1. Name: Yuchen Zhang
2. Project: E-Commerce Service
3. Features implemented

ID	Title
1	Customer signup
2	Customer sign in
3	Customer update
4	Customer add product to cart
5	Customer delete product from cart
6	Customer check out
7	Customer can review cart
8	Customer can review history orders
9	Admin signup
10	Admin sign in
11	Admin can add customer
12	Admin can update customer
13	Admin can set discount strategy
14	Storage admin can sign up
15	Storage admin can sign in
16	Storage admin can add storage
17	Storage admin can delete storage
18	Storage admin can check storage

4. Features unimplemented

ID	table
1	Customer can return products
2	Admin can refund customer

5. Final class diagram

   There are nearly 50 classes in total, so it is difficult to present the whole class diagram in one page. I display the class diagram without interface.

   I submit it as file Zhang_E-Commerce Service_Part6_Final Class Diagram.pdf

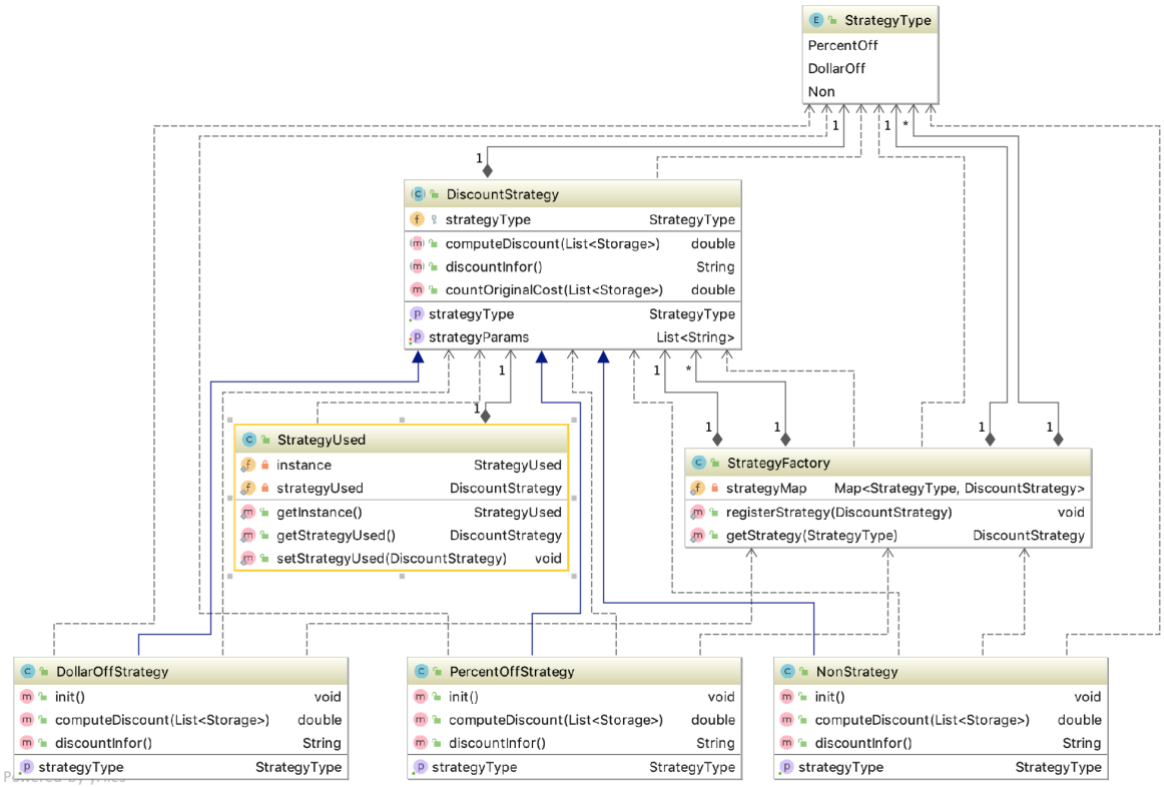
   I add many other classes compared with the first version. Because the some details has not been considered when design the first version class diagram. Such as I did not know to use model, dao, service structure to separate service level and database level.

