Software Design Specification
For
ThriftWerks: Point-of-Sale System

Team members:

Ricardo Lemus

Jane Ho

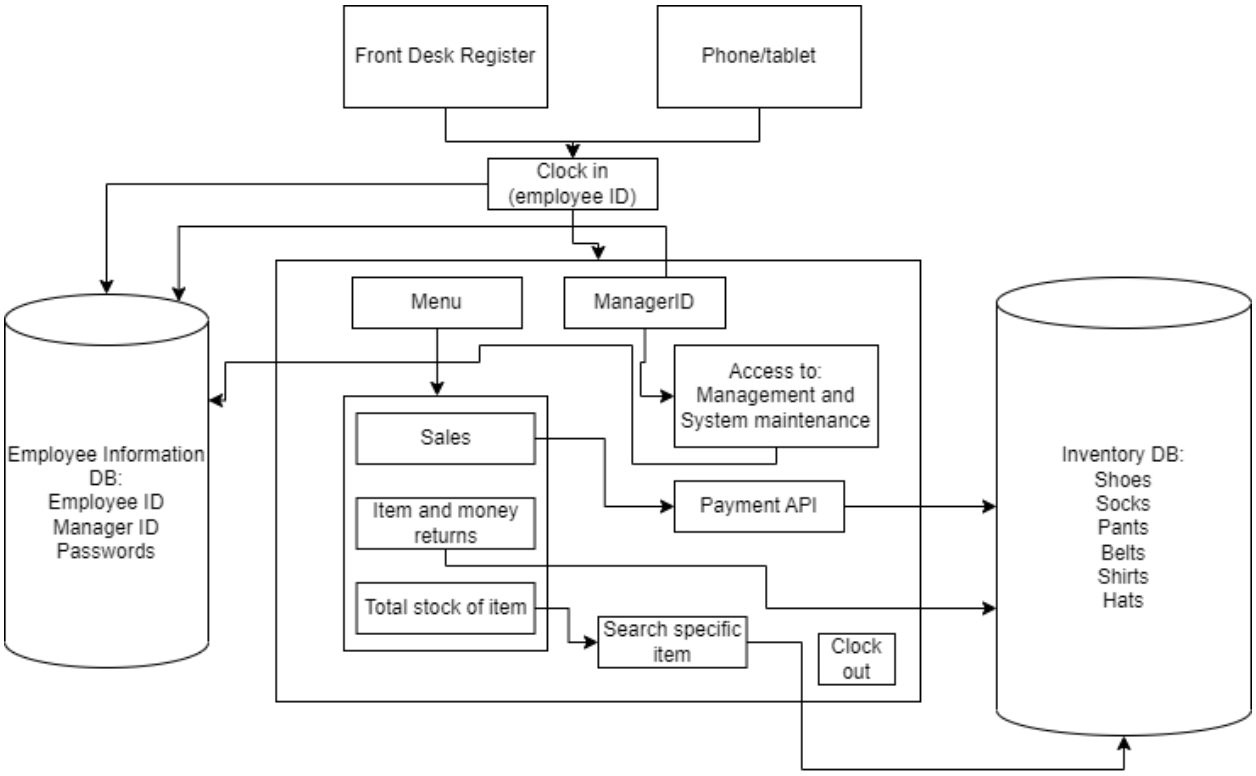
David Kaauwai

System Description

This software, which will be known as “ThriftWerks”, is a point-of-sale system that will facilitate the work of retail employees, most notably with processing sales, refunds, and inventory searching. The system can be accessed at front desk registers or through phones and tablets. From there, a login screen will appear where an employee ID and password must be inputted to proceed. The system will not only determine whether the account information matches what exists in the Employee Information Database, but will check if the ID matches that of a higher-level employee, such as a manager, so that more functions inaccessible by regular employees may be unlocked. Once the inputted information is confirmed to be valid, the system will direct the user to the main menu, which displays options of “Sales”, “Item and Money Returns”, and “Total Stock of Item”, all of which will utilize the Inventory Database. Higher-level employees will also have the options of “Management” and “System Maintenance”. “Sales” will use a payment API with the Inventory DB to carry out customer transactions when they purchase any item(s), and “Item and Money Returns” will directly access said DB to process customer refunds. “Total Stock of Item” will help search for specific items in the Inventory DB.

