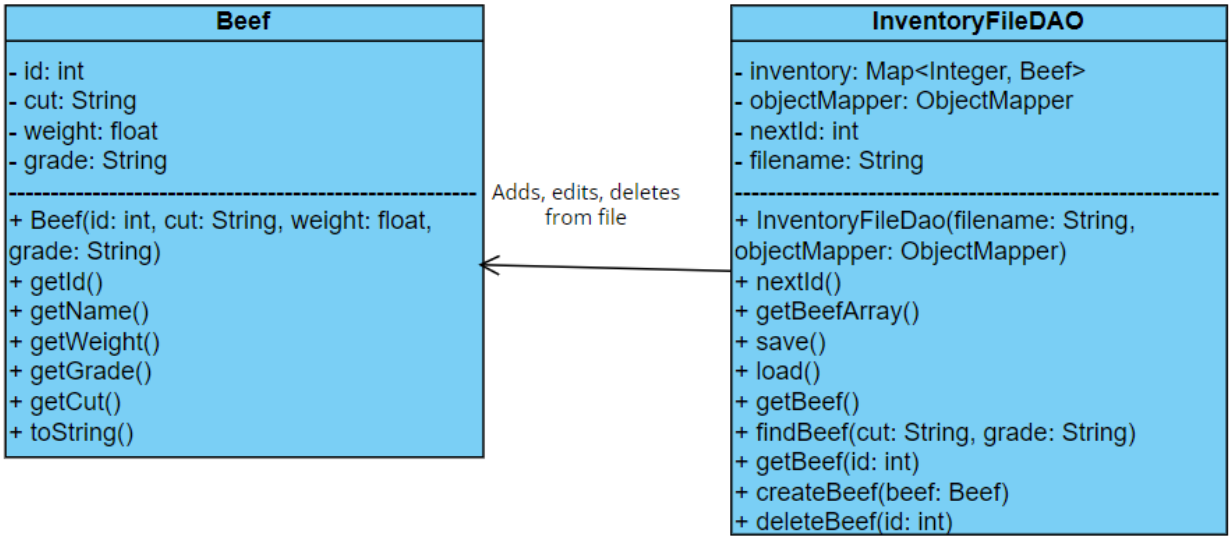
Single Responsibility

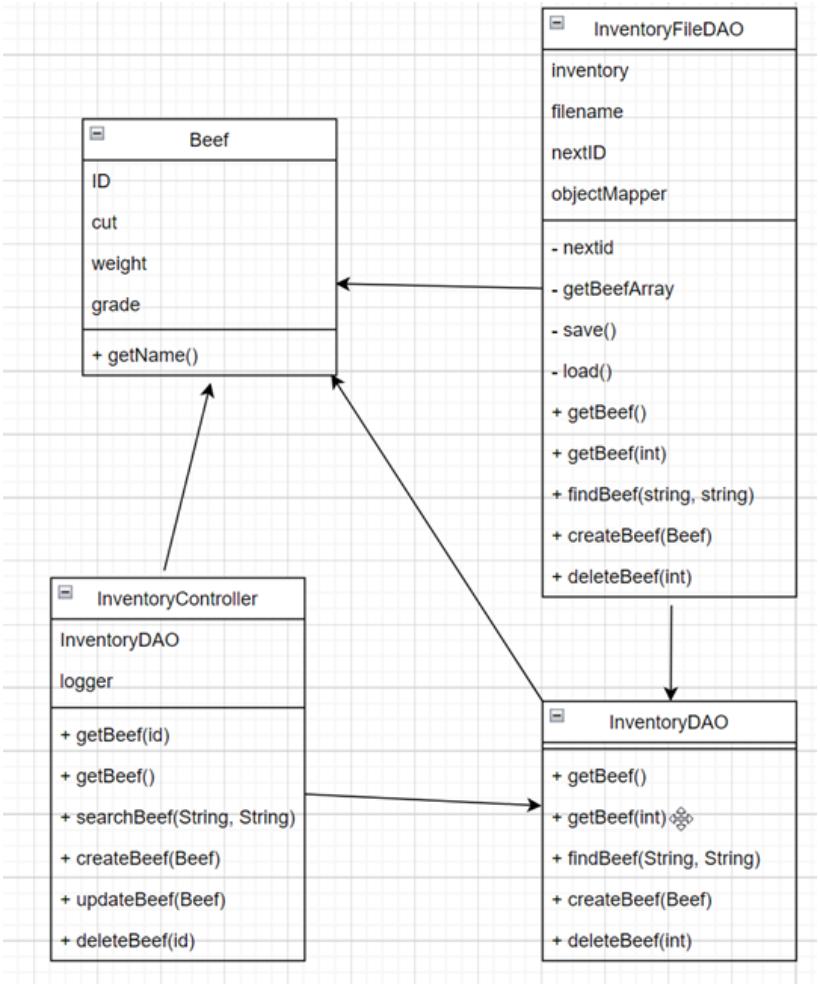


In the sample UML snippet above, both the Beef class and the InventoryFileDAO class have distinctly separate responsibilities. The Beef class handles the creation of Beef objects and provides methods for accessing the attributes of a given Beef object, including the item id, the cut, the grade, and the 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.

