1. Cover Page

Project Name: Inventory Management System (IMS)

Team Members: Rasul Can Çulğatay, Abdurrahman Arda Demir, Samet Bilgin

Course Name: Design Patterns (Tasarım Desenleri)

Date: November 2025

(Note: This report presents the detailed technical documentation of the Inventory Management System application, which was developed as a term project within the scope of the "Design Patterns" course.)

---

2. Table of Contents

---

3. Project Summary

The **Inventory Management System (IMS)** is software developed to enable businesses to effectively track their inventory and automate stock control. The project aims to provide solutions to common problems encountered in stock tracking (e.g., overstocking, understocking, manual tracking errors). IMS allows products and categories to be managed in a hierarchical structure and provides real-time notifications for critical stock levels, ensuring users are instantly informed. This is intended to prevent financial losses resulting from inventory errors.

Project Goal and Objectives:

The main goal of IMS is to increase accuracy and efficiency by automating inventory management. The system provides functions such as adding/updating products, category-based organization, continuous monitoring of stock levels, and providing alerts for stock that falls below a defined threshold. Within this scope, the project objectives are:

- To **automate** inventory processes, reducing manual intervention and errors.
- To **pre-emptively detect** under/over-stock situations with real-time stock updates and alerts.
- To **structure** and clarify the inventory by organizing products under a category hierarchy.
- To create a **modular structure** with a flexible and scalable architecture, allowing new features to be easily added.

Problem Solved:

Poor inventory management is a critical issue for many businesses, often leading to situations like holding excessive stock or stock shortages, causing financial losses. The IMS project addresses this by automating stock updates and alerts, tracking product categories and quantities in real-time, and organizing the warehouse/product tree in a **hierarchical structure**. The system gained **modularity** and **flexibility** through the clever use of Design Patterns (such as **Composite, Observer, Strategy, and Singleton**). This makes maintenance and updates easier, providing an infrastructure that is adaptable to new requirements.

General Functions:

The main functions offered by IMS include:

- **Category and Product Management** (add, edit, delete).
- **Inventory Organization** via a hierarchical category tree.
- **Instant** updating and tracking of product stock.
- **Real-time notifications** and an alarm mechanism for critical stock levels.
- **Search and Filtering** feature to quickly find the desired item in the inventory.
- **Export** of inventory data in formats like CSV.
- **Dynamic selection** of different stock replenishment strategies (e.g., automatic ordering of low stock).

In summary, this project is designed in accordance with modern software architectural principles and offers businesses an accurate, reliable, and effective inventory management solution by improving stock control processes.

---

4. Design Patterns Used

Object-Oriented Design Patterns were extensively used in the design of the IMS project. This enhanced the software's modularity and maintainability, creating an architecture open to extension. This section details the design patterns applied in the project.

4.1. Composite Pattern

Definition and Purpose:

The **Composite** design pattern allows us to compose objects into tree structures to represent part-whole hierarchies. It enables clients to treat individual objects (**Leaf**) and compositions of objects (**Composite**) uniformly. Its purpose is to reflect the part-whole relationship in the software. [Image of Composite Pattern UML Diagram] The client code can perform the same operations on a single object or a group of objects without distinction.

Project Implementation:

In IMS, the Composite pattern is used to represent the **category and product structure**. An abstract class/interface, $ProductComponent$, is defined as the common supertype for both categories and products. The **Category** class extends this to form the Composite structure (holding a list of child $ProductComponent$s), while the **Product** class represents the Leaf (not containing any children). This allows a category tree structure where operations (e.g., deleting a category) can be recursively propagated to all its sub-elements (sub-categories and products).

Example Classes:

- **ProductComponent** – Abstract Component: Defines common methods like $getName()$, $addChild()$, and $removeChild()$.
- **Category** – Composite Class: Extends $ProductComponent$. Manages the list of child $ProductComponent$s (products or sub-categories). Implements recursive operations.
- **Product** – Leaf Class: Concrete subclass of $ProductComponent$. Contains product-specific data (name, stock, price, etc.). It does not contain child elements.

