Inventory Management System

Name :

Mahmoud Ahmed Mahmoud

Hesham Osama Abdelmonem

Hager Helal Nafadi

Haidy Gerges Nakla

Febronia Attia Azmy

Overview

This document presents a comprehensive overview of the design patterns, their application, and the Graphical User Interface (GUI) utilized in the **Inventory Management System**. The system has been designed with a focus on flexibility, scalability, and maintainability, integrating several essential design patterns to achieve these objectives. The key design patterns used in the development of the system include:

1. **Singleton Pattern**
2. **Factory Pattern**
3. **Command Pattern**
4. **Observer Pattern**
5. **Decorator Pattern**
6. **Strategy Pattern**

Additionally, the system features a user-friendly and efficient GUI, enabling users to easily interact with and manage the inventory, while seamlessly integrating with the backend functionality. The GUI's design emphasizes usability and responsiveness to enhance the overall user experience.

---

1. Singleton Design Pattern

Purpose:

The **Singleton Pattern** ensures that a class has only one instance and provides a global point of access to that instance. This pattern is particularly useful when it is important to control access to shared resources, such as configuration settings or database connections.

Implementation:

In the Inventory Management System, the **InventoryManagementApp** class uses the Singleton pattern to ensure that only a single instance of

the application is created. This approach guarantees that the system maintains consistent state throughout its execution. By providing a static method to access the single instance, we ensure that no additional instances of the class are created.

```
public class InventoryManagementApp {
    private static InventoryManagementApp instance;

    private InventoryManagementApp() {
        // Private constructor to prevent instantiation.
    }

    public static InventoryManagementApp getInstance() {
        if (instance == null) {
            instance = new InventoryManagementApp();
        }
        return instance;
    }
}
```

Benefits:

- **Global Access**: Ensures that the application has a single, globally accessible instance.
- **Controlled Access**: Limits the creation of the instance to a single point, providing centralized control over system state.
- **Memory Efficiency**: Reduces memory overhead by ensuring that only one instance exists.

---

2. Factory Design Pattern

Purpose:

The **Factory Pattern** provides an interface for creating objects, but allows subclasses to alter the type of objects that will be created. This pattern promotes loose coupling between the client code and the classes that instantiate objects, thus simplifying future modifications and extensions.

Implementation:

In this system, the **ProductFactory** class is employed to create different types of product objects (such as **ElectronicProduct**, **FurnitureProduct**, and **GroceryProduct**) based on user input or business logic. The **Factory Pattern** allows new product types to be introduced without modifying the client code, ensuring extensibility.

```
public class ProductFactory {
    public static Product getProduct(String category) {
        if (category == null) {
            return null;
        }

        if (category.equalsIgnoreCase("Electronic")) {
            return new ElectronicProduct();
        } else if (category.equalsIgnoreCase("Furniture")) {
            return new FurnitureProduct();
        } else if (category.equalsIgnoreCase("Grocery")) {
            return new GroceryProduct();
        }

        return null;
    }
}
```

Benefits:

- **Loose Coupling**: The client code is decoupled from the specific product classes, which makes it easier to modify or extend the system.
- **Ease of Extension**: New product categories can be added without changing the structure of the existing code.

---

3. Command Design Pattern

Purpose:

The **Command Pattern** encapsulates a request as an object, allowing parameterization of clients with queues, requests, and operations. This pattern decouples the sender of a request from the object that performs the action, which increases flexibility in the system's execution.

Implementation:

In this system, the **Command Pattern** is used to represent actions such as adding, updating, and deleting products in the inventory. Each of these actions is encapsulated as a command object that can be executed by the system. The command object acts as a mediator between the user interface and the backend logic.

```
public interface Command {
    void execute();
}

public class AddProductCommand implements Command {
    private Inventory inventory;
    private Product product;

    public AddProductCommand(Inventory inventory, Product product) {
        this.inventory = inventory;
        this.product = product;
    }

    @Override
    public void execute() {
        inventory.addProduct(product);
    }
}
```

Benefits:

- **Separation of Concerns**: The sender of the request (e.g., the user interface) is decoupled from the object that handles the request (e.g., the inventory).
- **Undo/Redo Capability**: The system can store commands for later execution, allowing for easy undo/redo functionality.
- **Improved Flexibility**: Commands can be executed at any point, enhancing flexibility in handling various actions.

---

4. Observer Design Pattern

Purpose:

The **Observer Pattern** defines a one-to-many dependency between objects. When the state of the subject object changes, all its dependent observer objects are automatically notified and updated. This pattern is useful when you want to update multiple views or models in response to changes in the data.

Implementation:

In the **Inventory Management System**, the **Observer Pattern** is implemented by the **InventoryData** class, which maintains a list of observers (such as the user interface components) and notifies them whenever the inventory is updated. This ensures that all views of the inventory are kept in sync.

```
public class InventoryData {
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}

public interface Observer {
    void update();
}
```

Benefits:

- **Decoupled View and Model**: The observer pattern separates the view from the model, reducing tight coupling between components.
- **Automatic Updates**: Observers are automatically notified of changes, reducing the need for manual updates.

- **Scalability**: New observers can be added with minimal modification to existing code.

