Spring | 2019

Technical Report

Team: Team 1

Yuejun Ma,
Haoran Li,
Litong Xiao,
Sriharsha Bandaru

Instructor: Jia Zhang

Enclosed in this document is the technical report of 18653 Software Architecture and Design.

Table of Contents

Introduction:

Uno is an online sales platform for the developers and customers to trade software online. It is an Amazon-like website with a focus on the software domain. People using the platform can either register as a developer or a customer. The registered developers can post new products and set the prices on the platform, which can be viewed by customers. The registered customers can do the purchase on the platform. In addition, the website provides a social network for customers and developers to post comments and ask questions.

Motivation:

- **For developers:** Huge number of software developers in the world (we may be one of them) building software, and most of this useful software is difficult to find for users and developers have difficulties in reaching out to users.
- **For customers:** Higher number of people who are willing to buy and use this software. Users want to be able to find all the software they need on a single platform and buy them legally and conveniently.
- **For team members:** It helps us to be more familiar with the design patterns and implement. Also, the progress and the requirement from the professor and TAs helped us to work on real website development.

Related Works:

To actually work on this project, we looked into some other eCommerce website, like Amazon and eBay. These platforms are much more generic. We learned some features from them. Our platform focuses specifically on software products. As a matter of fact, it's an Amazon-like online store, but only for software trading.

System Design:

1. Architecture view - Monolithic:

1.1. The architecture view is shown in Fig. 1. The users of the platform can interact with the website through the front-end of the website. The languages used for the development of the front-end are HTML, CSS, Javascript, and Scala. The Web Server is the main part of the platform. It is built using Play Framework of Java. Play framework provides an MVC based architecture. Several services including searching, purchasing and fulfillment, delivery, notification, and products management are provided in the Web Server. All the data are stored in the backend MySQL server.

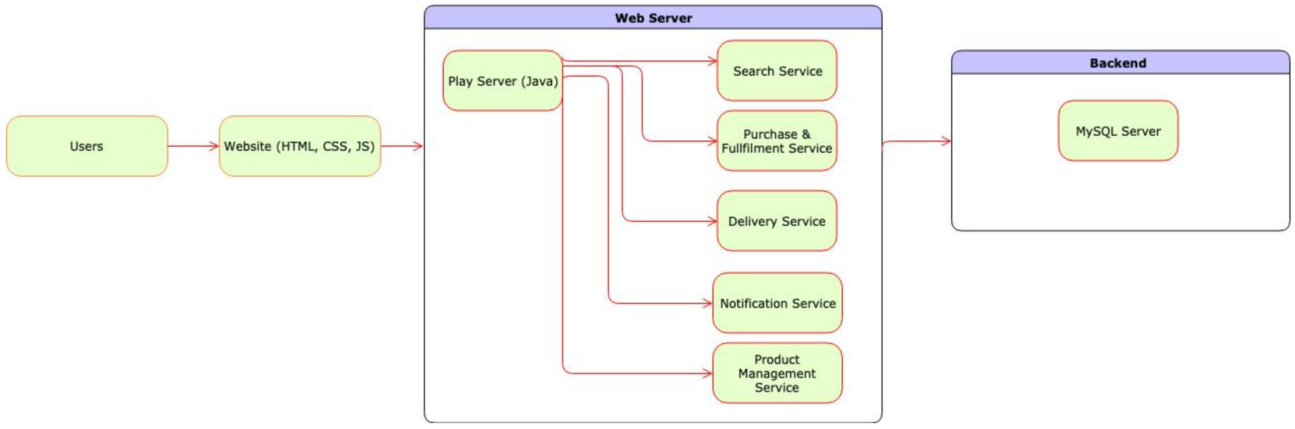

Fig. 1: Architecture View

2. 4 + 1 Views

2.1. Scenario View (Use Case Diagram):

The use case diagram is shown in Fig. 2. There are two main actors in the use cases, the customer and the publisher/developer. The customers can search through the website, choose preferred delivery options, make comments on specific products, view the comments made by others, do the purchasing, modify their own shopping cart, post questions related to the products they purchased, and do the payment. The publishers/developers can do pretty much the same actions as the customers, except that they can manage the products.

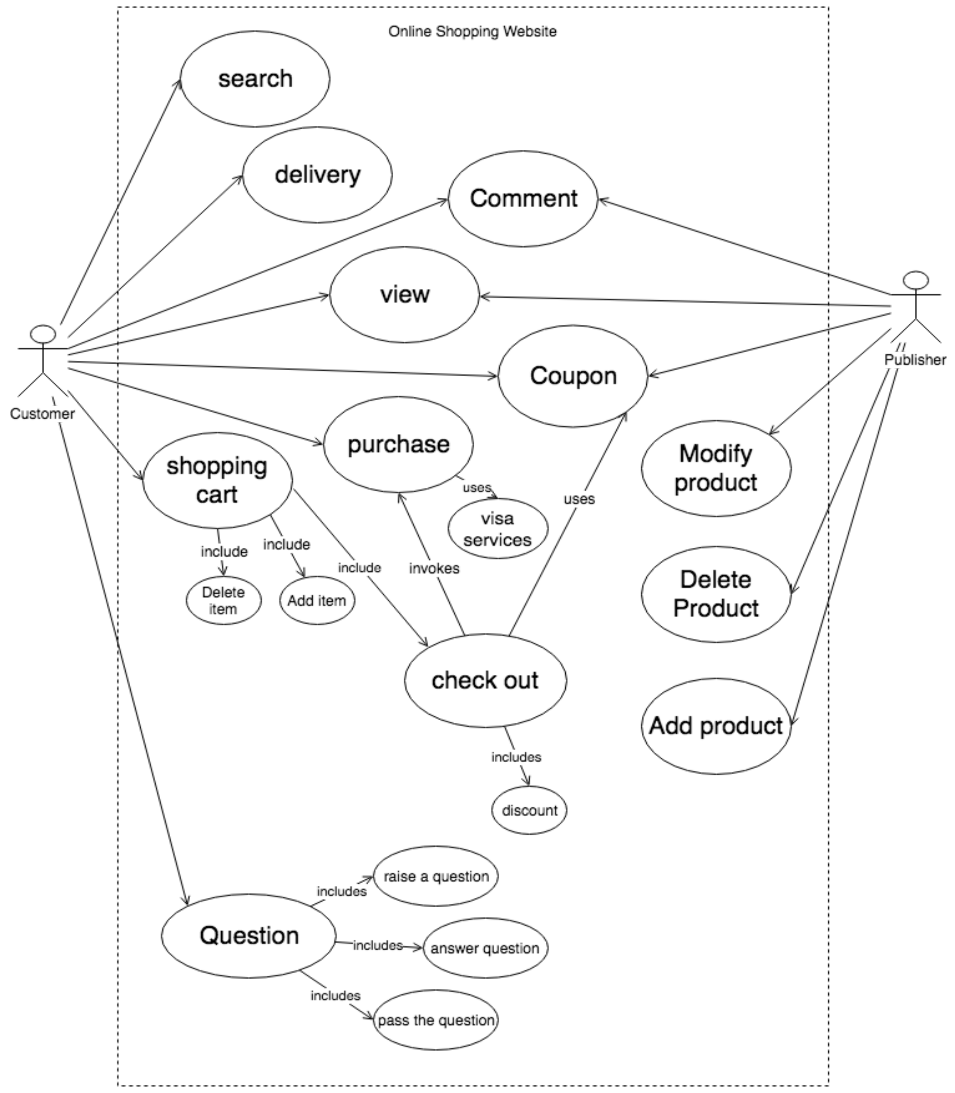

Fig. 2: Use Case

2.2. Logical View (Component Diagram)

The component diagram is shown in Fig. 3. The User component is connected to the website by the dashboard module. The dashboard module connects to the search module, delivery module, and the purchase and fulfillment module. The purchase module also connects to an external component of the payment service. Through the dashboard module, the users can execute the search, choose delivery, and purchase functions. The search module is connected to the product management module, which connects to the publishers/developers. Both the customers and publishers are connected to the notifications system.

