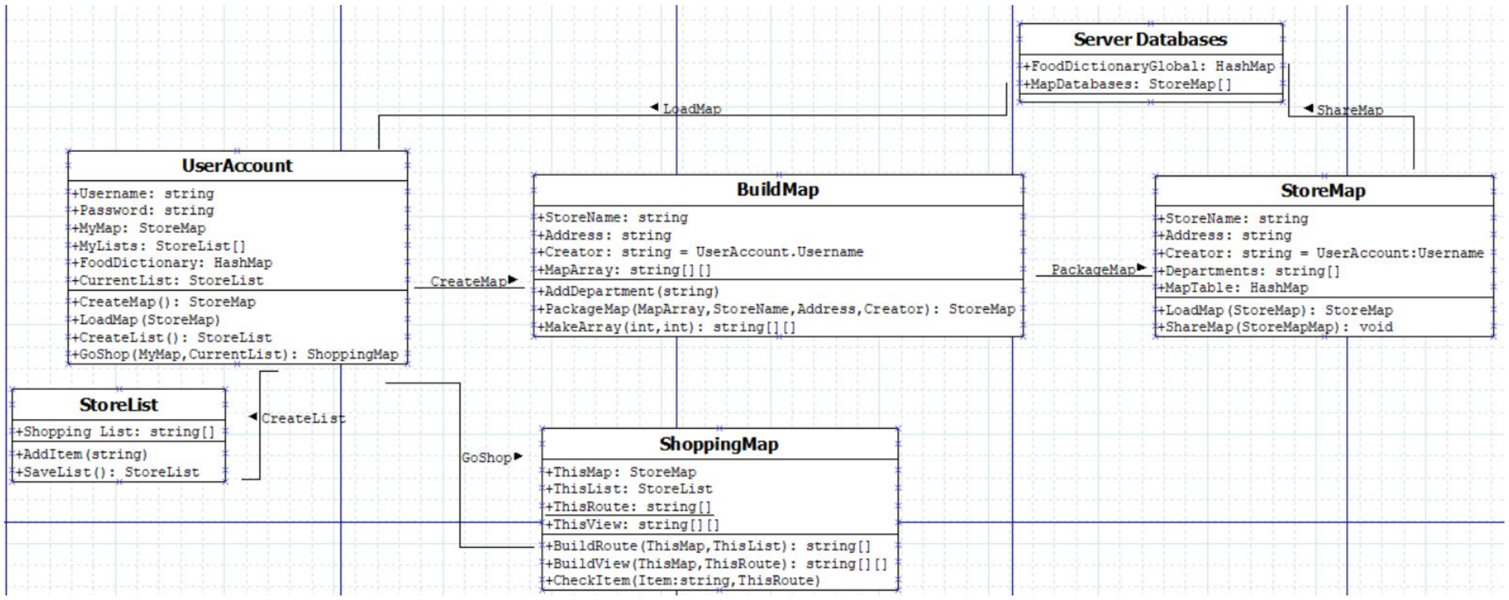
Grocery Shopping App Requirements

(Grocery Guide)

11/10/2019

Christopher Brown
Dane Emmerson
Evan Horne
Anthony Huynh
Briana Willems

UML Class Diagram



Packaging the implementations

Coupling

Content Coupling

- If the user builds a map, saves it, and then continues to make more changes to the map, this will require build map to edit the store map class.
- The user class will need to be able to modify the store list entity in order to build shopping lists and edit shopping lists.

Common Coupling

- User account and store map entities will need to read and write from the map class stored in the database

Control Coupling

- Shopping map will need to call build map which will then need to return the store map to the shopping map
- Shopping map will need to call store list in order to retrieve shopping lists and build shopping maps

Stamp Coupling

- Store map is going to need to provide the structured store map back to shopping map class so shopping map can build the appropriate route

Data Coupling

- Store list will need to provide an unstructured list to shopping map so shopping map can sort and build a route based off the list
- User account will need to provide credentials to the authentication system on the server

Cohesion

Functional/Informational Cohesion

- User Account and Shopping List work together for one purpose, creating a shopping list.
- Authenticator is used to add security
- GUI will help the user interact with the software
- Tracking list items and location in store

Communicational Cohesion

- User Account and Server Databases, both have information about food
- Map view and list view both use the same data (the routing information)

Procedural Cohesion

- User Account Creates a Map, which leads to Build Map Packaging the map. The Store Map then Shares the Map.

Logical Cohesion

- Only one version of the routing algorithm will be ran within shopping map.

Iterative vs Incremental Development

Development of our app would benefit greatly from the Agile Iterative Design model, because a cyclical prototyping, testing, and analyzing process fits our model's requirement needs better. Iterative development on this project allows us to have singular focus areas on specific features. Because these features are cyclically created and implemented, it's typical for a framework of code to naturally develop, enabling code reuse. Aside from natural code reuse, implementing features using the Iterative Design model's focused, cyclical system allows us to directly approach each and every requirement like a checklist.

