Final Project Documentation

1. Introduction

Our Final Project is an online **Furniture Store Management System**, the code was written in Python with a TDD approach. We used GitHub for version control and GitHub workflows to maintain a CI/CD pipeline, we also used MySQL to handle all our databases.

Our app aims to provide an e-commerce experience for a store that operates in Eilat, Israel and offers various furniture for sale (Dining Tables, Gaming Chair, etc.).

Key Features

- **User Authentication**: Register, login, and manage accounts.
- **Inventory Management**: Adding, updating, and deleting furniture items.
- **Order Processing**: Users can place and manage orders.
- **Matching Items**: After an item is added, our system offers the client matching items.
- **Credit:** Users get credit for future purchases from managers, or after completing checkout (buying furniture).
- **Coupon codes:** Our store allows users to get discounts via unique codes.

2. Additional features

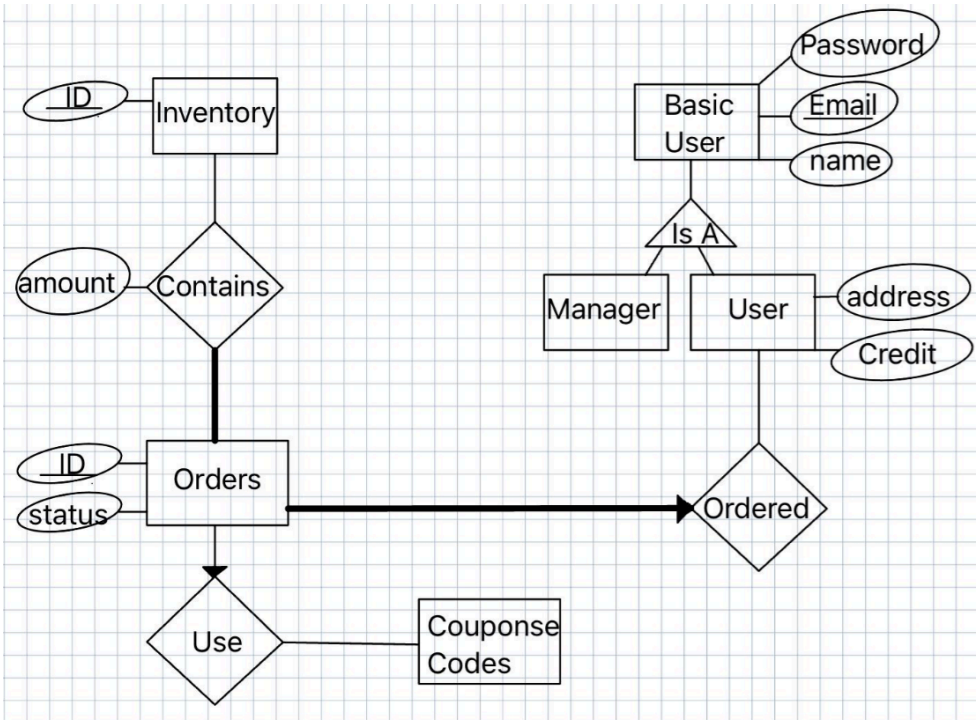
- We enhanced the project with a **credit system** and a **coupon-based discount system**. Customers earn 10% credit from each purchase, which is automatically applied to their account. For example, a user with 100 credit points buying for 120 will pay 20 and gain 2 new credits for next purchases. Credit data is stored for each user in the UserDB.

- The coupon system allows discounts via unique codes. Coupon data is stored in a unique CouponsCodes table. The system validates the code and applies the discount. For instance, using a "SAVE10" (10%) coupon on a 200 order reduces the total to 180.

- When a client adds an item to his cart, our system automatically checks for an available matching item and offers it to the client.
  A matching item to offer will be chosen by the unique set of features and by its availability in our inventory.

3. Database Schema

The following **Entity-Relationship Diagram (ERD)** represents the database structure:



Tables:

- **BasicUser**: Stores user credentials.
- **Users**: Stores user details.
- **Managers**: Special users with admin privileges.
- **Inventory**: Stores product details.
- **Orders**: Tracks all purchases and their status (PENDING, SHIPPING, DELIVERED).
- **CouponsCodes**: Manages discount codes.
- **OrderContainsItem**: Tracks items in each order.