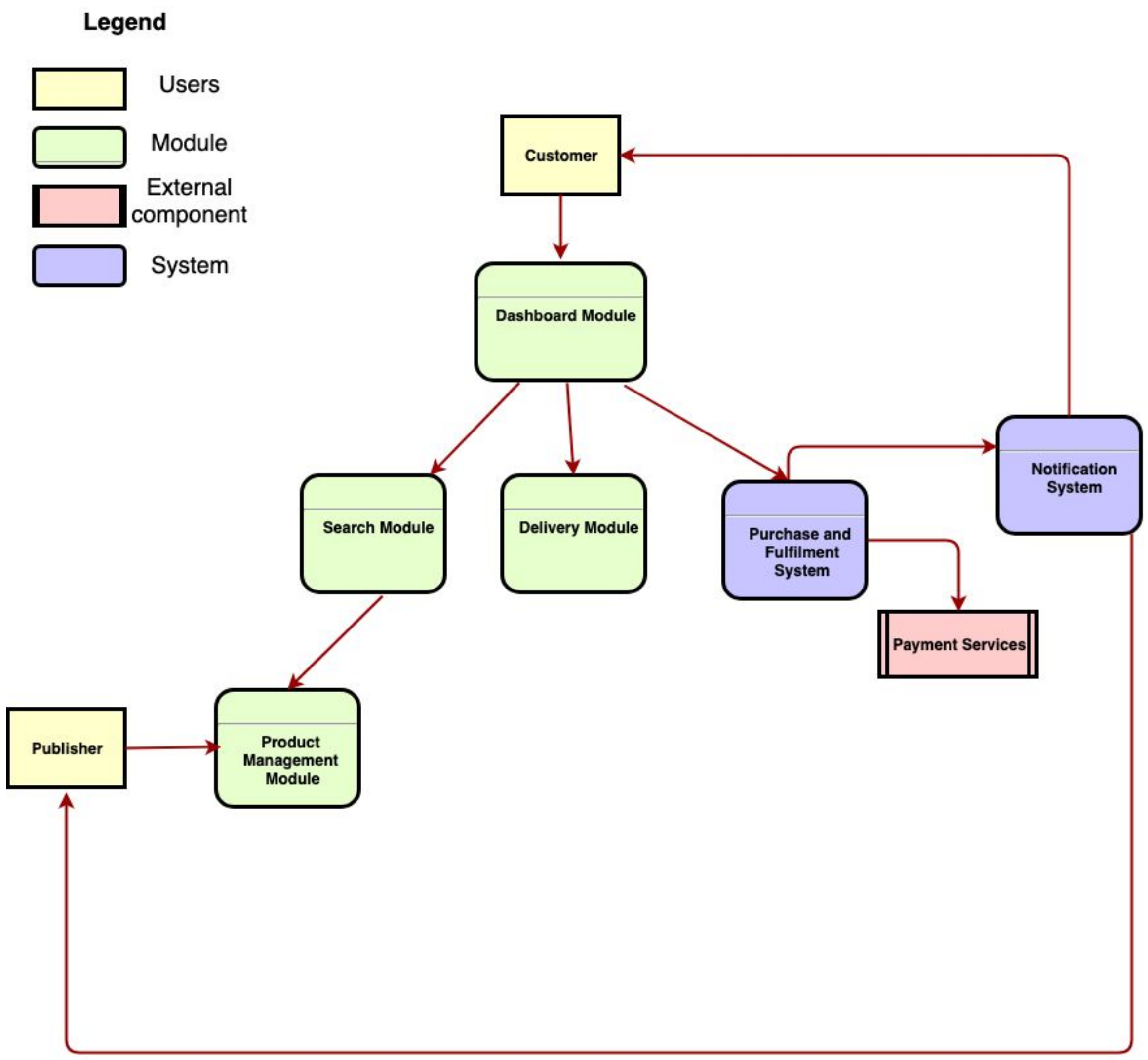

Fig. 3: Component Diagram

2.3. Development View (Class Diagram)

The Class Diagram is shown in Fig. 4. This is the overview class diagram of the whole system. More details are shown in later sections for each separate system.

