SWEN30006 Software Modelling and Design

Semester 1   2017

Project Part B
Design analysis Report

Group Number: 8

Group Members:

Name	Student ID	Login Name	Email
Jiru Sang	734508	jsang	jsang@student.unimelb.edu.au
Jiaying Li	734507	jiayingl3	jiayingl3@student.unimelb.edu.au
Yue Chen	734946	yuec6	yuec6@ student.unimelb.edu.au

Section A The advantage of the design

Controller
For every system, we need to know who should be responsible for handling the input system event outside the system. It worth noting that a controller should be a non-user interface object which is responsible for handling and receiving system events. In other words, it should define the method for the system operation.

In the domain layer, the first object beyond the UI layer which is responsible for receiving or handling system operations is the simulation class. For the Metro Madness program, the UI is related to the Metro Madness-desktop. All the information of train, line and station are extracted by MapReader and then transfer to the MatroMadness class. In the meanwhile, MatroMadness instance create the simulation instance to update all the trains in the simulation.

The controller used in the program increased potential for reuse, and pluggable interfaces. Moreover, it is easy for user to reason about the state of the use case. The controller, the responsibilities in the controller and the attributes in the controller are all clear and suitable for this system.

Creator
MetroMadness class is the creator for all the train, line and station. Since MetroMadness reads the all information read from XML, which means MetroMadness has the initializing data that will be passed to class train, line and station when they are created.

Information expert

To get the information of the next track and next station, class line is the information expert because the class has the arraylist of the station and track. The design is correct and with low coupling because they assign the responsibility(nextTrack() and nextStation()) to the class Line(information expert), which is the class that has the information to fulfill the responsibility.

Low coupling and high cohesion
Instead of using conditionals to select the alternative of DualTracks or other tracks, the design uses the polymorphic by giving a single interface to entities of different types of tracks, which makes the system suitable and easy to reuse and extend later, in the meantime, also reduce the coupling.

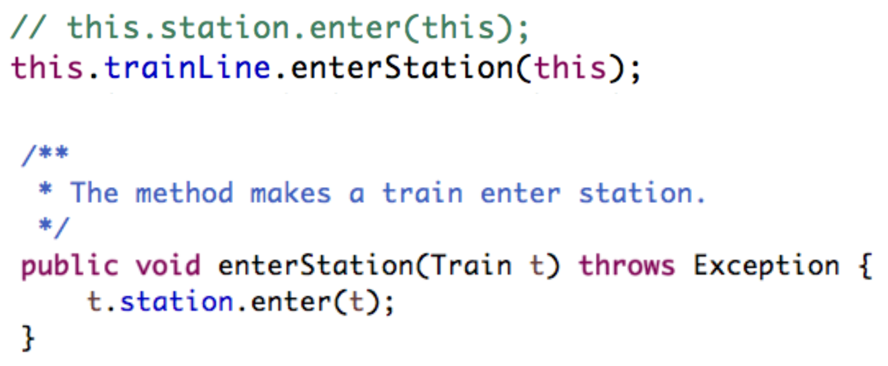
Section B The disadvantage of the design

Problem 1: Coupling and Cohesion

Based on the design class diagram, the class Simulation get all stations, lines and trains through the class MapReader, which can read the train maps from XML file. According to the context of this project, each line is composed of stations and tracks. In other words, the relationship between the class Line and two classes—Track and Station, is composite aggregation. Due to this reason, the class Line has the necessary information of stations and tracks. Meanwhile, the class Line should play an important role as information expert which has responsibilities to perform functions. Moreover, the class Train is also very important which keeps the simulation going. This class has key attributes—trainLine, passengers, station, track and state. In the existing implementation, the train has five states which are FROM_DEPOT, IN_STATION, READY_DEPART, ON_ROUTE and WAITING_ENTRY. When updating its states, each train will keep and update its current station and track, and then call functions in the class Station and Track directly, such as enter(Station station) and getDepartureTime().

Disadvantage:
Based on above analysis, this design leads to the problem of high coupling because that the class Line and Train must be both changed corresponsively when the class Station or Track has changed. Due to the increased dependency between classes, the existing design is harder to comprehend, maintain and reuse in the future.

Solution:
To solve this issue, the class Line should perform as information expert by adding functions that call functions in the class Station and Track. The class Train can call functions in the class Line to get information of stations and tracks. When the class Station or Track is changed, the class Train will not be impacted anymore.



```
// this.station.enter(this);
this.trainLine.enterStation(this);
```

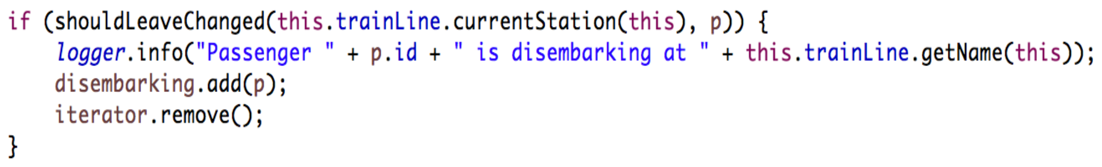
```
/**
 * The method makes a train enter station.
 */
public void enterStation(Train t) throws Exception {
    t.station.enter(t);
}
```

Graph 1

Advantage after redesign:
By modifying existing design, this project has been improved with high cohesion and low coupling. A change in the class Station or Track will not force an effect of changes in class Train that require less effort and time due to decreased dependency between these classes.

Problem 2:   Information Expert

According to the responsibility of train, when a train is in IN_SATION state, it will ask some passengers disembark from the train and wait for the incoming passengers. When deciding which passenger should leave the train, the existing design always call the shouldLeave () function in Station class to check all passengers. Meanwhile, the shouldLeave () method in station is implemented by the shouldLeave () method in PassengerRouter class. Therefore, the information includes current station and destination, which decides if a passenger should leave will be passed to the PassengerRouter class and judged by its shouldLeave () method.



```
if (shouldLeaveChanged(this.trainLine.currentStation(this), p)) {
    logger.info("Passenger " + p.id + " is disembarking at " + this.trainLine.getName(this));
    disembarking.add(p);
    iterator.remove();
}
```

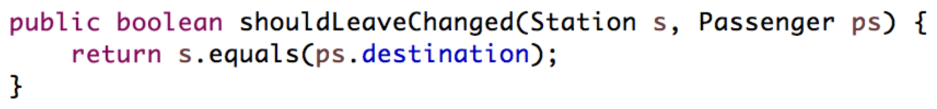
Graph 2

Disadvantage:

The existing design above violates the elemental principles of GRASP, which is information expert principle. Based on the analysis above, the information which decides if a passenger should leave the train can be found in train class, however, it is processed in PassengerRouter class. PassengerRouter isn't an information expert and it will increase the dependency between Train, Station and PassengerRouter class. Then it is hard to comprehend the class design.

Solution:

As discussion above, the information expert is Train class. It knows the current station and the destination station of all passengers. Then changing the shouldLeave () implementation in train class can solve the problem above. At the same time, the PassengerRouter class can be retained to simulate the future requirements of multi-line journeys.



```
public boolean shouldLeaveChanged(Station s, Passenger ps) {
    return s.equals(ps.destination);
}
```

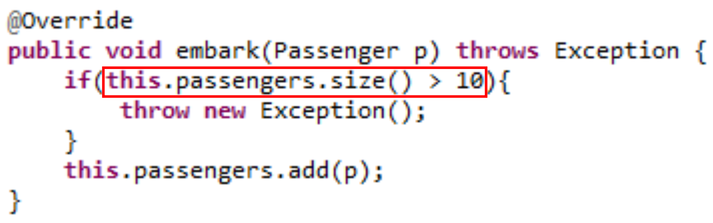
Graph 3

Advantage after redesign:

By modifying the existing design, the real information expert has made full application and assignment. Therefore, it decreases the dependency between classes to make a lower coupling class design.

Problem 3: Polymorphism

The existing simulation only allows one of two types of trains (10 passengers and 80 passengers), which does not allow the system to vary the train sizes to simulate the behaviors of passengers on Metro Melbourne with varying train sizes. There is no suitable member variable in the train type for the train size. For instance, one of the methods of the train, embark(), needs to calculate whether the current passengers on the train is less than the maximum passenger capacity size of the train type. Each time when a new train subclass succeeds the train class, the subclass needs to overrides the function everytime. The model now is not quite extensible and not makes full use of the polymorphism principle mentioned in GRASP2.

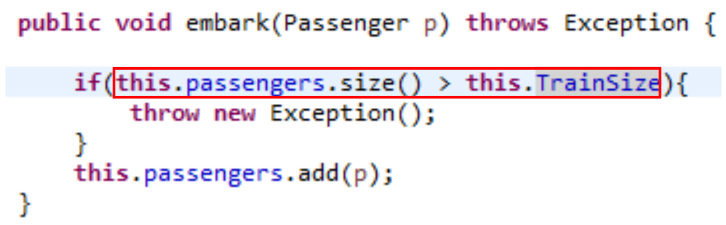


```
@Override
public void embark(Passenger p) throws Exception {
    if(this.passengers.size() > 10){
        throw new Exception();
    }
    this.passengers.add(p);
}
```

Graph 4

Disadvantage:
If we create different sizes of the train, we need to override and modified the embark() function every time, which makes it difficult to extend the program and the system.

Solution:
Make full use of the advantage of the polymorphism. When the related alternatives vary by type, we can use operations with single interface to the entities of different types. After redesigning the class design, we add a member variable, train size, and modified the embark() function in the train class (the super class).



```
public void embark(Passenger p) throws Exception {

    if(this.passengers.size() > this.TrainSize){
        throw new Exception();
    }
    this.passengers.add(p);
}
```

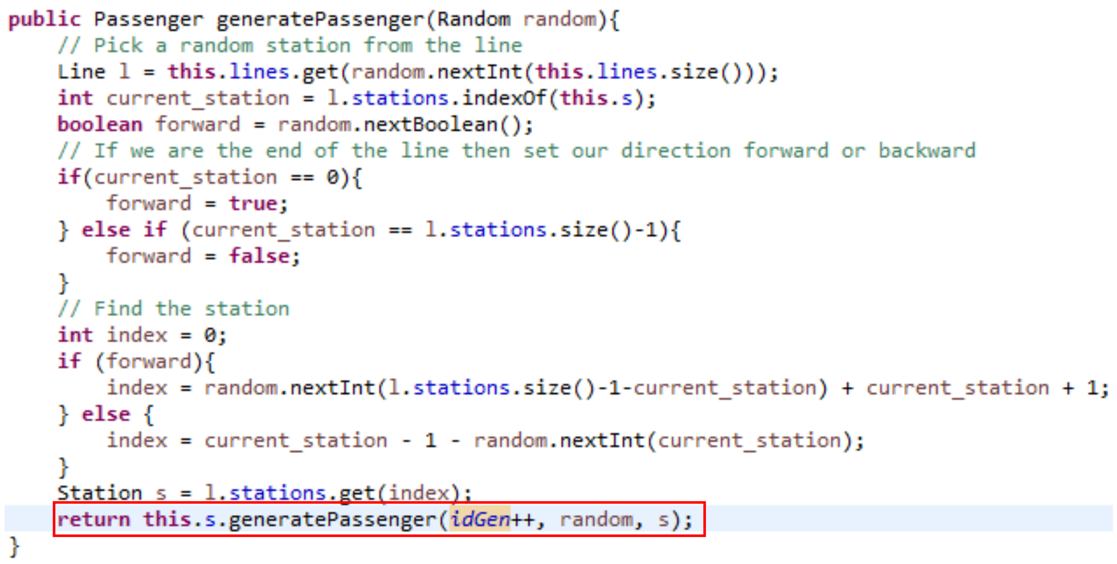
Graph 5

Advantage after redesign:

1. Decrease the coupling.
2. New variations can be added easily into the system.

Problem 4: Creator, Information Expert, Cohesion and Coupling

In the shoddy class design diagram, it is the Station class's responsibility to create a new instance of passengers. However, in the PassengerGenerator class, the function generatePassenger(Random random) needs to call the function, generatePassenger(int id, Random random, Station s), which is in the Station class.



```java
public Passenger generatePassenger(Random random){
    // Pick a random station from the line
    Line l = this.lines.get(random.nextInt(this.lines.size()));
    int current_station = l.stations.indexOf(this.s);
    boolean forward = random.nextBoolean();
    // If we are the end of the line then set our direction forward or backward
    if(current_station == 0){
        forward = true;
    } else if (current_station == l.stations.size()-1){
        forward = false;
    }
    // Find the station
    int index = 0;
    if (forward){
        index = random.nextInt(l.stations.size()-1-current_station) + current_station + 1;
    } else {
        index = current_station - 1 - random.nextInt(current_station);
    }
    Station s = l.stations.get(index);
    return this.s.generatePassenger(idGen++, random, s);
}
```

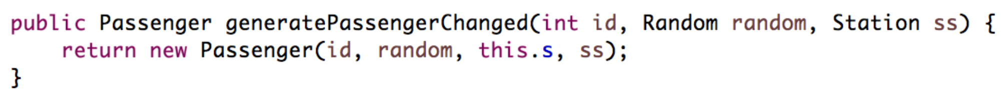
Graph 6

Disadvantage:

1. The design will result in low cohesion because the passenger related function is not strongly related and focused in one class. On the contrary, they are separated into different classes, which makes it hard to reuse, maintain and might be effected by change easily.
2. High coupling. The crossing call in the two class, Station and PassengerGenerator, increases intensity level of the dependency, which means changes in Station classes will force the local changes in the PassengerGenerator class. Moreover, it will be harder to understand in isolation and reuse.
3. The old version also violates the principle Expert information and creator. To create the passenger, PassengerGenerator class already knows all the related and needed information. PassengerGenerator ought to be the expert and creator.

Solution:

Since the PassengerGenerator class has already known the information of the station, we can move the function generatePassenger() into the PassengerGenerator class so as to reduce the cross call between these two classes.



```java
public Passenger generatePassengerChanged(int id, Random random, Station ss) {
    return new Passenger(id, random, this.s, ss);
}
```

Advantage after redesign:

1. Promise low coupling because the information encapsulation can be maintained easily. The instance only use their own information to complete the tasks.
2. Promise high cohesion and easy to understand and maintain. The function

generatePassenger () is more related and focused with the functions in the PassengerGenerator class.

3. Easy to maintain and resues.