Clothing Store Point of Sale Software Design Specifications

Contributors: E.J. Gordon, Matt McGuire, Brian Dang

1 Introduction

The Clothing Store Point of Sale System (abbreviated as CSPoSS) is a software system designed to facilitate transactions within a clothing store. The system is accessible by employees only and a third party will manage payment processing. Employees can view and update inventory and process transactions and returns.

2 Software Architecture Overview

The main components of the CSPoSS are as follows:

- Employee Device
- Employee App
    - Employee Interface
    - Admin Interface
- Cash Register
- External Payment Processor
- Transaction History Database
- Inventory Database

2.1 Employee Device

The ***Employee Device*** may be either a tablet or phone. The device must have a camera to recognize and scan item barcodes.

2.2 Employee App

The ***Employee App*** must be iOS and Android compatible. Its responsibilities include:

- Search inventory
- Update inventory
- View inventory

Additionally, the employee app has two interfaces: ***Employee*** and ***Admin***.

2.2.1 Employee Interface

The ***Employee Interface*** is the interface through which all employee operations are performed.

2.2.2 Admin Interface

The ***Admin Interface*** belongs to the ***Employee Interface***. The ***Admin Interface*** is different from the ***Employee Interface*** in that it allows the user to access the ***Transaction History Database***.

2.3 Cash Register

The ***Cash Register*** will be provided by a third party but must be able to connect to the ***Employee Device***.

2.4 External Payment Processor

The ***External Payment Processor*** will be able to accept debit and credit for transactions. It will be connected to the ***Cash Register***.

2.5 Transaction History Database

The ***Transaction History Database*** stores all transaction history data separately and securely and can be accessed from the ***Admin Interface***.

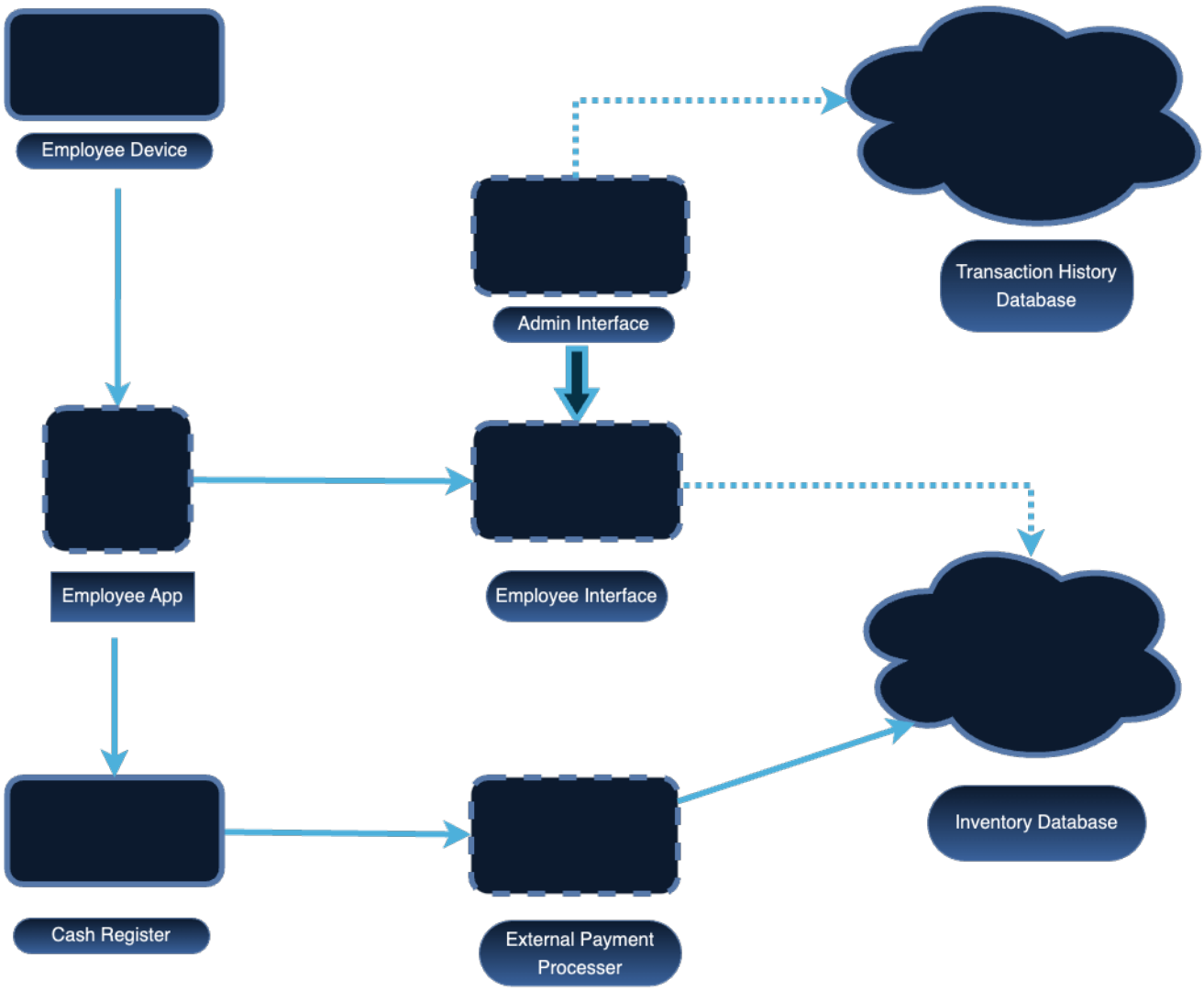
2.6 Inventory Database

The ***Inventory Database*** stores inventory data and can be accessed from the ***Employee Interface***.