---

5. Decorator Design Pattern

Purpose:

The **Decorator Pattern** allows new functionality to be added to an object without altering its structure. This pattern is particularly useful when you want to dynamically extend the behavior of an object in a flexible and reusable manner.

Implementation:

In the system, the **Decorator Pattern** is employed to add additional features to products, such as applying discounts or adding custom pricing strategies. The behavior is extended without modifying the underlying product class.

```
public class DiscountDecorator extends ProductDecorator {
    private double discountPercentage;

    public DiscountDecorator(Product product, double discountPercentage) {
        super(product);
        this.discountPercentage = discountPercentage;
    }

    @Override
    public void create() {
        super.create();
        System.out.println("Applying discount of " + discountPercentage +
"%");
    }
}
```

Benefits:

- **Flexible Extension**: New behaviors can be added to objects at runtime, offering flexibility in the system.
- **Reusability**: Decorators can be applied to different objects without requiring changes to the core logic.

---

6. Strategy Design Pattern

Purpose:

The **Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. In this system, the Strategy Pattern is utilized to handle various table models, allowing for dynamic creation of tables based on different data types (such as products, suppliers, users, and orders) at runtime. This provides flexibility, as new table models can be easily introduced without affecting the rest of the system.

Implementation:

In this implementation, we define the **TableModelStrategy** interface, which mandates the creation of table models. Different table models for various entities (such as products, suppliers, users, and orders) are encapsulated in their respective classes, such as **ProductTableModelStrategy**, **SupplierTableModelStrategy**, **UserTableModelStrategy**, and **OrderTableModelStrategy**. These strategy classes implement the **TableModelStrategy** interface and return the appropriate **DefaultTableModel**.

```
// Strategy Pattern to handle various table models
interface TableModelStrategy {
    DefaultTableModel createTableModel();
}

class ProductTableModelStrategy implements TableModelStrategy {
    @Override
    public DefaultTableModel createTableModel() {
        return new DefaultTableModel(new Object[][]{}, new String[]{"ID",
"Name", "Category", "Quantity", "Price"});
    }
}

class SupplierTableModelStrategy implements TableModelStrategy {
    @Override
    public DefaultTableModel createTableModel() {
```

```
        return new DefaultTableModel(new Object[][]{}, new String[]{"ID",
"Name", "Type", "Contact", "Phone", "Email"});
    }
}

class UserTableModelStrategy implements TableModelStrategy {
    @Override
    public DefaultTableModel createTableModel() {
        return new DefaultTableModel(new Object[][]{}, new String[]{"ID",
"Username", "Role"});
    }
}

class OrderTableModelStrategy implements TableModelStrategy {
    @Override
    public DefaultTableModel createTableModel() {
        return new DefaultTableModel(new Object[][]{}, new String[]{"Order
ID", "Product ID", "Quantity", "Total Price"});
    }
}
```

Benefits:

- **Dynamic Table Generation**: Depending on the data being displayed, the appropriate table model is created dynamically, ensuring that different views (e.g., product, supplier, user, order tables) are easily managed.
- **Extensibility**: New table models can be introduced by simply adding a new strategy class without modifying existing code.
- **Flexibility**: The client (user interface or controller) can switch between different table models at runtime, based on the context (e.g., displaying product data versus supplier data).

Graphical User Interface (GUI) Implementation

Purpose:

The **GUI** provides an interface through which users interact with the **Inventory Management System**. The primary goal is to make the system as intuitive and user-friendly as possible, allowing users to efficiently manage products, perform CRUD operations, and view inventory details in real-time.

Some of GUI Components:

1. **Main Screen**: Displays a list of products in the inventory and provides buttons for adding, editing, or removing products.
2. **Product Management Screen**: Allows users to add, update, or delete products in the inventory.

Technology Stack:

- **JavaFX**: JavaFX is utilized to design and implement the GUI, which provides rich features for building modern user interfaces. Key components such as buttons, tables, and text fields are created using JavaFX.
- **Event Handling**: Event listeners capture user actions such as button clicks, and these events are processed by invoking command objects to carry out the requested operations.

Benefits:

- **Intuitive User Interaction**: The GUI provides an easy-to-navigate interface for users to manage inventory operations.
- **Responsive Design**: The system responds immediately to user actions, providing feedback and updating inventory data in real time.
- **Seamless Integration**: The GUI is tightly integrated with the backend logic, ensuring that user actions are reflected in the system's operations.

Conclusion

This documentation describes the **Inventory Management System**'s architecture and the design patterns used to achieve an efficient, scalable, and maintainable solution. By employing the **Singleton**, **Factory**, **Command**, **Observer**, **Decorator**, and **Strategy** patterns, the system maintains flexibility and adaptability. The **GUI** offers a seamless

user experience, allowing easy interaction with the underlying inventory management functionality.

Key Takeaways:

- **Singleton Pattern** ensures a single, globally accessible instance.
- **Factory Pattern** decouples object creation from the client code.
- **Command Pattern** enables separation of request and execution logic.
- **Observer Pattern** ensures that all dependent components are notified of changes.
- **Decorator Pattern** extends object behavior dynamically without altering their classes.
- **Strategy Pattern** allows for flexible algorithm selection at runtime.
- **GUI** provides an intuitive interface for users to interact with the system.

This combination of design patterns and GUI design ensures that the system is both robust and easy to extend, making it well-suited for managing inventory efficiently.