**Clothing Store Point of Sale Software System**

Javier Garcia Ramirez

Matthew Humphrey

The Clothing Store Point of Sale Software System (CSPSSS for short) is a software system intended for use by employees. Employees have the ability to organize and update inventory, as well as the ability to handle transactions such as purchases and returns.The system will be safe and efficient.

Architectural Diagram:

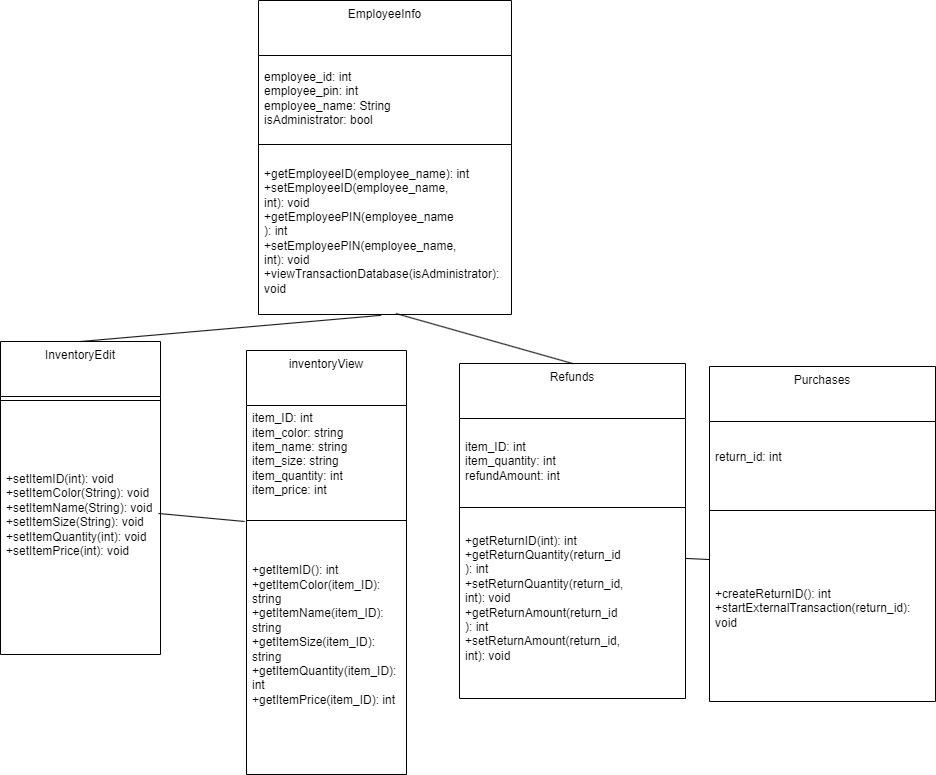
A diagram of a data flow

Description automatically generated

SWA Diagram description:

The diagram begins at the main intended user: The employee. The employee first enters the login component, which interacts with an external login server. This gives access to the two main fields, sales and inventory interaction. Employees can handle refunds, which directly writes to a transaction database, or they handle purchases, which writes to the external credit card/debit card processor, and then to the transaction database. The transaction database can only be read by the Administrator. Sales and refunds will then interact with the inventory database to update the quantity of products and such. The inventory field is the other main component. Logging in gives access to the inventory database. Employees can scan barcodes to retrieve item ID’s, from which they can either edit the inventory or view available stock. Employees are also able to manually enter item IDs.

UML Diagram:



Class/Method Descriptions:

**EmployeeInfo**: This class interfaces with the external login server, using the employee\_id and employee\_pin credentials. The external server then updates the status of isAdministrator, to allow the user access to the transaction database (if they’re an admin).

Administrator methods:

getEmployeeID(): retrieves the employee\_id stored on the logon server. setEmployeeID(): creates an employee\_id that will be stored on the logon server. getEmployeePIN(): retrieves the employee\_pin stored on the logon server. setEmployeePIN(): creates the employee\_pin that will be used to login. viewTransactionDatabase(): checks if the user is an administrator, and if they are, returns a copy of the transaction database.