Why Composite Pattern?

- **Hierarchical Structure and Scalability:** Naturally models the category-product hierarchy. Easily integrates new categories or products without altering existing code.
- **Uniformity:** Categories and products are handled as the same type ($ProductComponent$), simplifying operations like listing all items under a high-level category.
- **Maintenance Ease:** Facilitates recursive operations (e.g., calculating total stock for a category), reducing code repetition.
- **Real-World Modeling:** Directly reflects the real-world inventory structure to the software.

4.2. Observer Pattern

Definition and Purpose:

The **Observer** design pattern defines a one-to-many dependency between objects so that when one object (**Subject**) changes state, all its dependents (**Observers**) are notified automatically. This creates a loosely coupled notification mechanism, ideal for event-driven systems and real-time updates. The main components are the Subject, the Observer interface (with an $update()$ method), and concrete Observers.

Project Implementation:

The Observer pattern is implemented in IMS for the **stock update notification system**. The **InventoryManager** class acts as the **Subject**, where stock changes occur. It maintains a list of registered $Observer$s. When a product's stock quantity is updated (e.g., it falls below the critical threshold), $InventoryManager.notifyObservers()$ is called.

Example Classes and Interactions:

- **Observer** – Observer Interface: Defines the $update(Product)$ method to receive notifications.
- **InventoryManager** – Subject Class (The Observable): Holds the list of $Observer$s, provides $addObserver()$ and $removeObserver()$ methods. Triggers $notifyObservers()$ upon stock change.
- **InventoryUI** – Concrete Observer: Updates the user interface (e.g., highlights a product row in red) when $update()$ is called, especially for critical stock.
- **SupplierNotifier** (or AlarmService) – Concrete Observer: Can trigger an automatic action like sending an order email to a supplier when critical stock is reached.

Why Observer Pattern?

- **Decoupling:** The stock tracking component ($InventoryManager$) is loosely coupled from the notification components (UI, alarm service). Changes to one don't significantly affect the others.

- **Real-time Feedback:** Instantly alerts relevant parties (users, services) about critical stock changes, allowing for quick action.
- **Multiple Subscriptions:** Multiple Observers (UI, Supplier Service, etc.) can monitor the same event simultaneously.
- **Maintenance and Extensibility:** New notification methods (e.g., mobile push notifications) can be added simply by creating a new $Observer$ class and registering it, adhering to the Open/Closed Principle.

4.3. Strategy Pattern

Definition and Purpose:

The **Strategy** design pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the client to choose an algorithm at runtime. This avoids bulky conditional statements ($if/else$) and promotes cleaner, more maintainable code by defining each variant behavior in a separate class. The key components are the $Strategy$ interface, Concrete $Strategy$ classes, and a $Context$ object that holds a reference to the selected strategy.

Project Implementation:

The Strategy pattern is used in IMS to implement different **stock replenishment strategies**. The **InventoryManager** acts as the **Context**, holding a reference to the currently selected stock replenishment strategy, which implements the $StockReplenisher$ interface.

Example Classes:

- **StockReplenisher** – Strategy Interface: Declares an operation, e.g., $replenish(Product\ p)$.
- **ThresholdReplenisher** – Concrete Strategy: Implements threshold-based replenishment (e.g., if stock $\le$ minimum level, order a fixed quantity).
- **DemandBasedReplenisher** – Concrete Strategy: Implements a different logic, such as planning replenishment based on past sales data or demand forecasts.
- **InventoryManager** – Context Class: Stores a reference to the active strategy ($currentStrategy: StockReplenisher$) and calls $currentStrategy.replenish(product)$ when a critical stock event occurs. It also provides a $setStrategy()$ method for dynamic selection.

Why Strategy Pattern?

