***SOFTWARE PROJECT FINAL REPORT***

Inventory Systems

Ethan Pedrick (Backend), Nhat Nguyen (Database),

Devin White (EXE), Jake Staas (UI)

4/17/2025

**Table of Contents**

1. [Introduction](#Bookmark1)
   1. Purpose and Scope
   2. Product Overview
   3. Structure of the Document
   4. Terms, Acronyms, and Abbreviations
2. [Project Management Plan](#Bookmark2)
   1. Project Organization
   2. Lifecycle Model Used
   3. Risk Analysis
   4. Hardware and Software Resource Requirements
   5. Deliverables and Schedule
3. [Requirement Specifications](#Bookmark3)
   1. Stakeholders for the System
   2. Use Cases
      1. Graphic Use Case Model
      2. Textual Description for each Use Case
   3. Rationale for your Use Case Model
   4. Non-functional Requirements
4. [Architecture](#Bookmark4)
   1. Architectural Style(s) Used
   2. Architectural Model
   3. Technology, Software, and Hardware Used
   4. Rationale for your Architectural Style and Model
5. [Design](#Bookmark5)
   1. User Interface Design
   2. Components Design
   3. Database Design
   4. Rationale for your Detailed Design Models
   5. Traceability from Requirements to Detailed Design Models
6. [Test Management](#Bookmark6)
   1. A Complete List of System Test Cases
   2. Traceability of Test Cases to Use Cases
   3. Techniques used for Test Case Generation
   4. Test Results and Assessments
   5. Defects Reports
7. [Conclusions](#Bookmark7)
   1. Outcomes of the Project
   2. Lessons Learned
   3. Future Development

**List of Figures**

1. [Figure 1: Graphic Use Case Model](#Bookmark8)

**List of Tables**

1. [Table 1: Database Table](#Bookmark9)

**1. Introduction**

1.1. Purpose and Scope

The purpose of this program is real-time stock tracking which can help businesses manage and monitor inventory through our system.

1.2. Product Overview (including capabilities, scenarios for using the product, etc.)

The capabilities for our product inventory control, order management, reporting, and integration of their needs with use cases for companies with whole stock that needs to be monitored and tracked within the system. Distributors like warehouses will be able to use our system to track inventory and purchases to monitor their inventory.

1.3. Structure of the Document

The structure for this document will include an introduction, an explanation of our design model and database model, the restrictions & constraints, and the testing for each part of the project.

1.4. Terms, Acronyms, and Abbreviations

FIFO - First-In-First-Out

SKU - Stock Keeping Unit

UID – User ID

.NET MAUI: Multi-platform App UI, that utilizes the .NET framework and can be modified within Visual Studio, to create applications that can be used on Windows, Linux, Android, and IOS; for the scope of this project traditional .NET for windows was fine, but MAUI contains additional support that makes the coding easier.

MVC: Model-View-Controller is an architectural model that separates the data, user interface, and logical functions. This pattern is used to separate each work path and is very common with team projects, especially on a corporate level.

Supplier\_id: Information as a string that combines username and password information.

Consumer\_id: Information as a string for who is going to be receiving the inventory, this can be influx inventory for a user (supplier) or an outside party, whose name will be determined under the name string.

Name: Information on who is accessing as a function of a company in conjunction with the username (for example if Microsoft is accessing, it would be Microsoft), or who is receiving the product.

Contact\_info: Email and number information in the form of a string.

Order\_id: Number generated based on the order number as a chronological value.

Order\_date: A date set for the id’s launch that results in the date of transaction being recorded.

Quantity: An integer to showcase size of order request (buy or sell).

Subtotal: Floating value of monetary meaning based on quantity and price per unit.

Price: Unit cost for each product.

Description: A string that will provide a description for each item available.

Quantity\_in\_stock: Available quantity for a company to sell, or for a consumer to purchase from a company.

**2. Project Management Plan**

2.1. Project Organization

Our project organization consists of 4 students who each took on a task to help further the project. We assigned roles based on the strengths of each individual student and met up across the weeks of the projects to work together and help each other out with each part of the project.

2.2. Lifecycle Model Used

We used waterfall for the majority of our starting phases then swapped to agile to work on each of the individual parts from the UI, Database, backend codes, and the exe file. This helped us work on each individual part. This helped us divide up the task after getting the base set up.

