Deliverable 3 – Final Submission Documentation

December 19th, 2025

The University of Western Ontario
Department of Computer Science
CS 3307A - “Object-Oriented Design and Analysis”
Dr. Umair Rehman

- Group Members -

Omar Hossain, 251372369, ohossai2@uwo.ca

Ahmed Sinjab, 251381170, asinjab@uwo.ca

Table of Contents

Overview

This document presents the final implementation of our Inventory Management System for CS3307A. The system is now feature-complete, with all requirements from Deliverable 1 fulfilled. Building upon the 50% completion achieved in Deliverable 2, we have successfully implemented data persistence, comprehensive unit testing, a formal Factory pattern, and numerous UI enhancements.

Key Accomplishments

- Complete System Implementation: All core features operational, including product management, supplier tracking, user authentication, and inventory operations
- Factory Pattern: Implemented ProductFactory class for centralized object creation
- Data Persistence: Full file I/O system using JSON format for saving/loading inventory state
- Comprehensive Testing: 58 unit tests achieving 100% pass rate using GoogleTest and Google Mock
- Professional Documentation: Complete UML diagrams, design pattern explanations, and testing reports

System Architecture Overview

Our system maintains the three-layer architecture established in Deliverable 2, now enhanced with data persistence capabilities.

Presentation Layer (UI)

Components

- LoginDialog: Handles user authentication
- Dialogue (Main Window): Primary interface for inventory operations
- Qt Widgets framework for all UI elements

Responsibilities

- User input validation
- Display logic and event handling
- Delegates all business operations to the core layer

- No business logic in UI components

Key Features

- Add/Remove products with validation
- Buy/Sell product operations
- View inventory with filtering
- Save/Load inventory state
- User authentication and password management

Business Logic Layer (Core)

Components

- Product hierarchy (Product interface → Perishable, Food, Chemical, RawMaterial, NonPerishable)
- Inventory class (acts as facade) Supplier class
- User and Admin classes
- ProductFactory class
- FileManager class

Responsibilities

- All domain logic and business rules
- Product and supplier management
- Inventory operations (buy, sell, check low stock)
- Data validation
- User authentication logic

Data Persistence Layer

Implementation

- Format: JSON for human-readable, structured data
- FileManager Class: Centralized I/O operations
- Storage: Saves products, suppliers, and user data
- Error Handling: Graceful handling of file read/write errors

Key Methods

- bool FileManager::saveInventory(const std::string& filename, Inventory* inventory);
- bool FileManager::loadInventory(const std::string& filename, Inventory* inventory);

What is Persisted

- All product data (including type-specific attributes)
- Supplier information and ratings
- Product-supplier relationships
- User credentials (encrypted)

Data Flow

User Action → UI → Business Logic → Data Persistence

1. User Interaction: User performs action in Qt interface (e.g., "Add Product")
2. Input Validation: UI validates input format
3. Business Logic: Request forwarded to the Inventory class
4. Product Creation: ProductFactory creates an appropriate product type State
5. Update: Inventory updates internal state
6. Persistence: Changes are optionally saved to the file
7. UI Update: Interface refreshed to reflect changes

Example Flow - Adding a Food Product:

User clicks "Add Product"
→ Dialogue validates input fields
→ ProductFactory::createProduct(productType::FOOD, ...)
→ Inventory::addProduct(newFood)
→ Optional: FileManager::saveInventory(...)
→ UI table refreshes with new product

Use Case Diagrams

The use case diagrams will identify the main actors and their respective interactions with the system. It shows how different users will interact with the system's core functionalities, ensuring that the system is designed with clarity, usability, and scalability in mind.

Actors

User

The User actor represents any standard system user, such as an inventory staff member or employee, who engages with the system to perform routine tasks. Users have access to essential functionalities that allow them to interact with the inventory practically and efficiently. This role is focused on operational interactions with the system, ensuring users can access the information they need while maintaining data consistency and accuracy.

Admin

The Admin actor represents users with extra privileges, such as inventory managers, supervisors, or business owners. Admins inherit all the capabilities of a regular User but are also responsible for maintaining, configuring, and supervising the system at a higher level. The admin role is designed to ensure smooth system operations, effective oversight of resources and long-term maintainability of inventory processes.

