Birzeit University
Faculty of Engineering and Technology
Computer Science Department
SWEN6304: Software Design and Architecture
Project #1

Student Name: Duha Jarrar
Student ID: 1225272

Date: 25 Nov 2023

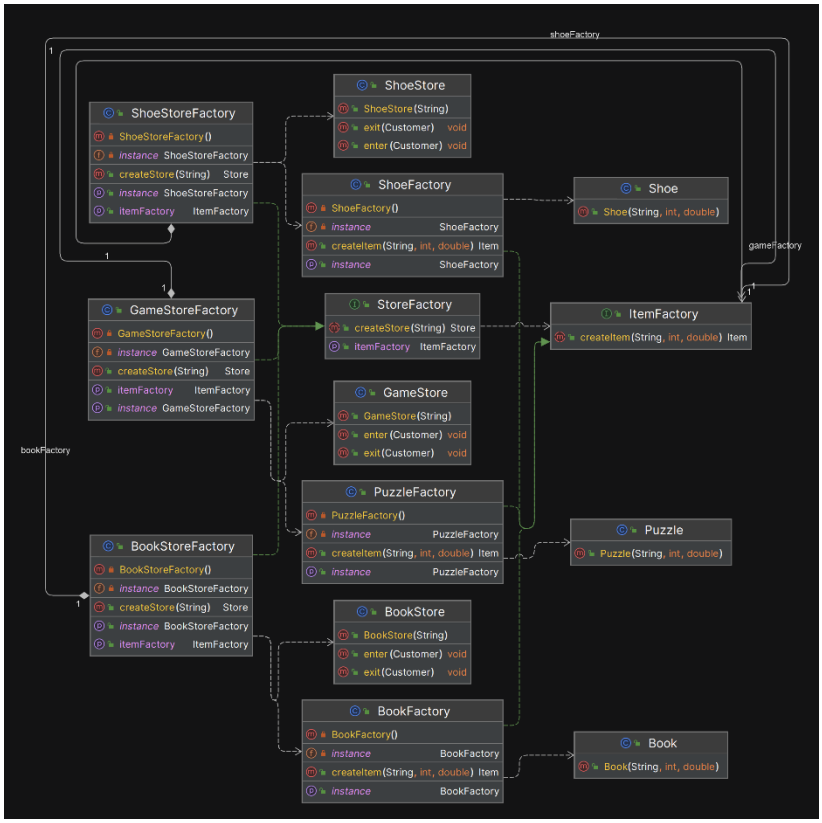
❖ Online Shopping Application Patterns

Source Code: https://github.com/duhajarrar/shop , you can find the UML images on GitHub too.

The following UML diagram shows the whole system we created.

- **Factory Method Pattern:**

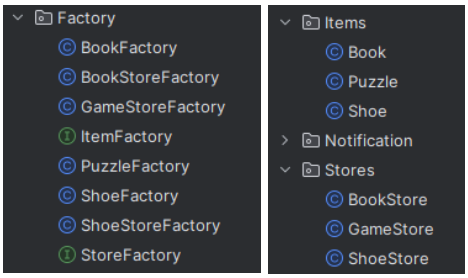
  The pattern solves the difficulty of creating different types of *Stores* and *Products* with flexible and extensible code. ***We apply it by*** defining factory interfaces (*StoreFactory*, *ItemFactory*) with methods (*createStore*, *createItem*), and concrete factories for each type of them, each of them will implement the create method as its respective type, for example, the *BookFactory* will implement the create store to return *new BookStore()*. ***Using this pattern*** allows us to add new types of stores and products easily with minimal changes and the code using it is open to addition and closed for modification. In addition, we added *ItemFactory* to each *StoreFactory* to allow the store to create its own items, the following image shows its UML diagram:



The following image shows how to use the pattern in creating items and stores:

```
StoreFactory bookStoreFactory = BookStoreFactory.getInstance();
Store bookStore = bookStoreFactory.createStore("Book Store");
Item book = bookStoreFactory.getItemFactory().createItem("Design Book 1",bookStore.getStoreId(),100);
```

The following image shows the classes created to apply the pattern:

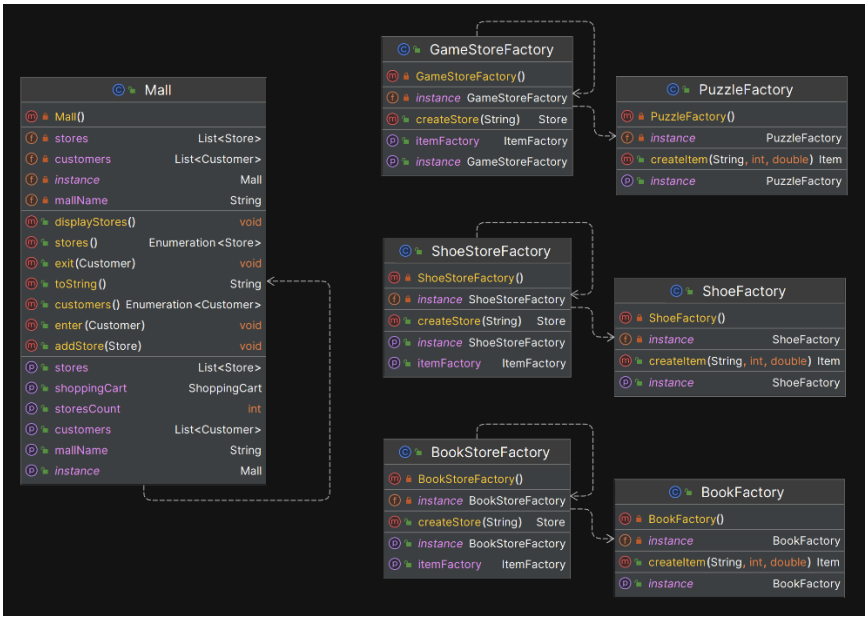


- **Singleton Pattern:**

  The pattern used to ensure that we have just one instance of any factory, ***we applied it*** by making the constructor of the needed class private and defining *getInstance()* static method that creates a new object or returns the instance if it exists, ***Using this*** will allow us to control the instantiation of the class, the following image shows how we use it:

```
Mall mall = Mall.getInstance();
StoreFactory gameStoreFactory = GameStoreFactory.getInstance();
StoreFactory shoeStoreFactory = ShoeStoreFactory.getInstance();
```

The following image shows the UML diagram of classes that used singleton:



- **Chain of Responsibility and Decorator Patterns:**

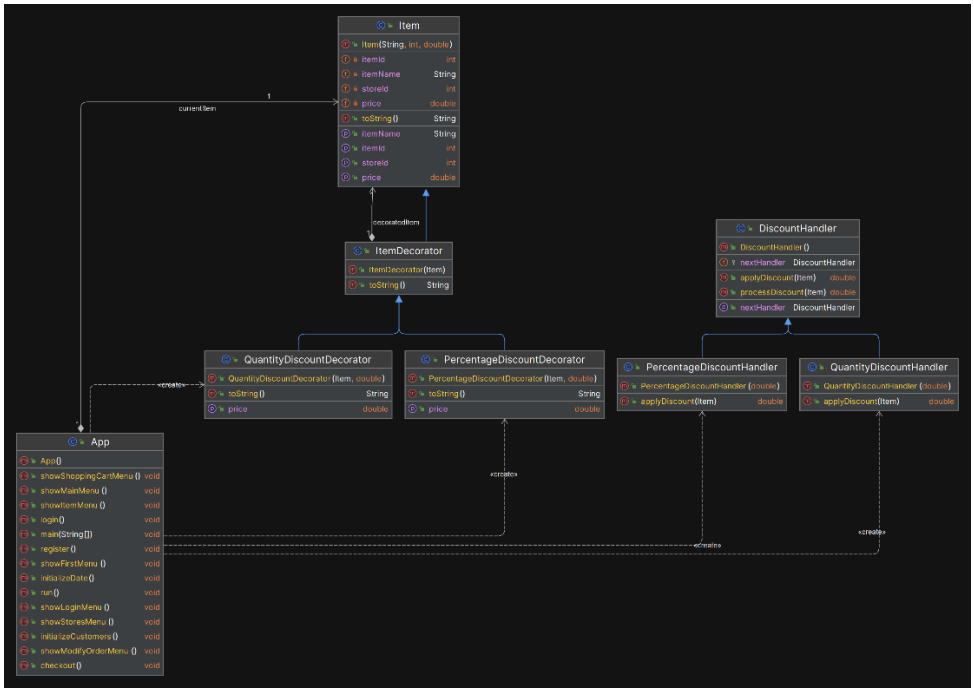
  Those patterns are used for adding discounts for the items. The *decorator pattern* is used to add one discount to the items, and the following image shows how to do that using it:

```
Item sportBoot1 = shoeStoreFactory.getItemFactory().createItem( itemName: "Nike boot",shoeStore.getStoreId(), price: 200);
Item discountedSportBoot = new QuantityDiscountDecorator(sportBoot1, discountQuantity: 100);
```

