Point of Sale System

By: Matthew Smith, Nicholas Stark, Alicia Loya

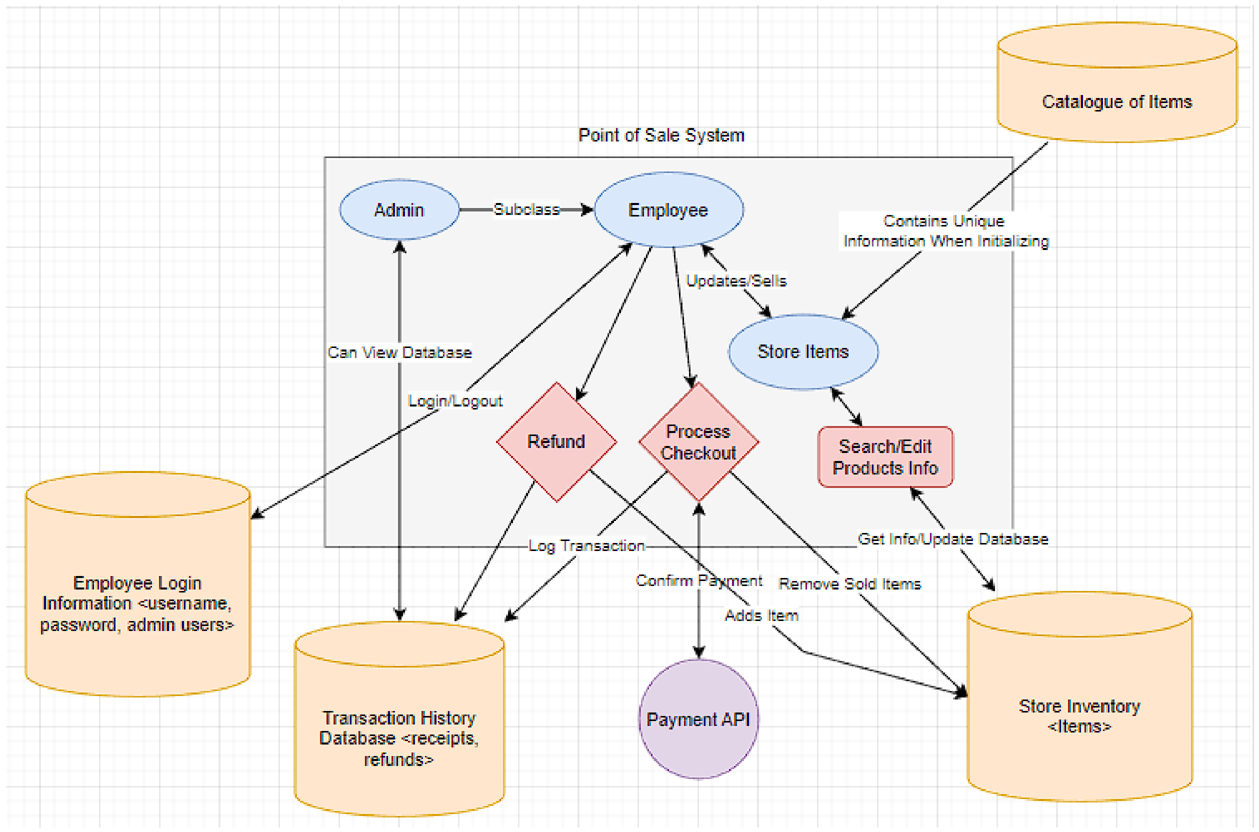
3/10/2023

1. Brief Overview

This point of sale system is designed to maintain a store inventory and monitor all transactions, such as updating inventory and managing selling items and processing refunds. This system will maintain several databases, employee login information, the store item inventory, and a record of all transactions. The user database will include the login information of all current employees which will enable them to access the system. The employee login will also have the subclass of administrator if the employee has been logged as an administrator in the database. The storage database will store the information of all products that are created and updated by the employee functions. An item class will be used to identify the product and its attributes, such as its unique ID, price, size, color, and quantity. There will also be a transaction history database, that is updated when an item is sold or refunded in order to maintain an accurate record for review. However, only the Admin subclass of the Employee will have access to the transaction history database in order to keep such information secure. Employee functions will be used for searching for available products and updating the database information. In addition employees will be able to process the sale of Items and refunds, which will in turn update the Items inventory and log transactions history. In addition, Employees will also be able to change both the quantity and price of an item, as well as enter a new item into the system with a unique barcode identification number. By utilizing these functions, store management will be convenient and easy for all Employees as this system will be implemented on mobile Ipad devices in a way that is convenient for employees to interact and use. Finally, it is important that this program contains up to date information in order for this program to be effective in retail store use.

