2022

Detailed Design for the BestPurchase App

ASSIGNMENT 6 (MODULE 6)

CS 682 INFORMATION SYSTEMS ANALYSIS & DESIGN

ABSTRACT

How do the elements of an activity diagram for a method within the *BestPurchase* app map one-to-one with lines in its pseudocode? How does the creation of various forms of design diagrams iteratively add to the understanding of what is needed in a class model? This report reviews the final steps considered to move the system design of the *BestPurchase* app towards implementation.

EDWARD T MYERS

Table of Contents

Introduction 1

6.1 Updated Class Model 1

6.2 Activity Diagram 2

6.2.1 Focused Sequence Diagram 4

6.3 Pseudocode 5

Conclusion 6

Appendix A – *BestPurchase* Use Analysis & Functional Requirements 7

Overview/Mission Statement 7

User Stories 7

First User Story 7

Second User Story 7

Functional Requirements 8

App Core Functional Requirements 8

Transactional Functional Requirements 8

Convenience Functional Requirements 8

Use Cases 9

First Use Case 9

Second Use Case 9

Appendix B – *BestPurchase* Diagrams & Models 10

State Transition Diagram 10

Sub-States 11

Sequence Diagram 12

Class Model 14

Documenting Classes and Relationships 16

First Business Class Selected: Store 16

Second Business Class Selected: Product 17

Third Business Class Selected: Customer 17

Non-Business Class Selected: ProductList 17

Appendix C – *BestPurchase* Non-Functional Requirements & GUI Sketch 19

GUI Sketch 19

Non-Functional Requirements 20

Appendix D – *BestPurchase* Packages and Data Flow Diagrams 21
Class Model 21
Packages 22
Logical Data Flow Diagram 24
Physical Data Flow Diagram 25
Appendix E – Conceptual Inspiration 27
References 28
Evaluation 29

Table of Figures

Figure 1 - Updated Class Model Diagram 2
Figure 2 – Activity Diagram of the searchByLocation() Method 3
Figure 3 – Focused Sequence Diagram 4
Figure 4 - Check Product Availability Use Case 9
Figure 5 - Specify Contactless Delivery Use Case 9
Figure 6 - State Transition Diagram 10
Figure 7 - Sub-State Diagram 11
Figure 8 - *BestPurchase* Sequence Diagram 13
Figure 9 - *BestPurchase* Class Model 15
Figure 10 - *BestPurchase* GUI Sketch 19
Figure 11 - *BestPurchase* Class Model w/ Packages 21
Figure 12 – *BestPurchase* Logical Data Flow Diagram 24
Figure 13 – *BestPurchase* Physical Data Flow Diagram 26

Introduction

BestPurchase is a retail app intended to integrate automation functionality and provide real-time recommendations to participant customers to augment the traditional in-store experience. Following the previous work of creating physical and logical data flow diagrams, this paper now ventures into the logic that approaches implementation. Using a critical method from a class within the fully realized class model for the *BestPurchase* app, an activity diagram will be created to demonstrate the logical steps the method takes to perform its goal. Afterward, a focused sequence diagram will show an aspect of the activity model and how it uses methods from other class instances in order. Lastly, pseudocode will be used to implement the activity using simulated coding structures and syntax.

6.1 Updated Class Model

The updated class model for the *BestPurchase* app is given below. Methods and attributes bolded in green text represent elements added when considering how the classes will interact with each other during the implementation phase. Overall, more 'set' and 'fetch' type methods allow for easier communication between the classes while keeping the member attributes private. The method searchByLocation() is specially colored in bolded red as that will be the method elaborated upon in the pseudocode and activity diagram to follow.

In StoreDirectory, the function searchByLocation() benefits from helper methods calculateDistance() which calculates the distance between customer and store coordinates and orderByDistance() which returns a list of indices that represent Store instances that are ordered nearest first. These two methods allow searchByLocation() to return an ordered subset of all store locations that are closest to the customer. To support the communication between Store and StoreDirectory, new Store attributes storeLat and storeLong along with their retrieval methods allow StoreDirectory to extract the coordinate pieces from each Store instance for comparison. Similarly, in the Customer class, current_location_lat and current_location_long hold the shopper's coordinate data. In the concrete classes StorePromotion and CustomerHistory, the 'set' and 'fetch' methods were added where needed.

The previous design goal of the Store package was sufficiency, as the goal of the interaction between StoreDirectory and Store is to return to the shopper an ordered list of the closest stores to their current location. The addition of the support methods within the StoreDirectory and Store classes allows the method searchByLocation() to complete this goal when called from the interface GUI. Sufficiency thereby is enhanced with their additions.

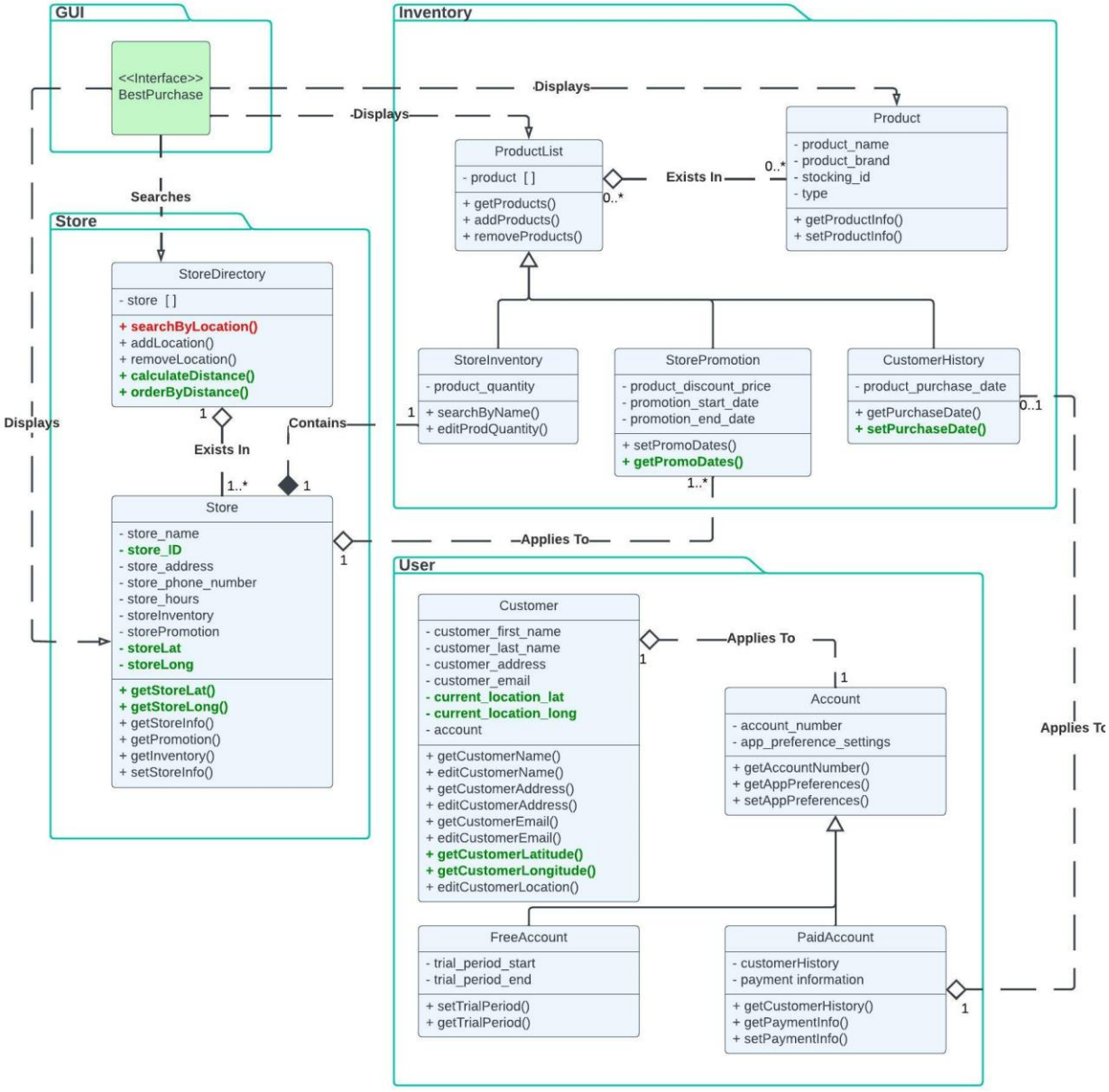

Figure 1 - Updated Class Model Diagram

6.2 Activity Diagram

The activity diagram detailing the searchByLocation() method within the StoreDirectory class is given below. This is an important method to model because the functional requirements and use cases all demonstrate the need for the *BestPurchase* app to provide the means for a shopper to view an ordered list of stores closest to them so that they may select one. All subsequent shopping activities hinge on the success of this process.

