A PROJECT REPORT
ON

TRAVEL RESERVATION SERVICE

Submitted to
UNIVERSITY BORDEAUX 1

BY

NGUYEN Quang Anh	Team Leader
TRAN Thanh Phu	Member

MASTER INFORMATIC - SOFTWARE ENGINEERING
UNIVERSITY BORDEAUX 1

2014-2015

ABSTRACT

Design pattern is a strong tool in Object Oriented Programming. The purpose of this project is to test the ability of understanding and implementing the design patterns.

In this project, I have implemented 11 design patterns. They are Singleton, Abstract Factory, Builder, Proxy, Composite, Decorator, Observer, Iterator, Visitor, Template

Contents

List of Figures

1 INTRODUCTION

In this project, we immitated a system that work for a travel agency. This system can book two main types of travel for the clients: travels without service and travels with services. Travel without service, means that the reservation is made only for the flight, whereas travel with services allows a demand for checking in an hotel an renting a car.

For the purpose of simplifying the input process, we prepared some input in advance. Below is the tables that present the available flights betweens the cities.

Departure	Destination				
	Paris	Bordeaux	Canberra	Tokyo	Delhi
Hanoi	x	x	x	x	x
Hochiminh	x			x	x
Hue	x		x	x	
Haiphong		x	x		
Paris		x	x	x	
Bordeaux	x			x	x
Canberra	x	x		x	x
Tokyo	x		x		x
Delhi	x		x	x	

Between Hanoi, Hochiminh, Hue and Haiphong, there are flights from one to the others.

2 PROGRAM 'S SPECIFICATION OVERVIEW

2.1 Brief Introduction

Our project 's purpose is to help a travel agency make a booking for clients. There're two kinds of service package supported

- Travel without service: consist of only flight's reservation
- Travel with services: apart from flight's reservation, hotel's reservation and car renting could also be added

There're three kinds of flight's reservation

```
public enum FlightTicketType {
   VIP, Normal, Pool
}
```

The *Pool* ticket is the type of ticket proposed to the travel agency by a flight agency. The only difference between *Pool* and *Normal* is that *Pool*'s price is cheaper than the other.

There're also three kinds of hotel's reservation

```
public enum HotelServiceType {
   Cheap, Normal, Lux
}
```

2.2 Software Usage

The creation of a Travel's object could be done using class *TravelBuilder*. In this class, there're two methods that can create a travel without service

```
public Travel buildTravelNoServiceForOneClient(Client c, FlightTicketType
    type, CityName destination)
public Travel buildTravelNoServiceForGroup(Client c, FlightTicketType type,
    CityName destination)
```

When a client register a service, if there isn't any direct flight between the client's current city and the destination city, a transit flight will be added.

For creating a travel with services, consist of only one trip, we could use

```
public Travel buildTravelSimpleService(Client c, CityName destination,
    FlightTicketType f, HotelServiceType h)
```

For a touring service, a trip through many cities, we need two functions

```
public Travel buildTravelTouringService(Client c, CityName destination,
    FlightTicketType f, HotelServiceType h)
public void addNextTour(TravelTouringService t, CityName destination,
    FlightTicketType f, HotelServiceType h)
```

The first method is for building the first trip of the travel service. After that, anymore trip will be added using the second one. There's one problem in this setup, that's starting from the second trip, a trip to a new city must be a direct flight.

In this project, I have used in total eleven design patterns. These patterns could be divided into three groups

- Creational: Singleton, Abstract Factory, Builder
- Structural: Proxy, Composite, Decorator
- Behavioral: Strategy, Observer, Iterator, Visitor, Template

Each of these patterns will be presented throughly in the next parts.

3 PROJECT'S REQUEST

In this project, we have to finish several request. The requests are stated below

- Give the class diagram for this application
- Give the collaboration doagram for the reservation of a simple travel without service
- Give the sequence diagram for the reservation of a touring travel with service
- Gice the state diagram for a reservation of a touring travel with service

3.1 Class diagram

Below is the original class diagram, no design pattern is included.

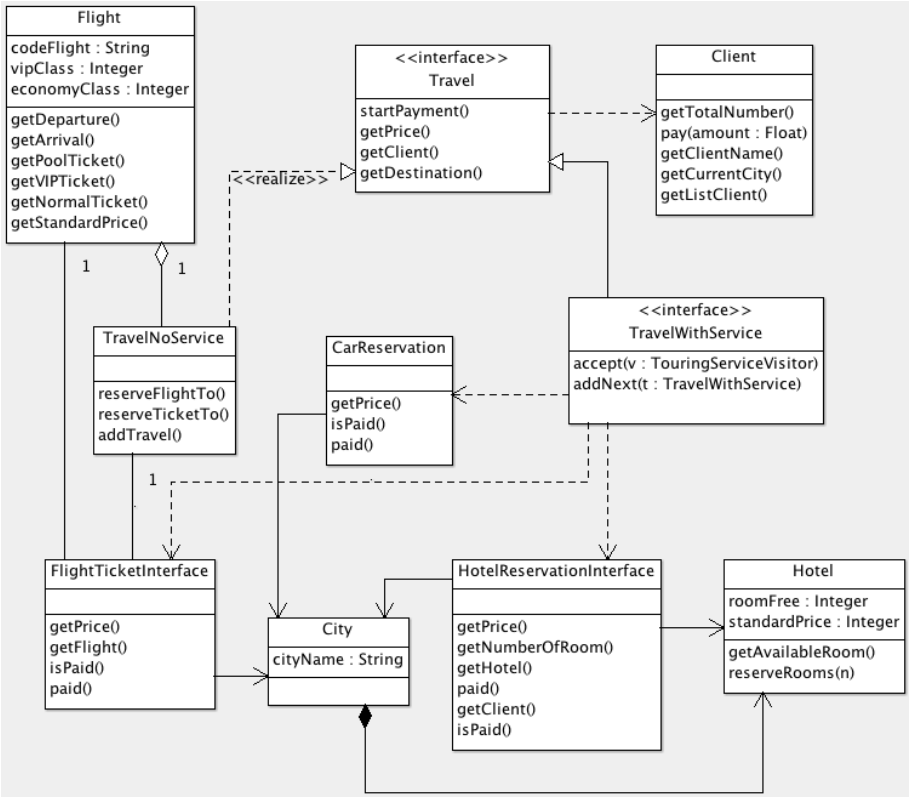

Figure 1: Class diagram of the project

3.2 Collaboration diagram

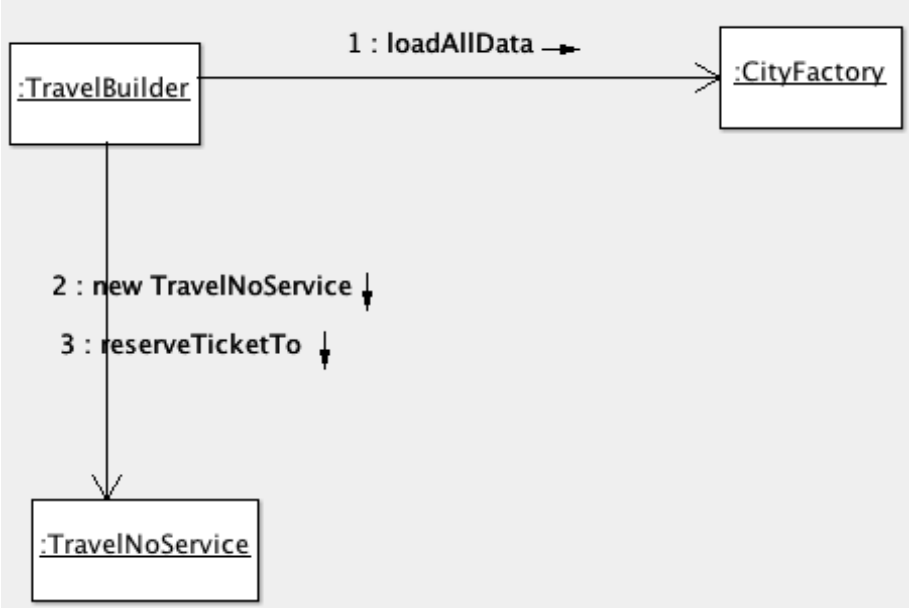

Figure 2: Collaboration diagram

To create a reservation without service, an instance of class *TravelNoService*, we need three steps.

- A factory need to load all data from file.
- TravelBuilder create an instance of *TravelNoService*. No specified data yet.
- TravelBuilder request this instance to use method *reserveTicketTo* to make a flight reservation, then return the instance.

3.3 Sequence diagram

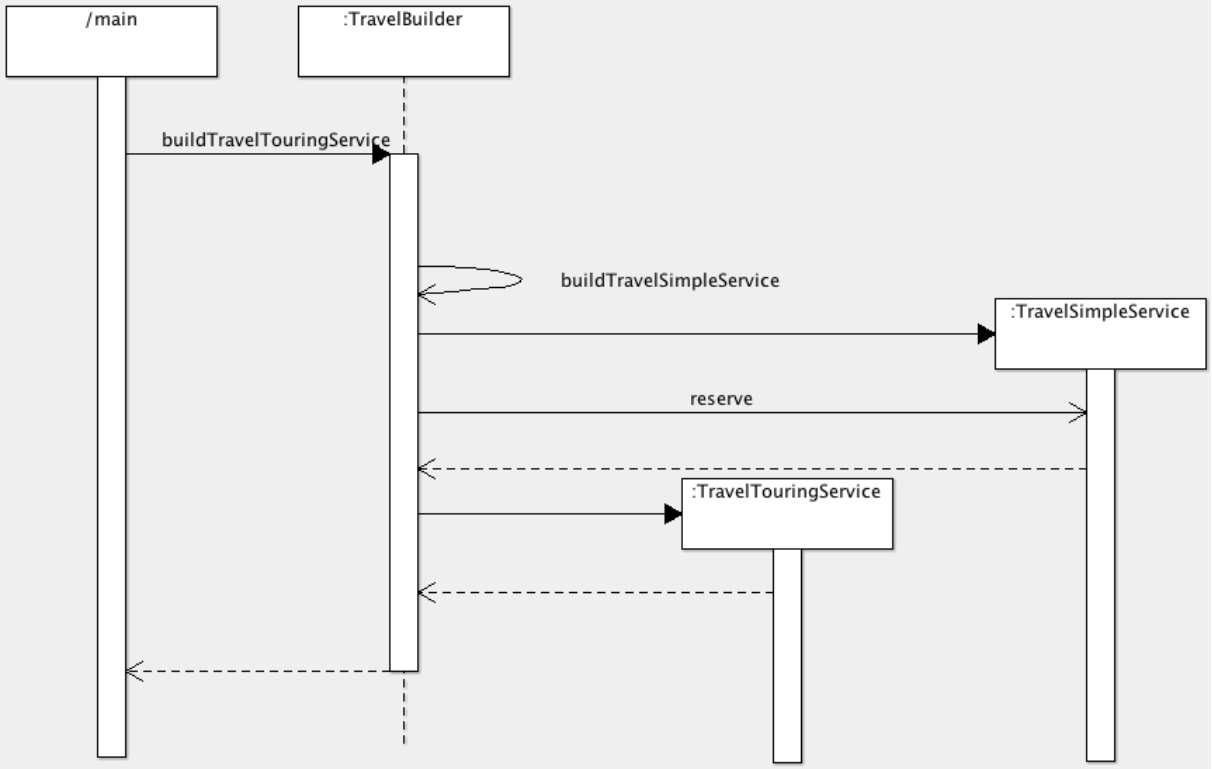

Figure 3: Sequence diagram

Making a reservation of a touring service have several steps :

- Main class will make call to method *buildTravelTouringService* of the class *TravelBuilder*
- TravelBuilder call its own's method *buildTravelSimpleService*
- The method *buildTravelSimpleService* create an instance of class *TravelSimpleService*, and use this instance to make a function-call *reserve*
- The instance created is now used to create an instance of class *TravelTouringService*, and return this new instance to the main class

3.4 State diagram

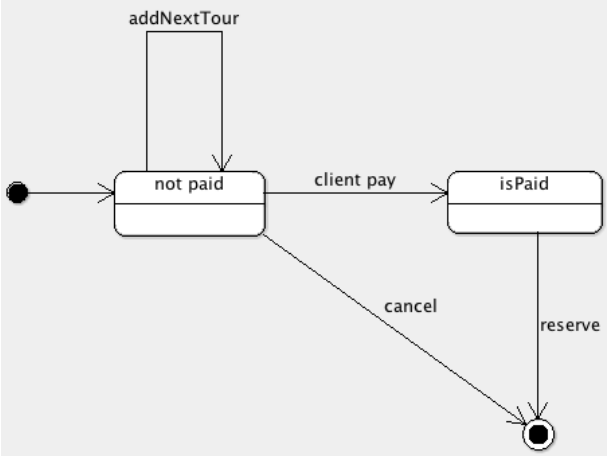

Figure 4: State diagram

After created, the reservation of touring travel have two middle states: *isPaid*, and *notPaid*. When in *notPaid*, we can add as many more tour as we can, or if we cancel the reservation, it will fall into final state. When the client pay for the reservation, object's state will change into *isPaid*, and no more modification could be done. And when the reservation for flight, hotel, car were done, it will come to final state.

4 CREATIONAL PATTERNS

4.1 Singleton

Singleton pattern assure that the number of objects of the classes implementing this pattern are limited to one. The reason for this restriction is that, the object of this class would hold the global resource, and the creation of new object could be meaningless.

I have implemented this pattern in a lot of classes. Some of them are CityFactory, FlightFactory, CheapHotelStrategy, PoolTicketStrategy, TravelBuilder.

Below is a piece of code from class TravelBuilder, which implement Singleton

```
private static TravelBuilder instance;

private TravelBuilder() {

}

public static TravelBuilder getInstance() {
   if (instance == null)
      instance = new TravelBuilder();
   return instance;
}
```

The constructor is set to private, so no other class can use this constructor to create new object of this class. Instead, we could call the static function *getInstance* to get the only instance of TravelBuilder.

4.2 Abstract Factory

Abstract Factory pattern is used, when there are many factories of related objects. In this case, FlightFactory, CityFactory, HotelFactory are the factories of Flight, City, Hotel.

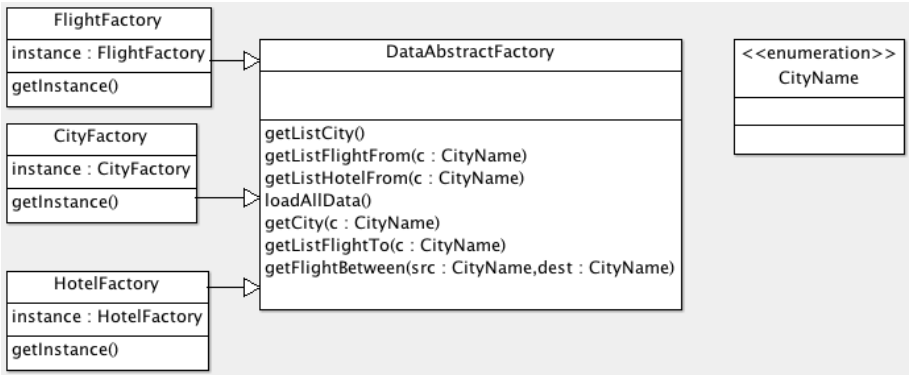

Figure 5: Class diagram of DataAbstractFactory

An example code of how to use these factories

```
DataAbstractFactory f = CityFactory.getInstance();
f.loadAllData();
List<City> cities = f.getListCity();
for (City c : cities) {
   System.out.println(c.printCityDetail());
}
```

4.3 Builder

Builder pattern is used to construct an object, sometimes could have complex structure, and would be approached using step by step method. In this case, the most complicated structures are the reservation classes.

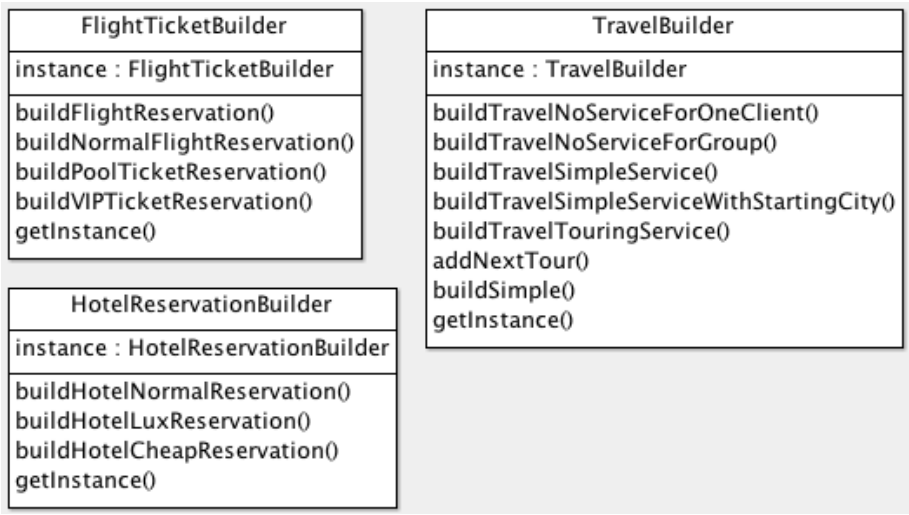

Figure 6: Class diagram of the builders

An example code showing the construction of a travel with touring service

```
Client group = new GroupClient(list);
Travel t = TravelBuilder.getInstance().buildTravelTouringService(group,
    CityName.Paris, FlightTicketType.VIP, HotelServiceType.Lux);

TravelBuilder.getInstance().addNextTour((TravelTouringService) t,
    CityName.Bordeaux, FlightTicketType.Normal, HotelServiceType.Cheap);

TravelBuilder.getInstance().addNextTour((TravelTouringService) t,
    CityName.Tokyo, FlightTicketType.VIP, HotelServiceType.Normal);
```

5 STRUCTURAL PATTERNS

5.1 Proxy

The purpose of Proxy pattern in our project is for hiding the real object. It will add security access to an original object. Any access to a real object would be examined by the proxy, and the proxy will decide if it's safe to delegate the request to the original object.

There're three proxy in our program :

- ProxyClient
- ProxyFlightTicket
- ProxyHotelReservation

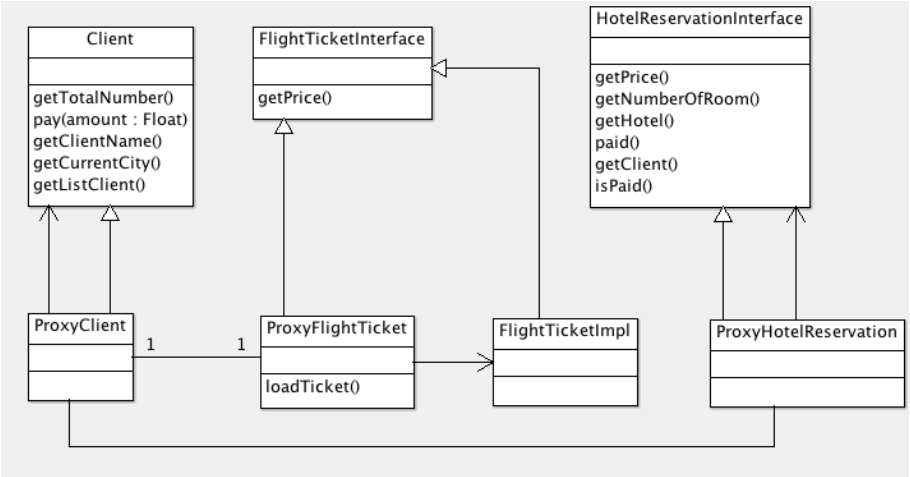

Figure 7: Class diagram of proxies

5.2 Composite

Composite pattern is used to describe a group of obejcts that could be treated almost as the same way as a single one. There are two composite in our program.

- Client: as clients could booking the services as a group, a group of clients should be considered like one client.
- Travel with services: a travel package that consist of many trips to many cities, is still a travel.

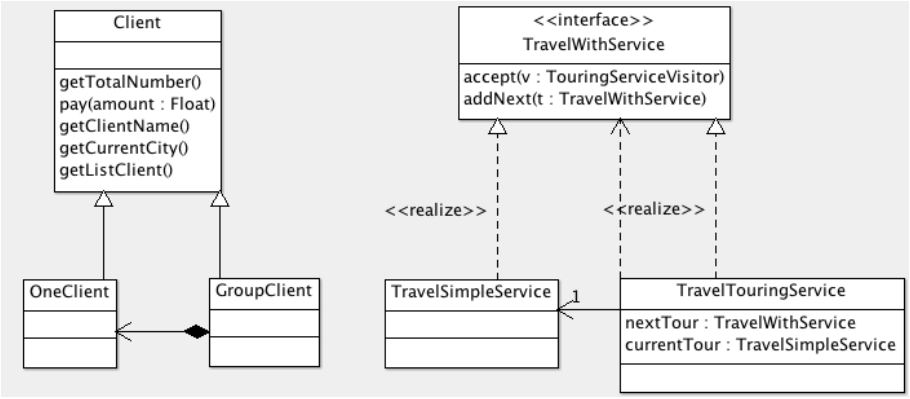

Figure 8: Class diagram of composite patterns

The implementation of composite pattern could be varied to adapt to the programer's intention. In this program's context, the *GroupClient* consists of a list of *OneClient*. In case of travel pattern, *TravelTouringService*, a travel that go through many cities, use a recurrence relation that point to one other object of the same class.

5.3 Decorator

Decorator is used to alter a behavior of the instances of a class, without changing the other behaviors or data of the same instance. We used decorator patterns in this project, is to define explicit the definitions of the kinds of flight reservation and hotel reservation

- *PoolTicketDecorator, VIPTicketDecorator*
- *HotelCheapDecorator, HotelLuxDecorator*

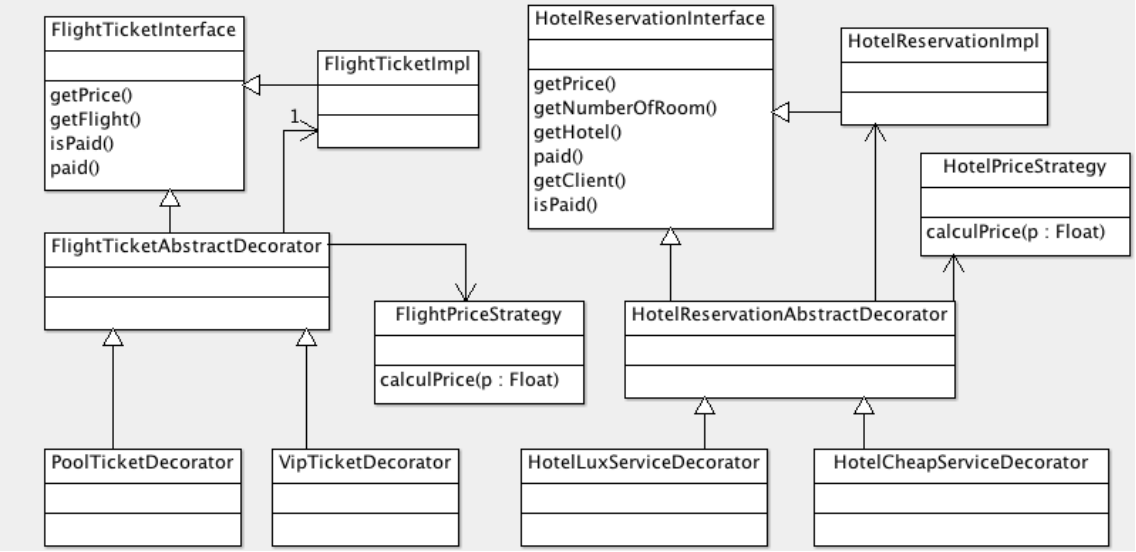

Figure 9: Class diagram of decorator patterns

The only different behavior between those decorators is the way they calculate the price of a service that they defined. More accurately, this different is taken care of due to the change of the strategy object that the abstract decorator hold onto.

```
public HotelCheapDecorator(HotelReservationImpl h) {
   this.reservation = h;
   this.strategy = CheapHotelStrategy.getInstance();
}

public HotelLuxDecorator(HotelReservationInterface h) {
   this.reservation = h;
   this.strategy = LuxHotelStrategy.getInstance();
}
```

6 BEHAVIORAL PATTERNS

6.1 Strategy

Strategy pattern permit the definitions of many algorithms which served the same purpose. In this case, we used strategy patterns to define the way of calculating the price of the reservations. As mentioned in the previous section, we have implemented four decorators, two for flight reservation, two for hotel reservation. There're also four strategy classes corresponding to the decorators.

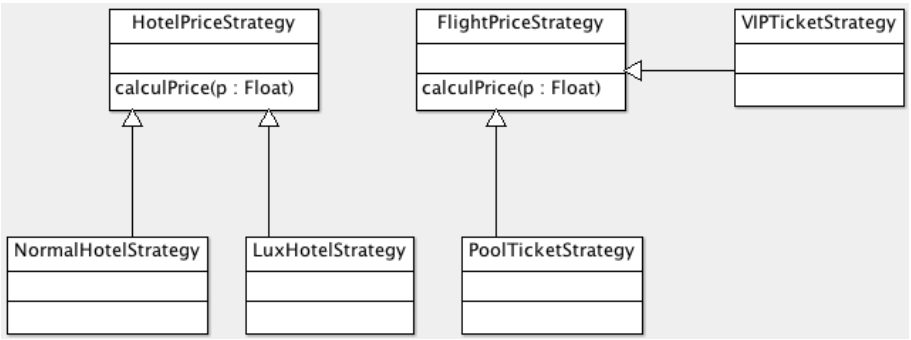

Figure 10: Class diagram of strategies

Take an example of flight reservation's price's calculation. The method *calculPrice* from two strategies are

```
public class CheapHotelStrategy implements HotelPriceStrategy {
...
   @Override
   public float calculPrice(float p) {
      // TODO Auto-generated method stub
      return p * 80f / 100f;
   }
...
}

public class LuxHotelStrategy implements HotelPriceStrategy {
...
   @Override
   public float calculPrice(float p) {
      // TODO Auto-generated method stub
      return p * 120f / 100f;
   }
...
}
```

6.2 Observer

Observer pattern allow the objects, called observers, to be notified when there's a state change in the objects that were observed. In the context of this project, two observer classes were implemented to observe the number of available place in a flight and the number of free rooms in an hotel. Whenever these numbers reach zero, means that there are no more available space, the observers will display a message to notify the agent.

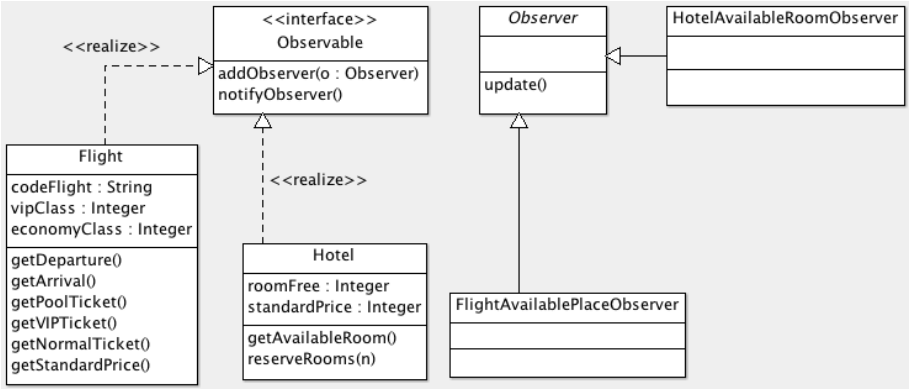

Figure 11: Class diagram of strategies

To implement this pattern, interface *Observable* and abstract class *Observer* are created. The classes that need to be observed will implement *Observable*. After that, they should decide when to notify the observers. Take an example of class *Flight*, in the code below we demonstrate the situation when the VIP's place is running out

```
public void secureVIPPlace() {
   if (vipTickets >= 1) {
      vipTickets--;
      if (vipTickets == 0) {
         notifyObservers(FlightTicketType.VIP);
      }
   }
}
```

6.3 Iterator

Iterator pattern allow the definition of an object, call *iterator*, which can traverse the element objects of a container. There are two composite classes which could consider as a container: *Client* and *TravelTouringService*. We choose *TravelTouringService* to deploy the iterator pattern.

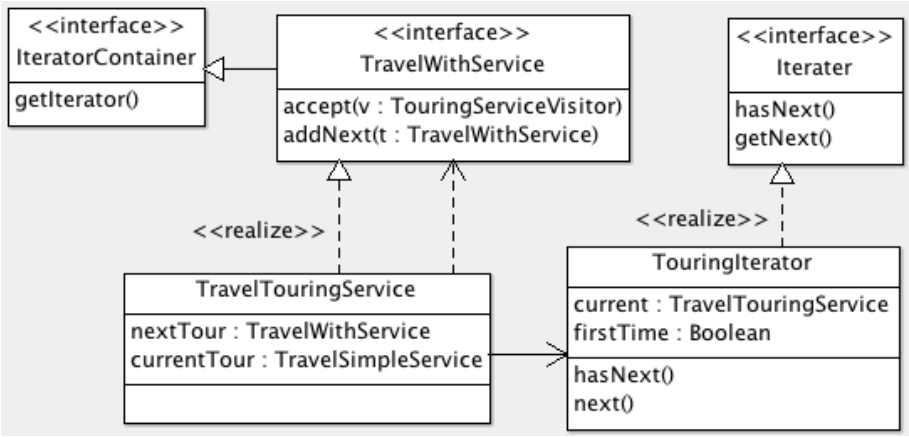

Figure 12: Class diagram of iterator pattern

I implemented *TouringIterator* as the inner class of *TravelTouringService*. With this arrangement, *TouringIterator* could freely access to *nextTour*, an attribute that was caching by *TravelTouringService*. *TouringIterator* will return *currentTour* and process to *nextTour* each times the method *getNext* is called.

```
public Object getNext() {
   if (hasNext()) {
      if (firstTime) {
         iter = current;
         firstTime = false;
      } else {
         iter = (TravelTouringService) iter.nextTour;
      }
      return iter.firstTour;
   }
   return null;
}
```

6.4 Visitor

Visitor pattern permit the addition of operations into one object without altering the structure of its class. The travel with services consist of two kinds of services that have multi-type: flight reservation and hotel reservation. To keep track of each service's type, like how many VIP flight ticket that the client had booked, or how many hotel's luxury services that the client had ordered, a visitor is created to manage these number.

