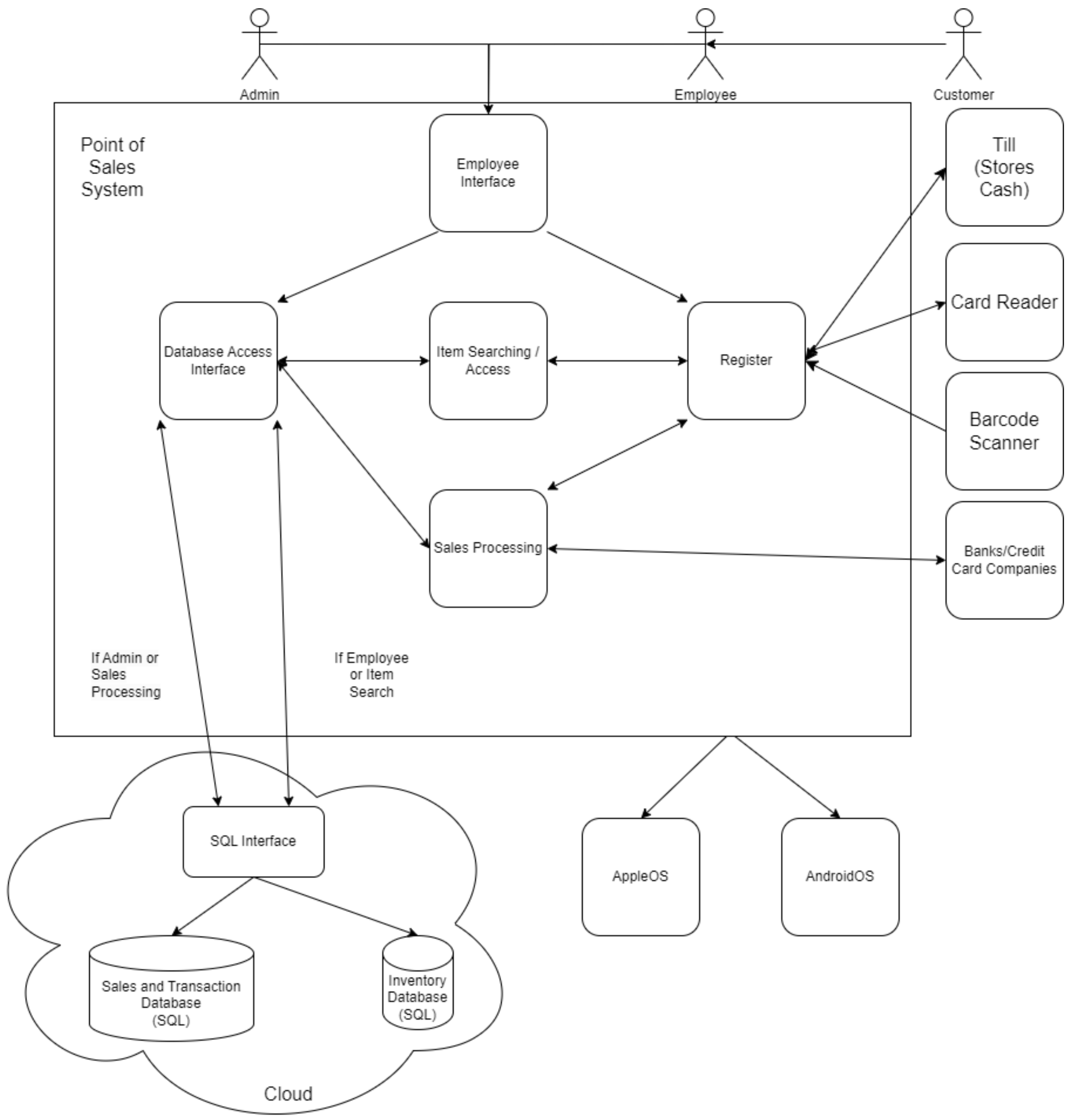
Clothing Store Point of Sale System

Aria Kafie, Nomar Salom, Anka Engin

Brief Overview:

The Clothing Store Point of Sale System is a system designed to manage and keep track of clothing store transactions, including purchases and returns. This system should keep track of and continuously update an inventory database, which should be searchable by employees. Employees should be able to use the system to check out items, and the system should accept payment via credit card, debit card, or cash. Item identification should be possible via a barcode scan or a manual ID entry, thus, devices with built in cameras like Android or Apple machines should be supported by the system. Additionally, transaction history should be stored in a database separate from the inventory, and this data should be secure and accessible only to administrative users.

Architectural Diagram



Difference between Assignment 2 and current SWA diagram:
We combined the Sales Database and Transaction Database into one since we feel that there are many relational similarities between the two. We also identified what types of databases we are using as well as added another interface to access the different databases that are separate from each other. We do not have to change anything else with our SWA diagram since our system accesses databases through an interface, meaning that there does not need to be any changes in connection from parts of the system to our databases.

SWA Diagram Description:
Customers will interact with employees for anything to do with this system.

Employee Interface:

- Both employees and administrators will access the system via the Employee Interface terminal. They will login to the system with their credentials that indicates if they have administrator privileges or not.
- From there, any user can pick between the Database access interface or the regular register interface
- Both Administrators and Employees can do the same tasks, but when it comes to database access, Administrators have more permissions in viewing different databases.

Database access interface:

- Employees will only have access to the inventory database.
- Administrators will be able to access both databases. The new databases they will be able to access include the Sales and Transactions Database.
- System will also use this interface to access the database
    - Item Search will only be able to peak at objects in inventory database
    - Sales processing can view and/or change values in all databases in cloud

Cloud:

- The cloud will contain the databases needed for this system. These databases include
    - Inventory
        - People who have access can view or edit values pertaining to specific items a store has. Values include item IDs, color, size, and inventory amount.
        - This information is stored in a SQL database.
    - Sales and Transaction
        - People who have access can view the history of sales, and see trends of what has been sold and profits of the store(s).
        - People who have access can view the history of transactions including payment history and sensitive information pertaining to customers.
        - This information is stored in a SQL database.

Register:

- The register is connected to the till (box that keeps the cash), a card reader, and a barcode scanner.
- Through the register, a user can access item searching/access as well as sales processing
    - Takes manual code or barcode scan and sends information to item searching/access to fetch item information.
    - Takes items in checkout and calculates totals and tax. Information is then sent to the sales processing function.

Item Search/Access

- Looks up items and returns information about an item through the Database Access Interface. Items can be imputed manually through the register, or takes the barcode from the scanner and translates it into the associated ID. After an item is found in the database, item information is sent back to register.