2. Software Architecture Overview

Software Architecture Diagram

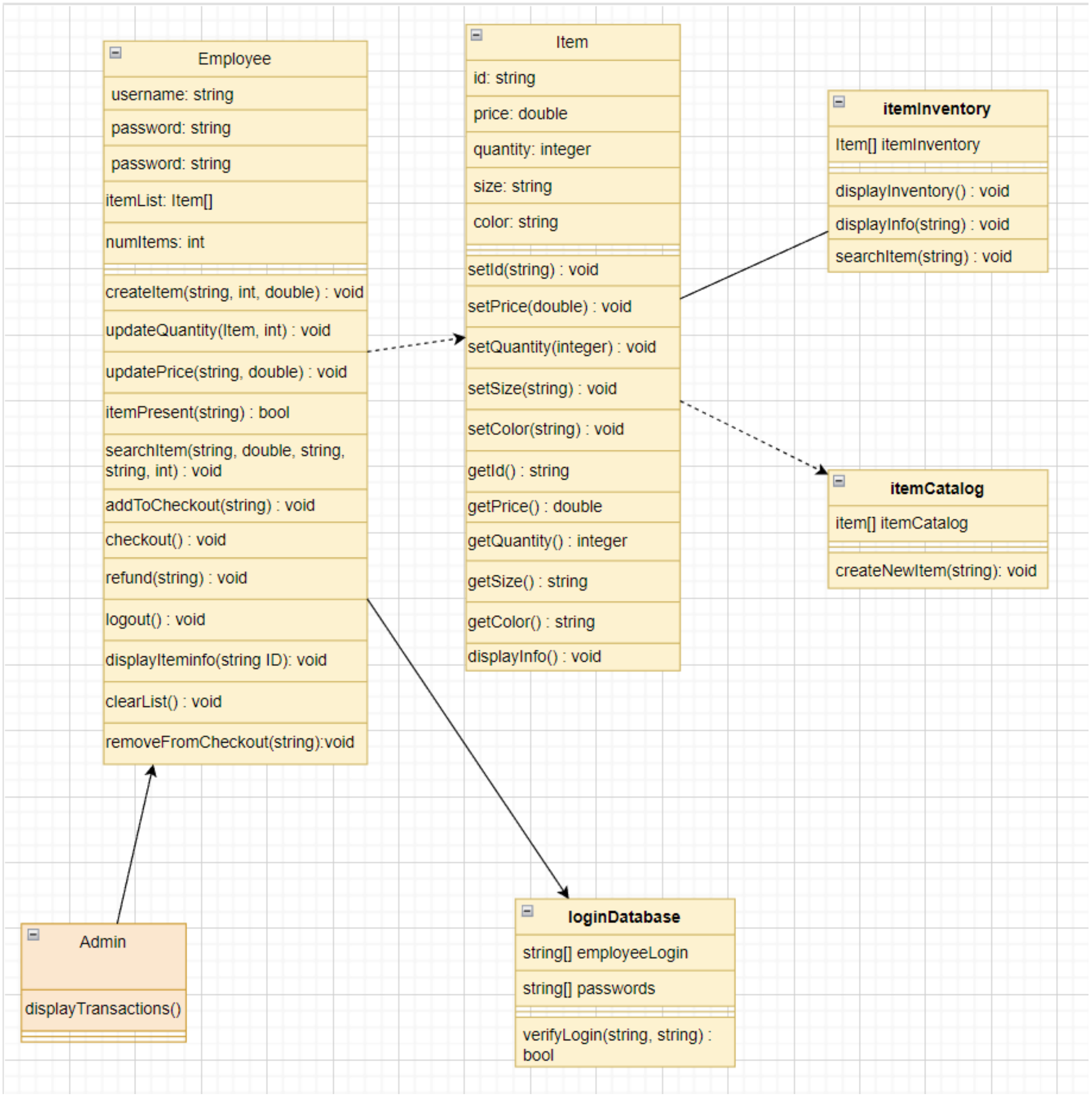


Description Software Architecture

The Point of Sale system consists of many functions that interact with store items in order to update and manage store inventory, process transactions such as sales and refunds, and includes convenient logging of information in order to maintain a concise and well maintained database. In the Diagram above, an Employee user, once logged in by entering a password that matches with the user database, is able to access and manipulate the information of store items. First, the employee is able to create new Items to add to the store inventory, this is done by choosing to create a new item with its unique ID, along with a selected price and quantity. Using the new item’s ID, the newly initialized item will retrieve information from an external catalog database that includes the information, such as the size, color of the item, this ensures that the information of the item is accurate to other retail chains. In addition to creating a new item, the employee is capable of updating the price in case of a desired sale, or updating the quantity of items in case there are incoming shipments of goods. Items that are created and updated through the employee's function are stored inside of the store's inventory, which stores each Item with its product information, which can then be accessed by the employee by searching by any of the item's characteristics. These items, in addition to being updated by the employee can then be added to the Employee's list of items, by either entering them with an ID or Barcode in order to be sold at checkout. When the employee chooses to checkout its selected items, the system will utilize an external payment API in order to handle the charging the customer, if the payment is accepted, the checkout function will proceed and sell all of the selected items. When this happens, each item will