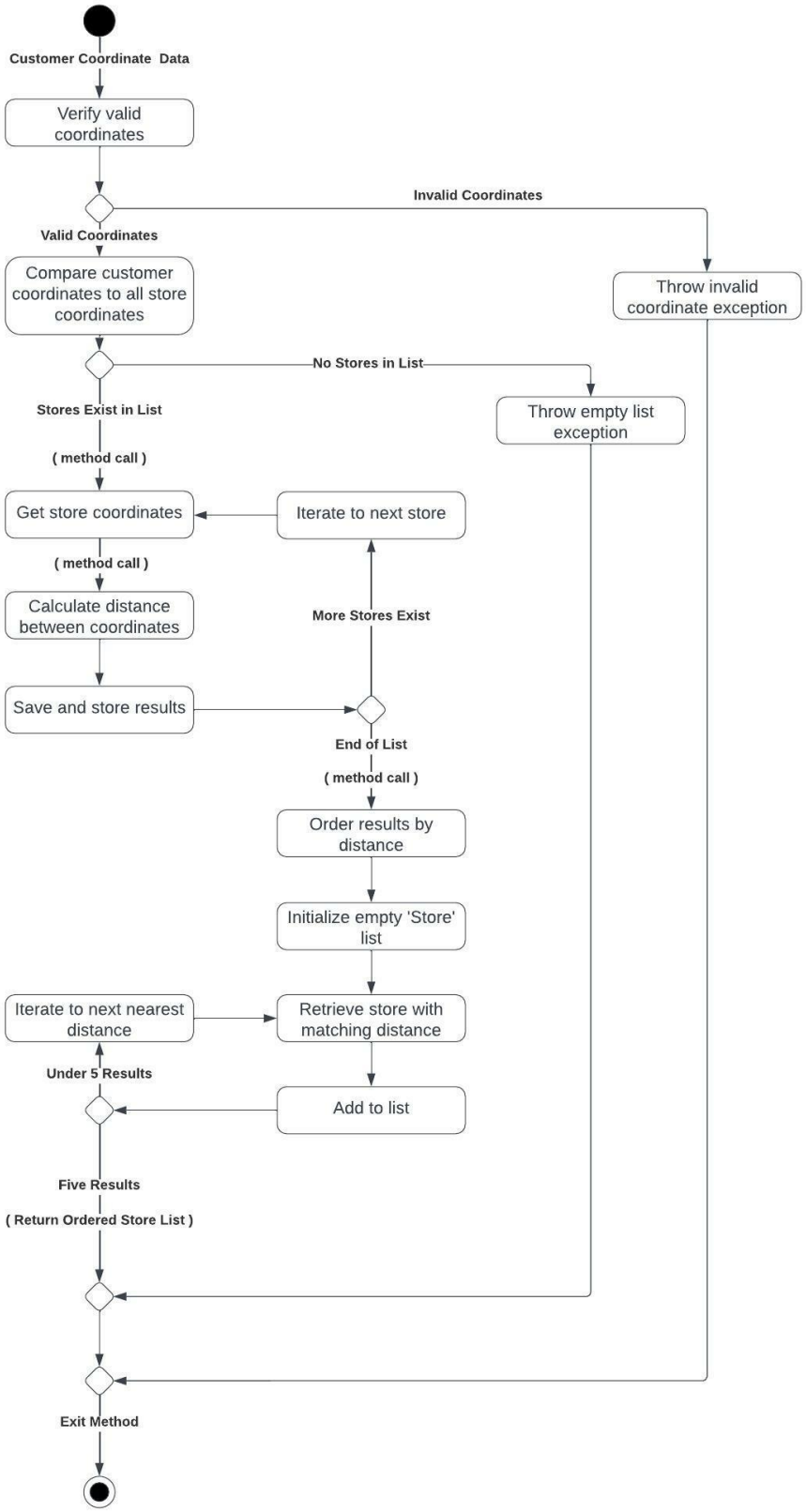

Figure 2 – Activity Diagram of the searchByLocation() Method

The method is passed customer coordinate data, which is first verified for correctness before continuing. If the input is not valid, the method throws an exception and exits. If valid, the method then begins comparing the customer coordinates with all store coordinates. To do this, it initializes a loop that cycles through every store in the list. In each iteration of the loop, the currently viewed store's coordinates are requested through a method call and compared to the customer coordinates. An internal method that calculates distance between the two sets of coordinates is utilized, with the distance value stored in a results list and the loop iterates. At the end of the loop, a list of distances exists that represent all the individual stores' distances from the customer. This list is passed to an internal method which orders the results by distance value, lowest first.

Next, an ordered Store list is initialized and populated using the distances list created from the previous step. Within another loop that iterates up to a maximum of 5 times, the first distance on the list which represents the closest Store location is used to set the first value of the ordered Store list. The loop is iterated to look at the second nearest distance, and so on. The final ordered Store list, containing up to 5 nearby store locations, is returned from the method.

6.2.1 Focused Sequence Diagram

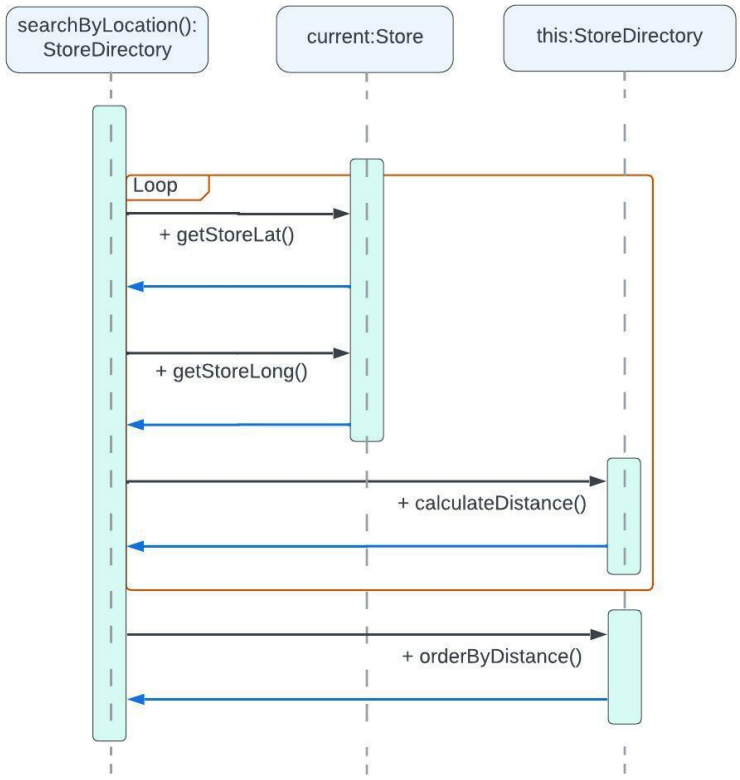

Figure 3 – Focused Sequence Diagram

The sequence diagram above expands on the looping process by which the method searchByLocation() within the StoreDirectory class calls upon external and internal methods to retrieve all store coordinates to compare with the known customer coordinates.

After the point of validating the customer coordinate data and checking to make sure there are Store instances in the list, the sequence diagram starts by initiating a loop which pulls the latitude, then the longitude of the current Store instance via the methods getStoreLat() and getStoreLong() respectively. The sequence continues by invoking an internally located method within the StoreDirectory class named calculateDistance() that gets passed the customer and current store coordinates and returns a numeric distance value. The loop continues iterating until there are no more stores in the list. Lastly, the accumulated distance values are ordered using another method internal to the StoreDirectory class, named orderByDistance().

6.3 Pseudocode

The following pseudocode of the searchByLocation method of the StoreDirectory class uses the methods and attributes laid out in the class model and follows the task order and behavior simulated in the activity and focused sequence diagrams. As noted in the code comments, the structure of the pseudocode is Java inspired. Actual code is laid out in bolded blue text to distinguish it more easily from comments.

```
/* The function searchByLocation is designed to receive the coordinate location of a customer and return a list of
the 5 stores nearest to the customer ordered nearest first. Pseudocode scheme is Java inspired, with some
operators swapped for English descriptions to improve understanding. */

searchByLocation (float customerLat, float customerLong)
{
 /* The first step is to validate the input.  Acceptable ranges for coordinates are -90 to 90 for latitude, and -180 to
180 for longitude.  If the ranges are not within the boundaries.  Validity checking stops here, but in concept to
improve robustness it would be beneficial to check for correct data type, and if both coordinate pieces are non-null.
If an error is found, an exception is thrown, and the program returns nothing. */

   if (customerLat < -90 OR customerLat > 90 OR customerLong < -180 OR customerLong > 180)
   {    throw new Exception ('Customer coordinate data out of range.');     }

/* Begin a loop to iterate through the local list of store instances to check distance against.  While the list is not
empty, keep going.  Store data structure is an array called storeList[]. */

   int index = 0;                              // iterator through Store instance array
   float storeDistances = new float[];         // array designed to hold store distances

   // Another robustness check, make sure the storeList[] array has elements in it.
   if (storeList[].length == 0)
   {    throw new Exception ('No stores exist in store list.');    }

   while (index < storeList[].length)
   {
      float distance = 0.0;       // initialized distance variable
```

```
        // local copy of current store coordinates
        float currentStoreLat = storeList[index].getStoreLat();
        float currentStoreLong = storeList[index].getStoreLong();

        // invoke this.calculateDistance() method inside class to calculate distance
        distance = this.calculateDistance(customerLat, customerLong, currentStoreLat, currentStoreLong);
        storeDistances[index] = distance;
        index = index + 1;                  // iterate to the next index location
    }

/* Invoke this.orderByDistance() method inside class to return an array of indices that represent Store instance
locations within the storeList[] array.  The contents are sorted by the distance values passed to the function, lowest
first. */

    int storeIndices = this.orderByDistance(storeDistances[]);

/* Lastly, perform a loop that creates an array of the five nearest store locations
to the customer using the indices from the sorted list */

    orderedStoreList = new Store [5];
    for (i = 0; i < 5; ++i)
    {       orderedStoreList[i] = storeList[storeIndices[i]];       }

    return orderedStoreList;                    // Return the list back to the calling entity
}
```

Conclusion

This report takes the previous design considerations of the *BestPurchase* app and demonstrates the steps by which the product would move toward implementation. The class model diagram is now polished to the point of identifying several non-intuitive methods and attributes that improve the sufficiency goal of the system packages. The activity and focused sequence diagrams show an example of the system behavior at the sub-method level, which eliminates the guesswork in what code functionality to include and why. Lastly, pseudocode of the diagrammed method reveals the considerations and mindset of the designer to the system developer writing its code, and potentially begins a dialogue which could avoid problems and oversight from both ends.

Appendix A – *BestPurchase* Use Analysis & Functional Requirements

The content below references many of the key requirements and use cases developed to explain how the *BestPurchase* app would work, what type of shoppers would use it and for what reasons. Below are the mission statement, functional requirements, user stories and use cases created to give body to the concept.

Taken from *Requirement Analysis and Operational Characteristics for the* BestPurchase *App,* MET CS 682 Assignment 3, Term Project Part 1 by Edward T. Myers, 9/28/2022.

