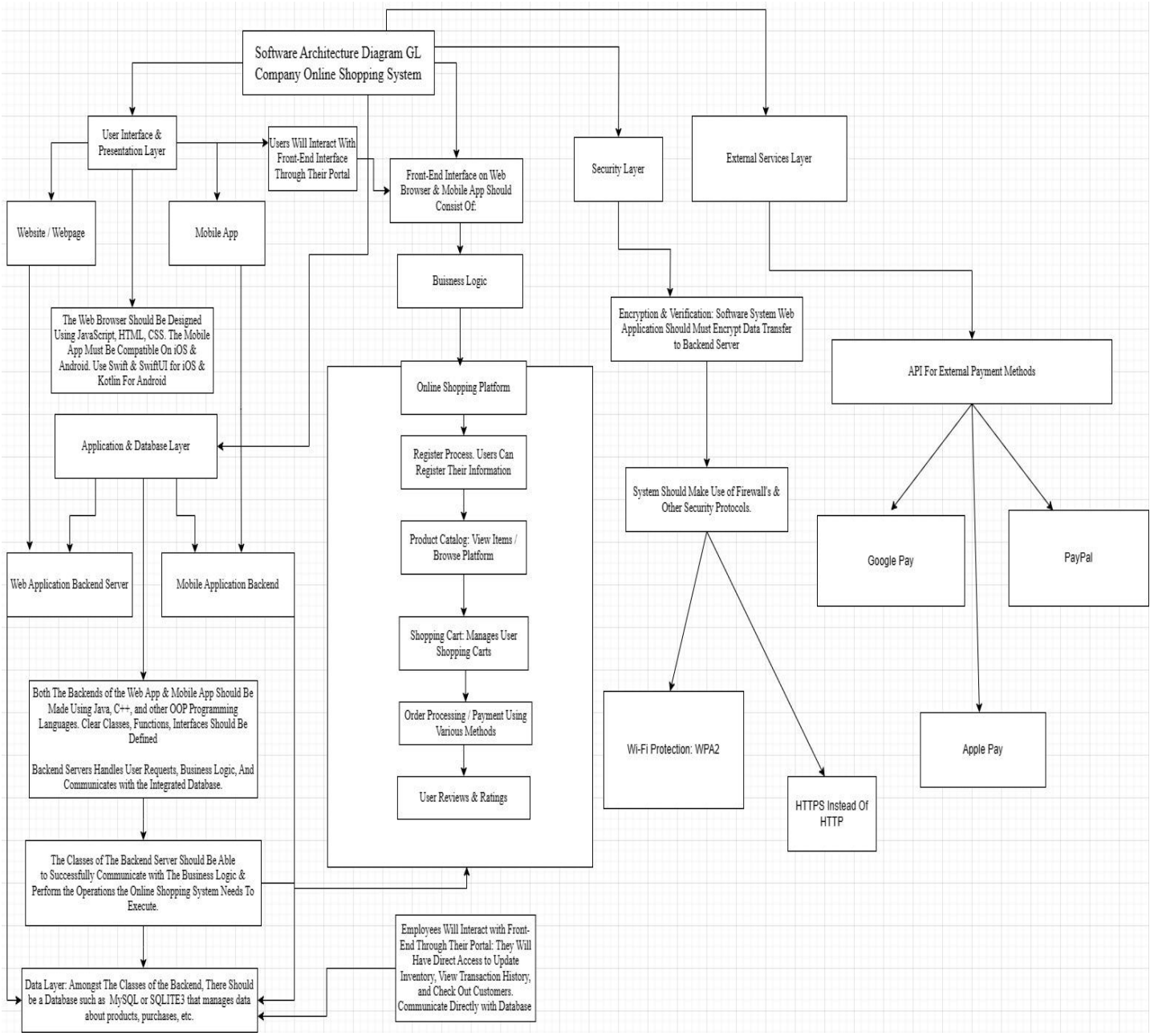
SDS: Online Shopping System: Test Plan

Aman Ambastha, Kyle Cristobal, Samantha Velazquez

Software Design Specification

Link : Architectural Diagram



Description Of Architectural Components / Attributes:

