PROJECT Design Documentation

Team Information

- Team name: Cow Related Pun
- Team members
  - Brendan Battisti
  - Alice Cauchi
  - Brian McNulty
  - John West

Executive Summary

This is a summary of the project.

Purpose

Our application is an e-store for a butcher that provides a variety of cuts of beef of different grades, as well as the ability to sponsor and reserve different cuts from cows.

Glossary and Acronyms

Term	Definition
CT	Catalog
LB	Logout button
LP	Login Page
FPP	Featured Products Page
SC	Shopping Cart
SPA	Single Page
INV	Inventory
AD	Admin Dashboard
PP	Product Page

Requirements

This section describes the features of the application.

> *In this section you do not need to be exhaustive and list every story. Focus on top-level features from the Vision document and maybe Epics and critical Stories.*

- Users login via LP
- Users logout via LB

- Users are directed to FPP upon sign in
- Customers are able to browse and search CT for beef
- Customers may click on beef to go to PP
- Customers may add beef to and view SC
- Admins may not add beef to or view SC
- Admins may add, remove, or update beef to INV via AD
- Admins may not add or update beef fields to be negative on AD

Definition of MVP

The MVP includes minimal user authentication for admins and users, with a reserved admin account and other usernames being user accounts, the ability for customers to search for products and add them to shopping carts, and the ability for admins to add, remove, and edit product data.

MVP Features

> ***[Sprint 4]*** *Provide a list of top-level Epics and/or Stories of the MVP.*

- Create User File
- New Account Page
- Catalog Page
- Admin Dashboard
- Landing Page
- Product Page
- Shopping Cart
- Admin/Customer Authentication

Enhancements

> ***[Sprint 4]*** *Describe what enhancements you have implemented for the project.*

Application Domain

This section describes the application domain.


Domain Model

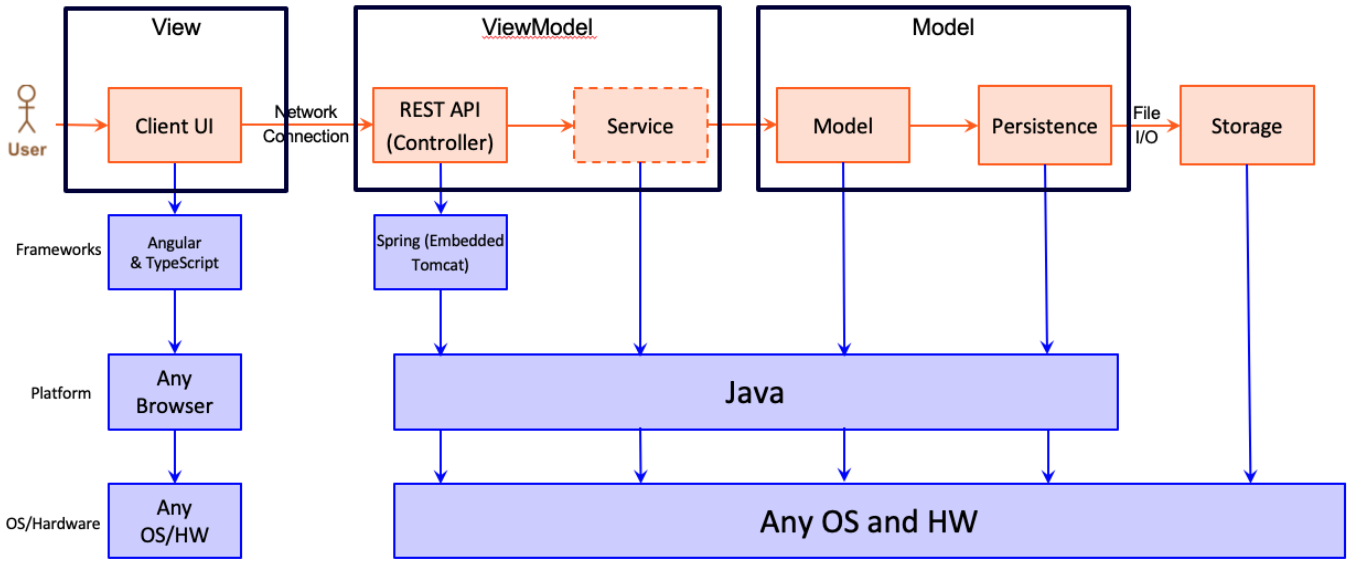
At the center of the domain model is the product entity, which represents beef. Beef is contained in the inventory, and the admin adds different beef items as they become available. These beef items are displayed on the catalog page for users to view and purchase, and users are able to search by name or partial name in order to find the type of beef they want. Both admins and customers are users that log in via the login page, which connects them to the rest of the e-store.Customers have access to an Account page, where they can reset their password, as well as add and remove credit cards for billing.Customers can then add products to their shopping cart. Customers can then purchase their shopping cart at the checkout using one of the cards on file.