Use Case Diagram 1

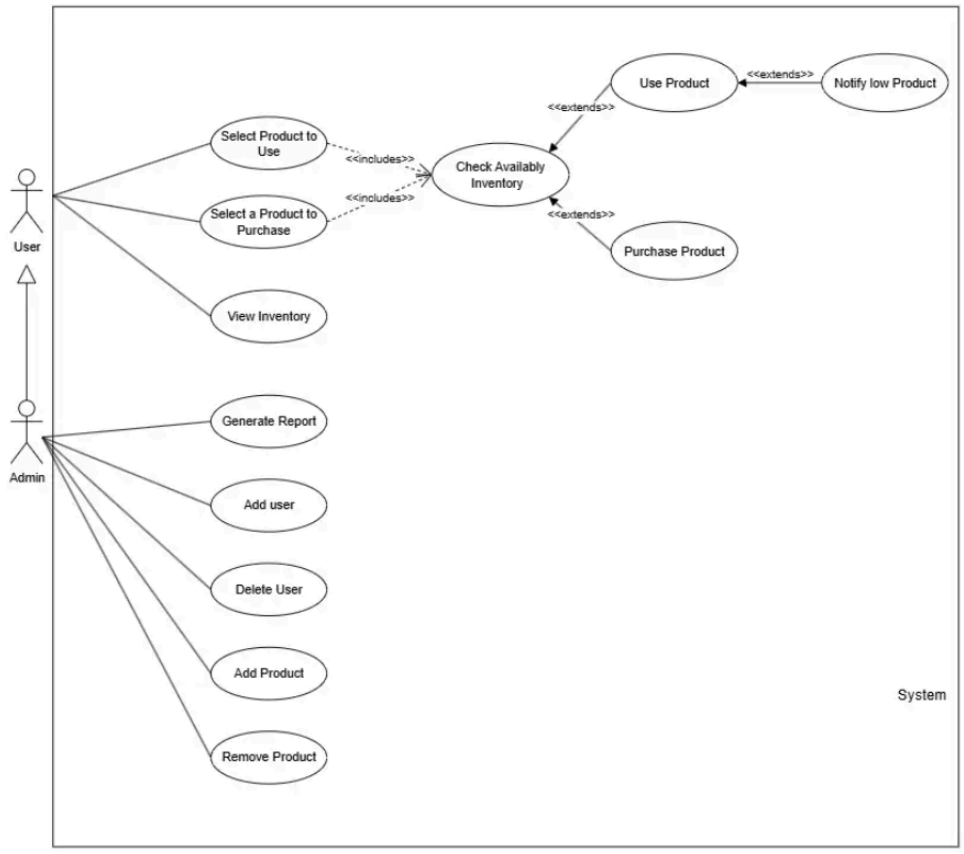

Figure 1.1

The use case diagram, presented in Figure 1.1, demonstrates how different actors interact with the inventory management system. For Users, the diagram shows the core interactions they can perform, such as viewing inventory, searching for products, and making purchases. For admins, this diagram extends these capabilities, demonstrating the additional actions they can take, including managing products, handling user accounts, and generating analytical reports.

Use Case Diagram 2

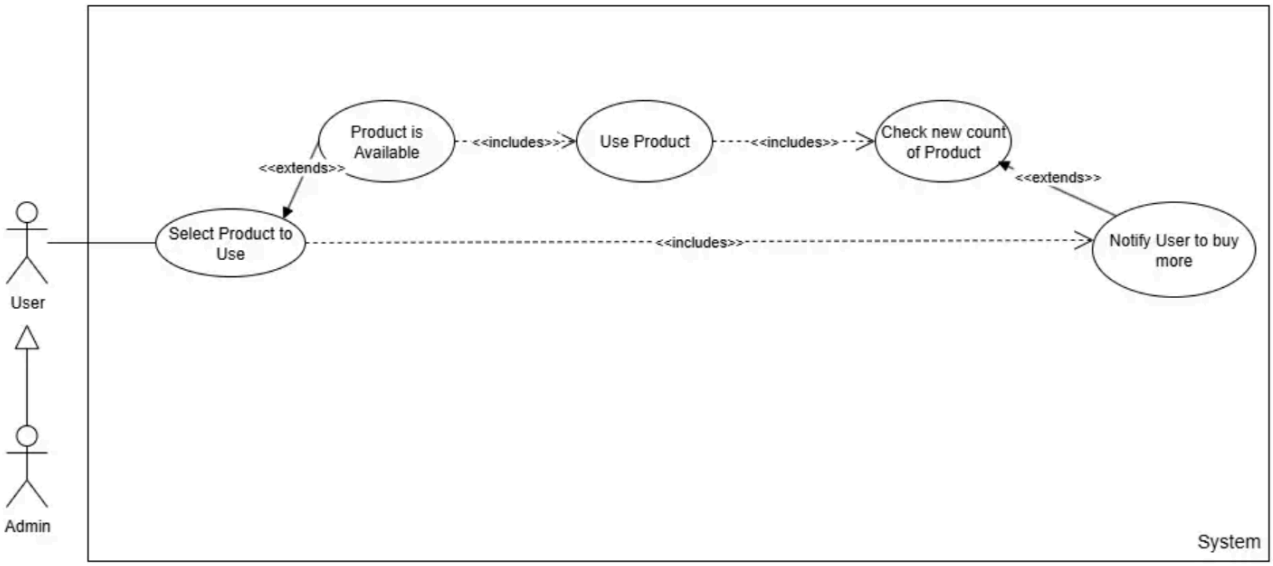

Figure 1.2

Table 1.1

Use Case	Using A Product
Primary Actor	User
Secondary Actor	Admin
Goal in Context	Allow the user to utilize a product from the inventory while ensuring accurate updates to stock levels.
Preconditions	- The inventory system is running and initialized correctly. - The active user is authenticated and authorized.
Trigger	The user initiates the “Use Product” action from the system interface.