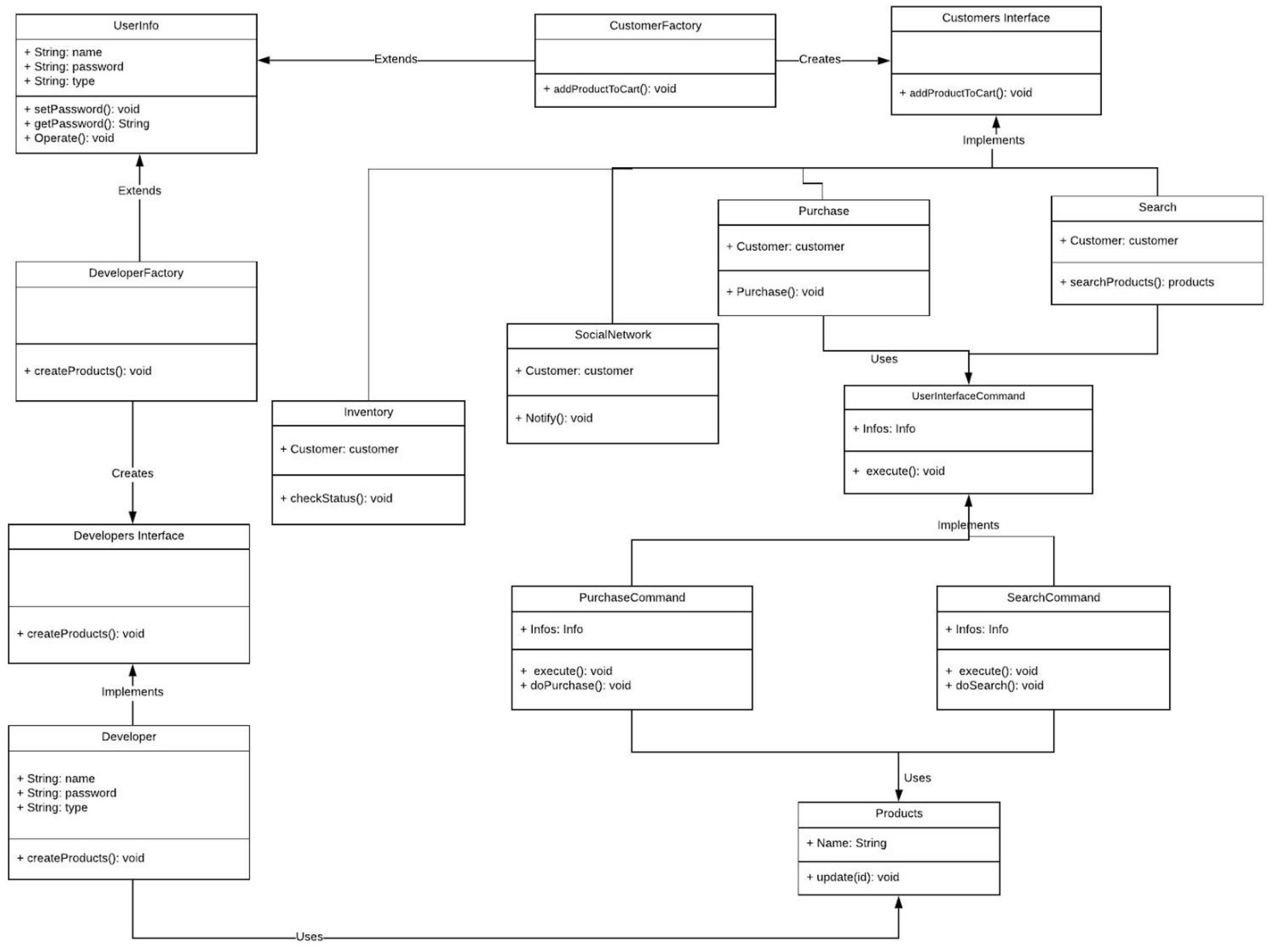

Fig. 4: Class Diagram

2.4. Progress View (Sequence Diagram)

The Sequence Diagram shows how the user can add items, checkout, and complete delivery is shown in Fig. 5. The users first add items to the shopping cart and it will be displayed in the shopping cart. Through the shopping cart, the user can check the items out. The platform will generate a purchase order. A notification will be sent out for the status of the purchase order (created or not). If it is successful, the platform will direct the users to the payment page. The payment status will be reflected in the users indicating whether the payment is processed successfully.

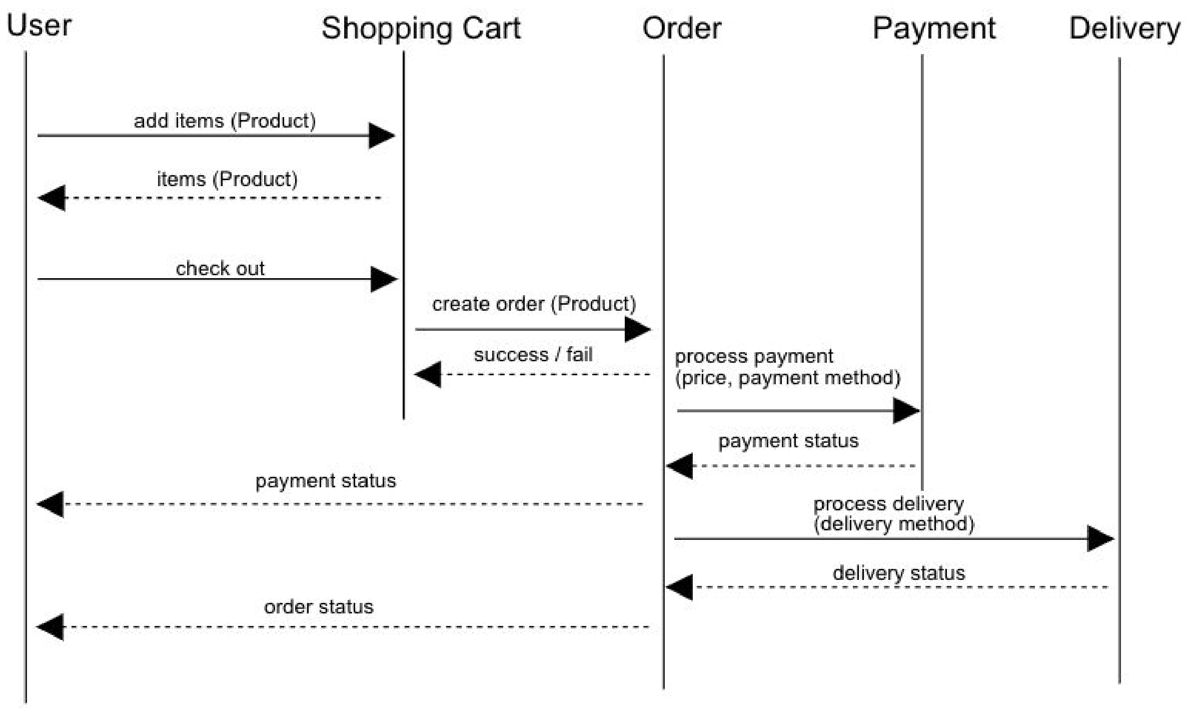

Fig. 5.1: Sequence Diagram 1

The Sequence Diagram showing how the user can search for items is shown in Fig.5.2. User can achieve search function through the search interface. User will first send product name to the search interface. The search interface will perform the search function and then return the information of the required product. After the search interface gets the product information, it simply passes the product information towards the user.

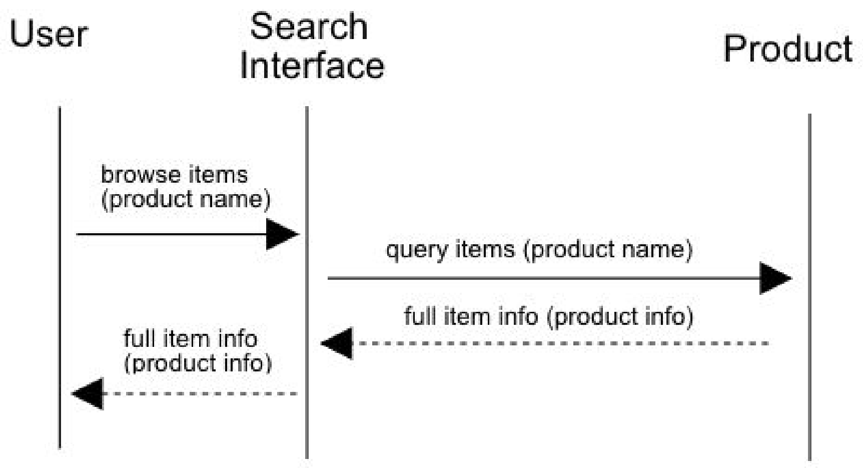

Fig. 5.2: Sequence Diagram 2

The Sequence Diagram showing how a developer can send messages to all customers who bought his product is shown in Fig.5.3. When the developer wants to send messages to the customer, the developer class will firstly send its corresponding message to send center. And send center will create the message object and then pass it to the message manager. Message manager is responsible for all operations on message objects. It will add the message in the desired message receive center and return success or fail. On the other hand, customer can call its message receive center to check all messages from developers.

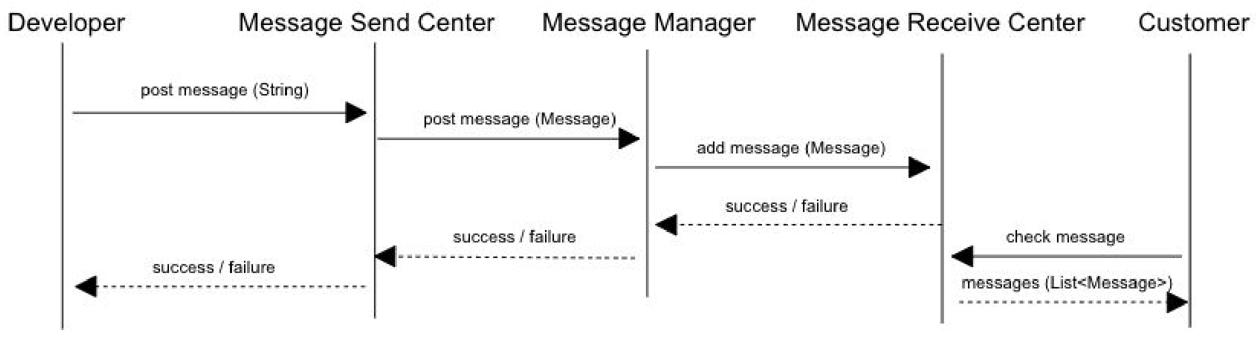

Fig. 5.3: Sequence Diagram 3

The Sequence Diagram showing how customers can post comments on certain products is shown in Fig.5.4. When customers want to post comments, it will first request the comment manager to handle the operation. And comment manager finds the particular product and then add the comment into corresponding *ProductComment* object. Also, all functions will return whether the operation is successful. On the other size, when customers want to check all comments of a certain product, it will directly send the request to corresponding product object, and product object will get all comments from its *ProductComment* object. Finally, all comments will be passed back to the customer.