6. Design Pattern Used
   1) Strategy and Factory Design Pattern

      Corresponding requirement:

      When customer checkout, there can be many discount strategies. Admin can set the discount strategy dynamically.

I need to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the customer clients that use it. That's why I choose strategy DP. Factory can define an interface for creating an object, but let subclasses decide which class to instantiate. It is very fit for my situation, which admin create an abstract class "DiscountStrategy" from factory, and set it as a static property of a singleton instance called "StrategyUsed.class". The class diagram is as below:



Class "DiscountStrategy" is an abstract class, and has two abstract methods computeDiscount, discountInfo, which should be overridden by three concrete strategies—DollarOffStrategy, PercentOffStrategy, NonStrategy. These four classes implemented Strategy Design Pattern.

Class "StrategyFactory" with the above four classes implement Factory Design Pattern, init() method of strategy class will register itself into the factory. Then other object can get strategy object using method getStrategy(StrategyType).

Singleton class StrategyUsed has a static property can reserve the current discount strategy, which is set by admin.

The classes are below:

```java
package com.ecommerceservice.strategy;

import java.util.List;

import org.springframework.stereotype.Component;

import com.ecommerceservice.model.common.Storage;


@Component
public abstract class DiscountStrategy {
    protected StrategyType  strategyType;
    protected List<String> strategyParams;

    public DiscountStrategy() {
    }
    public DiscountStrategy(StrategyType strategyType, List<String> strategyParams) {
        this.strategyType = strategyType;
        this.strategyParams = strategyParams;
    }

    public  abstract double computeDiscount(List<Storage> storagelist);
    public  abstract StrategyType getStrategyType();
    public  abstract String discountInfor() ;

    public List<String> getStrategyParams() {
        return strategyParams;
    }
    public void setStrategyParams(List<String> strategyParams) {
        this.strategyParams = strategyParams;
    }
    public double countOriginalCost(List<Storage> storagelist) {
        double totalBfDiscount = 0;
        for(Storage item:storagelist) {
            totalBfDiscount +=item.getPrice()*item.getNum();
        }
        return totalBfDiscount;
    }
}
```

```java
package com.ecommerceservice.strategy;

import java.util.List;

import javax.annotation.PostConstruct;

import org.springframework.stereotype.Component;

import com.ecommerceservice.model.common.Storage;

@Component
public class DollarOffStrategy extends DiscountStrategy{

    @PostConstruct
    public void init() {
        StrategyFactory.registerStrategy(this);
    }

    @Override
    public StrategyType getStrategyType() {
        return StrategyType.DollarOff;
    }

    @Override
    public double computeDiscount(List<Storage> storagelist) {
        double total = countOriginalCost(storagelist);
        if(total>=Double.valueOf(strategyParams.get(0))) {
            total -= Double.valueOf(strategyParams.get(1));
        }
        return total;
    }

    @Override
    public String discountInfor() {
        return "Buying "+strategyParams.get(0)+" saving "+strategyParams.get(1);
    }
}
```

```java
package com.ecommerceservice.strategy;

import java.util.List;

import javax.annotation.PostConstruct;

import org.springframework.stereotype.Component;

import com.ecommerceservice.model.common.Storage;

@Component
public class PercentOffStrategy extends DiscountStrategy{

    @PostConstruct
    public void init() {
        StrategyFactory.registerStrategy(this);
    }

    @Override
    public double computeDiscount(List<Storage> storagelist) {

        return countOriginalCost(storagelist)*(1-Double.valueOf(strategyParams.get(0))/100);
    }

    @Override
    public StrategyType getStrategyType() {
        return StrategyType.PercentOff;
    }

    @Override
    public String discountInfor() {

        return strategyParams.get(0)+"%OFF";
    }
}
```

```java
package com.ecommerceservice.strategy;

import java.util.List;

import javax.annotation.PostConstruct;

import org.springframework.stereotype.Component;

import com.ecommerceservice.model.common.Storage;

@Component
public class NonStrategy extends DiscountStrategy {

    @PostConstruct
    public void init() {
        StrategyFactory.registerStrategy(this);
    }

    @Override
    public StrategyType getStrategyType() {
        return StrategyType.Non;
    }

    @Override
    public double computeDiscount(List<Storage> storagelist) {

        return countOriginalCost(storagelist);
    }

    @Override
    public String discountInfor() {

        return "Non-discount.";
    }
}
```