Software Architecture Overview

Architectural diagram of all major components:



The above software architecture diagram shows all the major components of the system we are designing. It shows what happens when a certain user logs in (manager or employee), two databases that store different information and specifically what information, and the different actions an employee or manager can do such as giving refunds or checking inventory on an item. Below are different characteristics that go more in depth on what each component does in the software architecture diagram.

Description of the software architecture diagram:

Front desk register: Connects to the “Clock in” screen which is to verify that the user is an employee of the store.

Phone/tablet: Also connects to the “Clock in” screen to verify they are an employee.

Clock in (Employee ID): Extends to the “Employee Information DB” to see if the ID and password match an account. Once logged in the “Menu” is accessible.

Employee Information DB: Stores employees/managers credentials.

Manager ID: Extends to the “Employee Information DB” to verify they are a manager and see if an ID and password match. Manager ID also extends to the special tools managers have.

Access to: Management and system maintenance: Allows managers to access moderations, employee management and system maintenance.

Inventory DB: Database where all stock of clothing is kept at.

Menu: Extends to sales, item and money returns (refunds) and total stock of an item (inventory).

Sales: Goes into “Payment API” which is used to process purchase transactions and it also extends into the “Inventory DB” to update the stock on an item.

Payment API: Used to process transactions.

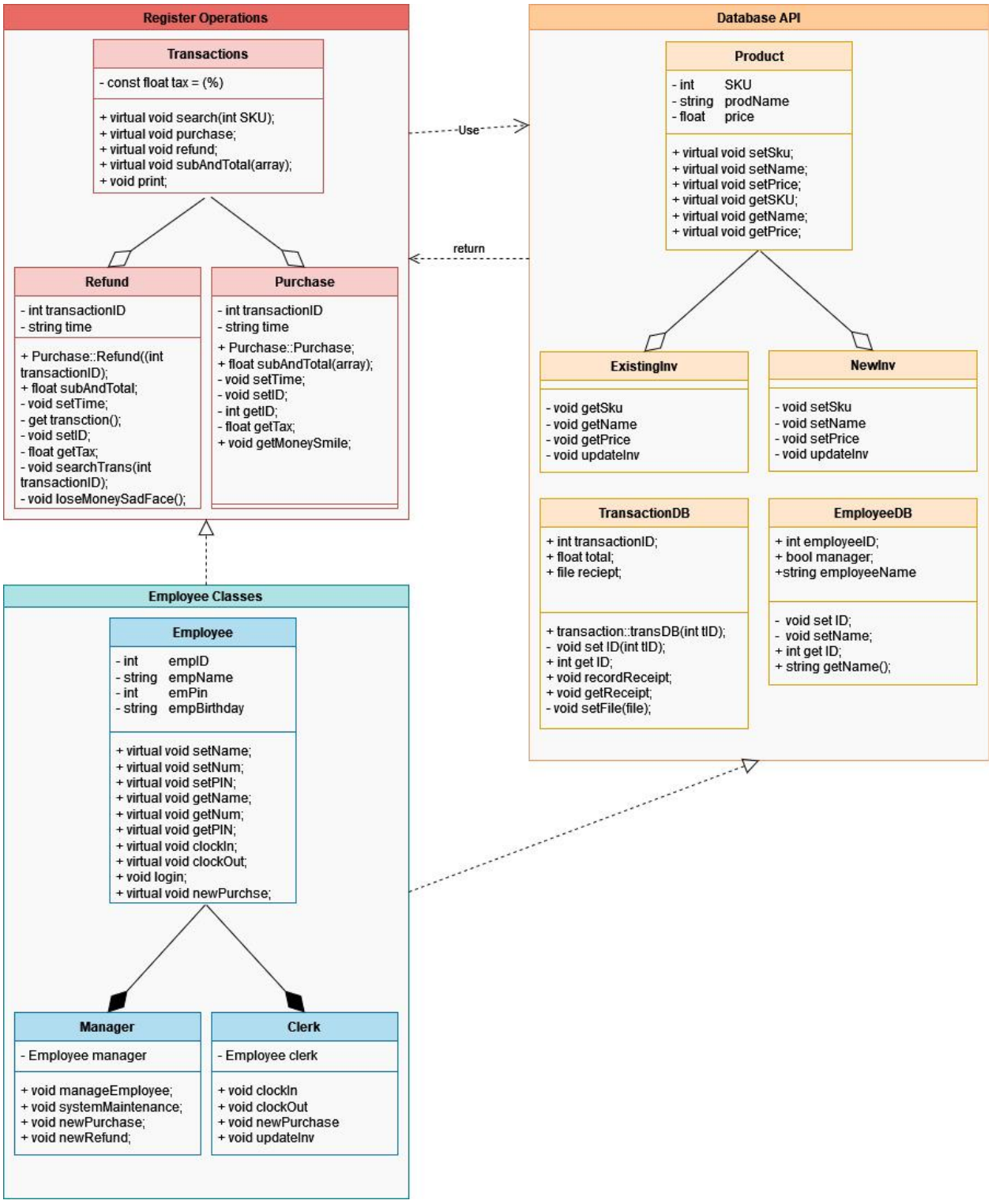
Item and money returns: Extend into “Inventory DB” to update the stock when an item is returned.

