Spoiled Milk Design Documentation

Team Information

- Team name: Spoiled Milk
- Team members
  - Yaro Khalitov
  - Jeremy Smart
  - Daniel Jara
  - Nate Saunders
  - Jerry Chen

Executive Summary

> We will make a functional e-store with different features based on whether the user is an admin or a customer.

Purpose

> This project will allow users to access a spoiled milk e-store. An admin will be able to update products in the inventory, and the customer will be able to view and add spoiled milks in their shopping cart

Glossary and Acronyms

> | Term | Definition |

> | SPA | Single Page |

> | MVP | Minimum Viable Product |

> | UI | User Interface |

> | MVVM | Model-View-ViewModel |

> | HTML | HyperText Markup Language |

> | CSS | Cascading Style Sheets |

Requirements

> - A user will have access to a login page where they can log in either as a customer or an admin
> - A customer will have access to a shopping cart where they can add or remove products
> - An admin will have access to an inventory where they can add, update, or delete products

> - A customer will have access to a dashboard where they can view the spoiled milks and add it to their shopping cart.
> - A customer will have an option to purchase the products they add to their shopping cart.

Definition of MVP

> For Sprint 2, these are our definition of MVP:

> - Minimal authentication - an admin signs in with the reserved username admin, and any other username is seen as a customer
> - Customer Functionality - customers should be able to search for a product and add/remove products from their shopping cart
> - E-store Owner Functionality - admins should be able to add, remove, and update products in the inventory

MVP Features

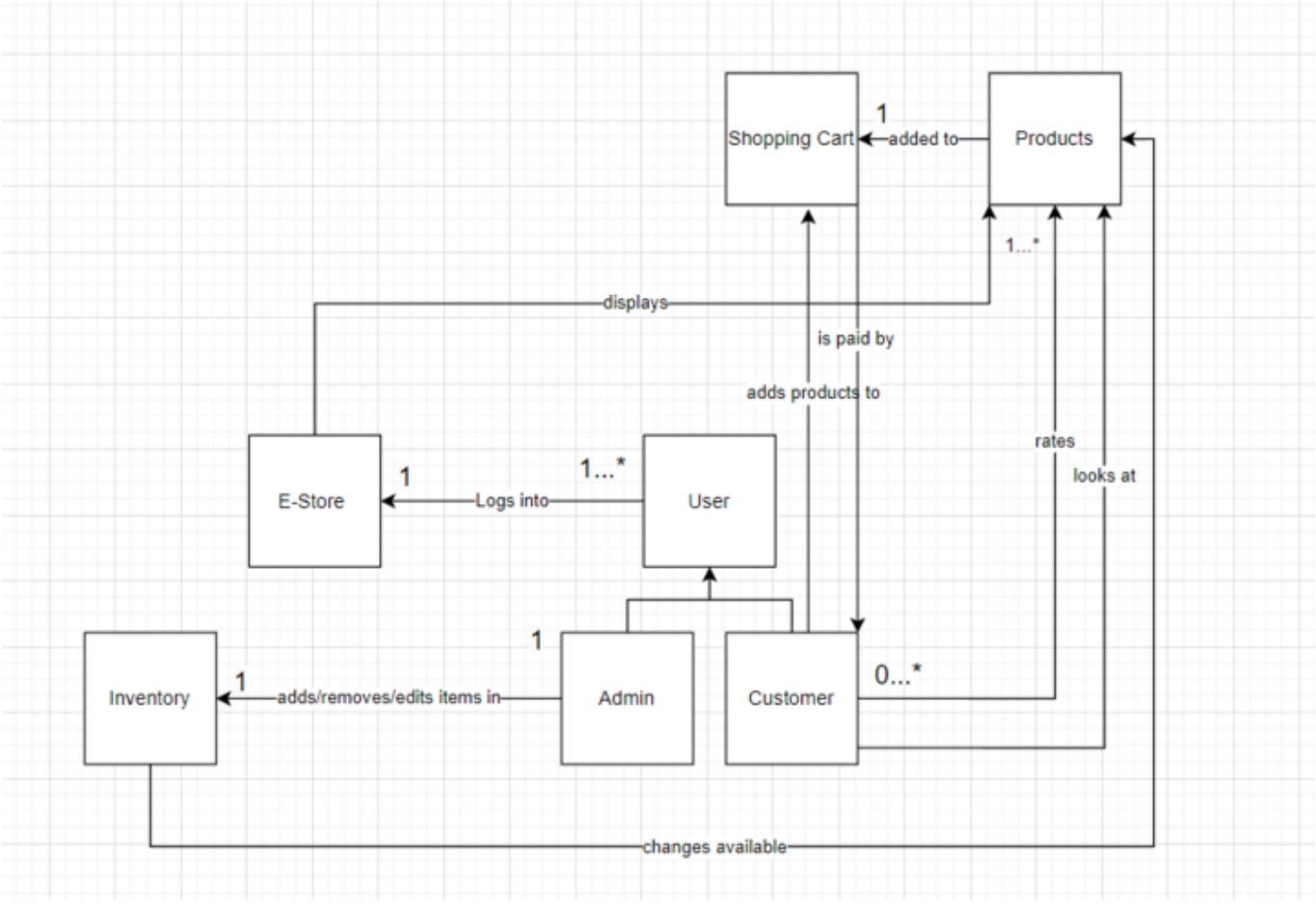
> Will be implemented in a later Sprint

Enhancements

> Will be implemented in a later Sprint. We plan on doing a password system as well as a rating system

Application Domain

This section describes the application domain.