Scenario	1. The user selects the “Use Product” option. 2. The user chooses a product from the inventory list. 3. The system checks the requested quantity against available stock. 4. If sufficient stock exists, the system deducts the specified amount from the inventory. 5. The system confirms the successful transaction and notifies the user. 6. The system re-evaluates the product’s stock level to determine if it has reached the minimum threshold or has been depleted. 7. If the threshold is breached, the system notifies the user that restocking is required.
Exception	1. The user selects the “Use Product” option. 2. The user chooses a product from the inventory list. 3. The requested quantity exceeds the available stock. 4. The system notifies the user of insufficient stock and prompts them to reorder or restock the item.
Priority	High: This is a critical function as it directly affects inventory accuracy and product availability.

Use Case Diagram 3

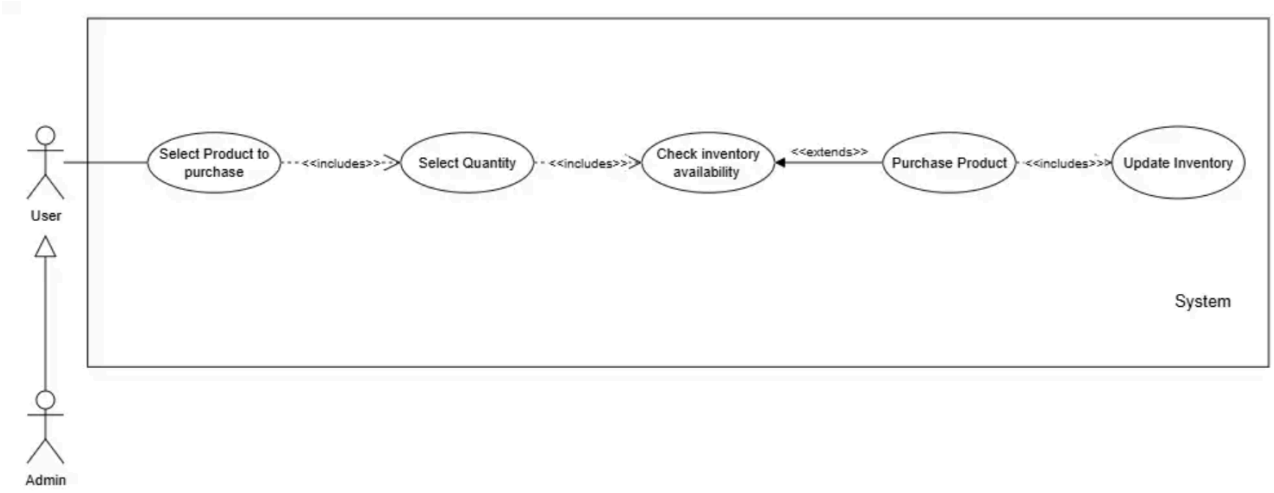

Figure 1.3
Table 1.2

Use Case	Buying Product
Primary Actor	User
Secondary Actor	Admin
Goal in Context	Allow the user to purchase a specified quantity of a product while ensuring inventory levels are updated accurately and storage constraints are respected.
Preconditions	- The inventory system is active and initialized correctly. - The active user is authenticated and authorized.
Trigger	The user initiates the “Purchase Product” action from the system interface.
Scenario	1. The user selects the “Purchase Product” option. 2. The user chooses a product to purchase from the inventory list. 3. The system verifies that sufficient storage space is available to

	accommodate the additional quantity. 4. If storage capacity allows, the system updates the inventory to reflect the purchase. 5. The system notifies the user of a successful purchase transaction.
Exception	1. The user selects the “Purchase Product” option. 2. The user chooses a product to purchase. 3. The system determines that there is insufficient storage capacity for the requested quantity. 4. The system notifies the user that the purchase could not be completed due to storage limitations.
Priority	High: This function is essential, as it directly impacts the ability to restock inventory and maintain product availability.

