OODP Assignment 3
Pruthvik Elemati

- **Intuition**

User Class :
- There are two kinds of users : Customer and Authenticator.
- Since users have some common attributes like firstName, LastName and other login details, we define a common class for them titled "User".
- **Customer** and **Authenticator** classes inherit from **User** and define their own attributes and methods based on their business requirements.

Customer Class :
- Inherits from User
- Customers should have access to a Cart wherein he/she can store can add items.
- Customers should be able to add or remove items from the Cart.
- Each Customer should also have a payment history about the persons previous purchases.
- If items are added into cart and they are purchased, the Cart should be empty and purchase_history has to be updated.
- If no purchase is made, then items have to remain in the customer's cart unless he or she removes them.
-

Administrator Class :
- Inherits from User
- Authenticator should be able to update the details of each item.
- He/ She should also be able to add other authenticators. So, he/She should have an option to create new users.

Item Class :
- Each item should have **type, description** and **price**.
- Along with above attributes, each item should have a **"Count"** attribute which decreases if a customer adds an item to cart. If an item is removed from Cart, the Count attribute should increase.

Cart Class :

- Each Customer has his/her own cart.
- The Cart class should contain a list of items a person added.
- If a person adds or removes items, the Cart should be updated.
- Changes to cart should also reflect changes to item.

Purchase Class :

- Each customer should be able to purchase items in his cart.
- Once a purchase is made, items have to be removed from the customer's cart.
- Each purchase should also be recorded in the purchase_history of the customer.

Store Class :

- The `Store` class acts as a central entity to coordinate interactions between users, items, purchases, and authenticators within the system.

Architectural Patterns to be Used :

1. Front Controller Pattern
2. Authorization Pattern

Design Patterns to be Used :

1. Factory
2. Command
3. Template

Other Design Patterns used :

Using singleton design pattern

- We have a storage class which serves as our main class. There are many classes which operate on the attributes of storage class. So it is important to operate on the same instance.
- To maintain a single instance of “Store” class across all implementations, we use the **Singleton design pattern**.

- The Singleton design pattern is used when you want to ensure that only a single instance of a class exists throughout your application. It provides a globally accessible point of access to that instance.

Front Controller Pattern :
It is used to centralize the handling of all incoming requests. It should act as a single entry point. In the code, there are many requests sent to server by client and previously they were all called using ItemManagementImpl, AuthenticationImpl and AdministratorManagementImpl. Now, we define a **FrontController** class in the server folder that will be able to handle all the requests.

The new FrontController class handles all incoming requests and dispatches them to appropriate handlers. The methods of authentication, item management and administrator management are all centrally handled by it. This helped in better organizing the server side logic.

So, we make changes in serverApp.java and clientApp.java accordingly. The client need to communicate with different classes, instead it can solely communicate with the FrontController.

The FrontController defined now has the following methods :

1. `public String processRequest(String username, String password)`
2. `public boolean registerUser(String userID, String firstName, String lastName, String username, String password, String role)`

3. `public void addItem(Item item) throws RemoteException {`

4. `public void removeItem(String itemId) throws RemoteException {`

5. `public void updateItem(String itemId, String key, String value)`
6. `public Item findItemById(String itemId)`
7. `public List<Item> getAllItems()`

8. `public void addAdministrator(Administrator administrator)`

9. `public void removeAdministrator(String username)`
10. `public Administrator getAdministrator(String username)`

11. `public List<Administrator> getAdministrators()`
12. `editCartRequest(String commandName)`

13. `processPayment(double amount, String paymentType)`

Authorization Pattern :

We have two types of users : customer and administrator. So, we start off by defining a user class, which is inherited by customer and Administrator. Authentication is implemented such that when a user gives his username and password, it either returns “no_user_found” or the role of the user (i.e customer or administrator.). If a customer or administrator is returned, then we show the respective options based on their role.

If “no_user_found” is returned, then he can re-enter login details or register as a new customer. So, an administrator account cannot act as a customer and vice-versa.

It is also important to note that a user cannot simply register himself as an administrator. Administrators can be added by only other administrators.