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.

When not logged in, the user is first directed to FPP. At the top of the screen, at this point in time, there is no navigation bar, so the user must enter links to links to FPP, CT, SC, LP, and AD. The user is not able to access SC if not signed in. If the user is logged in, the LP is replaced with LB, and they gain access to SC. On the FPP, the user sees the 4 featured products, the price, and the weight available. The user can click on a product to go to the PP. Alternatively, if the user goes to the catalog, they may then type in part of a product name to search for the product, and then click on the desired product to go to its PP. After going here, the user can add a quantity to their SC via a button. Admins have access to these features, but they may not add items to their SC or view SC page, as they are redirected upon going to the page. Admins can navigate to ND, where they see the product fields to add products. Any error messages from illegal operations are shown at the top of the screen. Below, each product and its fields are listed, with a textbox for editing the price and adding weight, and a button to delete the product.

> *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

> ***[Sprint 4]*** *Provide a summary of the View Tier UI of your architecture. Describe the types of components in the tier and describe their responsibilities. This should be a narrative description, i.e. it has a flow or "story line" that the reader can follow.*

> ***[Sprint 4]*** *You must provide at least **2 sequence diagrams** as is relevant to a particular aspects of the design that you are describing. For example, in e-store you might create a sequence diagram of a customer searching for an item and adding to their cart. As these can span multiple tiers, be sure to include an relevant HTTP requests from the client-side to the server-side to help illustrate the end-to-end flow.*

When the estore goes online, the user and the admins view the landing page that holds the feature products and the navbar, which will be on every page. This is controlled by the home-page and nav-bar components. From there the users are able to see the log-in page and log-in with their respective username, which is controlled by the login component. Both users, customers and admins, will be redirected to the landing page after log in. Both admin and customers can view the catalog and product details that are controlled via the product-details and catalog components. From there the admin can view the inventory in the admin dashboard, via the admin-dashboard component. Because the admin has no access to a shopping cart, if the admin were to try and access a shopping cart, they will also be redirected to the admin dashboard. Customers cannot access the admin dashboard, but they do have access to the shopping cart and checkout. The view of the shopping cart is controlled by the cart compononent. Once a customer is done shopping and the admin is finished, they can log out view the log out compononent.

ViewModel Tier

> ***[Sprint 4]*** *Provide a summary of this tier of your architecture. This section will follow the same instructions that are given for the View Tier above.*

> *At appropriate places as part of this narrative provide **one** or more updated and **properly labeled** static models (UML class diagrams) with some details such as critical attributes and methods.*

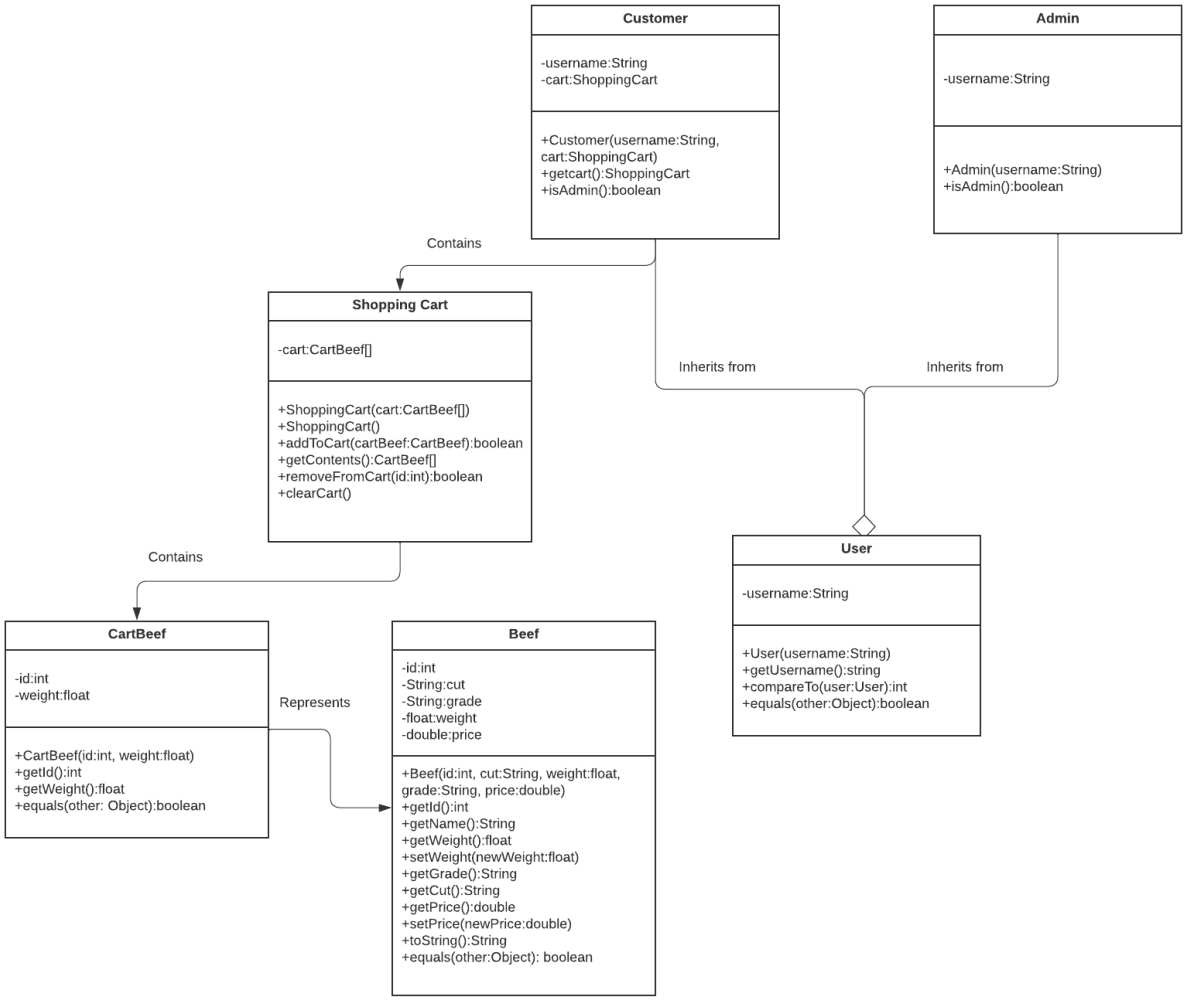
 Replace with your ViewModel Tier class diagram 1, etc.