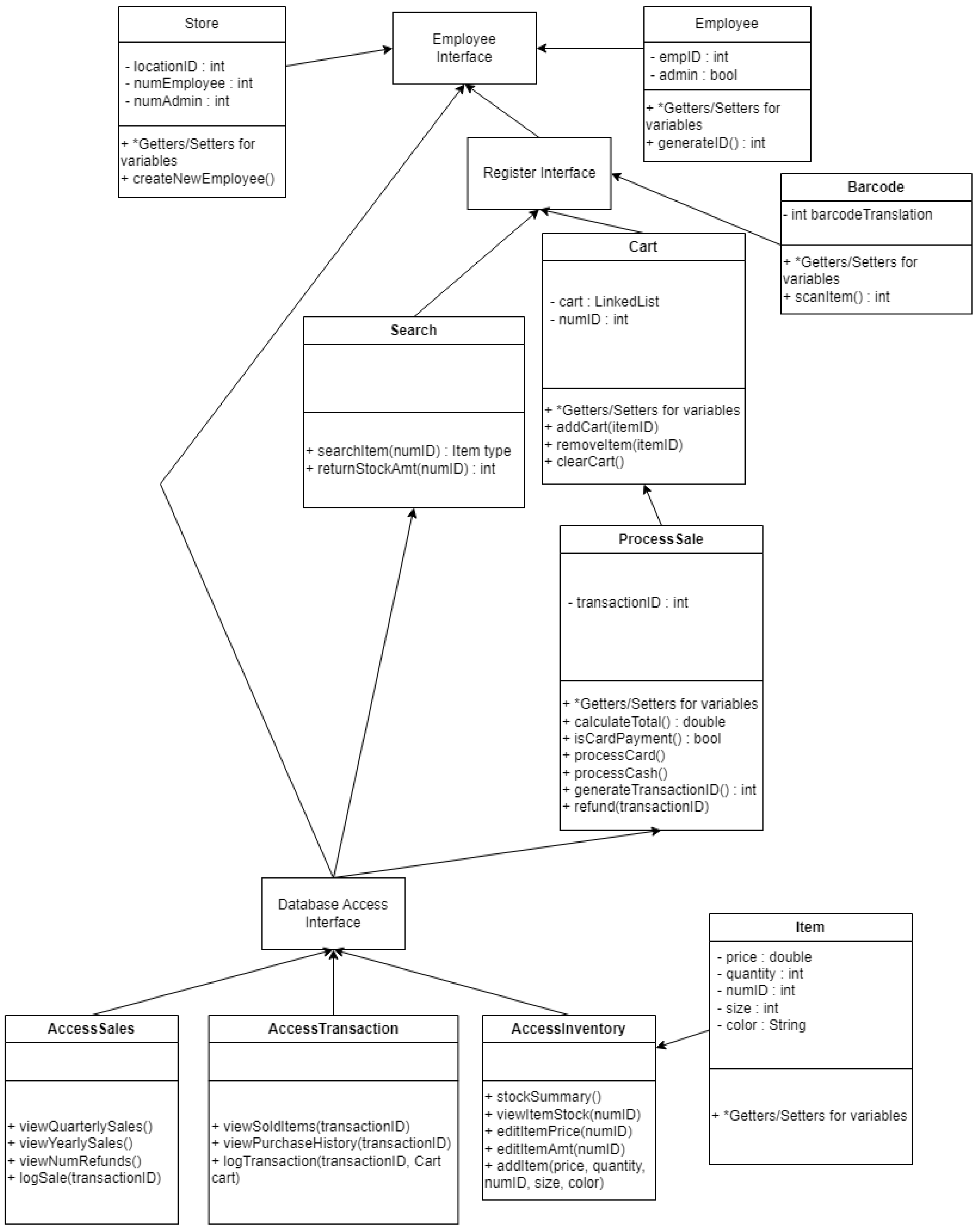
Sales Processing

- Takes information from register to update databases through Database Access Interface
    - Updates inventory amounts
    - Updates sales for items sold
    - Updates transactions for the specific transaction that just took place
- Connects with bank if credit/debit card is used
- Tells the register how much cash is owed or given back depending on what values it receives from the register.

Different Operating Systems

- Written in a way for AppleOS to accept/understand
- Written in a way for AndroidOS to accept/understand

UML Diagram:


Store
- locationID : int
- numEmployee : int
- numAdmin : int
+ *Getters/Setters for variables
+ createNewEmployee()
Employee Interface
Employee
- empID : int
- admin : bool
+ *Getters/Setters for variables
+ generateID() : int
Register Interface
Barcode
- int barcodeTranslation
+ *Getters/Setters for variables
+ scanItem() : int
Cart
- cart : LinkedList
- numID : int
+ *Getters/Setters for variables
+ addCart(itemID)
+ removeItem(itemID)
+ clearCart()
Search
+ searchItem(numID) : Item type
+ returnStockAmt(numID) : int
ProcessSale
- transactionID : int
+ *Getters/Setters for variables
+ calculateTotal() : double
+ isCardPayment() : bool
+ processCard()
+ processCash()
+ generateTransactionID() : int
+ refund(transactionID)
Database Access Interface
Item
- price : double
- quantity : int
- numID : int
- size : int
- color : String
+ *Getters/Setters for variables
AccessSales
+ viewQuarterlySales()
+ viewYearlySales()
+ viewNumRefunds()
+ logSale(transactionID)
AccessTransaction
+ viewSoldItems(transactionID)
+ viewPurchaseHistory(transactionID)
+ logTransaction(transactionID, Cart cart)
AccessInventory
+ stockSummary()
+ viewItemStock(numID)
+ editItemPrice(numID)
+ editItemAmt(numID)
+ addItem(price, quantity, numID, size, color)