Class Diagram

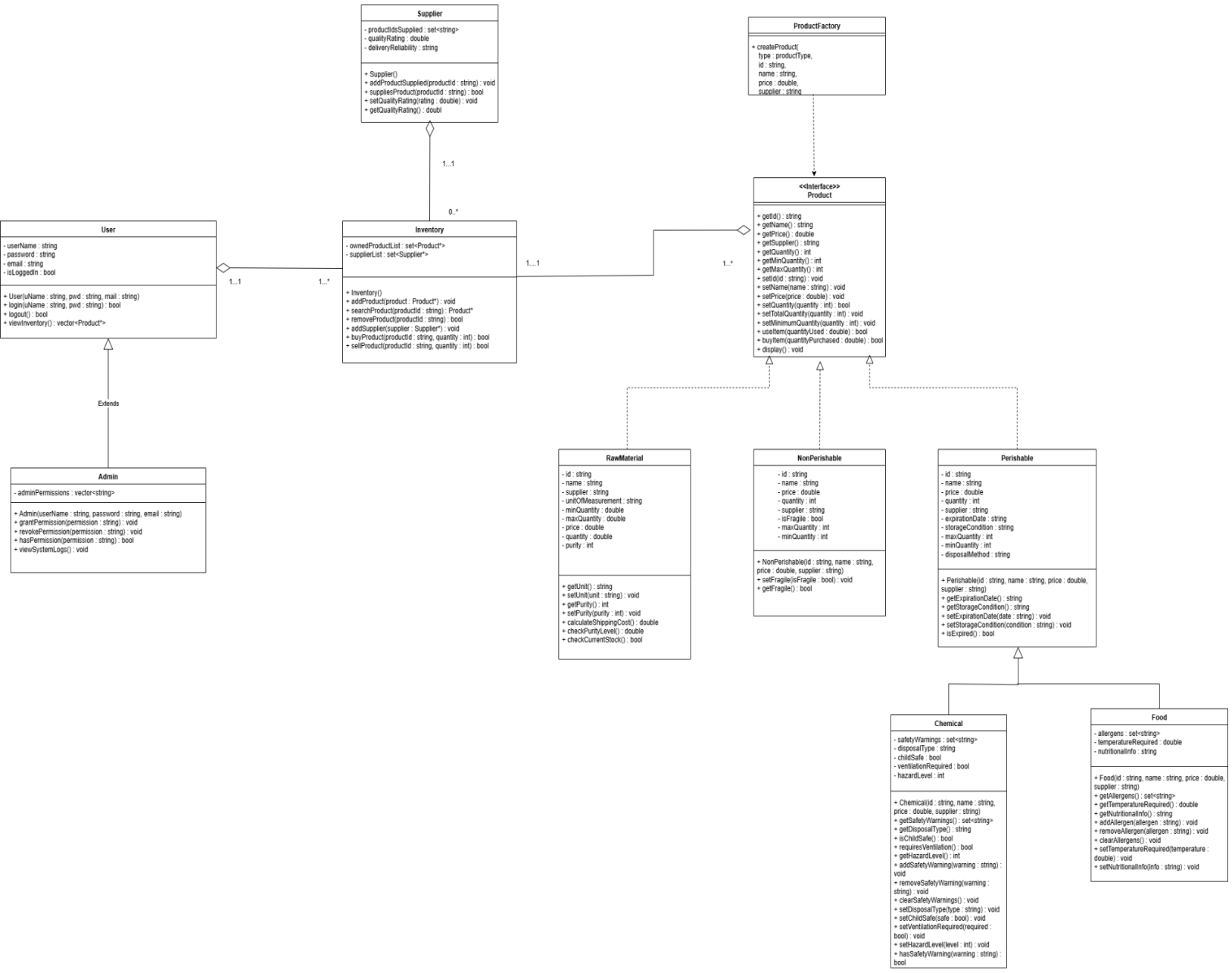

Figure 2.1

Sequence Diagrams

Sequence Diagram 1

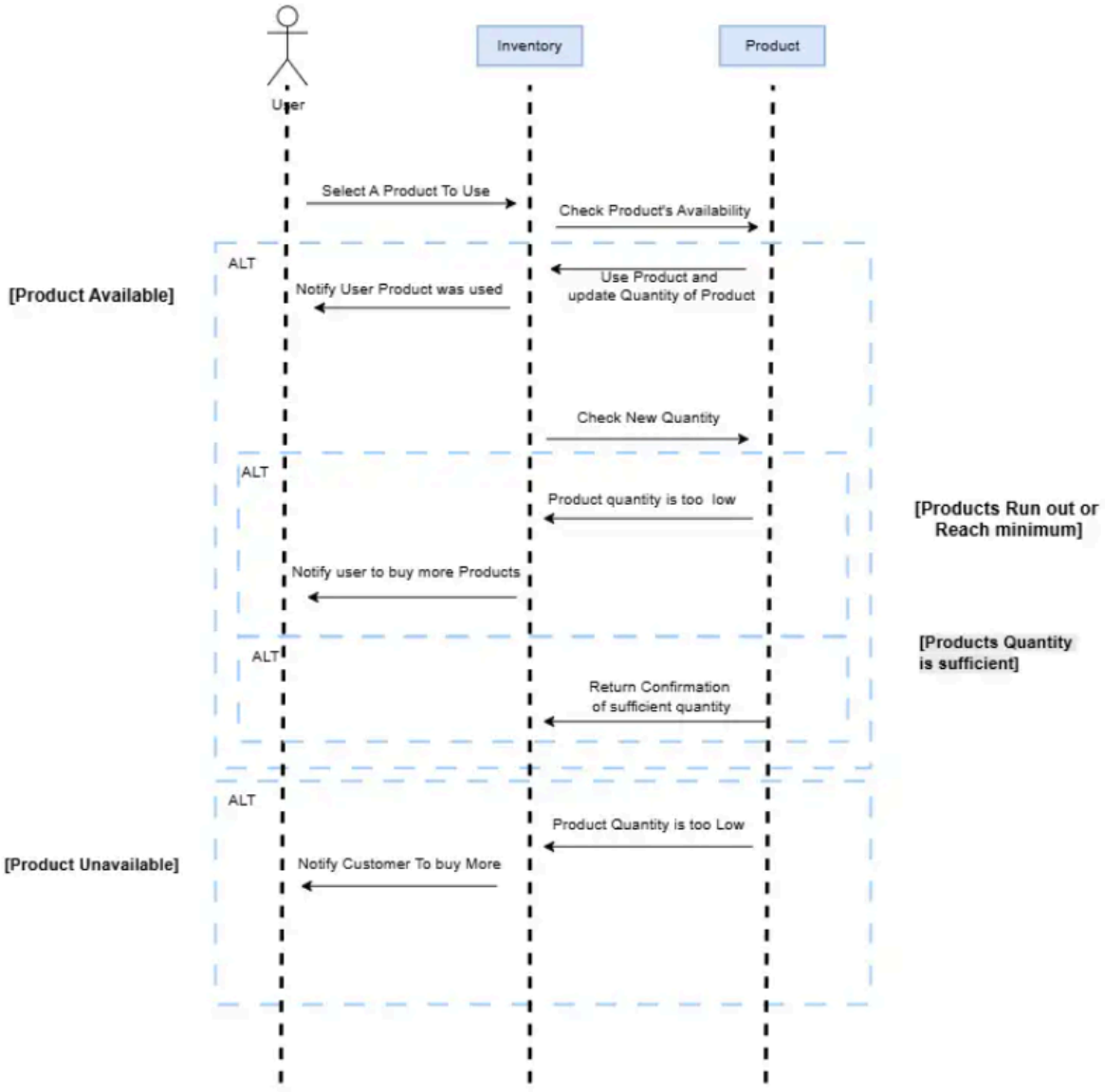

Figure 3.1

Sequence Diagram 2

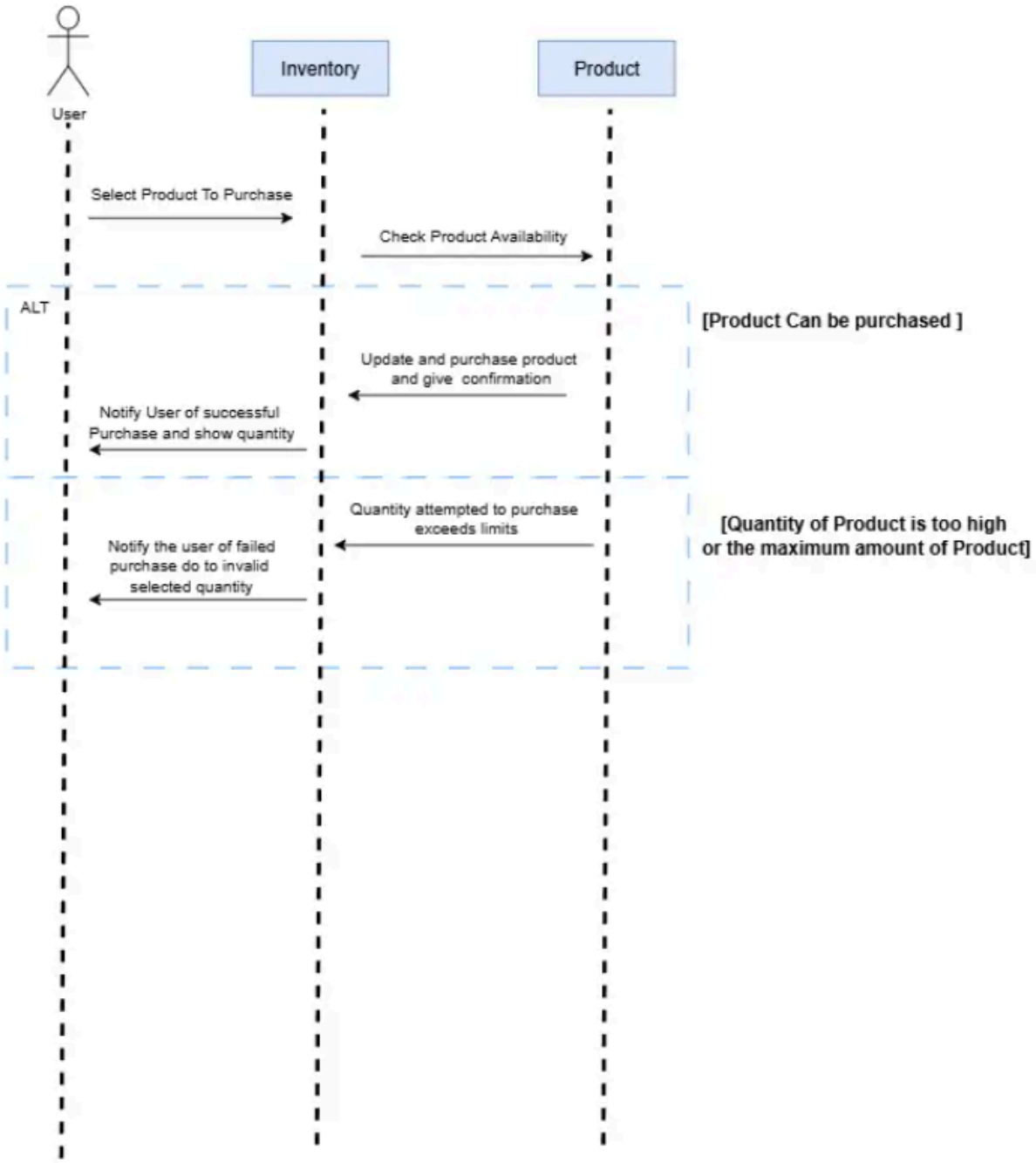

Figure 3.2

Design Patterns Implementation

Factory Pattern

Pattern: Factory Method
Pattern Location: ProductFactory class
Purpose: Centralize product creation and encapsulate instantiation logic

Implementation

```
class ProductFactory {
public:
   static Product* createProduct(
      productType type,
      std::string id,
      std::string name,
      double price,
      std::string supplier
   );
};
```

Why This Pattern

- Encapsulation: UI doesn't need to know about concrete product classes
- Maintainability: All creation logic in one place
- Extensibility: Easy to add new product types
- Type Safety: Enum-based type selection prevents invalid types

Benefits Realized

- Removed product creation logic from the UI layer
- Automatic ID prefixing (RM, NP, F, C, P) handled consistently
- Single point of modification for creation logic
- Improved testability through centralized creation

Example Usage

```
Product* apple = ProductFactory::createProduct(
   productType::FOOD,
   "001",
```

```
    "Apple",
    1.99,
    "Fresh Farms"
);
// Returns Food object with ID "F001"
```

Interface / Abstract Base Class Pattern

Pattern: Interface-based
Polymorphism Location: Product interface
Purpose: Define consistent behaviour across all product types

Implementation Hierarchy

```
Product (interface)
├── RawMaterial
├── NonPerishable
└── Perishable
    ├── Food
    └── Chemical
```

Why This Pattern

- Polymorphism: Inventory can manage all products through a single interface