The nature of an incremental model requires implementing the system in terms of a more broad scope, and thus would make building a framework for code reuse a slower, more intentional process. Because the scope of this project is fairly narrow and straightforward, incremental design would not add as much benefit as iterative design.

An additional key benefit of using the iterative design model on this project is that we would be able to work more closely with our customer during the software development life cycle. U Upon receiving feedback from the customer, we could quickly and efficiently make changes and improvements to the application at the customer's request. This will ensure we create a quality end product that fulfills the customer's expectations.

Evaluation of Design Patterns

Memento

The memento design pattern will be particularly useful in building Grocery Guide because we need to save the state of the user while using the application and allow the user to come back to this breakpoint at a later time. Building and organizing store maps could be a potentially time consuming process. It is important that we save the user's state throughout the map building process to allow the user to come back to a certain point if they want to take a break, they exit the app, or their system crashes, etc. Implementing the memento design pattern will also be helpful in allowing for "undo" functionally in the map building process. Additionally, while a user is shopping with Grocery Guide, designing with memento will allow us to let the user easily recover and come back to a certain place in their shopping route despite external events, such as the user's phone going to sleep.

Visitor

The visitor design pattern will be useful in the implementation of our chosen architecture. By implementing the visitor design pattern, this will allow us to export the grocery route algorithms to the client and store the majority of the map and shopping list data in the database on the server. Map information and shopping lists can then be sent to the client as requested and manipulated via the algorithms on the client before displaying information to the user. This design will help ensure a smooth user experience despite limited network connection.

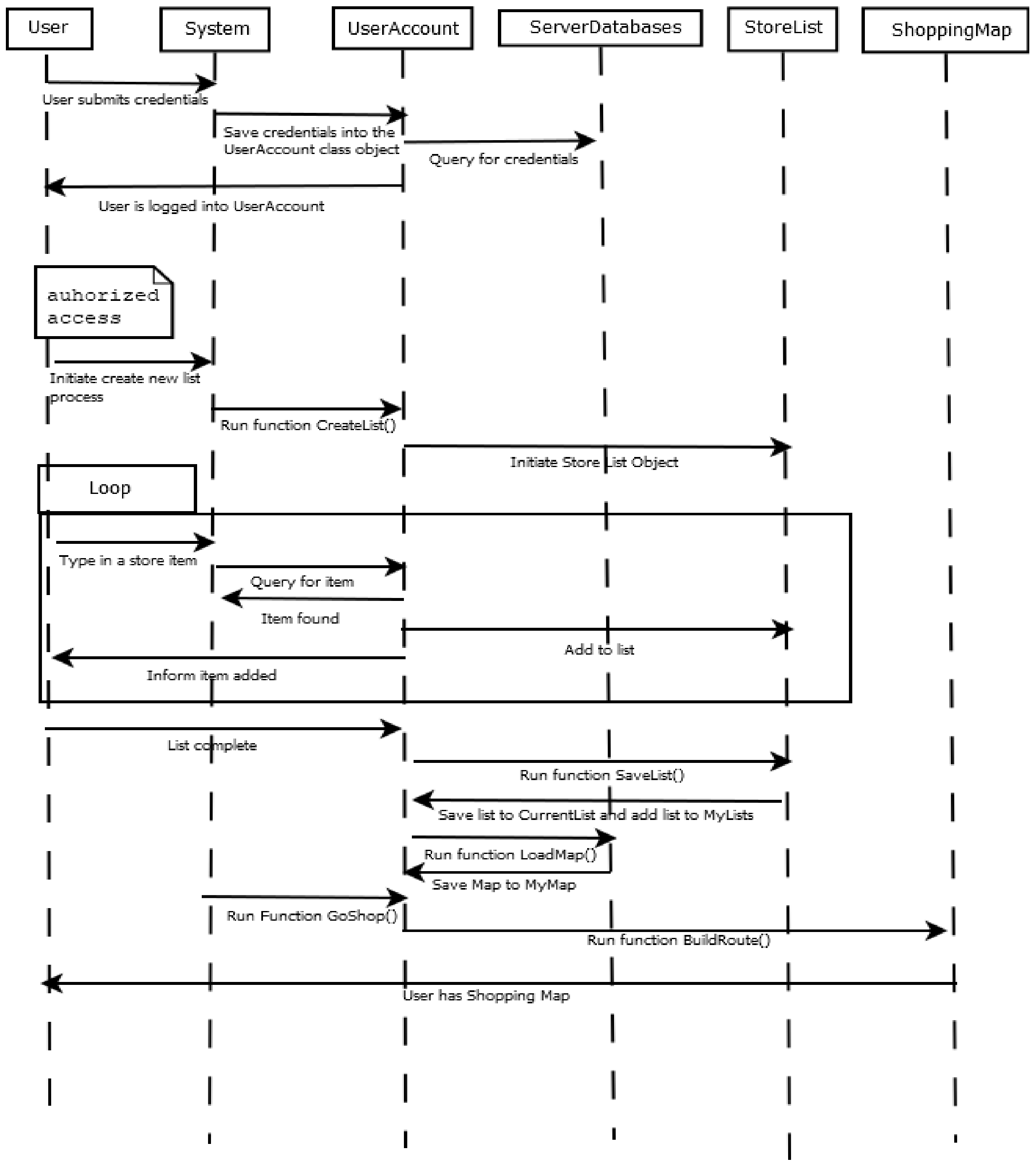
Composite

Due to the inherent complexity in building a map, the composite design pattern will be helpful in breaking down and reducing the complexity of the build store maps process. The composite design pattern will allow us to break down different sections of the store into various types of objects, but handle the various types of objects as a single object when actually building the store map to show the user. This will make adding/changing specific parts of the store a simpler process, while simplifying development of the program and improving maintainability. In addition to the maintainability that this design pattern provides, it will also make expanding this application to other types of stores (other than grocery stores, home improvement stores for example) a feasible task in the future, if we ever decided to embark on this endeavor.

Strategy

Strategy is important in the development of our grocery application as it will help us decide which routing algorithm to use at runtime, based off user input. Because the user input is not known until runtime, and the user will be able to edit/customize the grocery guide algorithm (for example, the user may specify that anything over 5 pounds constitutes a heavy item). We must design the program in a way that the program itself can choose the correct algorithm to use at runtime with various user customized inputs also specified at runtime. Furthermore, strategy will be helpful in allowing for even more specialized routing functionality, such the user choosing to pick up cold items just before heavy items last.

Sequence diagram



Interfaces

Our key interfaces include creating an optimal route store route and a number of REST requests:

- Creating optimal store route:
    - Preconditions: User is either logged in to their account which already has a store selected and a route created, or user is using application as guest and has selected grocery store and created a shopping list
    - Input: shopping list, store map, preferences for route
    - Output: organized shopping list based on store map and route preference, map or list view of store showing route and user's current location in store
- REST requests:
    - Preconditions: User has downloaded app and is currently using app
    - Client Input: User login information if not already logged in, otherwise encrypted token (similar to Javascript Web Tokens), retrieving list and store data
    - Client Output: Successful message from server, list of items and map of store, as well as an encrypted token from server for next request
- Inter-package interfaces:
    - Front End:
        - Input: information from authenticator, store creator, list creator, and marking items off a shopping list will be sent to the application GUI
        - Output: the application GUI
    - Authenticator:
        - Input: username, password
        - Output: a REST request to the server performing the authentication
    - Store creator:
        - Input: dimensions of store, size and location of departments
        - Output: map of store to scale
    - List creator:
        - Input: name of list and store, item names
        - Output: name of list and store, items on list (else inform client if store does not carry this item)
    - Marking items off list:
        - Input: items that have been picked up
        - Output: removing item from list and updating next department as necessary
    - Map route builder:
        - Input: map of store to scale from store creator, list
        - Output: map of store with potential routes

Exceptions

Network Errors:

This will likely be our primary source of exceptions due to the nature of internet-connected applications. Examples of network error exceptions would be incomplete downloading of store maps, incomplete/inaccessible product database searches, and unsaved/corrupted grocery lists. Methods for handling these exceptions will likely involve the client side doing an “end-of-file” check when pulling data from servers. This can be accomplished by ensuring that our store map data and grocery list data files have custom headers specifying their size. If the client device knows the size of the file to be downloaded, then it will be able to check if a file download has been interrupted based solely on size comparison. In regards to incomplete product database searches, since these are always going to be just hitting an endpoint to make a query, a maximum timeout can be set on the client device to signal that the connection has been lost.

Invalid/Insecure Input:

Invalid input is another common, sometimes terrifyingly powerful, exception. Invalid input exceptions occur when data fields in our database, client app, or application files have incorrect or corrupted datatypes in them. Invalid input is the basis of all Cross-Site Scripting (XSS) attacks. Since our clients will be registering by typing their own usernames, queuing our databases, and creating store maps that will be uploaded to our databases, sanitization of all data fields is extremely important. We need to ensure that all forms of input have specific data type fields, and handle sanitization gracefully. If a user inputs anything that does not pass a strict sanitization check, an error text should pop up below the input field describing what parts of the input did not pass sanitization. A user should NEVER be allowed to submit any data to our database that does not pass sanitization, as doing so could open our database up to SQL injection, our server to reflected cross site scripting, and our users to malicious code execution.

Contributions:

UML diagrams: Chris Brown
Packaging the implementations: Group
Incremental/iterative development: Evan Horne
Design patterns: Dane Emmerson
Sequence Diagram: Briana Willems
Interfaces: Anthony Huynh
Exceptions: Evan Horne