جامعة جدة
University of Jeddah

Course Project
CCSW 413 Software Design Patterns

MEMBERS:
Anmar Hani (2140004), Raef Shah (241165), ZIYAD ALGHAMDI (2142017), SAMER AWAJI (2140332).

Project Problem Statement:

Python Inventory Management System with Comprehensive Functionality. This system caters to the needs of suppliers, managers, and customers, providing a centralized platform for efficient inventory control

Usage of Design Patterns (Creational)

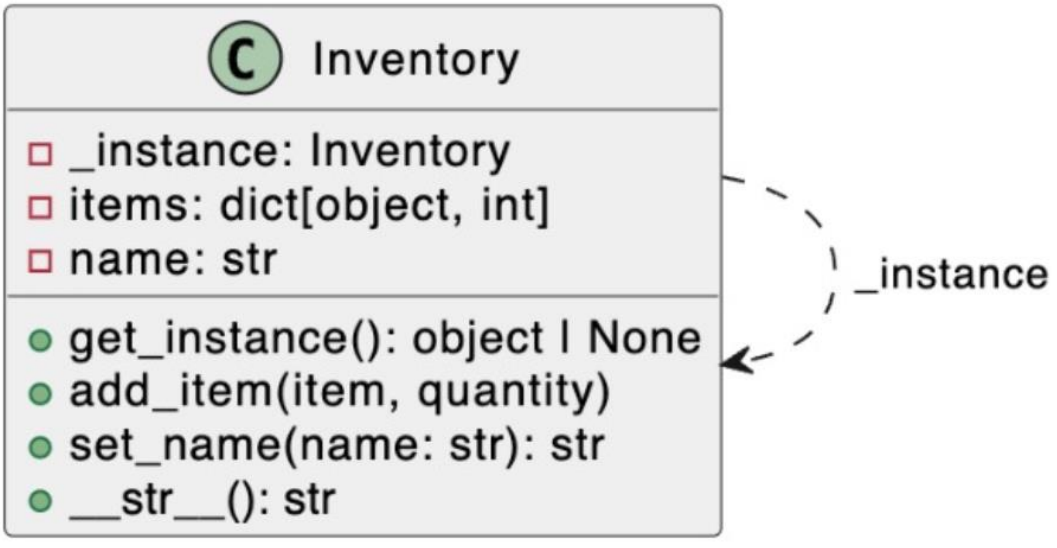
Singleton Pattern (Inventory):

We use the Singleton pattern for the Inventory class to ensure that there is only one instance of the inventory in the system. This is important because having multiple instances of the inventory can lead to inconsistencies in the inventory data.

Benefits:

- Ensures a single source of truth for inventory data. All code interacts with the same inventory object, preventing inconsistencies.
- Improves memory efficiency by avoiding unnecessary object creation. Since only one instance exists, memory usage is optimized.
- Provides a controlled access point for managing inventory data. The get_instance method acts as the single point of entry to interact with the inventory.

Inventory Class Diagram:



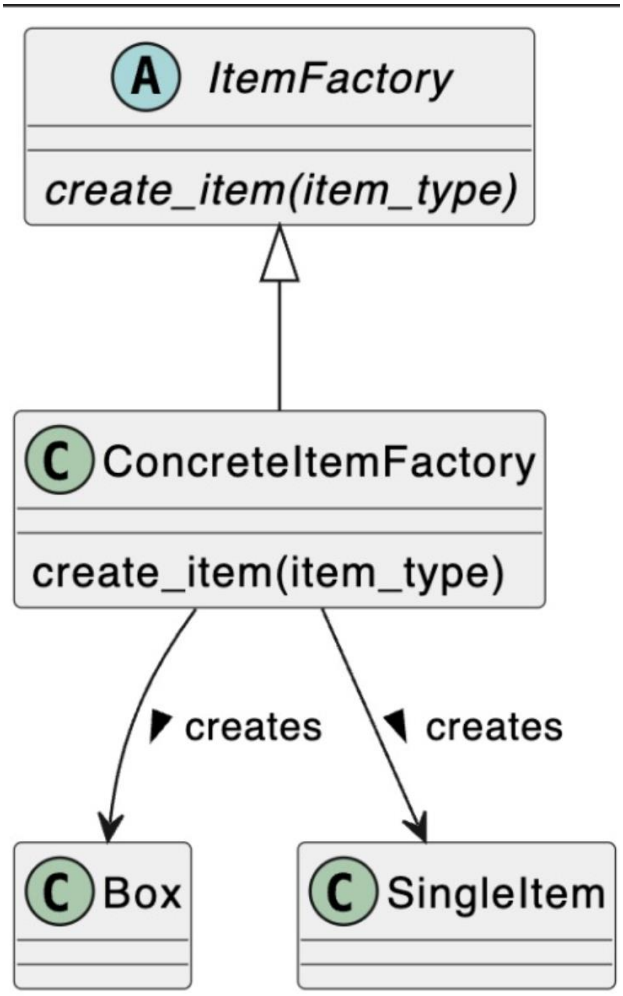
Factory Pattern (ItemFactory):

The ItemFactory class defines the interface for creating items. It uses the abstract method create_item which subclasses must implement to define the logic for creating specific item types. This enforces a standard way of creating items while hiding the concrete implementation details from the client code.

Benefits:

- Inherits the benefits of the Factory Pattern from its parent class.
- Defines the concrete logic for creating Box and SingleItem objects.

ItemFactory Class Diagram:



Usage of Design Patterns (Structural)

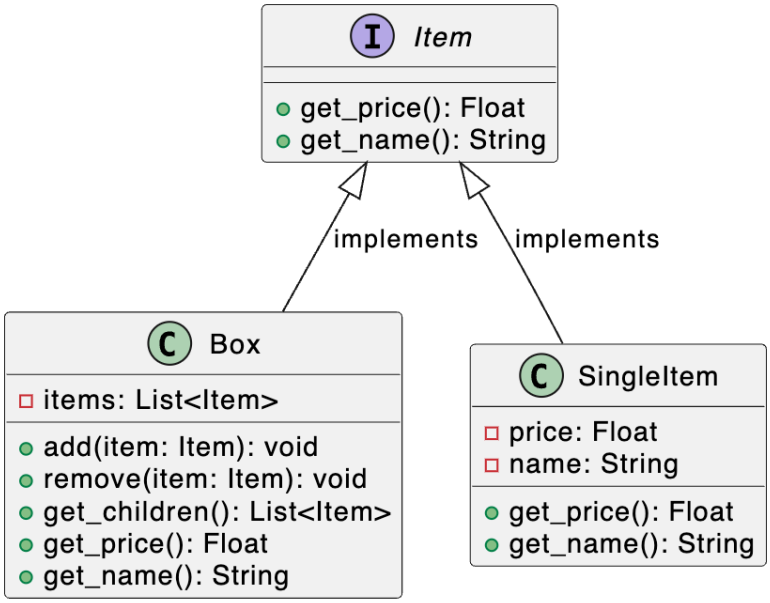
Composite Design Pattern (Item,Box,SingleItem):

The Composite Pattern allows you to create tree-like structures where both individual objects (SingleItem) and composite objects (Box) can be treated uniformly. This enables building complex hierarchical relationships between items in the inventory.

Benefits:

- Represents part-whole hierarchies effectively. Complex item structures can be modeled by nesting boxes and single items.
- Uniform treatment of individual and composite items. Client code interacts with both types using the same interface (Item), simplifying code and improving maintainability.
- Recursive operations. Methods like get_price can be implemented recursively to automatically traverse the entire item hierarchy and calculate the total price.

