Clothing Point of Sale System Report

Table of Contents

---

1. User Manual

1.1 Introduction

The proposed system is a Point of Sale (POS) solution that has been designed to simplify the process of making purchases within a clothing store, allowing for more efficient checkouts. This system has a standard form of sales across all stores in the same franchise, addressing the need for efficiency when helping customers purchase items. Also, through websites or applications, customers will have a quick and convenient way of buying items. This document provides information to guide all types of users that will interact with the Point of Sale system, and it dives deeper into the software implementation.

1.2 Functionality

There are numerous features implemented into this Point of Sale system to make it robust and efficient, to hold up with industry standards. A critical advantage of implementing this system is the automation of nearly all transactions of data and money that are involved in selling items of clothing. Consequently, a website application is provided, which allows for online shopping, and like mentioned earlier, automated transactions. When a user wants to purchase an item, they can add the item to a shopping cart. Once in the cart, they can make a purchase where an API is used to execute a monetary transaction. Furthermore, a reward's system is in place that will automatically accumulate points if a user is logged into the system. Once a purchase is made, the inventory for that item will be automatically updated to reflect the loss of that specific item (identified by barcode ID, and other qualities of the item). Despite the limited use of user accounts, employees and managers will have escalated privileges to sell items, and edit database respectively. Moreover, managers will have access to transaction history through their own customized logins (similar to users and employees). There will be a cloud system in place to ensure the consistency of data between store locations.

1.3 System Requirements

The system provided will require adequate technology in order to keep up to date with security measures and maintain efficiency. All equipment used on behalf of store locations must have fully up-to-date operating systems to limit vulnerabilities. Furthermore, hardware used must be less than five years old to bypass any hardware restrictions when using the system. Next, it is necessary to be connected to the internet, as that is how the cloud database will be accessed and modified. Additionally, barcode scanners will be needed for providing barcode identification on items at store locations. Payment devices will also be necessary depending on how store locations would like to handle their in store transactions. Easily enough, customers using the

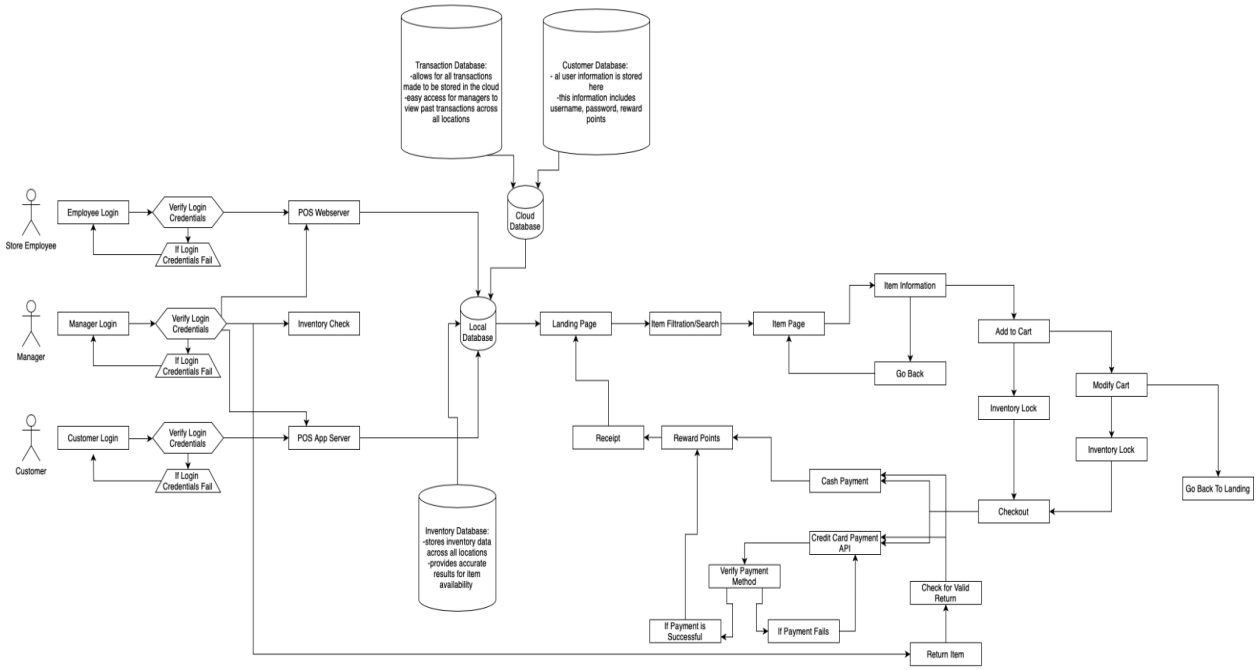
application online for shopping will only need a device that can connect to the internet and reach the website.