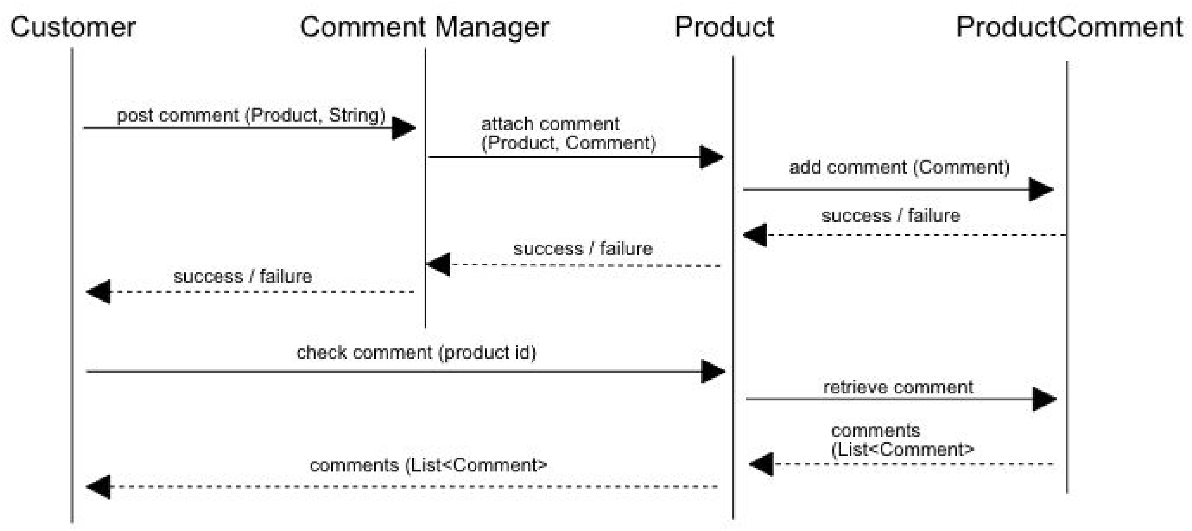

Fig. 5.4: Sequence Diagram 4

The sequence diagram showing how customer can post/view/answer/pass questions is shown in Fig.5.5. Each customer has a question *post* and *receive* center. When a customer wants to post a question, it will use its corresponding question post center to call question manager, which will do all the operations on the question objects. And question manager will attach the question to desired question receive center. Each function will return success or failure. For answering the question, the customer just directly interacts with the question manager, allowing the manager to modify question receive center which is the same as the passing question. *Question manager* is responsible for changing or removing the receiver of each question. As for checking all questions from other customers, customers directly call *question receive center* to get a list of questions.

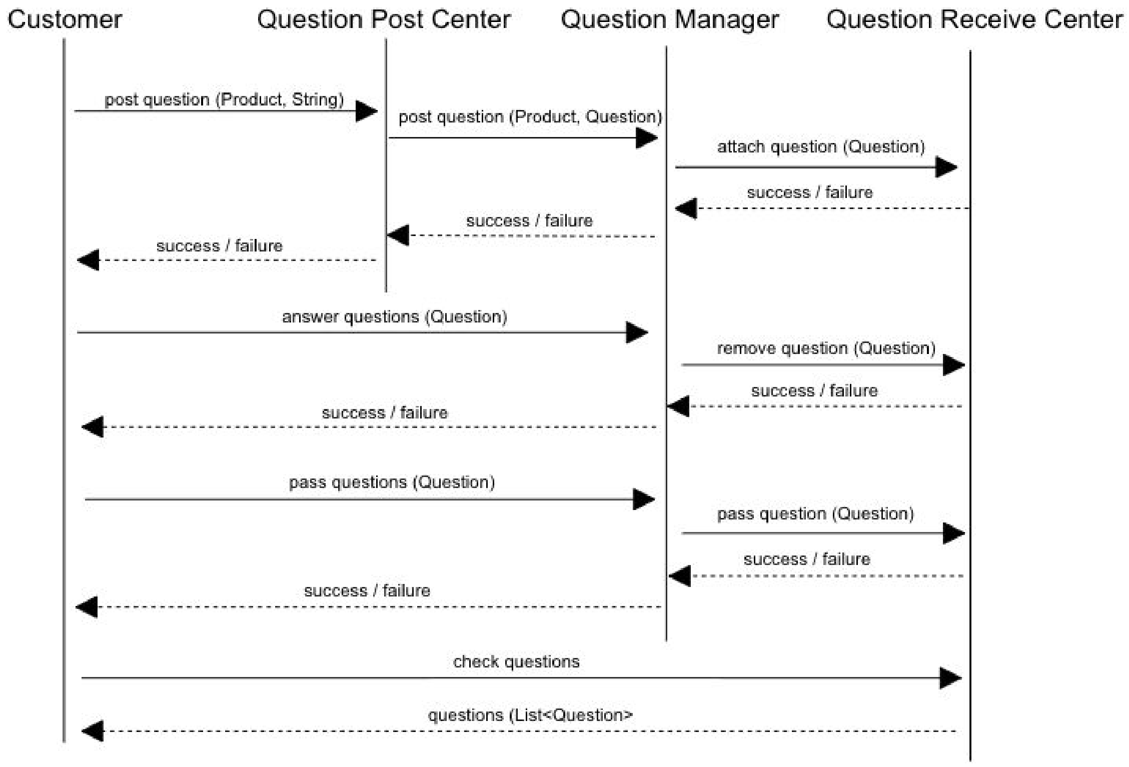

Fig. 5.5: Sequence Diagram 5

The sequence diagram showing how a developer can send coupons to all customers who bought his product is shown in Fig.5.6.When a developer wants to send coupons to the customer, the developer class will firstly call its corresponding coupon *send* center. And *send* center will create the Coupon object and then try to use notifyAll(Coupon) function to add the Coupon to all CouponReceiver (It contains all the customer who bought this product before). Then the Coupon Receiver will return success or fail. On the other hand, customer can call its coupon receive center to check all coupons.