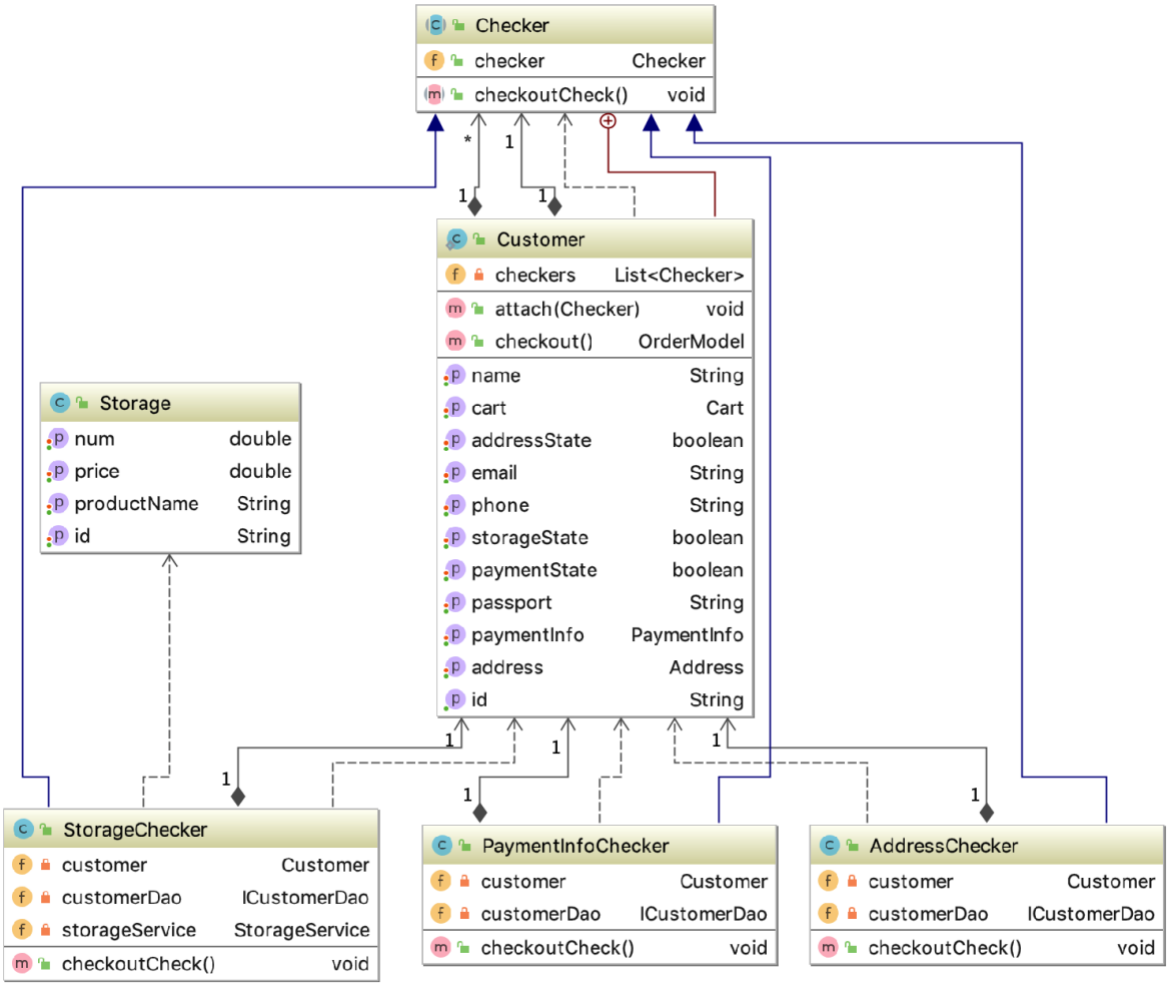
```java
package com.ecommerceservice.strategy;

import java.util.HashMap;
import java.util.Map;

import org.springframework.stereotype.Component;
@Component
public class StrategyFactory {

    private static Map<StrategyType,DiscountStrategy> strategyMap
    = new HashMap<StrategyType, DiscountStrategy>();

    public StrategyFactory() {
    }

    public static void registerStrategy(DiscountStrategy strategy) {
        strategyMap.put(strategy.getStrategyType(), strategy);
    }
    public static DiscountStrategy getStrategy(StrategyType type) {
        return strategyMap.get(type);
    }
}
```

```java
package com.ecommerceservice.strategy;

import org.springframework.stereotype.Component;

@Component
public class StrategyUsed {
    private static StrategyUsed instance;
    private static DiscountStrategy strategyUsed;
    private StrategyUsed() {}

    public static synchronized StrategyUsed getInstance() {
        if(instance==null) {
            instance = new StrategyUsed();
        }
        return instance;
    }
    public static DiscountStrategy getStrategyUsed() {
        return strategyUsed;
    }

    public static synchronized void setStrategyUsed(DiscountStrategy strategy) {
        if(instance==null) {
            instance = new StrategyUsed();
            strategyUsed = strategy;
        }else {
            strategyUsed = strategy;
        }
    }
}
```