- **Algorithm Family and Interchangeability:** Allows new replenishment methods to be added without modifying the core $InventoryManager$ class (OCP).
- **Reduced Conditional Logic:** Eliminates large $if/else$ blocks within the $InventoryManager$ for deciding how to replenish stock.
- **Dynamic Structure:** Enables the system to change its stock policy at runtime (e.g., switching from demand-based to threshold-based during peak season) via a UI selection.
- **Testability:** Each strategy can be unit-tested in isolation, increasing software reliability.

4.4. Singleton Pattern

Definition and Purpose:

The **Singleton** design pattern ensures that a class has only one instance throughout the application and provides a global point of access to it. It is typically used for central managers, loggers, or configuration handlers where system-wide uniqueness and consistent state are required. Implementation typically involves a $private$ constructor, a $static$ instance variable, and a $static$ $getInstance()$ method.

Project Implementation:

The **InventoryManager** class is implemented as a **Singleton**. It is the central authority for all product and category data, stock operations, and notifications. Since there is only one inventory for the system, it must be managed by a single central entity.

Implementation Snippet (Java-like):

```
public class InventoryManager {
    private static InventoryManager instance;
    private InventoryManager() { /* private constructor */ }

    public static synchronized InventoryManager getInstance() {
        if (instance == null) {
            instance = new InventoryManager();
        }
        return instance;
    }
    // ... other methods
}
```

Why Singleton Pattern?

- **Consistency:** Guarantees that all inventory operations (read/write) are handled by the same single instance, preventing data inconsistencies that would arise from multiple inventory copies.
- **Memory and Resource Efficiency:** Prevents the unnecessary creation of multiple large objects. Resources like database connections, if managed here, are also centralized.
- **Centralized Control:** Simplifies maintenance and error tracking by consolidating inventory-related logic in one well-known access point.
- **Global Access:** Provides controlled global access to the central management object from any part of the application (e.g., $InventoryManager.getInstance().addProduct(...)$).

Note: The $InventoryManager$ also serves as the Subject in the Observer pattern and the Context in the Strategy pattern.

---

5. System Architecture

The IMS application adopts a **multi-layered architecture**, separating the user interface, business logic, and data access concerns. This promotes modularity and maintainability.

5.1. Layered Architecture (UI / Business Logic / Data Layer)

The IMS system consists of three main layers:

1. **Presentation Layer (UI Layer):**
    - Contains the user-facing components (forms, buttons, tables, category tree).
    - Interacts with the Business Logic Layer but does not contain business rules.
    - Classes: $InventoryUI$ and associated GUI components.
2. **Business Logic Layer:**
    - The core of the application, implementing business rules and behavior.
    - Design patterns are predominantly applied here: $InventoryManager$ (Singleton, Subject, Context), $Product$, $Category$.
    - Receives requests from the UI, performs operations, and interacts with the Data Layer.
3. **Data Layer:**
    - Responsible for persistent data management, using the **SQLite** database.
    - Handles database connection, SQL queries, and table operations.

- Classes: $DatabaseHelper$ (and potential DAO classes).
- Provides a simple save/load interface to the Business Logic Layer, abstracting the physical data storage details.

This layered structure ensures that changes in one layer (e.g., changing the database from SQLite to PostgreSQL) have minimal impact on other layers, making the application scalable and maintainable.

5.2. Class Diagram and Inter-Class Relationships

The UML Class Diagram visualizes the structural design and the distribution of design patterns across the classes.

Figure 5.1: IMS Application Class Diagram. The diagram shows the core classes and their relationships. $ProductComponent$ is the common interface for $Category$ (Composite) and $Product$ (Leaf). $InventoryManager$ operates as a Singleton, integrates the Observer pattern via its list of observers, and the Strategy pattern via its strategy reference.

Relationship Summary:

- **Inheritance:** $Category$ and $Product$ inherit from $ProductComponent$ (Composite). $Concrete\ Observers$ and $Concrete\ Strategies$ implement their respective interfaces.
- **Composition (◆):** $Category$ has a composition relationship with $ProductComponent$ children ($children\ *$). Deleting a category deletes all its contents.
- **Aggregation (◊):** $InventoryManager$ has an aggregation relationship with $Category$ objects ($categories\ *$) and a one-to-many relationship with the $Observer$ interface ($observers\ *$).
- **Dependency:** $InventoryManager$ has a dependency on the $StockReplenisher$ interface via its $strategy$ field (Strategy pattern context).