The above is implemented by AuthenticationImpl.java in the server folder. The details are sent as input from client app to frontcontroller, from where methods of AuthenticationImpl.java are called.

Command Design Pattern :
Command design pattern encapsulates a request as an object. This object can be passed around and executed at a later time.

In order to implement command design pattern we need to define the following :

1. Command Interface
2. Concrete Command Classes

Command Interface is defined in the Common folder along with Concrete Command classes. These requests are passed using FrontController which handles all the requests.

A good application for this is the functionality of Cart. In the context of online shopping, adding or removing items from cart could be encapsulated as commands. By doing this, it not only gives us an opportunity to extend functionality of cart without changing previous code, but also facilitates undo and redo functionality. For example, if a user wants to remove an item from cart or add an item again, this design pattern is ideal.

Factory Design Pattern :

It is a design pattern that allows us to create objects without having to specify their exact classes. Instead of using a constructor, we delegate it to a factory class or method.

An application for it is creating user objects based on role. There are two types of users : customers and administrators. But since administrators cannot be created, we need to loo at other functionalities.

An example functionality could be using multiple payment methods. A payment can be done using : debit card, credit card or paypal.

So, that functionality can be implemented using the Factory design pattern.

We first create a payment interface, and then different payment classes like CreditCardPayment and DebitCardPayment. We then create payment method factories and finally a method to process payment.

PaymentProcessor class uses a factory to create instances of payment methods. Depending on the type of factory passed to PaymentProcessor, it can create different types of payment methods without knowing their concrete classes. This allows for flexibility and easy extension when adding new types of payment methods to the store.

Template Design Pattern :

The Template Method design pattern is a behavioral design pattern that defines the skeleton of an algorithm in a superclass. This superclass allows subclasses to override specific steps of the algorithm without changing its overall structure.

We have an algorithm in superclass, which can be overridden in subclasses. We can come up with one such functionality if we assume that there exists a purchasingProcess which can select items, calculate total and make payment. But lets say that customers and administrators have different methods to implement this. Based on this assumption, if customers and administrators have different methods for implementing payment process, then we can define a common workflow for creating payments, and then create abstract subclasses for Customer and Administrator.

So, in the project :

- `PurchaseProcessor` is the abstract class defining the template method `processPurchase()` which encapsulates the common workflow for processing purchases.
- `CustomerPurchaseProcessor` and `AdministratorPurchaseProcessor` are concrete subclasses that extend `PurchaseProcessor` and provide implementations for the abstract methods according to the specific requirements of customers and administrators.

Domain Model :

Class Diagram :

1. **User :**

Attributes :

1. UserID
2. Firstname
3. Lastname
4. Username
5. Password
6. Role

Methods :

1. Getusername
2. Getpassword
3. getrole

Relationships :

1. it's inherited by Customer and Administrator.

—--

2. **Customer :**

Attributes :

1. cart: Cart
2. purchaseHistory: List<Purchase>

Methods :

1. getCart()
2. setCart()
3. getPurchaseHistory()
4. setPurchaseHistory()
5. addItemtoCart()
6. removeItemFromCart()
7. makePurchase()

Relationships :

1. Inherits from User (inheritance)
2. Has-one relationship with Cart (**Composition**)
3. Has one-many relationship with Purchase (**Composition**)

—---

3. **Administrator :**

Attributes :

1. —-

Methods :

1. updateItemDetails(Item item)
2. addAuthenticator(Authenticator authenticator)

Relationships :

1. Inherits from User (inheritance)

—---

4. **Cart :**

Attributes :

1. items: List<Item>

Methods :

1. addItem(Item item)
2. removeItem(Item item)
3. getitems()

Relationships :

1. Has-one relationship with Customer
2. Has one-many relationship with Item

—---

5. **Item :**

Attributes :

1. ItemID
2. Type
3. Description
4. Price
5. Count

Methods :

1. getType()
2. getItemID()
3. setType()
4. getDescription()
5. setDescription()
6. getPrice()
7. Increase_count()
8. get_count()
9. setCount()

Relationships :

1. Many to One relationship with Cart
2. Has Many to One Relationship with Store

—---

6. **Purchase :**

Attributes :

1. Purchased_Items

Methods :

1. Purchased bill

Relationships :

1. Many to One Relationship with Customer

—---

7. **Store :**

Attributes :

1. Customers
2. Items
3. Administrators

Methods :

Admin Management :

1. addAdministrator (administrator)
2. removeAdministrator()
3. getAdministrators()
4. addCustomer()
5. removeCustomer()

Item Management :

6. addItem()
7. removeItem()
8. updateItem()
9. getItem()
10. findItemByID()

Relationships :

1. `Customer`: A one-to-many relationship, where one `Store` can have many `Customer` objects.
2. `Administrator`: A one-to-many relationship, where one `Store` can have many `Administrator` objects.
3. `Item`: A one-to-many relationship, where one `Store` can have many `Item` objects.

—---

8. FrontController

Attributes :

1. AuthenticationManager
2. ItemManager
3. AdministratorManager
4. commandMap
5. PaymentMethodFactory

Methods :

1. `processRequest()`
2. `registerUser()`
3. `addItem(Item item)`
4. `removeItem(String itemId)`
5. `updateItem(String itemId, String key, String value)`
6. `findItemById(String itemId)`
7. `getAllItems()`
8. `addAdministrator(Administrator administrator)`
9. `removeAdministrator(String username)`
10. `getAdministrator(String username)`
11. `getAdministrators()`
12. `editCartRequest(String commandName)`
13. `processPayment(double amount, String paymentType)`

```
Relationships:
   1. Association with AuthenticationImpl
```

2. `Association with AdministratorManagementImpl`
3. `Association with ItemManagementImpl`

—---

9. AddItemToCartCommand :

Attributes :

1. Cart
2. Item

Methods :

1. Execute

Relationships :

1. Inheritance with Command Interface

—---

10. RemoveItemFromCartCommand :

Attributes :

1. Cart
2. Item

Methods :

1. Execute

Relationships :

1. Inheritance with Command Interface

—---

11. CreditCardPayment

Attributes :

1. Amount

Methods :

1. ProcessPayment()

Relationships :

1. Association with PaymentMethodFactory

—---

12. DebitCardPayment

Attributes :

1. Amount

Methods :

1. ProcessPayment()

Relationships :

1. Association with PaymentMethodFactory

—---

13. CreditCardPaymentFactory

Methods :

1. createPaymentMethod

Relationships :

1. **Inheritance with** PaymentMethodFactory

—---

14. DebitCardPaymentFactory

Methods :

1. createPaymentMethod

Relationships :

1. **Inheritance with** PaymentMethodFactory

—---

15. PurchaseProcessor

Methods :

1. selectItems();
2. calculateTotal();
3. applyDiscount();
4. processPayment();
5. deliverItems();

Relationships :

1. **Association with** AdministratorPurchaseProcessor

—---

16. CustomerPurchaseProcessor

Methods :

1. selectItems();
2. calculateTotal();
3. applyDiscount();
4. processPayment();
5. deliverItems();

Relationships :

2. **Inheritance with** PurchaseProcessor.

—---

17. AdministratorPurchaseProcessor

Methods :

1. selectItems();
2. calculateTotal();
3. applyDiscount();
4. processPayment();
5. deliverItems();

Relationships :

1. **Inheritance with** PurchaseProcessor.