2.3. Risk Analysis

Data loss or corruption – High risk. If data is loss

Unauthorized access / data breach – High risk.

System downtime during peak hours – High risk.

Inaccurate inventory data - High risk.

2.4. Hardware and Software Resource Requirements

Windows 10/11, just about any hardware with entry level graphics, can handle this product.

2.5. Deliverables and Schedule

We plan to deliver a fully functioning system on April 28th that has a database and a functioning storage system that companies can use to fulfill their inventory management needs. This system will help users keep accounts of all the items in their inventory and the ones that are leaving that should make inventory management easier. This software was created over the course of 4 months.

**3. Requirement Specifications**

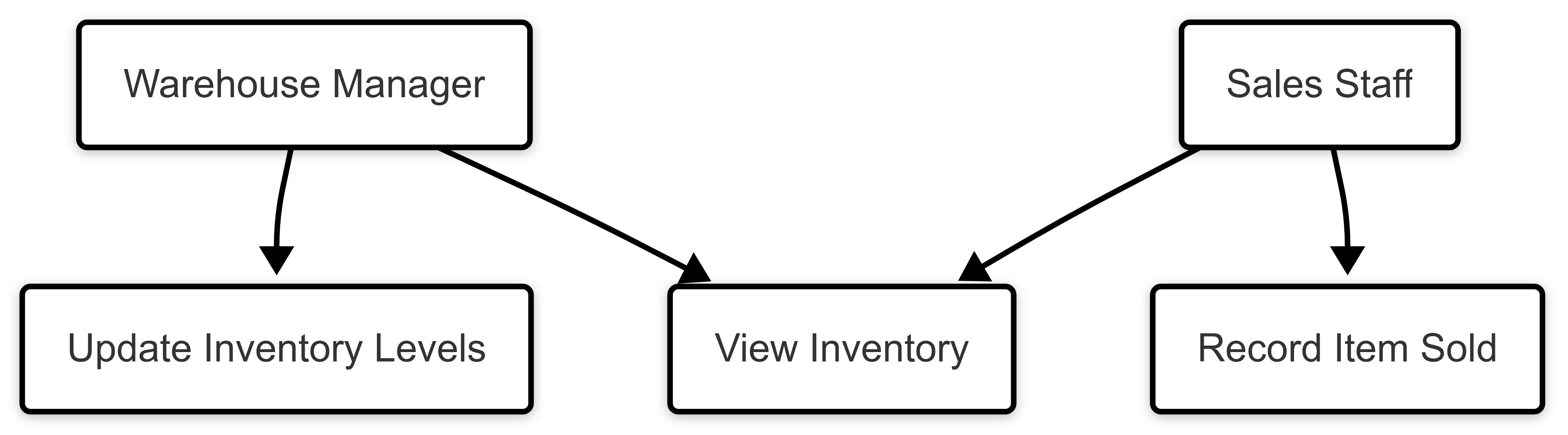
3.1. Stakeholders for the System

This product’s stakeholders are individuals and companies who have access to their own inventory of products that they need to be able to actively track, and market out (both buy and sell). Furthermore, implications that allow product addition and removal directly from this app allow for very easy posting of new content for customers to review. This provides a link to different users of this product to buy and sell directly amongst other users or to outsource their products.

3.2. Use Cases

3.2.1. Graphic Use Case Model

Figure 1: Graphic Use Case Model



3.2.2. Textual Description for each Use Case

In this use case a warehouse manager will update by adding or removing items from the inventory. The inventory will then push the changes so the sales staff can see the total amount of inventory the warehouse has then make sales, which will remove items from the inventory so each staff member can track how many items they have on hand for sales.

3.3. Rationale for your Use Case Model

The use case was designed to how a use case in a warehouse environment. This will show simple it is to use our product in this environment and how the inventory updates in real time to meet the needs of the warehouse.

3.4. Non-functional Requirements

The most import requirement for this software is to be able to update the data in real time if the warehouse makes sales on old data they could run into a problem where they don’t have the i inventory to fulfill the sale which will halt all production until the system is updated. The system needs to be available 99% of the time the warehouse is active. If the system is down, there would be no point to the software purpose. The UI needs to be simple and easy to use and understand. We need to scale the warehouse to be used by all the employees and be maintainable so that the warehouse can update the code to add functionality to match their needs.