- Open/Closed Principle: New types added without modifying existing code
- Liskov Substitution: Any Product implementation is interchangeable
- Contract Enforcement: All products must implement core methods

Key Benefits

- Type-safe polymorphic collections (std::set<Product*>)
- Runtime binding allows dynamic behaviour
- Clean separation of interface and implementation
- Supports dependency injection for testing

Facade Pattern

Pattern: Facade
Location: Inventory class
Purpose: Simplify complex subsystem interactions

Simplified Operations

```
// Complex multi-step operation simplified
inventory->addProductWithSupplier(product, supplier);

// Internally handles:
// - Adding product to inventory
// - Registering supplier
// - Creating a bidirectional relationship
// - Validating constraints
```

Why This Pattern

- Complexity Hiding: UI doesn't need to know relationship details
- Coordination: Single point for multi-object operations
- Business Logic Centralization: Rules enforced in one place
- Easier Testing: Test facade methods instead of individual steps

Key Facade Methods

- buyProduct(): Coordinates lookup, validation, supplier info, quantity update
- sellProduct(): Manages reduction, low stock check, notifications
- checkLowStock(): Analyzes all products, identifies issues, and retrieves supplier data
- generateInventoryReport(): Aggregates data from multiple sources

Composition / Aggregation

Pattern: Object Composition
Location: Inventory and Supplier classes
Purpose: Model real-world ownership relationships

Inventory Composition

```
class Inventory {
private:
   std::set<Product*> ownedProductList;    // Strong ownership
   std::set<Supplier*> supplierList;       // Strong ownership
};
```