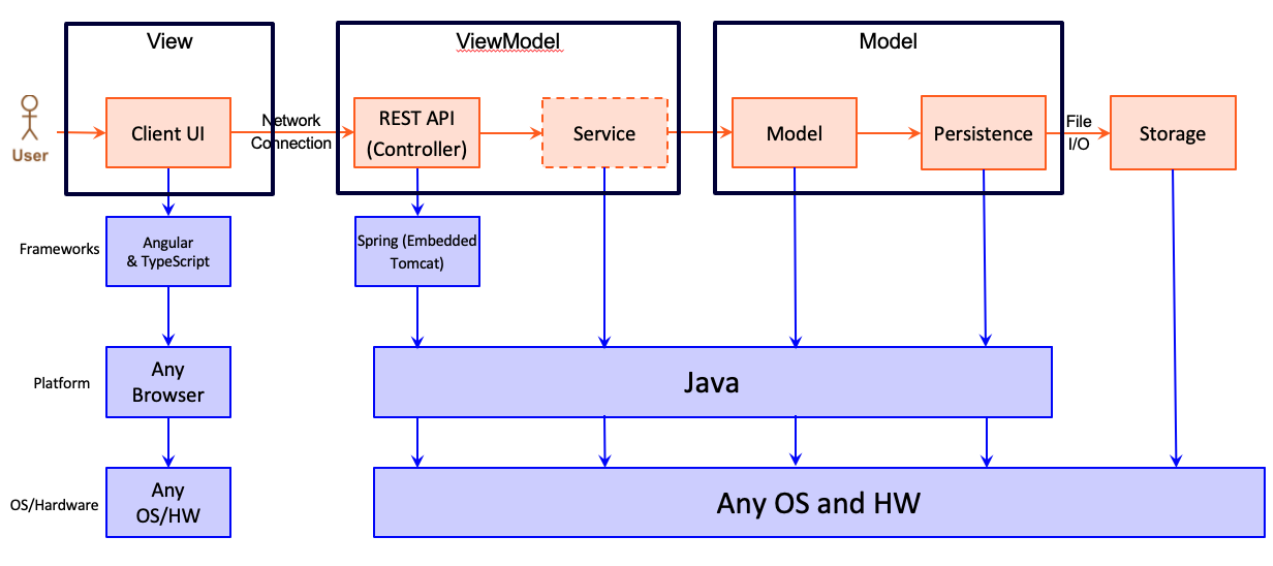


Architecture and Design

> This section describes the application architecture.

Summary

> The following Tiers/Layers model shows a high-level view of the webapp's architecture.



> The e-store web application, is built using the Model–View–ViewModel (MVVM) architecture pattern. The Model stores the application data objects including any functionality to provide persistance. The View is the client-side SPA built with Angular utilizing HTML, CSS and TypeScript. The ViewModel provides RESTful APIs to the client (View) as well as any logic required to manipulate the data objects from the Model. Both the ViewModel and Model are built using Java and Spring Framework. Details of the components within these tiers are supplied below.

Overview of User Interface

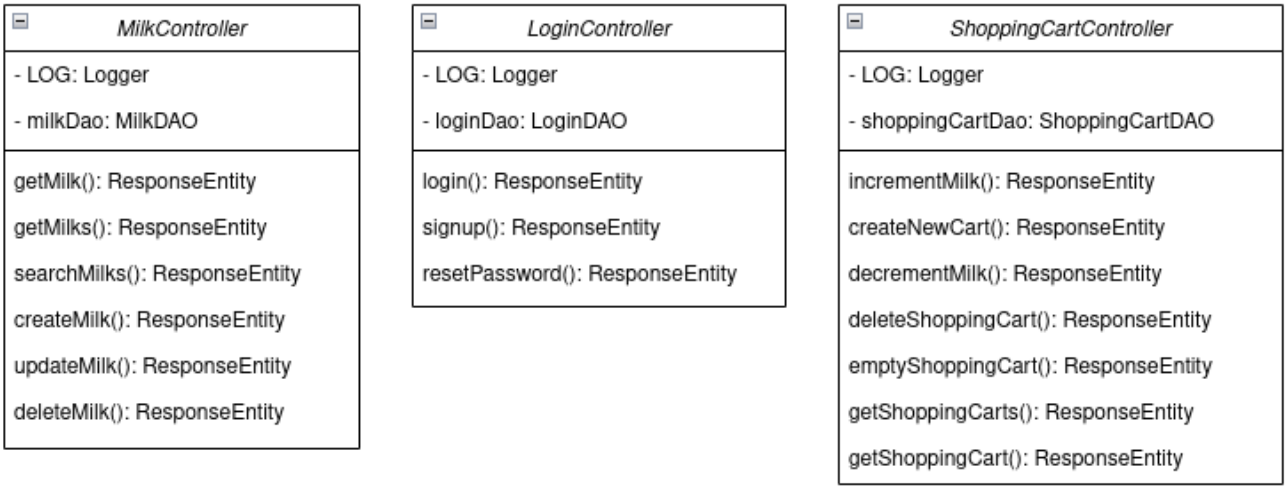
> This section describes the web interface flow; this is how the user views and interacts with the e-store application. *Provide a summary of the application's user interface. Describe, from the user's perspective, the flow of the pages in the web application.*