**4. Architecture**

4.1. Architectural Style(s) Used

The UI specifically is made up of event driven pages that trigger dynamic reactions and adjustments to the data in the database by calling the logic in the backend, this process is what makes our model an MVC. The backend will function on the same event triggers as it is a series of callable functions that trigger action. The database (SQL) is a relational database as it is a schematic made up of neatly organized tables of information.

4.2. Architectural Model (includes components and their interactions)

The architecture functions as MVC (Model View Controller), with separation of UI, data, and logic. Certain call functions will utilize a UI-backend so that some of the functions in the UI will directly interact with the database by functioning with its own backend, namely for selecting product.

4.3. Technology, Software, and Hardware Used

SQLite, C#, .NET, XAML, Visual Studio.

4.4. Rationale for your Architectural Style and Model

Given our separation into distinct sections, it makes sense for us to utilize MVC as it is an inherently separate parts model that comes together to craft a whole, we stray away from it for some declarations to simplify the backend in situations where MAUI has a better method to quickly request and receive its information. Event based styling was going to be necessary in the UI, given that we utilize buttons, requests and page transitions.

**5. Design**

5.1. User Interface design

The UI is designed in Visual Studio utilizing .NET MAUI and written in C sharp and XAML for the design specifications for each page. Upon startup the user is greeted to a login screen that will accept their user information and allow them to login and view the products in their inventory. Mr. Nguyen has suggested some modifications to give a more inventory system appearance and layout to the introduction and optioning, that looks quite nice, at this time the UI may be adjusted to reflect this old school design, and elements may be added to give a dated look to the product, or quality reduction to improve performance. Once logged in the main page will be loaded, this is where the inventory is stored and the functions for adding new products, buying and selling existing products, and removing products entirely will exist. There is a dedicated search function and methods to adjust the purchase and sell amount per transaction. Each transaction will produce a detailed order report that I will have emailed to myself but hypothetically could be emailed to the user (however these emails will be arbitrary or will be of people who do not want their emails blown up). Active inventory updating will be utilized in a secondary version of the UI code to showcase its abilities but will not be included in the final project as the constant influx of changing data will cause the product to crash without the use of an external server to run the data in and out smoothly, we will include a video of this process in action within our presentation. Active adjusted searching is utilized for a seamless and aesthetically pleasing experience for the user. Other creature comfort functions and simplifications will be implemented to improve user experience, whether the aesthetic will reflect an old school database-backed sorting system, or a newer era gridded list of items is to be determined, experimenting with both will bring us to our conclusion. Currently the UI aspect is cluttered and overly complicated to make up for the lack of proper implementation of the database data and use of the backend functions, calls, and queries; as we bring the UI to its fruition it is now a process of picking the perfect design and then simplifying the UI to correspond to backend calls and proper get and sets with the database. Therefore, pressing forward the UI coding will get dumbed down so that it isn’t functioning so heavily on its own devices and tanking performance but rather allowing external processes and proper linking to create a smooth flowing product for our users.