Overview/Mission Statement

BestPurchase is a mobile-based app for a grocery supermarket that complements the traditional grocery shopping experience independent from any physical store location. Shoppers will have the luxury of checking stock, loading a virtual shopping cart, and paying for goods after creating an account on the app. *BestPurchase* combines convenience with a touchless experience with its ability to schedule orders for parking lot pickup or at-home delivery services. Information about previous orders is saved on the app, allowing the shopper to quickly add common items to the virtual cart and receive directed promotions based on their purchase history. Shoppers will be able to use as much or as little of *BestPurchase* as they desire, based on if they wish to browse the inventory of a local store, check for deals, or place an order complete with a delivery option. As you travel, *BestPurchase* identifies participating stores within your area each time you open the app and offers a list of store amenities offered at each location. Through its customer feedback form, shoppers report issues and suggest new content to further improve their shopping experience.

User Stories

First User Story

As an illness-conscious shopper, I want to take advantage of a fully contactless shopping experience from beginning to end by placing an order, paying, and arranging for contactless delivery all through the app so that I can receive groceries with little person to person interaction and not spread sickness.

Second User Story

As an organic food and produce shopper, I only want to use part of an app to check if my favorite fruits and vegetables are in stock but then shop in person so that I can verify I am getting the freshest produce available.

Functional Requirements

App Core Functional Requirements

1. *BestPurchase* shall allow shoppers to create an account and password.
2. *BestPurchase* shall allow shoppers with an account to add a method of payment to their account (Roche, 2019).
3. *BestPurchase* shall save contact and address information about the shopper within the app.
4. *BestPurchase* shall allow shoppers to select a store by address or proximity to their location (Wadland, 2020).
5. *BestPurchase* shall allow shoppers to see the brand, stock quantity and price of items sold from the selected store location (Roche, 2019).

Transactional Functional Requirements

6. *BestPurchase* shall allow shoppers to add items to, edit quantity of, or delete items from the app shopping cart (Roche, 2019).
7. *BestPurchase* shall allow shoppers to add manufacturer or store coupon codes to the order for discount (Wadland, 2020).
8. *BestPurchase* shall allow shoppers to pay for their order with one or more valid payment methods.

Convenience Functional Requirements

9. *BestPurchase* shall save the shopper's default store location and purchase history within the app.
10. *BestPurchase* shall promote products or suggest sales based on the shopper's purchase history (Wadland, 2020).
11. *BestPurchase* shall allow shoppers to choose between delivery or parking lot pickup options to receive their order.
12. *BestPurchase* shall allow shoppers to select from a pickup or delivery time block:
    a. The time block shall be an hour in duration.
    b. A shopper may only select a time block if it is available.
    c. The window of time for a shopper to select a time block and receive their order begins 15 minutes after successful order completion and ends after two full days have passed.
13. *BestPurchase* shall allow shoppers to add notes to their order for clarity or to relay delivery details.
14. *BestPurchase* shall allow shoppers to provide feedback through the app to convey customer satisfaction or report issues.

Refer to Appendix E for product inspiration.

Use Cases

First Use Case

Use case Name	Check Product Availability	
Actor:	Organic Food & Produce Shopper	
Description:	This use case follows the steps a produce shopper takes to verify if pineapples are in stock at the closest participating store before physically traveling to the store and buying one. This case applies to functional requirements 4, 5, 9 and 10.	
Pre-condition:	The produce shopper is a current *BestPurchase* account holder and has used the app several times. The produce shopper has the app open on a mobile device and is logged in.	
Step #	**Actor**	**System**
1	The produce shopper selects 'Locations Near Me'	The app displays a list of the first 5 stores geographically near the shopper, sorted nearest to furthest
2	The produce shopper selects the desired store	The app displays the selected store location details, including store name, address and contact info
3	The produce shopper selects 'Shop Now'	The app displays the weekly digital promotions at the store location, along with recommended items based off the produce shopper's purchase history
4	The produce shopper types 'pineapple' into the search field and taps the 'Find' button	The app displays all products that have 'pineapple' in its name that are available at that store location
5	The produce shopper selects 'Produce – Whole Pineapple' from the list	The app displays the brand, stock quantity and price of the selected product
6	The produce shopper reads the quantity and shuts down the app	
Alternate Courses:	4. The produce shopper scrolls through the inventory displayed by the app manually until the correct pineapple product was found. The produce shopper then selects 'Produce – Whole Pineapple'. The app displays the brand, stock quantity and price of the selected product	

Figure 4 - Check Product Availability Use Case

Second Use Case

Use case Name	Specify 'Contactless Delivery' Upon Checkout	
Actor:	Illness-Conscious Shopper	
Description:	This use case follows the steps an illness-conscious shopper takes to complete a grocery order, request delivery to their house and add a note to the order to leave the groceries on the front porch and ring the doorbell. This case applies to functional requirements 3, 11, 12 and 13.	
Pre-condition:	The illness-conscious shopper is a current *BestPurchase* account holder and has used the app several times. The illness-conscious shopper has the app open on a mobile device and is logged in. The illness-conscious shopper has a payment method already associated with the account and has completed orders successfully in the past. The illness-conscious shopper has selected a store, chosen all the desired items, added them to the cart and selected payment. The illness-conscious shopper is now ready to specify how to receive the order.	
Step #	**Actor**	**System**
1	The illness-conscious shopper selects 'Choose Delivery Options'	The app displays the options 'Pickup at Store' or 'At-Home Delivery'
2	The illness-conscious shopper selects 'At Home Delivery'	The app displays the illness-conscious shopper's saved address and the option 'Pick a different Location'
3	The illness-conscious shopper selects their saved address	The app displays a list of hourly time boxes available for delivery
4	The illness-conscious shopper selects the preferred time box	The app displays the message 'Add a note to the order?' along with a text box
5	The illness-conscious shopper types a message into the text box requesting the groceries to be left on the front porch and to ring the doorbell.	The app completes the order and displays an order confirmation number along with the expected time window for delivery.
(more)		
Alternate Courses:	3. The illness-conscious shopper selects 'Pick a different location'. The app displays a fill form to receive the address details. The illness-conscious shopper manually types the desired address and clicks OK to proceed.	

Figure 5 - Specify Contactless Delivery Use Case

Appendix B – *BestPurchase* Diagrams & Models

The content below elaborates on the functional requirements and use cases by portraying the various diagrams that explain behavior, structure, and order of operations. Portrayed are the state transition diagram with substates, the sequence diagram and class model.

Taken from *Requirement Analysis and Operational Characteristics for the* BestPurchase *App,* MET CS 682 Assignment 3, Term Project Part 1 by Edward T. Myers, 9/28/2022.

Taken from *Diagrams and Design with UML for the* BestPurchase *App,* MET CS 682 Assignment 4, Term Project Part 2 by Edward T. Myers, 10/1/2022.

State Transition Diagram

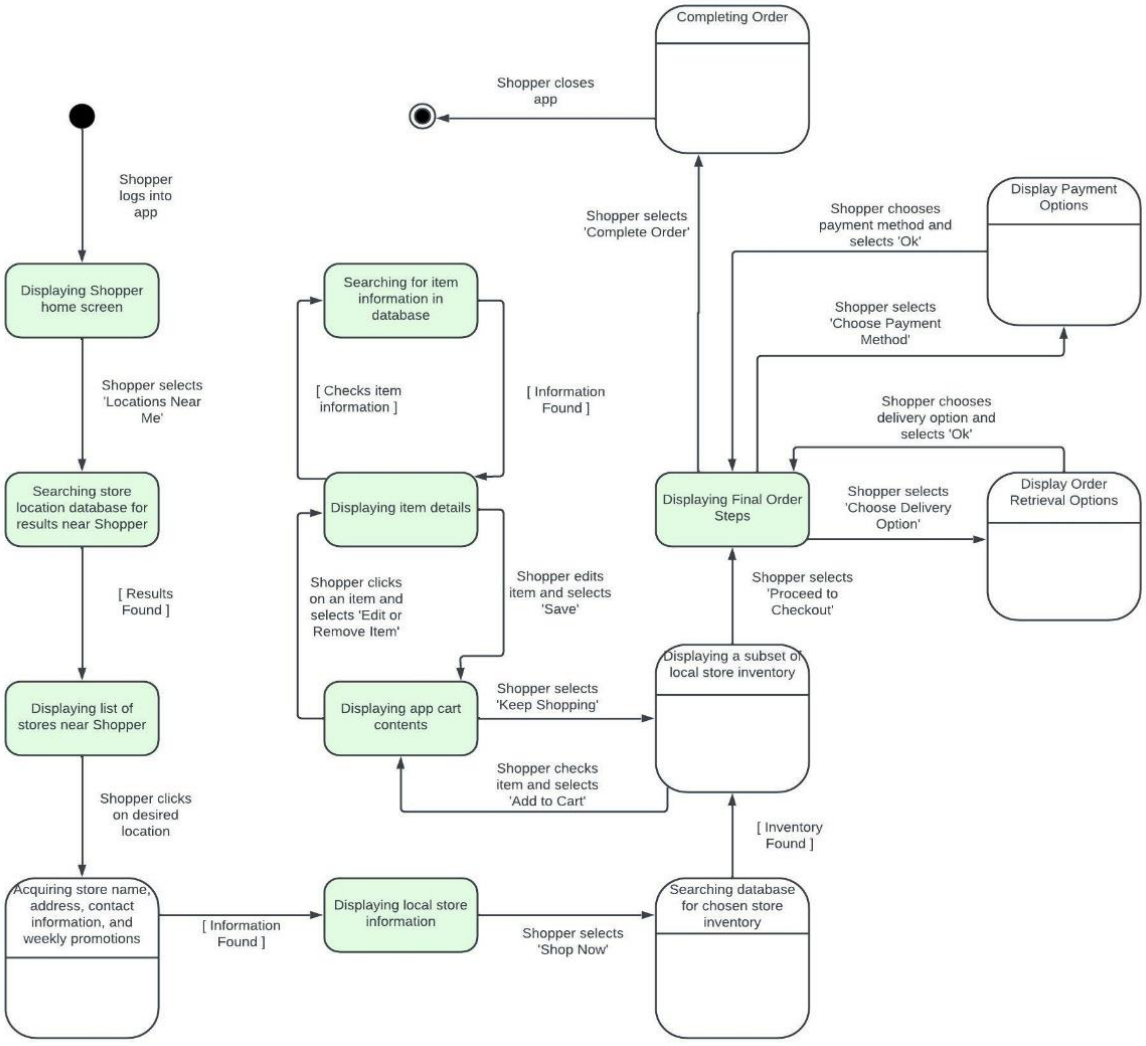

Figure 6 - State Transition Diagram

The shopper starts by logging into their app and searching for a participating store near their location. After selection of the store, the app pulls information about the store for display. The

This state transition diagram gives the *BestPurchase* process flow for how a shopper navigates the core purpose of the app: to select a store, view inventory and make an online purchase.

Some pre-conditions to this diagram are that the shopper is an account holder with payment options and home address already set up in the application (as per functional requirements 1, 2, and 3). Also assumed are specific hardware necessities, such as location services being enabled on the accessing mobile device.

shopper selects 'Shop Now' to begin shopping at that location. The app pulls the current inventory for the selected store and displays it. From here the shopper can move in multiple directions, either adding an item to cart or proceeding to the checkout. If viewing the cart, the shopper can edit quantity or remove an item present. The app checks any new quantity entered against the current stock in the database before allowing the change. Once in checkout, the shopper must confirm delivery options and payment method before proceeding to completion. Each of those can be expanded in their own composite states as there are multiple choices and lookups. The shopper completes the order, prompting the system to perform the composite tasks involved (decrease in-stock inventories of purchased items, perform payment request, send delivery preferences to order prep team, etc.). Lastly, the shopper closes the app.

State transition diagram design rules and creation credit: (Dennis, 2021)

Sub-States

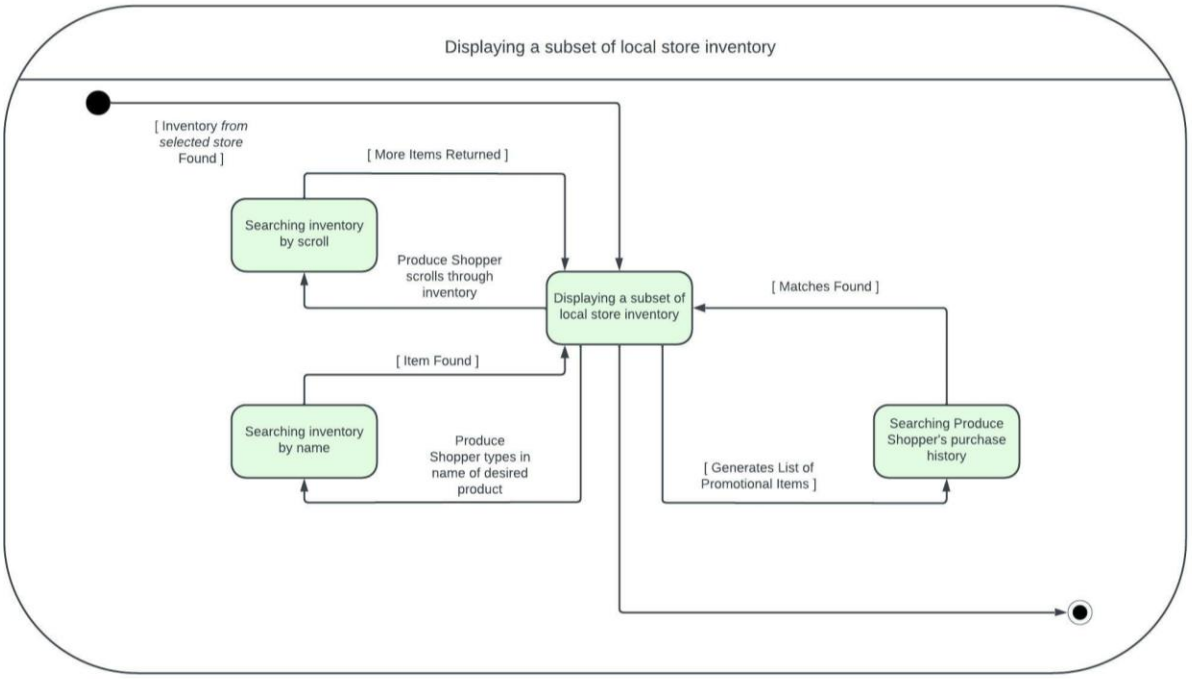

Figure 7 - Sub-State Diagram

This sub-state diagram gives the composite breakdown of the 'Displaying a subset of local store inventory' state introduced in Figure 3. This maps to the scenario outlined in Use Case #1, a produce shopper's interaction with the *BestPurchase* app while checking to see if a pineapple is in stock at the nearest store.

Combining the first state diagram with this sub-state diagram, the produce shopper logs into the app and selects the closest store to get information about. After selecting the location, the produce shopper searches for a product using a search field within the local store inventory page. Alternatively, the produce shopper could have scrolled through the app and located the item manually. At the same time, the system uses the produce shopper's history to add suggestions to the inventory readout. The produce shopper sees the desired inventory item and checks its stock quantity.

State transition diagram design rules and creation credit: (Dennis, 2021)

Sequence Diagram

The sequence diagram in Figure 2 demonstrates the interactions between the shopper, the app interface, and the background entity classes that communicate to provide the shopper with the requested information. The shopper begins by selecting the 'Locations Near Me' option on the app, which then runs a searchByLocation() method that is passed information about the shopper's location and returns stores that are within a distance radius of the shopper. The app interface sorts these results for the shopper in nearest-first order. After the shopper selects the desired store from the list, the getStoreInfo() method returns information about the store for display through the interface, including name, open hours, contact info, etc. The next three steps performed by the shopper (selecting 'View all Promotions', selecting 'Recommended for You', and selecting 'Shop Now'), yield very similar communication patterns between classes.

After selecting 'View All Promotions' from the interface, the app requests the Promotion instance from the selected store using getPromotion(), then uses that Promotion instance to request its list of products with prices using getProducts(). Now that the app has access to a list of products within that specific promotion, it communicates with each Product instance in a loop to request its type, brand and quantity using the method getProductInfo(). The app, now with all the information of the discounted products, displays the information to the shopper through the interface.

The shopper then desires to view recommendations by selecting 'Recommended for You' from the interface. The app calls upon an instance of the Customer object that relates to the logged in shopper's account and retrieves Product instances previously purchased using the getCustomerHistory() method. The last two steps of the previous selection are now repeated, with the app calling upon the function getProducts() from the CustomerHistory instance and

then looping through the resulting Product instances, obtaining the product information using

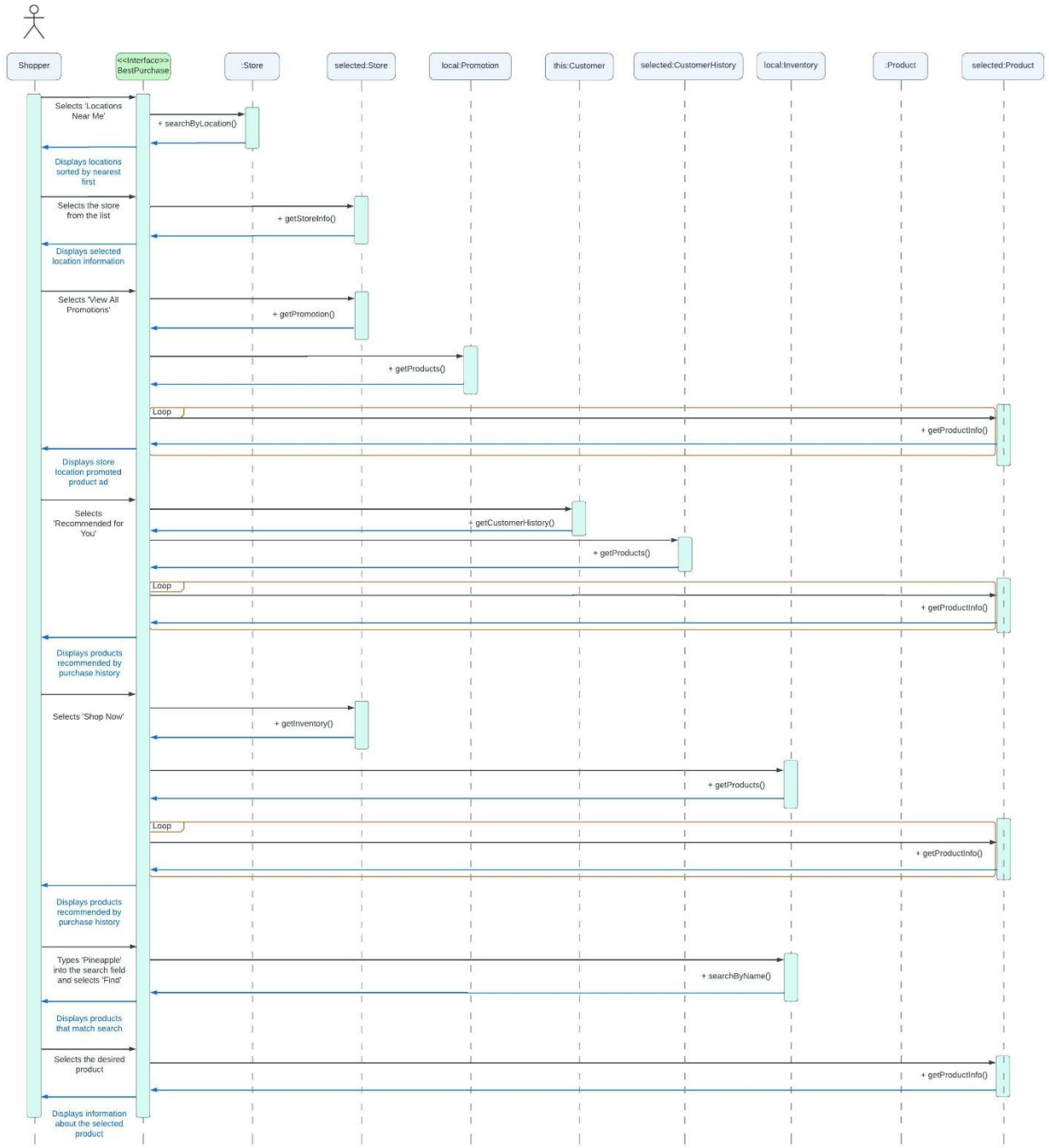

Figure 8 - *BestPurchase* Sequence Diagram

the getProductInfo() method. The shopper now would like to see the selected store's full inventory of products and selects 'Shop Now'. The app communicates with the selected Store instance and requests its full inventory using the getInventory() method. The app then requests from the chosen Inventory instance all products contained within by calling the getProducts() method. Again, the app requests the product information with each Product instance in a loop

through the method getProductInfo() which is relayed back through the interface to the shopper.

Now the shopper is ready to search for a specific product, in this case a pineapple. The shopper decides to enter the word 'pineapple' into the search bar and select 'Find' or magnifying glass. The app calls a method within the Inventory instance called searchByName() which is passed the keyword string typed by the shopper and returns all products having that string in its name. The results are passed back to the shopper through the app interface. Lastly, the shopper finds the product they are looking for in the list, and selects it through the app. The app communicates with the Product instance matching the selection and requests its information, specifically its quantity, using the getProductInfo() method. The app relays the information back to the shopper through the app interface.

Class Model

The class model in Figure 3 translates the entities identified in the sequence diagram into a class model framework. The 'search' and 'get' functions are all represented in the available methods for each class, as well as the logical private attributes that they are meant to extract. Communication between sets of entities in the sequence diagram maps to associations in the class model, with multiplicity identifiers demonstrating how many of one can be associated with another. The *BestPurchase* interface, identified as the <<Interface>> entity, acts as the passthrough for the shopper as they select options from the GUI (*See Appendix B –* BestPurchase *GUI Sketch for the* BestPurchase *GUI layout and how app options map to the use case used for the scope of the class model*). Here are some key differences between the communication paths of the sequence diagram and the logical interpretation to the class model:

1) From a text analysis of the use case referenced in 4.1 above, the app is to search for the stores nearest to the shopper and return a sorted list of nearby locations. To implement this using hierarchical classes, it makes more sense to encapsulate all Stores in a data structure within the StoreDirectory entity, which possesses the method searchByLocation() called from the app. Further, the store entity should have a getLocation() method which the StoreDirectory entity uses in the operation of its searchByLocation() method.
2) The shopper from the use case and sequence diagram performs several similar actions to look at different products, first by viewing the store promotion, then the recommended products based off purchase history, and finally the full inventory offered at the selected store. In the sequence diagram, the app communicated with each of these concrete entities (StoreInventory, StorePromotion and CustomerHistory) directly by accessing the same method getProducts() which returns the data structure attribute holding the Products from each. This parallel led to the realization that all these entities were lists of products, and that each would benefit from becoming a subclass of a

parent class ProductList. ProductList would hold the data structure of Products as an attribute and the method getProducts() which the subclasses would inherit. The benefits of modularity and reusability of code in this case were greater than the efficiency of denormalizing ProductList down into its subclasses, or even into Store itself. The dependency association from the app interface thereby leads to ProductList instead of StoreInventory, StorePromotion, and CustomerHistory individually.

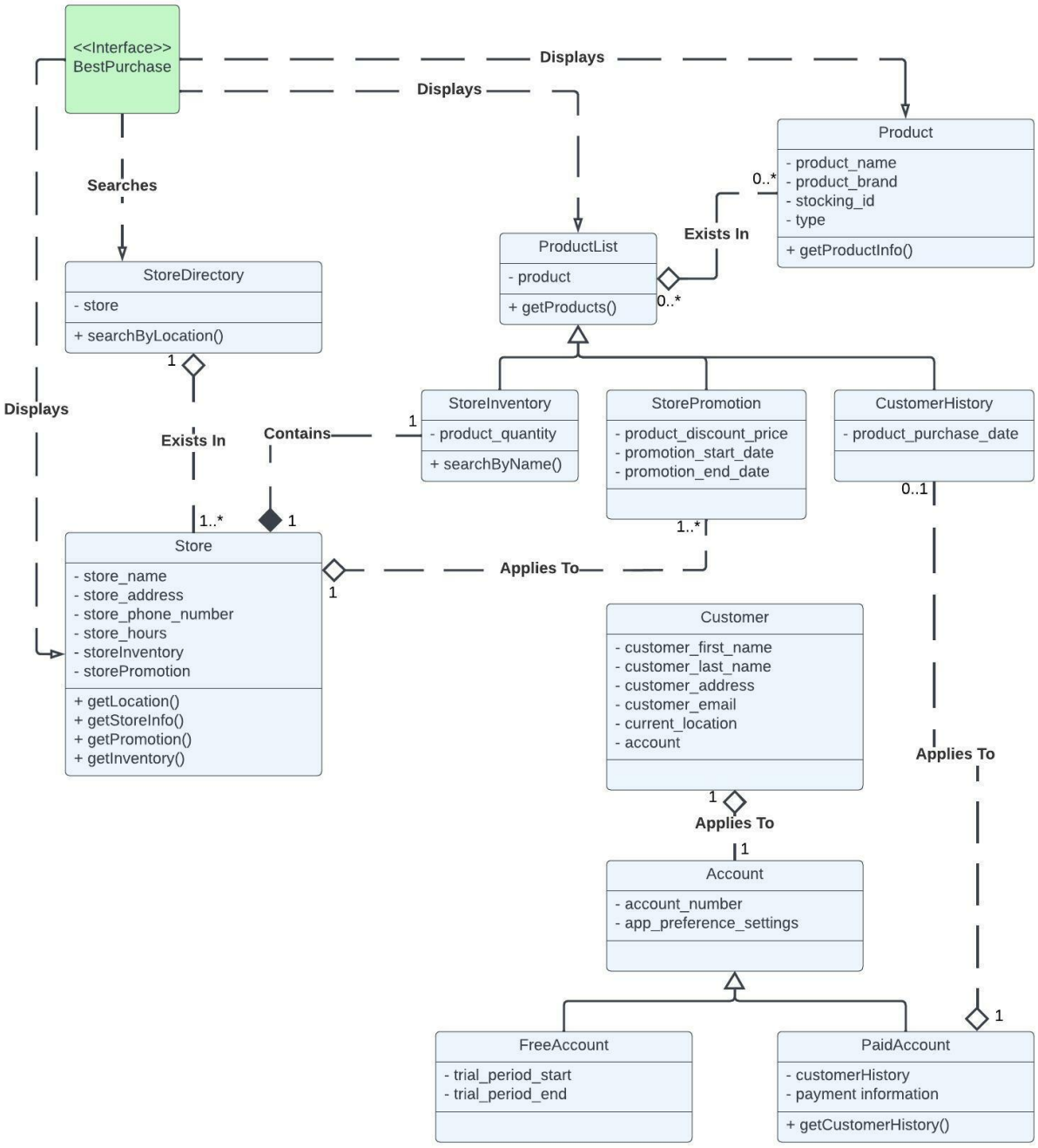

Figure 9 - *BestPurchase* Class Model

3) The use case identified the precondition that the shopper was already an account holder with the *BestPurchase* app. Therefore, entities were created to reflect the Account entity and its subclasses FreeAccount and PaidAccount. In this scenario, only paid account members get their purchases tracked and thereby receive recommendations based on their purchase history. For this to make sense, the getCustomerHistory() method was moved from the Customer entity down to the PaidAccount entity.

Examples of class inheritance are seen in the subclasses underneath ProductList and Account. The app interface has a dependency relationship with the Product, Product List, Store, and StoreDirectory entities. The bulk of the remaining relationships are aggregation relationships as they could exist independently from the entity they are associated with, except for Store to StoreInventory which is a composition relationship. The idea here is that every active store must have an inventory of products, or it makes no sense to be seen on the store directory.

Documenting Classes and Relationships

First Business Class Selected: Store

Importance for the Design: The Store class is one of the first things a shopper interacts with using the *BestPurchase* interface. All results regarding products, promotions, etc. rely on a particular store being selected first. Each store possesses a specific inventory, promotion, and location different from other stores, making the store selection decision very important to the success of finding a product.

Relationship with other classes:

Dependency Relationship

<<Interface>> BestPurchase – After a store location is selected from the StoreDirectory, the app directly communicates with the selected Store instance to make additional requests, including calling getPromotion() and getInventory().

Aggregation Relationship

StoreDirectory – Many Store instances may be held in the StoreDirectory, which can use its searchByLocation() method to invoke each Store's getLocation() method for determining nearest proximity to the shopper.

StorePromotion – A Store instance may have many StorePromotion instances over time, considering the tracking old promotion histories. The app accesses the Promotion through the selected Store instance's getPromotion() method.

Composition Relationship

StoreInventory – A Store instance must have one StoreInventory instance as a prerequisite to being selected by a shopper from the *BestPurchase* app.

Second Business Class Selected: Product

Importance for the Design: The Product is the core reason why anyone would wish to use the app in the first place. Without the Product class and its informational attributes, the shopper would not be able to complete the core purpose of the use case. No product inventory, promotions or recommendations based on purchase history would be visible. Moving outside this limited scope, dependent features such as adding to cart, scrolling for products, etc. would not be functional.

Relationship with other classes:

Dependency Relationship

<<Interface>> BestPurchase – After the app has selected to see either the store inventory, the store promotion or recommendations based on purchase history, the app eventually accesses the information within the Product class after several calls through the class hierarchy. Information about each Product instance is returned through the getProductInfo() method to the app then relayed to the shopper.

Aggregation Relationship

ProductList – A Product instance may be in multiple ProductLists, and a ProductList may have multiple Product instances.

Third Business Class Selected: Customer

Importance for the Design: The Customer class holds important information about the customer, including methods of communication, home address, name, account, and current location. If this information were not known, then the 'Find Near Me' app option would not be possible as the app would not be able to compare the location of the shopper to any of its stores.

Relationship with other classes:

Aggregation Relationship

Account – One Customer instance must be associated with one Account instance and vice versa. Since the shopper's account is tied to the Customer class, functionality outside of this scope such as delivering completed orders to the shopper's residence would also be broken.

Non-Business Class Selected: ProductList

Importance for the Design: The ProductList class is an entity identified due to the duplication of methods and attributes called upon from the app interface when accessing the concrete entities StoreInventory, StorePromotion and CustomerHistory. Each originally possessed the

product attribute as well as the getProducts() method from the sequence diagram. The ProductList class is logically useful in that it becomes the generalized parent of the three classes and takes the common factors from each of them. The three subclasses then inherit the attribute and method for their use, improving the efficiency of the program design (Dennis, 2021).

Relationship with other classes:

Dependency Relationship

<<Interface>> BestPurchase (cascading) – When the shopper wishes to see the selected store promotion, inventory, and recommendations based on their purchase history, the app interface communicates with the subclasses of the ProductList class.

Generalization Relationship

StoreInventory, StorePromotion, CustomerHistory – These classes are subclasses of the ProductList class. ProductList holds the product attribute and getProducts() method which each subclass inherits for its own interaction with the app interface.

Aggregation Relationship

Product – A ProductList may have many Product instances associated with it, and each Product may be a part of multiple ProductLists.

Appendix C – *BestPurchase* Non-Functional Requirements & GUI Sketch

The content below reviews specifically the quality & constraint requirements for the *BestPurchase* app and uses a GUI sketch to simulate the graphical layout to the potential shopper.

Taken from *Requirement Analysis and Operational Characteristics for the* BestPurchase *App,* MET CS 682 Assignment 3, Term Project Part 1 by Edward T. Myers, 9/28/2022.

GUI Sketch

The GUI sketch provided is a display that views a selected store's inventory after a shopper has successfully logged in and chosen a location.

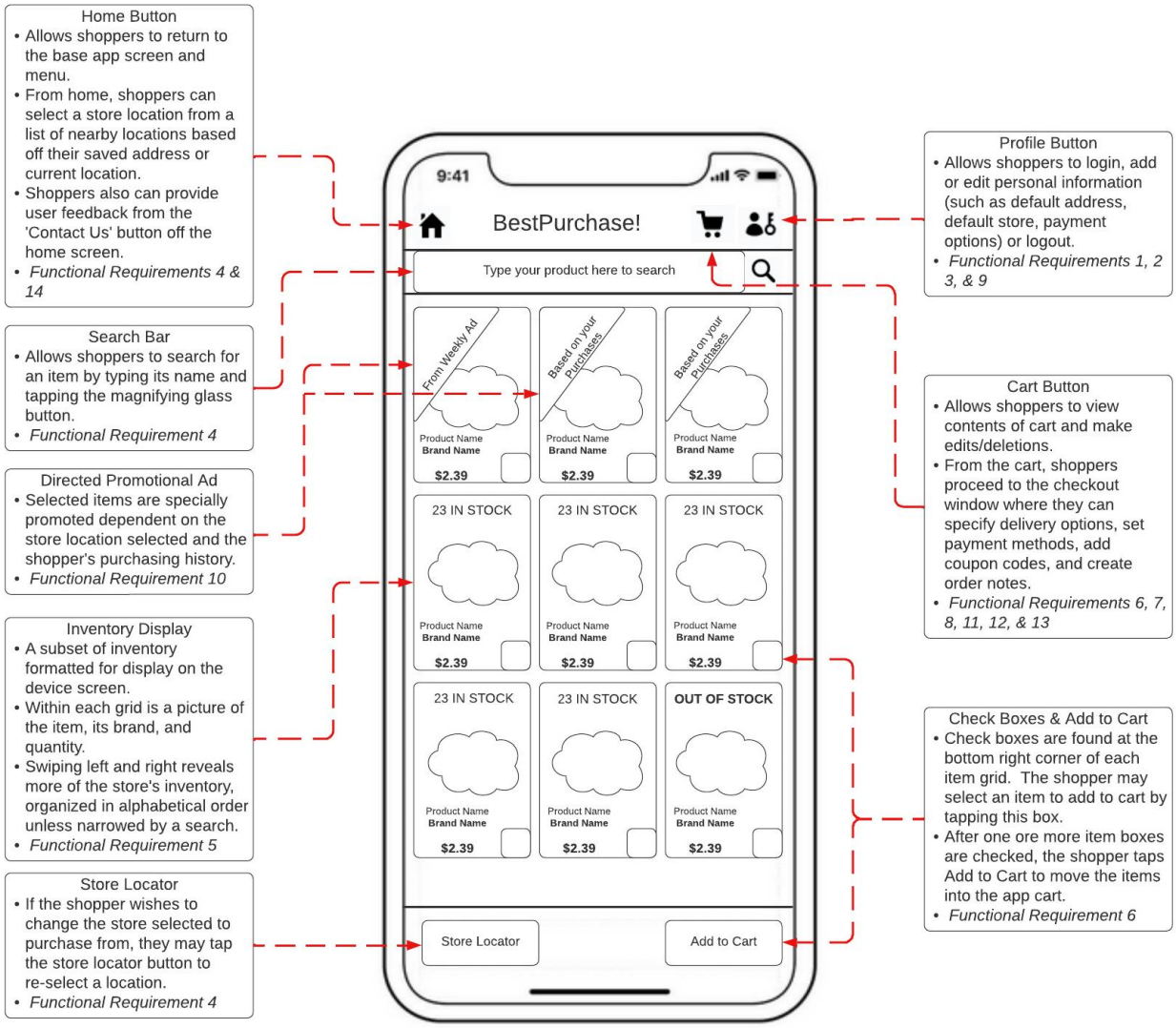

Figure 10 - *BestPurchase* GUI Sketch

GUI wireframe stock template credit: (Krajcovic, 2019)

Refer to Appendix E for product inspiration.

Non-Functional Requirements

1. *BestPurchase* shall operate on devices running IOS 14.0 or later and Android 10.0 or later.

Apple IOS and Android are the most used operating systems on mobile devices (David, 2019). The respective versions were the standard operating system to ship in Apple and Android smartphones in the 2020 production cycle (Muchmore, 2020). Therefore, these operating systems and newer represent a large share of phones that are in use within a standard renewal period for retail phone contracts. Both mobile operating systems are supported by their respective app stores, which vets all compatible versions of a particular app against the minimum version standard allowed. By ensuring that the *BestPurchase* app runs on older phones that have upgraded their operating system to at least these versions or that it runs on new phones purchased within the past two years, the app has a better chance of remaining current for the immediate future. This decision eliminates the need to create versions that work on depreciated operating systems and saves on excess design and coding time.

2. *BestPurchase* shall use a 256-bit Advanced Encryption Standard to secure its user account and payment transactions.

Potentially hundreds of thousands of people will sign up for and use the *BestPurchase* app due to the convenience of shopping for groceries online. The popularity of the app combined with the volume of transactions will make the app, supporting databases and server infrastructures a target for hackers or other cyber threats. 256-bit encryption is a high standard for protecting transmissions with a public/private key system (Bell, 2017). This is not a standalone measure and must be considered along multi-factor authentication such as a confirmation text message or integration with an authentication app. If inadequate security protocols were used to protect transmissions from the *BestPurchase* app, the publicized hacking and data loss would cost the company legal resources and negatively affect its reputation among its users.

Appendix D – *BestPurchase* Packages and Data Flow Diagrams

The content below shows the organization of the class model for the *BestPurchase* app in packages defined by specific design goals. Additionally, data flow characteristics between logical aspects of the program and physical components specify the nature of the entity communication that needs to exist during implementation.

Taken from *Design Goals and Data Flows for the* BestPurchase *App,* MET CS 682 Assignment 4, Term Project Part 3 by Edward T. Myers, 10/11/2022.

Class Model

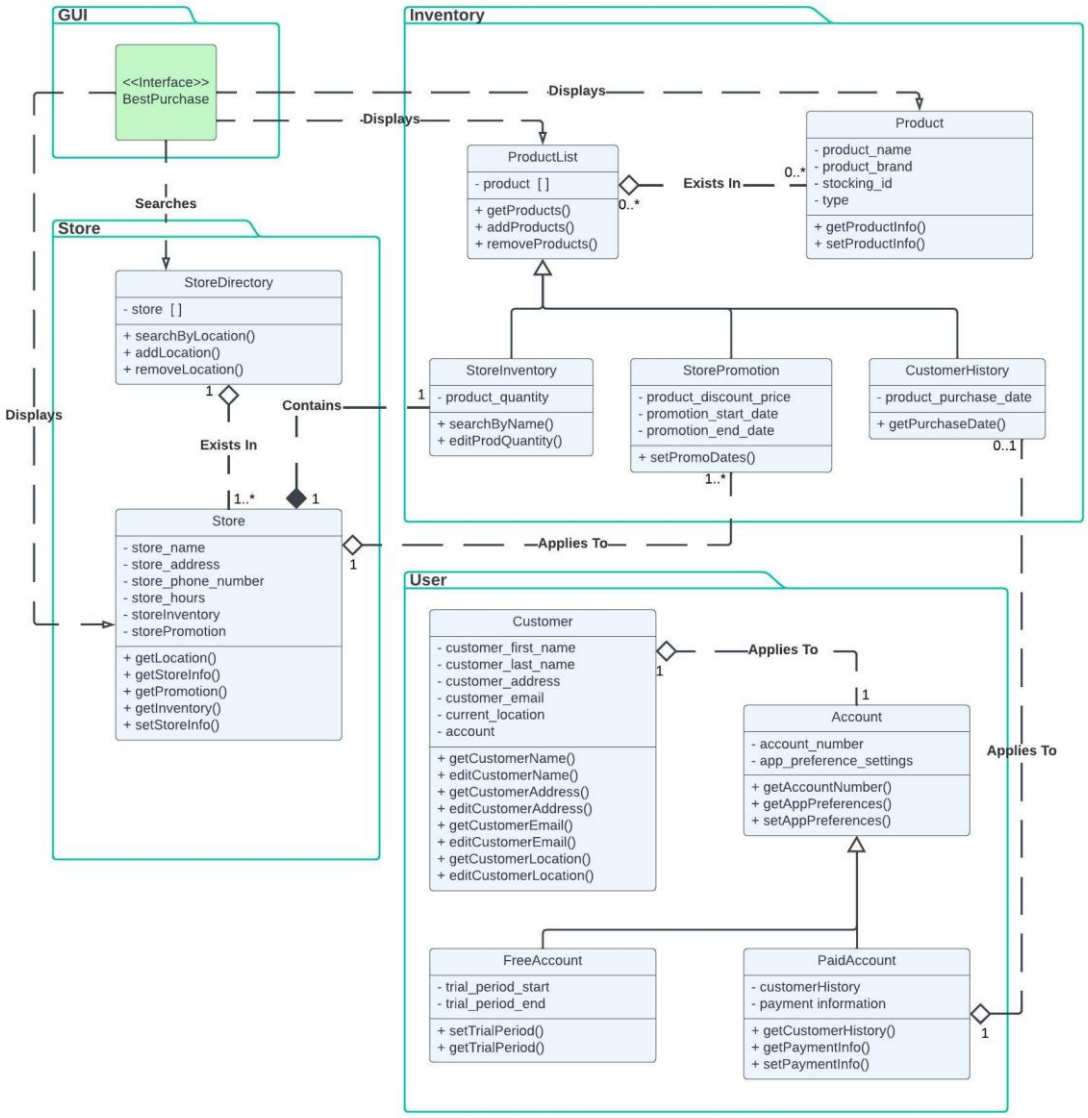

Figure 11 - *BestPurchase* Class Model w/ Packages

Above is an updated class model for the *BestPurchase* app with entities divided into logical packages based on design goals elaborated on in section 5.2 (*see Appendix B for the previous*

BestPurchase *app class model inspiration*). Attributes and methods within each entity have been expanded to demonstrate the hierarchy of communication that occurs between the interface and the package elements. For an example of how the app calls upon these methods to fulfill a request, see the Appendix B *BestPurchase* Sequence Diagram).

Packages

The *BestPurchase* App is a mobile device interface meant to be used by many people in several different ways for viewing information about a grocery store's inventory and potentially making a purchase (*See Appendix A for use cases and functional requirements considered for the* BestPurchase *app*). It is worth mentioning that beyond sufficiency, flexibility and reusability, this app needs to consider design goals that reflect large-scale, simultaneous use (efficiency), sensitive data communication (security) and acceptable uptime as necessitated by eCommerce (reliability). The packages identified section 5.1 are evaluated using these primary design goals below. The GUI package needs little explanation and is omitted from the breakdown.

- **Package name**: Inventory
- **Design goals for this package**: The primary design goals of this package are sufficiency and reusability.
- **Design tradeoffs for this package**: Sufficiency is a measure of how well a module handles the given requirements, while reusability demonstrates how components of a module can be recycled and used in other parts (Dennis, 2021). From the use cases and functional requirements outlined in Appendix A, the *BestPurchase* app interface needs to interact with products available for purchase that are members of several different lists, including what is in the selected store inventory, within the store's promotional ad, and from the shopper's purchase history. This goal of sufficiency is met within the inventory package. Next, the StoreInventory, StorePromotion and CustomerHistory entities are subclasses of the ProductList superclass. As mentioned above, each one of these entities is essentially a list of products with different focus. As a result, the product data structure and several methods can be generalized into a superclass and reused as the foundation of the subclasses, making the implementation less redundant.

  Cohesion is high within the Inventory package as all entities involve products or lists of products. No attributes from within any of the entities pertain directly to any entity outside of the package without cause. Coupling within the package is high, as all entities are linked with either an aggregation or generalization association. Coupling outside of the package is moderate, with the external associations being of the dependency from the GUI interface, aggregations to the Store and PaidAccount entities, and a composition to the Store entity. Although efficiency is sacrificed to maintain these associations, they are necessary as the Store instance dictates the specific StoreInventory and StorePromotion, and the Customer instance by way of the PaidAccount entity determines the unique CustomerHistory. An alternative would be to collapse attributes and methods from the StoreInventory and

StorePromotion into the Store entity, but then data structures of products would be stored in multiple packages, causing another dip in efficiency and reducing reusability.

- **Package name**: Store
- **Design goals for this package**: The primary design goal of this package is sufficiency.
- **Design tradeoffs for this package**: From the use cases (*Appendix A)*, a shopper through the *BestPurchase* app needs to be able to select a store from a list of stores within proximity to the shopper's location. The interaction between the interface and the methods of StoreDirectory to select the appropriate Store instance based off the getLocation() method satisfies this requirement within the Store package.

  Cohesion within the Store package is high, as all entities relate to a Store object or a collection of Store objects encapsulated within StoreDirectory. The package possesses moderate coupling as it does have two dependency associations from the GUI interface and two associations, one an aggregation and the other a composition, to the Inventory package. Like the Inventory package, these communications pose a tradeoff to efficiency in inter-package method calls. They are necessary as the interface must be able to see the StoreDirectory to select a Store, and the Store entity to determine the appropriate StoreInventory and StorePromotion.

- **Package name**: User
- **Design goals for this package**: The primary design goal of this package is sufficiency and flexibility.
- **Design tradeoffs for this package**: An important prerequisite identified in the original class model (*See Appendix B, Class Model Diagram*) is that a shopper desiring to use the *BestPurchase* app is required to set up some type of account, be it a trial or full-feature paid account. For aspects of the program to work as in use case #2 (*see Appendix A*), personal information about the shopper needs to be entered by the shopper regarding their name, contact, address, payment option, etc. The app then uses this information to auto-fill default order details. The elements of the User package satisfy these requirements. Additionally, entities within the User class are built in a modular fashion which separates Customer from Account and its subclasses FreeAccount and PaidAccount. This practice enhances flexibility in that elements of individual entities can be altered in the future given new functional requirements without substantially affecting the surrounding entities (Dennis, 2021).

  Cohesion within the User package is high as all entities highly relate to each other in processing information about the shopper and the shopper's account. A potential tradeoff to the package's flexibility is efficiency, since there is an aggregation association between Account and Customer which requires method calls to pass information. A possible solution

could be to collapse the Account entity into the Customer entity to eliminate the association and improve efficiency, however flexibility would be reduced as any part of the app using the Customer entity could possibly need alteration with future updates. Coupling is very low for the User package, with only an aggregation association from outside of the package to the PaidAccount entity.

Logical Data Flow Diagram

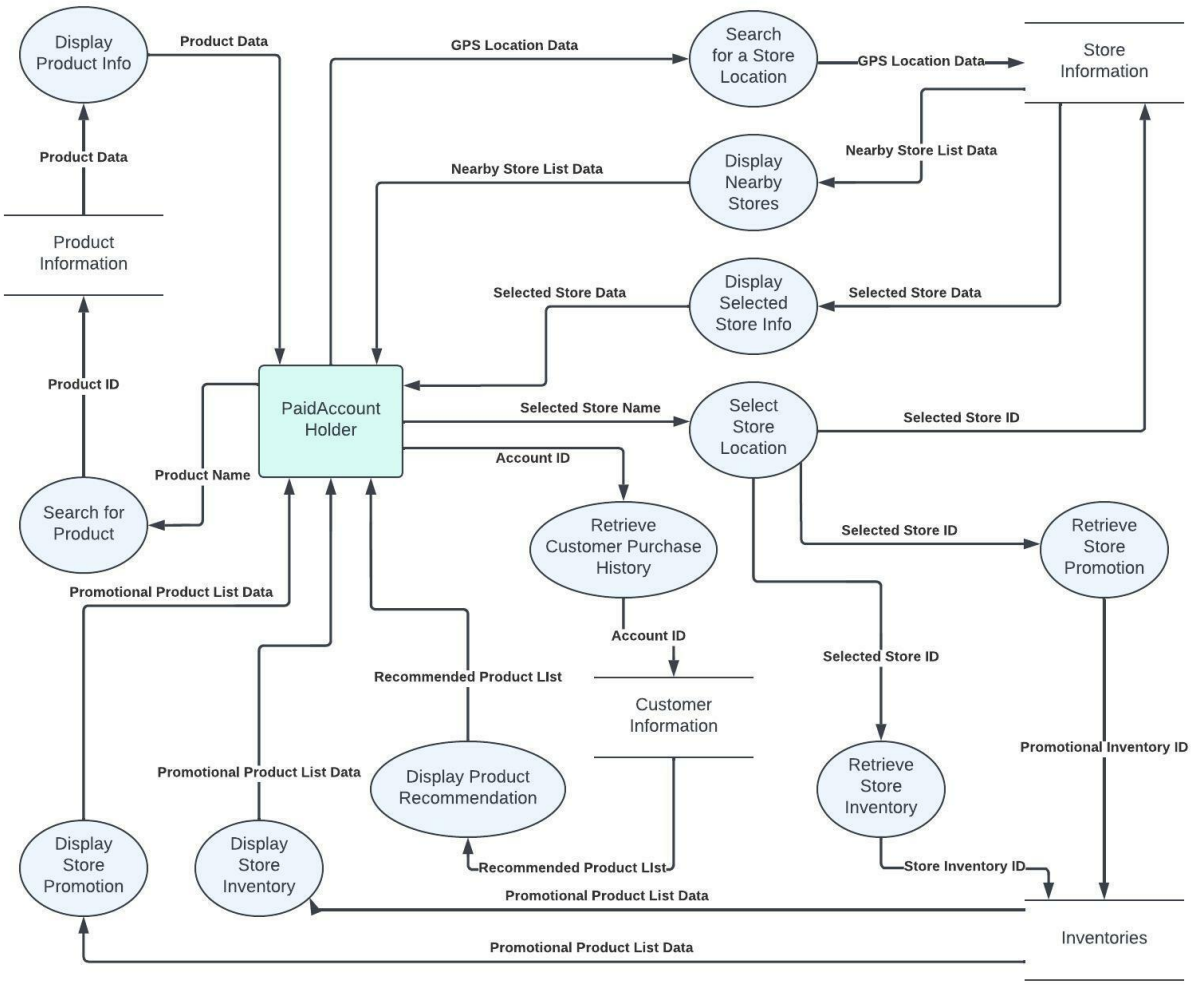

Figure 12 – *BestPurchase* Logical Data Flow Diagram

The logical data flow diagram for the *BestPurchase* app displays what type of information flows from the user interface through each aspect of the system as it performs the actions laid out from the functional requirements and use cases. The preconditions of the diagram are as follows:

- The information all stems from the external entity PaidAccount Holder, which represents a type of user who can fully manipulate the program. Although a free account is identified in the class model bound by the limits of a trial period, its

functionality is a subset of the paid account, offering no unique understanding of the data flow. It is omitted from this diagram.

- The range of functions available to the PaidAccount Holder and identified as processes in the logical data flow diagram map to the limits of use case #1 as described in Appendix A. Therefore, data flows prior to the user state of being a paying registered user of the app and fully logged in are omitted. Also left out are data flows representing activities possibly occurring after the shopper has searched for and found the desired product information.

Starting at the 12:00 position moving clockwise in Figure 2, data is transmitted first from the user interface relaying GPS location data to a process that searches for nearby store locations. The data is compared against a repository of store information to find the nearby results. The results are sent back to the shopper through a display process. Afterward, the shopper selects the desired store, and the interface invokes a process that sends the selected store data to the store information repository again and returns the selected store info through a display process. Concurrently, the store selection process invokes two other processes to retrieve the store promotions and store inventory from a product inventory repository and sends them to the interface through parallel display processes. Immediately afterward, the interface sends the account ID of the shopper through a retrieval process that requests the customer purchase history. That information is sent back through a display process to the interface. In real time, the *BestPurchase* app now shows the product information to the shopper represented by the GUI sketch in Appendix C. Lastly, the shopper types a product name into the interface for a direct search, which is sent to a searching process that accesses the product information repository. The product information is sent back to the interface using a display process, completing the use case.

Physical Data Flow Diagram

The physical data flow diagram maps the path of data flow between the shopper, the physical elements that make up the mobile device, and the 3-tier architecture used by the *BestPurchase* app. The scope envelops the mobile device since many peripheral features of modern devices are used for security and interface, such as two-factor authentication, touch screen input, location services, etc. Functionality and data flow of the *BestPurchase* app should complement these features. The app's non-functional requirements come into play here as understanding device capabilities is key to knowing what functionality to build into the app. (*See Appendix C for an example of the GUI interface and the beginnings of non-functional requirement development*)

Like the local data flow diagram, the source of data flow begins with the PaidAccount holder. The shopper first authenticates with the mobile device for secure access to the device's features. The success or failure of this action is sent to the device display in the form of a graphical confirmation on the screen. Once successful, the shopper manipulates the *BestPurchase* app through the mobile device touch screen, which feeds information into the

app and returns real-time processing of the input through the mobile device display. Login, navigation, and text data are passed to the app to manipulate the menu and options. To ensure that the app can check for nearby store locations, the app requests the mobile device's GPS location through the device location services.

From the local app installation on the mobile device, various request for information travel back and forth between the app GUI and the application server. The server in turn requests information about the customer, stores, and product info from the various aspects of the app database.

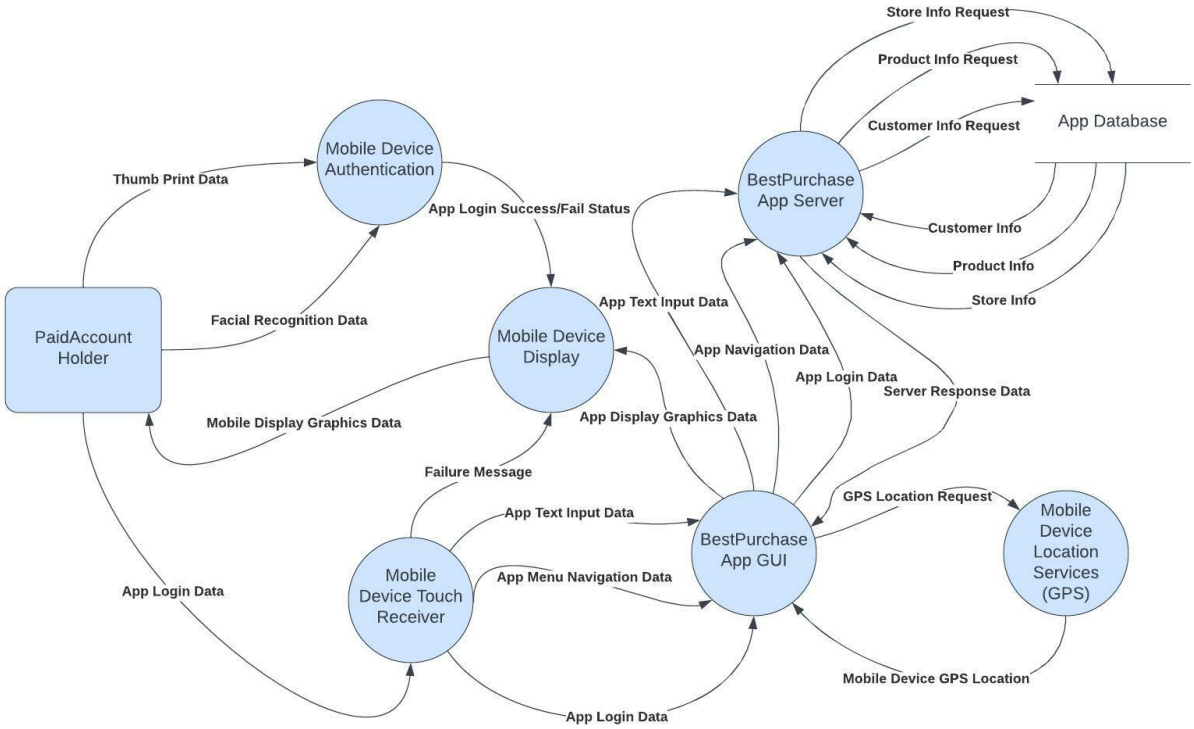

Figure 13 – *BestPurchase* Physical Data Flow Diagram

Appendix E – Conceptual Inspiration

The conceptual inspiration of the *BestPurchase* application is from Giant Food All-in-one app.

Giant offers a free mobile app that allows users to create an account, add a store loyalty card and payment option, and add stock-checked store items into a virtual cart for checkout (Wadland, 2020). The app works with all store locations and checks against the local amenities offered at each. The app offers hybrid functionality, allowing the shopper to order fully online or by using a ScanIt reader to scan items physically by barcode and fill a physical cart. The reader then docks to a self-checkout lane within the store and interacts with the user to process payment (Roche, 2019).

The Giant app uses assets acquired from its Peapod service to schedule delivery of completed orders or allow for parking lot pickup or drive-through at select locations. Where offered, in-store deli, pharmacy, food stalls and specialty services are integrated into the app (Wadland, 2020).

Credit: Giant Food https://giantfoodstores.com (Giant Food, 2022)

Taken from *Requirement Analysis and Operational Characteristics for the* BestPurchase *App,* MET CS 682 Assignment 3, Term Project Part 1 by Edward T. Myers, 9/28/2022.

References

Bell, L. (2017, 6 5). *Encryption explained: how apps and sites keep your private data safe (and why that's important)*. Retrieved 9 24, 2022, from Wired: https://www.wired.co.uk/article/encryption-software-app-private-data-safe

David, M. (2019, 6 28). *Compare the top mobile operating systems for developers*. Retrieved 9 24, 2022, from Tech Target: https://www.techtarget.com/searchsoftwarequality/feature/How-to-differentiate-leading-mobile-operating-systems

Dennis, A. W. (2021). *Systems Analysis and Design: An Object-Oriented Approach with UML* (6th ed.). Hoboken, NJ: Wiley.

Giant Food. (2022, 2 24). *The Giant Company Enhances Online Grocery Offerings for Giant Direct and Martin's Direct Customers*. Retrieved 9 23, 2022, from Giant Food: https://giantfoodstores.com/pages/the-giant-company-enhances-online-grocery-offerings

Krajcovic, J. (2019, 6 18). Free iPhone XS Wireframe Template. *Unblast*. Retrieved from https://unblast.com/free-iphone-xs-wireframe-template/

Muchmore, M. (2020, 10 2). *Android vs. iOS: Which Mobile OS Is Best?* Retrieved 9 24, 2022, from PC Magazine: https://www.pcmag.com/comparisons/android-vs-ios-which-mobile-os-is-best

Roche, B. (2019, 11 7). *Giant Food Stores tests app that allows shoppers to skip checkout line*. Retrieved 9 23, 2022, from WGAL News 8: https://www.wgal.com/article/giant-food-stores-tests-app-that-allows-shoppers-to-skip-checkout-line-scanit-express/29727730

Wadland, M. (2020, 6 29). *Giant Food Launches New All-in-One App*. Retrieved 9 23, 2022, from Zebra: https://thezebra.org/2020/06/29/giant-food-launches-new-all-in-one-app/

Evaluation

	D	C-	C+	B-	B+	A	Letter Grade	%
Clarity	Disorganized or hard-to-understand		Satisfactory but some parts of the submission are disorganized or hard to understand	Generally organized and clear	Very clear, organized and persuasive presentation of ideas and designs	Exceptionally clear, organized and persuasive presentation of ideas and designs		0.0
Technical Soundness	Little understanding of, or insight into material technically		Some understanding of material technically	Overall understanding of much material technically	Very good overall understanding of technical material, with some real depth	Excellent, deep understanding of technical material and its inter-relationships		0.0
Thoroughness & Coverage	Hardly covers any of the major relevant issues		Covers some of the major relevant issues	Reasonable coverage of the major relevant areas	Thorough coverage of almost all of the major relevant issues and researched where appropriate	Exceptionally thorough coverage of all major relevant issues and researched where appropriate		0.0
Relevance	Mostly unfocused	Focus is off topic or on insubstantial or secondary issues	Only some of the content is meaningful and on topic	Most or all of the content is reasonably meaningful and on-topic	All of the content is reasonably meaningful, and on-topic	All of the content is entirely relevant, and meaningful		0.0
						Assignment Grade:		0.0

The resulting grade is the average of these, using A+=97, A=95, A-=90, B+=87, B=85, B-=80 etc.
To obtain an A grade for the course, your weighted average should be >93. A-:>=90. B+:>=87. B:>83. B-:>=80 etc.