3 CSPoSS Software Architecture

The following section will cover the structure and nature of the software architecture with regards to the CSPoSS.

3.1 CSPoSS Software Architecture Diagram



3.2 Description

The CSPoSS architecture is designed to be flexible; it is buildable and can be added to at a later stage in development if necessary. Everything begins with the ***Employee Device***. The ***Employee Device*** gives necessary context to the rest of the architecture. The ***Employee Device*** is used to access the ***Employee App***, which opens to the ***Employee Interface***. The ***Admin Interface*** belongs to the ***Employee Interface*** and allows the user to access the ***Transaction History Database***. All users can access and update the ***Inventory*** *Database,* but only users logged in to the *Admin* ***Interface*** may access the ***Transaction History Database.*** The ***Employee App*** also connects to the ***External Payment Processor*** so that customers may pay with debit or with credit cards. The ***External Payment Processor*** will automatically update necessary data in the ***Inventory Database***.

3.3 Diagram Key Explanation

Below is the Key for understanding the Software Architecture Diagram.

3.3.1 Diagram Key

3.3.2 Explanation

Understanding the Diagram Key is simple. Each element is outlined below.

Belongs To – The hollow arrow ***Belongs To*** refers to the item's property of being a *child* of the parent element it is directed toward.

Connects To – The solid arrow ***Connects To*** refers to the item's property of having an abstract connection to the item it is directed toward.

Can Directly Access – The dotted arrow ***Can Directly Access*** refers to the item's property of allowing the user to *directly access* data within a database.

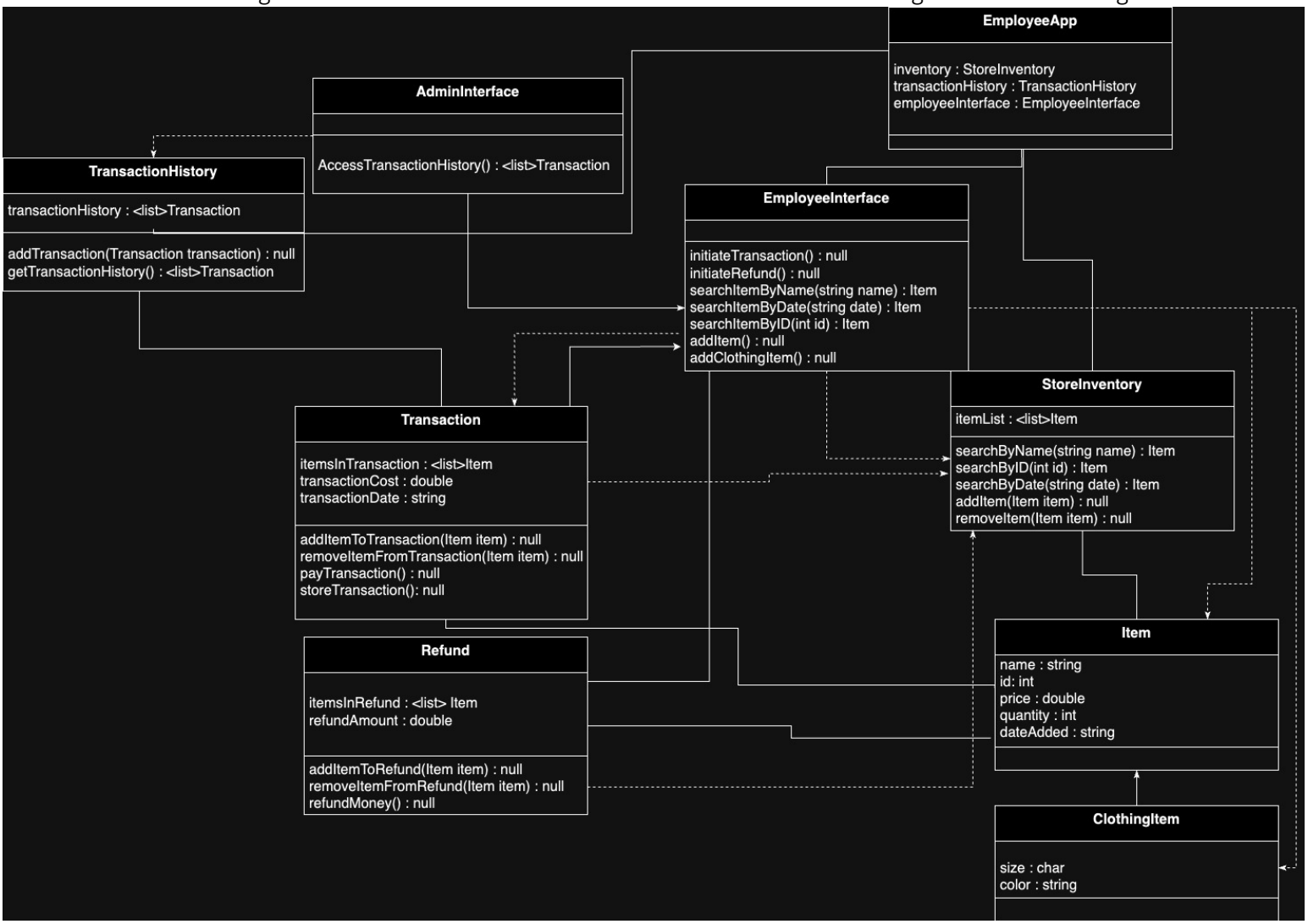
Is Software – The solid box ***Is Software*** refers to the item's property of being a software component.