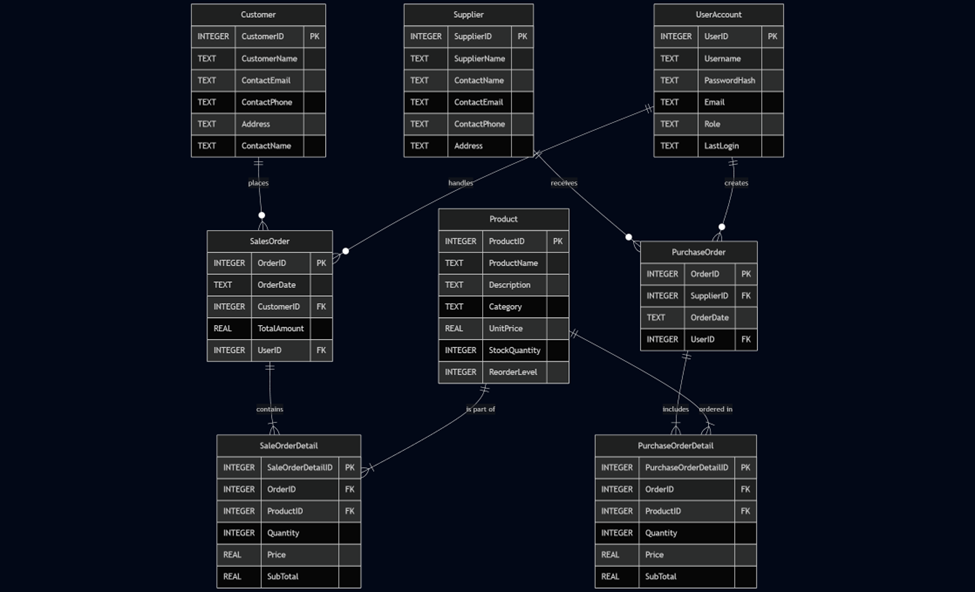
5.2. Components Design (static and dynamic models of each component)

Modeling for the UI system contains a plethora of dynamic models, as every button is an event trigger that causes change and updating to the system or display, the login button is a dynamic event that triggers the page display to completely change the file it will portray for the user. Buy, Sell, Add, Remove are all dynamic events that will call into the backend to make active adjustments to the database’s stored values. For the active inventory adjustment, the process is innate and adjusted on a loop running interval that will trigger multiple real-time events at once. One of the biggest static elements to this product is its initial launch past the login screen on the main page because on the first boot everything displayed is the predetermined products declared in the database; moreover, everything initially declared in the database will be static, and the interactions with the UI that changes these values are the dynamic processes that alter these values. The XAML backing for the design of the display is a group of static models that determine the appearance of each page, element, and its layout and stylization. Notice in terms of the UI, just about every called function and item has a dynamic aspect to it as it can be changed and adjusted, but the value itself as it is returned to the database and stored lays there statically until it is called for adjustment again.

5.3. Database Design

The database is a SQLite formatted table-based system, it contains several main tables, a sequencing table, and the corresponding marker tables (reads, writes, and the variants of those with names unique to the server-side software). SQL and SQLite are built around table systems and SQL commands, these table’s hold are main variables that will be utilized in the UI’s calls and trigger a backend response for pulling and placing, the UI contains classes that correspond to each of these main tables subsystem variable declarations, this allows for uniformity amongst the three aspects of code.

Table 1: Database Entity Relationship Diagram



5.4. Rationale for your Detailed Design Models

The waterfall design model is a tried and tested model that is still very common on the corporate level but was phased out minorly for the agile model; however, for our application the sheer amount of documentation and simplicity regarding the waterfall model made it ample for our uses. Given that we are not a true corporation and rather just four students in a group project, simple and widely accessible is the perfect situation for us. We determined our requirements early on, mapped out the design elements we wanted to implement, and are now developing our individual sides of the code to be able to work in conjunction and be tested both independently and in conjunction.

5.5. Traceability from Requirements to Detailed Design Models

Our product had requirements for the user to be able to view their inventory, adjust their inventory and manage their products in a true buyer/seller market. With the utilization of detailed order reports, active logging (primarily for the final product, not for the version with real time automatic / simulated product displacement and replacement, as this log would get cluttered and overload memory rapidly). We also made requirements for security, represented in our use of a login page, dedicated usernames and passwords, and the capability for password hashing for added user security. We had requirements for searching that we went above and beyond to get perfect, with implication in the backend and UI adjustment actively in the launched MAUI, we have functions that we had determined as non-essential and omitted like +/- buttons, and typing out buy and sell orders, instead utilizing a simple buy sell process makes the entire appearance more appealing and simplifies the product further. Our design modeling and its static placements of variables and dynamic event adjustments for these variables all align with the requirements we had set out to strive for at the beginning of this project.

**6. Test Management**

6.1. A Complete List of System Test Cases

Backend Test Cases: The backend can be tested by calling each function and ensuring it reads and writes thoroughly, you can implement debug techniques like console writeouts either in the UI code or directly in the backend to see what exactly is happening with each execute.