5.3. UML Representation of Observer and Strategy Relationships

The Class Diagram clearly shows the key relationships for the behavioral patterns:

- **Observer Pattern:** Highlighted by the $InventoryManager$'s one-to-many association with the $Observer$ interface ($observers\ 0..*$ list). The $InventoryUI$ and $SupplierNotifier$ are concrete implementations of $Observer$.
- **Strategy Pattern:** Shown by the dependency (dashed line) from the $InventoryManager$ (Context) to the $StockReplenisher$ interface (Strategy), and the implementation (solid line with hollow triangle) of $StockReplenisher$ by the $ThresholdReplenisher$ and $DemandBasedReplenisher$ classes.

These UML elements confirm that the system design adheres to the principles of the chosen patterns, ensuring modularity and flexibility in its dynamic behavior.

---

6. Functional Features

This section details the functional features offered to the user by the IMS application, covering user interaction and core scenarios.

6.1. User Interface

The UI is designed to present inventory data simultaneously and systematically:

- **Category Tree:** A hierarchical tree structure (left side) that visually represents the Composite category structure. Users can expand/collapse nodes to view sub-categories and products.
- **Product Table:** A table (right side) listing the products belonging to the selected category (and its sub-categories). Columns include Product Name, Stock Quantity, Unit Price, etc.
- **Toolbar and Buttons:** Contains buttons for frequent actions: "Add New Category," "Add New Product," "Edit," "Delete," "Search," and "Export." These buttons trigger core CRUD operations handled by the $InventoryManager$.

The UI follows a master-detail design: category selection (master) on the left dictates the product list (detail) on the right.

6.2. Category and Product Addition / Editing / Deletion (CRUD) Operations

All CRUD operations involve the $InventoryManager$ and $DatabaseHelper$ to ensure data persistence and model consistency. The UI is updated via the **Observer** mechanism.

- **Adding a Category/Product:** The user fills a form, which triggers $InventoryManager.addCategory()$ or $InventoryManager.addProduct()$. The data is saved to the database, and the UI ($InventoryUI$ Observer) receives a notification to update the tree and table views.
- **Editing a Category/Product:** The user modifies an item's details (e.g., stock quantity, price). $InventoryManager.updateProduct()$ or $updateCategory()$ is called. Changes are persisted, and the Observer mechanism updates the UI instantly. Stock updates are particularly important as they trigger alarm checks (see 6.3).
- **Deleting a Category/Product:** $InventoryManager.deleteProduct()$ or $deleteCategory()$ is called after user confirmation. For categories, the **Composite pattern** ensures that deletion is **recursive**, removing all nested sub-categories and products from the memory model and the database. The Observer then removes the items from the UI.

6.3. Stock Alarm Scenarios and Operation

The critical stock alarm mechanism uses the **Observer** and **Strategy** patterns to provide proactive alerts and automatic actions.

- **Threshold Definition:** Each product has a min_stock_level defined (critical stock threshold).
- **Stock Reduction:** When stock is updated and falls to or below the threshold, the $InventoryManager$ checks the condition.
- **Alarm Trigger (Observer):** If the stock is critical, the $InventoryManager$ sends an $update(product)$ notification to its $Observer$s.
  - $InventoryUI$ (Observer) receives the notification and provides a visual warning (e.g., highlights the product row in red, displays a pop-up alert).
  - $SupplierNotifier$ (Observer) receives the notification and triggers an automated action, such as sending an urgent order email to the supplier.
- **Automatic Action (Strategy):** Before/After notifying observers, the $InventoryManager$ invokes the currently selected replenishment strategy: $currentStrategy.replenish(product)$. This might trigger an immediate order (if $ThresholdReplenisher$) or a planning process (if $DemandBasedReplenisher$).
- **Alarm Clearance:** If stock is subsequently increased above the critical level, the UI is updated to remove the alarm status.