Is Hardware – The dotted box ***Is Hardware*** refers to the item's property of being a hardware component.

Is a Database – The cloud-shaped box ***Is a Database*** refers to the item's property of being a distinct database.

4 UML Class Diagram

The following section covers the internal structure of the software through a UML Class Diagram.



4.1 EmployeeApp

The ***EmployeeApp*** class covers the point-of-sale aspect of the CSPoSS.

4.1.1 EmployeeApp Attributes

- ***inventory*** **:** ***StoreInventory***
- *transactionHistory* : ***TransactionHistory***
- *employeeInterface* : ***EmployeeInterface***

4.1.2 EmployeeApp Description

EmployeeApp is the class that encompasses the entire point-of-sale application on the employee's device. It has an association with the ***StoreInventory*** class, the ***TransactionHistory*** class, and the ***EmployeeInterface*** class. ***EmployeeApp*** has no operations.
The inventory attribute contains a ***StoreInventory*** object, representing the instantiated inventory of the store.
The ***transactionHistory*** attribute contains a ***TransactionHistory*** object, representing the instantiated list of transactions for the store.
The ***employeeInterface*** attribute contains an ***EmployeeInterface*** object, representing the employee's interface within the app.

4.2 EmployeeInterface

4.2.1 EmployeeInterface Operations

- ***initiateTransaction()*** : null
- ***initiateRefund()*** : null
- ***searchItemByName(string name)*** : Item
- ***searchItemByDate(string date)*** : Item
- ***searchItemByID(int id***) : Item
- ***addItem()*** : null
- ***addClothingItem()*** : null

4.2.2 EmployeeInterface Description

EmployeeInterface is a significant class as it is integral to the program-wide functionality. ***EmployeeInterface*** has no attributes, only operations. The operations contained within this class allow an employee to perform all client-specified employee tasks.

The ***initiateTransaction()*** operation creates a ***Transaction*** object, thereby opening a window for the employee to perform a transaction. This means ***EmployeeInterface*** is dependent upon the ***Transaction*** class, which is represented within the UML diagram. The return type is null because this operation doesn't need a value to be returned.

The ***initiateRefund()*** operation creates a Refund object, thereby opening a window for the employee to perform a refund. ***EmployeeInterface*** accesses and is depended upon the refund class, which is represented in the UML diagram. The return type is null because this operation doesn't need a value to be returned.

The ***searchItemByName()*** operation takes a string as an argument, and calls the ***StoreInventory*** class's ***searchByName()*** operation, passing the initial string argument. This method makes ***EmployeeInterface*** dependent upon ***StoreInventory***, which is reflected within the UML diagram. This method allows the employee to search the store's inventory by item name. The return type is an Item object because the search will return the item matched to the name.

The ***searchItemByDate()*** operation takes a string as an argument, and calls the ***StoreInventory*** class's ***searchByDate()*** operation, passing the initial string argument. This method makes ***EmployeeInterface*** dependent upon ***StoreInventory***, which is reflected within the UML diagram. This method allows the employee to search the store's inventory by item date. The return type is an Item object because the search will return the item matched to the date.

The ***searchItemByID()*** operation takes an int as an argument, and calls the ***StoreInventory*** class's ***searchByID()*** operation, passing the initial int argument. This method makes ***EmployeeInterface*** dependent upon ***StoreInventory***, which is reflected within the UML diagram. This item allows the employee to search the store's inventory by item ID. The return type is an Item object because the search will return the item matched to the ID.

The ***addItem()*** operation allows the employee to create an item and add it to the store's inventory. When called, the employee will be prompted to enter the item's name, id, price, and quantity. An item object will be instantiated with these fields and added to the store's inventory via the ***StoreInventory***'s ***addItem()*** operation, with the newly created item passed as an argument. This operation is dependent on the ***StoreInventory*** class, which is reflected in the UML diagram. The return type is null, because there is no value needed to be returned, this operation is merely a mutator.

The ***addClothingItem()*** operation allows the employee to create a clothing item and add it to the store's inventory. Much like the aforementioned ***addItem()*** operation, the employee is prompted to input the clothing item's name, id, price, and quantity, with the addition of size and color of the clothing item. This newly created ***ClothingItem*** is then passed to ***StoreInventory's addItem()*** method as an argument. The return type is null because this no value needs to be returned, this operation is merely a mutator.

4.3 StoreInventory

The StoreInventory class covers the storage of inventory items.

