PROJECT Design Documentation

Team Information

- Team name: Quackers
- Team members
    - Mason Bausenwein
    - Travis Hill
    - Beining Zhou
    - Eric Choi
    - Andrew Le

Executive Summary

This is a summary of the project.

Purpose

Our goal is to create an onlne e-store that sells highly customizable ducks. We aim to be able to fulfill both pre-made ducks and user customized ducks.

Glossary and Acronyms

Term	Definition
SPA	Single Page
MVP	Minimum Viable Product
UML	Unified Modeling Language
OCP	Open/Closed Principle

Requirements

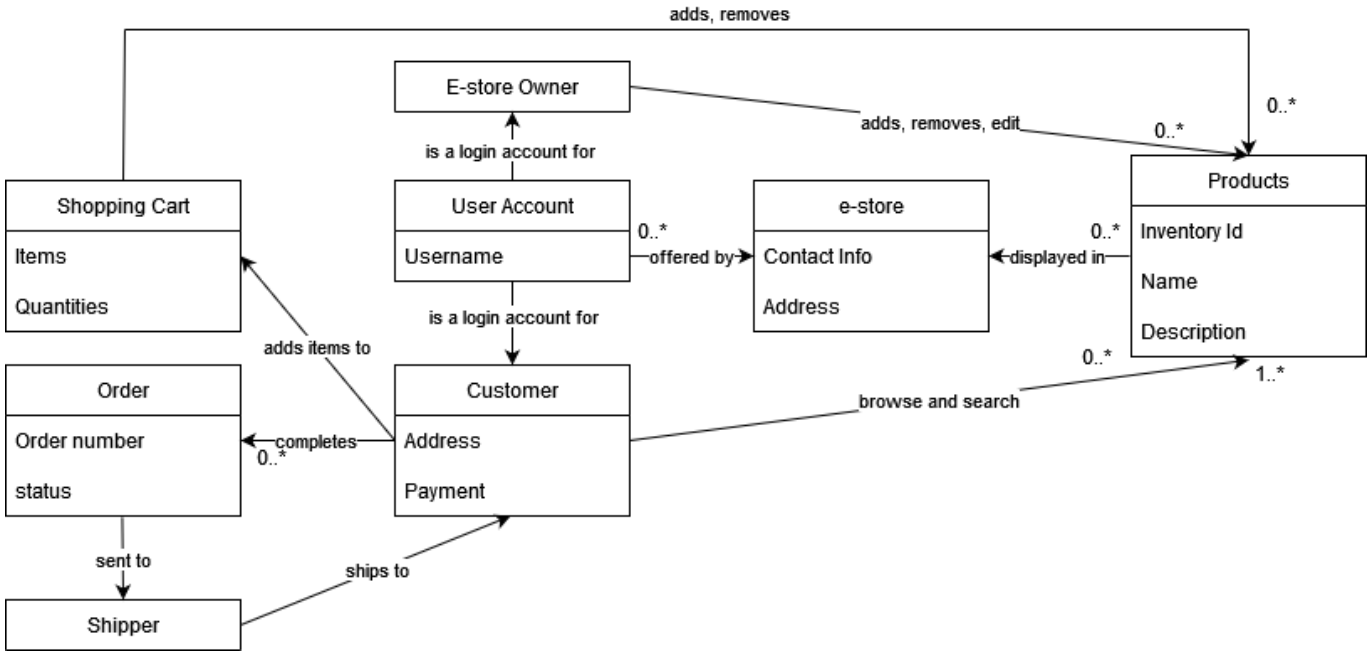
This section describes the features of the application.

Definition of MVP

As of right now, our customers can select and search through a variety of pre-made ducks from our store catalog and add or remove them from their shopping cartsas they please. Users can register accounts to save their previous shopping sessions and login at a later date to resume them. From an owners perspective, they have access to the whole store catalog and can add/remove new products or even update existing ones at the click of a button.

Application Domain

This section describes the application domain.



