Project Group Number/Name: Group Number 8
Group Members:

- Christ Rodrigues
- Yash Zaveri
- Viral Suchak
- Kevin Rodrigues
- Omkar Vaity

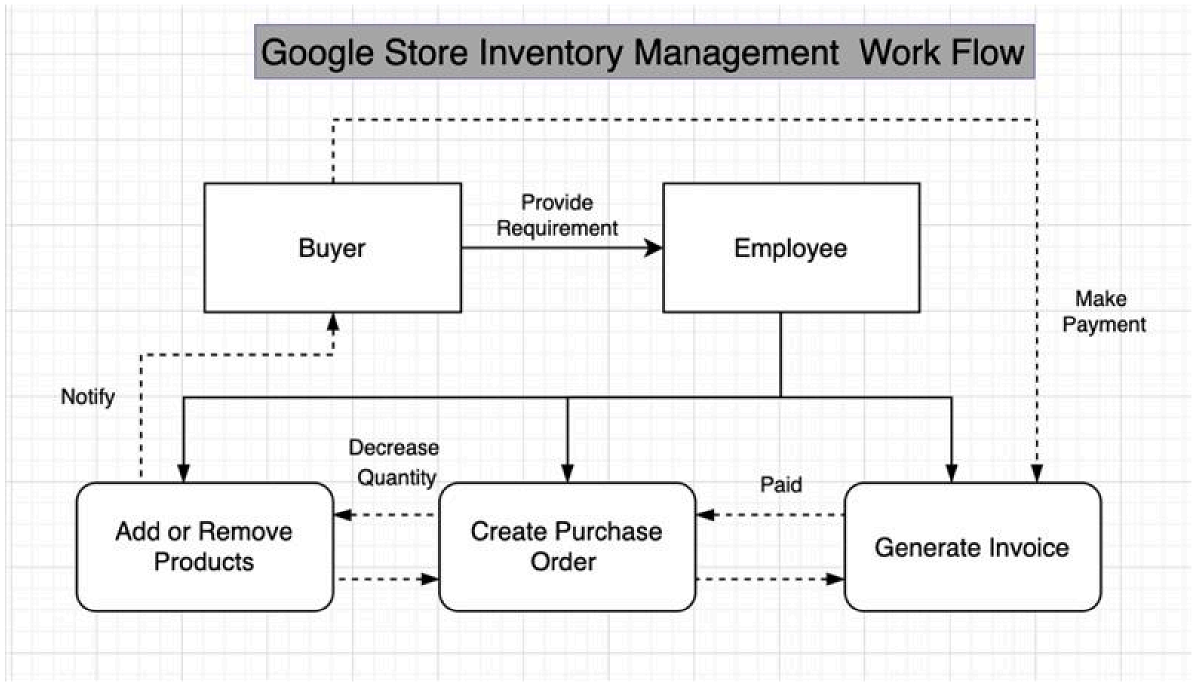
Project Topic/Name: Google Store Inventory Management System

Problem Statement:
When inventory isn't managed well, it can lead to items going out of stock, excess products piling up, and delays in getting orders to customers—all of which can be frustrating for both the business and its shoppers. To keep things running smoothly, a smarter system is needed to balance stock levels, cut down on waste, and make the supply chain more efficient. By using automated tracking, predictive analytics, and real-time monitoring, inventory management can become more accurate and hassle-free. This project aims to create a reliable system that keeps products available when needed while keeping costs under control.

UML/ER Diagram:

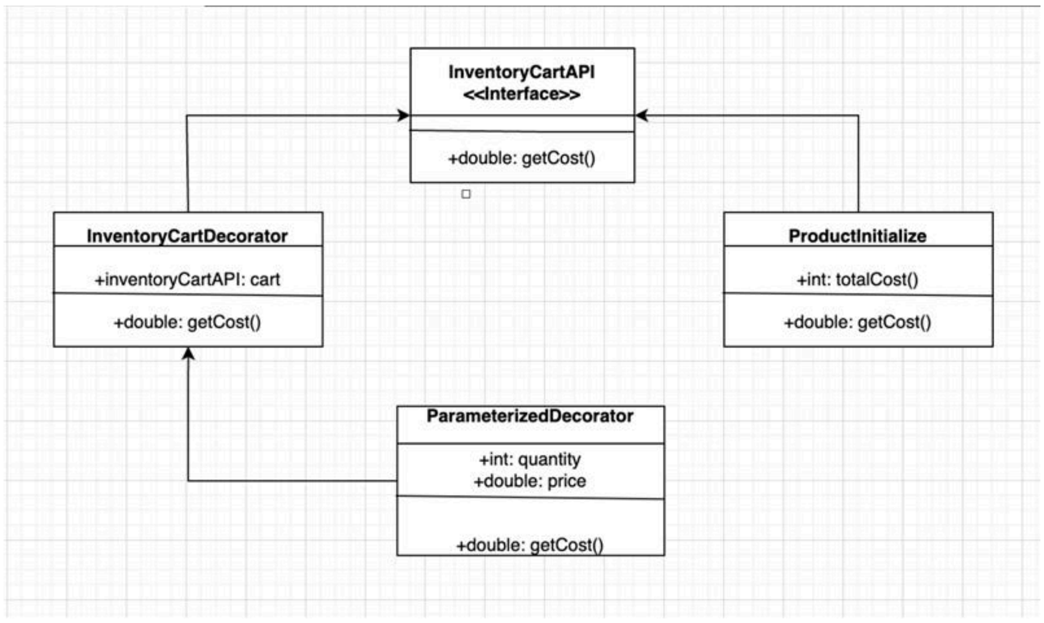
1. **High-Level Workflow Diagram:**

The following diagram illustrates the high-level workflow of the Google Store Inventory Management System, depicting interactions between key entities such as Buyer, Employee, and Inventory-related operations.



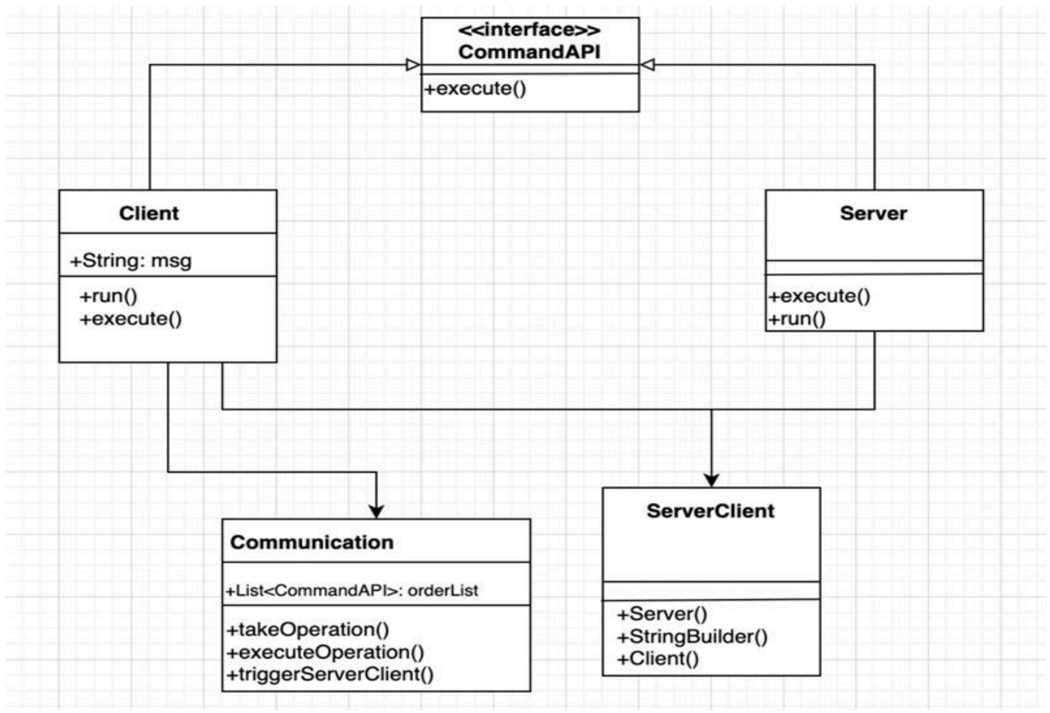
2. **Decorator Pattern UML:**

   This UML diagram demonstrates the Decorator Pattern implementation used in the system for adding products during the creation of a new purchase order.



