Team:
Rohit Gupta
Corin Sandford
Mike Gore

Title:
Marketplace

Question 1. List the features that were implemented (table with ID and title).

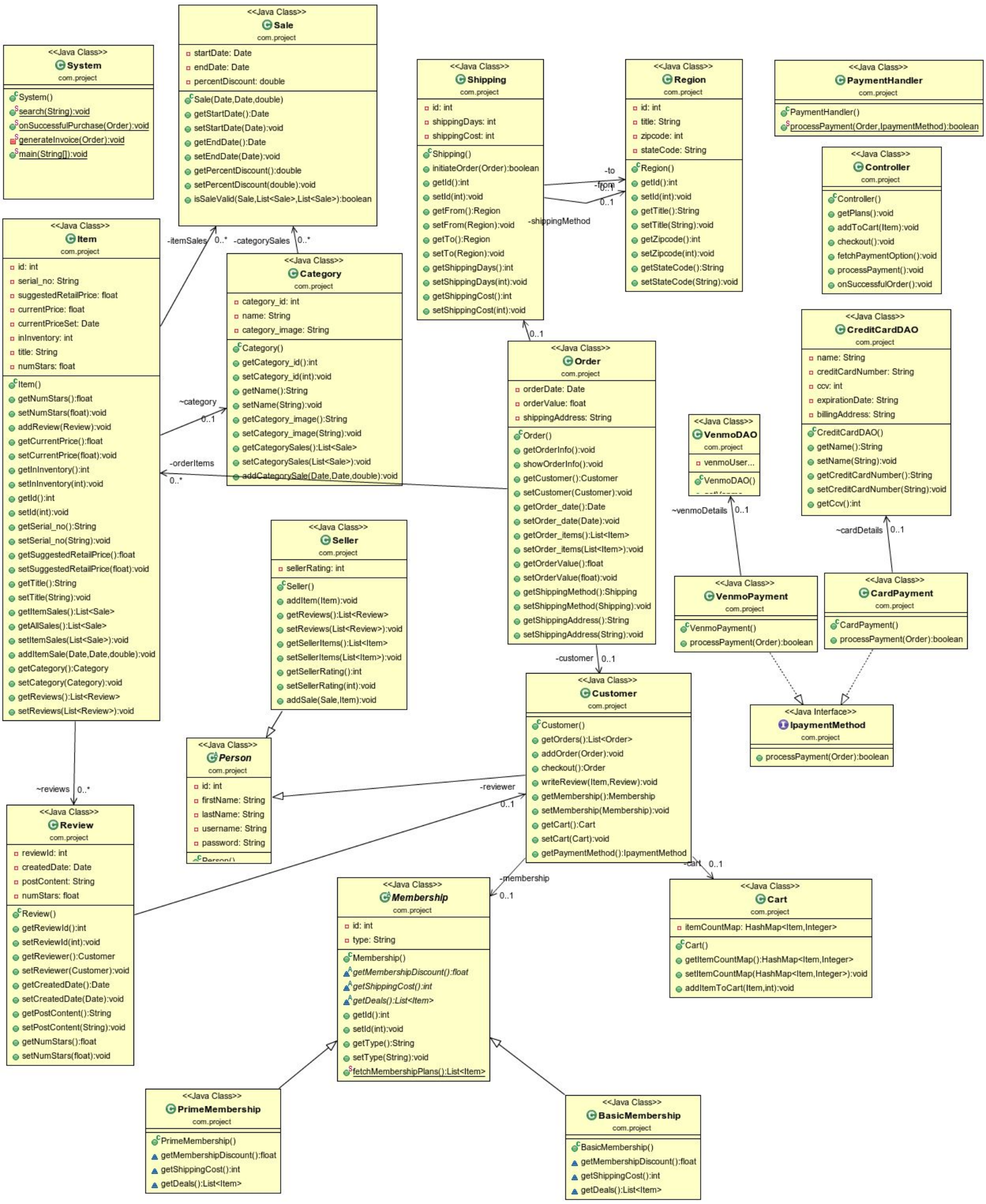
Features Implemented	
ID	Title
UR-01	As a customer, I want to be able to search for an item and view its specifications and details.
UR-02	As a customer, I want to be able to review a product.
UR-03	As a customer, I want to be able to view my purchase history on the website.
UR-06	As a customer, I want to be able to log in to my account.
UR-07	As a customer, I want to be able to make a purchase.
UR-09	As a seller, I want to be able to set a temporary sales price for one of my items.
UR-10	As an administrator, I would like to be able to set temporary sales prices for a category of items on our site.
UR-12	As a seller, I want to be able to access the Seller Interface page to manage my storefront.
UR-13	As a customer, I want to add my credit card information to my account.