be removed from the inventory, the checkout process will create a receipt that will be added to the transaction history database that includes information about the transaction such as the time, items included, the total price. Along with checkout transactions, employees are also able to refund items by entering an item's ID, however refunds will only be processed in cash, outside of our system's scope. These refunds will also be logged in the transaction history database with the same information as the checkout. Although the employee can manage transactions, only the Admin subclass of the employee class will be able to access the transaction history in order to keep information secure, this will allow them to view all previous transactions that have been logged by the employee functions.

UML Class Diagram ***update 3/23/2023*** new classes

- **itemInventory, itemCatalog, loginDatabase**



Description of UML Diagram

Classes

Employee/Admin

The most important class for the user interface of this program is the employee class, along with the admin employee subclass. In order for the user to access the functions of this class, the user will have to utilize the login function, which will take an input string for both the username and password, if the username and password matches, a boolean value of login status will be updated to true, allowing the employee/admin to utilize the classes methods and store the String username of the logged in class. After logging in, the employee will be able to use the employee functions which will only work if the loginStatus variable is true. The class will also store the int numItems for the number of checkout items and an array of Items called itemList which stores items that will be sold at checkout. Here is a list of functions that the employee class will be able to utilize. All functions that update inventory information will also add a record of each transaction to a database that stores all transactions.