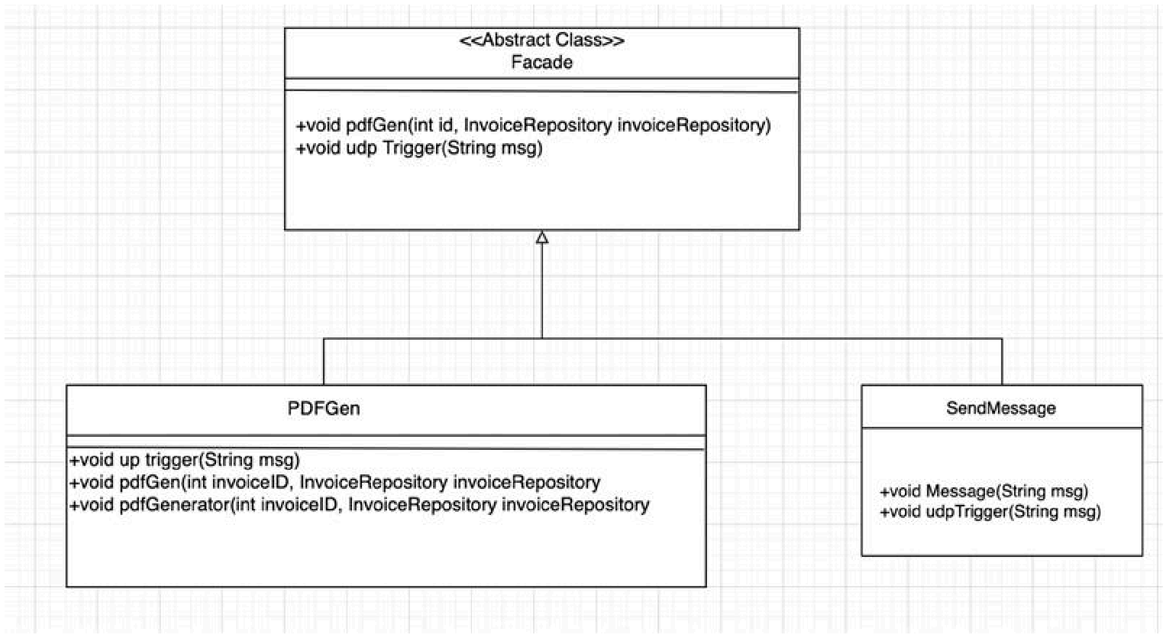
3. **Command Pattern UML:**

   This UML diagram shows how the Command Pattern will be used to facilitate communication between clients and servers. Specifically, the server will send an alert when inventory quantity is low.



4. **Facade Pattern UML:**

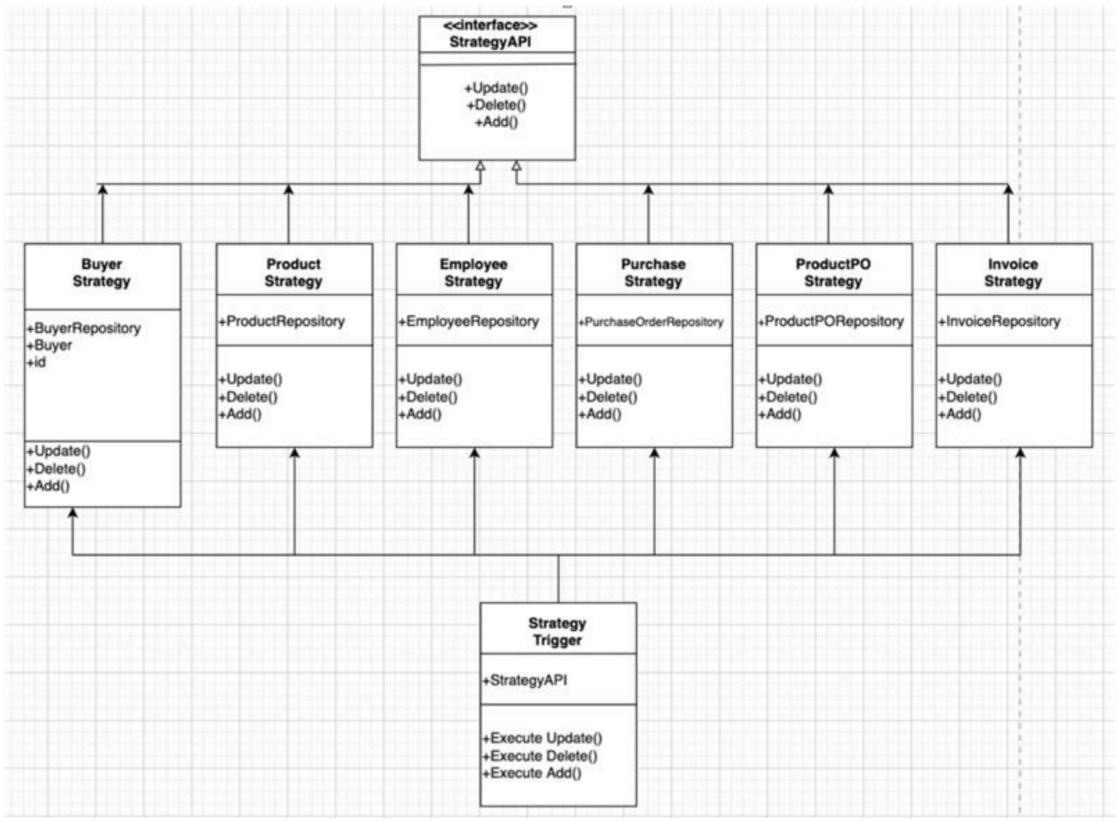
This UML diagram illustrates the Façade Pattern, which will be used to simplify the process of generating a PDF invoice once the payment transaction is completed.



