Cover page

PyWare Inventory System

* a Python developed prototype designed to manage inventory -

Student name: Maria-Cristina Aarmeniei-Pricope

Student ID: 30245077

Module: Object Orientated Programming

Assignment: 2 of 2

Submission Date:

**Contents**

1. **Introduction**
2. **Technical Design Document**
   1. Intended Architecture

**Business logic**

Data Managing

Graphical user interface

Controller layer

**OOP design principles**

**Component interaction**

**Scalability**

* 1. Chosen Language and Justification
  2. UML Diagrams, Wireframe and Flowchart

1. **Development and Debugging**

Setup

Development

Debugging

Unit testing

1. **Challenges and Solutions**
2. **Testing Documentation**
3. **Appendices and References**

GitHub URL

1. **Introduction**

This report aims to capture the development of an inventory prototype app for Light Logistics, a company specialised in logistics and supply chain management. The application will include aspects of the four pillars of object-oriented programming (encapsulation, inheritance, polymorphism, and abstraction) for best practice.

Python will be used for the development, due to its simplicity and vast libraries that facilitate database creation, SQL queries and user interface.

**Scope of the project**

The scope of the project is to develop a prototype application for Light Logistics and to demonstrate the four pillars of object-oriented programming. Please see in the table below the in scope and out of scope of the project:

|  |  |
| --- | --- |
| **In scope** | Development of a GUI based inventory prototype using Python and Tkinter to perform functionalities like adding, removing, updating, and viewing inventory.  Integration of the code with an SQLite database.  Use OOP principles in code development.  Unit testing and manual testing for functionality and performance.  Documentation of the development process, design decisions, and testing outcomes. |
| **Out of scope** | **main.py** console-based application intended only for development and testing, but not intended for end-user deployment.  Features such as concurrency, multi-user access.  Machine learning algorithms implementation for analysis and predictions of the inventory.  API integration or other external systems integration.  Deployment to production environments and related security measures. |

1. **Technical Design Document**
2. **Intended Architecture**

The intended architecture of the application is to follow object-oriented programming concepts to manage and interact with the inventory. The solution follows a modular structure which promotes code reutilisation, this includes the use of:

**Business Logic**

Classes:

StockItem class defines the properties and behaviours of all inventory items; it includes attributes like item id, name, quantity and category; it also had an update quantity method which is implemented by subclasses. *Upon completion, a code snippet will be uploaded to Appendix A.*

ConcreteStockItem is a class that extends the capabilities of StockItem class; it offers a method to update the quantity and ensures that the quantities don’t become negative when stock is updated. *Upon completion, a code snippet will be uploaded to Appendix B.*

Warehouse is a class that actions as the main centre for managing stock; it interacts with DatabaseManager to collect and use data and with StockItem and its subclasses to manage the inventory; it offers methods to add, eliminate and view inventory. *Upon completion, a code snippet will be uploaded to Appendix C.*

1. Data managing

DatabaseManager is a class which maintains the connection between the database and operations performed; it creates the database table and maintains CRUD (create, read, update and delete) operations. *Upon completion, a code snippet will be uploaded to Appendix D.*

1. Graphical user interface

WarehouseGUI is a class that offers the graphical user interface for the user to interact with the system; it uses Tkinter, a Python library, to create the main window of the application where users can add, remove or view inventory . *Upon completion, a code snippet will be uploaded to Appendix E.*

1. Controller layer

**main.py** script is the entry point when the application runs in command line interface mode; it handles the user text, and it interacts with the Warehouse class to perform inventory actions; it closes the connection once the program closed. *Upon completion, a code snippet will be uploaded to Appendix F.*

**OOP design principles**

Encapsulation is used in the Warehouse and StockItem classes to encapsulate the details of managing and stocking items; users can interact with high level functions like adding or removing items from inventory without the need to understand how they work. *Upon completion, a code snippet will be uploaded to Appendix G.*

Inheritance is found in the relationship between StockItem and ConcreteStockItem; ConcreteStockItem inherits from StockItem allowing extension but keeping at the same time the common attributes and methods; in this example, inheritance promotes code reutilisation and efficiency. *Upon completion, a code snippet will be uploaded to Appendix H.*