Employee

Void createItem(string ID, int quantity, double price) - will create a new instance of the Item class, with the ID set to the input ID that is present on the new item, the specified quantity, and the selected price. If an item of existing ID is found, it will update the quantity and price of the existing item, information for item attributes will be updated for the new instance of the item class by checking using the Item's ID in an external database to retrieve the information about the item with the matching ID. The new item that is created will be stored in the item Inventory.

void updateQuantity(Item item, int amount) - will update the item quantity stored in the inventory with either a positive or negative amount, however the value cannot be changed to negative, instead displaying zero.

void updatePrice(string ID, double price) - this function will update the item with the matching ID to the new price entered.

bool itemPresent(string ID) - this function will return true or false if the item with the matching ID is present in the inventory.

void searchItem(string ID,double price, string size, string color, int quantity)

This function will search for Items in the inventory with the matching characteristics, if a characteristic is left empty, the list of items will not be filtered according to the empty parameter, allowing for precise filtering.

void addToCheckout(string ID) - adds the item selected by ID to itemList[], and updates the int numItems by 1. The ID is either added manually by the employee or entered by utilizing the mobile devices camera in order to scan the item's barcode.

void checkout() - this function will take all items that have been added to the employee's itemList[] and calculate the price of all the items combined. From there it will access an external online payment system that will take the form of payment, if the payment is accepted, all items in itemList[] will be removed from the Item's quantity stored in the database and the numItems will be set to 0 and the array itemList[] will be cleared of all items.

void refund(string ID) - this function will take the ID of the item that is scanned or entered and will increase the inventory quantity by 1, similar to other transaction functions, it will update the transaction history for record.

void logout() - this function will log the employee out of the app, set the loginStatus to false, clear the username,remove all items from itemList and setnumItems to 0.

*****Updated functions 3/21/23*****

void displayIteminfo(string ID) - this function will display all of the information of the specified Item in order to view information about a specific product

void clearList() - this function will remove all items in the employees item list, however it will not modify the quantity of the Items

void removeFromCheckout(string ID) - will remove the item with the matching ID, if there are multiple of the same item, it will remove the first instance of the item from the list

Admin subclass

void displayTransactions() - this admin exclusive function will access the transaction history database and display it to the admin user.

Item

The item class is used to store every unique item based on its unique barcode. Each item class is stored in a database filled with every item that exists. Items will generally be accessed by the item ID when they are searched for in operation. Information, such as color, size, ID will not be changed once initialized,as is constant according to the unique barcode initialized according to an external catalog database. However, the item's quantity and price can be changed by the employee's functions. Below are the Item classes existing functions.

string ID() - this function returns the items ID. This function is useful for comparing the input search ID’s of the employee function for finding the item in the database.

string price() - returns the Items price

string color() - returns the Items color

string size() - returns the Items size

double price() - returns the Items price

int quantity() - return item quantity

void displayInfo() - display all info pertaining the specific Item, such as the ID, color, size, and price.

void changePrice(double price) - changes items price, this function is used by employee functions that change the price

void changeQuantity(int change) - changes items quantity adding or subtracting by the input change, this function is used by employee functions that change the quantity

Databases

Below is a list of databases that this program will access in order to store information that is used by the aforementioned classes.

Employee Logins

This database is used to store all employee information in the format of a hashmap, storing the username and the associated password for logging in.

Inventory

This database stores all of the existing Item classes as a dynamic list of classes, when an employee function is used to access this databse, it will search the elements of the database for the matching item ID in order to operate on that value.

Item Catalog

This database stores the relevant Item information that is needed for initializing a new Item into the inventory. This database is only accessed when creating a new Item in the inventory as it contains external information from each Item's manufacturer, such as the color, size, and ID. This database is important as it retrieves existing information that is used throughout all stores that sell the same items. As a result, certain information that is used to create an item into our inventory cannot be altered in order to prevent misinformation about our products as our store is a distributor.

Transaction History

This database stores all transactions that are recorded when they happen as a result of employee functions. All transactions are stored as digital receipts that are then uploaded into the transaction database. This information can only be accessed by the store administrators when using the displayTransactions function in order to keep customer information confidential. These transactions include the date and time of the transactions, the form of payment used, the checkout price total along with the items that were purchased for each transaction. This information is required in order to keep track of store earnings and keep record in ase of customer disputes. In order to make sure that the database does not run out of space, each transaction will be removed from the database automatically after 3 years.