1.4 How to Use

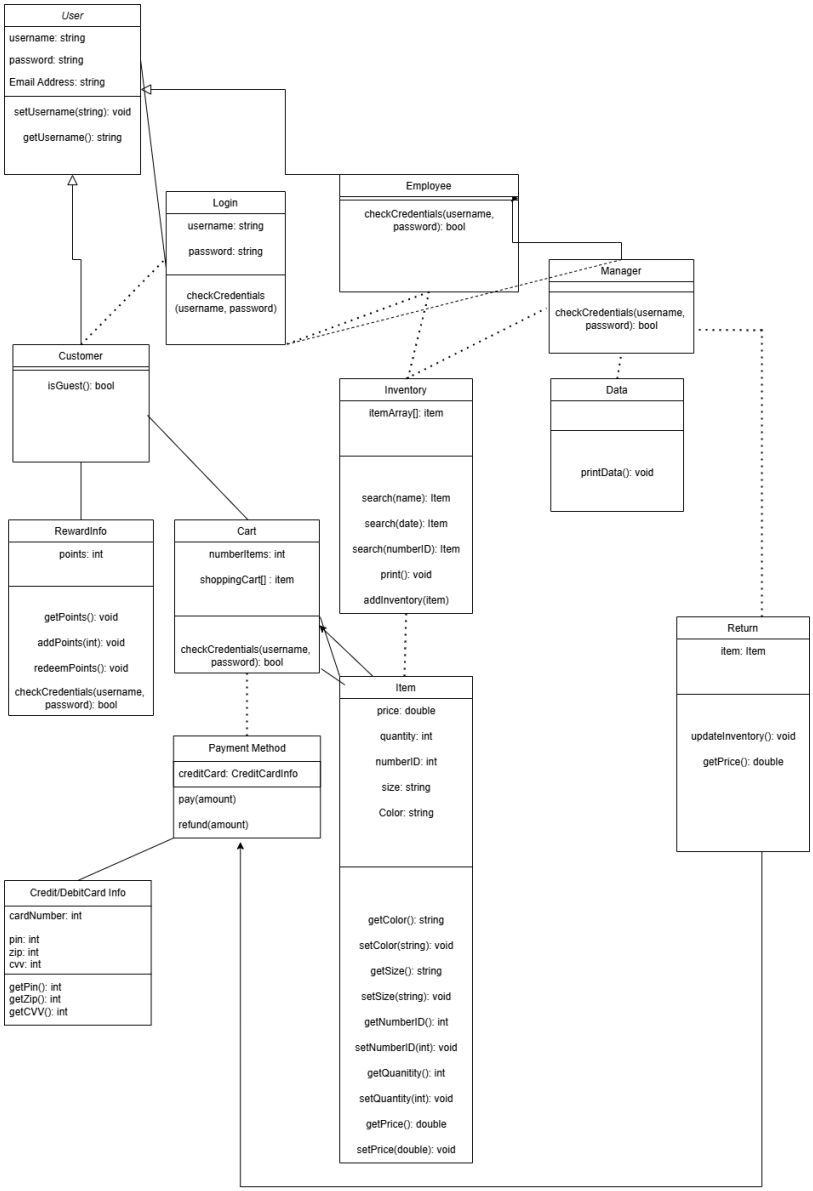
First, users in store will be prompted with a login page. Once entering valid user information, they will be able to access pages specific to their user status. On this login page, there will be a ‘continue as guest’ button for those who would like to make a purchase without an account. Customers will have a simple time selecting items, adding them to cart, and proceeding to checkout (as it is all guided by the website). Employees and managers will have separate tabs which will allow them to utilize the ability to sell items (as well as check transactions if the user is a manager).