6.4. Search and Filtering Feature

IMS provides quick access to inventory items through a search and filtering function:

- **Product Name Search:** Users enter text into the search bar, and the system filters the product table to show only items whose names match the text (case-insensitive, partial match).
- **Category Filtering:** The main category tree acts as a primary filter, restricting the product table to the selected category and its sub-categories. Search results can also be restricted to the currently selected category.
- **Result Update:** Search results are dynamically updated. If a product's details are edited (e.g., its name) while a filter is active, the $InventoryUI$ (Observer) updates the list, ensuring the product remains visible only if it still matches the filter criteria.

6.5. Data Export Feature

The application supports exporting inventory data for reporting and sharing:

- **CSV Export:** The primary format. Users click "Export," select a file location, and the product data (Product Name, Stock, Price, full Category Path) is saved to a CSV file.
- **Optional Formats:** The architecture is open to adding other formats like JSON or XML. If implemented with the

Strategy pattern, different $ExportStrategy$ classes could be used for each format.
- **Scope:** Users can typically choose to export the entire inventory or only the product list currently displayed (e.g., filtered by search or category).

6.6. Dynamic Strategy Selection Feature

This feature is the practical application of the **Strategy** pattern (Section 4.3). It allows administrators to change the system's stock replenishment behavior at runtime.

- **Selection Interface:** An "Settings" or "Admin Panel" screen provides options (e.g., a radio button) to choose between "Threshold-Based Automatic Replenishment" and "Demand Forecasting-Based Replenishment."
- **Runtime Change:** When a new strategy is selected and saved, $InventoryManager.setStrategy(new\ SelectedStrategy())$ is called. All subsequent critical stock checks will execute the logic of the newly set strategy instance, without requiring an application restart or code change.
- **Flexibility:** This allows the business to adapt to seasonal changes or policy shifts by simply toggling an option (e.g., switching to a more aggressive threshold-based strategy during peak holiday seasons).

---

7. Database and Data Management

The IMS application uses **SQLite** for data persistence.

7.1. Database (SQLite) and Table Structure

SQLite is chosen for its simplicity and file-based operation, ideal for a single-user or small-scale application.

Table Structures:

- **Categories Table:**
  - $category\_id$ (INTEGER PRIMARY KEY)
  - $name$ (TEXT)
  - $parent\_id$ (INTEGER, FOREIGN KEY to $category\_id$ - NULL for root categories)
  - *Used to model the Composite hierarchy.*
- **Products Table:**
  - $product\_id$ (INTEGER PRIMARY KEY)
  - $name$ (TEXT)
  - $category\_id$ (INTEGER, FOREIGN KEY to $Categories.category\_id$)
  - $quantity$ (INTEGER)
  - $price$ (REAL)
  - min_stock_level (INTEGER - Critical Stock Threshold)
  - *FOREIGN KEY constraints with ON DELETE CASCADE ensure data integrity when categories are deleted.*

7.2. DatabaseHelper and Versioning (Migration)

The **DatabaseHelper** class encapsulates all data access logic, abstracting the SQL details from the Business Logic Layer ($InventoryManager$).

DatabaseHelper Responsibilities:

- **Connection Management:** Opening and closing the SQLite connection.
- **Table Creation ($onCreate$):** Executes the initial $CREATE\ TABLE$ SQL commands for $Categories$ and $Products$.
- **CRUD Methods:** Provides high-level methods for data interaction (e.g., $insertProduct()$, $getAllCategories()$).

Versioning and Migration:

The $DatabaseHelper$ includes a version number ($dbVersion$). When the application is updated, the version number is incremented. The **$onUpgrade()$** method contains the necessary $ALTER\ TABLE$ or $CREATE\ TABLE$ commands to update the database schema without losing existing user data (**migration**). This ensures the application can evolve and add new data fields seamlessly over time.

```
// Example of migration logic in onUpgrade()
if (oldVersion < 2) {
    db.execSQL("ALTER TABLE Products ADD COLUMN supplier_name TEXT");
}
```

---

8. UML Diagrams

8.1. Class Diagram

The Class Diagram, first presented in Section 5.2, is repeated here for completeness. It illustrates the static structure and the roles of classes within the design patterns.

