PROJECT 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 |

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

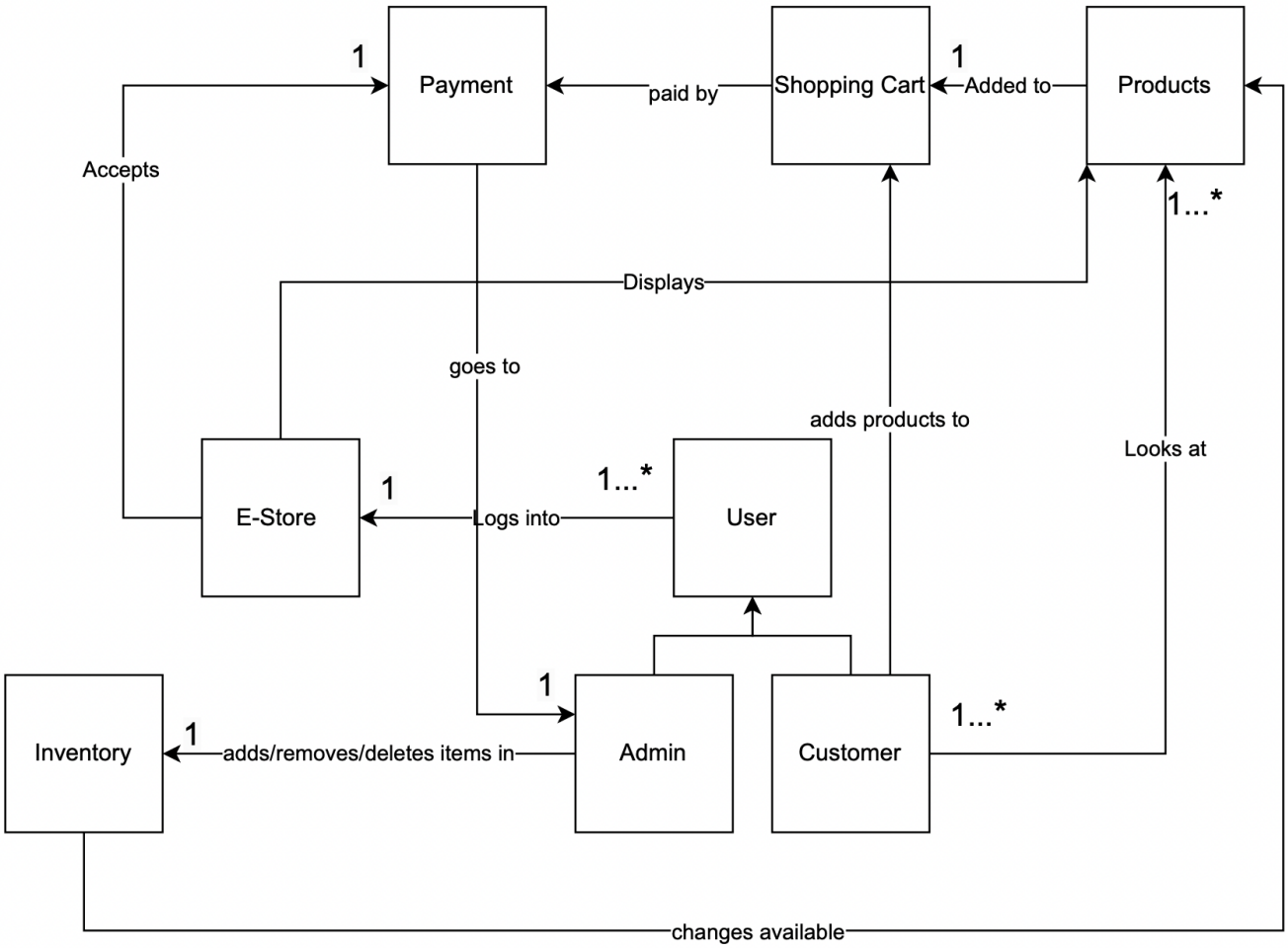
> Will be implemented in a later Sprint

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.