Supplier Association

```
class Supplier {
private:
   std::set<std::string> productIdsSupplied;  // Weak reference
};
```

Why This Pattern

- Clear Ownership: Inventory is responsible for memory management
- Lifecycle Management: Objects destroyed with the owner
- Relationship Modelling: Reflects the business domain accurately
- Memory Safety: Prevents leaks through defined ownership

Iterator Pattern

Pattern: STL Iterators
Location: Throughout Inventory and UI
Purpose: Safe collection traversal

Implementation

```
std::set<Product*> products = inventory->getProductList();
for (Product* p : products) {
   // Process each product safely
}
```

Why This Pattern

- Standard Practice: Idiomatic C++
- Type Safety: Compile-time type checking
- Encapsulation: Internal structure hidden
- Range-based Loops: Modern C++ syntax support

Testing Report

Testing Framework Setup

We integrated GoogleTest v1.14.0 and Google Mock using CMake's FetchContent module, which automatically downloads and configures the testing framework during build.

CMake Configuration

```
include(FetchContent)
FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG v1.14.0
)
FetchContent_MakeAvailable(googletest)
enable_testing()
```

Test Library Structure:

We created a separate static library (inventory_core) containing only backend business logic, completely independent of Qt UI components. This separation enables testing core functionality without UI dependencies.

Test Organization:

- test_food.cpp: Food class unit tests
- test_inventory.cpp: Inventory class unit tests with mocking

Test Coverage Overview

We implemented comprehensive unit tests covering two critical classes that represent different aspects of our system.

