SDSU | San Diego State University

CS250 Introduction to Software Systems
Professor Miranda Parker

Clothing Store Point of Sales System

[Final Report]

By Bryce Jarboe

Table of Contents

User Manual

[Section 1]

Introduction and Overview

The Clothing Store Point of Sales software system will function as the digital backbone of any clothing or retail store that chooses to integrate it. It serves as a median and toolset for inventory management, transaction processing and history, and security. As such, the main *pillars* that enable operation are its inventory system, transaction system, SQL databases, and the authentication system. Overall, switching to our software system will promote efficiency through automation and data integrity. Basic operation of the fully implemented system would proceed as follows: Management staff can login to preset administrative accounts and will add any necessary users. Next, the permitted staff will build the inventory database, ideally with an already-prepared inventory data set via a setter function. Now, during store hours, the external payment terminals will be connected to the software system (API) and be able to approve transactions that will automatically update the inventory accordingly as well as keep record of said transactions in its own database entity. See the *Design* section for more information.

Features

Key features of the software system will include the following:

Inventory management

- Creation and deletion of clothing entries
- Automatic update following verified transactions

Transaction processing and verification

- Built-in functions to successfully process transactions
- Verification via external payment terminals

Administrative privileges

- Admin access to up-to-date transaction history
- Management of user database

System Requirements

Hardware requirements

- Central server
    - The server will house the data and its initial footprints
    - Connectivity to SQL Entities
    - This should be store-mandated such that only authorized personnel should be able to physically access the server/computer
    - The server's hardware should be able to run simple tasks efficiently; there is nothing that is notably hardware-intensive present in the system
- Smartphone network
    - The software system will extend to a network of authorized smartphones via its own user authentication system
    - Store-mandated smartphones with scanner attachments would be ideal as to streamline the installation process (See iOS / Android API below)

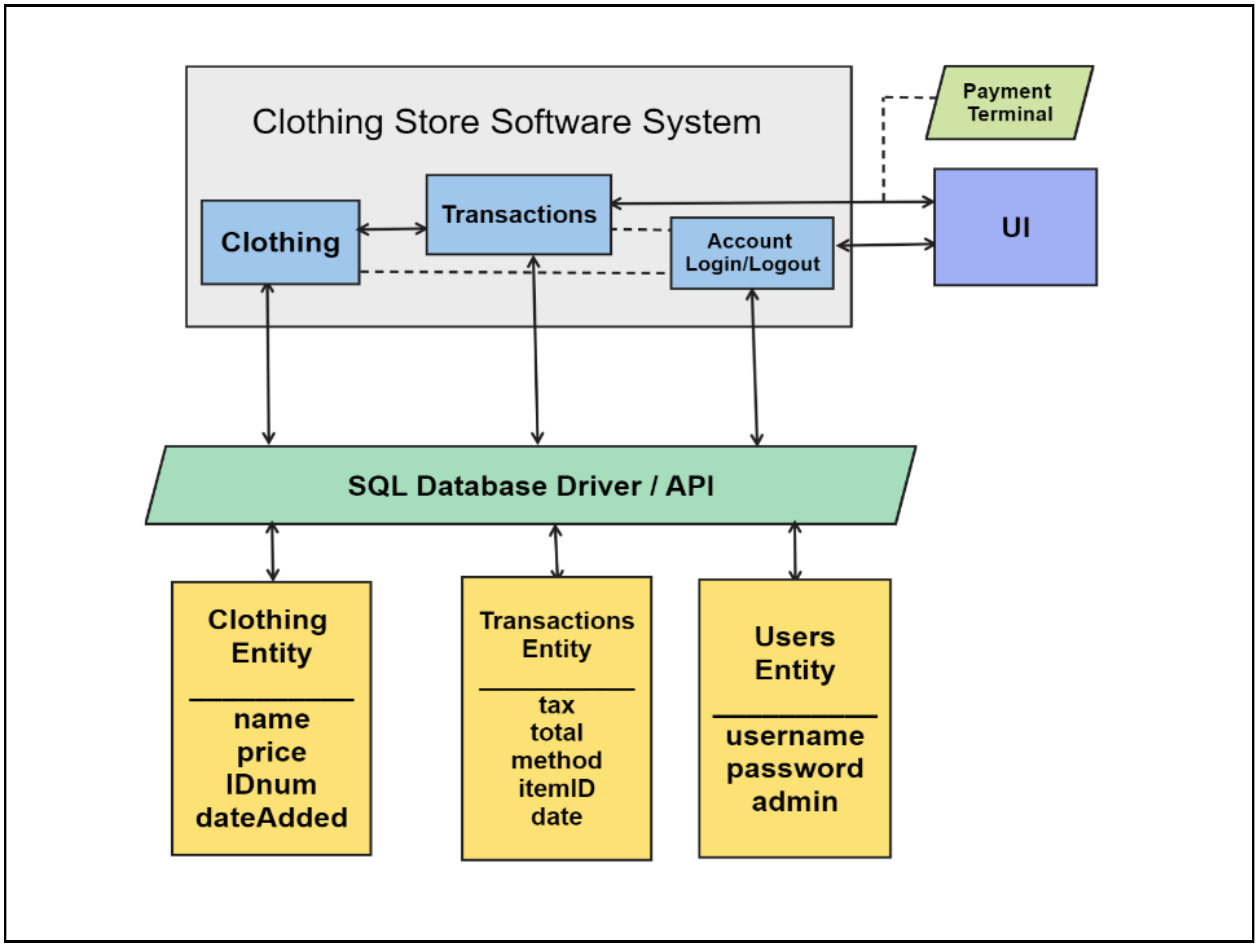
Software requirements

- iOS / Android API
    - Compatibility with a smartphone operating system of choice
    - Implementation of both options would prolong the development process
- SQL Driver / API
    - Internal communication with the SQL database entities (inventory, transactions, users)
    - Ability to query the data and enable a majority of the system's core functionalities

System Design

[Section 2]

Software Architecture Diagram



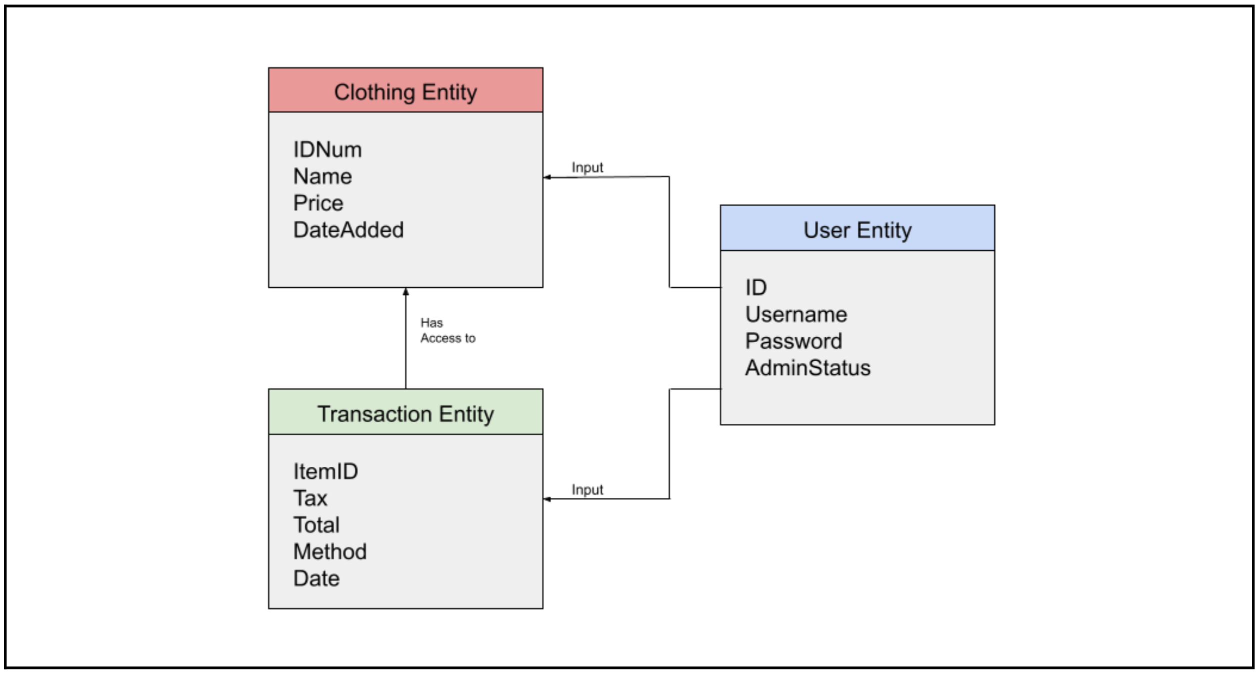
Architectural Diagram

Overview of Software Architecture

The following are the core components that service the functionality and operation of the system:

- Clothing/Inventory
    - This component is the central structure that the software system is built on, allowing for the creation and storage of Clothing entries in the respective table/entity via user interaction, and carrying essential data to the transaction processing system.
- Transactions
    - This component takes data from the Inventory through search, and performs transactions via an external terminal, updating the transaction history all while calculating the tax, printing receipts, and mandating the payment method (admin-controlled).
- Account Login/Logout
    - This component provides users with sophisticated interaction as well as added security to the system, maintaining an entity of user accounts that hold usernames and passwords, administrative status, automatic logout of idle users, option to change a username or password, and payment management.
- SQL Database
    - The data in question will be stored under a SQL database management system. See *Data Management* section for details.

Data Management Diagram



Entity Relation Diagram

Data Management

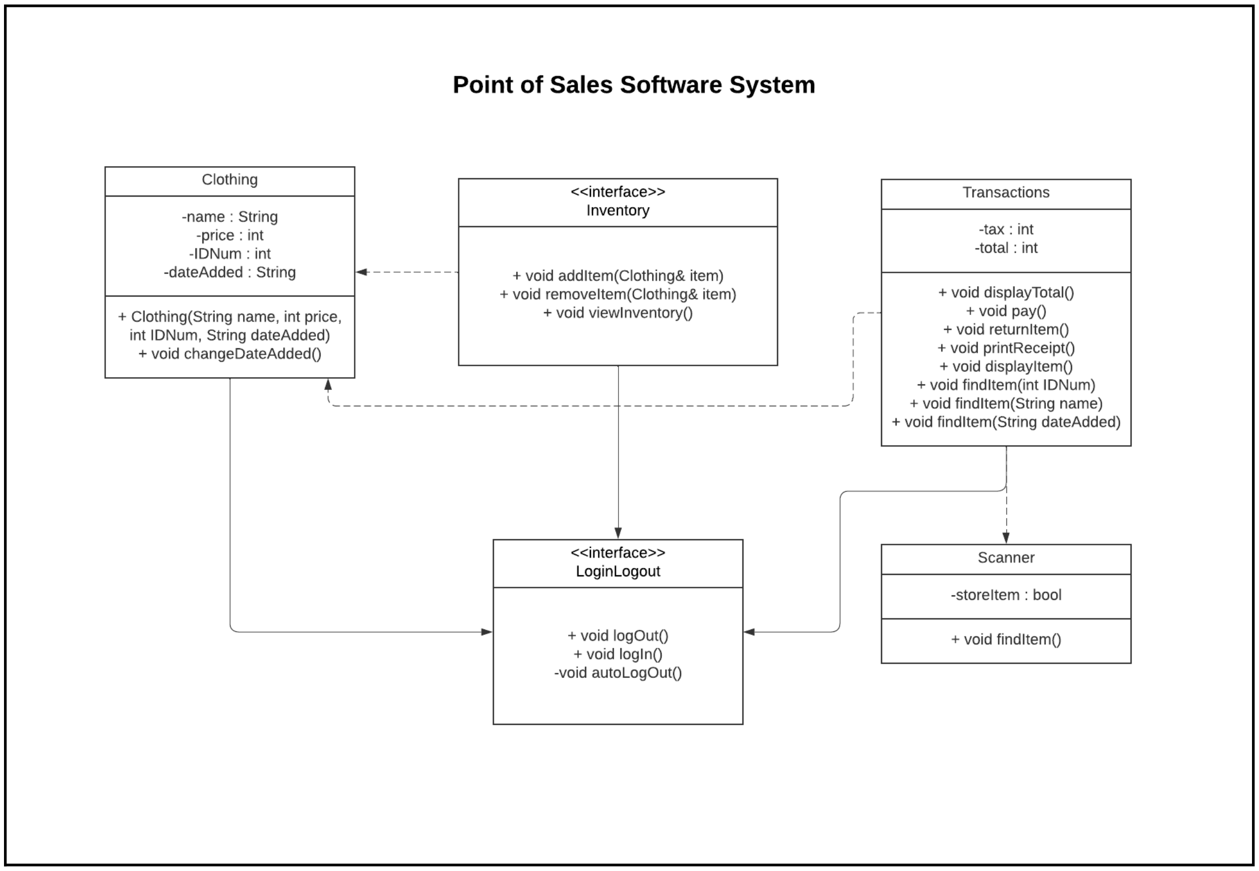
The software system will manage data through the traditional SQL model via the integration of a SQL database driver or API, the choice of which depends on the development process and test results. Methods within the system's main classes that modify necessary data will be capable of issuing SQL statements through the driver/API and thus reading and writing to the database directly.

As for the organization of the data itself, the database will be divided into 3 main entities (tables) and exhibit the following behaviors:

- Clothing:
  - Indexed by ID Number
  - Fields: Name, Price, DateAdded
- Transactions:
  - Indexed by ID Number
  - Fields: Total, Tax, Method, Date
- Users:
  - Indexed by arbitrary ID
  - Fields: Username, Password, AdminStatus

Regarding the tradeoffs of using SQL, it is worth mentioning that we will be, to a degree, *confined* to the capabilities of SQL's methods of data management. In the case of very basic inventory and transaction history systems, SQL will suffice. However, the flexibility of a new or crafted data management system is out of the question.

UML Class Diagram



UML Class Diagram

UML Classes

Clothing

This class will create the bulk of the primary data. Each Clothing object will hold data that, in tandem with the other classes, will prove essential to the cohesion and overall functionality of the system. See below for details.

- name : String
  - The *name* attribute is a means of identification for the inventory's search functionality.
- price : int
  - The *price* attribute (in cents) will be carried through transactions to the payment terminals, being what is effectively charged and/or refunded.
- IDNum : int
  - The *IDNum* attribute is another means of identification for the inventory's search functionality. In addition to inflating the identifiable data, the int data type would allow for hashing (i.e. faster search times).
- dateAdded : String
  - The *dateAdded* attribute, while serving as an additional means of identification, holds data of interest to the inventory and transaction history, keeping records of each entry regarding when it was added.

+ Clothing(String name, int price, String dateAdded)
  + The instantiation of a new Clothing object will read user input and implement the respective parameters. Its constructor will generate an IDNum not currently used throughout the current inventory.
+ void changeDateAdded()
  + *Administrative Access Only*
  + This setter allows Admins to modify the *dateAdded* attribute manually, should it be necessary.

Inventory

Inventory is an interface connected to the Clothing class that can access Clothing objects in order to modify the current inventory of the store.

+ void addItem(Clothing& item)
  + *Administrative Access Only*
  + This will allow admins to add to the current inventory of items, mainly to be used when new products arrive

- + void removeItem(Clothing& item)
    - + *Administrative Access Only*
    - + This will allow admins to remove from the current inventory of items, to be used when there are defective products that need to be removed.
- + void viewInventory()
    - + *Administrative Access Only*
    - + This lets admins view the current inventory of a store but does not allow changes to be made from there.

LoginLogout

LoginLogout is an interface that is used by all other classes because it needs to be able to be accessed at any time.

- + void logOut()
    - + Allows the user to log out of the system at any time, even in the middle of a transaction.
- + void logIn()
    - + Allows the user to log into the system with a unique password tied to their user ID.
- - void autoLogOut()
    - - This function logs the user out if they do not interact with the system for a set period of time in order to maintain security.

Scanner

Scanner is the primary tool that the system uses in order to identify any clothing items that are presented. It is used to scan barcodes and instantly know all the attributes of the clothing item

- - storeItem : bool
    - - The storeItem variable shows if the barcode scanned is an item found in the store or not. This is used in order to prevent confusion.
- - void findItem()
    - - This method is called every time a barcode is scanned. It searches a map of all the items in the store using the *IDNum* and if the item is not found it will set the *storeItem* to false and will let the Transactions class know there was an issue finding the item.

Transactions

- tax : int
    - The *tax* variable is programmed to have the correct state tax based on the location of the store. The tax is used in the final calculation of the total.
- total : int
    - The *total* variable is used to calculate the total cost of all the items in a given transaction, add them all up, making sure to apply tax and any other partial payments like coupons or gift cards.

+ void displayTotal()
    + displayTotal() is automatically called after each item is scanned to show the current subtotal for the current transaction.
+ void pay()
    + Adds the option for a worker to select that lets the customer pay. It would be bad otherwise if the customer could pay at any time during the transaction as it would result in much confusion.
+ void returnItem()
    + Similar to the pay() function. It can be selected after all items have been scanned as well as a receipt and will refund the items to the card that bought it. Or cash could be refunded as well if the first does not apply.
+ void printReceipt()
    + Automatically called at the end of a transaction and prints a receipt for the customer to keep.
+ void displayItem()
    + Like displayTotal(), this method is called after every item is scanned, but instead it shows all relevant information about the item scanned to the customer such as price and name.
+ void findItem(int IDNum)
+ void findItem(String name)
+ void findItem(int dateAdded)
    + The findItem functions are all the same function that can be called with different parameters through the use of overloading. These functions are used to find the price and other info of an item whose barcode will not scan or is missing. The clothing can be found via *IDNum*, *name*, or *dateAdded*.

Test Validation Overview

This portion of the document will describe, in detail, individual test plans for different components of the Clothing Store Point of Sales Software System. Entries will be organized by a scope category (Unit, Functional, and System) and contain the following information: Function Name, Function Parameters, 2 Test Cases (each with a description of the test itself as well as an Expected Output), and Significance to the software system's functionality and overall operational capabilities.

*Note that return types are not listed seeing as all methods being tested are void, deeming them not applicable.

Unit Tests

Log-Out

Function Name:

LogOut

Parameters:

None

Test Case 1:

Use logIn to log into the system. Then use logOut.

Expected Output:

The user can no longer access the system functions except for the LogIn function. If the user can still interact with anything other than the logIn function the test is a failure.

Significance:

This function is important because only employees and admin should have access to this system. If anyone is able to access the system when an employee leaves their station, it could result in theft of items or improper use of the system.

Test Case 2:

Halfway through the payment process use the logOut function. Then log back in again to confirm the transaction is canceled.

Expected Output:

The user can no longer access the system functions except for the LogIn function. If the user can still interact with anything other than the logIn function the test is a failure. Also, the transaction should be canceled and upon logging back into the system, it should not appear again.

Significance:

The reason the transaction should be canceled is the customer does not want to pay for items twice. Also if any gift cards or coupons had been applied to the purchase before the payment, they need to be refunded too.

Add-Item

Function Name:

addItem

Parameters:

Clothing& item

Test Case 1:

Call addItem with an arbitrary clothing item address. Then call viewInventory and verify the contents for the newly added item, validating the item's information.

Expected Output:

The item is viewable in the inventory, preserving all original data passed through the function.

Test Case 2:

Verify the inventory contents by calling viewInventory. Then call addItem with an invalid address. Call viewInventory again and verify the contents.

Expected Output:

An exception should be thrown and prevent any update to the inventory. Its contents should be unchanged since its previously checked state.

Significance:

The addItem function is essential to the operation of the software system as a whole. This is the primary method of adding malleable data to the inventory.

Functional Tests

<u>View-Inventory</u>

Function Name:

viewInventory

Parameters:

None

Test Case 1:

Use the addItem function to add an item to the inventory. Then, use viewInventory to check the inventory.

Expected Output:

Item was successfully added to the inventory and can be seen in the inventory.

Test Case 2:

Use the removeItem function to remove an item from the inventory. Then, use viewInventory to check the inventory.

Expected Output:

Item was successfully removed from the inventory and is no longer seen in the inventory.

Significance:

The viewInventory function needs to properly work in order for the inventory in stock to be 100% accurate. If an item is no longer in stock but still appears in the inventory, a customer could purchase an item that doesn't actually 'exist'. If viewInventory isn't working properly and items aren't being added, then this limits the customers inventory to choose to purchase from.

Change-Date-Added

Function Name:

changeDateAdded

Parameters:

None

Test Case 1:

Go into viewInventory and select an item in the inventory. Select the changeDateAdded function and change the date added to the correct date.

Expected Output:

The date added for the item that was selected was properly changed to the user's input.

Test Case 2:

Add an item to inventory using the addItem function. The dateAdded should be equal to that day. Use the removeItem function to remove the item from inventory. Re-add the item to inventory and ensure dateAdded still shows that day.

Expected Output:

dateAdded will be equal to the current day.

Significance:

The dateAdded function needs to properly work because it helps the store know when a particular item was added to the inventory and to be able to know when some items may be too old and out of date to sell anymore or out of season.

System Tests

Print-Receipt

Function Name:

printReceipt

Parameters:

None

Test Case 1:

Login via the UI. Utilize the check-out's designated scanner on the test item (holding a price of $0). Then, through the payment terminal interface, proceed as instructed on screen and pay for the test item. Verify the contents of any receipts printed at the end of the transaction process.

Expected Output:

The procedure above should successfully yield calls for the following functions in order: logIn(), findItem(), viewInventory(), displayTotal(), displayItem(), pay(), printReceipt(), autoLogOut(). The verifiable output of this entire process is a receipt with the correct data printed, preserving the item's name, ID number, and price within the individual item transaction bulk of the print.

Test Case 2:

Login via the UI. Manually create test item entries (over 20) and add them (each with a $0 price) to the inventory. Utilize the manual item search and check out all test items. Then, through the payment terminal interface, proceed as instructed on screen and pay for the test items. Verify the contents of any receipts printed at the end of the transaction process.

Expected Output:

The procedure above should successfully yield calls for the following functions in order: logIn(), addItem(), findItem(IDNum), viewInventory(), displayTotal(), displayItem(), pay(), printReceipt(), autoLogOut(). The verifiable output of this entire process is a receipt with the correct data printed, preserving all items' names, ID numbers, and prices individually within the transaction bulk of the print.

Significance:

These tests rely on the cohesive functionality of the system as a whole, calling methods from each class to yield an output with verifiable data. Should any sector of the software system fail, the process will be hindered and no verifiable output will be generated.

Item-Exists

Function Name:

itemExists

Parameters:

None

Test Case 1:

Login via the UI. Scan an item that belongs to the store's database. Check that the boolean variable storeItem is true. Double check that all information that is displayed by displayItem() matches the item that was scanned.

Expected Output:

The function itemExists should update the boolean variable storeItem to true since the item being scanned is chosen to be from the store. We then check the value of storeItem to confirm this info. We also double check that the correct item has been received from the inventory by using displayItem() to display the characteristics of the item and checking them against the item in question.

Test Case 2:

Login to the UI. Scan an item that is not an item from the store. Check that the boolean variable itemExists is false. Check to see if displayItem() displays an error message or not.

Expected Output:

The function itemExists() should update the boolean variable storeItem to be false because the item scanned will purposefully not be from the store. We expect there to be an error message when displayItem() is called because the item is not a store good.

Significance:

These tests are important for the system because it allows the users to know immediately if the item scanned is property of the store or not. While it is possible to check if the item is in the system by using other methods, this can alert the users to check the tag and make sure it is the correct one for the item at hand.

System Security

This portion of the document will describe potential threats and system vulnerabilities and propose solutions.

Backdoor

Threat description:

This would entail bypassing the authentication and gaining control of the system as an illegitimate user. This is a plausible threat since a degree of high-administration control is unavoidable for any maintenance or tweaking to the system.

Solution:

Integration of valid, third-party verification/security systems such as PhishER, IBM security QRadar SIEM, Dynatrace, etc..

Direct-Access Attack

Threat description:

This would entail unwanted, physical access to the central computer/server and tampering and/or sabotage. The plausibility of this kind of threat would admittedly be lower than the preceding, but is possible none-the-less.

Solution:

A simple solution would be to secure the store itself and have a dedicated room inaccessible by regular staff for the central server.

Social Engineering

Threat description:

This would entail a party convincing any staff or management to reveal sensitive information about the system–mainly usernames and passwords.

Solution:

Disclosure of confidentiality or a mandated formal confidentiality agreement to the staff.

Life Cycle

[Section 3]

The following timeline provides information regarding the specific phases of development, an estimation of how much time they will take, and the partitioning of tasks among team members.

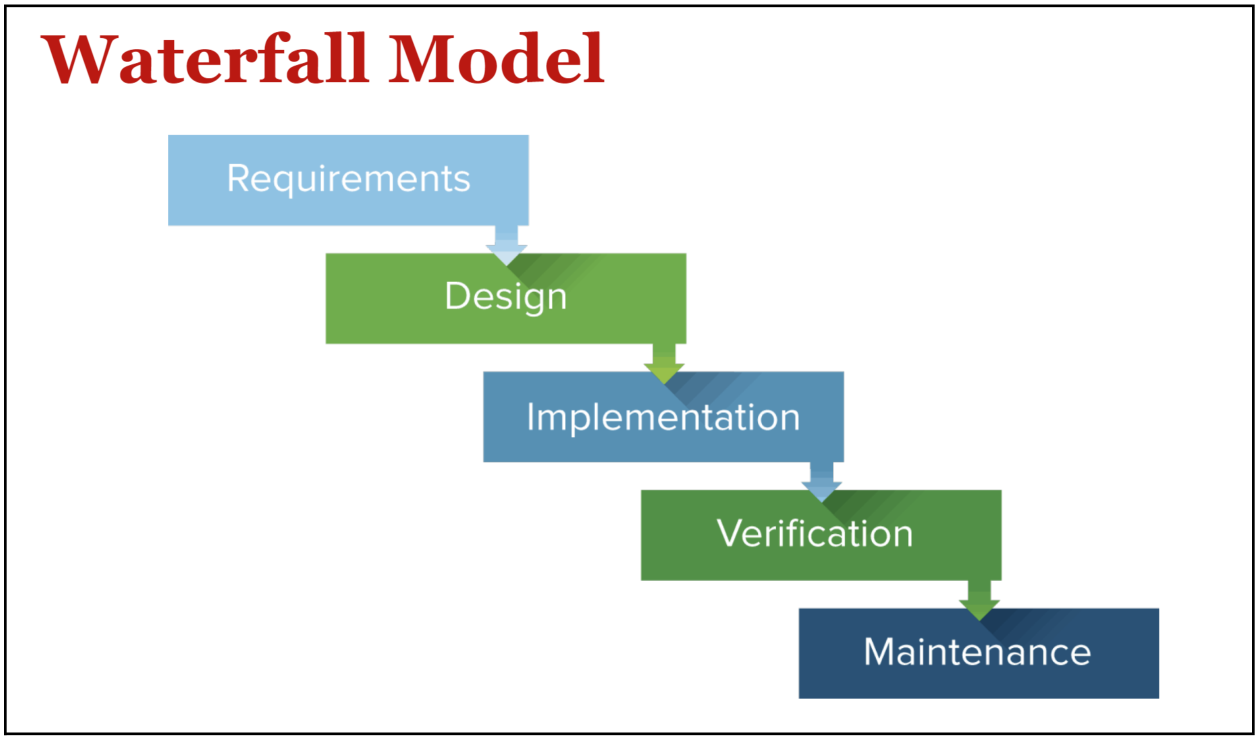
Development Timeline

	Prototype	Implementation	Integration	Validation	Deployment	Maintenance
Estimated Time	*(3-6 weeks)*	(3-6 weeks)	(3-6 weeks)	(6-12 weeks)	(3-6 weeks)	(Ongoing)
	For the creation of an initial prototype of the Clothing Store Point of Sales software system, the entire team will collaboratively write classes and methods. This should be a flexible, cooperative process, partitioning tasks where necessary, but ultimately coming together to create a simple, cohesive system build that will be used for further testing.	This testing phase will primarily focus on the behavior and cohesion of the software system, ensuring it is free from errors in its unintegrated form. Team members will be responsible for debugging and writing test cases for their respective system functionalities. <u>Andrea</u> Transactions: ensuring the correct data is output before implementing the terminals / registers. <u>Bryce</u> Inventory: the instantiation, removal, and search capabilities. <u>Liam</u> User login/logout: functionality of administrative privileges and compatibility with inventory and transactions.	This phase will shift focus towards the the hardware integration. Team members will be responsible for communicating with the client, finding out what hardware the client has budgeted and opted for, then implementing any necessary API's and/or making changes to the system to accommodate. <u>Andrea</u> Payment terminals / Registers: transaction capability via terminals (e.g. *Verifone*, *Dejavoo*, *PAX*...). <u>Bryce</u> Cloud-Backing: the insurance of a back-up system, building an easily recoverable transaction / inventory history (e.g. *iDrive*, *BackBlaze*, *Acronis*). <u>Liam</u> iOS / Android integration: scanners (e.g. XCover), inventory interaction, administrative access.	The validation phase will heavily depend on the responses of the client. Team members will be responsible for confirming with the client that the integrated systems is behaving optimally and producing desired results (significantly varies for different aspects of the software-system). <u>Andrea</u> Payment systems. <u>Bryce</u> Inventory and Cloud. <u>Liam</u> User/Admin interaction.	The deployment phase will consist of the final installation processes of the fully integrated software system. Team members will be responsible for finally verifying the operation of their respective system areas – these should have been rigorously tested beforehand and should achieve operational status without much complication. <u>Andrea</u> Payment systems. <u>Bryce</u> Inventory and Cloud. <u>Liam</u> User/Admin interaction.	Should the client request, routine checks will be performed. Additionally, the client can opt for changes and additions to the system. The team will also be responsible for accommodating updates to outsourced parts of the system (i.e. changes in API, terminals, cloud service, etc.). While members should specialize in previously assigned areas, this evolving portion of development warrants a flexible approach with *optional* task partitioning.

Development Timeline

Development and Life Cycle

The development of the software system followed the timeline found preceding this page. This timeline is a reflection of the life cycle model that our team chose: the Waterfall Model. At the time, this seemed to be the most straightforward and easy life cycle model to abide by due to its simplicity and linear fashion. However, significant issues sprung from this choice. Estimated time periods were designed to be generous, but would ultimately prove insufficient since validation succeeded the main block of actual development and implementation. Should the client have changes in requirements, suggested features, or even just criticism/feedback, the time of deployment approaches with little leeway for last minute additions and testing. Put simply, the Waterfall Model is *too* straightforward and easy.



Waterfall Model

(Life Cycles Lecture, Slide 44)

Life Cycle Reevaluation

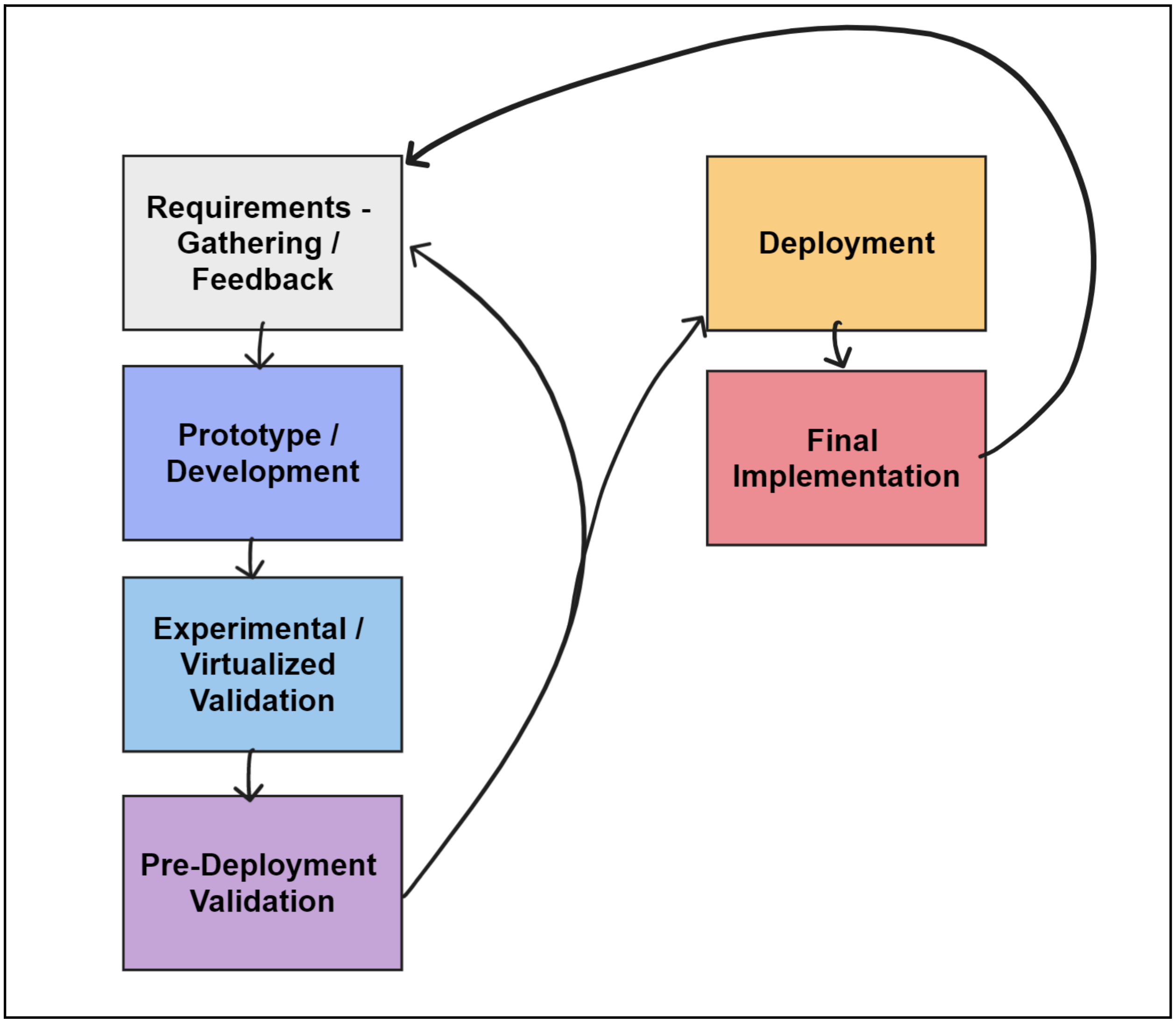
Due to restrictions of the Waterfall Model and the hindrances met with the development of the Clothing Store Point of Sales Software System, the next task would be to build a new life cycle model that meets the requirements and accommodates all parties involved in the development process.

The first requirement would be repetitive feedback–the model must encourage iterative development punctuated by valuable feedback from our clients. The next requirement would be a virtualization of the validation process–in order to ease the workflow of the developers, the model must minimize the amount of actual implementation through a virtualized testing and validation process that can produce experimental output used for feedback. The final requirement would be a repetitive maintenance procedure–building on the idea of iterative development, the system must preserve all experimental builds and documentation even after its deployment, enabling developers to continue to build and add features to the software system with accessibility in mind.

The Fine-Tune Model

This newly constructed life-cycle model is built upon the requirements stated on the preceding page. It begins with an initial gathering of requirements from the client. Once the amount of requirements produces an idea of a cohesive prototype, the software development can begin. Testing will occur throughout the prototyping process, but towards the end, the virtualized validation of the software will commence. The objective is to simulate the environment that the software will work within and produce a clear and concise output of its performance and significant parameters or behaviors. The team will run the performance report through the client for any additional feedback–the client may suggest features that come to their attention or optimizations where applicable. From here, the team has the option of repeating the process on their progressed build, or may take it to its final implementation and deployment. However, this process is not necessarily *final* in that the entire cycle can be repeated again through intended accessibilities, which would encapsulate the maintenance process. This model is named “The Fine-Tune Model”.

The Fine-Tune Model Diagram


Requirements - Gathering / Feedback
Prototype / Development
Experimental / Virtualized Validation
Pre-Deployment Validation
Deployment
Final Implementation

Fine-Tune Model Diagram

Authorship

Bryce Jarboe

*Main Formatting
User Manual Section
Architecture
Data Management
UML Classes
Testing
System Security
Life Cycle Section

Liam Carter

UML Classes (+Diagram)

Testing

Andrea Ruvalcaba

Original Architecture Diagram (replaced)

Testing