S/W Detailed Level Design

Project Name	Online Store		
Block Name			
Author	**GRUPPA**	**Approver**	Anna Kyselova
Team	3		

This document represents Detailed Level Design (DLD). It describes the detailed system design and implementation plan in alignment with Agile principles. The DLD is updated incrementally with each release to reflect system evolution.

Contents

■ Revision History

Version	Date	Revised contents	Author	Approver

■ Terms and Abbreviations

Term	Description

■ References

1. SW Requirements Specification

1. Overview

This document specifies the detailed design for a console-based e-commerce prototype written in C++. It defines classes, responsibilities, data structures, control flows, and file mappings. The system currently supports:

- Basic login (admin vs customer by username)
- Admin product management (seeded catalog, add product, list products)
- Customer area skeleton (menus only)

The DLD aligns with incremental Agile delivery. It evolves per release:

Release 4 – Persistent Data Storage

All exchange records are now stored in a text file and reconstructed into dynamic memory at program start. The Repository remains the central in-memory component and now manages both:

- In-memory data (dynamic array of Transaction objects)
- Persistent data (text file: data.txt)

Data transfer model:

- File ⟶ loadAll() ⟶ Repository
- Repository ⟶ saveAll() ⟶ File

Stakeholders and roles:

- Administrator: seeds and manages products.
- Customer: browses and (future) places orders.
- Developers/QA: implement and verify features.
- Product Owner: prioritizes backlog.

Out of scope for the current release:

- Persistent storage
- Order placement and management
- Full search
- Authentication/authorization beyond username prompt

2. System Overview / Architectural Context

Design follows a simple layered structure:

- Presentation Layer:
  - app/main.cpp (program entry point, console I/O)
- Service/Logic Layer:
  - services/LoginService
  - services/AdminService
  - services/CustomerService
  - services/SearchService (stub)
- Domain Layer:
  - domain/Product, domain/Order (+ OrderStatus), domain/Customer, domain/Administrator
  - domain/types (enums, structs; ReportStruct stub)

- Repository / Persistence Extension:
  - A lightweight persistence mechanism is added without altering existing classes or relationships.
  - The Repository now provides four additional operations required for loading and saving domain objects:
  - – loadAll()
  - – saveAll()
  - – parseLine()
  - – toLine()
  - These functions operate on plain-text representations of domain objects and extend the current architecture without modifying service logic or domain models. Services still work with in-memory collections; persistence is an optional extension beneath them.

Dependency directions (one-way): app → services → domain

Simple schematic:

1. main → LoginService
2. LoginService → (AdminService, Administrator) or (CustomerService, Customer)
3. AdminService ↔ Product collection (in-memory)
4. CustomerService ↔ Product collection (in-memory)
5. Order aggregates Product (not yet used by services)
6. Repository: loadAll / saveAll / parseLine / toLine (optional persistence for collections)

3. Class Specifications

Class: Administrator
Type: Concrete domain entity
Purpose: Represents an admin user (identity only, for routing)
Attributes:

- `username: string`
  Methods:
- `Administrator(const string& username)`
  Constraints:
- Non-empty username

Class: Customer
Type: Concrete domain entity
Purpose: Represents a customer using the app
Attributes:

- `username: string`
  Methods:
- `Customer(const string& username)`
- `string getUsername() const`
  Constraints:
- Non-empty username

Class: Product
Type: Concrete domain entity (value-like)
Purpose: Products offered for sale
Attributes:

- `id: int` (positive, unique within catalog)
- `name: string` (non-empty)
- `description: string`
- `price: double` (>= 0)
- `quantity: int` (>= 0)
  Methods:
- `Product(int id, const string&, const string&, double price, int quantity)`
- Getters: `getId`, `getName`, `getDescription`, `getPrice`, `getQuantity`
- Setters: `setName`, `setDescription`, `setPrice`, `setQuantity`
  Invariants:

- `price >= 0`
- `quantity >= 0`

Class: Order
Type: Aggregate domain entity
Purpose: Represents a purchase order comprising multiple products
Attributes:

- `id: int` (positive, unique per store)
- `products: vector<Product>` (non-empty)
- `delivery_address: string` (non-empty)
- `total_price: double` (>= 0, auto-calculated from products' prices)
- `order_time: chrono::system_clock::time_point`
- `delivery_date: chrono::system_clock::time_point` (>= order_time)
- `status: OrderStatus`
  Methods:
- Constructor with auto total_price calculation
- Full set of getters/setters; `setProducts` recalculates `total_price`
  Invariants/Contracts:
- `delivery_date >= order_time`
- `total_price == sum(products.price)`
- `status ∈ {Scheduled, Delivered, Canceled}`

Enum: OrderStatus
Values: `Scheduled=1`, `Delivered=2`, `Canceled=3`

Class: LoginService
Type: Service
Purpose: Route user to admin or customer flows based on username
Methods:

- `void loginMenu()`

Class: AdminService
Type: Service
Purpose: Manage catalog (in-memory)
Attributes:

- `vector<Product> products`
  Methods:

- `AdminService()` — seeds products
- `void loadProducts()` — populate products with static catalog
- `void addProduct()` — console workflow to create product, append to vector
- `void adminMenu()` — loop for admin actions
  Constraints:
- Generated `id = products.size()+1` (risk of collision if deletions added later)
- Validate price, quantity; handle input errors

Class: CustomerService
Type: Service
Purpose: Customer operations (skeleton)
Attributes:

- `vector<Product> products`
  Methods:
- `CustomerService()` — seeds products
- `void loadProducts()`
- `void customerMenu()`

Class: SearchService
Type: Service (stub)
Purpose: Placeholder for product search capabilities

Class: Repository
Type: Persistence / in-memory manager
Purpose: Centralized storage for dynamic and persistent data
Attributes:

- `data` — dynamic array of Transaction objects
- `count` — number of valid records currently loaded
- `capacity` — allocated array size
  Methods:
- `loadAll()` — read from text file into memory, skip invalid lines, no exceptions thrown
- `saveAll()` — full rewrite of text file from memory
- `parseLine(const string&)` — convert a line of text into a Transaction object

- `toLine(const Transaction&)` — convert a Transaction object into a text line

  Behavioural constraints:
- `capacity ≥ count`
- `loadAll` skips invalid or malformed lines; exceptions should not propagate
- `saveAll` overwrites the file with current in-memory data

5. Interfaces and Abstractions

Planned abstractions to decouple I/O, time, and storage:

IProductRepository

- Purpose: Abstract product storage (file/DB/memory)
- Key Methods: `getAll()`, `add(Product)`, `update(Product)`, `remove(int)`, `findById(int)`
- Planned For: Release 2

IOrderRepository

- Purpose: Persist and retrieve orders
- Key Methods: `add(Order)`, `getById(int)`, `listByCustomer(string)`, `updateStatus(int, OrderStatus)`
- Planned For: Release 3

IClock

- Purpose: Time abstraction for testing order dates
- Key Methods: `now()`
- Planned For: Release 3

IConsole (or IIO)

- Purpose: Abstract console input/output for testing
- Key Methods: `readLine()`, `readNumber<T>()`, `write(string)`
- Planned For: Release 2

ISearchService

- Purpose: Search/filter products
- Key Methods: `searchByName(string)`, `filterByPrice(min,max)`
- Planned For: Release 3

Repository (Release 4)

- Purpose: Central in-memory manager with optional persistent storage
- Key Methods:
  - `loadAll()`
    - Purpose: Load all Transaction records from text file into memory
    - Inputs: None
    - Outputs: Array of Transaction objects stored in `data`
    - Behaviour: Invalid/malformed lines are skipped; no exceptions propagated
  - `saveAll()`
    - Purpose: Save all in-memory Transaction records to text file
    - Inputs: Array of Transaction objects in `data`
    - Outputs: Overwrites file contents (`data.txt`)
    - Behaviour: Full rewrite; preserves only valid in-memory records
  - `parseLine(const string&)`
    - Purpose: Convert a single line of text into a Transaction object
    - Inputs: Text line from file
    - Outputs: Transaction object or `null`/skip if invalid
    - Behaviour: Lines that cannot be parsed are skipped silently
  - `toLine(const Transaction&)`
    - Purpose: Convert a Transaction object into a text line for storage
    - Inputs: Transaction object
    - Outputs: Formatted string line ready for writing to file