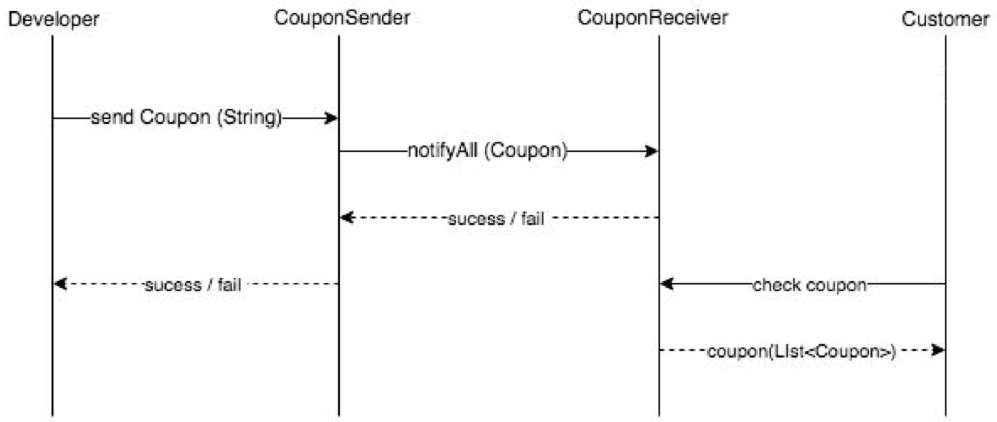

Fig. 5.6 Sequence Diagram 6

System Implementation:

1. User Interface:

Use Cases: Customers will be using the user interface to perform actions such as search, purchase, etc.

Design Pattern: Facade Pattern + Command Pattern

Reason: Facade pattern to hide the internal details of the functions from the user. Command pattern for internally performing the right function (search, purchase, etc.) for the user. The class diagram is shown in Fig. 6.

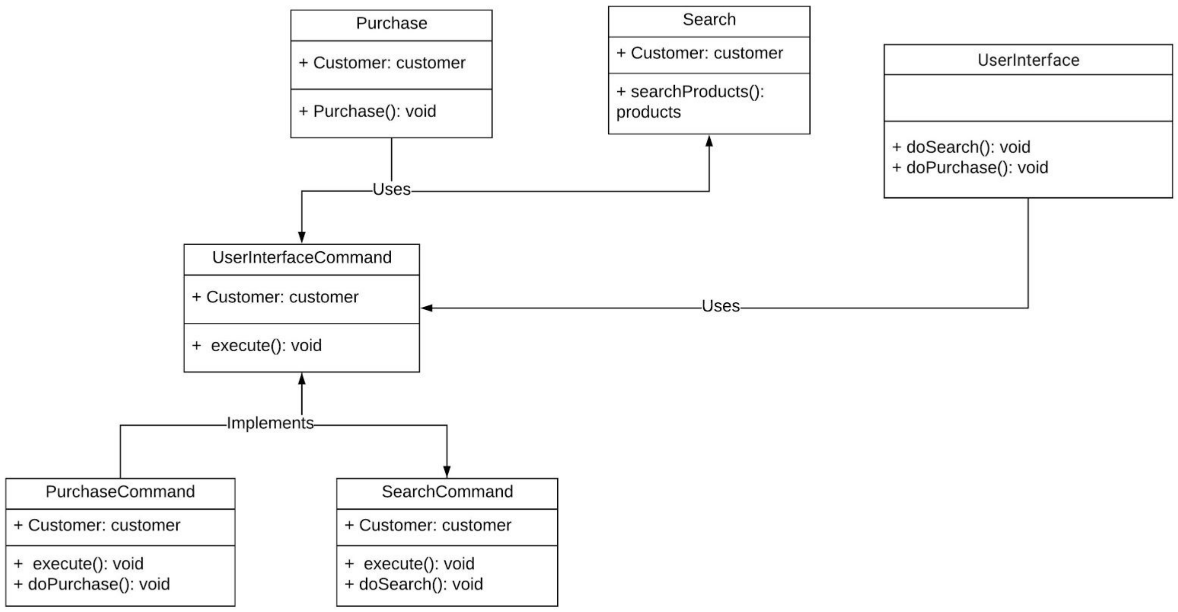

Fig. 6: Class Diagram of User Interface

2. Purchase and Discount System

Use Cases: Purchase and Discount systems helps the customers to view the total price and discounts applied on the products.

Design Pattern: Strategy Pattern + Decorator Pattern + Adapter Pattern

Reason: Strategy pattern used for choosing the proper discount based on the total number of products. Decorator pattern used for applying the discount on the the purchase fee during checkout. The class diagram is shown in Fig. 7.

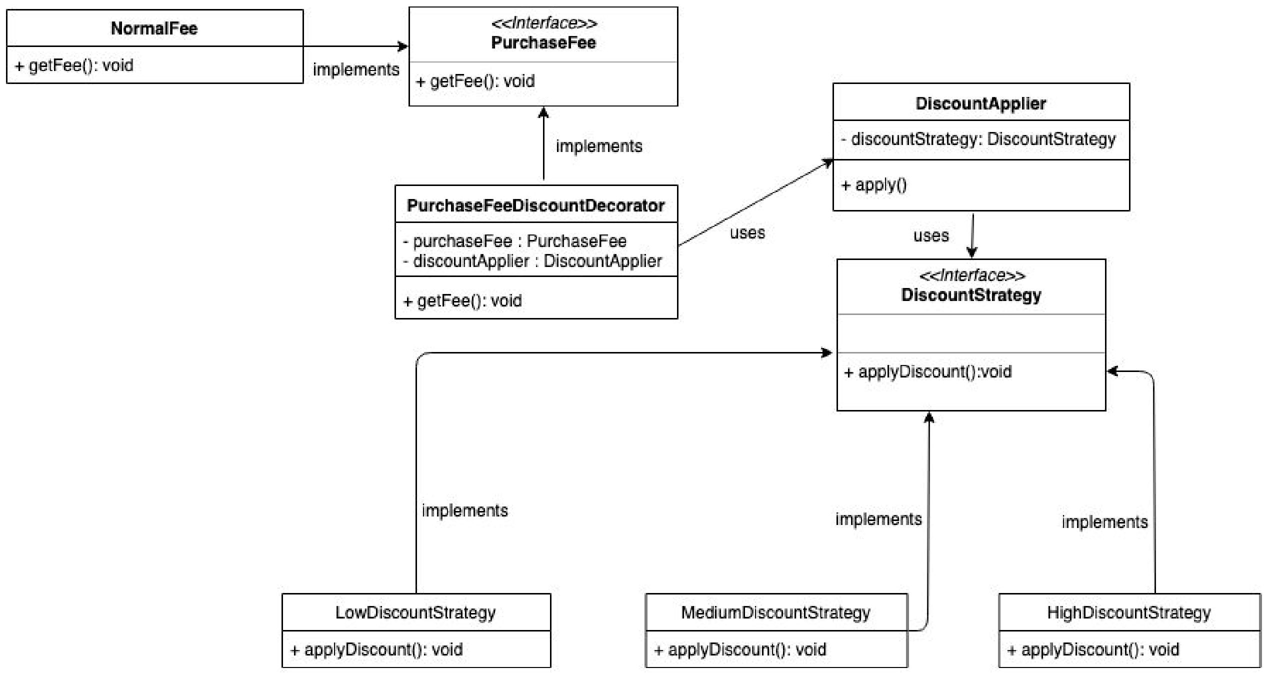

Fig. 7: Class Diagram of Purchase and Discount System

Adapter pattern used for the applied paypal service in the purchase system. Since the PayPal service has a different interface from our system, so the adapter pattern invokes a class which is responsible to transform the data required for the PayPal service. The class diagram is shown in Fig. 8