External Systems

For external systems, the most important system will be an external payment service that is accessed when the checkout function is used. This system will be required in order to process payments and return whether or not the payment has been accepted. If the payment has been declined, the selected items will not be purchased. In addition, this external system will be able to approve cash payments, contactless payments and credit or debit cards. The external system will be the square mobile reader, which is compatible with ipads which is the device that this program is intended to function on.

3. Development plan and timeline

Partitioning of Task & Team Member Responsibilities

Because this project consists of multiple classes and databases, it will be important to develop each class separately before checking to see how they interact with each other. In the initial phases it will be important to focus on the methods of the item class and the employee functions. Because the employee class requires the Item class to work, the Item class will be developed first in order to be stored in to be used in employee functions, which means that it is important to also confirm that the Item class can interact with a test inventory and catalog database. After the item class has been developed,the Employee functions will be developed and tested in order to confirm that the Employee class can interact with the item and update the databases. Following this, it will be important to develop how the Employee function will interact with the external payment system, in order to confirm that the function is usable for common in-store use. Once the basic function of the Employee and Item class is tested, the Admin class will be implemented in order to view the transaction history, at this time the employee function will be updated to include updating the transaction history database into these functions since the operations themselves will have been tested and confirmed to function properly. Once the classes and databases have been proven to work in a testing environment, this system will be implemented to select stores that will operate separately from each other in order to test real world operations and view how useful and reliable our system is. Once our system has been tested to work with individual stores properly, it is expected that it will be possible to combine the inventory between stores so that customers can access stock that can be found in another store at long distance. Once this phase is reached, it will be important to revisit some of how some of the functions operate in order to include relevant information between the traversal of objects between stores and each item's location. Finally, because the system utilizes an Ipad in order to run the software, the user input will mostly be using the touch screen to use functions, with parameters being entered using the keyboard, and outputs being displayed on the same screen. Because of this, it is important to make sure that along with functional testing, the user interface is also clear and easy to interact with for employee use.

*****Update 3/23/2023*****

Task

Matthew - Employee class development, System testing

Alicia - Item class development, unit testing

Nicholas - database initialization/management, Functional testing

Timeline

April - initial development
Early May - testing of classes and operations
Mid/Late May - testing of functions with test database
June/July - fine tuning of program/interface for employee use
August-2024 - test isolated in individual stores

2024 - implementation to stores in larger area, integrate databases to single region wide database

Verification Test Plan

Below is a list of test plans that will be used in order to test functions that are present in our Point of Sale System.

Unit Test

1. **Change Item Quantity: updateQuantity(int) : void**

This test will verify that the updateQuantity method correctly updates the quantity of an item by the input amount. The method will be tested by creating a new item with its quantity initialized to an integer amount. The item's quantity will be checked to verify it was initialized correctly. Then the updateQuantity method will update the item's quantity by the integer input. The item's quantity will be checked again to make sure it was updated by the amount specified. An error will be thrown if the quantity is negative and will set the item quantity to 0. The test cases will cover updating by a positive amount and by a negative amount

Function test Code

```
Item shoes = new Item("0000000001",4.99,0,"L","Blue"); //constructor,
initializes price with 4.99, quantity to 0, size to "L", and color to
"Blue"
println("Quantity: " + shoes.getQuantity());
shoes.updateQuantity(3); //adds 3 to quantity
println("Quantity: " + shoes.getQuantity());
shoes.updateQuantity(-2); //removes 2 from quantity
println("Quantity: " + shoes.getQuantity());
shoes.updateQuantity(-5); //removes 5 to quantity, more than items
quantity
println("Quantity: " + shoes.getQuantity());
```

Successful Use Cases

Case 1. The item's quantity is initialized to 0 and updated by an amount of 3. The item's quantity is successfully updated to 3**.**

Case 2. The item's quantity is initialized to 3 and updated by an amount of -2. The item's quantity is successfully updated to 1.