Question 2. List the features were not implemented from Part 2 (table with ID and title).

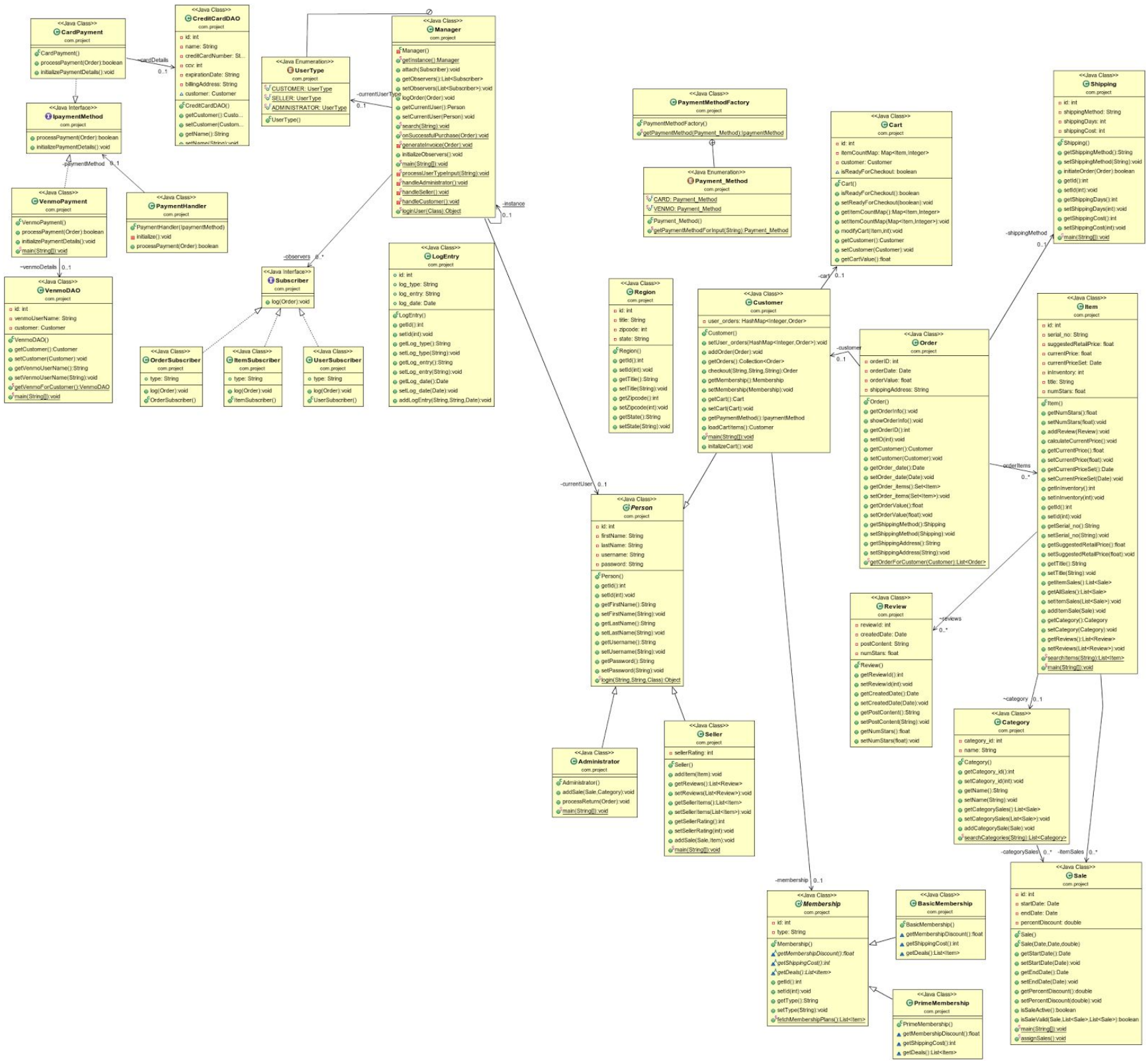
Features not implemented	
ID	Title
UR-04	As a customer, I want to be able to change the membership status on the website from “basic” to “prime” and vice-versa.
UR-05	As a customer, I want to able to track the order I have placed on the website.
UR-08	As a seller, I want to be able to put my product up for sale.
UR-11	As a site administrator, I would like to be able to process returns that users ship back to us.