<<Abstract Class>>
Facade
+void pdfGen(int id, InvoiceRepository invoiceRepository)
+void udp Trigger(String msg)
PDFGen
+void up trigger(String msg)
+void pdfGen(int invoiceID, InvoiceRepository invoiceRepository
+void pdfGenerator(int invoiceID, InvoiceRepository invoiceRepository
SendMessage
+void Message(String msg)
+void udpTrigger(String msg)

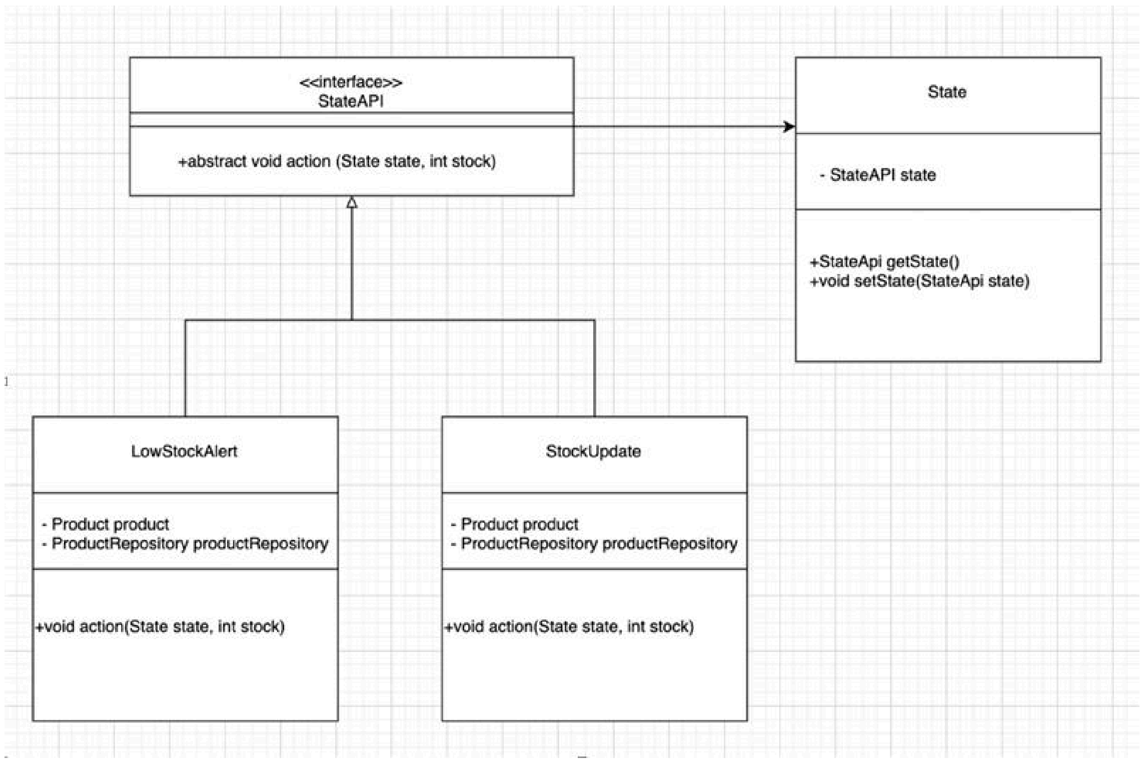
5. **Strategy Pattern UML:**

We will use the Strategy Pattern for carrying out CRUD operations on all entities.