StockItem is an abstract class which offers a common interface for stocked items; by defining behaviours and common properties, the system is flexible and allows for extension or modification without changing the main logic of the application. *Upon completion, a code snippet will be uploaded to Appendix I.*

The application is designed based on modular components: DatabaseManager – used to manage data, Warehouse and StockItem – used for the business logic and WarehouseGUI – used to manage the user interface. Please see the UML diagram further down in the **UML Diagrams, Wireframe and Flowchart** section.

**Component interaction**

The user interacts with the GUI through WarehouseGUI to perform actions to add, remove or view inventory; the choices will then initiate methods from the Warehouse class to interact with the inventory to make the necessary changes; finally, the Warehouse class interacts with the DatabaseManager class to update the database. For example, when the user adds an item, WarehouseGUI initiates add\_item() method in the Warehouse class, which then calls the add\_item\_to\_db() in DatabaseManager.

In CLI mode, main.py interacts directly with Warehouse and StockItem classes to allow the user to perform actions; the communication is based in and out text and it takes place without a graphical interface.

DatabaseManager takes care of the interaction with the SQLite database, it hides the way the functions as other classes don’t need to know the how the database functions. Other classes can access the DatabaseManager through getter and setter methods like add\_item\_to\_db() or get\_items\_from\_db().

**Scalability**

The system s projected to expand in order to allow for new item types to be added; for example new subclasses could be added from StockItem, like CustomStockItem. Functions to generate inventory reports could be added to help the business align with its strategic goals, it could include machine learning algorithms to detect and predict patterns in the data. *Upon completion, a code snippet will be uploaded to Appendix J.*

Because the application is based on a modular approach, the structure allows for new functions that could help in integrating with new databases, or for adding new characteristics to the GUI, for example.

1. **Chosen Language and Justification**

For this application Python has been deemed the most suitable programming language. Python is a high-level interpreting language known for its simplicity and versatility; it allows developers to focus on the business logic rather than on a complex syntax.

Python syntax is easy to understand and as a result it aids in faster development and maintenance, which is especially useful when for small applications or developing prototypes.

Classes like Warehouse and DatabaseManager are easy to read and follow, which helps developers understand the flow and logic of the system.

Python has an extensive library that includes support to manipulate data, files and database interactions which reduces the dependencies and speeds up the development process; for example, sqlite3 is a module used for database interactions. Another example is unittesting module, which can be used for test and debug individual units in the code. *Upon completion, a code snippet will be uploaded to Appendix K.*

Python is versatile and can run on various operating systems like Windows, macOS and Linux with little or no modifications which means that it is accessible to a variety of users without the need to rewrite the code.

Python accepts object-oriented programming concepts to create classes (like StockItem, Warehouse and ConcreteStockItem), define attributes and methods, use polymorphism, inheritance, encapsulation and abstraction.

Python is easy to maintain and extend, it can allow for new functionality and support to be added, thanks to its extensive libraries, which makes it ideal for the development of an inventory management ecosystem(including back-end and front-end development) enriched by libraries and community support.

1. **UML Diagrams, Wireframe and Flowchart**

The UML (Unified Modeling Language) diagram created for the inventory prototype represents the structure and relations between various classes in the system in order to facilitate inventory management. The UML diagram displays the following relationships:

- main.py uses DatabaseManager and Warehouse

- Warehouse uses DatabaseManager

- WarehouseGUI uses Warehouse

- ConcreteStockItem inherits from StockItem

A screenshot of a computer

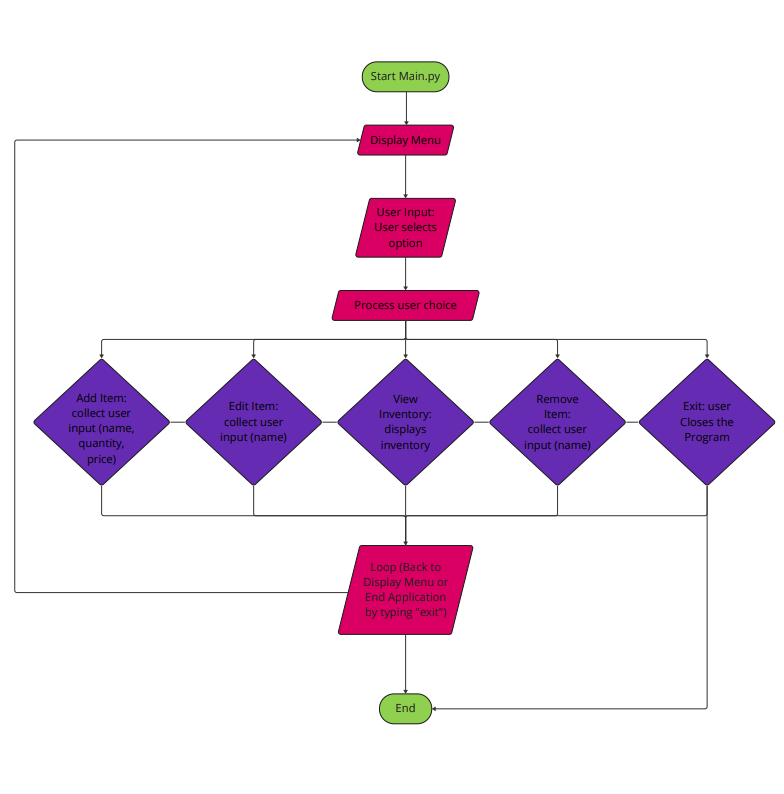
Description automatically generated

GUI (gui\_warehouse.py) Wireframe

A screenshot of a computer

Description automatically generated

Console (main.py) Flowchart



1. **Development and Debugging**

**Setup**

Development begun with structuring the code into separate files, achieving warehouse\_system.py for business logic, gui\_warehouse.py for user interface and main.py to allow the main app to run in CLI.

**main.py** provides a CLI entry point to the application:

A black screen with white text

Description automatically generated

**warehouse\_system.py** contains the business logic and data management - DatabaseManager class has been created to manage operations using SQLite, the create\_table() method to create a table for the inventory stock when the application is launched.

A computer screen with text and images

Description automatically generated

Business logic to managed stock items was implemented in StockItem and ConcreteStockItem classes; Warehouse class was developed to manage the inventory process using DatabaseManager which is used to retain the data.

**gui\_warehouse.py** handles user interface using Tkinter - WarehouseGUI class was built using the Tkinter library to create a simple graphical user interface that allows the users to interact with the inventory; it includes buttons to add and eliminate items and the Treeview widget to show the inventory elements.

A screen shot of a computer program

Description automatically generated

**Development**

The system was developed in a modular design, each unit has been tested individually and added in the larger application. The distinct modules created are:

* **warehouse\_system.py** designed to handle the business logic and data management.
* **gui\_warehouse.py** provides the user interface using the Tkinter library.
* **main.py** acts as a command-line interface for testing and development purposes only.

The main functionalities of the prototype to add, view and eliminate items were later improved using more functions, for example showing the inventory in an easy-to-use interface.

After the implementation of each characteristic, unit testing was created; for example the DatabaseManager class was assessed independently to validate that CRUD operations (Create, Read, Update, Delete) were functioning correctly. This involved creating a test SQLite database to simulate database interactions without interfering with the real database, as shown in the code snippets below:

A computer screen shot of a program

Description automatically generated

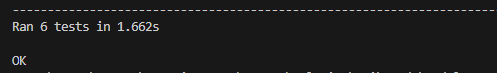
A computer code on a black background

Description automatically generated

The StockItem and Warehouse classes were also assessed to ensure OOP compliance and business logic. For example, tests confirmed that the update\_quantity method prevented negative stock levels.

A screen shot of a computer code

Description automatically generated



**Debugging**

Throughout the development process, error management was integrated in the code to prevent crashes due to unexpected input. To do this try-except block were used to detect errors caused by invalid user input, for example adding negative quantities.