Error Cases

Case 1. The item's quantity is initialized to 3 and updated by an amount of -5. An error is thrown, and the item's quantity is set to 0 instead of -2.

2. **Change Item Price : setPrice(double): void**

This test will verify that the setPrice method correctly updates the price of an item to a new input amount. The method will be tested by creating a new item with its price initialized to a double amount. Then the setPrice will be called with the item's id to the specified amount. The item's amount will be checked to verify it was updated to the corresponding amount.The test cases will cover updating the price to a positive amount, zero, and a negative amount.

Function test Code

```
Item shoes = new Item("0000000001",4.99,2,"L","Blue"); //constructor,
initializes price with 4.99, quant 2, size to "L", and color to
"Blue"
println("Price: " + shoes.getPrice());
shoes.setPrice(8.99); //changes price to 8.99
println("Price: " + shoes.getPrice());
```

```
try{
shoes.setPrice(8.99); //attempt to change price to 0. Cause error
}
catch(e Invalid_Argument){
     println("Please enter a value greater than 0");
}
println("Price: " + shoes.getPrice()); //price stays unchanged


try{
shoes.setPrice(-2.99); //attempt to change price to -2.99. Cause
error
}
catch(e Invalid_Argument){
     println("Please enter a value greater than 0");
}
```

Successful Test Cases

Case 1. The item's price is initialized to 4.99 and updated to 8.99

Error Cases

Case 1. The item's price is initialized to 4.99 and updated to 0. An error is thrown, and the user is asked to input an updated price greater than 0.

Case 2. The item's price is initialized to 4.99 and updated to -2.99. An error is thrown, and the user is asked to input an updated price greater than 0.

Functional Test

1. **Searching for an item in inventory**

For this test, the searchItem function will be tested to verify that an item added to the inventory database returns the correct item searched. The test will be performed assuming all items in the inventory have initialized values. First, the employee will enter the name and/or description of the item. The system will then use a loop on getID() for all items, and add the items with the specified ID for the item searched into the ItemList array. From there, the system will use the getPrice(), getQuantity(), getSize(), getColor(), and displayInfo() functions on each of the items in the inventory. After the info is displayed, the ItemList array is cleared. The amount of items displayed will narrow based on the input received. For example, if the employee searches for “slippers”, it will display fewer results than if they searched for “shoes”. If the item searched for is not present in the catalog (making the ItemList array empty), the additional methods will be skipped and instead the system will display an “Item not found in inventory” message. If a more descriptive input is required, the system will accept the following: “[color] [item]” for item color, “size:[size] [item]” for item sizes, and “[item] price>/</=[price]” for price. The barcode ID for an item cannot be accepted as input since each barcode is unique for each item in the inventory.

Function test Code

```
searchItem(“slippers”); //testing for an item
slippers.getID(); //gets portion of barcode referring to slippers
… //iterates through the entire inventory and adds barcodes including portion
//identifying slippers to ItemList
if (ItemList.length==0) {
print(“Item not found”); //output if the list is empty.
}
Else {
For (int i=0; i<ItemList.length; i++){//output if the list is not empty.
ItemList[i].getPrice();
ItemList[i].getQuantity();
ItemList[i].getSize();
ItemList[i].getColor();
ItemList[i].displayInfo(); //this line is what produces the output.
}
}

ItemList.clear(); //clears the list so it can be used for other methods.
```

Successful Use Cases

Case 1. The item exists in the inventory, so the information of all the items attributed to the item searched will be displayed. This display will show each of the relevant items from top to bottom. A specified order of the items is not necessary for the display.

Case 2. An item is searched for with the descriptive input. The items with the specified information will be displayed from top to bottom. The display will show the information of the items from top to bottom in no specified order.