2) Observer Design Pattern

Corresponding requirement: When customer checkout, system should check storage, check payment information, check address. Only if all these three are good, customer can checkout.

Using Observer Design Pattern, three checkers -- StorageChecker, PaymentChecker, AddressChecker -- will do checkoutCheck() method when customer checkout.
Class diagram as below:



Class “Checker” is an abstract Observer. class “StorageChecker”, “PaymentInfoChecker”, “AddressChecker” are inherited from “Checker”. When “Customer” checkout, it notifies all checkers to execute checkoutCheck() method.
Classes are shown:

```
package com.ecommerceservice.model.common;

public abstract class Checker {
 public Checker checker;
 public abstract void checkoutCheck();
}
```

```java
package com.ecommerceservice.model.common;

import java.util.List;

import com.ecommerceservice.dao.CustomerDao;
import com.ecommerceservice.dao.ICustomerDao;
import com.ecommerceservice.model.user.Customer;
import com.ecommerceservice.service.StorageService;
import com.ecommerceservice.service.StorageServiceImpl;

public class StorageChecker extends Checker{
    private Customer customer;
    private ICustomerDao customerDao = new CustomerDao();
    private StorageService storageService = new StorageServiceImpl();

    public StorageChecker(Customer customer) {
        this.customer = customer;
        this.customer.attach(this);
    }

    @Override
    public void checkoutCheck() {
        List<Storage> storageList = customer.getCart().getStorageList();
        customer.setStorageState(true);
        for(Storage storage: storageList) {
            boolean state =storageService.checkStorage(storage);
            if(state==false) {
                customer.setStorageState(false);
                break;
            }
        }
        customerDao.updateCustomer(customer);
    }

}
```

```java
package com.ecommerceservice.model.common;

import com.ecommerceservice.dao.CustomerDao;

public class AddressChecker extends Checker{

    private Customer customer;
    private ICustomerDao customerDao = new CustomerDao();
    public AddressChecker(Customer customer) {
        this.customer = customer;
        this.customer.attach(this);
    }
    @Override
    public void checkoutCheck() {
        Address add = customer.getAddress();
        if(!add.getAddressLine().isEmpty() && !add.getCity().isEmpty()
                && !add.getState().isEmpty() && !add.getZip().isEmpty()) {
            customer.setAddressState(true);
        }else {
            customer.setAddressState(false);
        }
        customerDao.updateCustomer(customer);
    }

}
```