1. **User Interface & Presentation Layer**: The front-end interface where users will communicate with the online shopping system will be composed of a webpage / website

and a Mobile App that will be compatible with both Android & iOS Operating Systems. The user interface should be visually appealing to its customers. The software development and engineering team should use programming languages such as HTML, CSS, and JavaScript for the web page user interface. For the mobile application, the software developers should use Swift programming language for the app on iOS devices while using Kotlin for Android based devices.

2. **Business Logic Layer:** The users of the software system should be able to do the following activities on the web application and the mobile app. Users should be able to make accounts by registering themselves with GL Company, browse the inventory, add items to their shopping cart, make purchases, and also leave reviews and ratings about their experience on the website/mobile app. The front-end and back-end of the software system should collectively be able to support these actions by the user.

3. **Application Backend Layer:** The backends of both the website and mobile app must be programmed using OOP Programming principles and OOP languages such as Java, C++, Python, and other programming languages. Clear classes, functions, and interfaces should be accurately described and explained. The purpose of the classes, interfaces, and function is to handle the business logic, user requests, and communicate with the integrated database. Several classes make up the backend server. More details about the classes and interfaces are provided in the UML Diagram in the next section and the description of classes and operations.

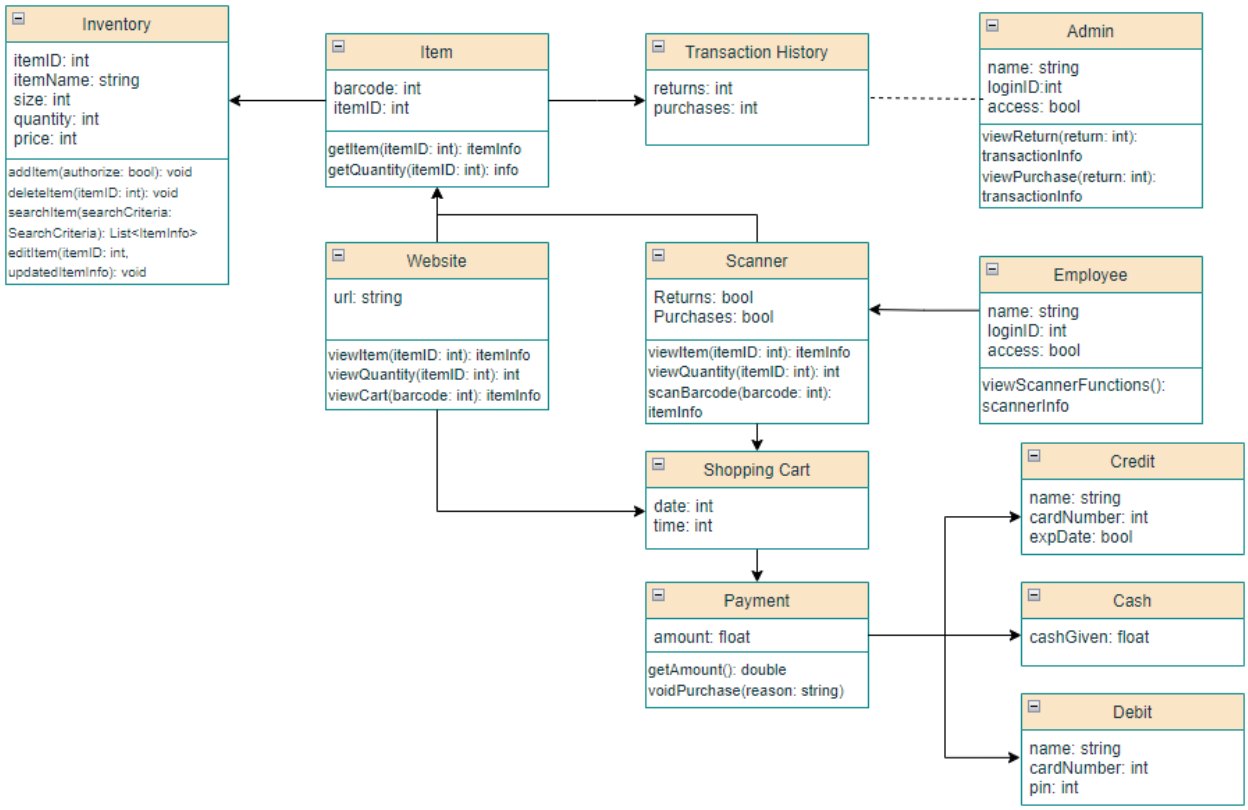
4. **Database Layer:** A database such as MySQL or SQLITE3 should be used and integrated within the backend server and the classes. The database is responsible for keeping track of information relating to products, transaction history, and registered users. For employees, they can login through the user interface to their portal and be able to access, change, and update information stored in the database. Employees can directly have communication access to the database. Along with the database later, a extra caching layer could be added that helps with accessing frequently used data quicker to improve optimization and performance.