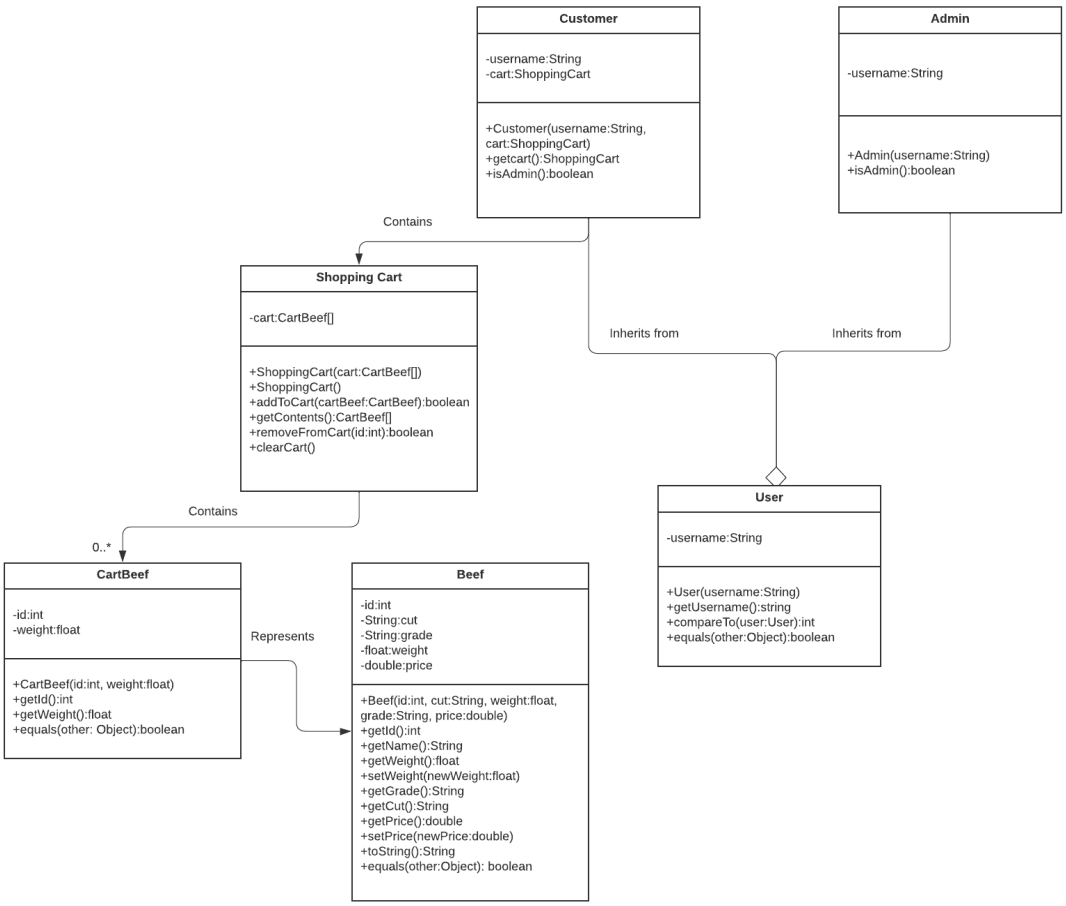


In regard to our domain model, our Customer and Administrator both inherit from the User class, and while this may not be as low coupling as possible, it would violate DRY principle to separate them entirely. Nothing else in our domain model will directly translate to classes so a low coupling analysis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

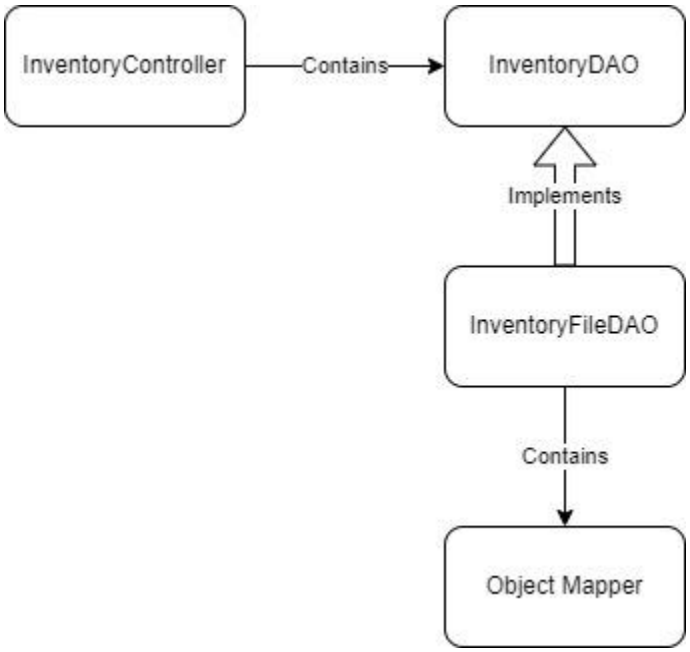
The information expert principle is also applied within the InventoryFileDAO class as it contains a map of the current inventory, and while we are able to get that entire list we can also use extra methods which are contained within the same class for more accessible options rather than getting the list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