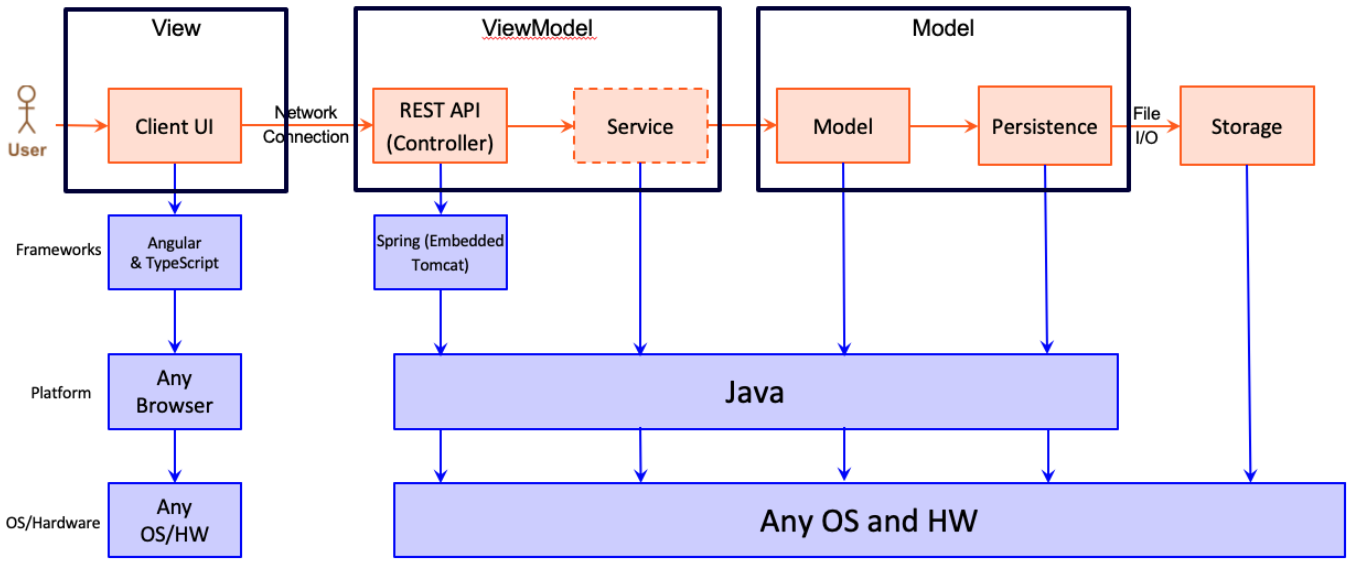
As our goal is to make an e-store, our domain contains entities, such as products, customer, shopping cart, and more. Products are connected to most things are our domain model, which exemplifies how important they are as an entity. For instance, products have to be added and removed from the shopping cart, and customers need to be able to view the products. Customers are also vital to our domain model, as we can't make money if customers cannot use our website. As seen in the model above, customers can browse products, add items to shopping carts, complete orders, and more.

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.

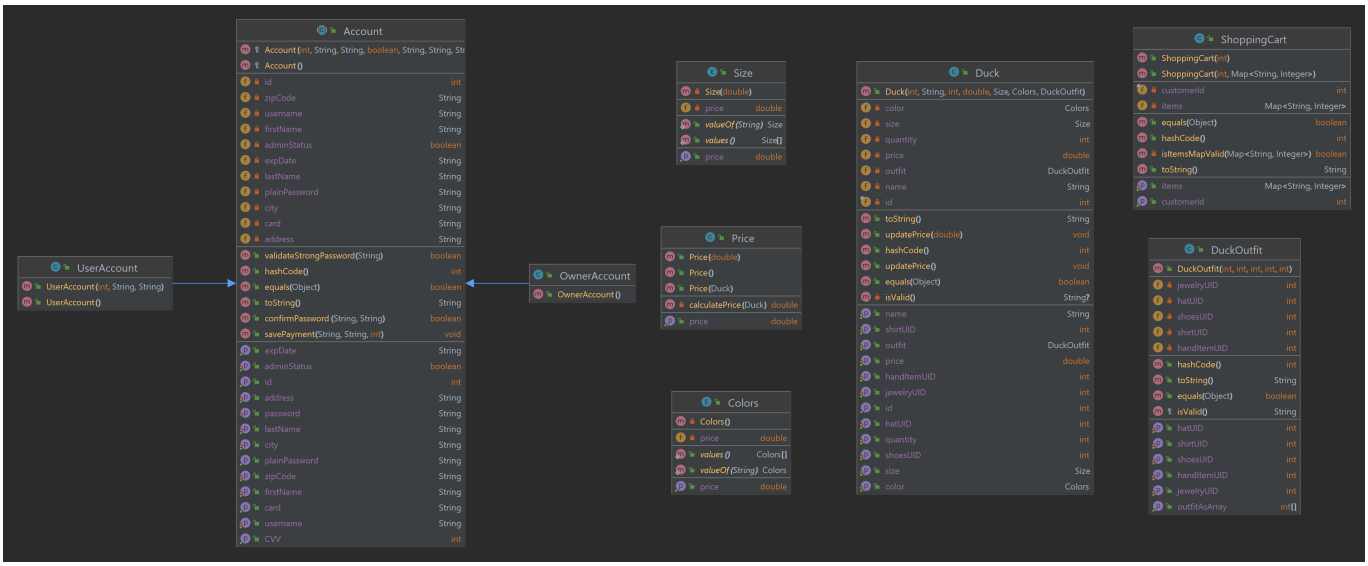
Overview of User Interface

This section describes the web interface flow; this is how the user views and interacts with the e-store application.

Upon opening our website, the user will be greeted with a login page where they can either log in or register. Users who log in with an admin account will be directed to the inventory management page. From this page, they will be able to create, modify, and delete products. If a non-admin attempts to access the inventory management page, they will be redirected to the login page. If the user logs in as a buyer, they will be redirected to a catalog page where they can view all of the available items. Furthermore, they can use a search box to filter through the available items. They are also able to add these items to their cart. Once they have added the items they want, they can proceed to the shopping cart to modify the quantity of each item and checkout. After checking out, they are directed to a page where they are given a receipt.