UML Diagram Description:

Employee Interface:

- Has access to “Store” and “Employee” class, as well as “Register” and “Database Access” interface.

Store Class:

- Used for store information and to create new employee profiles for stores.

Employee Class:

- Used for employee information.

Register Interface:

- Has access to the “Search”, “Cart”, and “Barcode” class.

Search Class:

- Used to search the Inventory database for items and returns it to the register interface.
- Accesses the “Database Access” interface to search the Inventory database.

Barcode Class:

- Used to translate barcode scan to item ID.

Cart Class:

- Used to keep track of what is in a customer’s cart.
- Uses a linked list to keep track of items.
- Utilizes the “ProcessSale” class to calculate totals and process sales made to the customer.

ProcessSale Class:

- Used to process a sale and calculate the total owed to store and tax.
- Differentiates between cash and card payments.
- Process card transactions with banks
- Accesses the “Database Access” interface to log sales and transactions data into the “Sales” and “Transactions” database.

Database Interface:

- Has access to the “AccessSales”, “AccessTransactions”, and “AccessInventory” class.

AccessSales Class:

- Can view and log sales into the “Sales” database.

AccessTransactions Class:

- Can view and log transactions into the “Transaction” database.

AccessInventory Class:

- Can view, edit, add, and delete items in the “Inventory” database.
- Has access to the “Item” class.

Item Class:

- Used to store information about clothing items available for purchase.

Partitioning of Tasks:

Out of the three person team, one would be in charge of:

System Architecture and Interface Development

- Overall system architecture
- Interface for employees
- Interface for database
- Connection to cloud-based databases
- Storage and retrieval

Register and Hardware Integration

- Register interface
- Hardware component connections
    - Scanner
    - Card Reader
    - Cash Till
    - Item searching/access
- Sales calculations
- Code entry

Item Management and Sales Processing

- Item class and its functions
- Item data
- Sales processing functions
- Inventory updates

Bank connection

- Team Collaboration
- Compatibility with AppleOS and AndroidOS

Potential Timeline:

With all responsibilities in mind and the current requirements, this project will take approximately one month, but will take more time if changes are made.

Test Cases:

Test set #1:

Targeted feature: Database Read/Write

Unit test:

```
while(get_input() != “quit”) {
expectedOutput = get_string_from_user()
Item1.setColor(expectedOutput)
if (Item1.getColor() == expectedOutput)
        log(“Item.getColor test passed”)
else
        log(“Item.getColor test failed”)
}
```

This pseudocode represents a unit test for the Item class, which is the smallest component of the system’s database. In this case, the Item class’s getter and setter methods for the ‘color’ field are being tested using a random data selection method, whereby the user is prompted in a loop of arbitrary duration to enter arbitrary string values that are all tested, evaluated, and instantly logged one after another. In addition to testing Item’s get and set methods, this test also verifies the integrity and functionality of the ‘color’ field itself. An extension of this test would be to include similar tests for all other get and set methods of the Item class, as well as their associated fields. In this case, the test vector would be the set of all inputs for Item’s set methods, as well as the corresponding set of all outputs for the get methods immediately following calls to set. This test function would cover failures such as type mismatches between function return types and user input, as well as logical bugs regarding the functionality of get and set methods of the Item class.