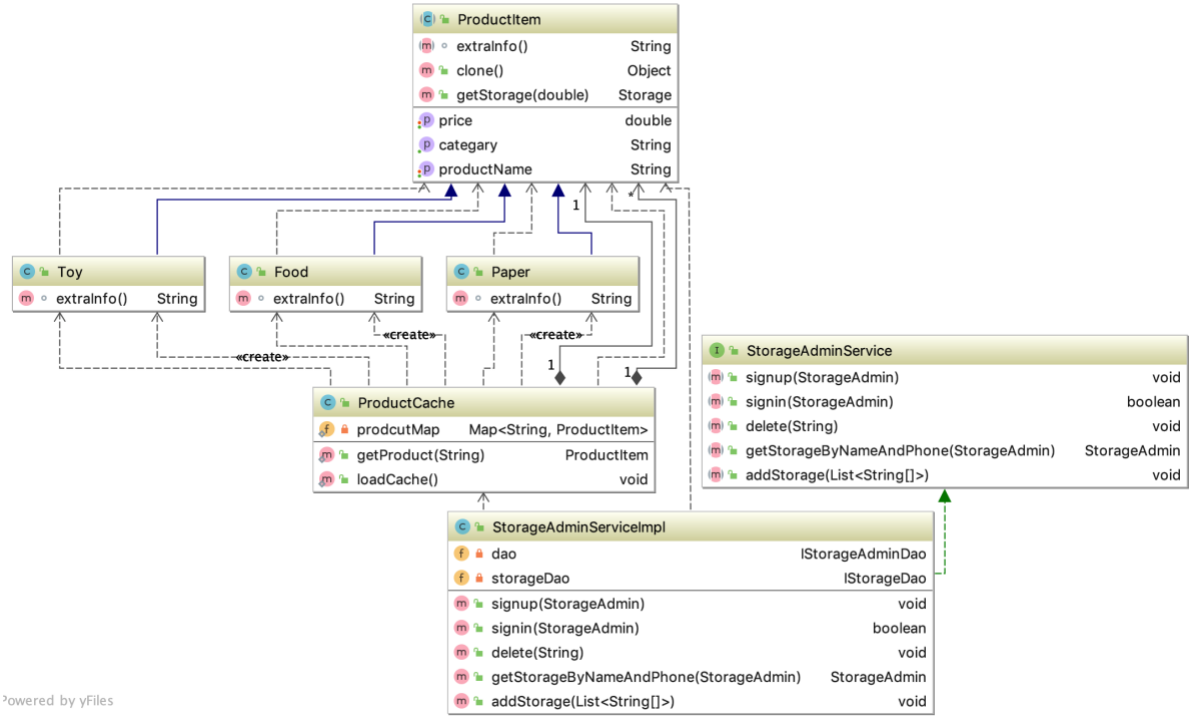
```java
package com.ecommerceservice.model.common;

import com.ecommerceservice.dao.CustomerDao;
import com.ecommerceservice.dao.ICustomerDao;
import com.ecommerceservice.model.user.Customer;
import com.ecommerceservice.model.user.PaymentInfo;

public class PaymentInfoChecker extends Checker{

    private Customer customer;
    private ICustomerDao customerDao = new CustomerDao();
    public PaymentInfoChecker(Customer customer) {
        this.customer = customer;
        this.customer.attach(this);
    }
    @Override
    public void checkoutCheck() {

        PaymentInfo payInfo = customer.getPaymentInfo();
        if(!payInfo.getAccountInfo().isEmpty() && !payInfo.getPayMethod().isEmpty()) {
            customer.setPaymentState(true);
        }else {
            customer.setPaymentState(false);
        }
        customerDao.updateCustomer(customer);
    }

}
```

```java
public class Customer {
    private String name;
    private Address address;
    private String phone;
    private String email;
    private Cart cart;
    private String id;
    private PaymentInfo paymentInfo;
    private boolean storageState;
    private boolean paymentState;
    private boolean addressState;
    private List<Checker> checkers = new ArrayList<Checker>();
    private String passport;

    public Customer() {

    public Customer(String name, Address address, String phone, String email) {

    public void attach(Checker checker) {
        checkers.add(checker);
    }

    public OrderModel checkout() {
        for(Checker checker: checkers) {
            checker.checkoutCheck();
        }
        if(storageState && paymentState && addressState) {
            DiscountStrategy discount = StrategyUsed.getInstance().getStrategyUsed();
            double beforeDiscount = discount.countOriginalCost(cart.getStorageList());
            double afterDiscount = discount.computeDiscount(cart.getStorageList());
            double tax=0.08 * afterDiscount;
            double total= afterDiscount +tax;

            OrderModel order = new OrderModel(cart.getStrageListStrFromList(cart.getStorageList()),
                    beforeDiscount-afterDiscount,tax);
            return order;
        }
        return null;
    }

    public boolean isStorageState() {
```