Figure 8.1: Inventory Management System – Class Diagram.

8.2. Sequence Diagram (Stock Update Scenario)

This sequence diagram illustrates the dynamic flow for the "Stock update triggers UI refresh and alarm (Observer activation)" scenario.

. If true, InventoryManager sends notify(Product) messages to Observer1 and Observer2. Observer1 then executes: updateUI/ShowAlarm(). Observer2 executes: sendOrderEmail().]

Figure 8.2: Stock Update and Alarm Scenario – Sequence Diagram. The flow demonstrates the Observer pattern's publish-subscribe mechanism triggered by a stock update, ensuring real-time feedback and automatic action.

8.3. Optional: Other Diagrams

Additional UML diagrams could be drawn to further clarify the project's structure and behavior:

- **Component Diagram:** To visualize the layered architecture (UI, Business Logic, Database components) and their dependencies.
- **State Diagram:** To model the life cycle states of a Product (e.g., Normal, Critical Level) and the transitions between them based on stock changes.
- **Activity Diagram:** To detail the sequential and conditional steps within a complex process like "Product Deletion" or "Stock Replenishment."

For the scope of this report, the Class and Sequence diagrams provide the essential structural and behavioral insights.

---

9. Conclusion and Evaluation

9.1. Project Strengths

The IMS application exhibits several key strengths:

- **Modular and Maintainable Design:** Achieved through the strategic use of **Composite, Observer, Strategy, and Singleton** patterns, resulting in low coupling and high cohesion.
- **Real-Time Update and Notification:** The Observer pattern enables instant stock alerts, making the system a proactive tool for preventing stockouts.
- **Scalability:** The hierarchical, object-oriented design (Composite) and centralized control (Singleton) allow the system to handle a growing number of inventory items and observers gracefully.
- **Flexible Business Rules:** The Strategy pattern allows the stock replenishment policy to be changed dynamically at runtime, adapting to varying business needs (e.g., seasonal changes).
- **Data Integrity and Persistence:** SQLite, along with the $DatabaseHelper$ and migration mechanism, ensures that data is stored reliably and can be updated across software versions without loss.
- **User-Friendly Interface:** The clean master-detail (Category Tree - Product Table) design and responsive search functionality promote good user experience.

Overall, IMS successfully meets its functional goals with a robust and well-engineered architecture.