2. Design Documents

2.1 Architecture



The Architecture Diagram representing the Point of Sale System we have created flows from an initial log in from a store employee, manager, or the customer. After the system verifies the login, depending on the user, it tracks through the POS web server, POS app server which leads to the local database that loads the landing page. Connected to the local database is the newly added inventory database that holds inventory numbers, allowing for accurate availability results when shopping and doing inventory checks. Additionally, the Cloud Database connects to the Local Database, which contains the Transaction Database and the Customer Database. The connected Transaction Database allows for all transactions made to be stored in the cloud and allows for managers to have easy access to past transactions. The connected Customer Database stores all user information which includes usernames, passwords and reward points. Once the loading page is running, users are able to search the inventory for what they wish to purchase. Once a selection is made, the item information generates and allows users to add to their cart or go back to searching. After an item has been added to the cart, the inventory locks in order to prevent the last item being purchased by someone at the exact same time. Modifying the cart is additionally an option before checking out. Once you land on the checkout page, you are presented with a request to input your payment information. Once a payment method has been processed, the page generates an updated rewards amount and prints/emails a receipt.



The UML Diagram above can be described through an overview of the classes:

User

- The User class is a super class for all the types of users, in order to provide the standard attributes that will be used amongst the subclasses (i.e. username and password).

Login

- The Login class checks user credentials and allows access to the system

Employee

- The Employee class is a subclass of User that is accessed with the correct credentials.
- Gives access to classes such as inventory, that are restricted from customers.

Manager

- The Manager class can be accessed through Login and is given access to the Data, Inventory, and Return classes.

Customer
- The Customer class has limited access to specific classes due to the security of the system.
- The classes accessible relate to purchasing items.

Inventory
- The Inventory class is dependent on and can be accessed by the Employee and Manager classes. It can search for and display items within it. It is also dependent on the Items class.
- The Inventory class can also add items from the Item class to it.

Data
- The Data class is a simple class that prints the data that is pulled from the local or cloud database.
- Dependent class on the Manager class.

Cart
- The Cart class can be accessed by the Customer class.
- This class keeps track of the number of items in a customer's cart.

RewardsInfo
- This class is to keep track of the rewards that are linked to a Customer account.

Item
- The Item class is dependent on the Inventory class because inventories need to use items.
- Marks all the information necessary to an item, so it can be purchased.

PaymentMethod
- The PaymentMethod class is dependent on the Cart class, and it uses information from the Credit/Debit Card Info class to either pay or refund a specified amount of money.

CreditDebitCardInfo
- The CreditDebitCardInfo class is used to hold the information of a credit card or debit card, and use it to purchase items using the PaymentMethod class.

Return
- The Return class is dependent on the Manager class.
- This class allows for returns to be made, while also updating inventory and getting the price of items.

Description of attributes:

username:string
- Stores username used to Login.

password: string
- Stores password used to Login.