Model Tier

We have three main classes in our Model Tier and various smaller classes that are utilized in these main classes. These classes are Duck, ShoppingCart, and Account. Our entire website will revolve around these base classes. A user will create an Account to add a Duck to their Shopping Cart to be eventually purchased. The Duck class utilizes three smaller classes defined in the model tier: Colors, Size, and DuckOutfit. Both Colors and Size are enums meant to describe the constant color and size of the duck. DuckOutfit will describe the accessories a duck can have at the time of purchase. All these classes will affect the total price of the duck for the shopper. Account is an abstract class that has two subclasses, UserAccount and OwnerAccount which both inherits Account's base properties. A UserAccount is what a regular shopper will have when they register to the site. A OwnerAccount is created on startup and can not be regsitered or created directly. The UserAccount's ID is linked to their own Shopping Cart ID, an OwnerAccount's ID is not and they do not have access to a shopping cart.

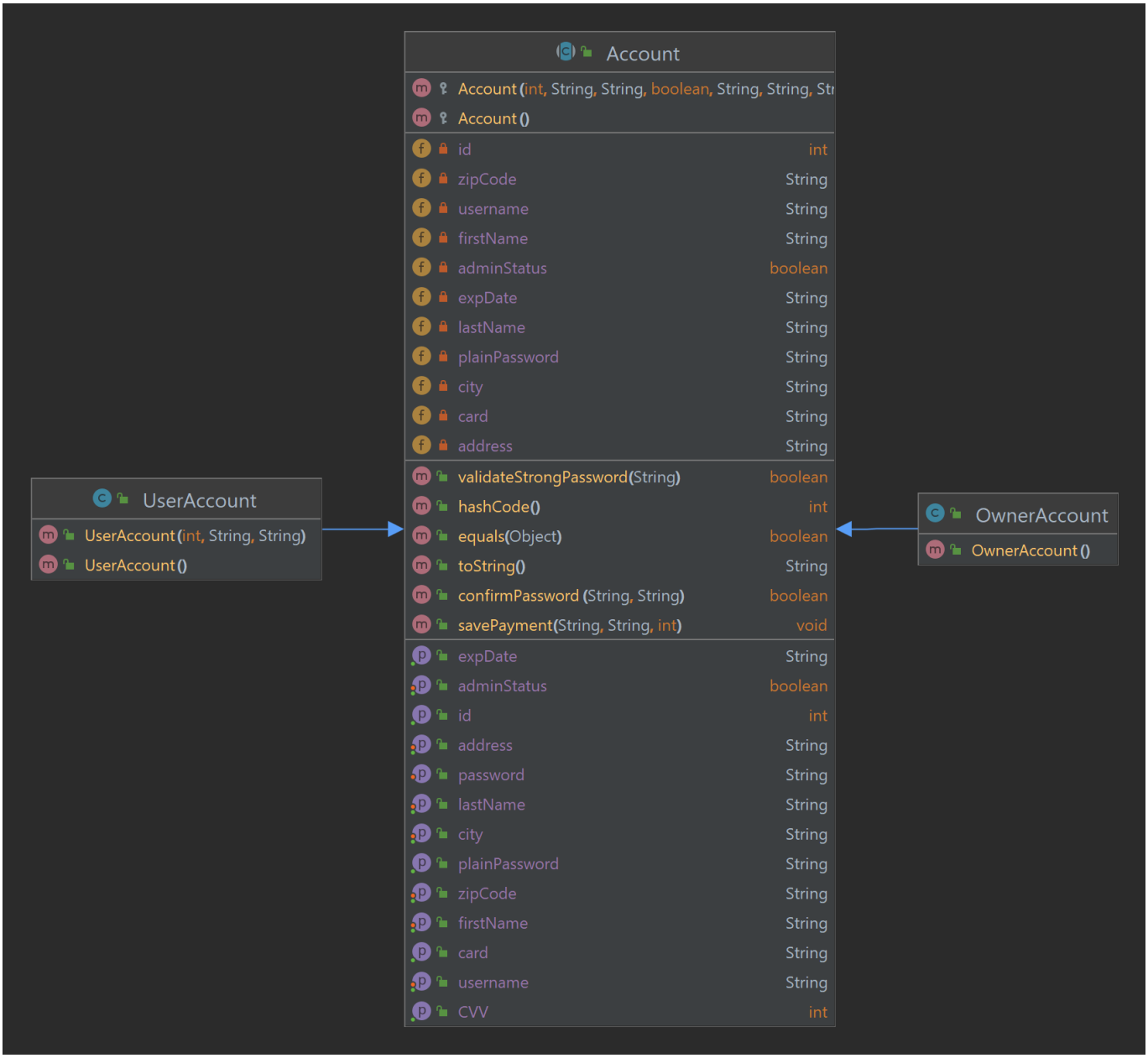


OO Design Principles

Open/Closed Principle

The Open/Closed Principle is a design principle stating that software entities should be open for extension but closed for modification, in order to allow for flexibility and maintainability in the software design.

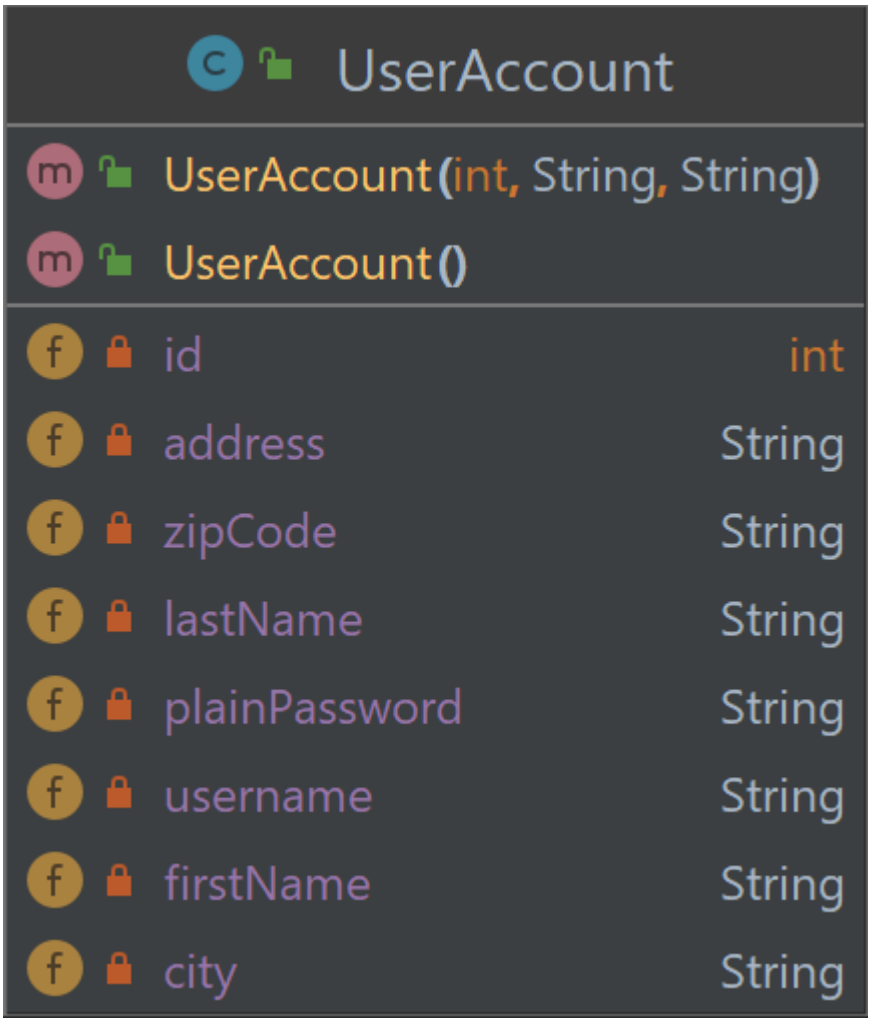
Applications:
The Open/Closed Principle is applied in our design by creating multiple states/ entities that are open for extension but closed for modification by other entities. To further adhere to this principle, we broke down our design into more specific components and creating additional entities that share functionality, allowing for easier extension without the need for modification. An example of this principle in action is our Account class, which is an abstract class that provides essential properties and basic functionality to classes like UserAccount and OwnerAccount, which implement it. UserAccount and OwnerAccount classes meet the Open/Closed Principle by extending the abstract Account class, which is closed for modification. Both UserAccount and OwnerAccount classes inherit the properties and basic functionality provided by the Account class without modifying it. This allows for easy extension of the Account class by adding new classes that also inherit from it, without changing the existing implementation.



Pure Fabrication

Pure Fabrication is a design principle stating that a class or module should be created solely for the purpose of fulfilling a certain functionality or responsibility, without being tied to a specific entity or behavior in the system.

Applications:
This principle is applied in our design with our user account, as seen above in the Model Tier UML diagram. This is merely a way to store a username, password, and payment information in an easy manner. Our authentication system is going to be handling all the creation, establishing, and verifying of the credentials within it. If the user account is being created, the authentication system will store it in its records. If an account already exists and somebody is trying to log in, then our authentication system will take the data from the user account in its records and parse it respectively with its own methods. If a user wants to delete their account, the authentication system removes it from its records. A user account has no functionality other than storing the data for an account.



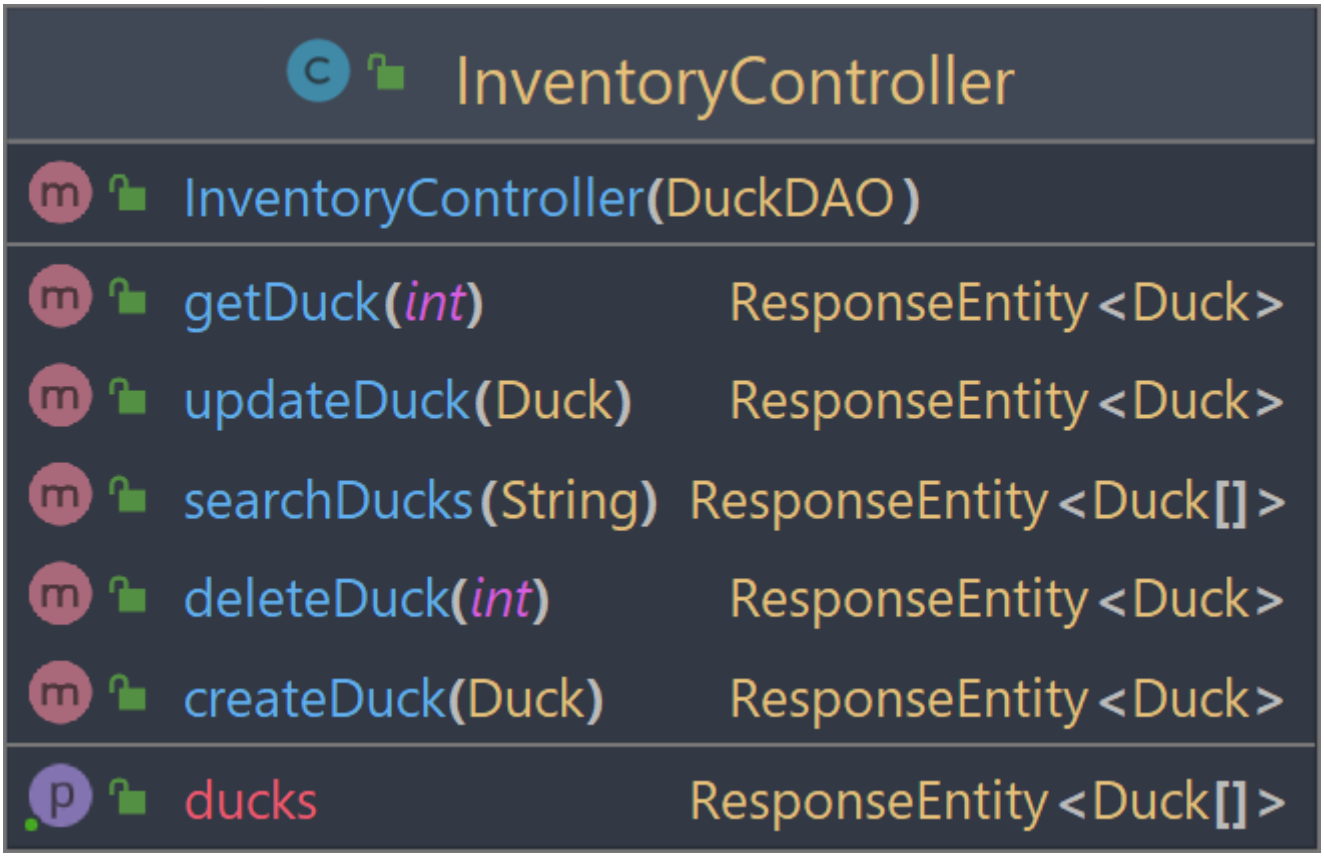
Our shopping cart would most likely serve to benefit from pure fabrication. We need something to handle the product methods. Right now, we would have to add multiple methods to our shopping cart class to gather all the information needed to properly calculate the total of all the items in our cart and display them. These methods have no logic related to a shopping cart so including them in this class would prove to be troublesome. So instead, we should create a checkout class that handles all these calculations making the shopping cart class more cohesive in the process. Not to mention, this code can also be applied in other situations than the checkout, such as showing the total value of all the items in the shopping cart when a user is not on the checkout screen.

Single Responsibility

Single Responsibility is a design principle stating that a class or module should have only one reason to change.

Applications:
As of now, our design makes use of the single responsibility object-oriented design principle by separating our entity objects from our data accessor objects. We use the entities to allow for runtime data persistence, and we use our data accessors to read data from a file, so it can be serialized to an entity. Additionally, our controller classes will only serve one group of endpoints. For example, our inventory controller will only serve endpoints relating to inventory management. Furthermore, if a controller class started to become very lengthy, we could divide the controller class into multiple classes that would encompass all the original endpoints. Finally, our inventory class will serve as an information expert on products, allowing products to be added, removed, edited, and searched based on specific parameters. Separating our responsibilities like this makes our code more readable and easier to work on.

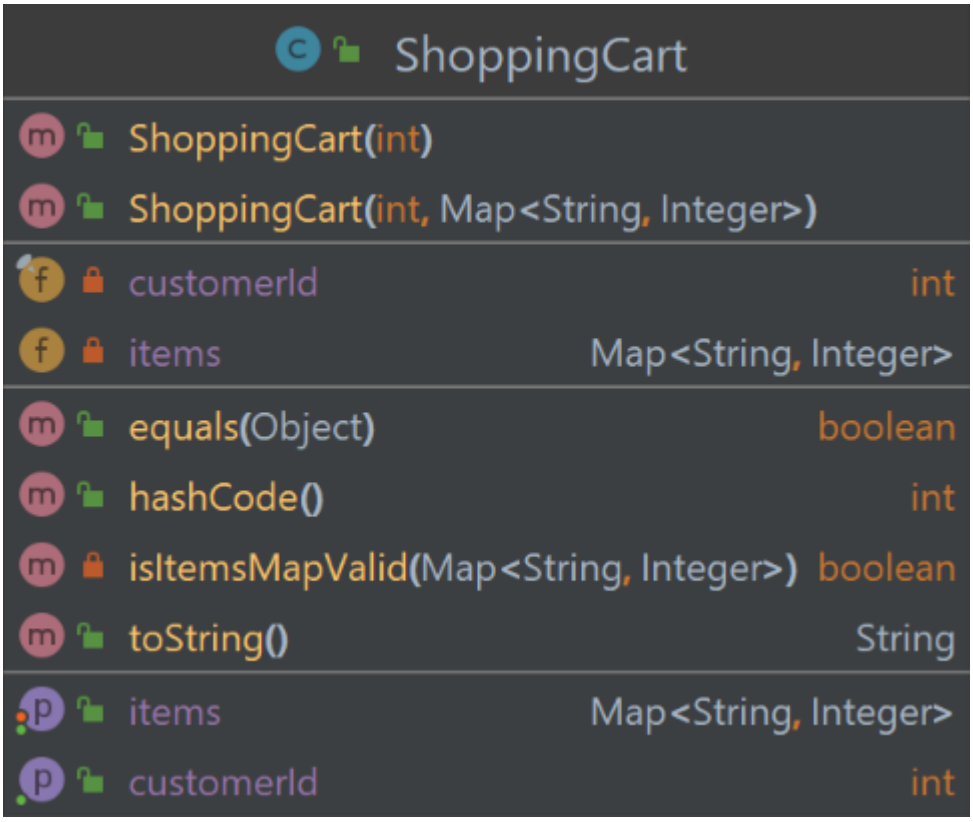
To incorporate the single responsibility object-oriented design principle even more into our design, I suggest that for our duck class that the property attributes of the duck, such as color, size, etc., are stored in a separate DuckProperties class. Then, the duck entity class could serve as an information expert for the duck's properties. Additionally, a customer's profile data could be saved in a profile entity object that is linked to the customer entity object. However, certain information, such as the customer's name, would remain stored in the customer entity object.



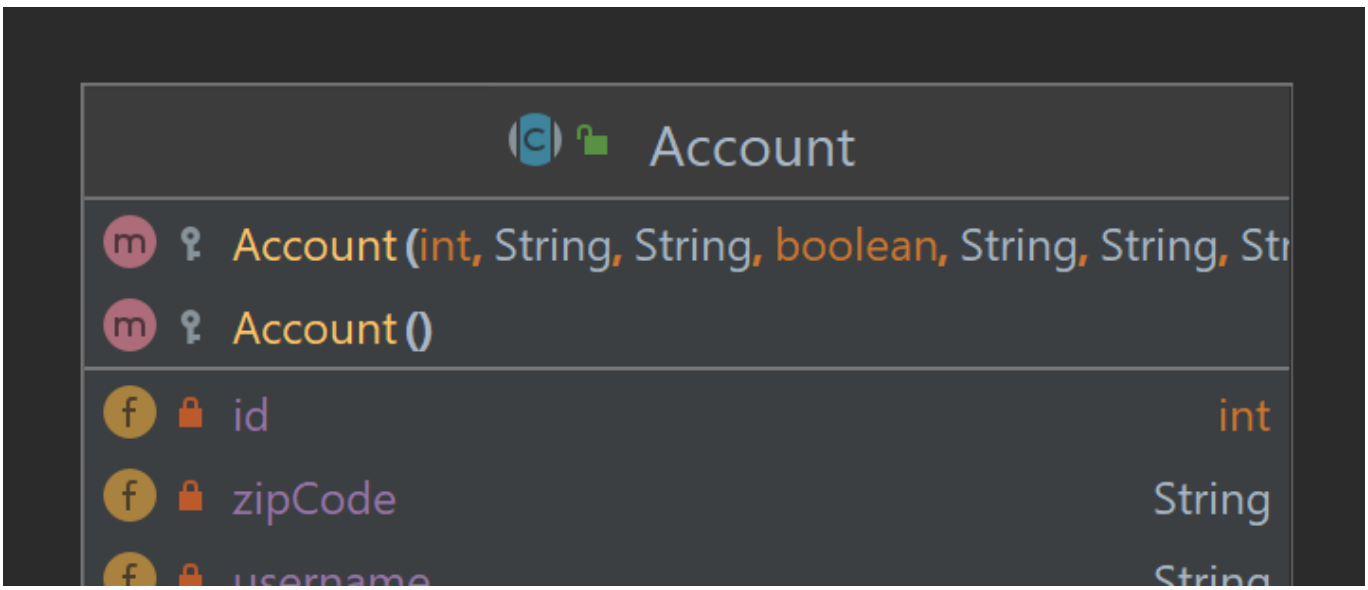
Information Expert

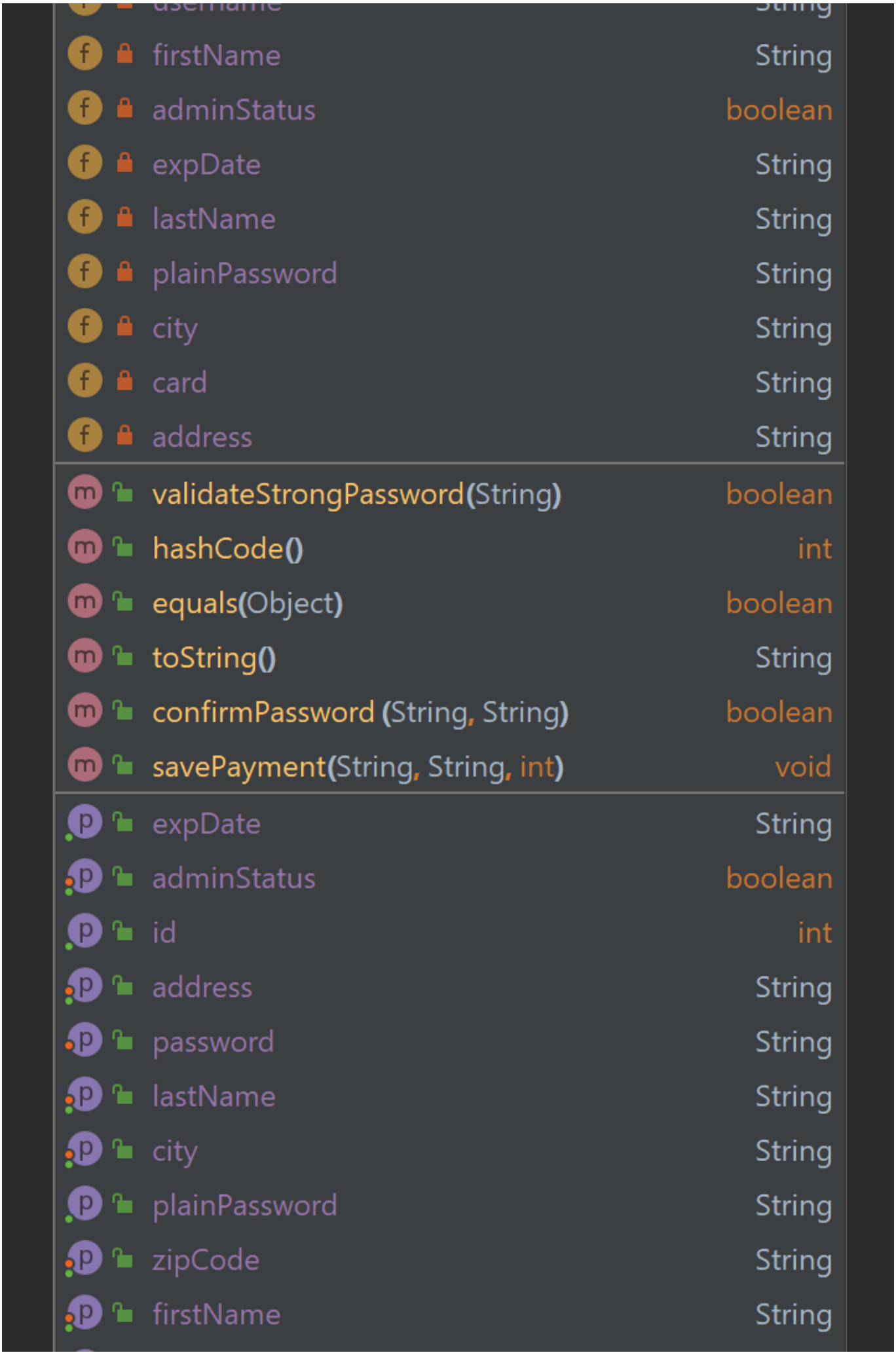
Information Expert is a design principle where responsibility is assigned to an object that has the information needed to complete a task.

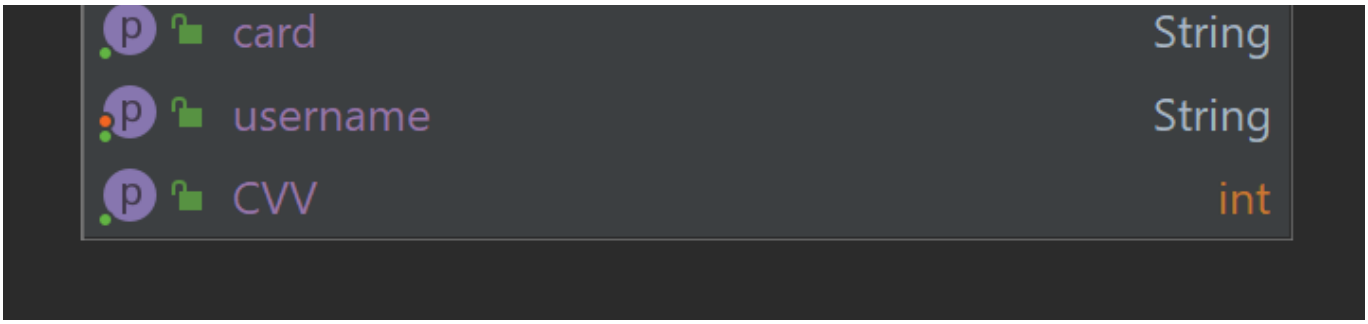
Applications:
The Model Tier UML diagram above adheres to the information expert principle. For example, in the shopping cart class, the shopping cart is given the responsibility of checking out and removing items from the cart. Because the shopping cart object holds the item array, we can assign it the responsibility of adding items, removing items, and editing the quantity of each item in the cart. This keeps the class UML diagram simple and easy to understand without creating complications. For example, if a user wants to remove an item from their cart, the Shopping cart class can check whether the item exists in the cart by searching through its list of items. If the item is found, the Shopping cart class can remove it from the cart, as it holds the necessary information about the item.



Another class that supports the information expert principle is the Accounts class. The Accounts class is responsible for updating the profile information and this is important as the Accounts class holds a profile object. Since the account class holds a profile object, it is appropriate to give the Accounts class the responsibility of updating the profile information.




firstName String
adminStatus boolean
expDate String
lastName String
plainPassword String
city String
card String
address String
validateStrongPassword(String) boolean
hashCode() int
equals(Object) boolean
toString() String
confirmPassword (String, String) boolean
savePayment(String, String, int) void
expDate String
adminStatus boolean
id int
address String
password String
lastName String
city String
plainPassword String
zipCode String
firstName String



Testing

Acceptance Testing

Out of our 64 acceptance critera tests all pass except for 7 of them. However, we expected these 7 to fail because the story card is for a feature that we deemed no longer necessary and removed. Other than those tests, everything else went well. Unlike Sprint 2, we made our acceptance criteria tests more in depth to cover more paths.

Unit Testing and Code Coverage

ducks-api Sessions

ducks-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ducks.api.ducksapi.model		94%		96%	8	142	7	263	5	99	0	9
com.ducks.api.ducksapi.persistence		99%		93%	5	79	1	214	0	42	0	5
com.ducks.api.ducksapi		88%		n/a	1	4	2	7	1	4	0	2
com.ducks.api.ducksapi.controller		100%		100%	0	83	0	245	0	43	0	6
Total	74 of 3,228	97%	8 of 240	96%	14	308	10	729	6	188	0	22

Created with JaCoCo 0.8.7.202105040129

ducks-api > com.ducks.api.ducksapi.persistence Source Files Sessions

com.ducks.api.ducksapi.persistence

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
AccountFileDAO		98%		90%	4	39	1	106	0	18	0	1
DuckFileDAOAbstract		100%		95%	1	24	0	63	0	13	0	1
ShoppingCartFileDAO		100%		100%	0	14	0	41	0	9	0	1
DuckFileDAO		100%		n/a	0	1	0	2	0	1	0	1
CustomDuckFileDAO		100%		n/a	0	1	0	2	0	1	0	1
Total	7 of 1,046	99%	5 of 74	93%	5	79	1	214	0	42	0	5

Created with JaCoCo 0.8.7.202105040129

ducks-api > com.ducks.api.ducksapi.model Source Files Sessions

com.ducks.api.ducksapi.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Duck		92%		94%	3	48	2	84	1	30	0	1
DuckOutfit		93%		100%	1	29	1	44	1	16	0	1
Account		92%		87%	3	33	3	60	2	29	0	1
ShoppingCart		93%		100%	1	16	1	28	1	10	0	1
Price		100%		100%	0	7	0	20	0	5	0	1
Colors		100%		n/a	0	3	0	12	0	3	0	1
Size		100%		n/a	0	3	0	9	0	3	0	1
UserAccount		100%		n/a	0	2	0	4	0	2	0	1
OwnerAccount		100%		n/a	0	1	0	2	0	1	0	1
Total	62 of 1,079	94%	3 of 86	96%	8	142	7	263	5	99	0	9

Created with JaCoCo 0.8.7.202105040129

ducks-api > com.ducks.api.ducksapi.controller Source Files Sessions

com.ducks.api.ducksapi.controller

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CheckoutController		100%		100%	0	18	0	70	0	5	0	1
UserController		100%		100%	0	20	0	61	0	9	0	1
AbstractInventoryController		100%		100%	0	17	0	54	0	7	0	1
ShoppingCartController		100%		100%	0	13	0	43	0	7	0	1
InventoryController		100%		n/a	0	8	0	9	0	8	0	1
CustomizeController		100%		n/a	0	7	0	8	0	7	0	1
Total	0 of 1,060	100%	0 of 80	100%	0	83	0	245	0	43	0	6

Created with JaCoCo 0.8.7.202105040129