—---

Interfaces :

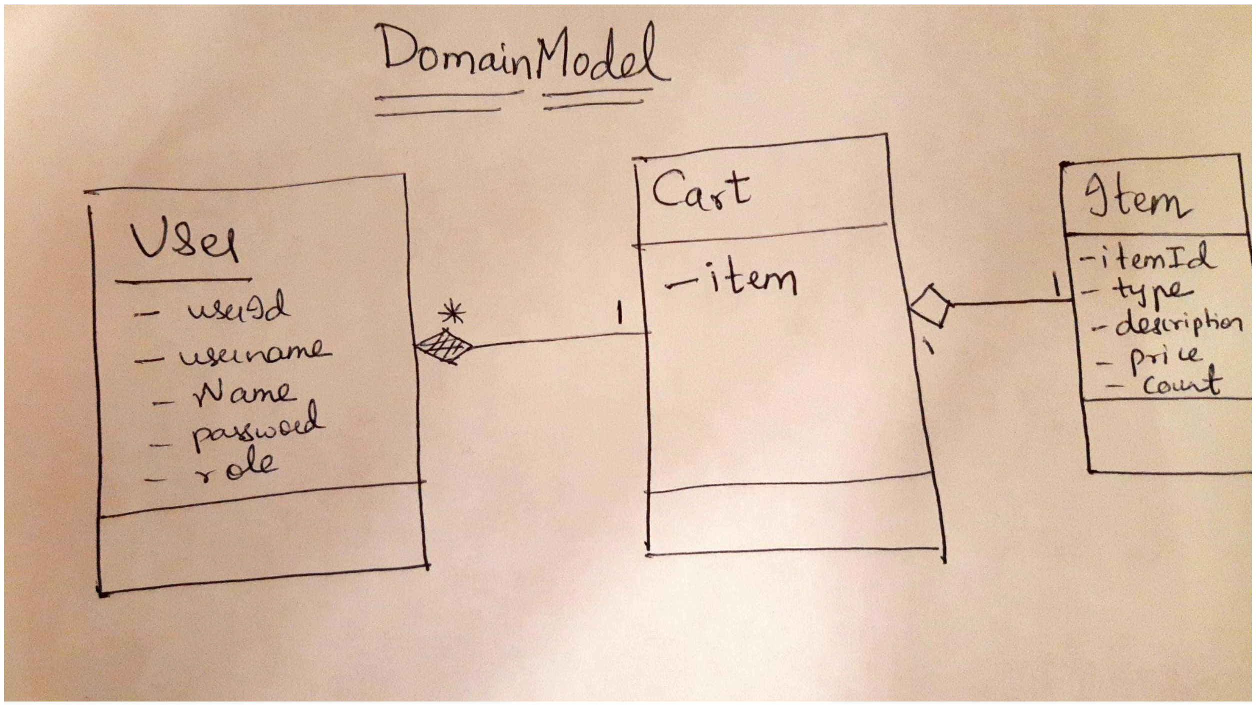
1. AdministratorManagement
    - addAdministrator
    - getAdministrator
    - getAdministrators
    - removeAdministrator
2. ItemManagement

- addItem
- removeItem
- getItems
- findItembyId

3. Command Interface
4. PaymentMethod
5. PaymentMethodFactory

Domain Model :