View Tier

> Will be implemented in a later Sprint

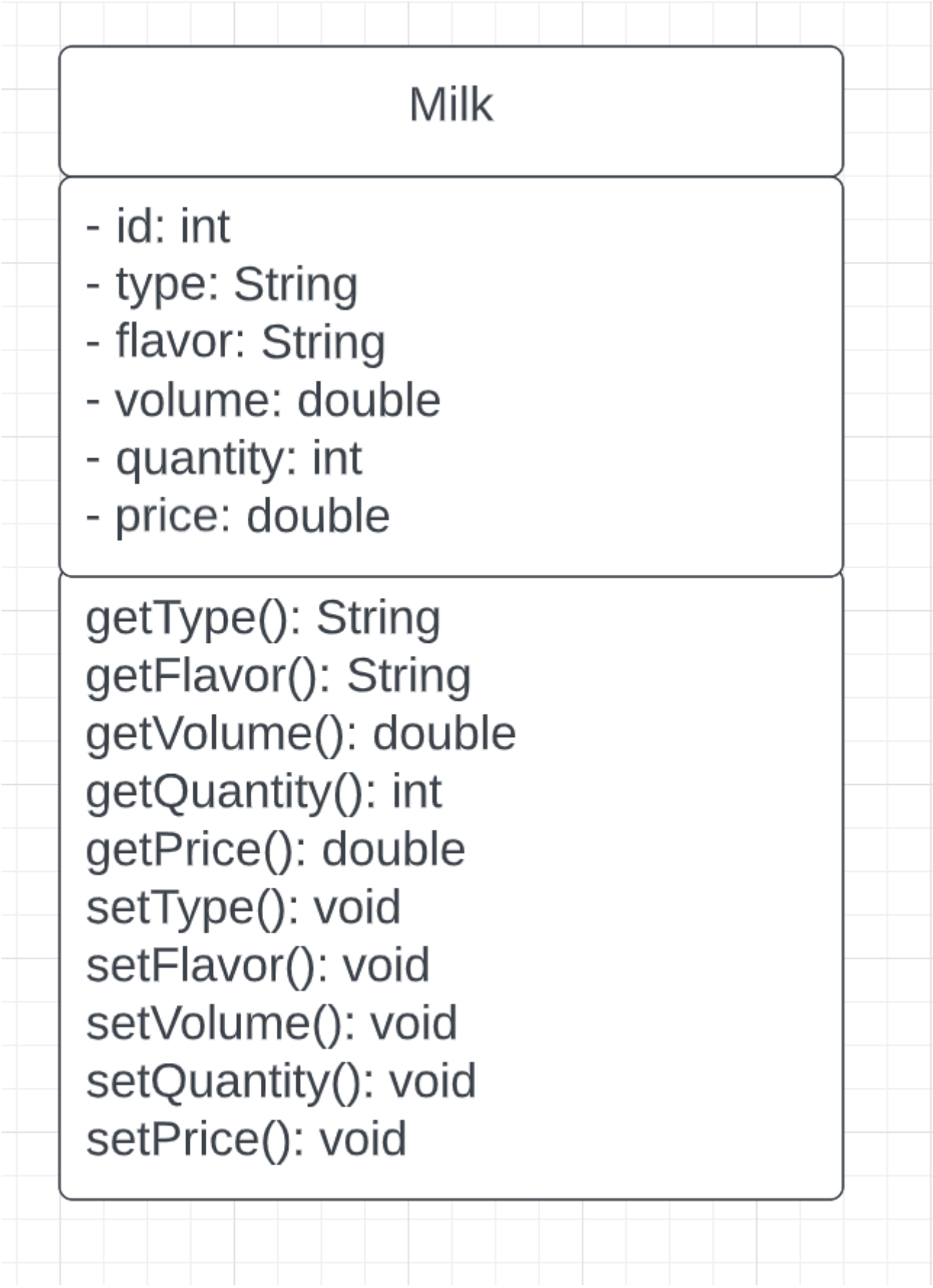
ViewModel Tier

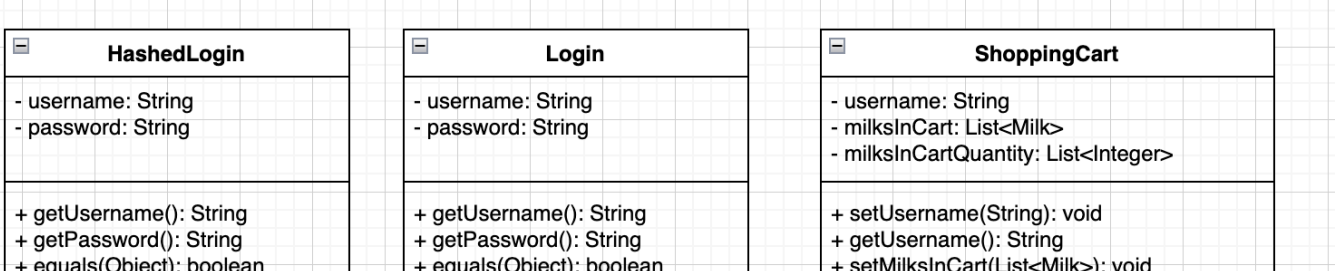
> We have a Controller class for Login, Milk, and Shopping