Email Address: string

- Stores an email address used to get or set a username/password.

points: int
- Stores points that can be added or redeemed.

numberItems:int
- Stores the number of items in the cart.

shoppingCart[]: item
- Stores a specified item in the cart.

itemArray[]: item
- An array that stores the items in the system.

price: double
- Stored number representing a specified item's cost.

item: item
- Item that will be used in the Return class.

creditCard: CreditCardInfo
- Stores the payment information from the user's credit card.

quantity: int
- Stores the amount available of a certain item.

numberID: int
- Stores the ID number of a specific item.

size: string
- Stores the size of a specific item.

color: string
- Stores the color of a specific item.

cardNumber: int
- Stores the card number of the credit/debit card.

pin: int
- Stores the pin of the credit/debit card.

zip: int
- Stores the zip of the credit/debit card.

cvv: int
- Stores the security code of the credit/debit card.

Description of operations:

setUsername(string): void
- Takes a string as an argument and sets the username to it.

getUsername(): string
- Returns the username as a string.

checkCredentials(username,password): bool
- Checks if the credentials fit Customer, Employee, or Manager.
- Takes the username and password as arguments.

- Returns true or false depending on if it fits the class that is being checked.
- If it is in Manager class, true means that credentials work for Manager.

search(name): Item
- Searches for Item in Inventory by the argument “name.”
- Returns an Item in the Item array that matches the name argument.

search(date): Item
- Searches for Item in Inventory by the argument of date added.
- Returns an Item in the Item array that matches the date argument.

search(numberID): Item
- Searches for Item in Inventory by the argument numberID.
- Returns an Item in the Item array that matches the numberID input.

print(): void
- Prints every Item in Inventory.

addInventory(item): void
- Adds an Item to the array stored within Inventory.

printData(): void
- Prints transaction and inventory data.
- Pulls data from the local database and cloud database.

getColor(): string
- Returns the color of the Item as a string.

setColor(string): void
- Sets the color of Item, using a string input.

getSize(): string
- Returns the size of the Item as a string.

setSize(string): void
- Sets the size of the Item, using a string input.

getNumberID(): int
- Returns the numberID of Item as an int.

setNumberID(int): void
- Sets the numberID of Item with the int input.

getQuantity(): int
- Returns the quantity of Item as an int.

setQuantity(int): void
- Sets the quantity of Item using the int provided as input.

getPrice(): double
- Returns the price of Item using a double.

SetPrice(double): void
- Sets the price of Item using a double input.

pay(amount): void
- Charges the card provided for the input amount.

refund(amount): void
- Refunds the card provided for the input amount.

getPin(): int
- Returns the pin of the provided card as an int.

getZip(): int
- Returns the zip of the provided card as an int.

getCVV(): int
- Returns the cvv of the provided card as an int.

updateInventory(): void
- This method lies within the Return class.
- This will return the Item from the transaction history back into the array of Items.

getPoints(): int
- Returns the number of points accumulated

addPoints(int): void
- Adds reward points after every purchase

redeemPoints(): void
- Redeems rewards points accumulated through past purchases

isGuest(): bool
- Checks if user is logged in or not
- If a user is not logged in they will not receive rewards points.

2.2 Test Plan

In this section, Unit, Integration, and System Tests are presented to represent our test plan for the Point of Sale system.

Unit Tests

Feature #1: Login: checkCredentials(username, password) This feature tests the inserted username and password to see if it matches with an account. Input: Username: “John” Password: “John123” Output: “Customer ‘John’ login successful” // Customer Login attempt, used customer credentials, was successful Input: Username: “FrancisManager” Password: “Sufenion!9824”	Feature #2: User: setUsername(string) The goal of this feature is to set the username of an account with valid characters and not already taken Input: “Bobby” Output: “setUsername was successful” Input: 1234 Output: “setUsername was not successful, input must be a string” Input:

Output:
“Manager ‘FrancisManager’ login successful”
// Manager Login attempt, used manager credentials, was successful
Input:
Username: “applebeesmyfavorite”
Password: “”
Output:
“Invalid Credentials. Please try again”
// Invalid credentials for login, redirected to the login page
Input:
Username: “MandoGamer”
Password: “workingNow29”
Output:
“Employee ‘MandoGamer’ login successful”
// Employee Login attempt, used employee credentials, was successful
	“”
	Output:
	“setUsername was not successful, input is invalid”
	Input:
	“Bobby”
	Output:
	“setUsername was not successful, username has already been taken”
	// Assuming the other User, Bobby, was already created