Model Tier

·–

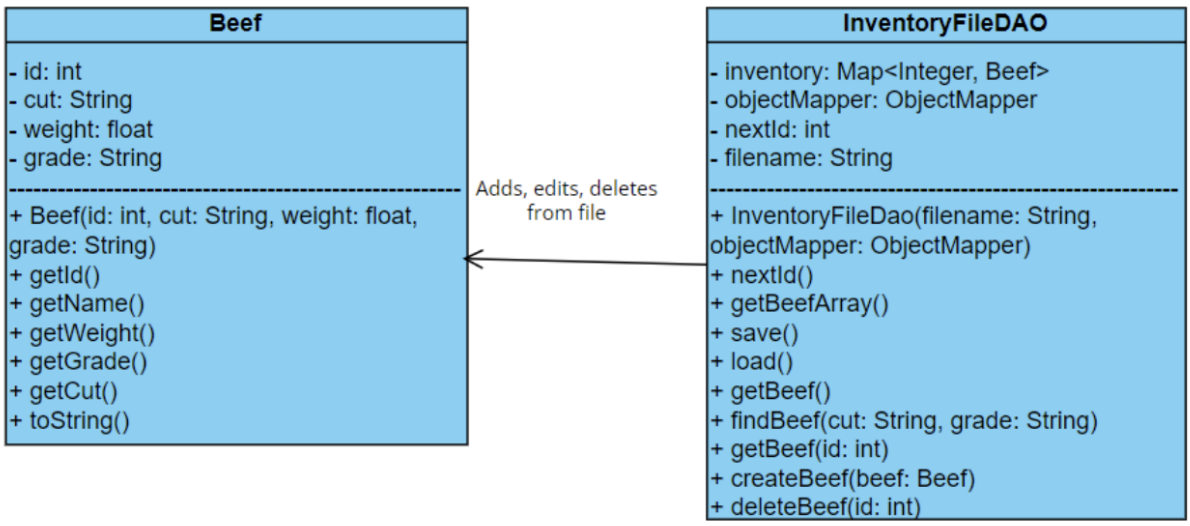
The model tier consists of beef, shopping carts, and users, which may be customers or administrators. Customers have a shopping cart, that may contain zero or more products, whereas admins do not. Shopping carts are represented by a class, with CartBeef representing Beef as an item in the shopping cart. Both users are identified by username. Beef has an incrementing id as its identifier, and has a grade, cut, weight, and price. Beef contains methods to add, get, delete, and get all beef objects, as well as updating the price and field of beef objects. Users can add items to their shopping cart field via a method, but this method and attribute does not exist for administrators. Within admin and customer classes, there exists a method to validate whether or not the use is an admin, which is used in the view tier for the acceptance criteria. There are

also methods to get users by username, and create adminis and customer if necessary from the backend.