4.3.1 StoreInventory Attributes

- ***ItemList*** : <list>Item

4.3.2 StoreInventory Operations

- ***searchByName(string name)*** : Item
- ***searchByID(int id)*** : Item
- ***searchByDate(string date)*** : Item
- ***addItem(Item item)*** : null

- ***removeItem(Item item)*** : null

4.3.3 StoreInventory Description

The S***toreInventory*** class is representative of the store's item inventory and is an integral part of the application. ***StoreInventory*** allows employees as well as transactions and refunds to add items, remove items, and search items. The ***StoreInventory*** class holds an association with the Item class and the ***EmployeeApp*** class.

itemList is the only attribute. This is a dynamic list of Item objects, essentially the "inventory" itself.

searchByName() takes a string as an argument and searches itemList for an item with a name matching the argument. Returns an Item object, if one is found.

searchByID() takes an int as an argument and searches itemList for an item with an ID matching the argument. Returns an Item object, if one is found.

searchByDate() takes a string as an argument and searches itemList for an item with a date matching the argument. Returns an Item object, if one is found.

addItem() takes an Item object as an argument and appends it to the ***itemList***. Returns null, since it is merely a mutator.

removeItem() takes an Item object as an argument, searches the ***itemList*** for matching Item object, and removes it. Returns null, since it is merely a mutator.

4.4 Item

The Item class covers the attributes of each item stored within the store's inventory.

4.4.1 Item Attributes

- ***name*** : string
- ***id*** : int
- ***price*** : double
- ***quantity*** : int
- ***dateAdded*** : string

4.4.2 Item Description

The ***Item*** class represents a singular store Item. It has no operations; however, it has many attributes. The Item class holds an association with the ***StoreInventory*** class.
The ***name*** attribute contains a string for the item's name.
The ***id*** attribute is an int containing the item's unique identification number.
The ***price*** attribute is a double containing the item's price.
The ***quantity*** attribute contains an int for the number of items held.

The ***dateAdded*** attribute contains a string for the date the item was initialized and added to the inventory. This is automatically generated by the item's constructor using the programming languages operating system communication API.

4.5 ClothingItem

The ClothingItem class covers the specific attributes of clothing items within the store's inventory.

4.5.1 ClothingItem Attributes

- ***size*** : char
- ***color*** : string

4.5.2 ClothingItem Description

The ***ClothingItem*** class inherits from the ***Item*** class. It inherits all the previous Item attributes, while it also has two new attributes of its own. Much like the item class, it has no operations.
The ***size*** attribute contains a char of either 'S', 'M', or 'L' for small, medium, or large (the size of the clothing).
The ***color*** attribute contains a string for the color of the clothing.

4.6 Transaction

The Transaction class covers necessary operations and attributes to carry out a transaction.

4.6.1 Transaction Attributes

- ***itemsInTransaction*** : <list>Item
- ***transactionCost*** : double
- ***transactionDate*** : string

4.6.2 Transaction Operations

- ***addItemToTransaction(Item item)*** : null
- ***removeItemFromTransaction Item item)*** : null
- ***payTransaction()*** : null
- ***storeTransaction()*** : null

4.6.3 Transaction Description

The ***Transaction*** class is put to use whenever the employee initiates a transaction. It allows for the creation, modification, and eventual completion of a transaction of items to the customer, in exchange for the customer's payment. The ***Transaction*** class has an association with the ***TransactionHistory*** class, the ***Item*** class, and the ***EmployeeInterface*** class. It has a dependency on the ***StoreInventory*** class, as reflected in the UML diagram.
The ***itemsInTransaction*** attribute contains a dynamic list of ***Item*** objects currently held in the ongoing transaction.
The ***transactionCost*** attribute contains a double that holds the total cost of the transaction, which is updated each time an item is added or removed.
The ***transactionDate*** attribute contains a string which holds the date the transaction is created.
The ***addItemToTransaction()*** operation takes an ***Item*** object as an argument and adds it to the list of items within the transaction. The ***Item*** object that is passed is derived from the inventory

either by the employee scanning it with their handheld device, or inputting the item's ID. When an item is added, the ***transactionCost*** attribute is updated to reflect the addition of the new item's price. The return type is null because the operation is a mutator.
The r***emoveItemFromTransaction()*** operation takes an ***Item*** object as an argument, matches the passed item object to one within the transaction's item list, and if found, removes the ***Item*** object from the list. The ***transactionCost*** attribute is updated to reflect the lower price after removing the item. The return type is null because the operation is a mutator.
The ***payTransaction()*** operation prompts the employee with two options to choose from: pay with credit/debit or pay with cash. If credit/debit is chosen, the payment is routed through a call to the external payment API. If cash is chosen, the employee performs the transaction manually, and physically exchanges the cash into the cash register. The return type is null because we do not require any sort of return value from the operation. The ***storeTransaction()*** operation is called once payment is completed.
The ***storeTransaction()*** operation closes out the transaction window on the device and stores the transaction using the ***addTransaction()*** operation from the ***TransactionHistory*** class, passing itself as the transaction to be stored.

4.7 Refund

The Refund class covers the necessary operations and attributes to carry out a refund.