Case 3. An item searched for that is not present in the inventory will result in the system displaying “Item not found”

Error cases

Case 1. Any input that is entered which does not include an item will cause the system to simply return and prompt the user to enter a clothing item and display parameters for descriptive searching.

Case 2. When a barcode ID is entered, the system will return and prompt the user to enter a clothing item and display parameters for descriptive searching.

Case 3. The system will only be able to search for one item at a time. If the user attempts to search for more than one item such as “[item] and [item]”, the system will return and prompt the user to enter a clothing item and display parameters for descriptive searching.

Case 4. Attempting to search using an invalid parameter (such as price<0) will cause the system to return and display “Invalid parameter” and display rules for descriptive searching.

2. **Displaying item information**

For this test, the displayItemInfo method will be tested to verify that the system correctly displays the information regarding a specified item (Note: this is different from the displayInfo() method used in the searchItem function). This test is done assuming the methods involved create proper assignments. The employee must input the barcode ID of the item specified twice. The system will then call the methods getPrice(), getSize(), getColor(), and displayInfo(). Aside from the quantity of the item, the remaining item information will be obtained from the barcode ID. The display will show from top to bottom: The item name (as specific as the item will allow), item size, item color, and item price.

Function test Code

```
displayItemInfo(“000000000001”); //testing the barcode entered
verifyItemInfo(“000000000001”); //testing the barcode entered correctly
if(displayItemInfo!=verifyItemInfo){
print(“Barcodes do not match”);
}
… //loops through the entire Inventory until a match is found
item.getPrice();
item.getSize();
item.getColor();
item.displayInfo();
Else {
print(“Invalid ID”);
```

Successful Use Cases

Case 1. When the barcode of an item present in the inventory is entered, the display shows the name, size, color, and price of the item from top to bottom and the status of the item as “for sale”.

Case 2. When the barcode of an item sold from the inventory is entered, the display shows the name, size, color, and price of the item from top to bottom.

Case 3. The same barcode entered twice will display the item name, size, color, price, and status of the item from top to bottom.

Error Cases

Case 1. A barcode ID which has not been initialized in the inventory database will cause the system to display “Invalid ID”

Case 2. To prevent a barcode ID from being entered incorrectly, the user will be prompted to enter the barcode twice. If the barcodes do not match, the system will alert the user that the barcodes are not the same.

System Test

1. **Employee Class will select and sell an item from the Inventory.**

In this test example, an authorized admin class will create an Item for testing purposes and give the item a positive non-zero Items and quantity. This test case assumes that the Items initialized have correct information, as it will be testing the interaction between the Employee Class, Item class and external databases and systems. Once the Item has been created, the employee class will add the existing items, testing a single amount or multiple amounts to the checkout list. Once the chosen items have been selected, the system will utilize the checkout method, which will calculate the subtotal of all the items, sending the system to an external payment system. After this, the external system will confirm whether or not the payment has been accepted, proceeding with the transaction and removing all of the selected items from inventory and logging the transaction into the transaction database.

System Test Code

```
Admin Matthew = new Admin();
Matthew.Login(“Matthew”, 12345) //testing user for methods
Matthew.createItem(“000000001”,1,10);//creates new Item with ID and price
…
//create multiple items for different scenarios
Matthew.addToCheckout(“000000001”); //test item
…
//test with multiple items or attempt to sell amount greater than quantity
try(){
       Matthew.checkout(); //may cause error if payment fails
}
catch (payment_fail_error){
       print("Payment Failed");
}

Matthew.displayItemInfo("000000001"); //confirms that Item has been updated
…
//check all items that have been sold to confirm updated quantity
Matthew.displayTransactions();//will show whether or not payment was successful
```

Expected Results/Alternative Outcomes

Successful Use Cases

Case 1. Items that have been added and will be sold with no payment error and the information of each item will be updated accordingly such as selling 1 item with 1 quantity will result in the selling of that item, charging the customer and resulting in the item's quantity being modified to 0, and record of the transaction will be displayed after the transaction, along with the items no longer being in the itemList[].