Item,Box,SingleItem Class Diagram:



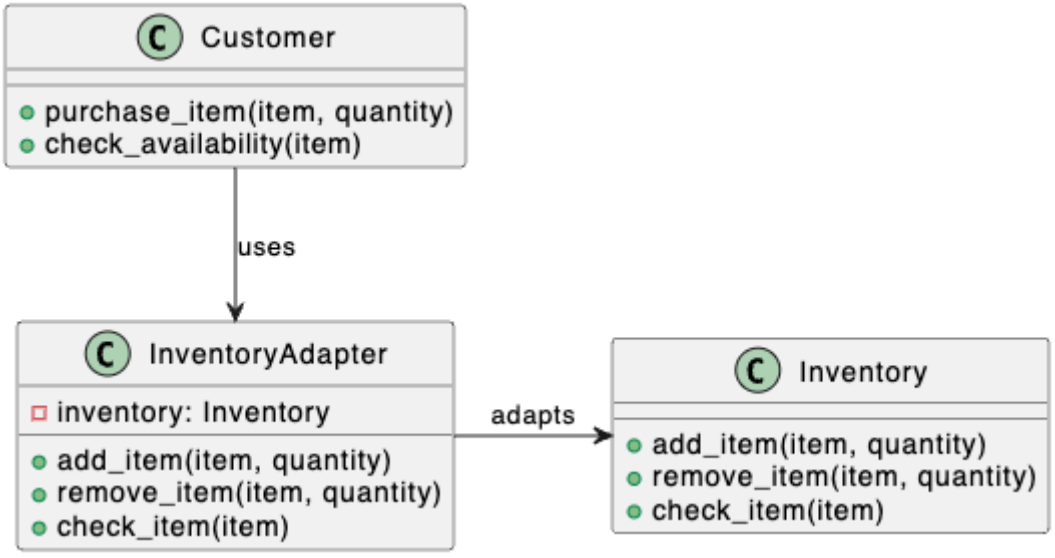
Adapter Patten (InventoryAdapter)

The Adapter Pattern provides a bridge between an existing interface (Inventory) and the client's expected interface. In this case, InventoryAdapter adapts the Inventory class to provide methods with a naming convention that might be more familiar to the client code (e.g., add_item instead of the original add_item with two arguments).

Benefits:

- Allows using an existing class (Inventory) with a different interface without modifying the original class.
- Improves code readability and maintainability by providing a more intuitive interface for the client.
- Promotes loose coupling between the client code and the adapted class.

InventoryAdapter Class Diagram:



Usage of Design Patterns (Behavioral)

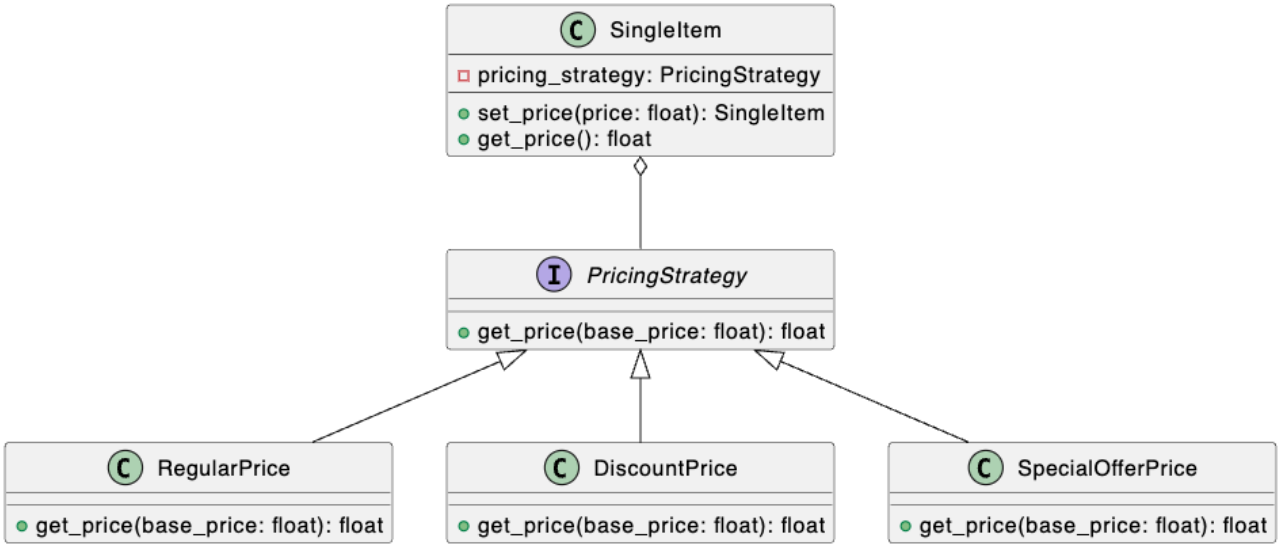
Strategy Design Pattern (PricingStrategy, RegularPrice, DiscountPrice, SpecialOfferPrice):