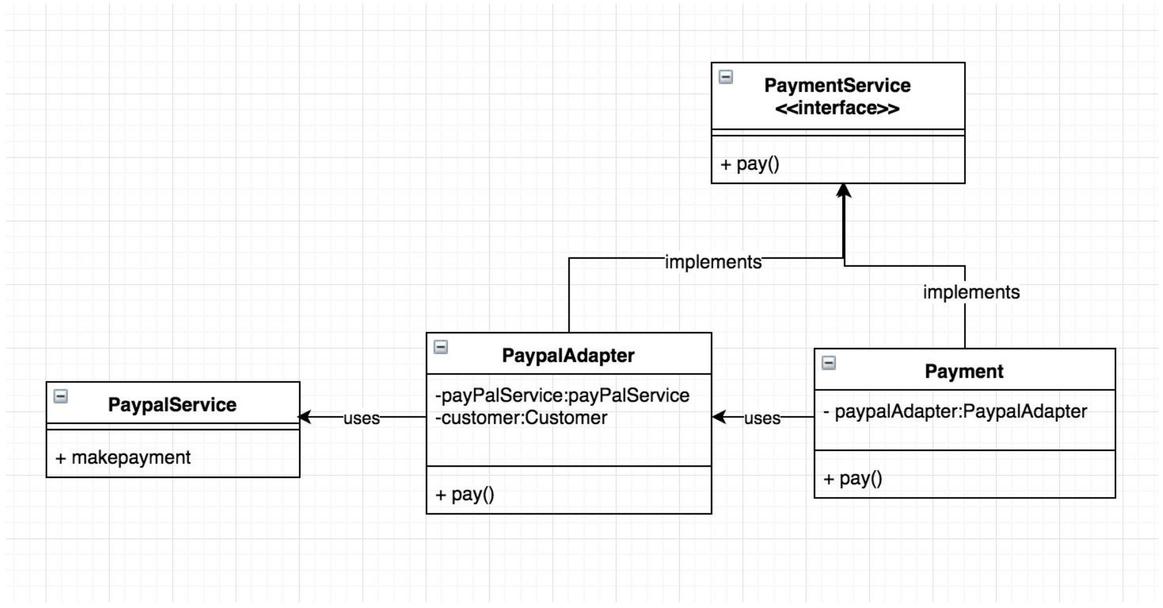

Fig. 8: Class Diagram of Adapter Pattern in Purchase and Discount System

3. Message system

Use Cases: Developers can send messages to all the customers that purchased a particular product before. Customer can view all the messages they receive.

Design Pattern: Mediator Pattern

Each customer has a message receiver center, and developer has a message send center. And all operations about message are done in MessageController. The class diagram is shown in Fig. 9.

Reason: Mediator pattern is used to reduce communication complexity between multiple objects or classes. Since message system involves multiple objects, mediator pattern can ease its complexity. This centralization is useful since it localizes in one place the interactions between objects, which can increase code maintainability. Besides, since we need message controller in Play framework, providing all function in the controller is an easy way for development.

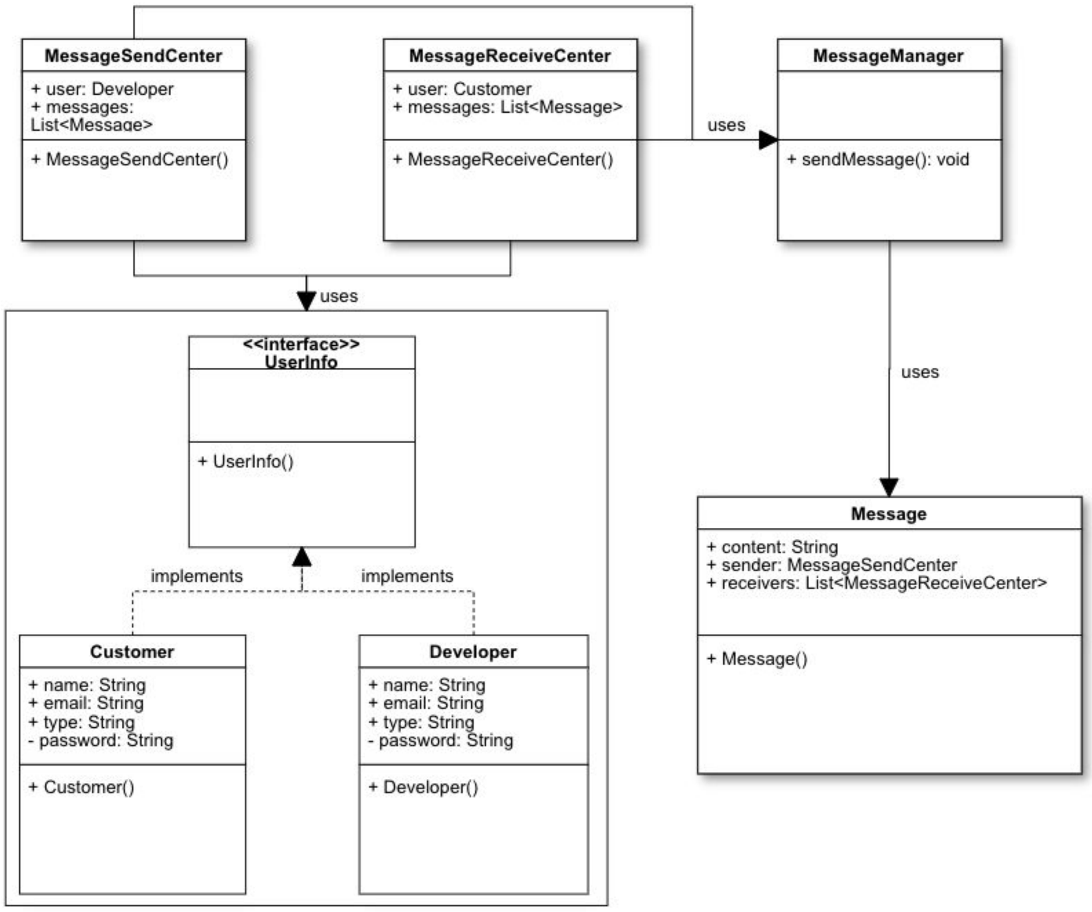

Fig. 9: Class Diagram of Message System

4. Comment system

Use Cases: Customers can make comments on a product. All other customers can view comments of any product. The developers can also view the comments made by others.

Design Pattern: Mediator Pattern

Each product has a special class to manage all comments of this product. Comment Manager will handle all operations about comments. The class diagram is shown in Fig. 10.
Reason: We used mediator pattern again here. The reason is the same as message system: using mediator pattern can decrease the complexity in interaction between multiple objects. This centralization is useful since it localizes in one place the interactions between objects, which can increase code maintainability. Besides, since we need comment controller in Play framework, it is an easy way to provide functions in the comment controller.

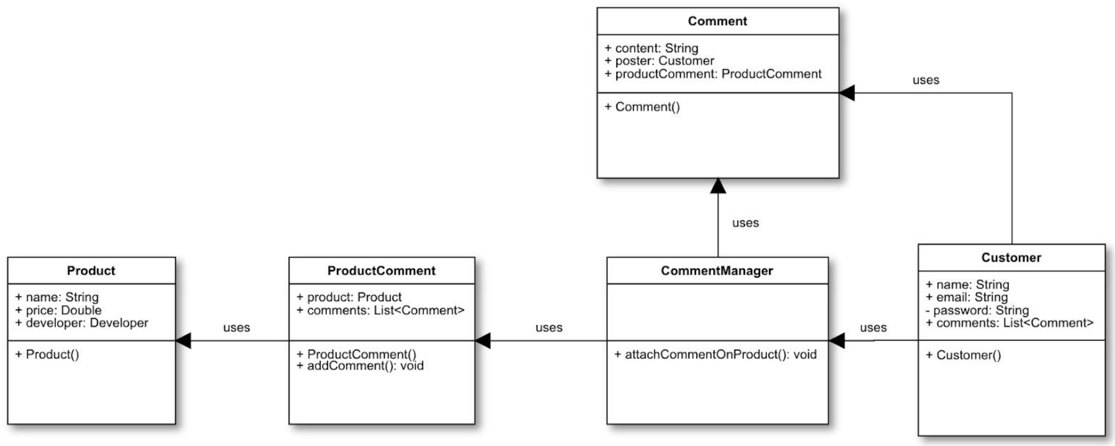

Fig. 10: Class Diagram of Comment System

5. Question system

Use Cases: Customers can post questions on particular product. And a customer who has already bought this product can see this question, they can either answer the question or just pass this question to the next customer who also has bought this product.