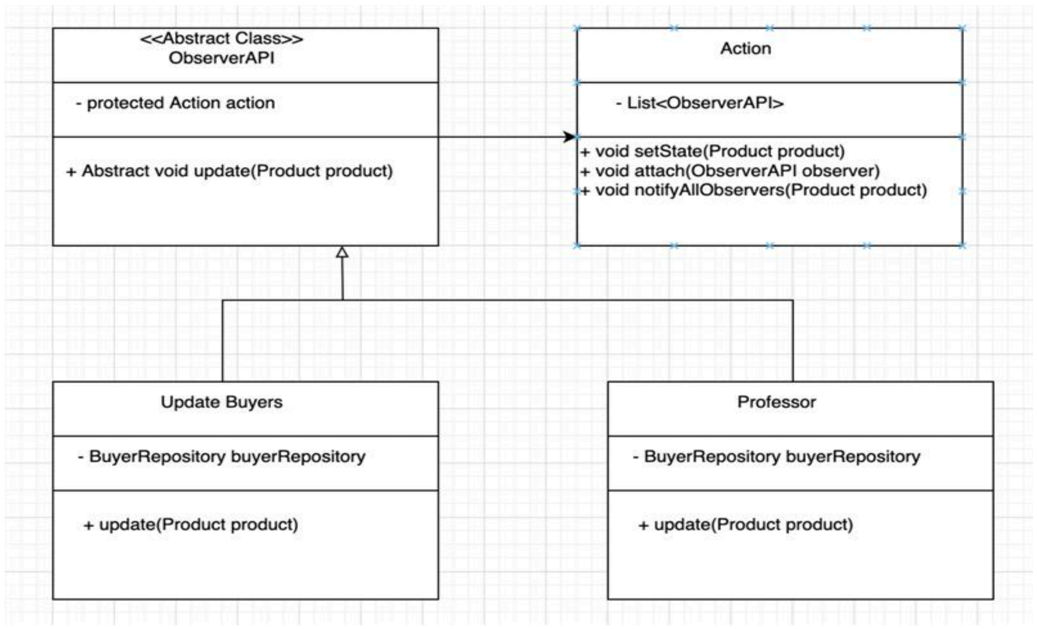
6. **State Pattern UML:**

We will use the State Pattern in a way that it allows an object to alter its behavior when its internal state changes by delegating the behavior to separate state classes.


<<interface>>
StateAPI
+abstract void action (State state, int stock)
State
- StateAPI state
+StateApi getState()
+void setState(StateApi state)
LowStockAlert
- Product product
- ProductRepository productRepository
+void action(State state, int stock)
StockUpdate
- Product product
- ProductRepository productRepository
+void action(State state, int stock)

7. **Observer Pattern UML:**

Through using the UML Observer pattern, we will send a Notification of new products to all the buyers whenever we add the products to the inventory.


<<Abstract Class>>
ObserverAPI
- protected Action action
+ Abstract void update(Product product)
Action
- List<ObserverAPI>
+ void setState(Product product)
+ void attach(ObserverAPI observer)
+ void notifyAllObservers(Product product)
Update Buyers
- BuyerRepository buyerRepository
+ update(Product product)
Professor
- BuyerRepository buyerRepository
+ update(Product product)

Design Patterns to be Implemented:

- **Singleton Pattern** – Used to ensure that certain objects are instantiated only once.
- **Decorator Pattern** – It is a Structural design pattern that allows behavior to be added to an individual object, without affecting the behavior of other objects.
- **State Pattern** – It allows an object to alter its behavior when its internal state changes.
- **Facade Pattern** – Makes it easier to use and reduces the complexity of interactions with the subsystem.
- **Abstract Factory Pattern** – Used to provide an interface for creating families of related objects without specifying their concrete classes.
- **Strategy Pattern** – Used when a family of algorithms has to share a common interface.
- **Observer Pattern** – Defines a one-to-many dependency between objects so that when one object changes state, all its observers are notified and updated automatically.
- **Command Pattern** – It turns the requests into a standalone object.

Tech Stack:

- **Frontend:** React and Bootstrap
- **Backend:** Java EE, Spring, Apache Tomcat
- **Database:** MySQL
- **Database Connectivity:** Hibernate

Functionalities to be Implemented by the End of Milestone 2:

- **Navigation Structure:** Users will be able to navigate between different products.
- **Product Addition:** Users will be able to add product details.
- **Data Integration and Fetching:** Product information and data will be structured properly and displayed to the users.
- **Product Categorization and Information:** Products will be categorized and displayed along with their quantities and prices.

Contributions:

- **Christ Rodrigues:** Project setup and problem statement
- **Yash Zaveri:** UML diagram / ER diagram
- **Viral Suchak:** UML diagram / ER diagram
- **Kevin Rodrigues:** Design patterns implementation, functionalities
- **Omkar Vaity:** Tech stack and overall documentation