Design analysis report
Group 97

Wuang Shen 716090
Ruifeng Luo 686141
Ivan Ken Weng Chee 736901

Introduction:
The objective of the report is to analyse the design on the provided simulation. According to their design, the provided system is hard to reuse, extend and maintain, even though the existing system have functioned as metro train simulation. Based on the GRASP patterns along with general object oriented code quality, the design analysis of the provided simulation along with their improvements is described below.

Analysis:
1. Train, Line, Track, Passenger and Station classes have high potential to have their derivatives in the future. In the provided design, however, most methods and attributes are designed to fit a specific circumstance without allowing for the **Polymorphism pattern**. Therefore, it's hard to **extend** those classes to their new type subclass, and it will also result in a large chunks of **code duplication** if trying to create new subclass by modifying those parent classes. Since these classes share the common behaviours, they could be implemented as abstract class so that those class could be more **extensible**. By doing so, it will also help to protect variables' **visibility** by implementing abstract class as stable interface, which will prevent elements' instability from having undesirable effects upon other elements in the system and deviation in the later coding.
By implementing those interfaces, all variation points and evolution points are easy to be created and under control of the system.

2. In the Train class, embark () method should be abstracted and be override in the subclass of its abstract class to support **Polymorphism** since the embarking behavior of the train varies with the type of the train. Besides, without abstracting the embark() method developers are more likely to neglect to override the embark(), then the behavior of the train may not match with the requirement. There is a chance that the system still can run without error warring even if the train does not run as what clients required, so it's dangerous to create the train class without making the embark() method abstracted. Similarly, enter() method in the station class is expected to be override as well, as different types of stations are expected to have different enter() method.

3. Some variables in the Train and Station are set to Static final making these classes less **extensible** to different types of trains and different stations. Once the variable set to static final, there's no way to alter in any other cases. It will be a problem when the system requires different types of the Train and Station, because part of their attributes are static and can't be changed. To support **polymorphism**, those static final variables should be attributes placed in the subclass of Train serving different types of station.

4. In the system, public visibility modifiers are heavily used throughout class variables instead of getters and setters, which defeats the purpose of class encapsulation. It violates the rule of **open-closed principle** which means values of attributes have a chance to be modified by anyone who can access this system. In this case, the system should set public attributes to private and use getter and setter methods to access them, which makes the system security and complete. According to **general object oriented code quality**, using getter methods would keep the data of the system security and using setter methods would limit the scope of accessing to the system.

5. Station class is combined with two types of responsibility, giving information about the station and operating the station. So it is **low cohesion** as the station class is taking responsibilities which can be distributed into two classes. Low cohesion makes the system **hard to comprehend, reuse, maintain and delicate**. Therefore, it will be much better to apply **pure fabrication** here to assign the station-operating responsibility (like methods called canEnter, registerLine, shouldLeave and etc.) to another class that is not in the problem domain to achieve **high cohesion and reusability** for the station class.

6. The Train class scores pretty low in terms of **cohesion**, as it is not very well-defined, and somewhat bloated. It has many dependencies with other classes, as can be seen in storing Line, Station, and Track as variables. Aside from that, the behaviour of this class is defined by the different train states, which unfortunately is stored as an enumeration variable within the class, and handled via large conditional statements. This impacts the maintainability and reusability of this class, as monolithic code blocks need to be added to cater for additional states if necessary. Designing a new abstracted State class to support **polymorphic pattern** can be used to resolve this issue, as the varying behaviour of the class can be encapsulated based on its internal state, hence resulting in cleaner code.

7.Accodring to the provided UML diagram, train class is **highly coupling** with Station class. In this case, it will increase the difficulty to comprehend the design of the system in the later coding. Thus, the high coupling between Station class and train class will make it harder to extend, maintain and reuse. To support a low coupling design, Line class as an intermediate class between Train class and Station class is expected to take the place of station's responsibility to control Train class to reduce the coupling between Train and Station. Using HashMap in the Line class to map Train and Station will be a potential method to implement the association between Train and Station through Line class.

8. From a design perspective, the creation of the Passenger class is handled poorly. The PassengerGenerator should be solely responsible for generating Passenger objects, as opposed to a call to the generatePassenger method under Station. As the PassengerGenerator randomly generates a passenger's destination and has access to the active station, it is redundant to rely on Station for creation. This defies both the **Creator** and **Information Expert** principles under GRASP. Another side effect of this design is an increase in coupling between these three classes. A more suitable approach would be to simply return a new instance of Passenger in PassengerGenerator, which is handled by Simulation instead of Station. A dependency between PassengerGenerator and Station should also be removed.

9.It can be seen that most classes like train, station and line in the design have their own render method. It will be hard to maintain or change the code once the system is expected to use different render method, as developers have to change every render method in those classes. Therefore, it will be better assign the responsibility to a single class called View to render all objects rather than every object render themselves, which makes the design more clean and easy to maintain.

10. With regards to general object oriented principles, classes such as Train, Track, and Station have some methods do not relate to the concept of the class itself, which disobeys the job delegation principle. In addition, a number of many-to-many relationships exist, such as Train-Station and Line-Station. Inheritance is applied in classes such as Train, Track and Station, although parent classes are being initialised, instead of being abstract. Constants are used in a well manner, although its use can be improved as there are a number of magic numbers present throughout the project. Cargo is created as an inner class under Passenger, which might not preserve code robustness, as the details of Passenger can be accessed by Cargo and vice versa.