**InventoryEdit**: The purpose of this class is to change attributes of items in the inventory, such as the item ID, color, name, price, size and quantity. The inventoryEdit serves as a subclass to the InventoryView class.

Employee Methods:

setItemID(): Creates/changes the ID of an item setItemColor(): Creates/Changes color of an item setItemName(): Creates/Changes name of an item setItemSize(): Creates/Changes specific size of item setItemQuantity(): Creates/Changes the quantity of an item setItemPrice(): Creates/changes the price of an item

**InventoryView**: The purpose of this class is to look up items and retrieve their corresponding attributes. This class contains the variables for ID, Color, name, size, quantity and price. This class serves as the parent class to InventoryEdit.

Employee Methods:

getItemID(): When the barcode is scanned, getItemID() runs, and returns the ID of a specific item.

getItemColor(): returns color of a specific item, given an item\_ID. getItemName(): returns the name of a specific item, given an item\_ID. getItemSize(): returns the size of a specific item, given an item\_ID. getItemQuantity(): returns the quantity of a specific item, given an item\_ID.

getItemPrice(): returns the price of a specific item, given an item\_ID.

**Refunds**: This class is responsible for handling returns, and also interacting with the external payment processor for purchases. Each return is created as an object that has a unique return\_id, which allows for employees to edit information about the return.

Employee methods:

getReturnID(): the return\_ID is retrieved from being manually entered in getReturnQuantity(): Retrieves the number of items being returned, given a return\_id setReturnQuantity(): Sets how many items are being returned, given a return\_id getReturnAmount(): Retrieves the cost refunded, given a return\_id setReturnAmount(): Sets the cost to be refunded given a return\_id

**Purchases**: This class is for handling the purchases in the store. Every time a purchase is made, a return\_id is created, in case a refund needs to happen. The Refunds class can then reference this number for its own purposes.

Employee methods:

createReturnID(): This creates a return\_id variable in order for the Refunds class to reference it.

startExternalTransaction(): This method takes return\_id in order to get information about the items in the purchase, and sends this information to an external payment processor. The validation of credit/debit cards and the rest of the payment procedure is not handled by this method, but by the external service.

Development Plan/Timeline:

Initial Planning (Weeks 1-2):

● Communicate with store staff to set expectations

● Identify what methods will take the longest to create

● Divide team into groups

Database Relations (Weeks 3-5):

● Ensure the integrity of external and internal databases

● Create a group to maintain the internal databases

Class Creation/Implementation (Weeks 6-10):

● Create the EmployeeInfo class, and ensure security.

● Test and implement the InventoryView, InventoryEdit, Purchases, and Refunds classes

Method Testing/Implementation (Weeks 11-15):

● Perform tests for individual methods to verify functionality.

● Make sure inter-class operations perform as expected. UI/App Creation (Weeks 16-20):

● Design an employee-centric interface

● Make input fields, connected menus, and graphical elements to make the software simple to use

System Testing (Weeks 21-23):

● Integrate the existing classes and logic with the new user interface.

● Create test cases for the system that include not only common workflows, but edge cases as well.

Documentation (Week 24):

● Write material that can be used by the store to train their employees

Maintenance (Ongoing):

● Communicate with the store as needed, in order to address bugs or workflow improvements.

System Test Plan

**Test Set 1: EmployeeInfo**

The unit testing will consist of trying the getEmployeeID and setEmployeeID functions. If the system works correctly, setEmployeeID should correctly change the value of employee\_id, and the getEmployeeID should return the intended employee\_id.

The Functional/Integration testing will consist of using the viewTransactionDatabase administrator function. Should the system work correctly, the method will allow only the administrator to view the Transaction database. If the method works without being an administrator, the system is integrated fine but unit testing should be reintroduced to ensure the method only works with the administrator.

The system test will consist of using EmployeeInfo to successfully set up a test employee, and then using functions from InventoryEdit, InventoryView, Refunds and Purchases to ensure that the employee has access to all the necessary classes.

**Test Set 2: InventoryView**

Examples of unit tests for the InventoryView class would be those that verify the functionality of the methods that belong only to InventoryView.

For example, you could test getItemID(), and make sure it returns the correct item\_ID. You could also test getItemColor(), and verify that it returns the correct color.

Examples of integration tests for the InventoryView class could be ones that use methods from InventoryView and InventoryEdit. To test integration between these classes, we should be able to verify that changes made by InventoryEdit will be reflected in the data retrieved by InventoryView’s methods.

This can include using methods like setItemSize and then getItemSize, or setItemQuantity and then getItemQuantity. Using these methods in succession will allow us to see if data integrity is maintained between the two classes.

System tests have the largest scope out of our 3 granularities. These tests focus on using the entire system, and considering the interaction between multiple classes.

One system test that we could perform would be to create our inventory system with multiple items and verify that InventoryView, along with InventoryEdit, can be accessed by an employee to retrieve information for all items.

Data Management Strategy:

Within our system, we have three main databases: our login database, our inventory database, and our transaction database. Our strategy will involve a SQL approach rather than a NoSQL approach. This is because our databases require many different fields and details, and SQL handles big data in an easy-to-follow and efficient way.

For our login server/database, we will use MySQL, and will divide the data into two tables. One table will be the Users table, which will store user information like passwords, system roles, and usernames. The second table will be the Session table, which will store information about the user’s current login session, like when their login expires, and to track data like IP addresses and what device they are using.

For our inventory database, we will also use MySQL, since it is both highly efficient and also easily scalable. These are necessary traits for an inventory management system. We will split up the system into three main tables: the Product table, the Inventory table, and the Category table. The Product table will contain information such as product SKU, name, price, and a description. Our Inventory table will track things like product quantity and will be updated by our external purchasing methods. Lastly, our Category table will be used to group large amounts of items in our Product table into manageable categories.

Our transaction database will also utilize MySQL. This database will utilize two tables, a dedicated Purchases table and a dedicated Refunds table. The Purchase table will store the purchase amount, itemID, itemQuantity, and customer payment information. The Refund table will store the returnID, the returnQuantity, returnAmount, as well as the itemID. It is important to note that these tables will only be accessible through an administrator account. Expected data types for these fields will mostly be integers and floats, as most of the data involves IDs and quantitative information. We expect to have numerous records, as each record will represent a new purchase or refund.

We chose these three databases because it strikes a balance between consolidating for efficiency and splitting work across multiple instances. Since we have multiple classes interfacing with our inventory, having that as one database allows us to maintain integrity across only one database, rather than update multiple systems. However, for our login and transaction database, we had to make those separate as they contain sensitive information that can only be accessed by administrators.

As far as alternative approaches are concerned, we could have used different SQL protocols, such as MongoDB for the login server (a noSQL language that is easy to set up for system administrators and is very fast), or Oracle Database for the transaction database (which would provide third-party cloud backup). Within the system that we have set up, we could have also arranged our tables differently. For example, the inventory system maybe wouldn’t need a Category table, and instead we could store categories as information within the Product table. There are several reasons why we chose not to implement these changes, though. For our SQL languages, we determined that the consistency of using one sublanguage is more important than finding disparate, though more optimized sublanguages. For our inventory system, we thought that having 2 granularities, at the category and product level, would be important to workers operating a storefront.

A diagram of a data flow

Description automatically generated

When designing our initial Software Architecture Diagram, we foresaw the need for proper database management and allocation. This is because our system deals with a lot of data, as Point-Of-Sale requires tracking of inventory as well as transactions, which necessitates databases for these areas. We also realized the need for a login server database, as each individual employee has a unique login. We recognized the importance of the flow of access and management of the databases, so we made sure to indicate how these databases will be accessed. First, the Login Server database is strictly for entering the software system. Next, for the transaction database, it was imperative that it can be written to from Sales and Refunds, with access only for the administrator. Lastly, the inventory Database can be written to from sales as well as accessed by employees to either view or edit. For these reasons, we did not see a need to change the diagram, as we already made these implementations from the start.