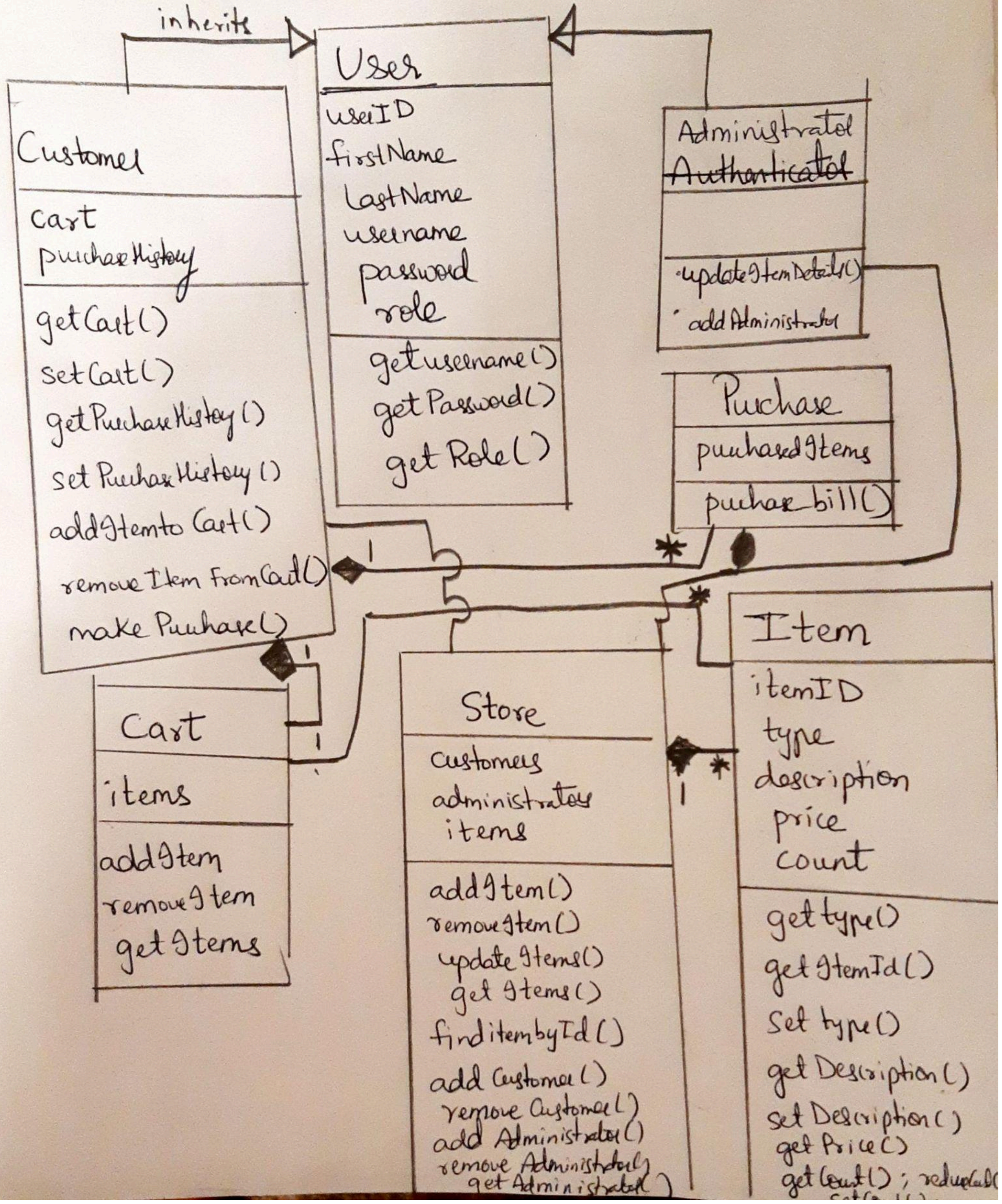
In the model, we only define the important classes at a higher level.

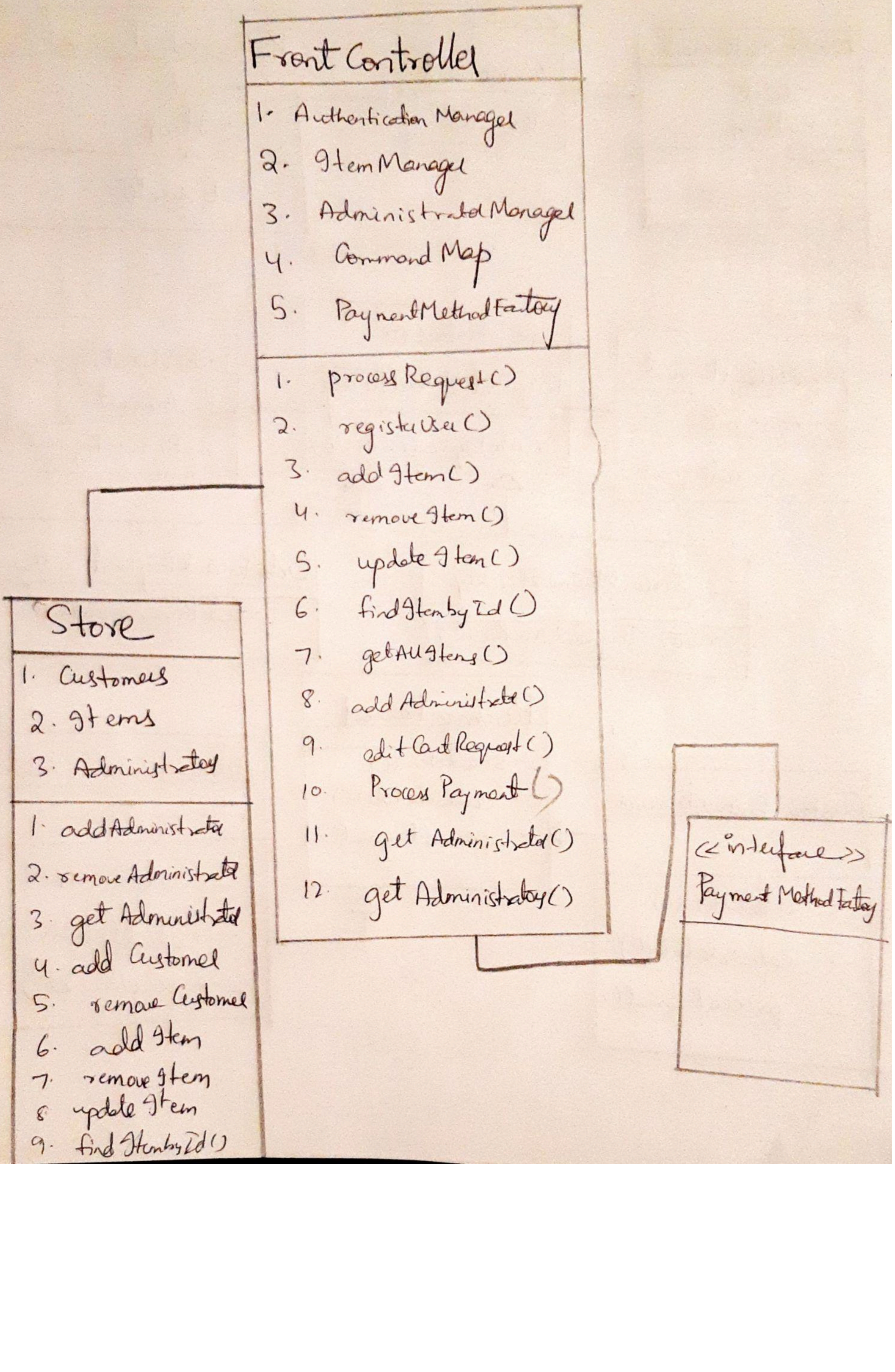


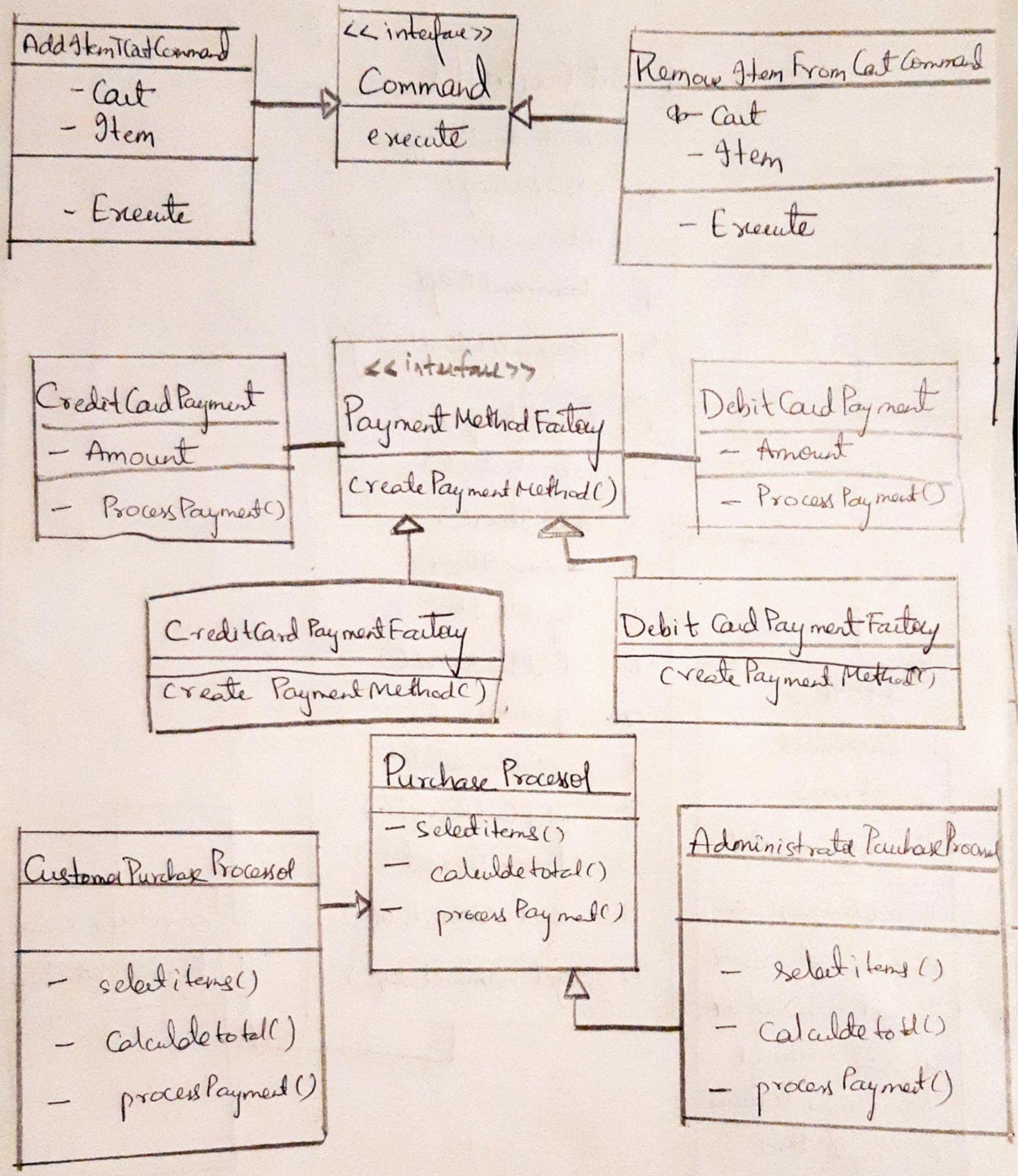
Class Diagram :

Implementing various design patterns resulted in many classes and interfaces which are difficult to be constrained in a single paper. So, please consider the following three diagrams as a single unit :

Class Diagram




Front Controller
1. AuthenticationManager
2. ItemManager
3. AdministratorManager
4. Command Map
5. PaymentMethodFactory
1. processRequest()
2. registerUser()
3. addItem()
4. removeItem()
5. updateItem()
6. findItembyId()
7. getAllItems()
8. addAdministrator()
9. editCardRequest()
10. Process Payment()
11. get Administrator()
12. get Administrators()
Store
1. Customers
2. Items
3. Administrators
1. addAdministrator
2. removeAdministrator
3. get Administrator
4. add Customer
5. remove Customer
6. add Item
7. remove Item
8 update Item
9. find ItembyId()
<<interface>>
Payment Method Factory


AddItemToCartCommand
- Cart
- Item
- Execute
<<interface>>
Command
execute
RemoveItemFromCartCommand
- Cart
- Item
- Execute
CreditCardPayment
- Amount
- ProcessPayment()
<<interface>>
PaymentMethodFactory
CreatePaymentMethod()
DebitCardPayment
- Amount
- ProcessPayment()
CreditCardPaymentFactory
CreatePaymentMethod()
DebitCardPaymentFactory
CreatePaymentMethod()
PurchaseProcessor
- selectitems()
- calculatetotal()
- processPayment()
CustomerPurchaseProcessor
- selectitems()
- calculatetotal()
- processPayment()
AdministratorPurchaseProcessor
- selectitems()
- calculatetotal()
- processPayment()