Design Pattern: Chain of Responsibility Pattern, Mediator Pattern

Each customer has a question post center and question receiver center used to post and receive questions. And there is a mediator class named question controller who is responsible for all operations done on questions. The question controller also manages the forward function of question, since that the forward of question need interaction with customers. Once the question is answered, the question poster won't be passed to the next customer and the question poster can see answers in their own questions pool. The class diagram is shown in Fig. 11.

Reason: The reason to use chain of responsibility pattern is that we need to create a list of question receiver to be ready to receive questions, which is exactly what chain of responsibility pattern provides. When a customer can't answer someone else's question, he can choose to pass this question to the next customer who also purchased the same product. Using the customer chain provided by chain of responsibility pattern, it is easy to achieve that function. The question will be kept forwarding to the next customer until some customer is able to answer this question.

Besides, we also use mediator pattern here. Because question object is just like comment or message object. Using mediator pattern can reduce the complexity in communication. Also, some functions need interaction with customers such as forwarding question to the next customer. It is an direct and efficient way to implement this kind of functions in the mediator class to boost flexibility.

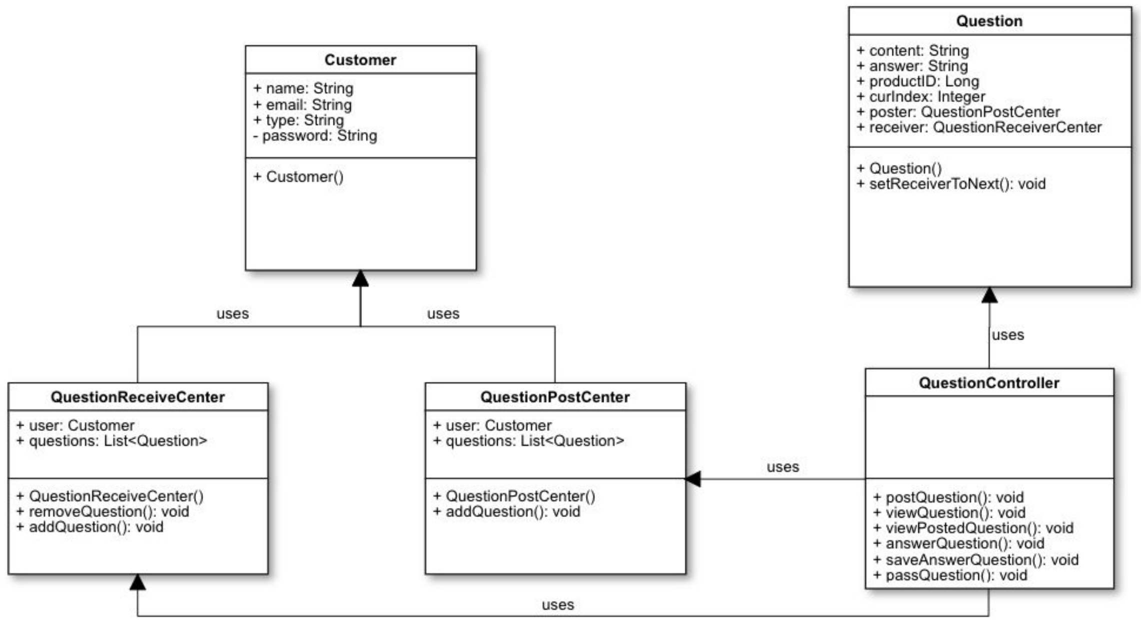

Fig. 11: Class Diagram of Question System

6. Inventory system

Use Cases: Each customer has an inventory to monitor the lifecycle of products they ordered.
Design Pattern: State Pattern

Each product order has a status field that can be set by a status object
Three status: ordered - paid - delivered

- Once the product is added into order from cart, its status becomes "ordered"
- Once customer pay for the order, its status becomes "paid"
- The delivered status need information about logistics, currently not support

Each customer has an inventory used to manage his all product orders. Each product order has a state. There is no initial state. Once the customer adds this product into order, the product order object would be created and then would be added into inventory and it status would be ordered. (Or we can say that the initial status is ordered)

The class diagram is shown in Fig. 12.

Reason: The reason that we used state pattern is that our product order object has a field to represent its current status, and we need to check its status in the inventory. So the behavior of product order class need to change based on its state: it should correctly reflect its current status based on its state field. Using state pattern can easily handle with this problem. Each time the state need to change, we just call a setStatus function in the product order object which is very efficient.

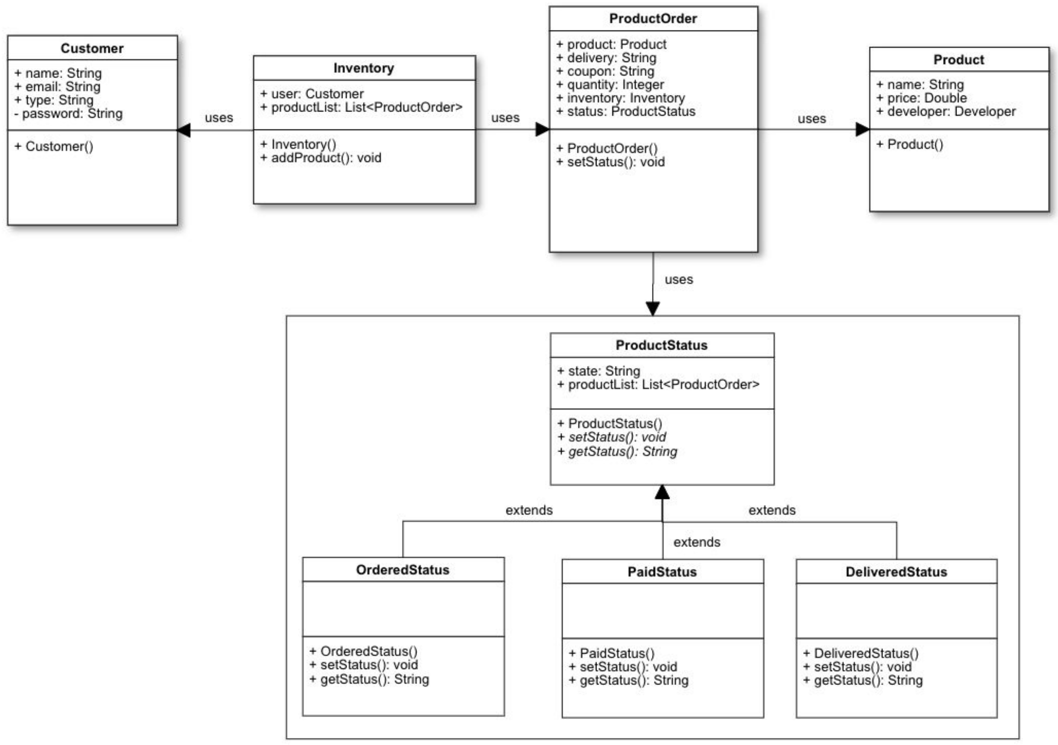

Fig. 12: Class Diagram of Inventory System

7. Coupon System:

Use Cases: Developers can send coupon to all the customer that purchased their product before; they can choose the coupon discount between 20%/40%/60%.