6. Function Responsibilities

Class	Method	Purpose	Input	Output	Notes
LoginService	loginMenu	Prompt username; dispatch to admin/customer menus	stdin username	none	username=="admin" => AdminService
AdminService	AdminService	Construct and seed products	-	-	Calls loadProducts
AdminService	loadProducts	Seed vector with predefined items	-	10 items seeded	-
AdminService	addProduct	Interactive add; confirm/save/edit/cancel	name, description, price, quantity	-	Generates id; prints result

AdminService	adminMenu	Menu loop: add product, view products, logout	numeric choice	-	Prints product list
CustomerService	CustomerService	Construct and seed products	-	-	Calls loadProducts
CustomerService	loadProducts	Seed vector	-	-	Mirrors AdminService seeds
CustomerService	customerMenu	Menu loop (skeleton)	numeric choice	-	View Orders/View products TBD
Order	constructor	Build order and compute total	id, products, address, times, status	Order	total_price = sum(products.price)
Order	setProducts	Replace products and recompute total	vector	-	Recalculates total

7. Operation Flow

Startup Flow

- File (`data.txt`) opens and provides lines.
- Each line is validated and converted using `Repository::parseLine()`.
- Valid objects are stored in memory (`Repository::data`).

Shutdown Flow

- Objects in memory are formatted using `Repository::toLine()`.
- File is rewritten completely via `Repository::saveAll()`.

Login and Routing

- `main.cpp` starts the program.
- `LoginService::loginMenu()` prompts: “Enter your username:”
- If `username == "admin"`:
  - Create `Administrator("admin")`
  - Create `AdminService`
  - Enter `AdminService::adminMenu()` loop
- Else:
  - Create `Customer(username)`
  - Create `CustomerService`
  - Enter `CustomerService::customerMenu()` loop

Admin Add Product Flow

- From `adminMenu`, select "Add product"
- `addProduct()` reads: `name`, `description`, `price`, `quantity`
- Show review + confirmation menu:
  - **Save:** Append `Product(id=products.size()+1, ...)` to in-memory vector; display created product details
  - **Cancel:** Print "Operation cancelled" and return to `adminMenu`
  - **Edit:** Restart capture loop

Data Path Example (Read-Only Product List)
ConsoleUI → AdminService → products (in-memory vector) → Console output

8. Enumerations & Constants

Name	Value / Type	Description
OrderStatus enum class {Scheduled=1, Delivered=2, Canceled=3} Order lifecycle states	OrderStatus enum class {Scheduled=1, Delivered=2, Canceled=3} Order lifecycle states	OrderStatus enum class {Scheduled=1, Delivered=2, Canceled=3} Order lifecycle states

9. Validation Rules & Future Work

Validation Rules

Product

- `id`: positive integer, unique within repository
- `name`: non-empty, length ≤ 128
- `description`: length ≤ 1024
- `price`: ≥ 0, reject NaN/INF
- `quantity`: ≥ 0

Order

- `products`: non-empty
- `delivery_address`: non-empty, reasonable length
- `total_price`: auto-calculated; not directly set by services
- `delivery_date` ≥ `order_time`

Login

- `username`: non-empty, trimmed; case-sensitive "admin" for admin

Input / Console I/O

- Clear input state after extraction failures
- Use `std::getline` for strings; validate numeric conversion
- Use loops instead of recursive menu re-entry to prevent stack growth
- Display user-friendly error messages for invalid input

Persistence Rules (Release 4)

- Each file line must contain the correct number of fields
- Currency codes and numeric values must follow correct formats
- Invalid or malformed lines are ignored during load
- Empty file results in an empty Repository
- All rules extend the validation philosophy established in Release 3

