DESİGN PATTERN GROUP PROJECT

20210808030	MEHMET ALİ KESKİN
20180808086	AKIN BUZKUŞ
20230808622	ŞEVVAL YÖNTEM

Inventory Management System

USER VERIFICATION AND STOCK CONTROL

Effectiveness and appropriateness of the chosen design patterns

We used composition, observation and factory method patterns in our project.

1-Composite pattern

Definition: is a structural design pattern that allows you to compose objects into tree-like structures to represent part-whole hierarchies.

Areas where we apply composite pattern in Inventory Management System.

The Composite Pattern is used to model a product hierarchy (ProductComponent, Product and ProductGroup classes). It provides a structure where individual items (SmartPhone, Charger) and collections of items (Phonebox, Mainbox) are treated in the same way.

This is crucial for organizing products and ensuring that the system handles both single products and groups seamlessly.

Effectiveness:

The use of a shared interface (ProductComponent) ensures consistent operations (Add, Remove, Show) for both individual products and groups.

The hierarchical structure (ProductGroup with children) facilitates dynamic addition or removal of components, reflecting real-world scenarios such as product bundles or collections.

Appropriateness:

The problem domain (managing individual products and product groups) is ideal for the Composite Pattern.

The pattern simplifies the traversal of the BOM and allows operations such as displaying the hierarchy or calculating total quantities to be applied recursively.

2-Factory Method Pattern

Definition: The Factory design pattern abstracts the object creation process by delegating the creation of an object to a centralized factory. It creates objects dynamically depending on subclasses or parameters.

Areas where we apply Factory method pattern in Inventory Management System.

The Factory Method Pattern is used to encapsulate the creation logic of different product lines (SmartPhoneBoxFactorySamsung, SmartPhoneBoxFactoryApple) and allows easy extension for new product categories.

Effectiveness:

The pattern ensures that product creation is consistent and centralized. It allows new product types or manufacturers (for example, additional factories for other brands) to be added without changing the core logic.

The abstraction of the build logic reduces complexity in the client code, making the system easier to maintain and extend.

Appropriateness:

The problem domain (managing various product groups) is perfectly aligned with the Factory Method Pattern.

Each factory class can encapsulate specific rules for product creation (for example, packaging specific chargers or accessories for a Samsung or Apple product).

3-Observer Pattern

Definition: The Observer design pattern automatically notifies other connected objects when an object changes.

Areas where we apply Observer pattern in Inventory Management System.

The Observer Pattern is implemented to monitor stock levels and notify the relevant components (Orders, Class Name) when inventory changes occur. This enables real-time updates and reactive behavior.

StockNotifier acts as the subject, while classes like Orders and Class Name act as observers..

Effectiveness:

This model provides decoupling between the stock management system and dependent components (e.g. order processing). This increases scalability and maintainability.

Real-time notifications ensure that changes in inventory are addressed immediately.

Appropriateness:

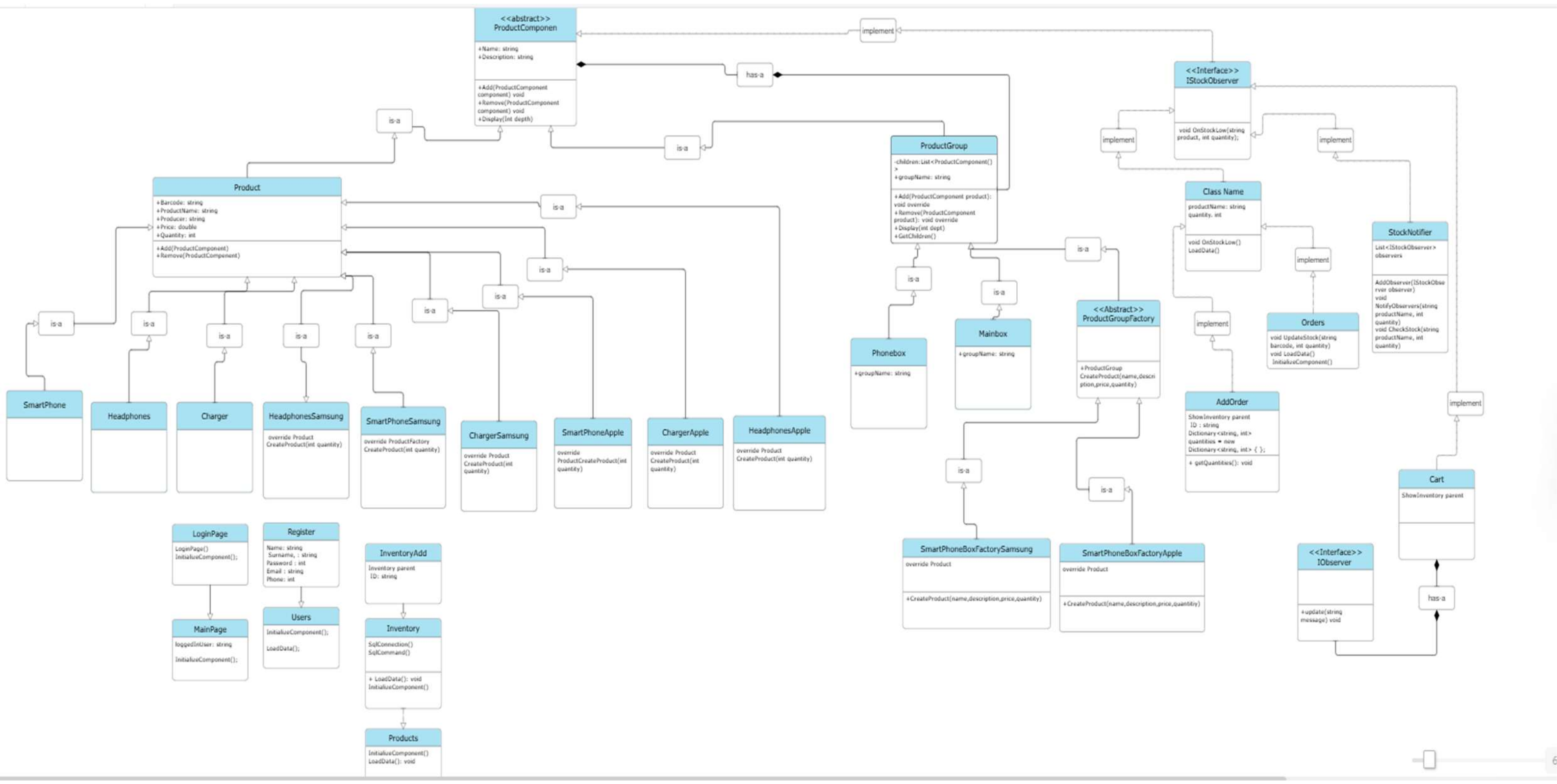
While the model is suitable for handling inventory changes, the design could be further optimized:

There is no clear distinction between critical and non-critical updates (e.g. low stock and normal stock).

If not managed appropriately, it can lead to over-notification (e.g. unnecessarily notifying multiple observers for minor changes).

Implementing more sophisticated observer management (e.g. priority-based notifications or filtering) can improve its effectiveness.

UML diagrams and explanations



DEFİNATİON:

Composite Pattern

Objective:

Composite Pattern is a design pattern used to organize objects in a tree structure. In this pattern, individual objects (Leaf) and groups of objects (Composite) share the same interface and behave as a whole.

Application in Diagram:

Abstract Structure ProductComponent

ProductComponent is an abstract class that represents both individual products (e.g. a headset) and groups of products (e.g. a phone and charger in a box).

The following methods are defined in it:

Add(ProductComponent component) - To add subunits.

Remove(ProductComponent component) - To remove subunits.

Display(int depth) - To display the hierarchical structure of subunits.

Leaf Classes (Leaf):

SmartPhone, Headphones, Charger: Each derives from the Product class and represents simple products. These classes do not contain any child elements.

Composite Classes (Composite):

ProductGroup holds both product groups and individual products together.

For example: A Phonebox class can contain a phone, headphones and a charger.

It maintains a list called children and stores all its children in this list.

Special classes: Phonebox, Mainbox.

Relationships:

The ProductGroup and Product classes extend the ProductComponent abstract class.

ProductGroup can have other ProductComponent types under it, creating a hierarchical structure.

Example: When a Phonebox contains both a SmartPhone and a Headphones, both objects share behavior through the ProductComponent interface.

Observer Pattern

Objective:

The Observer Pattern allows changes in the state of an object to be automatically notified to other objects that depend on that object. In this UML diagram, it is

Application in Diagram:

Interface: IStockObserver

Defines which behaviors the observers (Observer) will apply:

OnStockLow(string product, int quantity) - Triggered when the stock of a product is low.

Subject (Subject/Observable): StockNotifier

StockNotifier manages multiple observers and notifies observers of status changes.

Properties:

List<IStockObserver> observers - List of registered observers.

AddObserver(IStockObserver observer) - Adds a new observer.

NotifyObservers(string product, int quantity) - Sends a stock alert about the product to all observers.

Observers (Observers):

Orders: Monitors stock changes and has an impact on order processing.

For example, if the stock is low, it can stop related orders or suggest alternative products.

Class Name: Another observer that takes a specific action when stock is low.

The details are not specified in the diagram, but there is a data load method like LoadData().

Associations:

StockNotifier manages the observers that implement the IStockObserver interface.

Example Flow:

When the stock of a product is low (NotifyObservers is called),

The Orders and Class Name classes react to this (for example, notify the user or update the stock).

Factory Method Pattern

Objective:

The Factory Method Pattern is a design pattern used to abstract and subclass the object creation process. This was used in UML to dynamically create different product lines (for example, Samsung or Apple products).

Application in the diagram:

Abstract Factory: ProductGroupFactory

Abstracts the method to create a product group.

Method: CreateProduct(string name, string description, double price, int quantity).

Concrete Factories:

SmartPhoneBoxFactorySamsung: Creates groups of Samsung products (for example, a box containing a Samsung phone and headphones).

SmartPhoneBoxFactoryApple: Creates Apple product groups.

Relationships:

ProductGroupFactory is extended by subclasses.