Functional/Integration test:

```
Item testItems[test_iterations]
populate_with_random_data(testItems);
for i in range test_iterations:
        Inventory.addItem(testItems[i])
for i in range test_iterations:
    If Inventory.viewItem(testItems[i].getID()) != testItems[i]
        log("Item " + testItems[i] + "failed.\n")
```

This functional test's primary target is the 'AccessInventory' component of the system's architecture. In this example, the test vector consists of an array of randomly generated Item values, which are to be tested against themselves once they are passed into and subsequently retrieved from the databases's 'AccessInventory interface via view and edit methods. This test verifies the functionality of viewItemStock, editItemPrice, editItemAmt, and addItem, because if any of these were to fail, all of the subsequent test comparisons in the array loop would cause error messages to be logged. This is because each Item object that was initially added to the Inventory Interface is revisited and tested in the original order it was added. Additionally, because of the nature of the comparison (!=) operator overload specific to this test case, a runtime crash during this test would bring light any previously undetected data type mismatches between the fields of the item object, and the return types of any and all 'view' methods in the inventory interface. Because of the random nature of the data being used as inputs for this test case, this test relies heavily on consistently uniform data distributions in order to sufficiently cover the range of all possible inputs that a user would be able to enter. This test would cover any failures regarding incorrect access specifiers in both the Item and AccessInventory class. This is because, in the case that a private/inaccessible method or field is being accessed from outside either of the two classes, the code would not compile, and would alert the developers to the issue.

System test:

```
employee e = new employee()
e.login()
sale s
e.processSale(s)
```

This test aims to simulate the control flow that a typical user would follow while using the point of sale system. First, a new employee is created to simulate an already existing employee. Using this employee to login into the system allows for coverage of system failures involving all methods and fields of the employee class, as well as the access specifiers of each of these components. By logging in, the interface that mediates employee/store interactions is also tested, which allows for the detection of errors involving the violation of read-only protocols regarding database entries and search functionality. Using different types of employees to test employee-database interactions would also allow for verification of the administrator system that prohibits certain users from reading or writing to certain parts of the system, such as Item objects in the database. By calling processSale from the user object, this test verifies the functionality of the processSale class's methods and fields, as well as its ability to interact with a user and read data from the Item database.

Test set #2

Targeted feature: Register

Unit Test:

```
bool purchaseFinished;
if(purchaseFinished && purchase.isCardPayment()){
        processCard();
        if(Integer.PaseInt(creditCardNum) != “error” && (string)creditCardNum.length == 16){
                if(creditCardBalance -= purchase.calculateTotal() >= 0){
                        generateTransactionID();
                }
                else{
                        System.out.println(“Insufficient funds”);
                        [revert back to checkout screen]
                }
        }
        else{
                System.out.println(“Error reading card”);
                [revert back to checkout screen]
        }
}
```

For my unit test, I decided to primarily target the ProcessSale class, which was one of the smallest and most specific components of the system’s register interface. In this test, the validity of the customer’s credit card number is being tested. First, I needed to verify that the purchase was complete, and that the employee scanned all the items that the customer had requested. Once the employee pressed continue to checkout, then purchaseFinished would become true. Next I needed to check that the customer requested to pay with a card. If both of these are true, then the card can be processed. Once the card is processed and the card’s number is retrieved, then the system can test the validity of the card. It does this by checking if the card’s number contains only valid numbers, and if the length of the card’s number is 16 digits. If these aren’t true, it returns back to the checkout screen. If they are true, then the system sees if the card’s balance has enough money for the purchase. If it doesn’t it returns back to the checkout screen. If it does, a transaction ID is made and the transaction is complete. This test covers failures involved with card validity, insufficient balance, and so that the customer can checkout when they are ready and with the right type of payment.