4.7.1 Refund Attributes

- ***itemsInRefund*** : <list>Item
- ***refundAmount*** : double

4.7.2 Refund Operations

- ***addItemToRefund(Item item)*** : null
- ***removeItemFromRefund(Item item)*** : null
- ***refundMoney()*** : null

4.7.3 Refund Description

The ***Refund*** class can be used by the employee to refund a customer's money. It allows the employee to scan multiple items to be refunded and process the refund. The ***Refund*** class holds an association with the ***Item*** class and the ***EmployeeInterface*** class.
The ***itemsInRefund*** attribute contains a dynamic array of Item objects that holds the items currently stored in the refund.
The ***refundAmount*** attribute contains a double that holds the amount set to be refunded. This is updated whenever items are added or removed from i***temsInRefund***.
The ***addItemToRefund()*** operation takes an ***Item*** object as an argument. Similar to the ***Transaction*** class, the item is either scanned, or the employee can enter the item id to add it to the ***itemsInRefund*** list. The ***refundAmount*** attribute is updated to reflect the new amount of money to be refunded with the addition of the item. The return type is null because the operation is a mutator.
The ***removeItemFromRefund()*** operation takes an Item object as an argument. When the item is scanned or the item's ID is inputted by the employee, the item is removed from the

itemsInRefund list. The ***refundAmount*** attribute is updated to reflect the new amount of money to be refunded. The return type is null because the operation is a mutator.
The ***refundMoney()*** operation completes the refund and performs the transfer of money. The employee is prompted with two options, either refund via credit/debit or refund with cash. If the employee chooses credit, an API call to the external payment system is made to issue a refund. If cash is chosen, the employee must manually remove cash from the register and give it to the customer.

4.8 AdminInterface

The AdminInterface class covers the unique operation that an Admin can perform.

4.8.1 AdminInterface Operations

- ***accessTransactionHistory()*** : <list>Transaction

4.8.2 AdminInterface Description

AdminInterface inherits from employee interface and is only available to employees who are designated admins. The class has no attributes and one operation. It has a dependency on the ***TransactionHistory*** class, because it accesses it.
The ***accessTransactionHistory()*** operation makes a call to ***getTransactionHistory()*** in the ***TransactionHistory*** Class. ***AdminInterface*** is the only class with permission to access ***TransactionHistory***. The call to ***getTransactionHistory()*** returns a dynamic array of ***Transaction*** objects to ***accessTransactionHistory***, which then returns that same array to the admin who called the operation. The return type is a dynamic array of ***Transaction*** objects.

4.9 TransactionHistory

The TransactionHistory class represents the storage of TransactionHistory data for the clothing store.

4.9.1 TransactionHistory Attributes

- ***transactionHistory*** : <list>Transaction

4.9.2 TransactionHistory Operations

- ***addTransaction(Transaction transaction)*** : null
- ***getTransactionHistory()*** : <list>Transaction

4.9.3 TransactionHistory Description

The ***TransactionHistory*** class stores every transaction performed using the point-of-sale system. It holds an association with the ***EmployeeApp*** class and the ***Transaction*** class. When ***TransactionHistory*** is constructed at the start of the application, the transaction history day is pulled from the database to be stored in this ***TransactionHistory*** instance.
The ***transactionHistory*** attribute holds a dynamic array of ***Transaction*** objects. This serves as the stored list of transactions.
The ***addTransaction()*** operation takes a ***Transaction*** object as an argument and appends it to the ***transactionHistory*** list. The return type is null because the operation is merely a mutator.
The ***getTransactionHistory()*** operation gets the transactionHistory list and returns it. This exists simply to allow the ***AdminInterface*** class to retrieve the transaction history.

5 CSPoSS Development Timeline

The following section covers the anticipated timeline for developing the CSPoSS software system.

5.1 Quarter 1 Front-End Development

- The sales process begins to facilitate purchases from customers that support cash, card, and mobile payment
- Inventory Management in real-time with reporting and analytics for product trends and other important sales data

5.2 Quarter 2 Curating the employee backend and updating the security

- Employee access to customer services such as assistance payment transactions, additional deductions or discounts
- Tracking for employee work hours, sales performances, and statistics
- Return and Exchange feature added for employees to facilitate product returns from customer

5.3 Quarter 3 Upgrade to Service

- Loyalty Program to allow customers to accumulate points and discounts.
- Expand systems to enable multiple store chains to access data synchronously
- Implement security for different level access of the hierarchy of business command

5.4 Quarter 4 UX Design

- Increase the Design Features on the customer interface
- Increase the endergonic features of the employee interface
- Integration with additional payment methods and emerging technologies.

6 Verification

The following section covers test cases for our design on three levels of granularity: Unit, Integration, and System.

6. 1 Unit Test Cases

6.1.1 EmployeeInterface Operations

- initiateTransaction()
  - Input: call **initiateTransaction()**
  - Expected Output: A new transaction window is opened.

- initiateRefund()
  - Input: call **intiateRefund()**
  - Expected Output: A new refund window is opened.

- searchItemByName()

  Note: This function must be hard-coded with a pre-set item to test the logic and make no call to the ***StoreInventory*** class

  - Case: Item Exists
    - Input: "Shirt"
    - Expected Output: Returns the matching ***Item*** object named "Shirt"

  - Case Item Does Not Exist
    - Input: "Toy"
    - Expected Output: Returns null.

- searchItemByDate()
  Note: This function must be hard-coded with a pre-set item in order to test the logic and make no call to the ***StoreInventory*** class

  - Case: Item Exists
    - Input: "Mar 10 2023"
    - Expected Output: Returns the matching ***Item*** object added on March 10, 2023

  - Case: Item Does Not Exist
    - Input: "Mar 10 1900"
    - Expected Output: Returns null.

- searchItemByID()
  Note: This function must be hard-coded with a pre-set item in order to test the logic and make no call to the ***StoreInventory*** class

  - Case: Item Exists
    - Input: 239403948
    - Expected Output: Returns the ***Item*** with matching ID.

  - Case: Item Does Not Exist
  - Input: 000000000
  - Expected Output: Returns null.

- addItem()
  Note: This function must be hard-coded with a pre-set item in order to test the logic and make no call to the ***StoreInventory*** class
    - Input: Create a new ***Item*** object
    - Expected Output: The ***Item*** object is added to the ***StoreInventory***

- addClothingItem()
  Note: This function must be hard-coded with a pre-set item in order to test the logic and make no call to the ***StoreInventory*** class
    - Input: Create a new ***ClothingItem*** object
    - Expected Output: The Item object is added to the ***StoreInventory***

6.1.2 StoreInventory Operations

- searchByName()

  - Case: Item Does Not Exist
    - Input: "Seesaw"
    - Expected Output: Returns null.

- searchByID()

  - Case: Item Exists
    - Input: 239403940
    - Expected Output: Returns the matching ***Item*** object.

  - Case: Item Does Not Exist
    - Input: 000000000
    - Expected Output: Returns null.

- searchByDate()

  - Case: Item Exists
    - Input: "Mar 10 2023"
    - Expected Output: Returns the matching Item object.

  - Case: Item Does Not Exist
    - Input: "Mar 10 1900"
    - Expected Output: Returns null.

- addItem()
    - Input: Create a new ***Item*** object
    - Expected Output: The ***Item*** is added to ***StoreInventory***

- removeItem()

  - Case: Item Exists
    - Input: Call **removeItem()** on an ***Item*** in ***StoreInventory***
    - Expected Output: ***Item*** is no longer in ***StoreInventory***

  - Case: Item Does Not Exist
    - Input: Call **removeItem** on an ***Item*** that is not in ***StoreInventory***
    - Expected Output: No change is made to ***StoreInventory***

6.1.3 Transaction Operations

- addItemToTransaction()

    - Input: Call **addItemToTransaction()** on an ***Item***.
    - Expected Output: ***TransactionCost*** is updated accurately.

- removeItemFromTransaction()

    - Input: Call **removeItemFromTransaction()** on an **Item**.
    - Expected Output: ***TransactionCost*** is updated accurately.

- payTransaction()

    - Input: Call **payTransaction()**
    - Output: ***storeTrasaction()*** is called

- storeTransaction()

    - Input: Call **storeTransaction()**
    - Expected Output: ***Transaction*** is appended to ***transactionHistory*** list.

.

6.1.4 Refund Operations

- addItemToRefund()
    - Input: Pass an Item to **addItemToRefund()**
    - Expected Output: The ***Item*** object is added to ***itemsInRefund*** and ***refundAmount*** is updated accurately.

- **removeItemFromRefund()**
    - Input: Pass an ***Item*** to **removeItemFromRefund().**
    - Expected Output: The ***Item*** object is no longer in ***itemsInRefund*** list and ***refundAmount*** is updated accurately.

- **Note**: Operation **refundMoney()** is not included in the list of Unit tests because the logic and output is held within a third-party program

6.1.5 AdminInterface Operations

- **accessTransactionHistory()**
  Note: Because this operation is dependent on the ***TransactionHistory*** class, in order to perform a unit test we must first abstract away any implementation of the ***TransactionHistory*** class and instead operate on a predefined list
    - Input: Call **accessTransactionHistory()**
    - Expected Output: Our predefined list is returned.

6.1.6 TransactionHistory Operations

- **addTransaction()**
    - Input: Call **addTransaction()** with a ***Transaction*** object as its parameter.
    - Expected Output: The ***Transaction*** object is appended to the ***transactionHistory*** list.

- **getTransactionHistory()**
    - Input: Call **getTransactionHistory()**
    - Expected Output: Return ***transactionHistory*** list.

6.2 Integration Testing Cases

Preface: The tests outlined below are intended to verify every functionality that relies upon multiple classes or application components. Every functionality listed is dependent upon the successful performance of two or more application components.

6.2.1 Search Item by Name

This test verifies the functionality of the related searchItemByName() methods located within the EmployeeInterface and StoreInventory classes. The functionality is intended to allow the employee to access the store's inventory component to search for an item by its name.

- Case: Item Exists in StoreInventory
    - Input: "Shirt"
    - Expected Output: Item object named "Shirt" is found in StoreInventory and returned

- Case: Item Doesn't Exist in StoreInventory
    - Input: "Toy"
    - Expected Output: Returns null, since no Item object named "Toy" found in the Store's Inventory

6.2.2 Search Item by Date

This test verifies the functionality of the related searchItemByDate() methods located within the EmployeeInterface and StoreInventory classes. The functionality is intended to allow the employee to access the store's inventory component to search for an item by its date.

- Case: Item Exists in StoreInventory
    - Input: "Oct 15 2023"
    - Expected Output: Item object dated "Oct 15 2023" found in StoreInventory and returned
    -
- Case: Item Doesn't Exist in StoreInventory
    - Input: "Jan 20 2005"
    - Expected Output: Returns null, since no Item object dated "Jan 20 2005" found in Store's Inventory

6.2.3 Search Item by ID

This test verifies the functionality of the related searchItemByID() methods located within the EmployeeInterface and StoreInventory classes. The functionality is intended to allow the employee to access the store's inventory component to search for an item by its ID number.

- Case: Item Exists in StoreInventory
    - Input: 195206594
    - Expected Output: Item object with ID 195206594 found in StoreInventory and returned

- Case: Item Doesn't Exist in StoreInventory
    - Input: 123456789
    - Expected Output: Returns null, since no Item object with ID 123456789 found in StoreInventory

6.2.4 Add Item

This test verifies the functionality of the related ***addItem***() methods located within the EmployeeInterface and StoreInventory classes. The functionality is intended to allow the employee to add an item to the store's inventory component.

- Input: A newly created Item object
- Expected Output: No value returned, the inputted Item object is added to StoreInventory.

6.2.5 Add Clothing Item

This test verifies the functionality of the related addClothingItem() methods located within the EmployeeInterface and StoreInventory classes. The functionality is intended to allow the employee to add a clothing item to the store's inventory component.

- Input: A newly created ***ClothingItem*** object
- Expected Output: No value returned, the inputted ***ClothingItem*** object is added to ***StoreInventory***.

6.2.6 Access Transaction History

This test verifies the functionality of the accessTransactionHistory() method located within the AdminInterface class. This function is dependent on the TransactionHistory component of the application.

- Input: N/A
- Expected Output: The ***TransactionHistory*** component is accessed, and a list of previous transactions is displayed to the admin

6.2.7 Add Item to Transaction

This test verifies the functionality of the addItemToTransaction() method located within the Transaction class. This function is dependent on the StoreInventory component of the application

- Input: An ***Item*** object from within the ***StoreInventory*** component
- Expected Output: No value returned, the passed ***Item*** object is removed from ***StoreInventory*** and added into the transaction.

6.2.8 Remove Item from Transaction

This test verifies the functionality of the removeItemFromTransaction() method located within the Transaction class. This function is dependent on the StoreInventory component of the application.

- Input: An ***Item*** object contained within the current transaction
- Expected Output: No value returned, the passed ***Item*** object is removed from current transaction and placed into ***StoreInventory***

6.2.9 Store Transaction

This test verifies the functionality of the storeTransaction() method located within the Transaction class. This function is dependent on the TransactionHistory component of the application.

- Input: N/A
- Expected Output: Transaction process is closed and transaction is stored within ***TransactionHistory*** component

6.2.10 Add Item to Refund

This test verifies the functionality of the addItemToRefund() method located within the Refund class. This function is dependent on the StoreInventory component of the application.

- Input: An ***Item*** object from within the ***StoreInventory*** component
- Expected Output: No value returned, the passed ***Item*** object is removed from ***StoreInventory*** and added into the refund.

6.2.11 Remove Item from Refund

This test verifies the functionality of the removeItemFromRefund() method located within the Refund class. This function is dependent on the StoreInventory component of the application.

- Input: An ***Item*** object contained within the current refund
- Expected Output: No value returned, the passed ***Item*** object is removed from current refund and placed into ***StoreInventory***.

6.3 System Testing Cases

6.3.1 Functional Testing

- Barcode Scanning
  - **Case:** Barcode Scanning
    - **Input:** Employee uses the device's barcode scanner to scan an item barcode.
    - **Expected Output:** The system accurately recognizes the item, retrieves its information and displays it on the cash register.

- Login Component Success
  - I. **Case:** Employee Login
    - **Input:** Employee enters credentials and attempts to log in.
    - **Expected Output:** The employee gains access to the Employee Interface if the login credentials is correct.
- II. **Case:** Successful Admin Login
  - **Input:** The administrator gains access to the administrator Interface.

        - **Expected Output:** The admin gains access to the Admin Interface. The admin gains access to the Employee Interface if the login credentials is correct.

- Inventory Access
    - **Case:** Accessing Inventory
        - **Input:** Employee attempts to access the inventory by searching for an item.
        - **Expected Output:** The employee can view the inventory data without issues displaying the price, the stock, the detail, and all recent transactionhistory related to searched item.

- Inventory Modification
    - **Case:** Inventory Update
        - **Input:** Employee updates an item in the inventory by either selling the item, adding, or subtracting the item from the inventory.
        - **Expected Output:** The inventory reflects the changes accurately that are left in stock of the respected item.

- Payment Processing
    - **Case:** Successful Payment Processing
        - **Input:** Various payment methods such as through Venmo, Google Pay, Amazon Pay, or Card transactions that will be processed through the External Payment Processor or Cash Register.
        - **Expected Output:** Payments are completed successfully using the chosen methods if the payment method is valid.

- Refund Processing
    - **Case:** Refund
        - **Input:** Employee initiates a refund process, which deducts the refund amount from revenue and adds items back to stock.
        - **Expected Output:** Refunds are processed accurately, reflecting revenue deductions and item restocking.

6.3.3 Performance Testing

- Transaction Volume
    - **Case:** Maximum Transaction Volume
        - **Input:** Stress testing to identify the maximum number of transactions the system can handle within a specified time frame should be up to the number of tables or cashiers within a given store.

        - **Expected Output:** Determine the system's capacity for handling transactions. A valid number of transactions should be equal to or above 1 over the expected cashier number in the physical store.

6.3.4 Security Testing

- Data Access
    - I. **Case:** Employee Data Access
        - **Input:** Employee attempts to access admin data.
        - **Expected Output:** Security measures prevent unauthorized access to admin data if credentials are invalid.
    - II. **Case:** Inventory Protection
        - **Input:** Employee attempts to tamper with inventory data.
        - **Expected Output:** Safeguards prevent unauthorized modifications to inventory by only allowing admin to access stocks.

- Fraud Detection
    - **Case:** Fraudulent Transaction
        - **Input:** External Payment Processor detects a fraudulent or invalid payment transaction.
        - **Expected Output:** The system identifies and handles fraudulent or invalid payment transactions securely and display invalid payment info on cash register.

6.3.5 Usability Compatibility Testing

- Wi-Fi Environment Usage
    - **Case:** Usability in Wi-Fi Environment
        - **Input:** The system is used in a store with Wi-Fi coverage, including front desk and customer areas.
        - **Expected Output:** The systems functions smoothly in areas with WiFi.

6.3.6 Compatibility Testing

- Platform Compatibility
    - I. **Case:** iOS Compatibility
        - **Input:** The Employee App is tested on iOS devices.
        - **Expected Output:** The Employee App functions seamlessly on iOS.
    - II. **Case:** Android Compatibility
        - **Input:** The Employee App is tested on Android devices.
        - **Expected Output:** The Employee App functions seamlessly on Android.

6.3.7 Regression Testing

- Software Updates

- **Case:** Update in Employee and Employee Interface
  - **Input:** Software updates are applied to Employee and Employee Interface components.
  - **Expected Output:** Updates do not adversely affect inventory or payment processing functionality.

6.3.8 Recovery Testing

- Data Recovery
  - **Case:** Data Recovery
    - **Input:** Test the system's ability to recover data in case of unexpected failures.
    - **Expected Output:** All lost data, including transaction history and inventory, can be successfully restored.

6.3.9 Installation and Configuration Testing

- Installation
  - **Case:** Laptop, Tablets Computer Installation
    - **Input:** Attempt to install and configure the software on laptops, tablets, and desktop computers.
    - **Expected Output:** Successful installation and configuration on these all devices

6.3.10 Documentation Testing

- Documentation Accuracy
  - I. **Case:** Documentation Verification
    - **Input:** Review the provided documentation regarding historical transaction data of any buying transaction.
    - **Expected Output:** Ensure documentation is complete, accurate, and comprehensive of all transaction history.

7 Data Management

The following outlines two databases for storing data in the CSPoSS--one for the Store Inventory, and another for the Transaction History. They are split in two because Store Inventory and Transaction History don't interact, thus it would be better to isolate both databases into two for both simplicity and security's sake.

7.1 Store Inventory Database

The Inventory Database is a SQL database consisting of one table, "Product Table". The primary key for items in the Product table has the ProductID -- a unique 9-digit number for each unique item. The Table also includes index lists ProductNames, ProductIDs, DatesAdded, and Prices.

These index lists allow for quicker lookup time. Since products are searchable by name, ID, date added, and price, having each of these traits in a sorted index list allows for products to be searched in O(logn) runtime using binary search., rather than in O(n) time using linear search.

Product Table				
ProductID (Int)	ProductName (String)	ProductQuantity (Int)	DateAdded (String)	ProductPrice (Double)
904830492	Maxi Dress	8	Mar 10 2023	$20.99
959930493	Frilly Socks	23	Jan 05 2021	$12.99
948392857	Pleated Skirt	5	Jul 30 2022	$15.99
948730284	Winter Gloves	12	Oct 04 2023	$11.99

7.2 Transaction History Database

Transaction Table				
transaction_id (int)	items_in_transaction (int[])	item_amount (int)	transaction_date (string)	transaction_cost (double)
000000000	[158690150, 158690151]	2	“Feb 19, 2023”	$6.97
000000001	[…]	4	“Mar 8, 2023”	$12.82
000000002	[…]	17	“Jul 15, 2023”	$84.35
000000003	[…]	9	“Oct 4, 2023”	$41.11

Item Table		
item_id (int)	item_name (string)	item_price (double)
158690150	"Bananas"	$3.42
158690151	"Milk"	$3.55
158690152	"T-Shirt"	$11.99
158690153	"Baking Soda"	$6.87

The Transaction History database is an SQL database that consists of two tables: *Transaction* and *Item*.

The *Transaction* table contains five fields: transaction_id, items_in_transaction, item_amount, transaction_date, transaction_cost. Each field's data type is shown within the heading box of the field. Note: The "items_in_transaction" field contains an array of Item ID's in the form of integers. The added columns are included as examples of how the actual data would look.

The table contains index lists associated with transaction_id, transaction_date, and transaction_cost. These index lists allow for fast and efficient lookup of transactions by transaction ID, date, or cost.

The *Item* table contains three fields: item_id, item_name, item_price. As with the *Transaction* table, the data type of each field is included within the heading box of the field, and the additional columns are included as examples.

This table has index lists for all three fields.

SQL was chosen for this database for a variety of reasons. First, the data models for this application are not overly complex, and therefore it was simpler and more cost effective to use SQL. The robust querying interface associated with SQL allows us to write highly effective queries into our application. Additionally, the widely available resources and support for SQL databases will save us time in building the application.