Total stock of an item: Goes into “Search specific item” which searches in the database of “Inventory DB” for a particular item.

Search specific item: Searches inside “Inventory DB” for an item the user wants to find.

Clock out: Logs a user out to prevent any non employee from tampering with the system.

UML Class Diagram



The image above is our UML diagram. The software will be divided into the three major sections of Register Operations, Inventory, and Employee Classes. Register Operations contains classes,

attributes, and operations pertaining to calculating transaction totals and storing transaction information. Inventory serves to hold information for all products in the store, retrieving existing product information and creating new ones. Employee Classes will have employee information that is necessary for identification and account login, along with functions that they have access to.

Description of classes:

Transactions: Base class for common transactions made by employees, which includes processing purchases and refunds, searching inventory, and calculating purchase/refund totals.

Refund: Child class of Transactions, includes functions for searching refunds made, getting/setting IDs of refunds, and setting times for them.

Purchase: Child class of Transactions, includes functions for getting/setting IDs for purchases made and setting times for them.

Product: Base class for basic product information that is in the inventory, such as SKU codes, product names, and prices.

ExistingInv: Retrieves information of an existing inventory item, important for searches.

NewInv: Sets information for a new inventory item.

Employee: Base class for employee account information, which includes ID numbers, names, PINs, and birthdays.

Manager: Child class of Employee, gives access to manager-only functions like system maintenance and managing employee information.

Clerk: Child class of Employee, includes functions for clocking in and out for work.

Description of attributes:

const float tax (Transactions): Will be accessed to calculate purchase/refund totals with tax.

int transactionID (Refund/Purchase): The ID assigned to a specific refund/purchase.

string time (Refund/Purchase): The time at which the refund/purchase was made.

int SKU (Product): The product's identifying code, will mainly be used for inventory searches.

string prodName (Product): The name of a product, can also be used for searching.

float price (Product): The price of a product.

int empID (Employee): The ID number of an employee, will be needed for login.

string empName (Employee): The name of an employee.

int empPIN (Employee): The PIN/password of an employee, will be needed for login.

string empBirthday (Employee): The birth date of an employee.