In the first feature test set, we test the main different inputs that would reflect normal and abnormal use of the checkCredentials method. We do so by putting in both valid and invalid username and password arguments. For the second feature test set, we do the same thing to test normal and abnormal use of the setUsername function. This helps us see if there was anything that we needed to add to our diagram. We tested these options by putting in strings that should not produce any output, as well as inputting other data types.

Integration Tests

Feature #1: Inventory: addInventory(item)
Verifies that the system can successfully add items to its inventory:
//Item added successfully
Input:
Item name: Sweater
Item price: 65
Item quantity: 25
Item numberID: 1234
Item size: Medium
Item color: Black
Output:
Item name: Sweater
Item price: 65
Item quantity: 25
Item numberID: 1234
	Feature #2: Inventory: search(numberID): item
	Verifies that the system can find an item using its ID number:
	//item ID number search successful
	Input:
	ID number: 1234
	Output:
	Sweater
	Shows how the system reacts to an invalid search:
	//item ID number search unsuccessful(put in letters instead of numbers)
	Input:
	ID number: abcd
	Output:

Item size: Medium
Item color: Black
Shows how the system reacts to invalid input such as a blank:
//item added unsuccessfully(Price was left blank)
Input:
Item name: Sweatpants
Item price: (blank)
Item quantity: 50
Item numberID: 4321
Item size: Large
Item color: Gray
Output:
Item name: Sweatpants
Item price: (blank) //failed because price was left blank
Item quantity: 50
Item numberID: 4321
Item size: Large
Item color: Gray
	failed(due to letters being inputted rather than numbers)

We did an integration test on the Inventory: addInventory(item) feature where we ran a successful test by putting valid inputs and we ran an unsuccessful test by leaving an input slot blank. After that we did an integration test on the Inventory: search(numberID): item feature where we ran a successful test by inputting a valid ID number (used the id number from the previous addInventory test) and we ran a failed test by inputting letters rather than numbers for the ID number.

System Tests

Feature #2: Search for a Product: searchProduct(string) Input: Search Phrase: “Black Jacket” Output: “Results for Black Jacket” Item results loaded successfully Feature #3: Selecting an Item: selectItem(string) Input: Selected Clothing: “Black Leather Jacket” Output:	System Test: Attempting to Make a Return Feature #1 Manager Login: validateCredentials(username, password) Input: Username: “Manager1” Password: “Manny250” Output: “Manager ‘Manager1’ login successful” Feature #2: Return Process: processReturn(item, paymentInfo, rewards) Input: Item: “Black Leather Jacket”

Item Information: Name: “Black Leather Jacket” Price: $499 Size: S, M, L Color: Black Item information generated successfully Feature #4: Adding an Item to Cart: addItemToCart(string, size, quantity) Input: Item: “Black Leather Jacket” Size: “Small” Quantity: 1 Output: “Black Leather Jacket added to cart successfully” Adding item to cart ran error free Feature #5: Cart View: displayCart() Input: N/A (Customer is just viewing the cart) Output: Cart Summary: Item: Black Leather Jacket” Price: $499 Size: Small Color: Black Quantity: 1 Total: $499 The cart loads all information accurately Feature #6: Checkout: proceedToCheckout() Input: None (Customer begins to checkout) Output: “Checkout page loaded successfully” The checkout page loaded error free Feature #7: Select Delivery Method: setDeliveryMethod(option) Input: Delivery Option: “2 day shipping” Output: “Selected method: 2 day delivery; has successfully been saved” Feature #8: Processing Payment Details: enterPaymentDetails(cardNumber, expDate, securityCode, useRewards) Input: Card Number: “1234 5678 8765 4321”	Price: $499 Payment Info: Card ending in “4321” Rewards: 100 points Output: “Refund processed to card ending in ‘4321’. 100 reward points = $1 have been reissued.” The return process has been completed