Integration Test:

```
int[] testCart[testIterations]
for(int i = 0; i <= testIterations; i++){
        for(int j = 0; j <= testIterations; j++){
                if(testCart[i].getID == testCard[j].getID){
                        System.out.println("Duplicate item detected, please remove duplicate
                        item");
                        break;
                }
        }
}
ProcessSale();
```

For my Integration test's primary function I decided to test the Cart class. Specifically, I decided to test for when there are duplicate items. In this test we have an array of randomly generated test ID's to simulate a cart containing random items throughout the store, going to be wrung up and checked out. The test makes 2 indented for loops that compares the ID of each item with the ID of every other item. If any of the ID's match up, the system reports an error and breaks the loop, allowing the employee to remove the accidental duplicate item. If none of the ID's match up, then none of the items in the simulated cart have accidentally been scanned twice, so the system allows the user and customer to proceed to checkout. This system tests for failures involving employee mistakes with scanning duplicate items, so that the customer doesn't get charged more than once for each item.

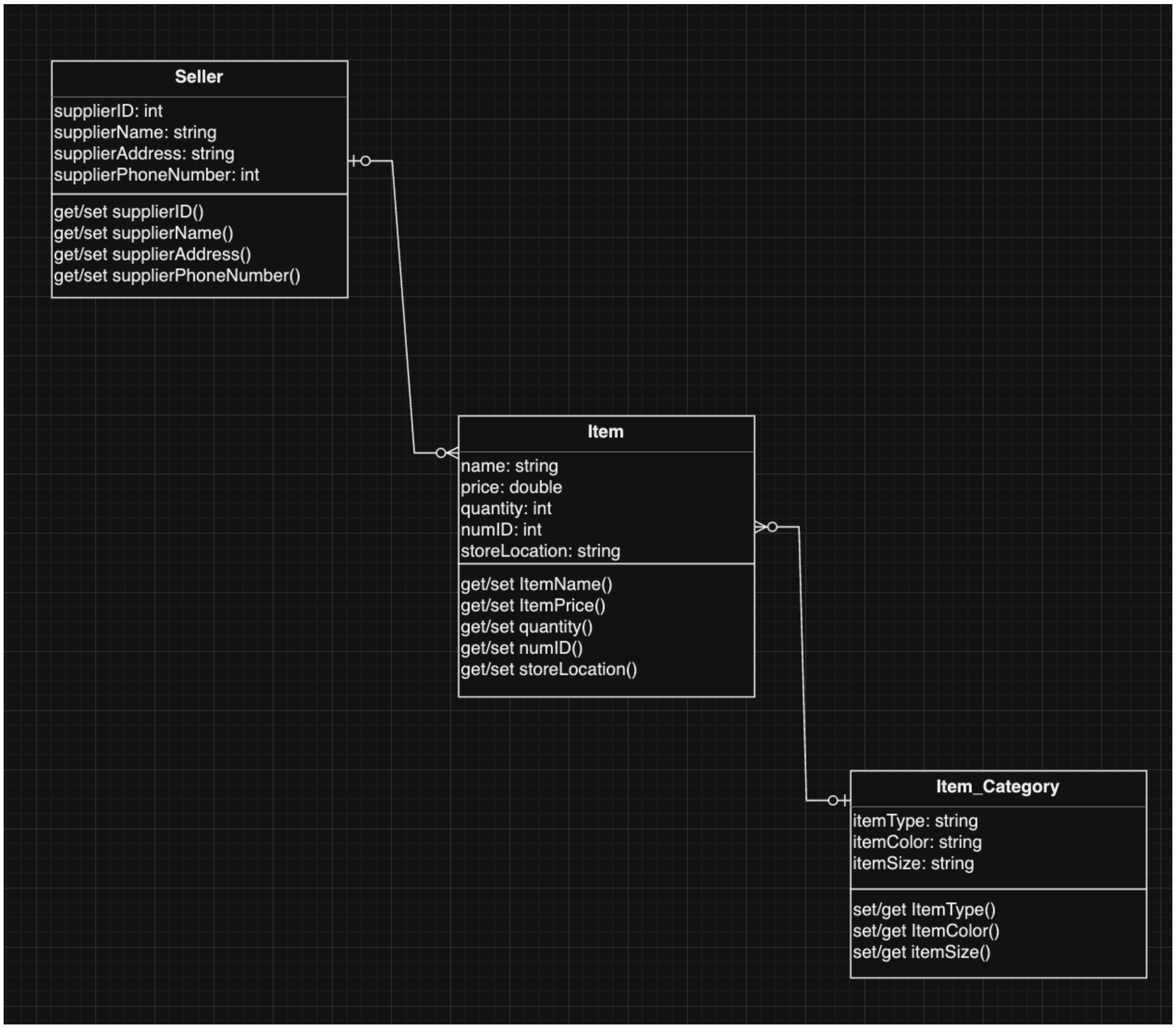
System Test:

```
employee e = new employee()
e.login();
e.getIsAdmin();
e.getLocationID();
sale s;
e.processSale(s);
```