Customers only can use the coupon when they buy the same product again (there is no option for coupon if customer don't have the coupon); In addition, they can view all the coupon they have (including discount and product).

Design Pattern : Observer Pattern + Decorator Pattern

Since the coupon is sent to all the customers who bought the product before, it makes sense to use the observer pattern here by registering the users to the products they purchased. We can then use the Notifyall() function to send the coupons to these customers. It is one to many relationship. The class diagram is shown in Fig. 13.

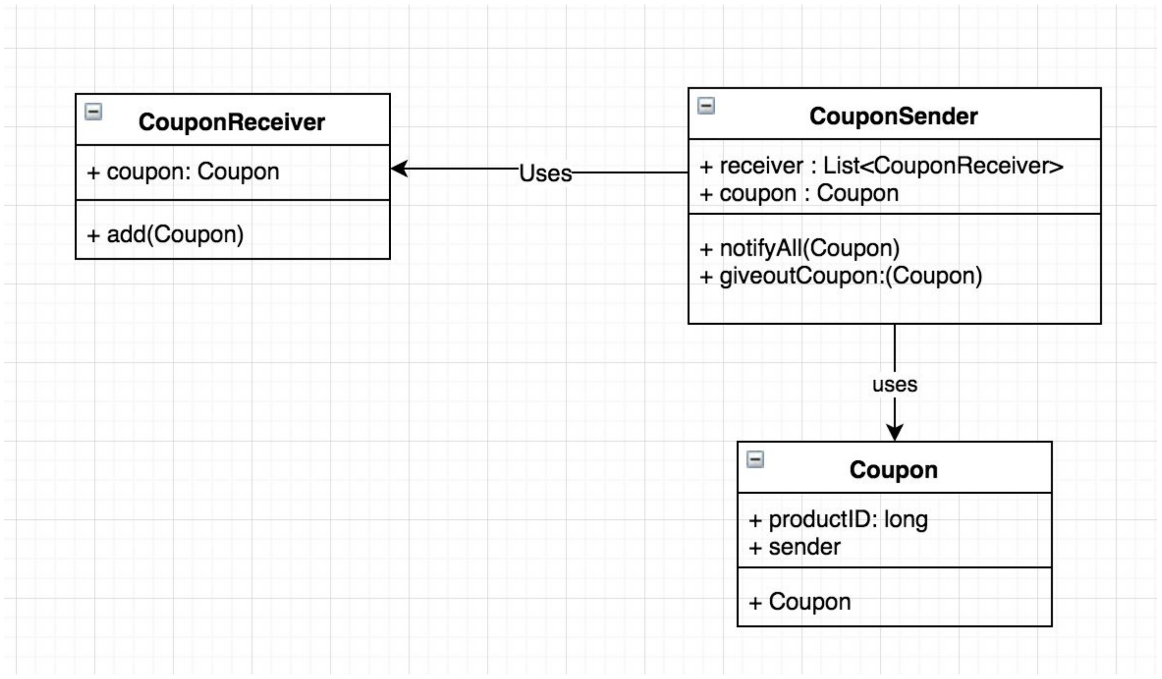

Fig. 13: Class Diagram of Coupon System

CouponReciever is the customers who bought the product before.
In addition, in the purchase system, it uses the same Decorator Pattern (DiscountApplier) to calculate the price after applied the coupon.

Service Oriented Architecture

We provide a Service Oriented Architecture for our system. SOA requires the application to broken down into services. So, we recognized that Authentication, Order Processing, Billing and Payment as the higher level applications and identified the services which will be serving these applications. This can be seen in the Figure 14 below.

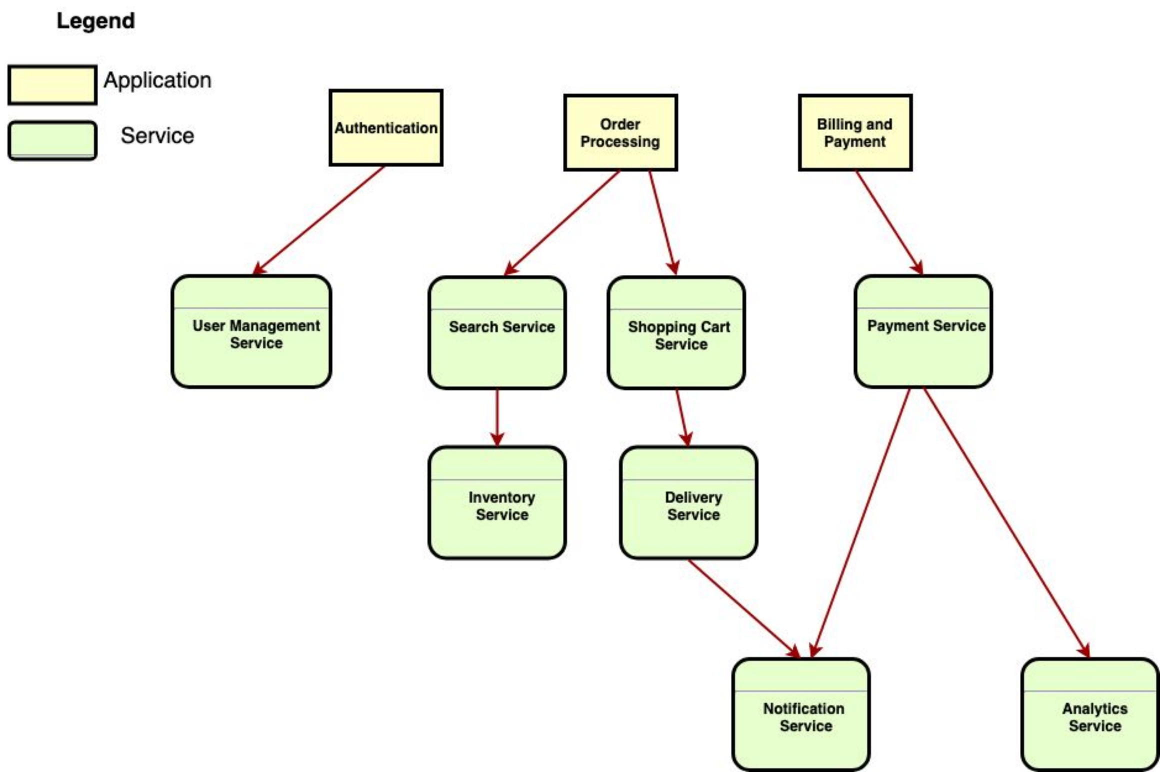

Figure 14: Service Oriented Architecture

Experiments and analysis

We did a manual test on our system. We basically went through each functionalities based on the requirements. We checked each system step by step and confirmed that all the functionalities work correctly and fulfill the requirements. During the test, there are no errors or bugs found.

Conclusions and future work:

Conclusions:

During the development of the software online market, we applied multiple design patterns and architecture principles for the application development. In addition, we became more familiar with the MVC architecture and web development.

Future Work:

- Multiple instances of the web server and database for increased availability of the server
- Improved security for authentication system
- As our application is gaining popularity, we need to use Devops for faster test and deploy. Docker containers can be used to achieve Devops.