Expiration: “10/25” Security Code: “842” Use Rewards Available: Yes (100 points = $1) Output: “Payment processed successfully and rewards have been applied” Running the payment and using the reward points available ran successfully Feature #9: Order Confirmation and Receipt: confirmOrder(receiptType) Input: Receipt Type: Email Output: “Order confirmation has been sent to email. Purchase ran successfully and email has been sent to customer	

We completed two System Tests, one being the process the system takes a customer to make an item purchase. This test began with the customer logging into their account and ended with a valid purchase of an item. The second test we ran was the entire process the system takes a manager through in order to make a return. This test began with the manager logging in and following the path the system takes to make a valid return/refund. Both tests ran successfully.

After concluding the test sets and describing not only the function, but results of them. It was evident that our Software Architecture diagram would need some edits in order to polish the software system. These edits will be reflected in the new Architecture diagram that has replaced the old one in this version of the document.

2.3 Data Management Strategy

Customer Database

Username (String)	**Users password (String)**	**User Email(String)**	**Users Reward Points (int)**
Aiden_Jajo	54321	aiden123@email.org	1800
Walid_Elassaad	12345	walid123@email.org	1200
Dean_Kajosevic	13524	dean123@email.org	1400

Inventory Database

Item	Number Id	Price	Quantity	Size	Color
Gucci	100	30	10	Large	Blue
Lebron	101	20	1	Medium	Red
Stussy_Hoodie	102	10	40	Small	Black
Baker_Tee	103	5	29	Small	Brown
Louis_Belt	104	300	100	Large	Black

Transaction Database

Username	shoppingCart	Total price	Credit Card #	Pin	cvv	zip	Transaction Date
Aiden_Jajo	Stussy_Hoodie	10	5422958903491510	1000	423	92119	11072024
Walid_Elassaad	Baker_Tee, Louis_Belt	305	5205347006385782	2000	324	92115	11072024
Dean_Kajosevic	Gucci, Lebron, Louis_Belt	350	5146088989511372	3000	243	91977	11072024

Overview of Data Management Strategy

How many databases did you choose and why?

We chose three databases based on the information that we would need to be shared across our software system. We split the data up into customer, inventory, and transaction databases. The reason why we chose these three specifically is that they do not fully overlap with each other, so it would be most effective to have them separate. Furthermore, we decided we need these as databases because they hold information that is vital to using our software system, such as customer for logging in, transaction history for manager viewing, and inventory to hold items that will be purchased.

How did you split up the data logically?

We split up the data into several databases, with each made to hold specific information based on its function in the system. This splitting into separate databases helps us manage the data more efficiently. Each database has its own role in providing all the data necessary for the steps in our use case diagrams. One example being that the inventory database holds all the Items in the store, so they can be added to cart or modified.

What are possible alternatives you could have used (both in technology and organization of data)?
Alternatively, we could have had people manually input data into each of our systems at different locations instead of utilizing databases that span across different locations, which would be an alternative in technology. However, this would require a lot of extra work and lead to error when it comes to having the correct data. In terms of technology and organizing data, we could have used noSQL instead of SQL. We decided to use SQL because it was more concise and easy to follow along with if we hired other people to maintain our software system. In terms of purely organization, we could have had more or less databases, which would have changed cost and how we structured our databases. In the end, we decided that the three we settled with was the perfect amount for our use cases.