5. **Security Layer:** The software system should have enhanced security measures and protocols in place to protect data being transferred from the backend to the frontend ans vice versa. Having data encryption and verification is very important for the software system to implement. Examples of secure measures include using HTTPS instead of HTTP to establish a secure connection between the web server and the website. Other examples include using firewalls and using WPA2 Wifi Access. Ensure that the website is only accessible to people connected to safe and secure internet connections.

6. **External Services Layer:** The above diagram explains the most important architectural components of the software system, a possible extension of the current architectural setup is using advanced API to use external payment gateways and methods via third-party services. Currently, payment is done through the backend server with the getAmount() and voidPurchase() methods. External third-party services include integrating PayPal, Google Pay, and Apple Pay that could possibly provide users with more payment options on how to pay for their items.

UML Class Diagram:

Link: UML Class Diagram



Description of Classes:

Making up our software system is a series of classes that each are unique, yet work with one another to create an efficient system usable for customers and employees.

1. **Inventory:** This class is the container for all the items the store has in stock. Given that the inventory holds numerous items, there are multiple variables responsible for categorizing unique items, for example, itemID (integer), itemName (string), size (integer), quantity (integer), and price (integer). Through this class, employees have a variety of capabilities, such as adding, deleting, editing, updating, and searching items in their inventory using the necessary parameters for each method. Customers too will be using the class by having access to the searchItem method

2. **Item:** This class holds the distinguishing blueprint for item objects, having two variables of integer values called barcode and itemID to efficiently store each item. The values of the variables are accessible with functions getItem and getQuality, which both require itemID in order to function. It's important to note that this class will be used in the Inventory class as Item class objects will be implemented in the inventory. Not to mention Item class objects will also be used in the Transaction History class.
3. **Transaction History:** Through this class, the stores' transaction history is viewable explicitly by the admin, which requires logging in via the Admin class. The class itself has two integer variables, returns and purchases. The two variables pertaining to the objects created by the Items class will be used in this class as well, similarly to Items class' implementation in Inventory class.
4. **Admin:** The Admin class is the login class for the administration, therefore requires variables name (string), loginID (integer), and access (boolean). Given that the administrator successfully logged in, there's two possible methods, viewReturn and viewPurchase, both requiring integer value return in order to run.
5. **Website:** The Website class is used throughout the software, its used in the Item, Scanner, and Shopping Cart class in order for all of them to function. In terms of the class itself it has only one string variable named url. The class has three methods, viewItem and viewQuantity which both take itemID as a parameter and viewCart which requires the barcode parameter.
6. **Scanner:** The Scanner class will be strictly for employee use, therefore only being accessible through the employees portal and not the customers portal. With two boolean variables named returns and purchases, employees can use three methods, viewItem and viewQuantity take in itemID as input, and scanBarcode takes in the barcode.
7. **Employee:** The Employee class is the class responsible for the login for employees. Hence, the class has variables name (string), loginID (integer), and access (boolean). With the class there's only one available method, viewScannerFunctions which redirects employees to the Scanners functionalities.
8. **Shopping Cart:** The Shopping Cart class uses two other classes, Scanner and Website in order to function. For the class itself there's two integer variables, date and time.
9. **Payment:** Following the hierarchy of the software, the ShoppingCart class is used in the Payment class to continue the process of the transaction. The class itself has only one float variable named amount, with its corresponding getAmount method. Along with getAmount, there's also the voidPurchase method that requires a string variable named reason. The below classes, Credit, Cash, and Debit all require the Payment class in order to function.
10. **Credit:** The Credit class is responsible for handling transactions made via credit card, therefore requiring the three variables, name (string), cardNumber (integer), and expDate (bool).