Classes Under Test

Food Class

- 27 tests
- Concrete product implementation inheriting from Perishable
- Tests data encapsulation, allergen management, and quantity operations
- Validates inherited behaviour from the base class

Inventory Class

- 31 tests
- Central business logic coordinator and facade
- Tests complex interactions between products, suppliers, and operations

- Validates error handling and edge cases

Total Test Suite

- 58 total tests across 2 test suites
- 100% pass rate, all tests passing
- 7ms total execution time, extremely fast
- Test Fixtures: Used for code reuse (SetUp/TearDown)
- Mock Objects: MockProduct class for dependency isolation

Testing Strategy

Technique	Purpose	Example
Unit Testing	Test individual methods	Food::addAllergen()
Boundary Testing	Test edge values	Zero, negative, maximum quantities
Error Handling	Test invalid inputs	Null products, negative quantities
Integration Testing	Test complete workflows	Complete buy/sell scenarios
Mock Objects	Isolate dependencies	MockProduct for Inventory tests

Key Test Cases

Rather than documenting every test, we highlight representative examples that demonstrate our testing approach.

Test Category 1: Basic Functionality

Example: Food Constructor Initialization

```
TEST_F(FoodTest, ConstructorInitializesCorrectly) {
   EXPECT_EQ(testFood->getId(), "F001");
   EXPECT_EQ(testFood->getName(), "Apple");
```

```
    EXPECT_EQ(testFood->getPrice(), 1.99);
    EXPECT_EQ(testFood->getSupplier(), "Fresh Farms");
}
```

Purpose: Verify correct object initialization
Assertions: EXPECT_EQ for value equality
Result: PASSED

Test Category 2: Business Logic Validation

Example: Respecting Maximum Quantity

```
TEST_F(FoodTest, BuyItemRespectMaxQuantity) {
    testFood->setTotalQuantity(20);  // Max is 20
    testFood->setQuantity(18);       // Current is 18

    bool success = testFood->buyItem(5); // Try to add 5 (would be 23)

    EXPECT_FALSE(success); // Should fail
    EXPECT_EQ(testFood->getQuantity(), 18); // Unchanged
}
```

Purpose: Ensure business rules are enforced
Assertions: EXPECT_FALSE for failure, EXPECT_EQ for state preservation
Result: PASSED

Test Category 3: Complex Interactions

Example: Product-Supplier Relationship

```
TEST_F(InventoryTest, AddProductWithSupplier) {
    Food* apple = new Food("F001", "Apple", 1.99, "Fresh Farms");
    Supplier* supplier = new Supplier();
    supplier->setQualityRating(4.5);

    inventory->addProductWithSupplier(apple, supplier);

    // Verify product added
    Product* found = inventory->searchProduct("F001");
    EXPECT_NE(found, nullptr);
```

```
    // Verify supplier added
    EXPECT_EQ(inventory->getSupplierList().size(), 1);

    // Verify bidirectional relationship
    Supplier* foundSupplier = inventory->getSupplierForProduct("F001");
    EXPECT_TRUE(foundSupplier->suppliesProduct("F001"));
}
```

Purpose: Test facade pattern and relationship management
Assertions: Multiple EXPECT statements verifying different aspects
Result: PASSED

Google Mock Implementation

We created a MockProduct class demonstrating proper use of Google Mock for dependency isolation.

MockProduct Class:

```
class MockProduct: public Product {
public:
    MOCK_METHOD(std::string, getId, (), (const, override));
    MOCK_METHOD(std::string, getName, (), (const, override));
    MOCK_METHOD(int, getQuantity, (), (const, override));
    MOCK_METHOD(bool, buyItem, (double), (override));
    MOCK_METHOD(bool, useItem, (double), (override));
    // ... other methods
};
```

Purpose of Mocking

- Isolation: Test Inventory without depending on real Product implementations
- Controlled Behaviour: Specify exactly what mocked methods return
- Verification: Confirm expected methods are called with correct parameters

Test Results

Complete Test Execution Output

[==========] Running 58 tests from 2 test suites.

[----------] 31 tests from InventoryTest (6 ms)

[----------] 27 tests from FoodTest (1 ms)

[==========] 58 tests from 2 test suites ran. (7 ms total)

[PASSED] 58 tests.

Changes from Deliverable 2

Major Features Added

ProductFactory Pattern

- Created ProductFactory class for centralized product creation
- Removed creation logic from the UI layer
- Automatic ID prefixing (RM, NP, F, C, P) for all product types
- Enum-based type selection for type safety

Data Persistence (FileManager)

- Implemented JSON-based file I/O
- Save/Load inventory state, including all products and suppliers
- Error handling for file operations
- "Save Inventory" and "Load Inventory" buttons in UI

Comprehensive Testing

- 58 unit tests using GoogleTest
- Google Mock implementation (MockProduct class)
- Test fixtures for code reuse
- 100% test pass rate

Architectural Improvements

Separation of Concerns

- ProductFactory removed creation logic from UI
- FileManager centralized I/O operations

- Inventory remains pure business logic

Error Handling

- Graceful null pointer handling
- Invalid input validation
- File I/O error recovery
- User-friendly error messages

Code Quality

- Comprehensive documentation
- Consistent naming conventions
- Reduced code duplication
- Improved maintainability

What Was Not Changed

Core Architecture: Three-layer architecture remains unchanged
Product Hierarchy: Inheritance structure validated through testing
Facade Pattern: Inventory class still coordinates operations
UI Framework: Qt Widgets continues to serve well

Challenges and Reflection

Technical Challenges

Challenge 1: GoogleTest Integration

Problem: Integrating GoogleTest with a Qt project while separating core logic from UI dependencies.

