Software Modelling and Design Project B
Analysis Report

Kaven Peng 696573
Li Lin 686458
Wenlai Song 641305

<u>General Object Oriented Code Quality Analysis</u>

1.Visibility modifiers are all public.
Bad OOD(Object Oriented Design) because:

- Class data is not being encapsulated and hidden properly from the user.
- Makes it easier for coupling between classes to exist since data is visible.
- Doesn't protect data from accidental or wrong usage/modification.
- Creates problems since data can be accessed without a clean interface.

2.Many confusing/unclear/repeated variable names.
E.g. this.s.generatePassenger(s), each s points to different stations.
Bad OOD(Object Oriented Design) because:

- Creates confusion about what the variable actually stores.
- Makes code unnecessarily difficult to read and understand.

3.Some variables initialized yet never used.
E.g. filename variable in MapReader is never used because the file being read is hardcoded into the function.
Bad OOD(Object Oriented Design) because:

- There is no reason to store variables that are never used.
- Hardcoding static values for variables like filenames is bad since it hinders the flexibility of the program.

4.Lack of useful comments and documents.
Bad OOD(Object Oriented Design) because:

- Makes code difficult to read and understand.

5.Repeated function names in separate classes.
E.g. generatePassenger function in PassengerGenerator and Station class.
Bad OOD(Object Oriented Design) because:

- Causes confusion regarding which function is actually being used.

6.Use of Magic Numbers.
E.g. 10 for small train size in embark function.

- Loss of clarity

7.Ineffective subclasses
E.g. Only difference between BigPassengerTrain and SmallPassengerTrain is a single magic number representing their relative size.

- Not very useful as a separate class.
- Could be done more easily and efficiently with constructor arguments.

8.Nondescriptive Exceptions Thrown
E.g. Exceptions thrown when trying to add passenger to a full train is just a regular Exception.

- Unhelpful for debugging code
- Makes code harder to understand when throwing generic exceptions.

9.Try Catch used improperly
E.g. embark function uses a try catch in place of a conditional

- No reason to use try/catch instead of conditional. Possible errors are non-exceptional.
- Try/Catch is much more expensive to use than conditional.

Design Pattern Analysis

Strategy Pattern is used with the PassengerRouter class. Using this pattern we can implement multiple routing strategies depending on the situation we need to simulate. I agree with using this design pattern here.

GRASP Principles Analysis

1.MapReader
MapReader is a fairly high cohesion, high coupling class that follows the Creator principle since it has the initializing data passed into it from the xml file and also processes and stores the Lines, Stations and Trains. It connects with all of the other classes that need the initial data from the xml files to create their respective classes.

2.MetroMadness
MetroMadness is using the Controller principle to handle input and initialize/start/render the simulation. This is determined from the functions included in MetroMadness, namely the create(), handleInput() and render() functions.

3.Simulation
High Cohesion, Low coupling class that follows Information Expert. Simulation class has all the information on all the elements in the simulation, so it follows that updating and rendering the entire simulation’s responsibility is given to this class.

4.PassengerGenerator
Noticeably higher coupling than previously analyzed classes. Is coupled with Station as well as Line, also stores an arraylist of lines. Given that Station also contains an arraylist of lines, an easy way to reduce the amount of coupling would be to remove the attribute of stored lines in PassengerGenerator and instead pass in the lines using a getter in Station. This class is high cohesion because it mainly sticks to its intended use of creating random passengers and nothing else.

5.SimpleRouter(Concrete Strategy Implementation)/PassengerRouter(Interface)
As stated before, this class follows the Strategy design pattern. Currently it is fairly lightweight with only the function shouldLeave(Station s, Passenger p) creating a coupling between the Router, Station and Passenger. Cohesion is high, coupling is fairly low, but could be better if implemented in Passenger instead of in Station as it is currently.

6.ActiveStation(Subclass)/Station(Superclass)
There is naturally high coupling between the subclass and superclass given how inheritance works. ActiveStation is highly dependent on Station. On top of that Station itself is high coupled with Line, Train, PassengerRouter and PassengerGenerator, with generatePassenger and shouldLeave functions contributing to a lower cohesion.

It follows the Information Expert principle quite closely because it contains all the information necessary for all it’s functions, but I would consider delegating some functions such as generatePassenger and shouldLeave elsewhere to lower cohesion at the cost of higher coupling, as other functions also contain the necessary information to be responsible for them.

However analyzing Station using Creator principle tells me that all of the functions should be the responsibility of Station because it aggregates and contains both trains and passengers, while also working with them closely. So in this case using Creator would provide lower coupling at the cost of lower cohesion, while Information Expert would provide higher cohesion at the cost of higher coupling. These tradeoffs will have to be addressed in the redesign portion of this project.

7.Track(Superclass)/DualTrack(Subclass)
Somewhat violates Information Expert principles because the track should not have any information on the train on the track. According to Information Expert the Train should contain the enter and leave functions instead since it’s natural for the train to have information on the track that it is currently on. Since it violates Information Expert this causes the class to have higher coupling and lower cohesion than it should for a simple boolean storage class.

8.Line
Good use of Information Expert, contains only functions that relate to the information stored in the line, which would be all the tracks and stations on a line. High cohesion, with fairly low coupling for such an integral class.

Also shows good use of Creator since it contains the addStation function that also creates tracks. Line fulfills most of the prerequisites for a Creator, aggregates, stores, contains and records all of the stations and tracks as well as using them to check for EndOfLine, nextTrack and nextStation.

9.Train(Superclass)/BigPassengerTrain(Subclass)/SmallPassengerTrain(Subclass)
Easily the class with highest coupling in the simulation because everything revolves around the train class. Train class is connected with Station, Track, Line and Passenger.

Violates Information Expert with one function embark(Passenger), the train should have no knowledge at all of passengers waiting to embark on it at the station/next station, so it shouldn't have responsibility of the embark function.

There's also no reason to increase coupling by having 2 train subclasses that essentially have the same function but only differ in passenger limit. Creating a size attribute and having a constructor that takes a size argument or train type would lower coupling by a large amount.

Overall Conclusion
Most of the time the current design adheres to the GRASP patterns and makes logical design decisions. However the coding style and readability is quite atrocious, I know because it gave me a few headaches trying to understand all the code.

There are definitely some questionable design choices though such as Station/Active Station class design, why have a station where people cannot disembark?

Redesign Priorities
Our redesign will focus on making the code more readable and flexible, i.e fixing visibility modifiers, adding comments/documentation, throwing named exceptions and adding getters/setters. We will add a RETURN_DEPOT state to the train so it will behave more realistically. We will also improve on the PassengerGenerator and PassengerRouter algorithms to provide multi-line functionality. Also high on our agenda is to allow disembarking at all stations and cleaning up Train types to support variable train sizes.

Redesign Implementations

Redesigned Router: The main routing is done in passenger generator class which uses DFS to recursively generate a route and stored in each passenger. The new router's job is to just follow the route stored in passenger and tell them whether to get on or get off the train at each station.

Redesigned Train:
Train now has one more state which is called TO_DEPOT. The old train class had problems handling situations where the station reaches maximum capacity and the track is occupied, so the train cannot leave the station and therefore gets stuck. Now after implementing this state situations like these are avoided since the train can be put back into depot. Now simulation also has the ability to set max number of trips for a train and actually put them back into depot again. Currently trains automatically stops at the end of line when they reach maximum number of trips and then wait for few seconds, they will then go back to station and continue their run. This is similar to real life situation since trains stop at the end of their trip and rest.

Other changes:
Getters and setters are added for most classes to improve on OOD principles for better encapsulation.
BigTrain and SmallTrain classes are removed to allow for flexibility in setting train sizes.