OO Design Principles

Single Responsibility:

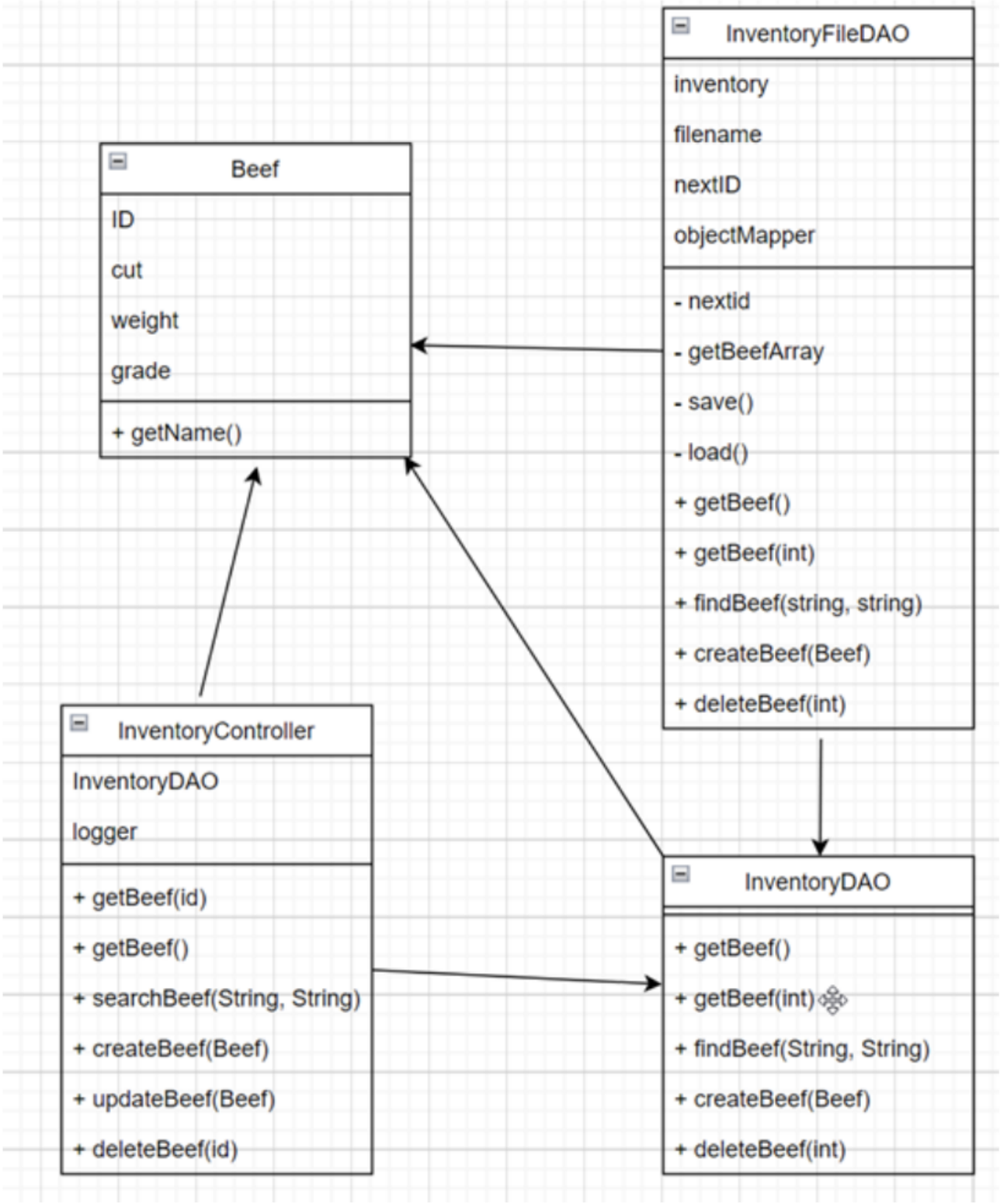


In the sample UML snippet above, both the Beef class and the InventoryFileDAO class have distinctly separate responsibilities. The Beef class handles the creation of Beef objects and provides methods for accessing the

attributes of a given Beef object, including the item id, the cut, the grade, and the weight. The InventoryFileDAO class handles the sole responsibility of adding, removing, and editing Beef objects from the inventory file, with various methods used to get specific Beef objects from the file given the attributes of the beef. InventoryFileDAO makes use of the methods found in Beef to handle the creation and deletion of Beef objects because it is the responsibility of InventoryFileDAO to handle inventory file concurrency. It would be possible to implement a nested class that served the same functionality in InventoryFileDAO for these purposes, but isolating the classes by responsibility allows for future classes to make use of the Beef class. Therefore, the separation of the Beef and InventoryFileDAO classes is representative of the single responsibility principle. Our front end also follows single responsibility by separating the responsibilities of each component and service as much as possible. We use a singular logging service for managing all of the error logging for our application. We also use separate services which correspond to each controller on our backend to maintain our separation of responsibilities all the way through our application