Case 2. Multiple Items will be sold successfully and the inventory of all items will be adjusted and clear from itemList[], along with the transaction being logged in the transaction database. If multiple items of the same ID are present, the transaction will update the inventory to the correct number that have been sold.

Error Cases

Case 1. A checkout with 0 items will simply return without proceeding to payment, as it is not possible to sell 0 items.

Case 2. The items that have been added will not be sold due to a payment failure, the items will not be sold, however the items will remain in the itemList[] array, allowing for another payment attempt be re-using the checkout function.

Case 3. If an item is added to the inventory more times than it has quantity, the addToCheckout() will not add the item, instead it will display that the item is out of stock. However the items that are present in the itemList will not be removed from the List. An example would be adding an item 3 times if it has a quantity of 2. The program will still be able to add more items to checkout if the List is not full.

Case 4. The maximum number of items, 10, is reached. In this case, the addToCheckout() function will not add the item, instead displaying that the maximum number of items has been reached. This value has a limited size in order to prevent customer theft.

Case 5. An item that does not exist is attempted to be added to the itemList[]. The addToCheckout function will display "Item does not exist" and not update the add the item to itemList[]. However, itemList[] will not be cleared, allowing for normal interaction to continue.

2. **Employee will create a new item and access/modify the items information**

In this test example, an admin class will be used to create a new item with entered information and attempt to update the item information that can be modified, such as the price and the quantity. In this example, the admin/employee functions will be used to modify the item's information, as the employee functions are used to access an Item's functions. In addition to making sure that the employee functions are capable of utilizing the item classes methods, it will also be important to make sure that the information that is being modified is reflected inside the item Inventory, so that the information is accurate and up-to-date for all employee users.

System Test Code

```
Admin Matthew = new Admin();
Matthew.Login("Matthew", 12345) //testing user for methods
Try{
       Matthew.createItem("000000001",1,10.00);//creates new Item with ID and price
       …
       //create multiple items for different scenarios
}
Catch (e ID_not_found){
       print("Item does not Exist");
       return;
}
Catch (e invalid_argument){
       print("Cannot have negative price/quantity");
       return;
}

Matthew.displayItemInfo("000000001"); //check initialized values before modifications
… //test other items

//trying positive Values that will not throw errors
Matthew.updateQuantity("000000001",2);
Matthew.updatePrice("000000001",15.00);
…

Matthew.displayItemInfo("000000001"); //check values after modification
itemInventory.displayInventory(); //make sure inventory reflects modified values
…  //test display items


Try{
       //examples of errors
       Matthew.updateprice("000000001",-15.00);//invalid price
```

```
        Matthew.updateQuantity(“000000001”,-500);//will result in negative quantity,
        throws error
        …
        //create multiple items for different scenarios
}

Catch (e invalid_argument){
        print(“Cannot have negative price/quantity”);
}

Matthew.displayItemInfo(“000000001”); //values unchanged if there is an error
itemInventory.display Inventory() //make sure inventory reflects modified values
… //etc

//end of test
```

Expected Results/Alternative Outcomes

Successful Use Cases

Case 1. Items will be successfully created and modified, in addition, information that is changed will be reflected in the itemInventory database.

Error Cases

Case 1. Attempting to initialize an Item with a nonexistent ID will throw an error and the item will not be created or added to the inventory.

Case 2. An Item that is initialized with a negative quantity or price will result in an error, which will not create the created items.

Case 3. Items that are created successfully will then be modified to have impossible values, such as negative quantities or prices. When this occurs, the item's price will not be updated, and the quantity will be set to zero.

Case 4. Item info is updated successfully, however the information is not reflected onto the database, meaning that the changes are only stored locally.