For the prototype, a logging system wasn’t implemented , instead, print statements were used to follow the workflow and identify potential problems. Before the system is deployed to production, a structured logging system can be implemented if necessary.

Clear and descriptive error messages were incorporated in the GUI to help the user understand what is missing or incorrect, for example the user would see a ValueError if trying to add an item with incomplete information.

VS Code was used as IDE for development and debugging, it highlighted inconsistencies and errors, and it also offered the Run and Debug function, which is especially useful for testing using breakpoints.

**Unit testing**

To ensure that individual code units function as expected, unit testing was developed, using the unittest module in file **test\_warehouse\_system.py**, especially for testing the methods interacting with the database, for example add\_item\_to\_db(), remove\_item\_from\_db() and get\_items\_from\_db(). To test the main operations, mock objects were used to simulate database responses without using a real database. For the business logic, the method update\_quantity() from the ConcreteStockItem class was tested to ensure negative quantities are not allowed and that stock levels are updated correctly.

**Manual testing**

Manual testing was performed, and test documentation were logged with pass and fail scenario to complement automatic tests, mostly related to user interface where manual checks are necessary.

1. **Challenges and Solutions**

The main challenge during development is to ensure a stable and fluid connection between the prototype and the SQLite database, to ensure the data is coherent and manages exceptions.

One challenge was to manage correctly the database connection, especially when multiple operations are performed; to solve this, DatabaseManager class was created to manage the opening and closing database connection; close() method ensures the connection is closed correctly when the program is terminated which prevents connection issues and database locks.

Another challenge was to validate the user entries through the GUI and the backend logic to accept only valid entries; to address this, input validation was added in both the GUI and backend logic; validation was added for empty fields and numeric values, together with error messages to warn the user of the invalid or missing data.

Another potential challenge is concurrency issues, which can arise when the database is modified from different points, which can cause inconsistency in the database; even though this wasn’t a concern at this stage of prototype, future versions could implement mechanisms to manage transactions and lock records when in use to ensure thread-free access to the database.

1. **Testing Documentation**

For this inventory management prototype application, testing was conducted to ensure functionality, reliability and performance; for this a unit testing and manual testing was done to validate the main characteristics of the system to add, delete and view inventory.

Unit testing was done to test individual components and validate the business logic to help detect errors in the development process. For this step, Python’s unittest module was used and focused on: effective manipulation of data input, validation for invalid entries like empty fields or negative values and main operations like viewing, adding and removing items.

Besides unit testing, test cases were created to include most scenarios of user interaction with the GUI and understand the behaviour end-to-end. Test cases aim to cover usual actions and isolated cases such as invalid input. Test scenarios include Pass and Fail cases, trying to simulate real examples of user interactions.

Both unit testing and test cases offer trust that the system can handle operations and function correctly in the real world.