Low Coupling:

Low coupling already exists within the back-end systems of our e-store API code. Within our four main classes so far, we have InventoryContoller, InventoryDAO, InventoryFileDAO and Beef. Beef relies on none of the other components within the code, InventoryFileDAO and InventoryDAO only rely on beef, and InventoryController relies on Beef and InventoryDAO. Within the code base we have so far, since there is not much currently there, the only thing I could think of that would improve the low coupling of our current four classes is by changing the implementation of inventoryController to use a generic type and instantiating it by passing the Beef type to it to reduce the dependance and increase the modularity of the class. The only issue with this is it may be increasing the project complexity by unnecessarily adding more classes as we would then have to make a second controller class to extend the generic one we have built which would then rely on the generic class and beef and inventoryDAO which has even more coupling than just using the current inventoryController class that's implemented, however it still gives more freedom with the genericController. Our application adheres to the low coupling principle by keeping our components separated and independent of each other as much as possible. This allows us to reuse these services and components throughout the application, making expansion easier. On top of this, should we need to remove a feature in the future, it would require minimal editing, keeping code clean and maintainable.

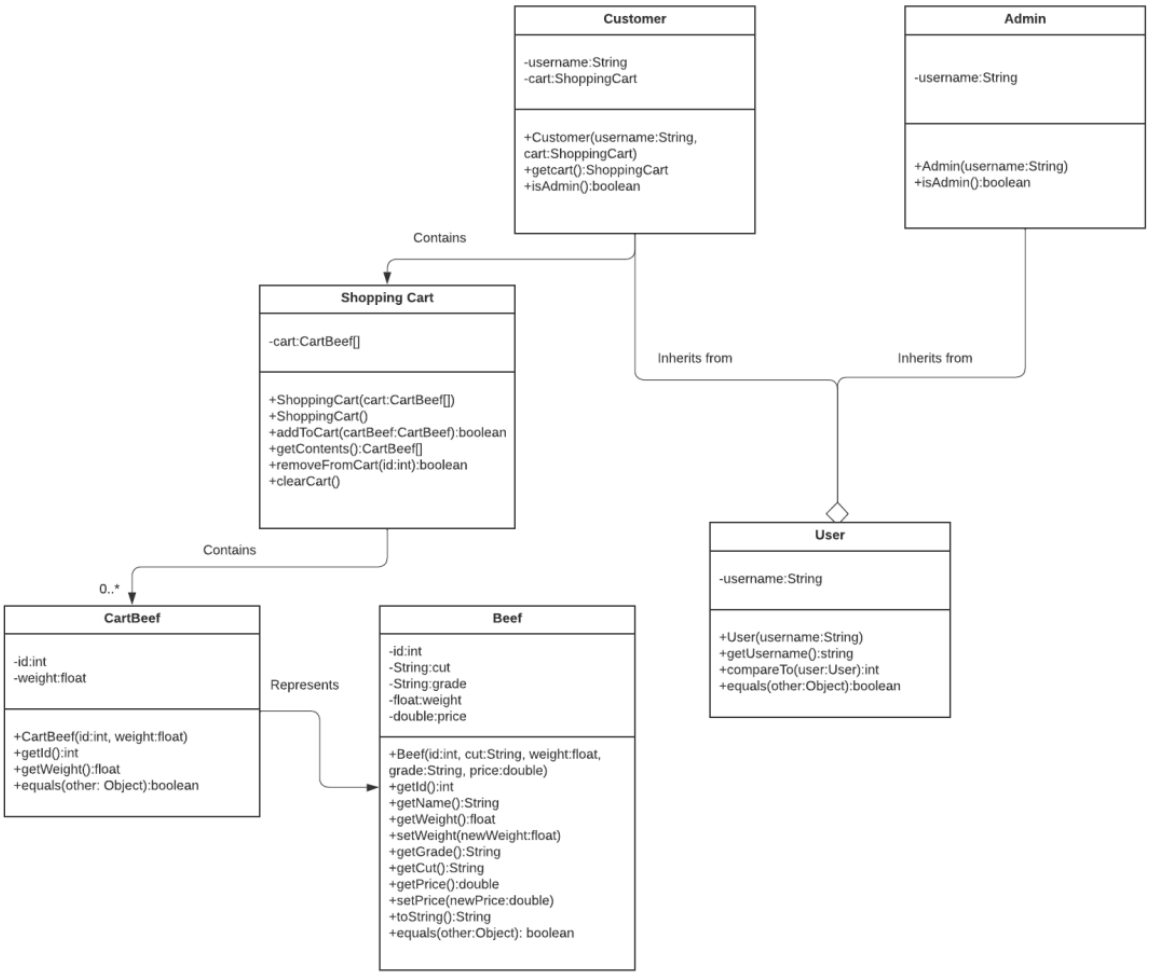


In regard to our domain model, our Customer and Administrator both inherit from the User class, and while this may not be as low coupling as possible, it would violate DRY principle to separate them entirely. Nothing else in our domain model will directly translate to classes so a low coupling analysis of this would not be feasible.