Future Work

Release 3 (mid-term)

- Order placement: `createOrder(Customer, cart, address)` and list orders
- Implement `IOrderRepository` for file/DB persistence
- Use `IClock` for deterministic timestamps
- Enforce status transitions and invariants
- Generate reports (daily sales, inventory)

Release 4 (optional / beyond current scope)

- Authentication with passwords and roles
- Internationalization and currency formatting
- Inventory reservations
- Repository-assigned or GUID-based ID generation to replace `products.size()+1`

Technical Debt & Cleanup

- Remove stray semicolons and duplicated headers in source files
- Correct header guards and avoid `using namespace std` in headers
- Implement or remove empty/placeholder files (e.g., `ReportStruct.h`)

- Ensure menu options in CustomerService are fully implemented

10. Traceability Matrix

Requirement (SRS)	Class / Method (DLD)
SRS-001: User can log in to the system	LoginService::loginMenu()
SRS-002: Admin can view the product catalog	AdminService::adminMenu() → list products
SRS-003: Admin can add a new product	AdminService::addProduct(), Product constructor
SRS-004: System maintains product data	AdminService::products (in-memory), loadProducts(), Repository::loadAll(), Repository::saveAll()
SRS-005: Customer can access customer menu	CustomerService::customerMenu()
SRS-010: Orders have lifecycle statuses	OrderStatus enum, Order::getStatus(), Order::setStatus()
SRS-011: Order total equals sum of product prices	Order constructor, Order::setProducts() (recompute)
SRS-020: System records order and delivery times	Order::order_time, Order::delivery_date
SRS-030: Product data validation	Section 9 rules; enforced in AdminService::addProduct(), CustomerService inputs, Repository::parseLine()
SRS-031: Handle malformed or invalid data	Repository::parseLine(), Repository::loadAll()

11. Code Structure and File Mapping

Class	File
main	src/app/main.cpp
LoginService	src/services/LoginService.h/.cpp
AdminService	src/services/AdminService.h/.cpp
CustomerService	src/services/CustomerService.h/.cpp
SearchService	src/services/SearchService.h/.cpp
Administrator	src/domain/Administrator.h/.cpp
Customer	src/domain/Customer.h/.cpp
Product	src/domain/Product.h/.cpp

Class	File
Order	src/domain/Order.h/.cpp
OrderStatus enum	src/domain/types/OrderStatusEnum.h
ReportStruct (stub)	src/domain/types/ReportStruct.h

12. Revision History

Date	Version	Change Summary	Author
28.10	2	Initial DLD from provided codebase; added plans, validation, and tech-debt notes	GRUPPA

Release 3

13. Validation Rules & Preconditions/Postconditions

< Describe how your system checks input data and enforces correctness before and after operations. Each method should list clear preconditions (what must be true before it runs) and postconditions (what must be true after it completes). Indicate the layer where validation happens - UI, Logic, or Repository. Use the same class and method names as in Release 2. >

For each key method involved in the operation, write preconditions and postconditions and indicate the validation level (UI, Logic, Repository). Use a single sentence explanation for why validation happens at that layer.

Class	Method	Preconditions	Postconditions	Validation Level (UI/Logic/Repo)	Explanation
StoreService	readNonEmptyString	Input must not be empty	Empty → retry Not empty → accept	UI	Input must not be empty
StoreService	validatePrice	Price given to logic	Invalid → throw Valid → continue	Logic	Price must be higher than zero
StoreService	productExists	ID passed to repository	No ID → fail Exists → modify	Repository	Entity must exist before modification
Repository	loadAll	File exists or empty; Repository initialized	Repository data/ populated with valid Transactions; count updated; invalid lines skipped	Repo	Loads all persistent records into memory; ignores malformed lines
Repository	saveAll	Repository data/ contains valid Transactions	File overwritten with current in-memory records	Repo	Persists all in-memory transactions to file; performs rewrite;
Repository	parseLine	Line string contains expected number of fields	Returns valid Transaction object	Repo	Converts a text line into a Transaction
Repository	toLine	Valid Transaction object	Returns correctly formatted text line for storage	Repo	Converts a Transaction to a string line

14. Behavioral Models

< Attach two UML diagrams that describe one key operation in your system, for example, processing an order, confirming a booking, or updating a record. Both diagrams must describe the same operation to show consistency.

- Activity Diagram – show the internal logic of the operation, including both normal and error flows. Mark decision nodes as preconditions and final nodes as postconditions.
- Sequence Diagram – show how UI ⟶ Logic ⟶ Repository interact and how exceptions propagate upward to the UI layer. >

Create an Activity Diagram and a Sequence Diagram for the same operation. Mark decisions as preconditions and end nodes as postconditions.
In Sequence, show UI ⟶ Logic ⟶ Repository and exception propagation back to UI.

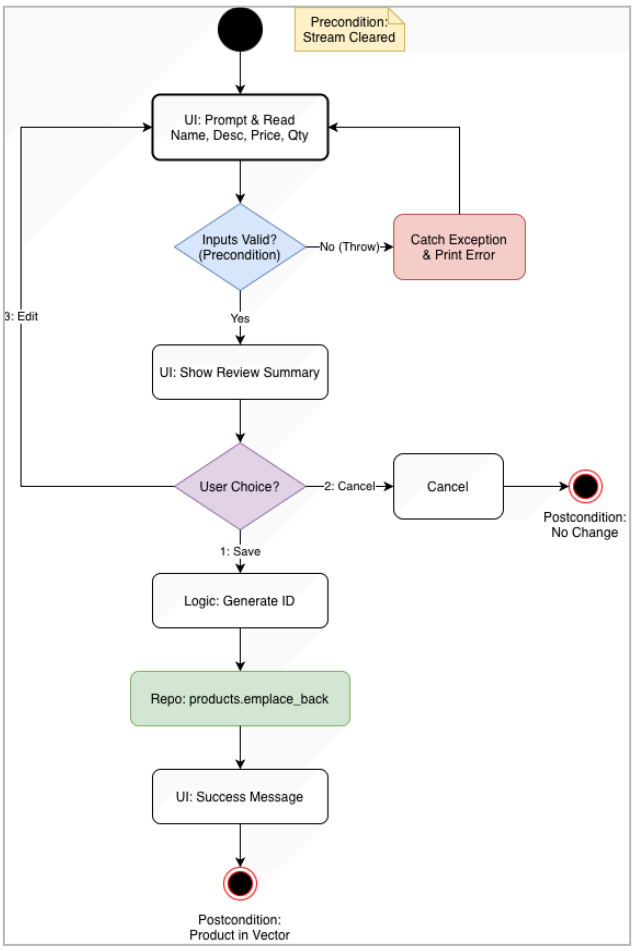

Figure: Activity diagram

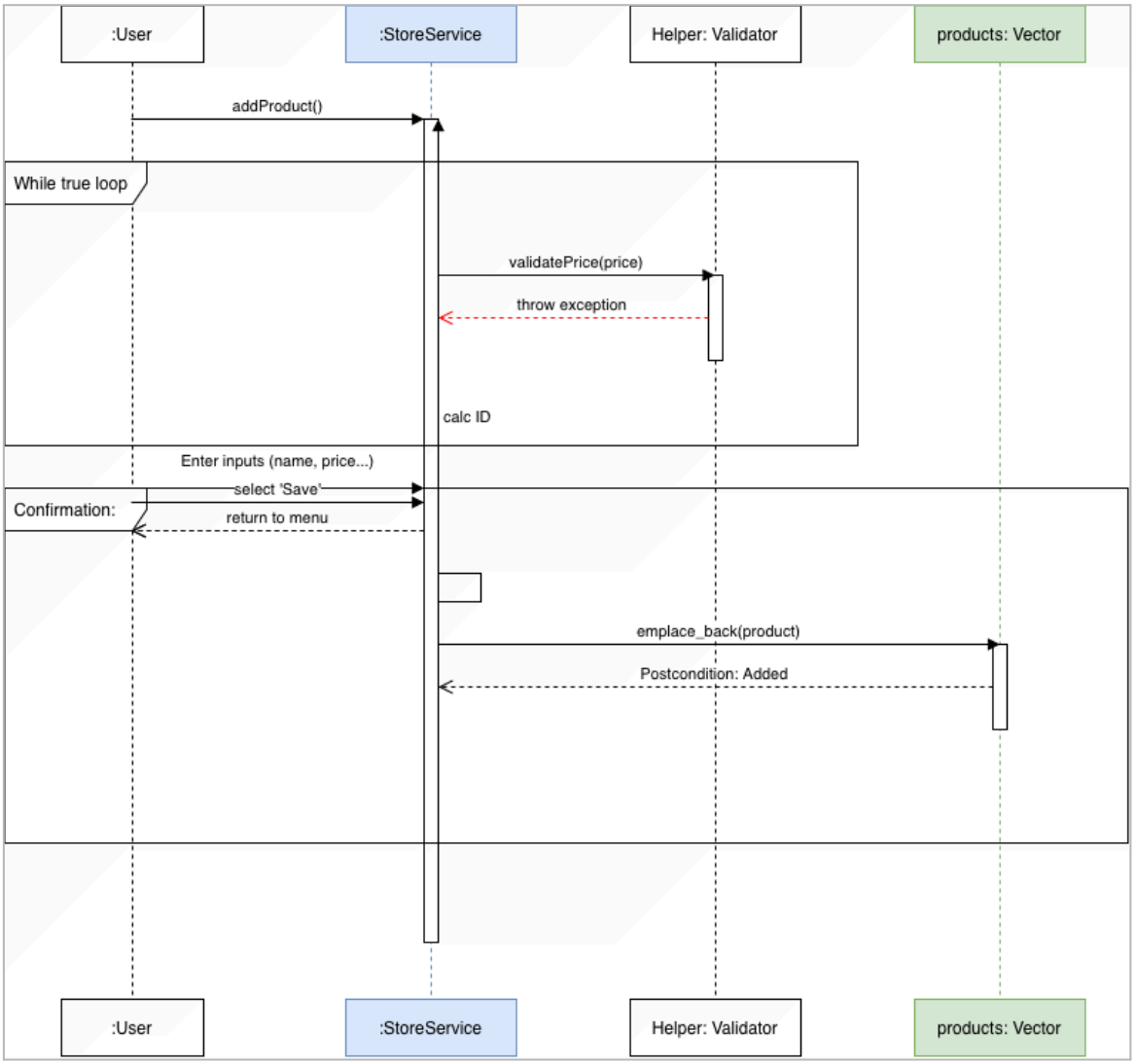

Figure: Sequence diagram

15. Error & Exception Handling Policy

< List all exceptions that may occur in your system. Describe where each one is thrown, where it is caught, what message it shows, and what the program does afterwards. For this release, use only the three architectural layers: UI, Logic, Repository. >

List exceptions that can happen in this operation. For each, specify where it is thrown, where it is caught, the user message, and the default action (message, stop, retry).

Exception Type	Thrown By (Layer / Class)	Caught At (Layer)	Message / what()	Default Action (message / stop / retry)
InvalidQuantity	Logic / StoreService::validateQuantity	UI	"Quantity must be greater than 0."	Show message → ask user to re-enter
UsersFileMissing	Repository / LoginService::ensureUsersFile	UI	"Cannot open users file at specified path."	Show message → stop operation
MalformedFileLine	Repository::loadAll	Repo (internal)	"Line ignored due to invalid format."	Skip line silently; continue processing
FileWriteError	Repository::saveAll	UI	"Error writing to data file."	Show message → stop operation

16. Revision History

< Track document changes across releases. >

Date	Version	Change Summary	Author
25.11	3	Added error handling chapters	GRUPPA members
15.12	4	Extended the system with persistent storage and a complete file-to-memory lifecycle.	GRUPPA members