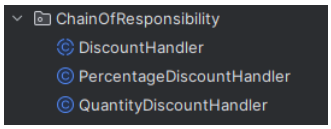
  The *chain of responsibility pattern* is used to add a sequence of discounts as shown in the following:

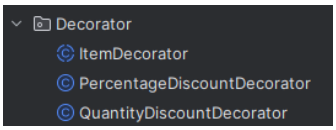
```
DiscountHandler quantDiscountHandler = new QuantityDiscountHandler( discountQuantity: 10);
quantDiscountHandler.setNextHandler(new PercentageDiscountHandler( discountPercentage: 20));
book.setPrice(quantDiscountHandler.applyDiscount(book));
```

  We applied them by defining abstract classes (*DiscountHandler* for chain, *ItemDecorator* for the decorator which extends from the Item) and then implemented from them concrete classes, the chain interface includes a method *applyDiscount* to apply the sequence of discounts, and the decorator includes *getPrice* method to calculate the new price after applying discount, ***Using those patterns*** allows us to flexible discount and extendable system, the following image shows the UML class diagram of the patterns classes:



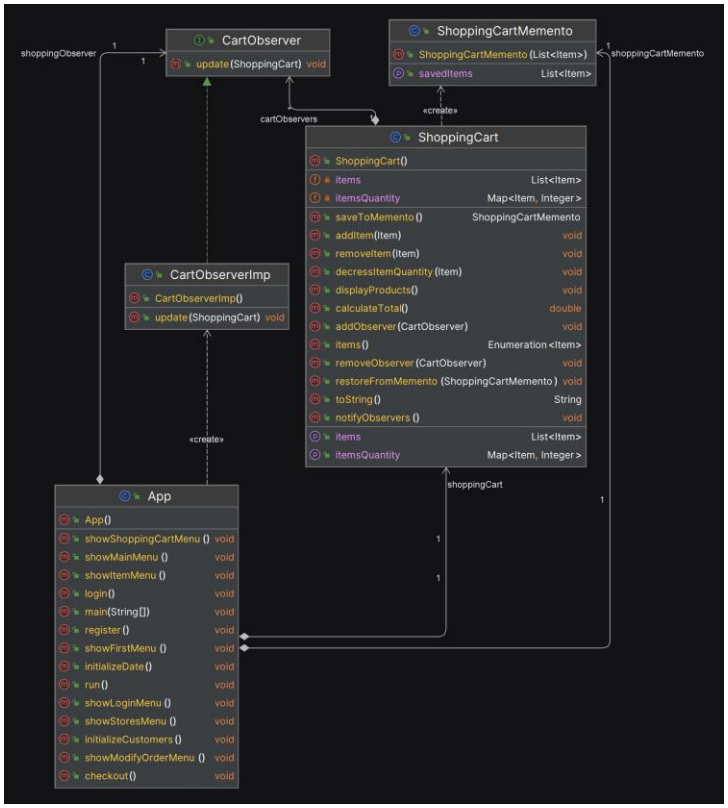
The code of those patterns shown in the classes shown in the following image:





- **Memento and Observer Patterns:**

  The memento solves the saving and restoring the previous state of the cart problem, and ***the observer solves*** the maintenance problem between the cart and inventory, it's used to notify any changes that happen to the cart. ***We applied the observer*** by defining a listener that includes an update method to notify the changes, ***We applied the memento*** by defining *ShoppingCartMemento* class that includes *getSavedItems* and adding *saveToMemento* and *restoreFromMemento* in the cart class, ***Using the patterns*** we can revert any changes in the cart which helps in the user experience and the notifications which reflected by any immediate changes keep the application observers synchronized with the cart, The image below shows the UML diagram of the patterns:



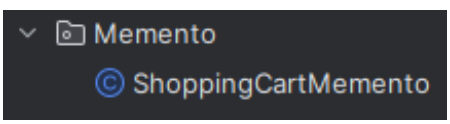
To use memento pattern we do the following:

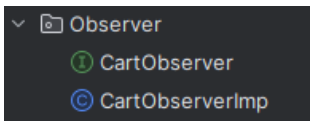
```
ShoppingCartMemento shoppingCartMemento = shoppingCart.saveToMemento();
```

To use cart observer we do the following:

```
CartObserver shoppingObserver = new CartObserverImp();
shoppingCart.addObserver(shoppingObserver);
```

The code of the patterns is found in the following file:



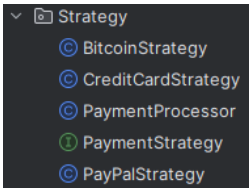


- **Strategy Pattern:**

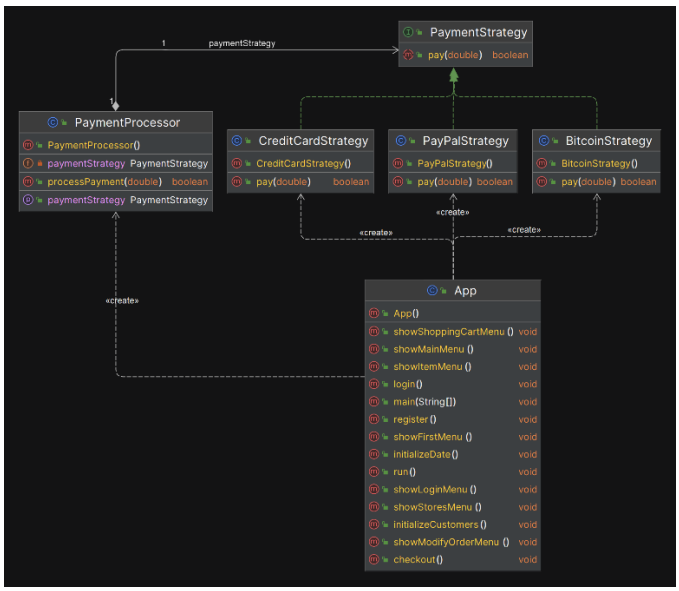
The pattern solves paying with different ways of payment. ***We applied it*** by creating the interface *PaymentStrategy* with the pay method and concrete strategies with different pay ways and each of those classes implements the pay method with its strategy, we also defined *PaymentProcessor* to be a client of the pattern, ***Using this pattern*** allows us to switch payment methods easily without changing the payment processing logic, To use it we do the following:

```
PaymentProcessor paymentProcessor = new PaymentProcessor(
paymentProcessor.setPaymentStrategy(new CreditCardStrategy())
paymentProcessor.processPayment(totalAmount);
```

The code files of the pattern:

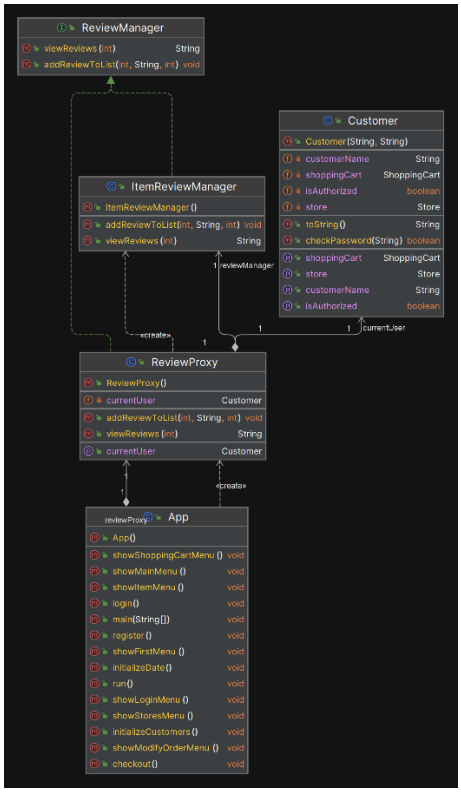


The image below shows the UML diagram of the patterns:



- **Proxy Pattern:**

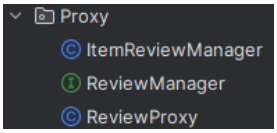
  The pattern solved the accessibility problem to prevent unauthorized users from adding reviews of the products and viewing reviews, ***we applied it*** by implementing *ReviewManager* Interface that includes needed methods (as *addReview*) and a concrete class (*ReviewProxy*) that implemented by it and checks the authorization of users, ***Using this pattern*** helped in decoupling the security logic and access control logic and provide a security layer, the following image shows the UML diagram of the classes used to apply it:



To use it we logged in using the AuthManager and then did the following:

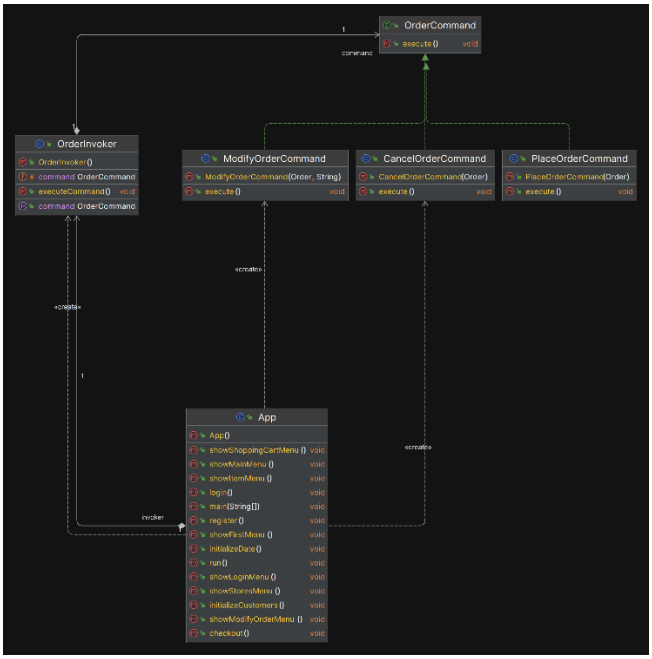
```
ReviewProxy reviewProxy = new ReviewProxy();
reviewProxy.setCurrentUser(currentUser);
reviewProxy.addReviewToList(currentItem.getItemId(),review_message, rate);
reviewProxy.viewReviews(currentItem.getItemId());
```

You can find the code files of it in the following:



- **Command Pattern:**

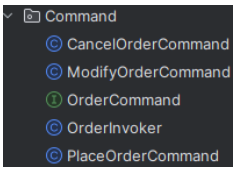
  The pattern reduces the coupling between the sender and executer by encapsulating the requests as objects as place, modify, and cancel, ***It's applied by*** defining the *OrderCommand* interface with the *execute* method and implementing it into concrete classes (as *PlaceOrderCommand*) that have its own implementation for execute, ***Using this pattern*** make the order management flexible and extensible, the following is a UML diagram of it:



The following is how we use the pattern:

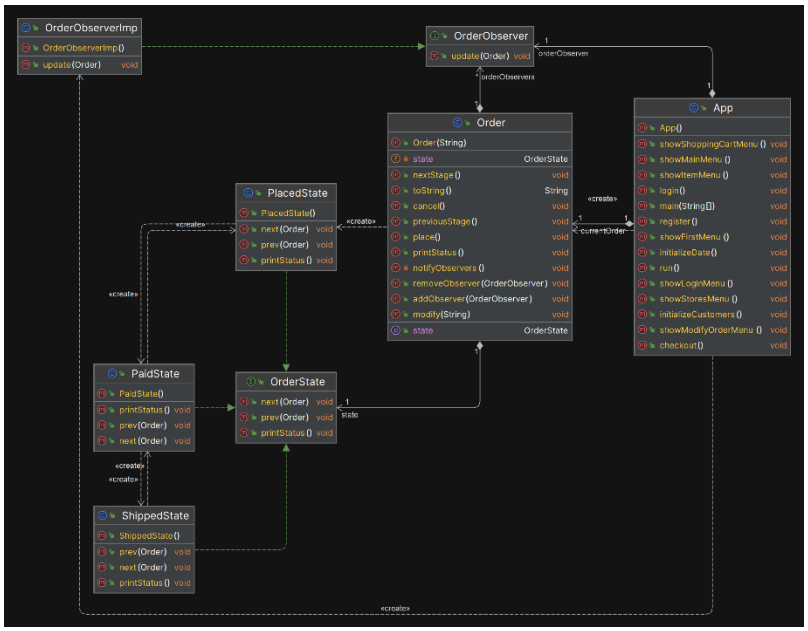
```
OrderCommand modifyOrderCommand = new ModifyOrderCommand(currentOrder, shoppingCart.toString());
invoker.setCommand(modifyOrderCommand);
invoker.executeCommand();
```