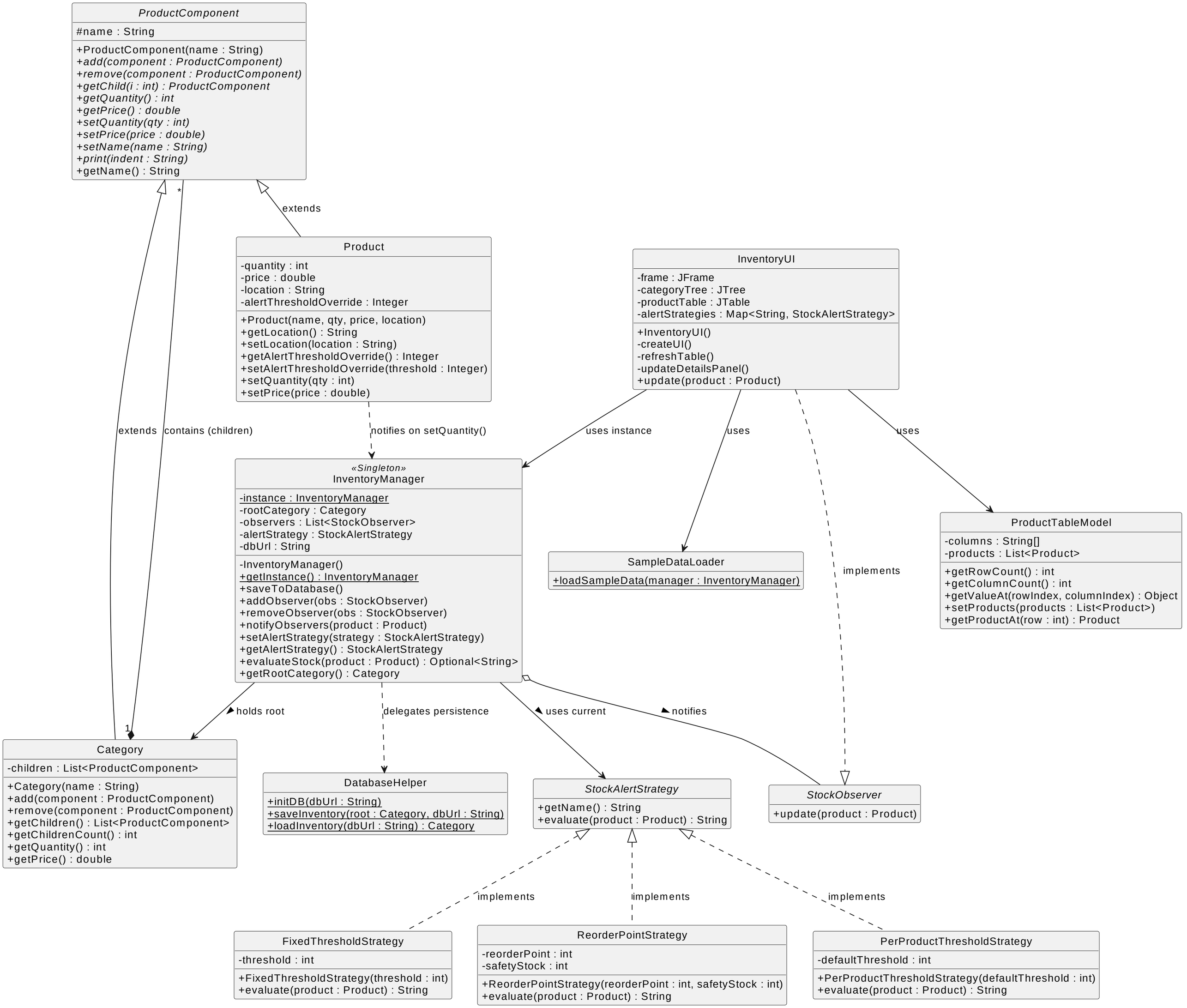
9.2. Lessons Learned

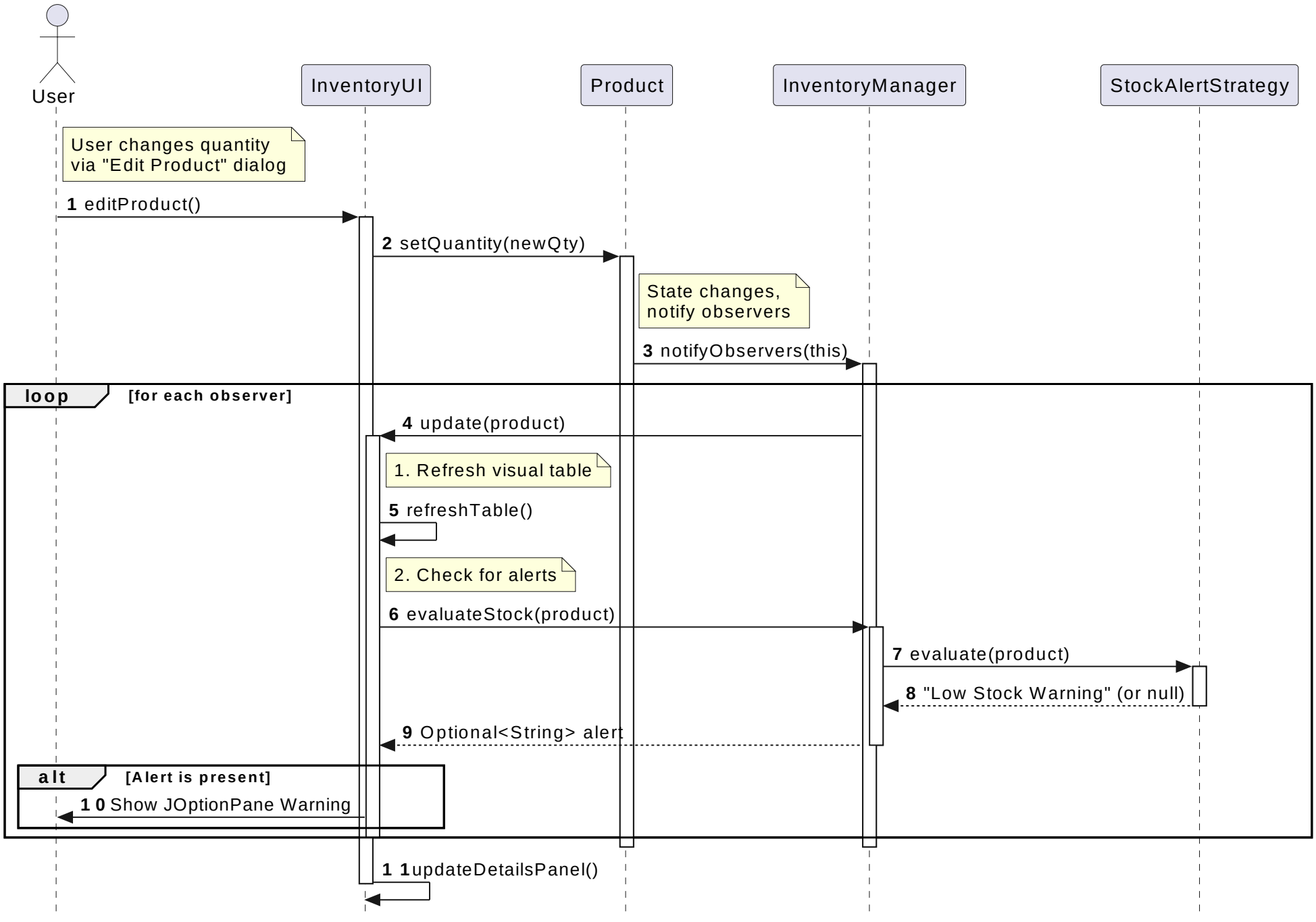
The project provided valuable practical experience:

- **Practical Application of Design Patterns:** Confirmed the importance of using the right pattern in the right context (e.g., Composite for hierarchy, Observer for decoupling events).
- **Coding for Flexibility (OCP):** Demonstrated how patterns like Strategy and Observer enable the code to be open for extension but closed for modification, facilitating adaptation to changing requirements.
- **Teamwork and Version Control:** Effective use of Git for collaborative development and seamless code integration, aided by the modular design.
- **Systematic Testing:** Gained practice in unit testing critical components like the $InventoryManager$ and the Observer notification flow.
- **Database Management:** Learned the practical aspects of SQL database design, foreign keys, and implementing data migration to handle schema changes.
- **User Experience (UX) Consideration:** Recognized that software design requires not only functional code but also an intuitive and efficient user interface flow.

The Inventory Management System project successfully translated theoretical design patterns knowledge into a concrete, well-structured, and functional application, providing the team with significant software engineering experience.

Inventory Management System - Class Diagram




Edit Product & Alert Trigger Sequence
User
InventoryUI
Product
InventoryManager
StockAlertStrategy
User changes quantity via "Edit Product" dialog
1 editProduct()
2 setQuantity(newQty)
State changes, notify observers
3 notifyObservers(this)
loop
[for each observer]
4 update(product)
1. Refresh visual table
5 refreshTable()
2. Check for alerts
6 evaluateStock(product)
7 evaluate(product)
8 "Low Stock Warning" (or null)
9 Optional<String> alert
alt
[Alert is present]
10 Show JOptionPane Warning
11 updateDetailsPanel()