Cart routes. Their UML diagrams are shown below.

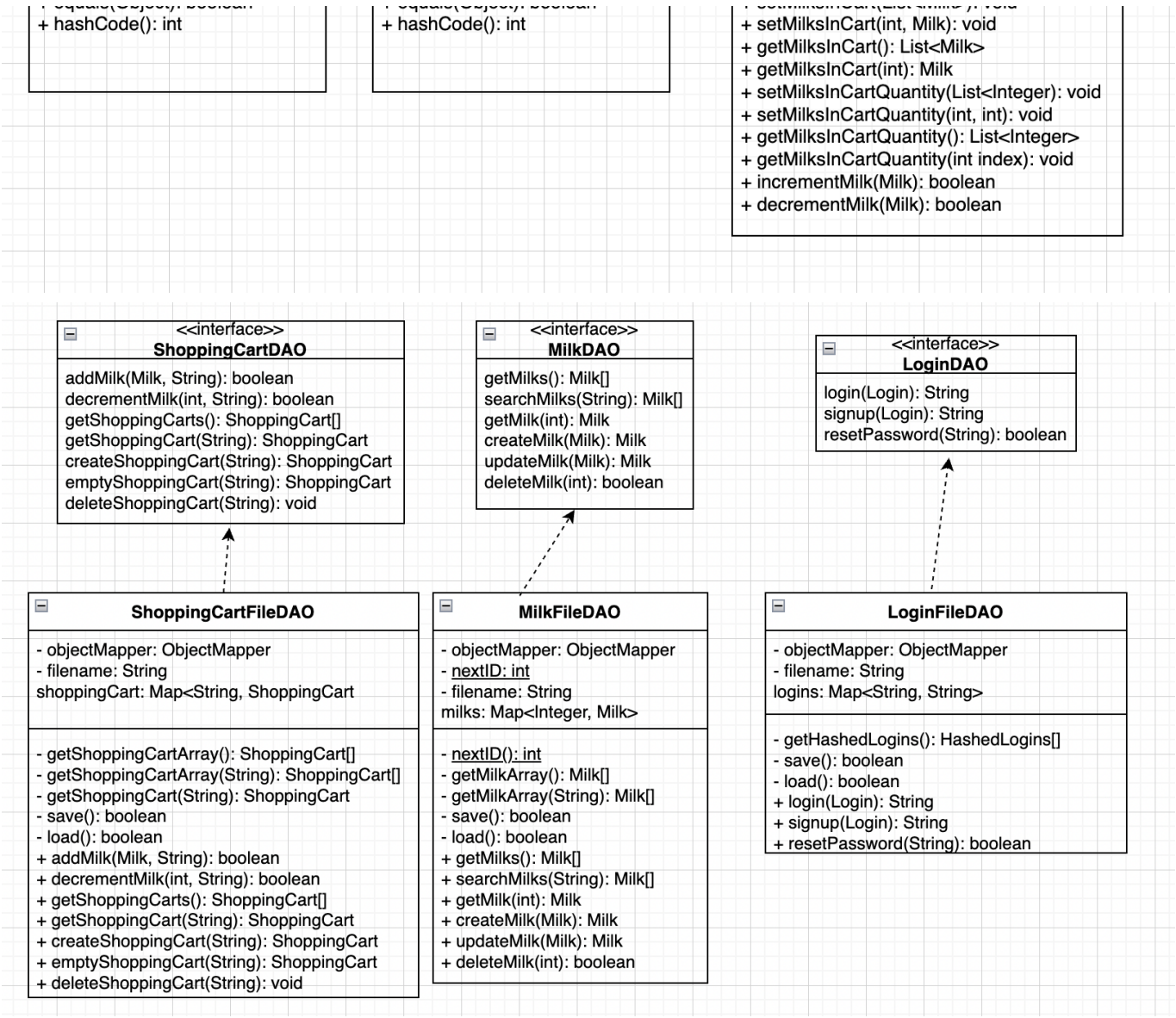


Model Tier

> We have a model for login, shopping cart, and the milk object itself. The UML diagrams is shown below.