Each factory produces a specific product group (for example, Phonebox or Mainbox).

Example Flow:

SmartPhoneBoxFactorySamsung is called when the user wants to create a Samsung box.

The CreateProduct method creates a Phonebox and adds SmartPhone, Headphones, etc. inside.

Relationships between Patterns:

Composite and Factory Method:

The Composite Pattern is used to create a dynamic product hierarchy, while the Factory Method determines how that product hierarchy is created.

For example, a SmartPhoneBoxFactorySamsung can create a Phonebox and a SmartPhone and a Charger will be added inside.

Observer and Composite:

StockNotifier keeps track of stock levels and informs observers such as Orders. These observers update the relevant product hierarchies (e.g. ProductGroup).

Observer and Factory:

Once a factory (e.g. SmartPhoneBoxFactorySamsung) creates a product group, the stock levels of these products are monitored by StockNotifier.

Requirements and functions encountered:

Requirements

1. Product Management

Requirement: Manage individual products (e.g. Smartphone, Headsets) and composite products (e.g. Phone Box, Mother Box) uniformly.

Functionality Addressed:

Composite Model:

Enables representation of both single products and groups of products using ProductComponent and its subclasses.

Supports operations such as adding, removing and viewing products or groups.

2-Inventory Grouping

Requirement: Enable hierarchical organization of inventory (for example, grouping products into bins or categories).

Functionality Addressed:

Classes:

Mainbox for top-level inventory containers.

Phonebox for subgroups or bundled products.

Recursive hierarchy via ProductGroup allows to create multi-level structures.

3-Stock Tracking

Requirement: Monitor stock levels and inform relevant parts of the system (e.g. low stock alerts).

Functionality Addressed:

Observer Model:StockNotifier maintains a list of observers (e.g. Orders, Class Name) and notifies them of stock changes.

Observers respond to changes with appropriate actions (e.g. processing new orders or updating the UI)

4-Dynamic Product Creation

Requirement: Provide a flexible mechanism to dynamically create different product types.

Functionality Addressed:

Factory Method Pattern:Factories (e.g., SmartPhoneBoxFactorySamsung, SmartPhoneBoxFactoryApple) encapsulate the creation logic for different product types or groups.

Supports the addition of new product types with minimal changes to the system.

5-Inventory Reporting

Requirement: Create reports to display inventory hierarchy and stock details.

Functionality Addressed:

Display(int depth) in ProductComponent:

Allows recursive traversal of the inventory tree to generate structured reports.

Supports depth-specific visualization for better clarity.

6-Stock Updates and Order Processing

Requirement: Dynamically update stock quantities when orders are processed or new inventory is added.

Functionality Addressed:

Orders Class: Manages order processing by updating stock levels and loading order data via UpdateStock.

AddOrder Class: Manages quantities and associates orders with inventory items.

7-User Management

Requirement: Support user registration, login and management of user-related data.

Functionality Addressed:

Registration Class: Handles user details such as name, password, email and phone.

LoginPage Class: Authenticates users for system access.

Users Class: Stores and loads user data for other operations.

8-Inventory Addition

Requirement: Add new inventory items and dynamically associate them with existing structures.

Functionality Addressed:

InventoryAdd Class: Provides a mechanism for integrating new items into existing inventory.

Relies on parent-child relationships (Inventory parent) for association.

9-Scalability and Extensibility

Requirement: To ensure that the system can handle new product types, inventory groups or features without major redesign.

Functionality Addressed:

The Factory Method Pattern allows adding new factories for new product types without changing the existing code.

Composite Pattern supports dynamic reconfiguration of inventory with minimal effort.

Functions Encountered

1. Composite Pattern Functions

-Add(ProductComponent component): Adds a child component (for example, a product or product group).

-Remove(ProductComponent component): Removes a child component from a group.

-Display(int depth): Displays the inventory hierarchy recursively with depth-based indentation.

2. Observer Pattern Functions

-AddObserver(IStockObserver observer): Registers a new observer to the stock observer.

-NotifyObservers(string productName, int quantity): Notifies all registered observers about stock changes.

-CheckStock(string productName, int quantity): Checks stock levels and triggers notifications if below a threshold.

-OnStockLow(string productName, int quantity) (implemented by observers): Handles low stock notifications by taking appropriate actions.

3. Factory Method Mold Functions

-CreateProduct(name, description, price, quantity): Factory method for creating a product, encapsulates the creation logic. Methods overridden in subclasses (e.g. SmartPhoneBoxFactorySamsung,

-SmartPhoneBoxFactoryApple): Customize the product creation process for specific product types or manufacturers.

4. Inventory and Order Management Functions

-UpdateStock(string barcode, int quantity) in Orders: Updates the stock level of an item based on an order.

-InitializeComponent(): Initializes components of various classes (e.g. Users, Inventory, Products).

-LoadData(): Loads data for a specific component (e.g. user data, product inventory).

-getQuantities() in AddOrder: Gets the quantities of items associated with an order.

5.User Management Functions

-LoginPage(): Performs login operations by verifying user credentials.

-Register(): Captures user details such as name, password, email and phone for registration.