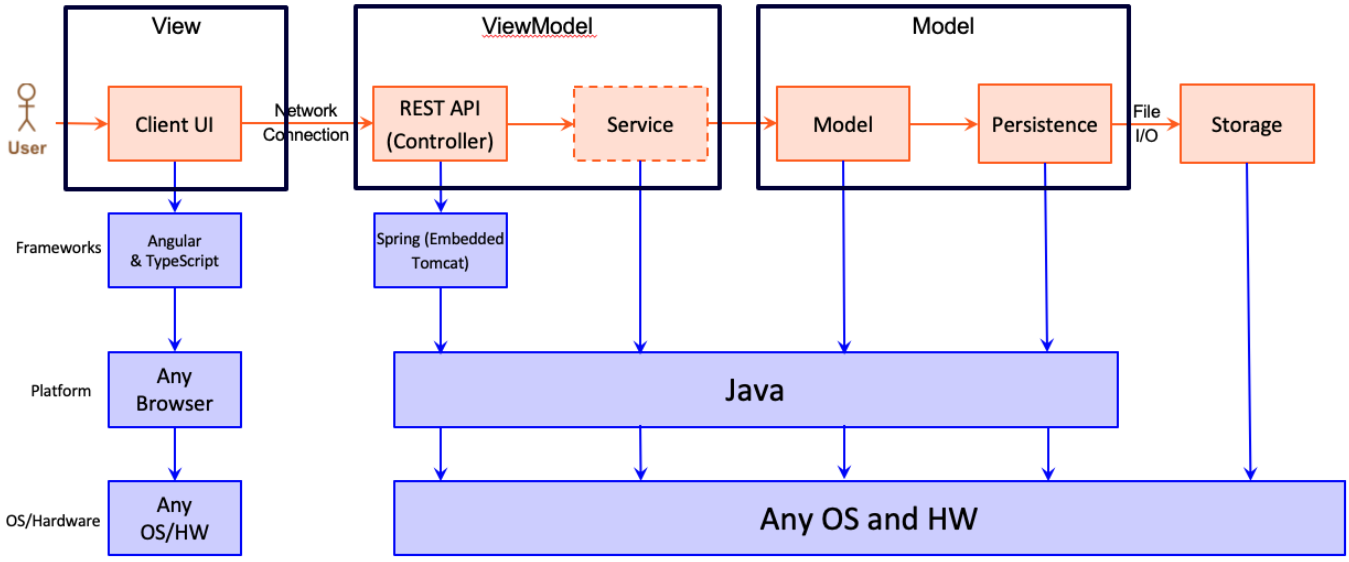


Architecture and Design

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

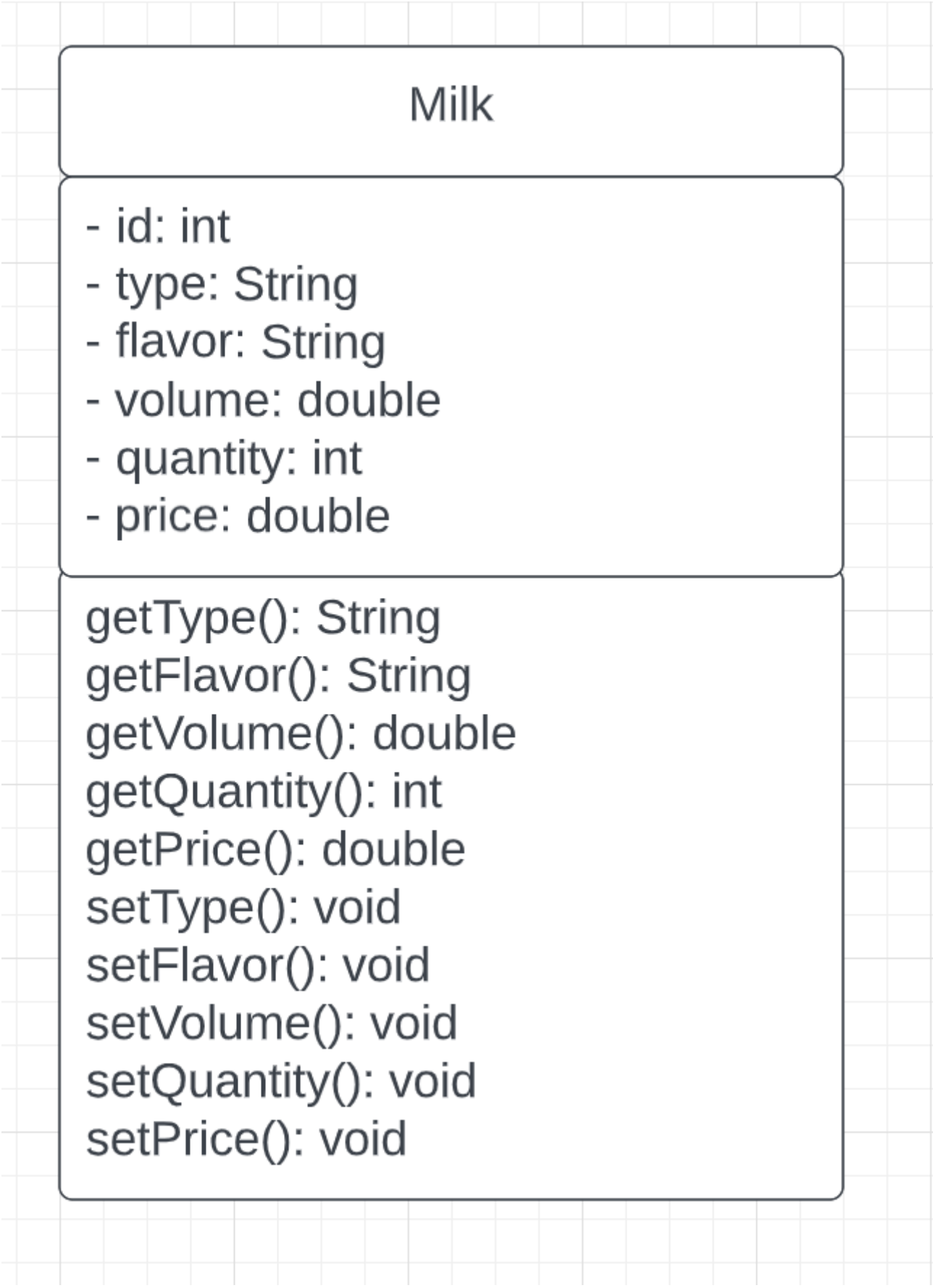
> Will be implemented in a later Sprint

ViewModel Tier

> Will be implemented in a later Sprint

Model Tier

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

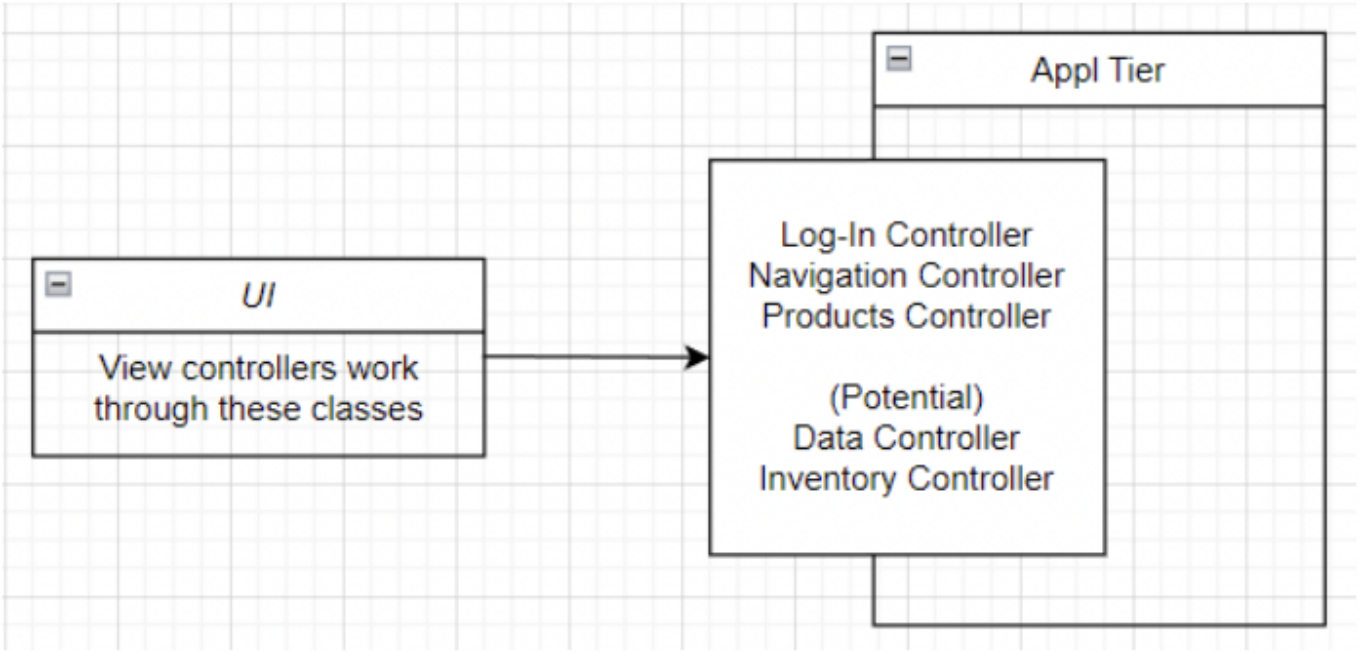


OO Design Principles