Information Expert:

Information Expert refers to the design principle that decides where to delegate certain responsibilities of classes. The idea behind it is that the methods and computed fields and such should be organized along with where the relevant data for these methods and fields are. In the project we have a beef class that does not have a name field but it does have a name that can be derived from two of the fields within the class,

concatenating the grade with the cut of meat. Doing this outside of the class to compute the name would not follow the information expert design principle, which is why instead we have a getName() that exists within the beef class that instead computes the name of the beef by concatenating the two properties internally and then returning the string. We also have similar methods for retrieving the cut, price, weight, and grade, when the attributes are used for display on admin dashboard, product pages, and the home page.



This principle is also upheld within the Shopping Cart class, as we can retrieve the contents of the shopping cart using a method rather than searching from the list outside of the class. Similarly, the CartBeef object also has methods to get the id and weight of a CartBeef object within the shopping cart for later use on the front end.

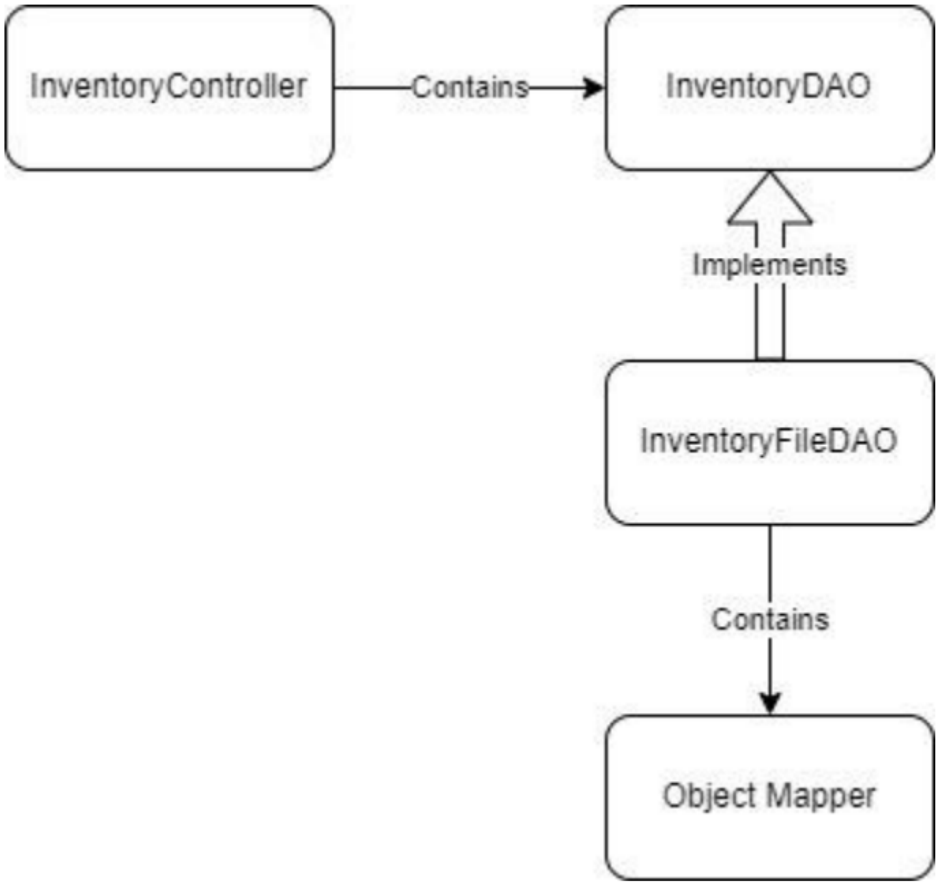
The information expert principle is followed within both subclasses of User, Admin and Customer. Both classes contain the username, with a related method to get the username for use in front end when logging in the user and when displaying messages related to the user.

The information expert principle is also applied within the InventoryFileDAO class as it contains a map of the current inventory, and while we are able to get that entire list we can also use extra methods which are contained within the same class for more accessible options rather than getting the list and searching it outside of the class. The methods within this class that most fit the information expert principle are the findBeef and getBeef(id).

Due to the fact that the information expert is such an implementation level concept, we do not have much code for me to recommend any new suggestions for at this time as we have only implemented the create and get products for now. While we were able to look towards the domain diagram for bits and pieces of information towards some of the other principles, this will not apply to information expert as it looks more towards methods and class data rather than a high level view of the classes that are presented within the domain model.