We chose to work with MySQL to create and manage our database so our project will be scalable and will fit large amounts of data. We chose this specific ERD structure so we could track our inventory, orders, and different users of our store.

4. Design Patterns

The project incorporates several design patterns to enhance modularity, maintainability, and scalability. Below is an overview of several patterns used and why they were chosen for this specific implementation. In addition, we implemented core OOP principles as taught in class, including **Inheritance, Abstraction, Encapsulation** and **Polymorphism** to ensure a structured, maintainable, and scalable codebase.

Factory Pattern

The Factory Pattern is used to centralize and manage the creation of furniture objects, ensuring a clear separation between object creation and its usage. Instead of creating furniture instances throughout the code, the FurnitureFactory class handles all object creation in one place, improving organization and maintainability.

The FurnitureFactory class provides a `create_furniture` method that generates different furniture objects (DiningTable, WorkDesk, CoffeeTable, WorkChair, GamingChair) based on the given type. This keeps the object creation logic structured and prevents duplication across the codebase.

Why We Chose It:

- Centralized Object Creation – All furniture instances are created in one place.
- Easier Maintenance – Adding new furniture types requires minimal code changes.
- Cleaner Code – Prevents repetitive instantiation logic throughout the project.

Singleton Pattern

The Singleton Pattern ensures that only one instance of a class is created and used throughout the application. This prevents redundant object creation, ensures consistency, and improves performance.

Implementation examples:

Authentication (Users.py) – Ensures user authentication is managed through a single instance, preventing redundant authentication logic and ensuring all user sessions are validated uniformly.

Inventory Management (Inventory.py) – Maintains a single source of truth for inventory data, preventing inconsistencies due to multiple instances modifying inventory quantities.

Decorator Pattern

The **Decorator pattern** allows extending or adding functionality to a function or class **without modifying its original code**. This is achieved by wrapping the function or class with another function, typically using the `@` symbol.
Decorators are used to enhance functionality in a **concise and reusable** way.
These decorators improve **code safety** and **readability**.

Usage:

`@typechecked` - This decorator performs runtime type checking, ensuring that the parameters passed to functions match their declared type hints. If an incorrect type is passed (e.g., an integer instead of a string), a `TypeError` is raised at runtime.

@property: Controls access to class attributes, ensuring encapsulation.

ENUM Pattern

ENUM's define **fixed constant values**, improving code clarity and preventing errors.
These ENUMs ensure values remain **consistent** and **immutable** throughout the system.

Usage:

- **OrderStatus**: Defines order states (`Pending`, `Shipped`, `Delivered`).
- **FurnitureType**: Categorizes furniture items (`DiningTable, WorkDesk, CoffeeTable, WorkChair, GamingChair`).

Composition

Composition is used when a class contains another class but does not inherit from it.
This approach increases modularity and prevents unnecessary deep hierarchies.

Usage example:

- **User** "has a" **ShoppingCart**, meaning a user owns a shopping cart, but it is not a subclass of User.

5. OOP Principles

Inheritance

Inheritance allows a **class (child)** to derive properties and methods from another **class (parent)**, enabling **code reuse** and **hierarchical structuring**.

Users Inheritance

- **User** and **Manager** inherit from **BasicUser**, gaining common attributes and methods.
- **BasicUser** serves as a **parent class** for both, enforcing shared behavior.

Furniture Inheritance

- **DiningTable, WorkDesk, CoffeeTable** inherit from **Table**.
- **WorkChair, GamingChair** inherit from **Chair**.
- **Table** and **Chair** inherit from **Furniture**.

ABC (Abstract Base Class) ensures that classes that inherit from it won't be instantiated.

Abstraction

Abstract Classes (ABC) and Abstract Methods enforce a contract for subclasses, ensuring they implement necessary functionality. BasicUser (Users module) and Furniture (FurnituresClass module) are abstract classes requiring their subclasses (User, Manager, or various furniture types) to implement key methods, maintaining consistency and structure.