Milk
- id: int
- type: String
- flavor: String
- volume: double
- quantity: int
- price: double
getType(): String
getFlavor(): String
getVolume(): double
getQuantity(): int
getPrice(): double
setType(): void
setFlavor(): void
setVolume(): void
setQuantity(): void
setPrice(): void

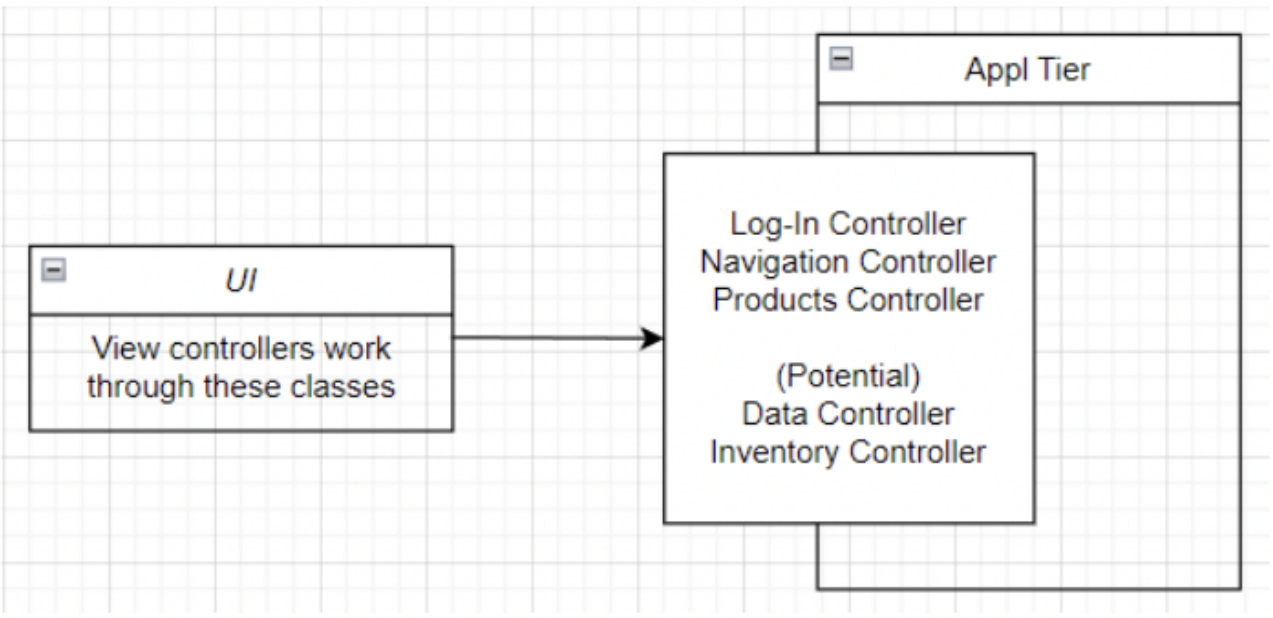

HashedLogin
- username: String
- password: String
+ getUsername(): String
+ getPassword(): String
Login
- username: String
- password: String
+ getUsername(): String
+ getPassword(): String
ShoppingCart
- username: String
- milksInCart: List<Milk>
- milksInCartQuantity: List<Integer>
+ setUsername(String): void
+ getUsername(): String



OO Design Principles

Controller

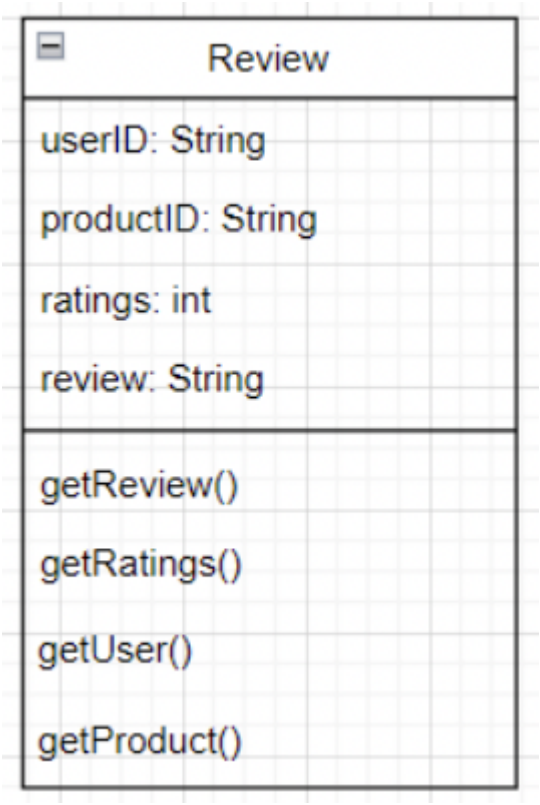
> When it comes to our estore, we should be using more than just one controller. Several functions don't relate to each other well enough to be contained within just one, so our current design will be implementing several. One section of our store that should use its own controller is the log-in page. The UI controller would communicate with a log-in controller to perform tasks, such as creating new accounts, checking to see if a password matches with the user, and checking if the user in question has admin access. Another controller could be used for the navigation of the store itself. Taking commands from the UI, a controller for the main page would control which pages are being shown. The last main controller that should be on our list is one for the list of products. Being mostly contained in a single page, there is a lot of functionality on the list of those products, including the ability to, potentially, interact with the main page controller to change pages to the selected product. This is our current design, and I believe there could be some improvements to be made. For one, there was not too much involvement in terms of working with data storage. It is unclear how we intend on going about this, but I was thinking a controller for data submission and retrieval would be a good way to go about it. For example: when a user adds a review on a particular product, the UI controller should be able to call the data controller to add that review to the data storage system,

> most likely to not be a database. This same controller could also be used to create new users. It could take the data to be added as well as the destination file to add to so that it can be used frequently. Additionally, there could also be a controller that can only be accessed by admins used to manage the inventory of products. This could potentially work with the controller for manipulating data, as it would update the data of the inventory as changes are made. Below is an image of the different controllers that could interact with the UI.
>
> 
>

Single Responsibility

With single responsibility, many classes will be made to perform specific tasks. One quickly distinguishable class is the Product class. This class specifically holds the data for products, including the flavor, size, and age of the product. As also stated earlier, specific classes for controlling the site can also be used. Site navigation can be limited to at least one class, and the same can be said for logging in and product lists. Also, however, functions specific to account information can be held within an Account class. After properly logging in, an Account object would be accessed, and information from it will be used by it. The user should be able to edit passwords and other information, while also having a ShoppingCart object attributed to it. The ShoppingCart object would manage the functions of a shopping cart and hold Product objects. ShoppingCart objects should be able to add and remove products from its array of products, sending the array to the checkout page once the user is ready to purchase. Another thing that we can do to better show this practice is to have more controllers for specific functions. Another addition could be a checkout controller that manages the process of checking out. Alternatively, however, it would also be possible to split up something larger into different responsibilities, each one with a class to handle it. An example of this in action would be splitting up the data controller into their respective data types. For example, there would be one controller to handle the saving of accounts in a text file while another works on saving Review objects in a review text file. The usage of a Review object would also follow this practice, as instead of using the data controller to read and write from a review text file, a Review object would hold the attributes of the ratings, the text review itself, the id account who made the review, and the id of the product the review was left on. All of this can be organized into a single class, which would give it the sole responsibility of being

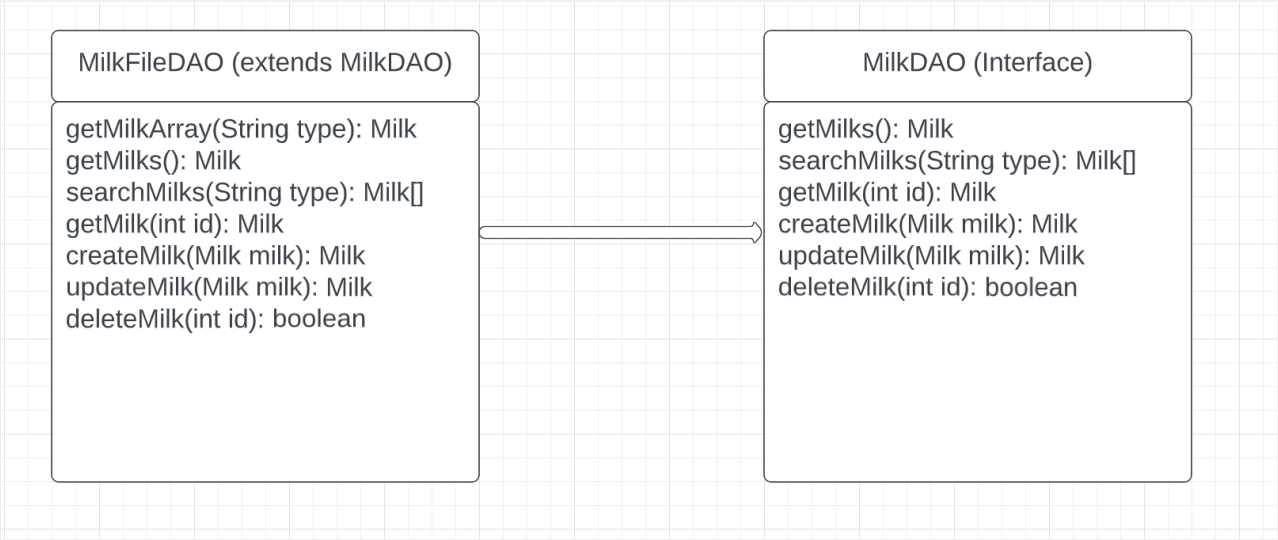
able to be accessed, read, and edited. Below is a rough example of what a review object would look like.