What are the tradeoffs between your choice and alternatives?
When it comes to implementing three different databases, it makes the use of data easier across locations and also lacks room for error when transferring data. In terms of using SQL over noSQL, the implementation of our database using SQL is more straightforward because it has its own syntax. Whereas, noSQL does not have a specific syntax, which could make it harder for new hires to maintain our software system. We could have also implemented more databases, which would split up data more. However, organizing data into more databases than we already have would cost more money and could have a lot of redundancy in holding the same data. On the flip side, having more databases could make the data more specific, which has the potential to be beneficial.

Examples in SQL

Add new Item: Add a new item to the database, along with specific characteristics for that item.

INSERT INTO Inventory_database
VALUES (Nike_Tech, 105, 100, 50, Medium, Black)

Example output:

Item	**Number Id**	**Price**	**Quantity**	**Size**	**Color**
Nike_Tech	105	100	50	Medium	Black

Find transaction: Search for transaction based on username, shopping cart, price, credit card, pin, cvv, zip, transaction date

SELECT username, shoppingCart, price, creditCard, pin cvv, zip, transactionDate.
FROM transaction_database
WHERE username = Dean_Kajosevic;

Example Output:

Username: Dean_Kajosevic
Cart: Gucci, Lebron, Louis_Belt
Total Price: 350
Credit Card: 5146088989511372
Pin: 3000
CVV: 243
Zip: 91977
Date: 11072024
—-----—-----—-----—-----—-----—-----—-----—-----—-----—-----—-----—-----—-----

Remove account from CustomerDatabase: Remove an existing account from the database.

DELETE FROM CustomerDatabase
WHERE Username= "Walid_Elassaad" ;
—--------------------------

2.4 Security Analysis

Through the Confidentiality, Integrity, and Availability (CIA) framework, we came up with potential security risks that would arise with our Point of Sale System. In terms of dealing with confidentiality, the website application could be susceptible to SQL injection. This could be used in text entry areas, such as the login page. SQL injection can be prohibited by blocking certain characters and phrases that are often used to craft SQL injection payloads. Next, a possible vulnerability in the integrity of the system is inventory and transaction manipulation. If manager passwords are compromised, then the new user on the manager account will be able to modify inventory data. In order to combat this, there would need to be measures in place to secure accounts in the website application to make sure that elements of logins (such as cookies) are not manipulated to allow for privilege escalation and that passwords are not compromised. There should be a standard ruleset for manager passwords to make sure that they are not vulnerable to be brute-forced. Finally, availability can be threatened through denial-of-service attacks, putting down the website services related to our software system. One way to circumvent this would be implementing some software that listens to network traffic and ensures that there is no unusual traffic bombarding the websites that the point of sale system is hosted on.

3. Lifecycle Model

3.1 Overview

In the Clothing Point of Sale System, we used a spiral lifecycle model. This model utilizes a development process of planning, risk assessment, development and testing, and evaluation. We chose this model because we would be able to spend every evaluation and planning phase with hypothetical clients and teammates, which would allow for consistent feedback to ensure that the project stays on the right track. Additionally, using the spiral model enables vast amounts of analysis on our work, which brings to light flaws or features that must be addressed. Moreover, the phases of the spiral model were extremely collaborative, which facilitated strong communication and collaboration throughout the development process.

3.2 Reflection

Looking back, using the spiral model worked out perfectly, we were able to get enough feedback to stay on track with what the client wanted, and we had enough wiggle room to express creative freedom in the development cycle. Furthermore, the cycle was effective in allowing us to continuously assess risks and the consequences of our implementation. This was extremely significant during development because releasing a flawed system would be detrimental to users, especially when handling important data.

3.3 Modifications

Based on our reflection on the spiral model, we would likely not recommend switching to a different lifecycle model unless the team working on the software system expands drastically. The reason why this model would not be the best for a large group is because it requires the people working on the project to all assess the direction of the project as it is being developed. Therefore, there is not a set group of objectives that can be split amongst many people, rather the developers will work through the entire cycle together. If there is a large expansion of the software system, then there may be better lifecycle models available to powerfully handle modular development and distribute work amongst a greater amount of people.