The Strategy Pattern allows defining different algorithms (pricing strategies in this case) for a specific operation (get_price) and switching between them at runtime. This promotes flexible behavior without modifying the core functionality of the class that uses the strategy (SingleItem).

Benefits:

- Encapsulates different algorithms as separate classes (strategies).
- Promotes loose coupling between the context (the class using the strategy) and the concrete algorithms.
- Enables dynamic selection of algorithms at runtime based on specific conditions.

PricingStrategy, RegulatPrice, DiscountPrice, SpecialOfferPrice Class Diagram:



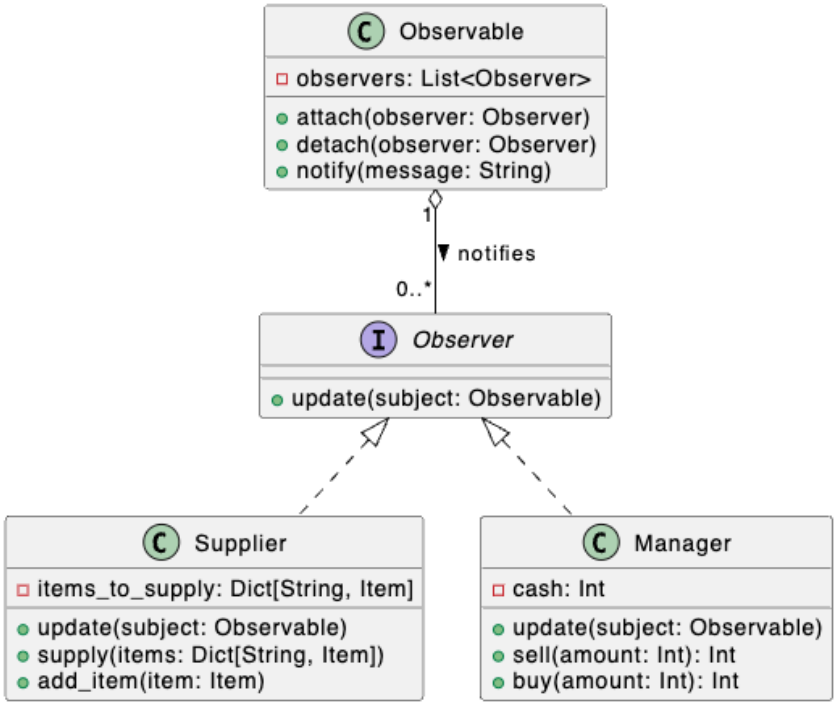
Observer Design Pattern (Observer, Supplier, Manager,Observable):

The Observer Pattern defines a one-to-many dependency between objects. It allows an object (the Subject or Observable) to notify other objects (the Observers) about any changes in its state. This creates a loose coupling between objects, as observers don't need to know the specifics of the subject's implementation details

Benefits:

- Decouples objects that need to stay informed about each other's state changes. Observers don't need to directly call the subject to check for updates.
- Promotes flexibility in adding or removing observers without affecting the subject or other observers.
- Enables efficient notification to multiple observers at once.

Observer,Supplier,Manager,Observable Class Diagram:



Updated Source Code:

https://github.com/AnmarHani/InventoryManagementSystem