3) Prototype Design Pattern

Corresponding requirement: storage admin need to add a lot of Product objects.

Using Prototype DP, storage admin can create new object by its prototype based on the product category. Class diagram as below:



Class “ProductItem” is an abstract class, which implements Cloneable Interface. Class “Food”, “Toy”, “Paper” extends “ProductItem”. Use class “ProductCache” to reserve “ProductItem”. Storage admin can get “ProductItem” from “ProductCache”.

```java
package com.ecommerceservice.model.product;

import com.ecommerceservice.model.common.Storage;

public abstract class ProductItem implements Cloneable{

    protected String productName;
    protected String categary;
    protected double price;

    abstract String extraInfo();

    public Object clone() {
          Object clone = null;
          try {
             clone = super.clone();
          } catch (CloneNotSupportedException e) {
             e.printStackTrace();
          }
          return clone;
    }
    public Storage getStorage(double num) {
        Storage storage =  new Storage();
        storage.setProductName(productName);
        storage.setPrice(price);
        storage.setNum(num);
        return storage;
    }

    public String getProductName() {
        return productName;
    }

    public void setProductName(String productName) {
        this.productName = productName;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }
```

```java
package com.ecommerceservice.model.product;

public class Food extends ProductItem{

    public Food() {
        categary = "food";
    }

    @Override
    String extraInfo() {
        return "please reserve in reftigerator";
    }

}
```

```java
package com.ecommerceservice.model.product;

public class Paper extends ProductItem{
    public Paper() {
        categary="paper";
    }
    @Override
    String extraInfo() {
        return "Please reserve in dry area.";
    }

}
```

```java
package com.ecommerceservice.model.product;

public class Toy extends ProductItem{

    public Toy() {
        categary="toy";
    }
    @Override
    String extraInfo() {

        return "None.";
    }

}
```

```
package com.ecommerceservice.model.product;

import java.util.HashMap;

public class ProductCache {
    private static Map<String, ProductItem> prodcutMap
    = new HashMap<String, ProductItem>();

 public static ProductItem getProduct(String categary) {
     ProductItem cachedProduct = prodcutMap.get(categary);
    return (ProductItem) cachedProduct.clone();
 }

 public static void loadCache() {
    Food food = new Food();
    prodcutMap.put(food.getCategary(),food);

    Paper paper = new Paper();
    prodcutMap.put(paper.getCategary(),paper);

    Toy toy = new Toy();
    prodcutMap.put(toy.getCategary(),toy);
 }
}
```

7. During the process of analysis and design, I learned that design pattern is a tool to let the code organized better, not the target. Design pattern should be apply in appropriate situation. Especially, when I consider design pattern, I must be clear what requirement I need to implement, then I can compare different DP to choose the appropriate one.