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

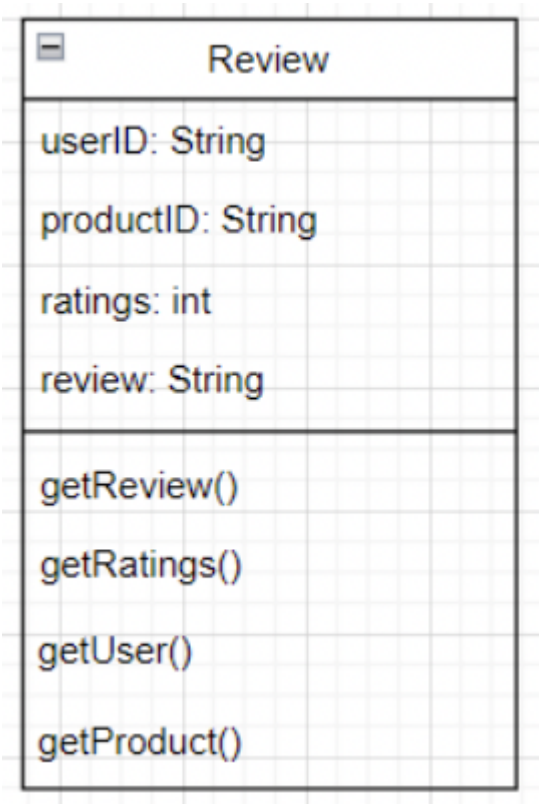
several. One section of our store that should use its own controller is the log-in page. The UI controller would communicate with a log-in controller to perform tasks, such as creating new accounts, checking to see if a password matches with the user, and checking if the user in question has admin access. 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