You can find the code files of it in the following:



- **State Pattern:**

The pattern solved the management of states in the order lifecycle as placing shipping and others, ***We applied it*** by defining *OrderState* interface with next and previous methods that are used to describe the next or previous state of each order and then we implemented from it concrete classes (as PaidState), ***Using this pattern*** the order will be manageable and maintainable, the following is a UML diagram of it:

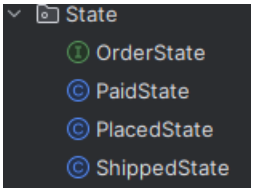


The following is how we use the pattern:

```
public Order(String orderDetails) {
    this.orderDetails = orderDetails;
    this.state = new PlacedState();
```

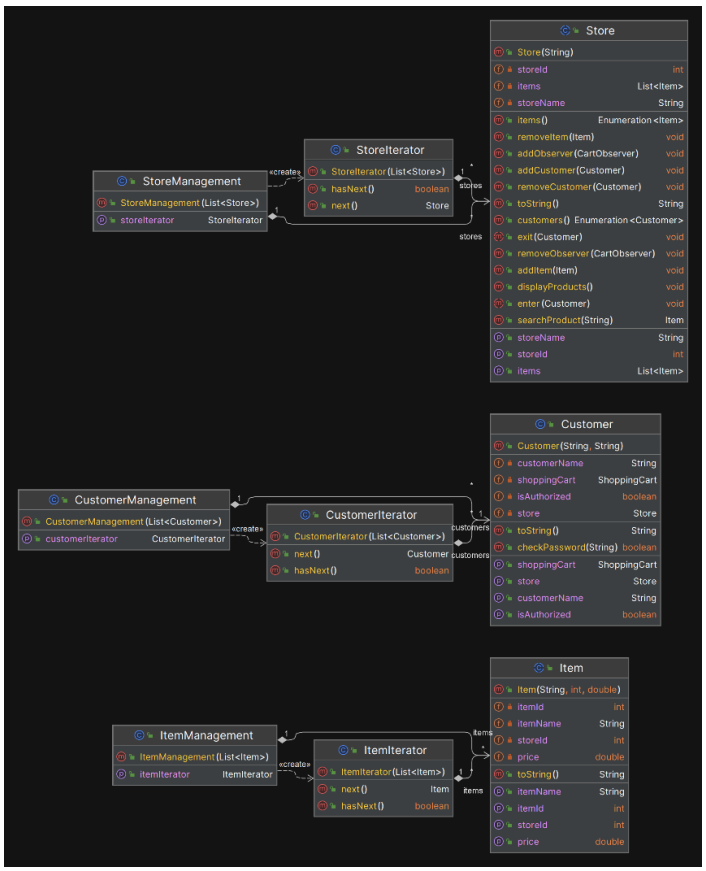
```
currentOrder.printStatus();
currentOrder.nextStage();
```

You can find the code files of it in the following:



- **Iterator Pattern:**

  The pattern used to enable sequential iteration in a collection, ***We applied it by*** creating an interface of the iterator with *next* and *hasNext* methods which are used in accessing the data, and then implement from it a concrete class for management, ***Using this pattern*** reduces coupling between the collection classes and the client code, the following is a UML diagram of it:



We didn’t use it because there is a built-in function that did that, but the following is how we can use it:

```
ItemManagement itemManagement = new ItemManagement(items);
ItemIterator iterator = itemManagement.getItemIterator();
while(iterator.hasNext()){
    System.out.println(iterator.next());
}
```

You can find the implementation code of it in the following: