Req 1.1	System rejects duplicate entry				
Case 7: Edit Product Quantity (Valid Update)	edit “Chips” qty from 20 → 15	Req 1.1	Quantity updated and reflected immediately				
Case 8: Deactivate Product with Sales History	product: “Water Bottle”, status change to inactive	Non-functional: Data Integrity	Product becomes inactive, remains in history				
Case 9: Manual Low Stock Alert Trigger	Update qty: 12 → 4 (min threshold: 5)	Req 1.2 (Low Stock Alert)	System generates low-stock alert and displays it on UI				

Test Case	Test Data	Assoc. Requirement	Expected Result	Actual Result	Pass/ Fail	Comment	Screenshot
Case 10: Low Stock Not Triggered When Above Threshold	Update qty: 12 → 10 (min=5)	Req 1.2	No alert generated				
Case 11: Sales Deduct Inventory – Valid Sale	product: qty=20, saleQty=3	Req 1.3 (Sales Deduction)	New qty = 17; sale logged; stock movement recorded				
Case 12: Sales Deduction Triggering Low Stock	qty=7, saleQty=3, min=5	Req 1.3 + Req 1.2	New qty=4 and low stock alert generated				
Case 13: Prevent Sale Exceeding Quantity	qty=2, saleQty=5	Req 1.3	Sale rejected, error displayed, qty unchanged				
Case 14: Search by Product Name	search: “Chips”	Req 1.4 (Search /Filter)	Relevant product appears in results				
Case 15: Filter by Category	category: “Snacks”	Req 1.4	Only products matching category displayed				

Test Case	Test Data	Assoc. Requirement	Expected Result	Actual Result	Pass/ Fail	Comment	Screenshot
Case 16: Search with Empty Query	search: “”	Req 1.4	System returns either full list or prompt message				
Case 17: View Stock Movement History	product: “Chips”	Req 1.5 (Track Movement)	List shows additions, sales, edits in order				
Case 18: History Cannot Be Edited	attempt to edit stock movement entry	Req 1.5	System prevents editing; log is read-only				
Case 19: Generate Summary Table on Exit	at system logout with activity history	Req 1.6 (Reports)	Summary appears within few seconds with accurate data				
Case 20: Skip Summary Table	choose “No” when prompted	Req 1.6	System closes without displaying table				
Case 21: Role Restriction – Staff Cannot Add Products	login as Staff	Non-functional: Authorization	Add/edit product buttons disabled or blocked				

Test Case	Test Data	Assoc. Requirement	Expected Result	Actual Result	Pass/ Fail	Comment	Screenshot
Case 22: Password Encryption (Indirect Test)	inspect database record	Security Requirement	Stored password is encrypted, not plaintext				
Case 23: Session Timeout After Inactivity	15 minutes idle	Security Requirement	User is logged out and must re-login				

GitHub Repo: https://github.com/mads966/Nardo-s-Inventory-System.git