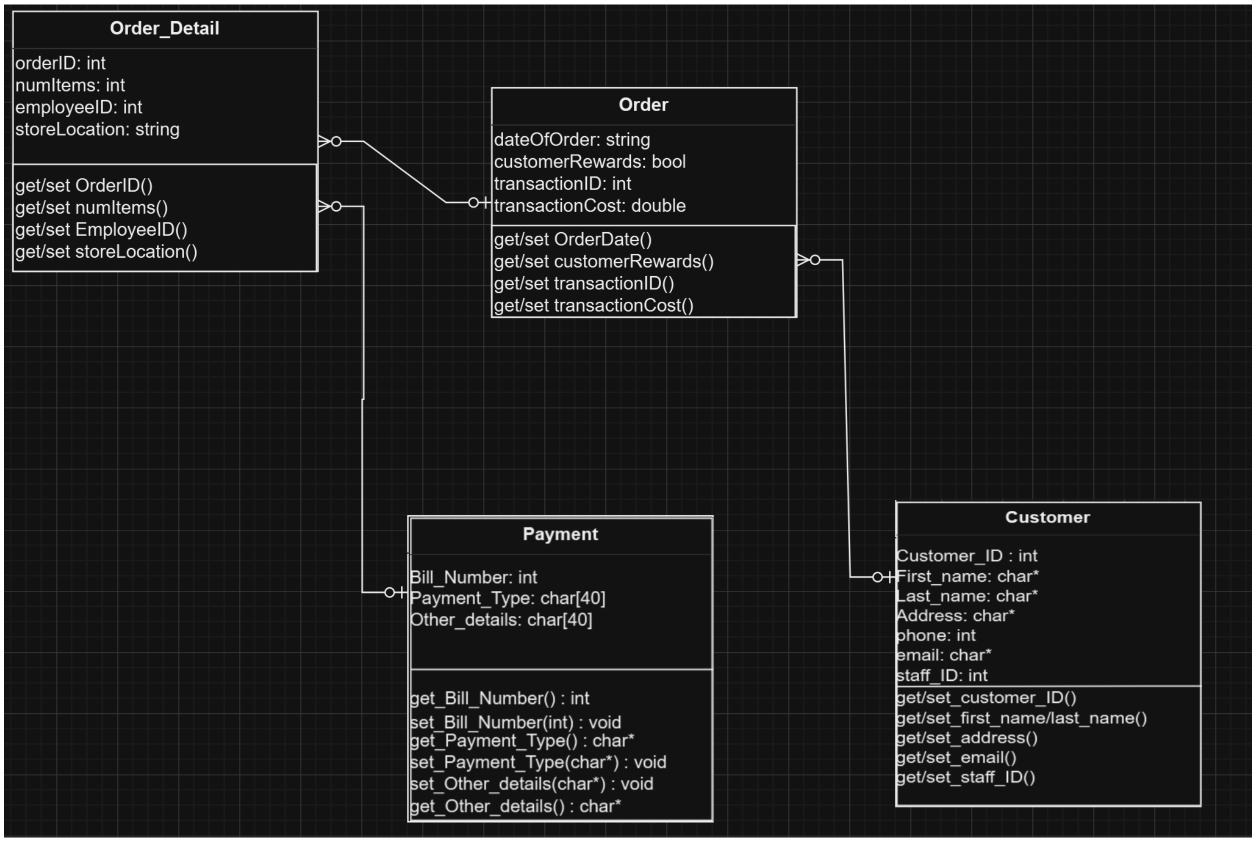
For my system test I decided to target the employee interface as a whole. This test simulates the dashboard that the employee sees, the permissions they have and also a test sale. First, a test employee is created, and then they log into the system. With their new login info, the administrator capabilities and store location they work at is determined. With the admin info, the employee can see if they have access to the database and access to edit other employees. Also, they can see what type of transactions they can handle, and which baseline tests they can bypass. This simulation tests to see failures in faulty administration access, and faulty location info. Also, it gives a general overview of potential failures regarding the dashboard and checkout systems.