Employee manager (Manager): An object of an employee, specifically for managers. Should have access to manager-specific functions.

Employee clerk (Clerk): An object of an employee, specifically for clerks.

Description of operations/connectors:

The software is composed of three categories of classes, the primary interaction of ThriftWerks is done through register operations by Employee classes, which instantiates all transactions. Transactions are either refunds or purchases. Those are implemented by employee accounts, which are instantiated and constructed from the Employee abstract class. Employees may also instantiate new products through the NewInv class. Transactions fetch and update existing inventory through the ExistingInv class.

Development Plan and Timeline

Partitioning of tasks:

Implementation (1-2 months): Each person will work on programming one of the three sections shown in the UML class diagram (Register Operations, Inventory, Employee Classes) and parts of the SWA diagram relating to it. Jane will work on Register Operations, the menu of options after login) and the manager-only functions like system maintenance and employee management. Adam will work on Inventory, the Inventory DB, and the item searching function. Ricardo will work on Employee Classes, the Employee DB, and the login/logout function.

Testing (2-3 months): Test the implementation for verification and validation, ideally each section is tested by all three people to garner as many unique perspectives as possible to

minimize the amount of problems slipping through in the final product. (Alternatively: each person can do a different kind of testing, e.g. black-box, white-box)

Release and Maintenance (continuous): Send out the product for use and continuously fix bugs and add features when needed.

Team member responsibilities:

This project was split evenly between the three of us.

Ricardo Lemus:

- Work on the Employee class
- Finish the Employee DB and the login/logout functionality
- Will test the system with black-box testing methods

Jane Ho:

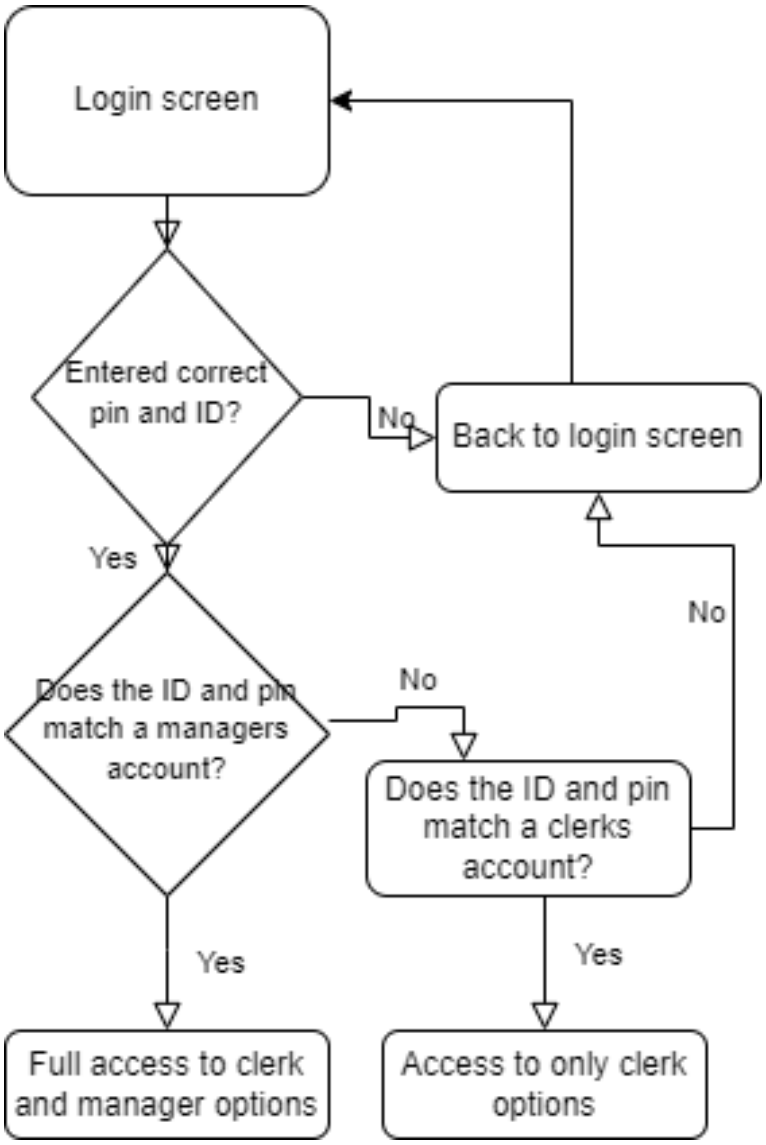
- Work on the Register Operations
- Menu options and manager only functions
- Will test the system with white-box testing methods