Question 3. Show your Part 2 class diagram and your final class diagram. What changed? Why? If it did not change much, then discuss how doing the design up front helped in the development.

Part 2 Class Diagram:



Final Class Diagram:



Major changes in the class diagram from part 2:

1: We added utility classes DateUtilities.java, DisplayUtilities.java, Utility.java classes. Their functionality is as follows:

DataUtilities: This class contains a number of helper functions related Date calculations and arithmetic—primarily because we used Java's built-in Date class, many functions of which are now deprecated.

DisplayUtilities: Our project is a command line based project with no user-interface. This class contains functions which handle printing the output of uses cases like view cart, search item to the terminal. The purpose was to keep all of the "print to screen" code in one class.

Utility: This class has helper functions like taking input from user, validating user input etc.

2**:** Added a class DatabaseManager, which is the central point for database related operations. It is a singleton class which has utility retrieve, save functions, along with functions which open and close sessions. The purpose was to manage database session centrally from a single point.

3: We changed the design of the Shipping class by removing Region instances from it, instead we now just have three type of shipping with flat pricing irrespective of regions. This design follows the standard shipping model used by ecommerce companies.

Question 4. Did you make use of any design patterns in the implementation of your final prototype?

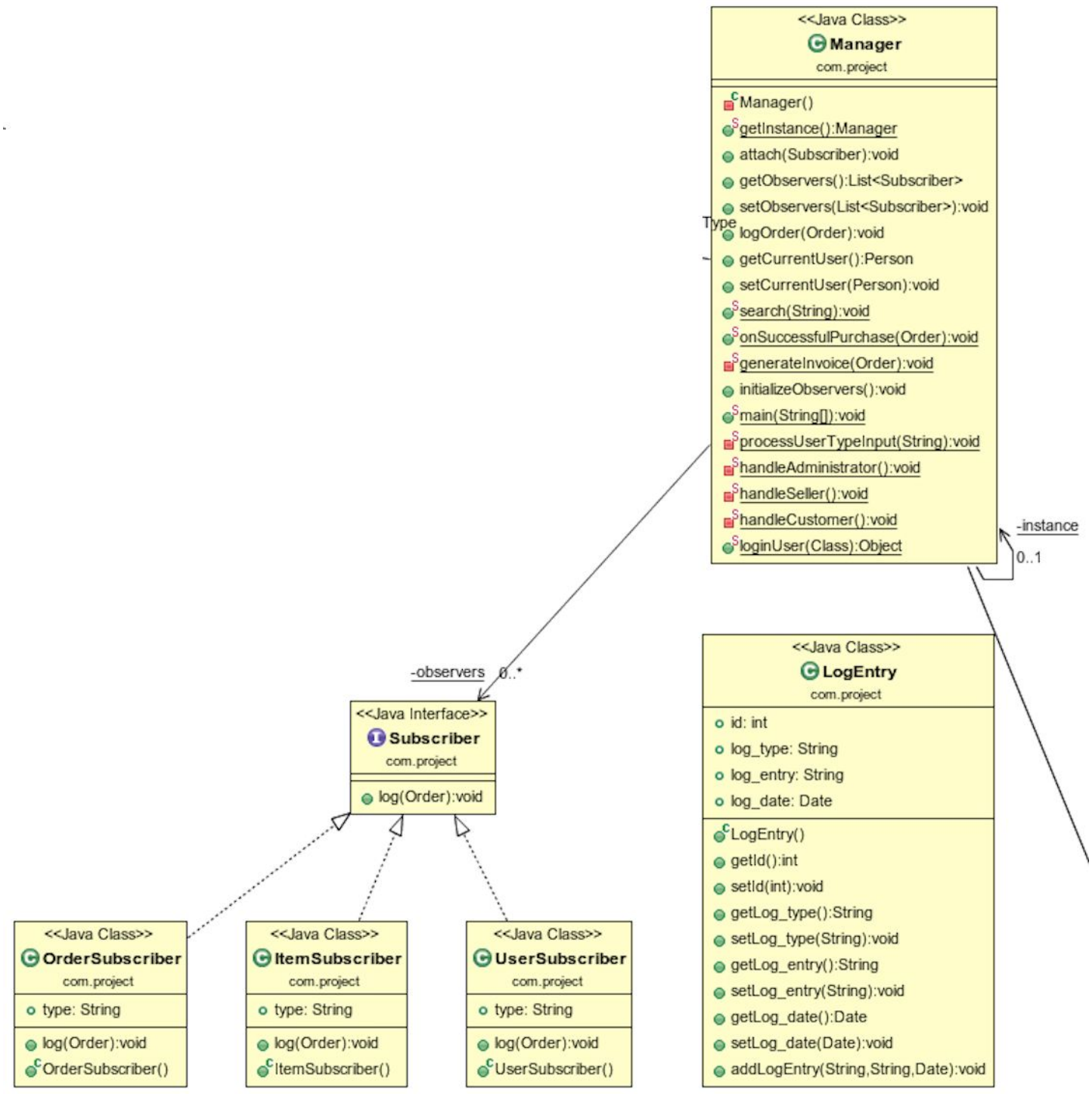
If so, how? Show the classes from your class diagram that implement each design pattern (each design pattern as a separate image in the .PDF).

Design Patterns used

Observer Design Pattern:

We used the observer design pattern to keep logs of activity within our marketplace. Upon creation, our Manager class (the entry point for users) creates three observer objects and attaches them to the Singleton Manager. The three observers keep logs in the database orders that were placed. This includes logging the items that have been ordered, the users that have placed orders, and the orders that have been placed. Each LogEntry stores it's type, a timestamp, and a String "entry" describing the LogEntry. When an order is placed, the static function Manager::logOrder(Order) is called which then calls Subscriber::log(Order) on every observer attached to the Manager. Based on which type of Subscriber log is called upon, a different type of LogEntry is saved in the database. In addition, a function to display logs (DisplayUtilities::displayLog(String)) was implemented. To use this, give the function the type of log to display (User, Item, Order) to view all the LogEntry's of this type.

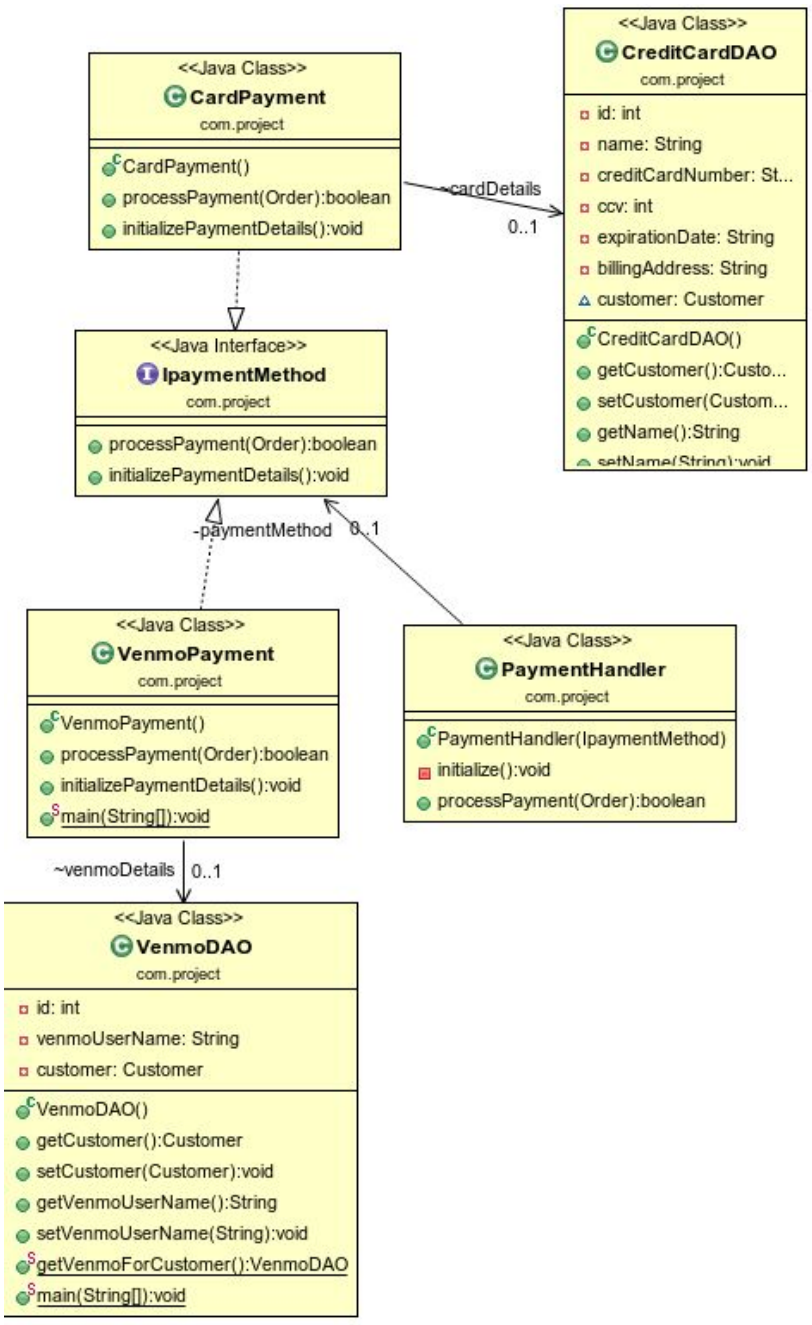
Observer Design Pattern image:



Strategy Design Pattern

We used strategy design pattern to handle the payment in the application. Our application supports two mode of payment: credit card and venmo. Mode of payment is decided on runtime based on user's input. Moreover, overall class diagram follows the open-closed principle meaning that we can add a new mode of payment without modifying any existing code, just by adding a new class. The strategy design comprises of the interface IPaymentMethod and its implementations CardPayment, VenmoPayment. On runtime, based on user's input, we instantiate the correct payment mode and process the payment.

Strategy Design Pattern image:



Question 5. What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?

One of the most important things we've learned from stepping through the process is the importance of starting early on thinking about a design. It was interesting to see how the decisions we'd made earlier influenced the implementation. The DatabaseManager class was a great example of this: by setting this up upfront, before we'd written any database-accessing code, we were able to greatly streamline development and save a lot of time when adding database calls. It was particularly nice for a student project: otherwise, when you're trying to implement something based off another programmer's work, you'd have to think more about each line of code they wrote, rather than just go with the existing methods. The DisplayUtilities we had were also very useful in this regard.

Another thing we learned was the importance of thinking at appropriate levels of abstraction. For example, we discussed making our Sale design more flexible, so we could set sales by not just item and category, but by region, etc. In the end, we just used a very simple Sale class that other classes (like Item) were responsible for storing. This made the Sale class work a little like an interface, without separating it into a concrete implementation. This made it very easy to implement the sales we did use, without creating a lot of scaffolding for Sale types we didn't implement.