11. **Cash:** The Cash class handles the transactions made with cash in person, hence only requiring the singular float variable cashGiven.
12. **Debit:** Similar to the Credit class, the Debit class handles all transactions made via debit card. With that being said it also has three variables, name (string), cardNumber (integer), and pin (integer).

Description Of Operations:

In our point of sale system many of our functions, and in turn our operations, pertain to viewing, adding/deleting and editing items for a store. The following breaks down the operations in the most notable classes:

1. **Inventory**: This class has four operations we felt were notable for the class. The ‘addItem’ operation represents the action of adding an item to the point of sale system. It takes as a parameter an object representing the item to be added, which includes that object's details such as its name, price, size and quantity. The ‘deleteItem’ operation represents the action of removing an item from inventory. For the ‘searchItem’ operation, it is used for searching and retrieving information within the point of sale system. This operation uses variables ‘itemID’ and ‘barcode’ as a means of identifying an item. Operation ‘editItem’ allows for the modification of item details, more specifically mentioned before; name, price, size and quantity. The changes made through these operations of the Inventory class should be visible in the integrated database.

2. **Item**: Our item class has two operations, both being get operations. This class represents the item and the information that is used to identify an item. ‘getItem’ and ‘getQuantity’ gathers the information held within the inventory class to utilize in other classes and operations such as viewing the items.

3. **Website and Scanner**: These classes utilize similar operations almost all being view operations. Operations ‘viewItem’, ‘viewQuantity’ and ‘viewCart’ are responsible for retrieving the information for their respective functions and displaying it via the website or screen. The operation ‘scanBarcode’ represents the process of scanning a product's barcode in order to access other functions such as viewing an item and/or eventually purchasing an item.

4. **Payment**: Payment operations ‘getAmount’ and ‘voidPurchase’ represent functions typical for the payment process of a store, respectively they take an object's detail of price and voids the purchase if necessary at any time in between checking out and paying for an item.

5. **Admin and Employee**: As requested of the client, they required it so that the point of sale system would store transaction history where only an administrative user ,denoted as admin, is able to view this information. Operations ‘viewPurchase’ and viewReturn’ represent said function that is only accessible to one with administrative privileges. An employee, through their login privileges, are able to access functions through the scanner, separate from that of the website's functions.

Verification Test Plan (Test Sets)

- **Test Set 1: Scanner Test Set**

    - **Unit Level:** At the unit level the Scanner class is tested. In order to determine whether or not the class is functioning as intended, we will check if using the physical Scanner in-store would retrieve the item's information. In other words, the scanBarcode(barcode: int) method would be called when the Scanner scans an item’s barcode, which in this case is the argument. Assuming that we already know the items’ information we will check if the scanBarcode returns the correct information. For example, if we were to scan a black shirt the Scanner would tell us that the black shirt is a black short-sleeve shirt that has more color options and its ID number. A clear indicator that the test doesn’t work would be if Scanning the black shirt would return let's say a pink hoodie, which isn’t the same item. Not to mention another potential indicator would be that no item at all is returned. For future reference some test vectors that can be used are as follows: a blue crewneck, a brown t-shirt, and a green hat. The above explanation is all considering that there is no human error such as failing to update inventory.

    - **Functional Level:** At the functional level, the Scanner class’ implementation in the Shopping Cart class is then tested. At its core, the test determines whether or not the Scanner successfully adds the scanned item(s) to the shopping cart. Therefore, it would be considered a failure if the scanned item wasn’t added to the shopping cart or even if the wrong item was added to the cart. With this test being an extension of the previous unit test, the same test inputs can be used and observed in the output. Unlike the previous test, this test level doesn’t have much human error of its own, if it does it’s human error that occurred at the unit level of scanning the item.

    - **System Level**: Lastly, at a system level we are now testing if scanning an item via the Scanner class would update the Shopping Cart using the Shopping Cart class, which ultimately leads to the Payment class. Essentially what this test examines is

whether or not the scanned item that's added to the shopping cart is properly charged in the payment stage. There's much variability in the way this can be tested as one item can be tested through the stages, or multiple unique items with their prices. Therefore, for it to be determined that the system works at a system level in terms of scanning an item, adding it to the shopping cart, and then checking out and paying, we would need to do calculations and see what the shopping cart total would add up to and ensure that Payment class' getAmount() returns the same.

- **Test Set 2: Admin login and View functionality**

    - **Unit Level: Check If Admin Login Works :** For this test case we test the admin login. For us to determine if this class is working we will check if an admin login will work in order to gain admin functions. Via any working device, we can access the website or app page and select employee functions. This test should account for three scenarios: first a valid admin login and password, which in turn provides admin function to the user; second a invalid admin login with a wrong password, where user submits a valid login but an invalid password; and lastly an invalid admin login with a wrong password, where a user inputs an unrecognizable login and incorrect login. The variable; access: bool should return true signifying a successful login.

    - **Functional Level**: **Check Transaction History, Which Only Is Accessible By Admin :** In this level of testing, we test the admin class relation with the transaction history class. This test will see if the admin class is able to view transaction history via methods viewReturn(return: int) and viewPurchase(return: int), where the transaction history is storing that data. Similarly to the Unit Test, an admin would be able (or not be able) to log in to their account, with similar expected results, a successful or unsuccessful login. At this level of testing, we would expect new outcomes to determine if an admin is able to access transaction history. If an admin is successful in logging in, they should be able to access these functions. Through a successful login, the user should be able to see transaction info by methods of viewReturn(return: int) and viewPurchase(return: int). If they are unsuccessful in logging in, they will not be able to view these.

    - **System Level**: **Admin Checks Inventory With Login:** At the system level we will be checking if inventory is accessible through an admin login. Through the admin class, the viewing options use the transaction history class and then use the item and inventory class in order to view inventory. This level of testing requires

multiple inputs that may vary, for here we can see if an item is updated properly via a purchase or transaction. After a customer purchases (or returns) an item, the system should promptly update the inventory count. For us to test this we should note the inventory count for an item prior to a transaction and then following the transaction. For our expected outcome we should notice the item count decrease (or increase for returns) after the transaction has been made. Obviously after a successful login, the admin should be able to have 'view' functions and be able to note the above changes in inventory. Similarly we can also separate testing depending if one is making a transaction via in-store or on the website. Following suite to the above testing, both should have identical outcomes in that the inventory count changes after a single transaction.

Development Plan and Timeline:
Test Set 1: Samantha
Test Set 2: Kyle
Software Design Specification: Aman

Timeline:
Rough draft -
Revising -
Due Date - Oct. 27