City University of Hong Kong

Department of Computer Science

CS3343 (A) Software Engineering Practice

2014/15

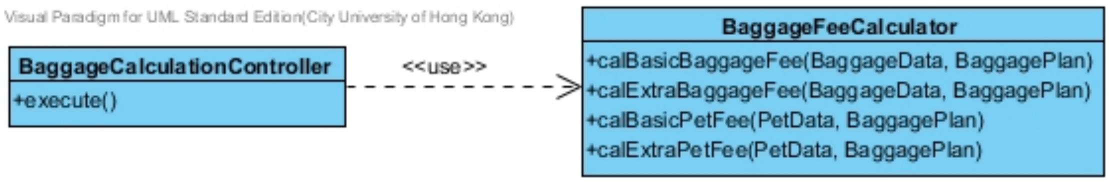
Refactoring Report

Student Names	Student ID
HO, Wai Kit	53144248
WONG, Chung Man	53145233
YIU, Yiu Yeung	53144144
Kong , Tsz Kit	53143798
Lau, Kam Yu	53144170
So, Chun Hei	53144525

Refactoring #1: Apply Dependency Inversion Principle (DIP) and Reduce Duplication

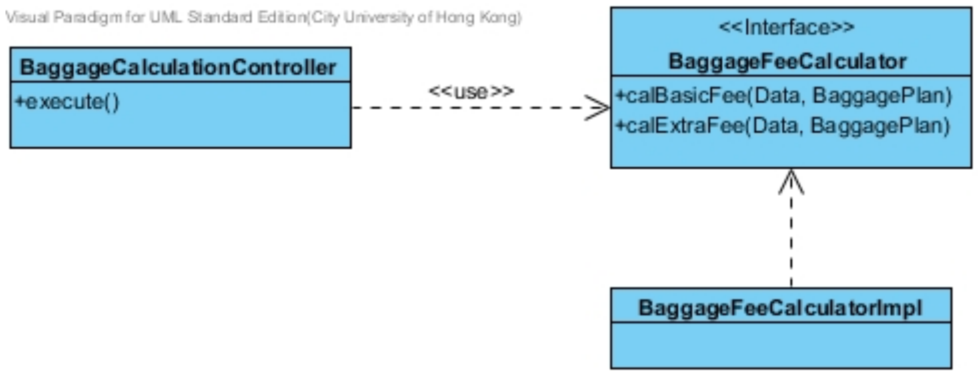
Situation:

The BaggageFeeCalculator violates DIP that the baggage controller depends on this calculator directly. Also, it has so many duplicate codes and method. The problems are shown on class diagram as the following:


Visual Paradigm for UML Standard Edition(City University of Hong Kong)
BaggageCalculationController
+execute()
<<use>>
BaggageFeeCalculator
+calBasicBaggageFee(BaggageData, BaggagePlan)
+calExtraBaggageFee(BaggageData, BaggagePlan)
+calBasicPetFee(PetData, BaggagePlan)
+calExtraPetFee(PetData, BaggagePlan)

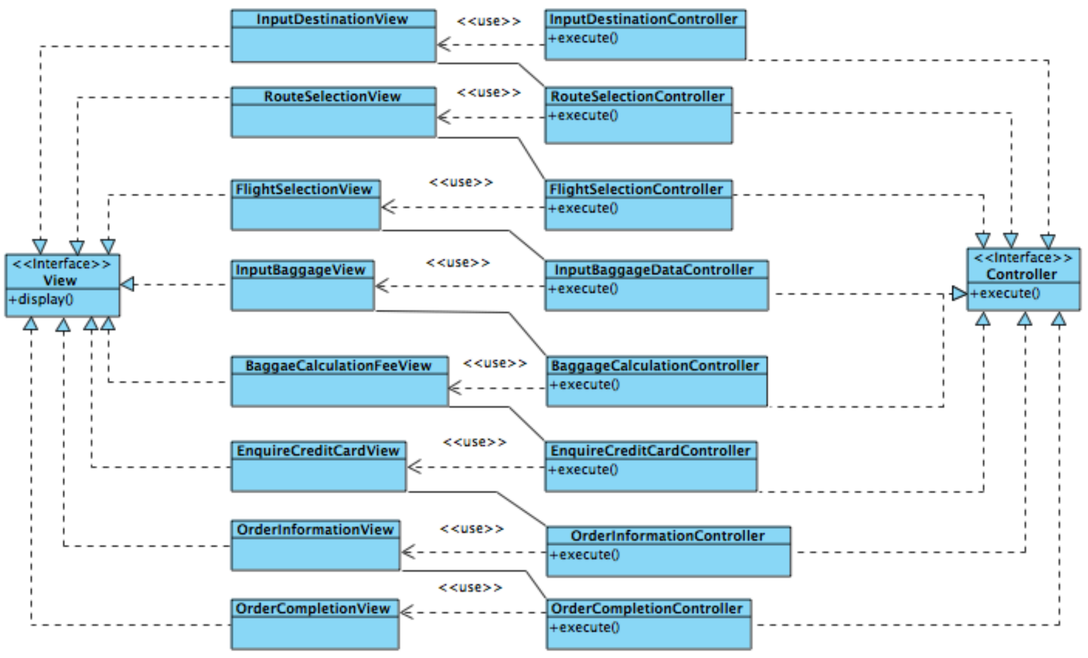
Solution:

Apply DIP that create an interface for the BaggageFeeCalculator such that any class which needed to depend on a baggage fee calculator can depend on its interface. Thus, it can reduce the dependence that the system can also fulfill Open-Close Principle if a new calculator is created. In addition, the duplicated codes are grouped together. Common methods are created for similar behaviors. Such that the program can reuse the method and reduce amount of modification when the behaviors is in development. The refactored codes are shown by the following class diagram:


Visual Paradigm for UML Standard Edition(City University of Hong Kong)
BaggageCalculationController
+execute()
<<use>>
<<Interface>>
BaggageFeeCalculator
+calBasicFee(Data, BaggagePlan)
+calExtraFee(Data, BaggagePlan)
BaggageFeeCalculatorImpl

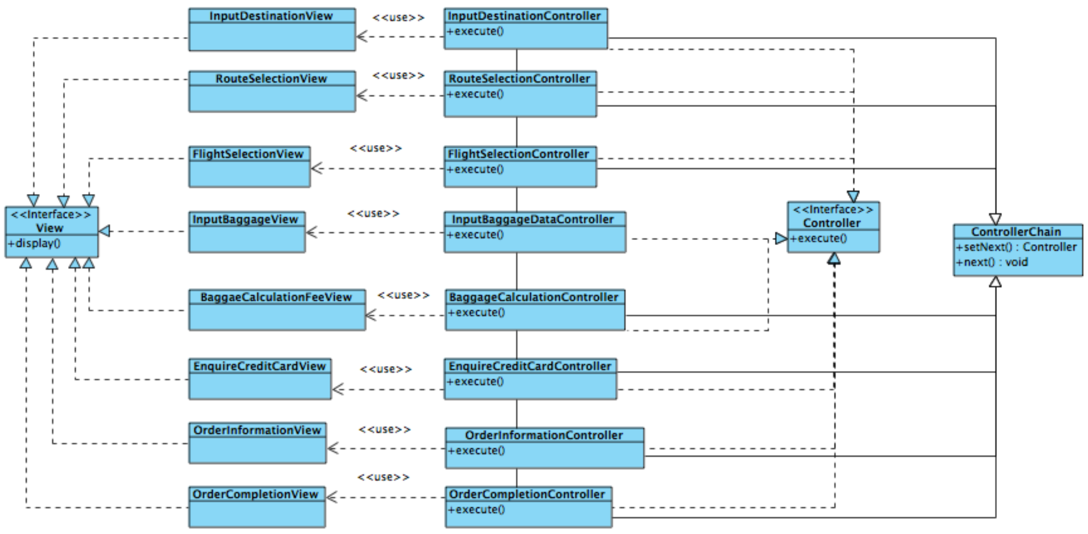
<u>Refactoring #2: Apply Chain Pattern to reduce the complexity of the relationship between Controller and View</u>

The relationship of controllers and views is too complexity, when creating the controller object, it needs to accept the view in the constructor for the execution. On the other hand, when initializing the view object, it also needs to put the next controller after completed the task of the view.



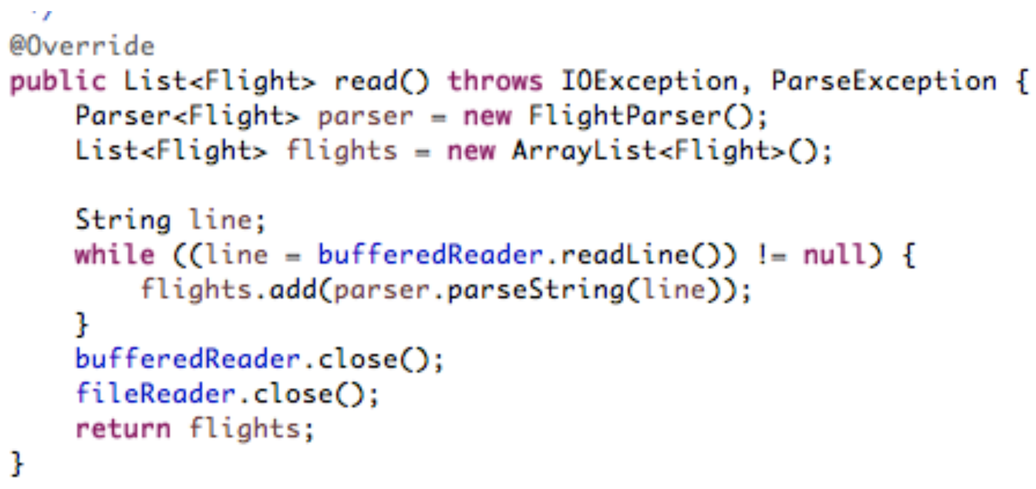
Solution:

Chain pattern is used for all the controllers. It seems like a chain that it will call the next() method to invoke the next controller for the execute(). It can prevent two classes replied to each other and the complexity of view and controller relationship. Therefore, the modification and extension will be easily since we only need to change the sequence of the chain in the main() method or add the new controller in the middle smoothly if any modification needed.


InputDestinationView
RouteSelectionView
FlightSelectionView
InputBaggageView
BaggaeCalculationFeeView
EnquireCreditCardView
OrderInformationView
OrderCompletionView
<<Interface>>
View
+display()
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
InputDestinationController
+execute()
RouteSelectionController
+execute()
FlightSelectionController
+execute()
InputBaggageDataController
+execute()
BaggageCalculationController
+execute()
EnquireCreditCardController
+execute()
OrderInformationController
+execute()
OrderCompletionController
+execute()
<<Interface>>
Controller
+execute()
ControllerChain
+setNext() : Controller
+next() : void

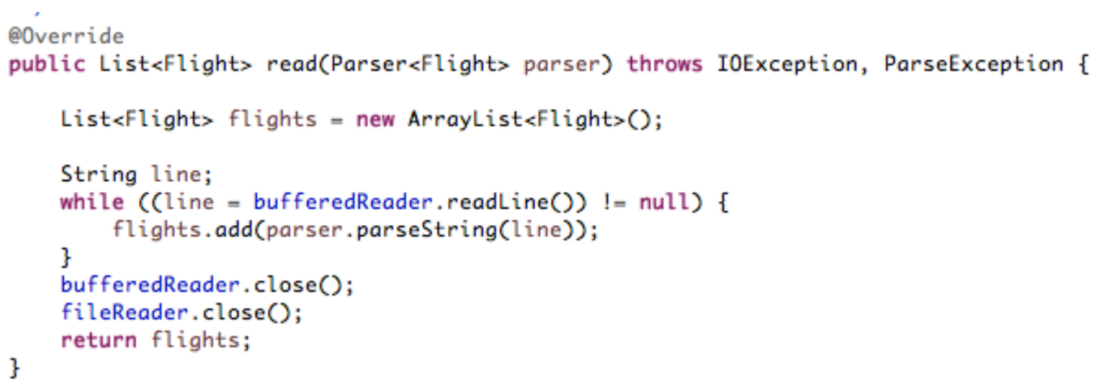
Refactoring #3: Open Close Principle using for future extension

The parser will be created inside the read() method, it makes difficult to change or extend to achieve in the future. It violates the OCP that open for the extension and closed for the modification. So we need to modify it for better code structure.



```
 */
@Override
public List<Flight> read() throws IOException, ParseException {
    Parser<Flight> parser = new FlightParser();
    List<Flight> flights = new ArrayList<Flight>();

    String line;
    while ((line = bufferedReader.readLine()) != null) {
        flights.add(parser.parseString(line));
    }
    bufferedReader.close();
    fileReader.close();
    return flights;
}
```

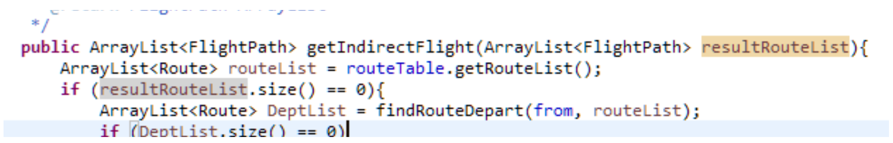
To provide the flexible software architecture and the increase maintainability, the read() method changes to accept the parameter parser. If the other flight parser will be used later. It don’t need any changes of the code.