As for ideas later down the line of development when we hit the 10% feature we will need to look closely into its application within the cow class. Things such as owner comparison rather than just get owner and having to compare outside the class will be essential there. Looking back at the beef class, we will consider adding a sellBy date and an information expert method would be checking if it is expired. We also adhere to the information expert principle. By keeping our components and services focussed on their individual responsibilities. This creates a guideline for us to separate our components and services, which helps us adhere to other design principles. For instance, our user service only deals with users, and our beef service only deals with beef. While these could easily be combined into a backend service, keeping them separate keeps the code clean and organized.

Dependency Injection:



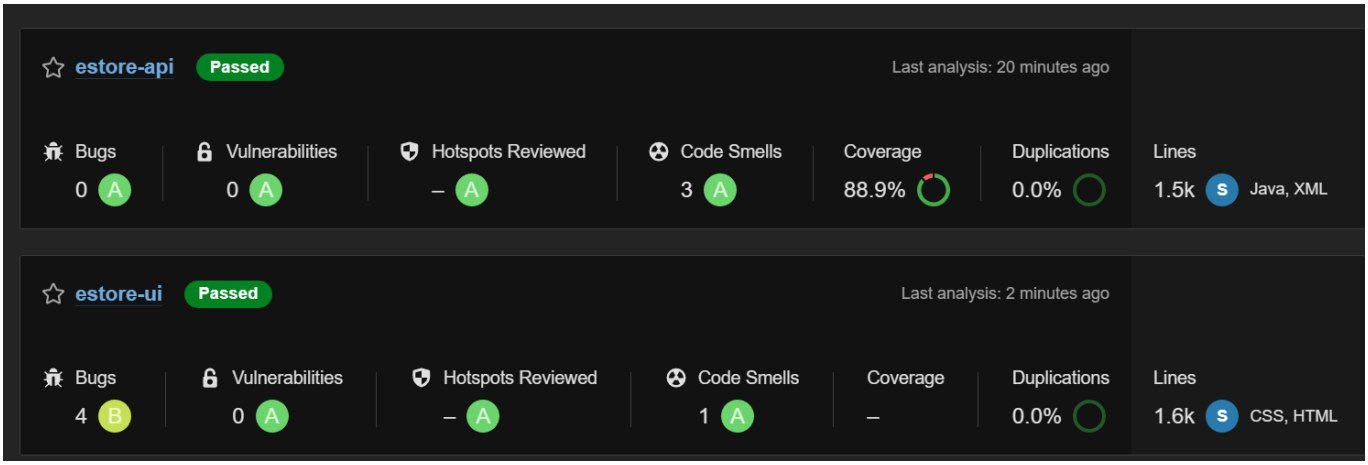
Dependency Injection is used commonly in our application. It is used in the constructor functions of the InventoryController and the InventoryFileDAO constructor. The InventoryController is injected with an InventoryDAO object for inventory interfacing. The InventoryFileDAO's constructor is injected with an ObjectMapper object which controls the conversion of Java objects into a JSON representation. These dependencies allow for a looser coupling in the code. This way the InventoryFileDAO does not need to control

any of the object mapping, making the code easier to organize and maintain. The InventoryController also benefits greatly from dependency injection as it reduces the responsibility of the class, once again leading to more organized and maintainable code. Dependency injection is also used heavily on our front end for allowing components to communicate with services. It allows components such as new-user, login, and logout, to each have access to the userService service. This keeps our code clean and separated from adhering to other design principles, and it makes code maintenance easier.

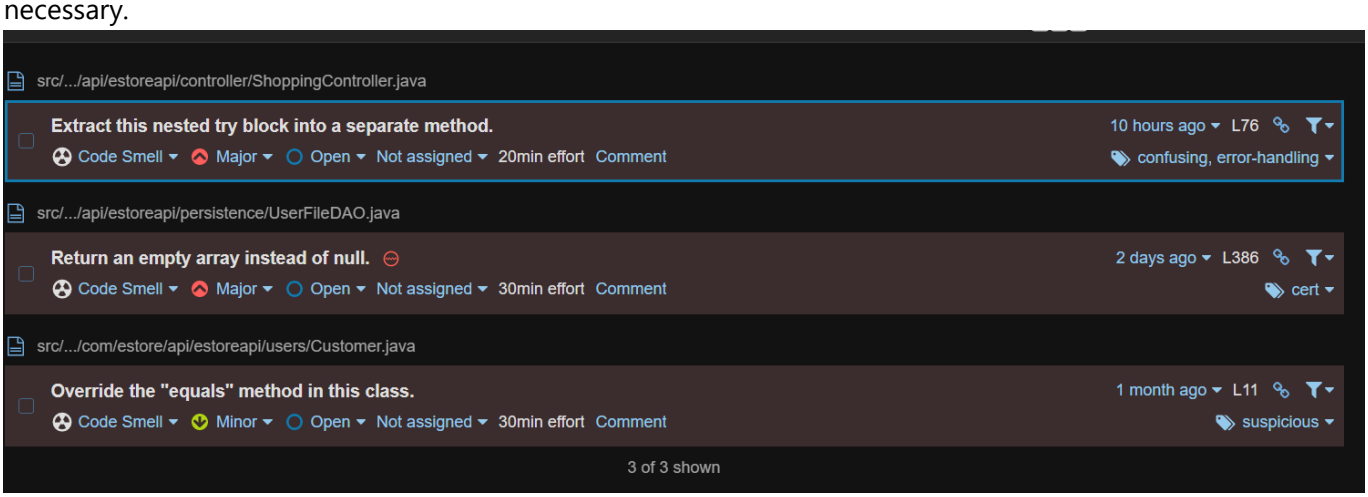
Static Code Analysis/Future Design Improvements