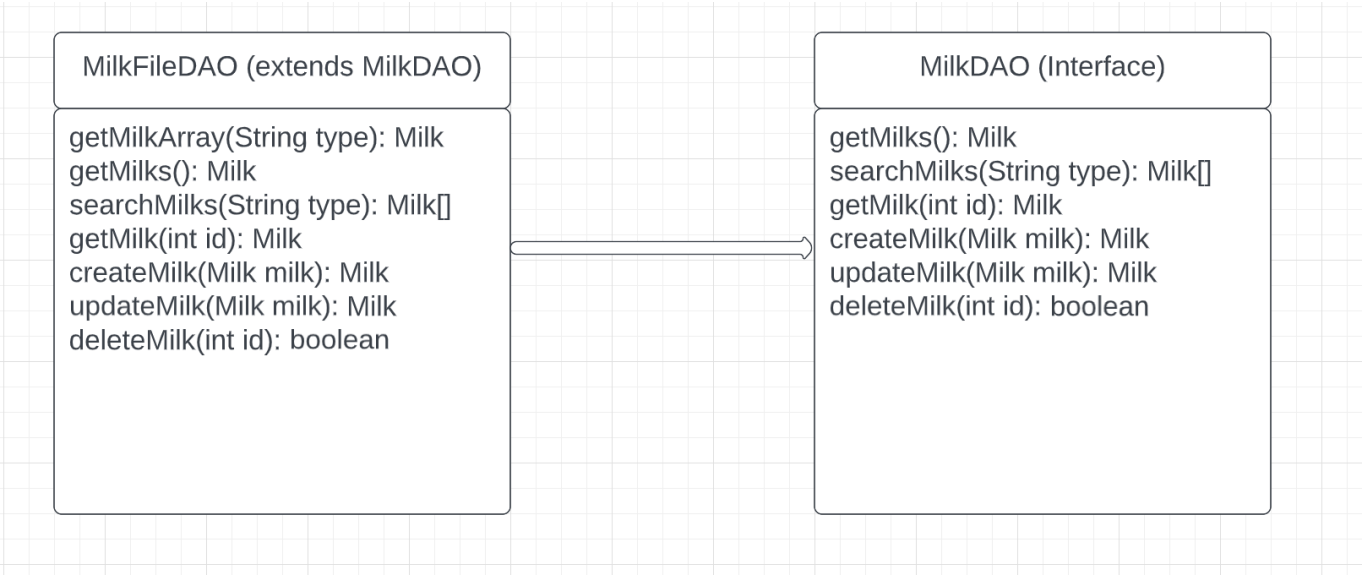
controller that manages the process of checking out. Alternatively, however, it would also be possible to split up something larger into different responsibilities, each one with a class to handle it. An example of this in action would be splitting up the data controller into their respective data types. 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