David Kaauwai:

- Will finish the Inventory
- Inventory Database and item searching functionality
- Will send out the product for use and report back to the team on what needs to be fixed or add any features requested

Verification Test Plan

Unit Tests:



1. Logging in as a clerk:
   The User will begin by inputting an integer into the point of sale interface. If the employeeId is contained in the employee database, it will return true and allow access to the rest of the system, showing clerk options.

   Expected input:
   1) Create mock employees to test login function and insert them into employee database. Set manager value to for specified mock employees to test blocking to manager system features. Note the associated names and employeeID.

2) Run function through set of mock employee ids that include both valid and invalid employeeIDs

Function:

```
login(empID)
        if(employeeDB::get())
                Switch: EmployeeDB::manager
                        Case true: //manager POS options
                        Case false: //clerk POS options
```

Expected output:
For all employeeIDs that are contained in the employee database and are input into the system, the system returns TRUE.
For all employee IDS that are not contained in the employee database, the function returns false.
If the manager bool variable is set to true in the employee database for the associated employee ID, the system does not show options that would be available to managers.

2. Logging in as a manager:

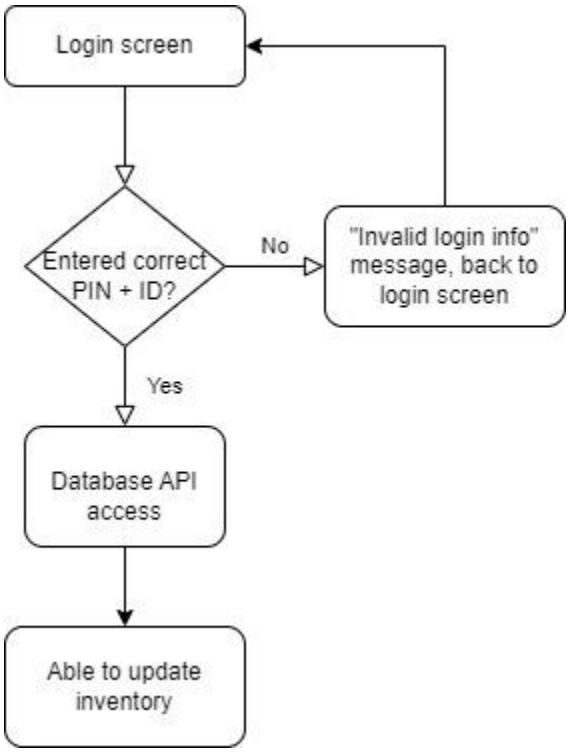
1. The user will first start with typing a **int empID** and **int emPIN**
2. Upon logging in if the **int empID** and/or **int emPIN** are invalid then no access to the system will be given
3. Upon entering the correct **int empID** and **int emPIN** then the system will check if it's a employee manager or a employee clerk
4. If it is a employee manager then the user will have access to all clerk functions and manager functions

The reason for this unit test is to check if the manager can even log in as the manager has exclusive options such as **voidmanageEmployee** which is set to customize an employee's profile.

The expected output is for the user to log in as a manager using an ID and pin that belongs to a manager and having access to all clerk options AND manager options. If a wrong pin/ID is entered then no access to the system is given and if the ID/pin belongs to a clerk then only access to clerk options is given.

Functional Tests:

1. Updating inventory as a clerk:



In order to access inventory-changing features, the login procedure must function as expected. If account information inputted does not match with existing information in the Employee Information DB, the user should be taken back to the login screen to try again. If a match is found, the clerk can move on to accessing the Inventory DB and utilizing the updating features. Adjustments made to an inventory item must be accurately reflected in the DB.

The objectives are to ensure that user login allows for expected users to pass through while blocking out unexpected ones (utilizes empID, empPIN, Employee clerk), and that the users who passed through are able to manipulate inventory data properly, along with said data being displayed and stored accurately (updateInc, setSKU, setName, setPrice, getSKU, getName, getPrice).

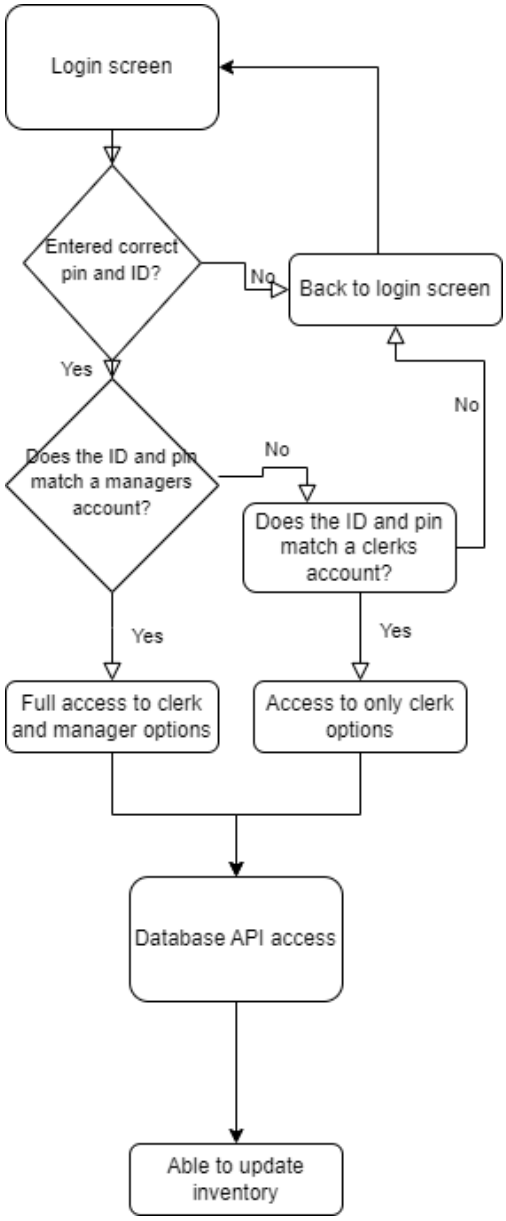
Example Expected Cases:

Set empID and empPIN for an *existing* clerk account //input
If empID == getID AND empPIN == getPIN
    Login is successful, unlock inventory access //expected output

Set empID and empPIN for a *nonexistent* clerk account //input
If empID == getID OR empPIN == getPIN
    Login failed, return to login screen //expected output

Have Clerk (Employee object) call updateInv, setSKU, setName, and setPrice. If calling getSKU, getName, and getPrice match what was inputted when called, the test has been passed. If they differ from expected, the test failed and should be investigated.

2. Updating inventory as a manager:



1. The user will first have to log in as a manager or clerk to get full access to all features
2. Once logged in as a manager, access to the **Database API** will be given which from there, the user can update inventory accordingly
3. Using the functions **int SKU** to set the SKU, **string prodName** to keep track of products, and **float price** to set the price on the product
4. The user will then use the function **void updateInv** to update the inventory

The reason for this test is verify that the manager can still access inventory just like the clerk can because in this current system both the clerk and the manager can do the same functions in the **Database API**

The expected output would be upon successful completion of logging in as a manager, then the user can still update the inventory accordingly. If it is a clerk pin/id then the result should be the same. If it is a invalid id/pin then no access to the system is given at all.

System Tests:

1. Refund register functions:

In order to execute any refund operations, the user must log in with an ID and PIN that matches with one in the Employee Information DB. If no match is found, the user will be taken back to the login screen. Otherwise, the user is directed to a menu with access to all clerk functions, and if the account information inputted matches that of a manager's, it will also include manager-exclusive functions. When selecting the Refund option, the system should proceed to the refund page and prompt the user to scan or input codes of items being refunded, drawing item information from the Inventory DB. The operations that take place in this action are calculating the refund total and setting the time and ID of the transaction. With every transaction made, the screen will reset to a clean slate for the next customer's refund, and the process is executed again until the user decides to return to the menu and/or log out.

One of the major objectives is to ensure that the software's operations work as expected as it passes through numerous classes. For example, if employee information somehow changes instead of inventory information as refunding operations are handled, the interactions between functions should be investigated. Other objectives are ensuring that manager-specific features are only available to managers, SKU code inputs and transaction information are retrieved and stored properly, refund calculations are consistently accurate, and the user can smoothly return to the menu and/or log out when desired.

Example Expected Cases:

Set empID and empPIN for an *existing* employee account //input
If empID == getID AND empPIN == getPIN
    Login is successful, unlock menu access //expected output

Set empID and empPIN for a *nonexistent* employee account //input

If empID == getID OR empPIN == getPIN
    Login failed, return to login screen //expected output

If empID belongs to a manager, manageEmployee and systemMaintenance methods should be accessible. Otherwise, clerks should not be able to access said methods.

Set SKU, prodName, and price for an inventory item //input
Have clerk make new transaction under Refund class and input SKU code //input
If SKU is found in Inventory DB
    getName equals prodName //expected output
    getPrice == price //expected output

Clerk (Employee object) calls subAndTotal, and getPrice and getTax are called for calculation. If the total calculated doesn't match the expected final calculation, the test failed and should be investigated.

2. Purchase register functions:

System Test for Use Case Clerk Processes a purchase:
In order for an employee to process a transaction, the employee must login with either a clerk or an manager ID, the employee then selects an option to start a transaction. Once the transaction is begun, until the clerk indicates through changing the option that the transaction is completed, the clerk enters SKUs through the point of sale interface. That SKU is searched in the product database and returns the associated item name and price and stores it locally and displays that item and price on the screen of the point of sale interface. When the clerk indicates that the transaction is concluded, the system then totals the amount and adds tax to the sale. The system then inserts a copy of the receipt into the transaction database, along with the associated transaction ID.

Transaction class generation pseudo code:

```
Puchase(int transactionID, file){
   Trans(transactionID);
   setID();
   setTime();
   setFile(file);
}
```

Expected output:
Find each transaction id and return the receipt file that is associated with the transaction ID.

Expected input: transactionID from employee input.

```
main()                    //employee chooses which kind of transaction in main;
login()
Switch
   Case refund:
         loseMoneySadFace();
         break;
   Case purchase;
        getMoneySmile();
        break;
    Case updateInv;
        updateInv;
        break;

Void getMoneySmile()
        bool done = false
        int receipt[]
        int item = 0
        Int price = 0
        while(done == false)
                cin >> SKU
                Receipt[x][y] = search(SKU);          //search returns price and item name, stores
                                                        to receipt array
        subAndTotal(receipt[][]);
        print(getID);
        //prints receipt writes to a receipt file of refund transaction,, stores in transaction
        database;
```

Expected output: receipt file from original transaction;

A new receipt file inserted into transaction db under generated transactionID
The refunded bool variable set to FALSE in transaction db.
The file includes transaction ID internally as a checksum and.