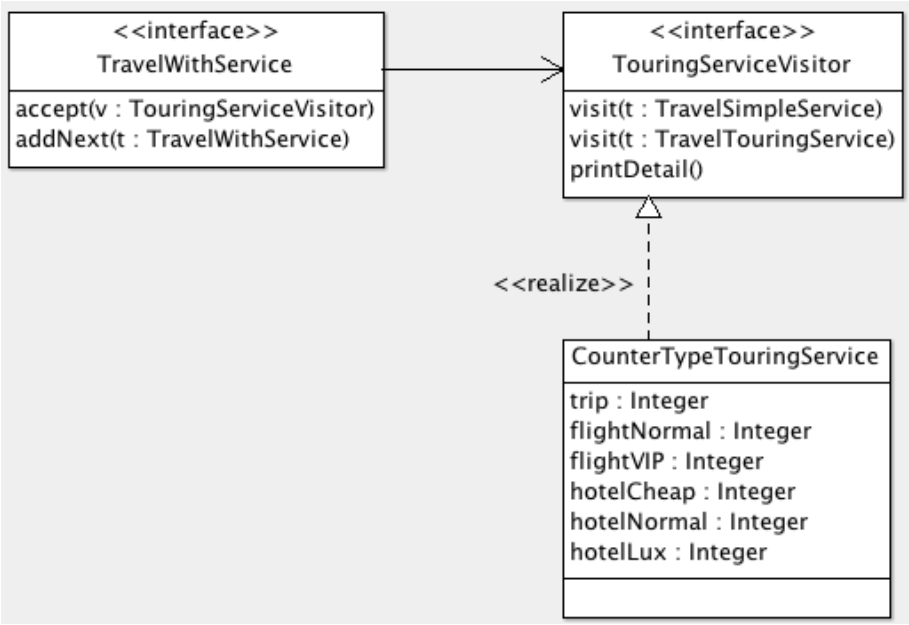

Figure 13: Class diagram of visitor pattern

At first, in *TravelTouringService*, method *accept* is where the visitor could start visit his targets.

```
@Override
public void accept(TouringServiceVisitor v) {
   v.visit(firstTour);
   if (nextTour != null)
      nextTour.accept(v);
}
```

Next, in the visitor class, *CounterTypeTouringService*, in method *visit*, the number of service's type is managed. For example, flight service's type.

```
FlightTicketType f = travel.getFlightTicketType();
switch (f) {
   case Normal:
      flightNormal++;
      break;
   case Pool:
      flightNormal++;
      break;
   case VIP:
      flightVip++;
      break;
   default:
      break;
}
```

6.5 Template

Template pattern defines the way to run the operator in the class. In *TravelSimpleService* and *TravelTouringService*, we have three methods that do the reservation: *flightReserve*, *hotelReserve* and *carReserve*. We want that the reservation must follow the order: flight, hotel then car.

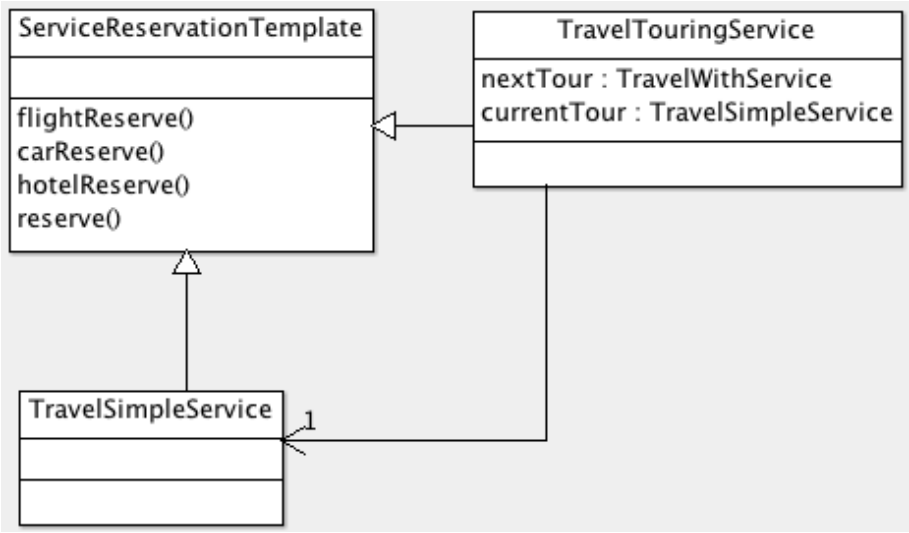

Figure 14: Class diagram of template pattern

In *ServiceReservationTemplate*, only one method *reserve* is implemented, this one method has defined the order of execution of the three reservations.

```
public abstract class ServiceReservationTemplate {
   public abstract void hotelReserve();
   public abstract void carReserve();
   public abstract void flightReserve();
   public void reserve(){
      flightReserve();
      hotelReserve();
      carReserve();
   }
}
```

7 CONCLUSION

After this project, I have obtained many knowledge, first of all, is the usage of many design patterns and the way to implement them. Secondly, I have learned about the flexibility when apply the design pattern, there are many way to implement a design pattern. You should choose a way that suit the situation the best rather than just mimicking the sample code. Lastly, I have learned the way to use Latex more efficiently in making a report