> ***[Sprint 4]*** *With the results from the Static Code Analysis exercise,* ***Identify 3-4*** *areas within your code that have been flagged by the Static Code Analysis Tool (SonarQube) and provide your analysis and recommendations.*
> *Include any relevant screenshot(s) with each area.*

> ***[Sprint 4]*** *Discuss* ***future*** *refactoring and other design improvements your team would explore if the team had additional time.*



The majority of our issues in sonarQube came from improper return values and incorrect coding practices. There were 2 different instances where we should have been returning one value type, but were returning null.The other instance is us not overriding a equals method, which for our implementation was not really necessary.



Testing

Acceptance testing was successful, excluding the checkout component. Improving code coverage required changing various parts of our implementation due to inadequate return types, but was ultimately successful at

93 percent.

Acceptance Testing

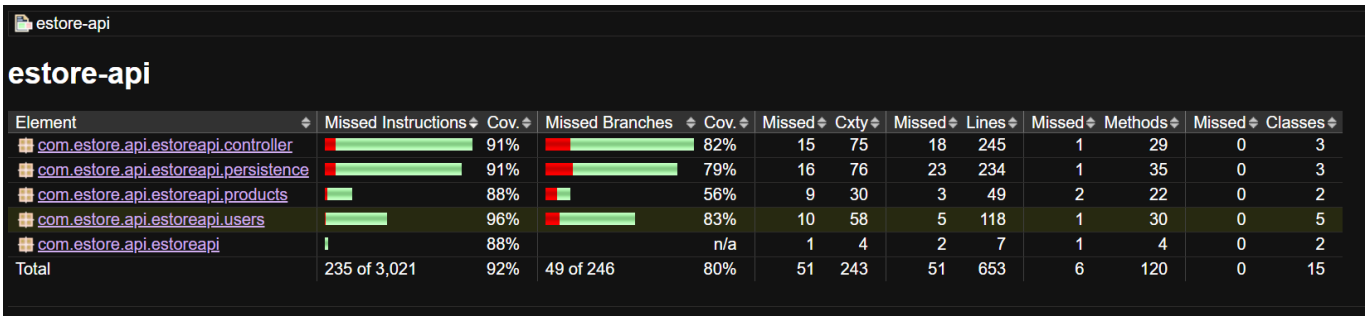
> ***[Sprint 2 & 4]*** *Report on the number of user stories that have passed all their acceptance criteria tests, the number that have some acceptance criteria tests failing, and the number of user stories that have not had any testing yet. Highlight the issues found during acceptance testing and if there are any concerns.*

The number of user stories that passed all criteria is 20.

We did not have any stories that did not pass all the acceptance criteria. This is due to the team doing consistent bug checks throughout the process.

Unit Testing and Code Coverage

> ***[Sprint 4]*** *Discuss your unit testing strategy. Report on the code coverage achieved from unit testing of the code base. Discuss the team's coverage targets, why you selected those values, and how well your code coverage met your targets.*



estore-api

estore-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.estore.api.estoreapi.controller		91%		82%	15	75	18	245	1	29	0	3
com.estore.api.estoreapi.persistence		91%		79%	16	76	23	234	1	35	0	3
com.estore.api.estoreapi.products		88%		56%	9	30	3	49	2	22	0	2
com.estore.api.estoreapi.users		96%		83%	10	58	5	118	1	30	0	5
com.estore.api.estoreapi		88%		n/a	1	4	2	7	1	4	0	2
Total	235 of 3,021	92%	49 of 246	80%	51	243	51	653	6	120	0	15

Using mock objects, our team tested the mode, controller, and persistence tiers. We achieved an overall code coverage of 92 percent. Given an average of 90 percent was the goal for the overall average, we set a target for 90 percent minimum for each of these tiers, with the main estore-api folder and the estore-api products folder being the exception at 88 percent. Overall, our unit testing for this phase was cohesive.