The test cases were documented in the following section:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test ID** | **Test Case** | **Input** | **Expected Output** | **Result** | **Notes** |
| FT 01 | **Add item with valid details** | **Name: box**  **Quantity: 10**  **Category: BOX** | **Item added to inventory, no errors** | **Pass** |  |
| FT 02 | **Add item with empty name** | **Name: “”**  **Quantity: 10**  **Category: INLAY** | **Error:**  **“Invalid input: Name and category cannot be empty.”** | **Pass** |  |
| FT 03 | **Add item with negative quantity** | **Name: sup001**  **Quantity: -10**  **Category: INLAY** | **Error:**  **“Quantity cannot be negative.”** | **Fail** | **Item was added to inventory. Add item validation in main.py and gui\_warehouse.py** |
|  | | | | |
| RT 03.1 | **Regression testing**  **Add item with negative quantity** | **Name: sup001**  **Quantity: -10**  **Category: INLAY** | **Error:**  **“Invalid input: Quantity cannot be negative.”** | **Pass** | **Passed in main.py and gui\_warehouse.py** |
| FT 04 | **Add item with non-numeric quantity** | **Name: core**  **Quantity: 1oo**  **Category: CORE** | **Error:**  **“Invalid quantity: invalid literal for int() with base 10: '1oo'”** | **Pass** |  |
| FT 05 | **Remove existing item** | **Item ID: 4** | **Item removed successfully from the databse.** | **Pass** |  |
| FT 06 | **Remove non existing items** | **Item ID: 199** | **Error:**  **Item with the specified ID does not exist.** | **Fail** | **Fails when run in CLI via main.py displaying that the item has been removed successfully, even thought this doesn’t exist. Doesn’t fail in GUI as it requires the user to highlight by clicking on the line to delete, and item ID 199 is not in the list.** |
|  | | | | |
| RT 06.1 | **Regression testing**  **Remove non existing items** | **Item ID: 199** | **Error:**  **“Item with the specified ID does not exist.”** | **Pass** |  |
| FT 07 | **View inventory** | **View inventory** | **Inventory is displayed by selecting choice number 3 in CLI or by running GUI** | **Fail** | **Fails with message: “Current Inventory:**  **No items in inventory.” Even though items are in inventory.** |
| **Action: The Warehouse class is using an in-memory list (self.items) to store items, while the DatabaseManager manages items in a SQLite database. Update the Warehouse class to use the database directly.** | | | | |
| RT 07.1 | **Regression testing**  **View inventory** | **Select option 3 in Main.py to view inventory** | **Inventory is displayed by selecting choice number 3 in CLI or by running GUI** | **Pass** |  |
| FT  08 | **Update item** | **Update item ID 1 with valid values**  **Item ID: 1**  **Name: box**  **Quantity: 100**  **Category: BOXES** | **Item ID: 1**  **Updated as expected.** | **Pass** |  |
| FT  09 | **Update item** | **Update item ID 1 with invalid values**  **Item ID: 1**  **Name: box**  **Quantity: -10**  **Category: BOX** | **Error:**  **“Quantity cannot be negative.”** | **Pass** |  |
| FT  10 | **Invalid item ID removal** | **Item ID: 1a** | **Error:**  **"Invalid choice. Please try again."** | **Pass** |  |
| FT  11 | **Remove item with ID as String** | **Item ID: one** | **Error message: "Invalid ID. Please try again."** | **Pass** |  |
| FT  12 | **Remove item with Invalid ID** | **Item ID: -1** | **Error message: "Invalid ID. Please try again."** | **Pass** |  |
| FT  13 | **Add item with only Category** | **Name: "", Quantity: 10, Category: "Category E"** | **Error message: "Name and category cannot be empty."** | **Pass** |  |
| FT  14 | **Exit Application** | **No input, just click "Exit" button** | **Application closes without errors** | **Pass** |  |

Testing has been executed at different stages of development to identify issues and improvements. Changes to the code were followed by regression testing to ensure the system still functions as expected.

1. **Appendices and References**

Appendix A

A screen shot of a computer program

Description automatically generated

Appendix B

A screen shot of a computer code

Description automatically generated

Appendix C

A screenshot of a computer program

Description automatically generated

Appendix D

A computer screen with text

Description automatically generated

Appendix E

A computer screen shot of a black screen

Description automatically generated

Appendix F

A screen shot of a computer

Description automatically generated

Appendix G

A black background with white text

Description automatically generated

Appendix H

A screen shot of a computer code

Description automatically generated

Appendix I

A computer screen shot of a code

Description automatically generated

Appendix J

A black background with white text

Description automatically generated

Appendix K

A computer screen shot of a program code

Description automatically generated

A computer screen with text on it

Description automatically generated

Appendix L: GitHub and Git

GitHub URL: <https://github.com/mp3296/Warehouse-System/tree/main>

Version control using GitBash to push changes into GitHub repository:





A computer screen shot of a black screen

Description automatically generated

Version history:

Worked on a local test-branch, which has been merged with master branch. The test-branch has been deleted after the last merge to follow GitHub and Git good practices:

A computer screen shot of a black screen

Description automatically generated

A computer screen shot of a test

Description automatically generated

A screen shot of a computer

Description automatically generated