Database Test Cases: The database is indirectly tested with every function, as you can check to see if it received the data change (add / remove), to test directly you can overload the database with massive amounts of data and see how it is handled by the other codes, this would be a method of limit testing.

UI Test Cases: Trace down every on click event to ensure it accurately responds to the stimuli, add a plethora of products to see how the UI handles a massive stretching of the grid, buying a large number of product to see how the values overflow and display, ensure numbers cannot go below 0.

6.2. Traceability of Test Cases to Use Cases

With our uses including multiple users and a plethora of products, these tests are an absolute necessity, we need to ensure that we can handle large amounts of inputs and outputs quickly and efficiently.

6.3. Techniques used for Test Case Generation

For direct database testing the technique would be active placement, as we are calling SQL executives to place new info in, and lots of it, to test for overload. For backend and UI testing the majority of it is classic debugging using console printouts and try catch blocks to get a good mental note of what exactly is and is not happening with each call.

6.4. Test Results and Assessments (how good are your test cases? How good is your software?)

The database can handle a lot of data so no true worry there, the communication through MAUI + backend into the database is a little difficult, to an experienced programmer it might be a breeze but it has taken me (Jake) quite a few tries to get it running well, between traditional SQLite connection and ASYNC I simply don't have the know-how to get exactly which is best for our situation from the get-go, so there is a learning curve that is to be expected. The traditional try statement with printouts method for testing is phenomenal, it is a very well-known and common method in debugging, I would argue that the software is solid, it is still a little cluttered because we have left open room for expansion but it is otherwise well endured, I am still yet to create a version that showcases rapid passive buying and selling but I will get to it around 3 AM tonight when I'm off work.

6.5. Defects Reports

The UI when launched independently has a chance to not even load up properly because of overloading through Visual Studio.

Initially the login page would receive data from the user table in the database but upon launching and entering the data, it appears it will only allow one user to function at a time! Debugging and adjustments to handling multiple table points was necessary to alleviate this.

The product load in was fairly situated in its own private memory, preferring to ignore database calls and hold onto its own declarations selfishly, this was the most difficult part to remedy as we are backstabbing MAUI’s innate nature and preferences to override it with our goals.

**7. Conclusions**

7.1. Outcomes of the Project (are all goals achieved?)

The project actively functions on a MVC interconnected model and was well placed in our git, and all launches properly with an exe, once all the communication is complete it should be a project that meets almost every one of our goals and will meet every one of our requirements. Some of the functions and things we wanted to add simply weren't worth the time investment or took away from the retro laid back look we were going for, so they were omitted, certain variable names under the main tables just were not necessary for our product as well.

7.2. Lessons Learned

Backend frontend coding is a phenomenal team-based method of product development. A lot of the architectural models, styles, and patterns are very helpful in laying foundation for your software process. MAUI’s innate abilities and memory is a comprehensively designed product and could be utilized well for simple app development and outreach. SQLite’s simple table design and access to SQL’s more advanced nature and executable functions while being in a smaller easier to navigate package is an amazing asset to have for storage and true concrete memory usages as a programmer. C sharp, C++ and a plethora of other languages all make for great backend and frontend separation as their nature of callout products, especially with C sharps access to .NET and its insane collection of libraries. Java is also an excellent choice for MVC development, with its object-oriented nature and reliance on abstraction, interfaces, and large networks of superclass's it can make for a great method to larger products and even game development if you prefer to work with it over Python, albeit the C# method with .NET capabilities is well endowed as the easier method to get into, both are well suited for backend-frontend coding and product development as a whole.

7.3. Future Development

Moving forward to perfect and grow this product we could introduce an absolute ton of new content and more pages to access, like more information regarding suppliers, consumers, more email routing, more animation to imagery and aesthetic content, we could push the product using MAUI to Android, IOS, Linux etc. When we designed this, we ensured that we kept modularity and growth in mind; therefore, our creation as it sits is ready to be built upon and expanded, the database we know can handle a ton of more information, the frontend can have all its model classes expanded easily and the inclusion of more packaged XAML and CS files for great user interfacing and routing to simple backend calls that can be added endlessly to our query CS file within the data folder makes for endless possibilities with this project. Since we followed our models and our goal of scalability, we ended up with a project that can be utilized properly and could be developed far beyond where it sits today.