Open/Closed

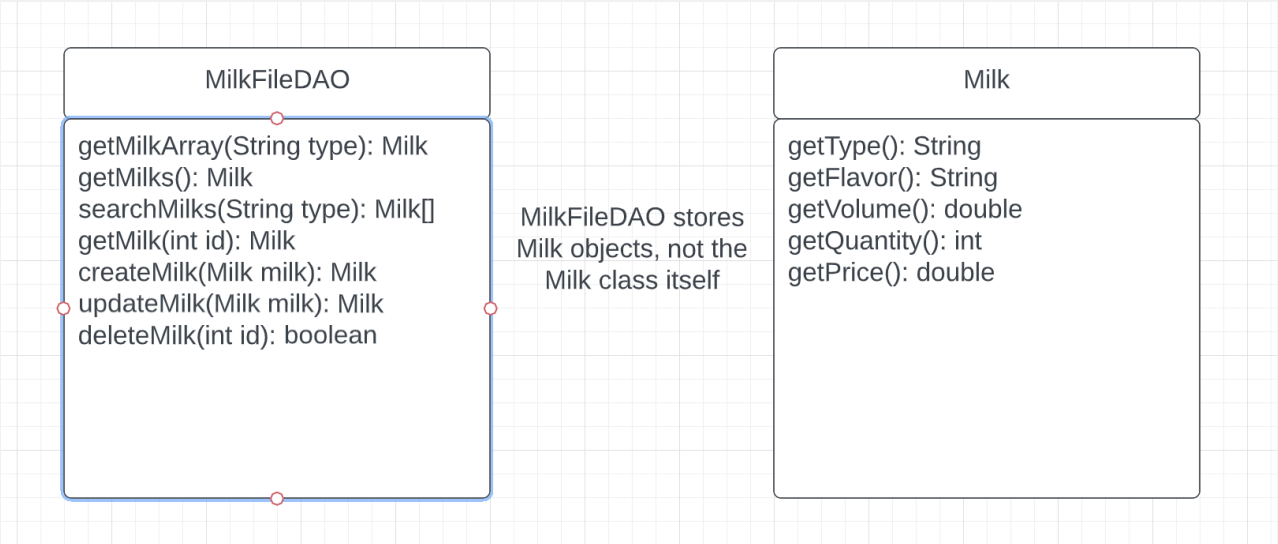
> The principle of open/closed states that software entities should be open for extension but closed for modification. A good example of this is inheritance, interfaces, and polymorphism. A class should only be designed for one purpose (as stated by single responsibility) and should be polished to do that one thing. Adding or changing things to it would not make it closed for modification. In our spoiled milk e-store project we are designing it in such a way that we do not create redundant code and instead make use of OOP and OOD principles. For example, we will have a general product class that will define base things like price, quantity, and name. From that ,we can create children classes that will build upon that adding other special product qualities. In this way, we do not modify the base class but instead, open it up for extension. The extension of making multiple milk objects is possible without modifying the base class. If possible, making use of polymorphism will also be great to adhere to this principle. Perhaps we as a team should consider creating an abstract class that defines the final prices for each product. This way in a different class we can call upon the price class without modifying anything else in that different class. Adding these little changes will help our

project be better and adhere to this principle.



Pure Fabrication

Pure fabrication principle states that in order to support single responsibility and low coupling, a non-domain entity should be created when necessary. This principle focuses on creating artificial entities (when necessary) that are not necessarily part of the domain object but instead do some of the work that the domain objects should not. This will be vital when storing inventory data of our products. We do not want the product classes to also perform storage of the data. This should be done by a separate class where we can store our milk objects in a file or something similar. This helps enable our entities to stay within their single responsibility domains, and encourages low coupling. I think for our current design we should definitely consider adding more stories on trello for entities such as these. The inventory entity is one example, but we should also consider doing it for the login page too. It's easy to sometimes start implementing stuff without thinking ahead, so we want to discourage that when working on our project.



Static Code Analysis/Future Design Improvements

One issue that occurred in our milk-app project which tested our TypeScript code was in the file: src/app/milk-detail-customer/milk-detail-customer.component.ts. In our rateMilk() function, it flagged three errors which said, ‘Unexpected var, use let or const instead.’ Our analysis reports that

> there is a significant distinction between the variable types created by var and by let. A switch to let would help alleviate many of the variable scope issues we've seen in our project. An easy way to fix this is simply to switch out the instances where we used var with const, similarly to how we used const in our getMilk() function just above the rate function. Another issue within our milk-app was in the file: src/.../java/com/estore/api/estoreapi/persistence/LoginFileDAOTest.jav. When testing our login and signup components, it flagged twice that we should have combined the three login tests into a single parameterized one, as well as the three signup tests. Since our multiple tests differ only by a few hardcode values, if given more time we should have refactored them into two parameterized tests, one to test all the login components and one to test the signup components. A third issue was flagged in our estore-api which tested our Java code in the file: src/.../java/com/estore/api/estoreapi/persistence/MilkFileDAO.java. In our load() function, the error states to 'Make the enclosing method "static" or remove this set.' If given the proper time, our team should look into this issue and analyze how to correct this. The report states that correctly updating a static field from a non-static method is tricky to get right and could lead to bugs but will continue to raise an issue each time a static field is updated from a non-static method. Overall, our team produced a project with very few errors and if given more time I would recommend we correct any little issues that don't seem important to really produce a polished, high end project.

estore-api src/.../java/com/estore/api/estoreapi/persistence/LoginFileDAOTest.java See all issues in this file

```
69  jere…          String result = loginFileDAO.login(new Login(username, password));
70  jere…          assertEquals(null, result);
71              }
72
73              @Test
74              public void testSignup() throws IOException {
```

Replace these 3 tests with a single Parameterized one.

```
75                  String username =  1 "admin1";
76                  String password = "admin";
77
78                  String result = loginFileDAO.signup(new Login(username, password));
79                  assertEquals( 2 "admin1", result);
80              }
81              @Test
82              public void  3 testSignupExistingUsername() throws IOException {
83                  String username = "admin";
84                  String password = "admin";
85
86                  String result = loginFileDAO.signup(new Login(username, password));
87                  assertEquals(null, result);
88              }
89              @Test
90              public void  4 testSignupEmptyField() throws IOException {
91                  String username = null;
92                  String password = "admin";
93
94                  String result = loginFileDAO.signup(new Login(username, password));
95                  assertEquals(null, result);
```

estore-api src/.../java/com/estore/api/estoreapi/persistence/LoginFileDAOTest.java See all issues in this file

```
44  jere…          when(mockObjectMapper.readValue(new File("random.txt"), HashedLogin[].class)).thenReturn(testLogins);
45  jere…          loginFileDAO = new LoginFileDAO("random.txt", mockObjectMapper);
46              }
47
48              @Test
49  jere…       public void testLogin() throws IOException {
```

Replace these 3 tests with a single Parameterized one.

```
50  jere…           String username =  1 "admin";
51                  String password =  2 "admin";
52
53  jere…           String result = loginFileDAO.login(new Login(username, password));
54  jere…           assertEquals( 3 "admin", result);
55  jere…       }
56              @Test
57  jere…       public void  4 testLoginWrongPassword() throws IOException {
58  jere…           String username = "admin";
59                  String password = "admin1";
60
61  jere…           String result = loginFileDAO.login(new Login(username, password));
62  jere…           assertEquals(null, result);
63  jere…       }
64              @Test
65  jere…       public void  5 testLoginEmptyField() throws IOException {
66                  String username = "";
67  jere…           String password = "admin";
68
69  jere…           String result = loginFileDAO.login(new Login(username, password));
70  jere…           assertEquals(null, result);
```

estore-api src/.../java/com/estore/api/estoreapi/persistence/MilkFileDAO.java See all issues in this file

```
109          /**
110 yaro…      * Loads {@linkplain Milk milks} from the JSON file into the map
111 1134…      * <br>
112          * Also sets next id to one more than the greatest id found in the file
113          *
114          * @return true if the file was read successfully
115          *
116          * @throws IOException when file cannot be accessed or read from
117          */
118         private boolean load() throws IOException {
119 yaro…       milks = new TreeMap<>();
120 1134…       nextId = 0;
```

Make the enclosing method "static" or remove this set.

```
121
122 nas2…      // Deserializes the JSON objects from the file into an array of milkArray
123 1134…      // readValue will throw an IOException if there's an issue with the file
124          // or reading from the file
125 yaro…      Milk[] milkArray = objectMapper.readValue(new File(filename),Milk[].class);
126 1134…
127 yaro…      // Add each milk to the tree map and keep track of the greatest id
128          for (Milk milk : milkArray) {
129              milks.put(milk.getId(),milk);
130              if (milk.getId() > nextId)
131                  nextId = milk.getId();
```

Make the enclosing method "static" or remove this set.

```
132 1134…      }
133          // Make the next id one greater than the maximum from the file
134          ++nextId;
```

Make the enclosing method "static" or remove this set.

```
135          return true;
136      }
```

milk-app src/app/milk-detail-customer/milk-detail-customer.component.ts See all issues in this file

```
25
26   getMilk(): void {
27     const id = parseInt(this.route.snapshot.paramMap.get('id')!, 10);
28     this.MilkService.getMilk(id)
29       .subscribe(milk => this.milk = milk);
30   }
31
32   goBack(): void {
33     this.location.back();
34   }
35
36 daj4…  rateMilk(rating: number): void {
37 daj4…    if (this.milk != undefined) {
38           var newRatings: number[] = this.milk.rating;
```

Unexpected var, use let or const instead.

```
39           var len: number;
```

Unexpected var, use let or const instead.

```
40           var sum: number = 0;
```

Unexpected var, use let or const instead.

```
41           if (newRatings == null) {
42             newRatings = [rating];
43             len = 1;
44           } else {
45             len = newRatings.push(rating);
46           }
47           this.milk.rating = newRatings;
48           for (let i = 0; i < len; i++) {
```

Testing

> *This section will provide information about the testing performed and the results of the testing.*

Acceptance Testing

> Most of the user stories have passed their acceptance tests, but an acceptance criteria that did not pass was to have a message saying "No products available" when there are no items in the inventory. This is not a huge concern and we will fix that next Sprint.

Unit Testing and Code Coverage

> We have included unit testing for the Milk object, shopping cart, and login. We have not finished the unit testing for the login page, but we will do that in the next Sprint.

estore-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.estore.api.estoreapi.persistence		90%		62%	11	47	13	115	2	27	0	3
com.estore.api.estoreapi		88%		n/a	1	4	2	7	1	4	0	2
com.estore.api.estoreapi.controller		100%		100%	0	23	0	88	0	18	0	3
com.estore.api.estoreapi.model		100%		85%	2	39	0	64	0	32	0	2
Total	60 of 1,275	95%	17 of 64	73%	14	113	15	274	3	81	0	10