Encapsulation

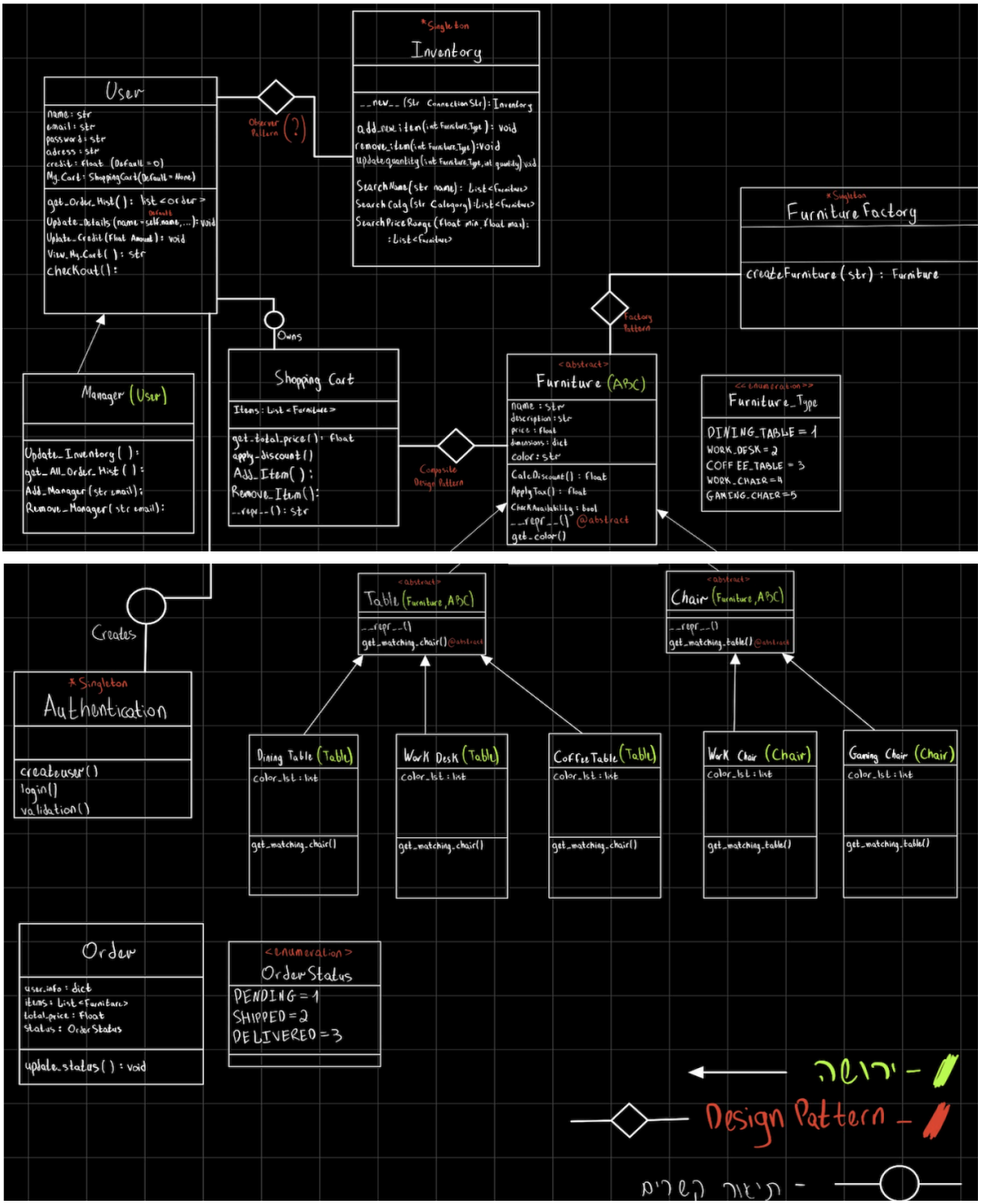
Encapsulation is enforced through private (`__`) and protected (`_`) attributes, with getter and setter methods ensuring controlled access.

Polymorphism

Polymorphism is evident in the Furniture module, where multiple furniture types implement common methods with identical names, ensuring a unified interface while allowing specific behaviors.

5. Project Evolution-

- **Initial design, submitted in HW4:**



Design Refinements

1. Initially, we planned for **Manager** to inherit from **User**. After realizing the Manager didn't need all User functionalities, we introduced an abstract class **BasicUser** as a parent class for both **Manager** and **User**.
2. Due to rejects given on home assignment #4, we added that when a user commits checkout and purchases items from his cart, the inventory immediately updates to the correct amount to prevent inconsistencies.