Data Management Strategy

For our data management strategy we decided to do SQL for the entire database. We decided to partition it up into inventory, sales history, and transaction history, where sales and transaction history are grouped together. We did this to preserve the separation of administrator access privileges from regular employee access. We chose SQL for every part of the database because of the vast amount of sales and transactions that happen everyday, and all the parameters of the multiple items in each sale would be hard to keep track of in a NoSQL format. Also, with more growth in the business, an SQL database would adjust, but NoSQL would have a harder time.



This architecture allows seamless updates to supplier information, ensuring accuracy even if a supplier changes locations or contact details. Each item is meticulously stored with relevant details, and a categorization system enhances organization. The database's strength lies in its user-friendly approach, enabling users to navigate and modify information with ease. This ensures the reliability and efficiency of our inventory management process, forming a critical component of our Point Of Sale system.



For Sales and Transaction history, we decided to group them together. We also decided to do SQL for this as it would be compatible with scalability. The database also introduces new parameters, namely customerRewards and numItems, facilitating comprehensive analysis. This innovative approach allows us to conduct tests, such as evaluating the location with the highest items per sale and tracking the number of customer sign-ups for rewards. The inclusion of getters and setters for ID information ensures adaptability and addresses potential human errors in the data. This database architecture embodies a user-friendly and scalable solution, empowering our Point Of Sale system to handle sales data with precision and versatility.

Tradeoff Discussion

Implementing two separate SQL databases for a clothing store's point of sale system, one dedicated to sales and transactions and the other to inventory, was preferable for a variety of reasons. This approach allows for the distinct compartmentalization of different types of information. The sales and transactions database serves as a repository for customer-related activities, such as sales, returns, and payments, while the inventory database focuses on product details and stock levels. The use of SQL, a widely recognized and standardized language for relational databases, ensures a consistent and reliable means of managing, querying, and retrieving data. This setup promotes clarity, minimizes redundancy, and facilitates efficient data logging and analytics. Furthermore, the decision to use SQL for both databases emphasizes a straightforward and widely portable approach to data management within the context of the retail environment, contributing to the overall effectiveness of the point of sale system.

Considering alternative approaches for the clothing store's point of sale system, one option could involve using a single database for both sales/transactions and inventory, simplifying the data architecture. This grouping might seem more straightforward initially but could lead to challenges. Without a clear separation, updates to inventory might directly impact transactional processes, potentially causing disruptions or errors in customer transactions. Another alternative could be employing a NoSQL database instead of SQL. NoSQL databases are flexible and can handle unstructured data, which might be advantageous for certain applications. However, they lack the transactional consistency guarantees provided by SQL databases, potentially posing risks to data accuracy in a point of sale context. In terms of trade-offs, the single-database approach sacrifices the clarity gained from separate databases, making it harder to manage and analyze distinct types of information. Adopting a NoSQL database, while offering flexibility, may compromise the transactional integrity crucial for reliable sales and inventory operations. Ultimately, our chosen approach of separate

SQL databases strikes a balance, ensuring clear organization, data integrity, and compatibility with industry standards.