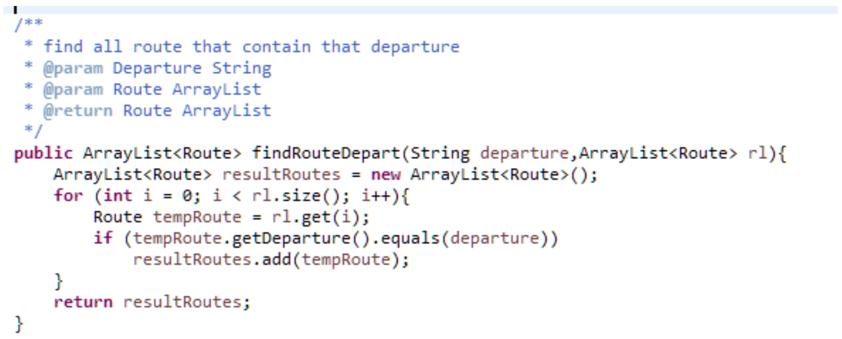
```
@Override
public List<Flight> read(Parser<Flight> parser) throws IOException, ParseException {

    List<Flight> flights = new ArrayList<Flight>();

    String line;
    while ((line = bufferedReader.readLine()) != null) {
        flights.add(parser.parseString(line));
    }
    bufferedReader.close();
    fileReader.close();
    return flights;
}
```

Refactoring #4: Single responsibility principle in different classes

Before refactoring, we get route list from RouteTable class, and then find out the route list which route are containing that departure in the PathFinding Class. But in single responsibility principle should encapsulate all the responsibility and data.



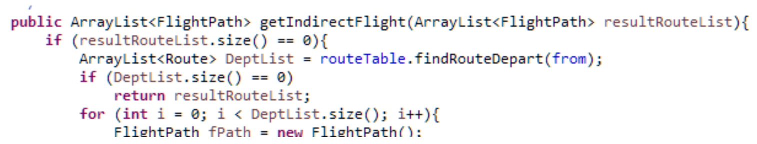
```
  */
 public ArrayList<FlightPath> getIndirectFlight(ArrayList<FlightPath> resultRouteList){
     ArrayList<Route> routeList = routeTable.getRouteList();
     if (resultRouteList.size() == 0){
         ArrayList<Route> DeptList = findRouteDepart(from, routeList);
         if (DeptList.size() == 0)
```



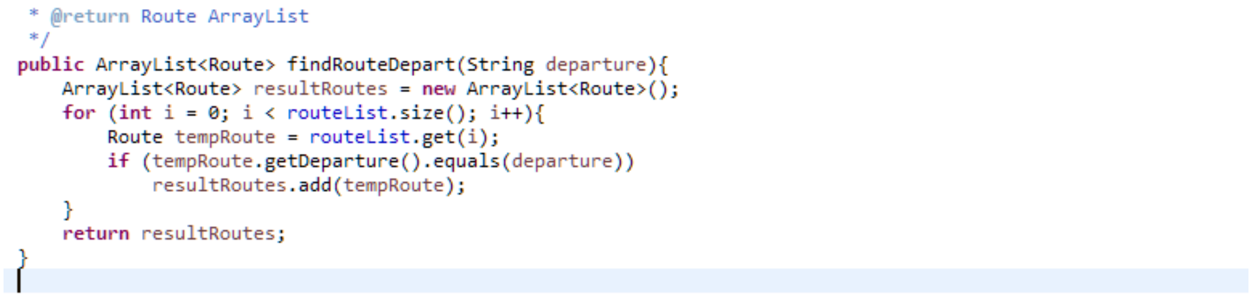
```
/**
 * find all route that contain that departure
 * @param Departure String
 * @param Route ArrayList
 * @return Route ArrayList
 */
public ArrayList<Route> findRouteDepart(String departure,ArrayList<Route> rl){
    ArrayList<Route> resultRoutes = new ArrayList<Route>();
    for (int i = 0; i < rl.size(); i++){
        Route tempRoute = rl.get(i);
        if (tempRoute.getDeparture().equals(departure))
            resultRoutes.add(tempRoute);
    }
    return resultRoutes;
}
```

After refactoring, I have mode the findRouteDepart method back into RouteTable class. Therefore, in PathFinding class we don't have to get the entire route list from RouteTable class anymore and call the findRouteDepart.

We call the findRouteDepart encapsulated from RouteTable class and the PathFinding class only handling one task, which are route path finding.



```
 public ArrayList<FlightPath> getIndirectFlight(ArrayList<FlightPath> resultRouteList){
     if (resultRouteList.size() == 0){
         ArrayList<Route> DeptList = routeTable.findRouteDepart(from);
         if (DeptList.size() == 0)
             return resultRouteList;
         for (int i = 0; i < DeptList.size(); i++){
             FlightPath fPath = new FlightPath();
```



```
 * @return Route ArrayList
 */
public ArrayList<Route> findRouteDepart(String departure){
    ArrayList<Route> resultRoutes = new ArrayList<Route>();
    for (int i = 0; i < routeList.size(); i++){
        Route tempRoute = routeList.get(i);
        if (tempRoute.getDeparture().equals(departure))
            resultRoutes.add(tempRoute);
    }
    return resultRoutes;
}
```