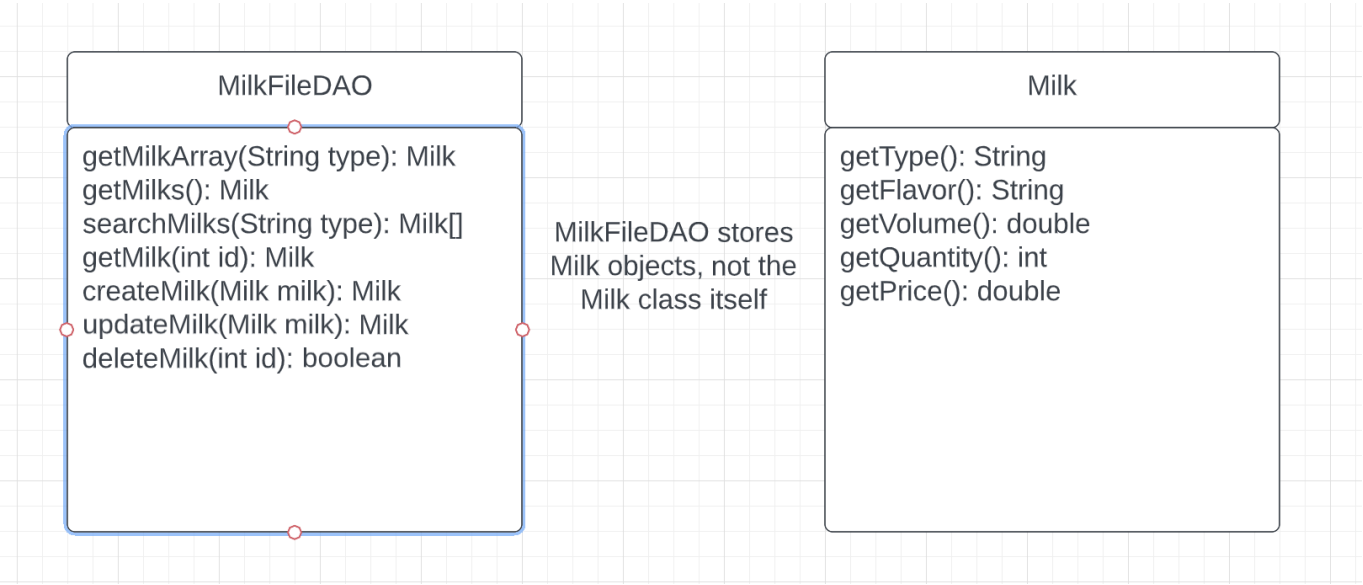
The principle of open/closed states that software entities should be open for extension but closed for modification. A good example of this is inheritance, interfaces, and polymorphism. A class should only be designed for one purpose (as stated by single responsibility) and should be polished to do that one thing. Adding or changing things to it would not make it closed for modification. If many people were using a library and the owner of that library modified the base code for it, essentially changing its function then that would lead to disaster. However if the owner instead built things on top of that code and inherited from it, then that is what we are looking for. 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. 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 called perhaps “StoreProducts” 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

> Will be implemented in a later Sprint ***[Sprint 4]*** *Discuss **future** refactoring and other design improvements your team would explore if the team had additional time.*

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