Solution:
- Created a separate inventory_core static library containing only backend classes
- Used CMake FetchContent for automatic GoogleTest download
- Proper linking of the test executable to both the core library and GoogleTest

Learning: Building system configuration is crucial for testing the architecture. Understanding CMake's library system enabled clean separation of concerns.

Challenge 2: JSON Data Persistence

Problem: Serializing complex product hierarchy with type-specific attributes.

Solution:
- Stored product type identifier with each product
- Type-specific attributes saved in structured JSON
- ProductFactory used for deserialization (creating the correct type)
- Error handling for corrupted or missing files

Learning: Polymorphic serialization requires careful design. Storing type information enables correct object reconstruction during loading.

Design Trade-offs

Trade-off 1: In-Memory vs Database

Decision: Used in-memory STL containers with file persistence

Rationale:

- Simpler implementation for the project scope
- Faster operations (no database queries)
- Easier testing (no database setup required)
- Limited to single-user access
- No concurrent access support

Reflection: For this project, simplicity outweighed scalability concerns. In production, a database would be necessary for multi-user scenarios.

Trade-off 2: JSON vs Binary Format

Decision: Used JSON for data persistence

Rationale:

- Human-readable for debugging
- Easy to edit manually if needed
- Cross-platform compatibility
- Larger file size than binary

- Slightly slower parsing

Reflection: Readability and debugability were more valuable than optimization for our use case.

Trade-off 3: Testing Real Objects vs Mocks

Decision: Used real Food objects in Inventory tests instead of extensive mocking

Rationale:

- Tests integration between classes
- Validates real behaviour, not mocked behaviour
- Simpler test setup
- Less unit isolation
- Tests multiple units simultaneously

Reflection: While MockProduct demonstrates mocking knowledge, using real objects provided more confidence in the actual system behaviour. A balance between unit and integration testing is important.

Lessons Learned

Lesson 1: Test-Driven Development Benefits

Insight: Writing tests revealed bugs we wouldn't have found otherwise.

Example: Tests revealed that buyProduct() didn't validate for negative quantities - a bug that could have reached production.

Takeaway: Invest time in testing early. The bugs you catch in development are far cheaper than those caught in production.

Lesson 2: Interface Design Enables Testing

Insight: The Product interface made testing possible by allowing polymorphic collections and mock implementations.

Example: MockProduct could be used anywhere a real Product is expected.

Takeaway: Designing for testability from the start saves refactoring later. Interfaces and dependency injection are key to testable code.

Lesson 3: Incremental Development Works

Insight: Building 60% in Deliverable 2, then completing in Deliverable 3 was the right approach.

Example: Having a core backend working in D2 allowed us to focus purely on persistence and testing in D3.

Takeaway: Breaking projects into phases with concrete milestones prevents overwhelming complexity and enables steady progress.

Lesson 4: Documentation Matters

Insight: Clear documentation made collaboration easier and helped us remember design decisions.

Example: Design pattern explanations from D2 guided our D3 implementations.

Takeaway: Time spent documenting design rationale pays dividends in implementation and maintenance.

Conclusion

This final deliverable represents a complete, tested, and documented Inventory Management System that demonstrates mastery of object-oriented design and analysis principles. The system successfully fulfills all requirements from our initial proposal while exceeding expectations in code quality, testing coverage, and documentation.

System Completeness

- All Core Features Implemented: Product management, supplier tracking, user authentication, and inventory operations
- Design Patterns: Factory, Interface, Facade, Composition, and Iterator patterns correctly implemented
- Data Persistence: JSON-based file I/O for saving/loading state
- Comprehensive Testing: 58 tests with 100% pass rate using GoogleTest and Google Mock

- Professional Documentation: Complete UML diagrams, pattern explanations, testing reports
- Qt Integration: Functional GUI with proper separation from business logic

Final Reflections

What Went Well

- Clean three-layer architecture maintained throughout
- Product hierarchy design was flexible and extensible
- Factory pattern cleaned up object creation significantly
- Testing caught bugs early and gave confidence in code quality
- Team collaboration was effective with clear task division

What Could Be Improved

- More comprehensive UI testing (Qt Test framework)
- Code coverage metrics (aim for 90%+)
- Performance testing with large datasets
- More extensive documentation of methods
- Additional design patterns (Observer for notifications)

Link To Loom Video

https://youtu.be/veLnmdeNsGk