Daewin, 679182 (dsv)
Julian Chua, 677491 (weec1)
Rahul Singh, 696272 (rahuls1)

This design analysis is for the existing implementation of the train network simulation. In general, there are quite a few downfalls in terms of code quality and not following good design principles. One of the biggest errors in the simulation as a whole is the lack of data encapsulation, one of the four fundamental principles of OO. Many of the attributes and method are kept with public visibility modifiers, which could lead to outside interference and misuse of data. The code is also at times quite unclear due to the fact that the code is not well documented and the naming of variables and methods are not informative enough. This might also be one of the reasons why the logic of some classes contain a multitude of design errors which will be discussed below:

MetroMadness.java

The metro madness class is an initializer class that creates the simulation, renders the graphics for it and handles user input. It also renders the simulation object and updates both the camera and simulation itself. Due to it being the initializer class, it makes good use of the expert pattern as it has the required information needed to instantiate the simulation. Also due to this, it is very cohesive as it only needs to do its own responsibilities that is create, render and handle the input. In terms of code quality it does have some flaws, for example, 'int VIEWPORT_WIDTH = 200' should be a public static final integer as it is a constant. Furthermore, there is a variable with the same name but not capitalized and a float type. Though this class has successfully been able to delegate tasks to its methods. It also could be said that this class is highly coupled because of the larger number of objects it is dependent upon such as the ShapeRenderer, OrthographicCamera, and BitmapFont.

Simulation.java

The simulation class can be thought as being the driver for the program because it updates trains and renders the lines, trains and stations. It is considered to be a controller class because when created, it takes in the data file as an argument of the constructor allowing the creation of MapReader to process the file and further resulting simulation to have the list of all station, lines and trains. Besides being well organized, having good abstraction and is well delegated, the simulation class is said to be an information expert because it has all the information needed to perform the updates of the trains and the rendering of the lines and tracks as well as the trains. Although coupled with MapReader, the Simulation class has low coupling because other classes would not affect the simulation class and simulation class does not heavily depend on the other classes. In addition, it has high cohesion due to being well delegated, meaning it uses other classes methods to perform the appropriate task instead of the simulation class itself.

MapReader.java

MapReader class reads the provided XML file then processes and stores the information. The three main pieces of data read in are the stations, lines and trains. This follows the expert pattern principle as it has the information required to fulfill the responsibilities need. A key problem with this class is that a file name is supplied in the constructor, yet MapReader reads in a separate file that is written inside the file handler code. The class itself is fairly cohesive as it only reads and processes the XML. One small issue with cohesion would be the fact that a PassengerRouter is created through a method in this class. The MapReader should not be creating a routing classes, rather this should be the job of a passenger class. Furthermore, there is a bit of data duplication as lists are stored in both MapReader and simulation.

Line.java

The Line class keeps track of specific train lines containing Stations and Tracks on that line. It color codes each line (and also track) based on the Station color that created it. Single or dual tracks can be added to the Line connecting two Stations, only if there already is a station on that line; else only a single Station is added. The attributes are well commented, and it delegates the tasks properly. It follows the Creator and Expert pattern as it is responsible for creating the tracks and dual tracks, while keeping information regarding the Stations and Tracks on that particular line. It exhibits low coupling as other classes depends on line but it is coupled with Stations and Tracks. We observe high cohesion as Line is focused on its own task and does not have methods that collide with other classes tasks. Line is a session controller as it adds station and tracks to itself. The notable disadvantages are that the attribute visibility modifiers are set to public when it should be private. Also, specific exception types have not been handled.

Track.java

The track class joins two stations together that are on the same line. It is a highly cohesive class as it only calculates where to render itself, and check if a train can enter or leave the track. It is also very lowly coupled. It has no important objects that it is coupled with apart from the rendering object. The consequence of this is that this is a good polymorphic class, allowing other types of classes to be branched off this like the DualTrack class. If a new track type is added, it can be added seamlessly without making any major design changes. Additionally, code maintenance and enhancements are much more simplified. There is a logical error in the DualTrack class where rendering a line and setting its color were called once too many. Also it was using a variation of the specified track color rather than the actually defined color itself.

Station.java

The station class is one the classes that gets rendered on screen, and it allow trains to enter into and depart from. The Object-Oriented quality of the class although uses good delegation since it ensures the use of other classes' methods, however there is the case of an inappropriate attribute assignment as seen when using a constant RADIUS to a local variable with the same name (radius), the use of returning a public static integer constant is also inappropriate. Based on the GRASP patterns, it can be said that the station class is neither a creator nor a controller, but arguably an expert because it has the information that it needs for it to perform its own operations. On the other hand, station has low cohesion because station should not be the one generating and returning a passenger, as it should be the passenger generator class to generate the passenger by passing in the station as a parameter. Furthermore, a station should not be checking whether a passenger should leave, but the passenger class itself should use the router instead. It can be said that station is highly coupled with line because a line has many stations, while a station must be on a single line, thus they both depend on one another.

ActiveStation.java

Much like the station class, active station is a station that allows passengers to disembark, but in additional also allows passengers to embark the train. It can be considered a creator because it creates a passenger generator to create new passenger classes for that station. It however, has low cohesion because active station is creating passengers when it should be passenger class to generate passengers themselves by passing the station as a parameter. It is only highly coupled with its super class station.

Daewin, 679182 (dsv)
Julian Chua, 677491 (weec1)
Rahul Singh, 696272 (rahuls1)

Passenger.java & PassengerGenerator.java

The passenger class represents the passengers that will be boarding the train in the simulation. Very cohesive class as it only has a method that updates the travel time of the passenger. The PassengerGenerator class is used to create a list of generated passengers. It follows the creator principle meticulously as it records and closely uses instances of passengers. In terms of a code design standpoint, it does a good job of abstracting a single passenger creator method from a multi passenger generator by using the single passenger creator. This also makes it an organized and cohesive class. Both passenger and passenger generator classes are coupled to the station class as a passenger needs a station to be created at and also needs one for its destination.

Train.java

The Train class represents the current state of the train at a particular instance of travel time. It delegates its task well by using the Line class to determine its next station etc., while adhering to proper naming conventions. It has the advantage of abstracting its functions to look like a simple *update* method but it is actually processing all the information necessary to move. It applies the Expert pattern as it has all the relevant information needed to run, such as the states that the train is in and what it should do at which state. As *embark* is not implemented here, it should have been made abstract. Train is highly coupled with Line because a train needs to keep track of what line it is on at all times. The train is missing the important state of going back to the depot, as this allows passengers to embark at Stations and end up at the depot.

Small/BigPassengerTrain.java

Small and Big Passenger Trains are subclasses of Train, implementing the *embark* and *render* methods which exhibits good abstraction. Train is highly polymorphic as it allows multiple types of trains. A key problem is of magic numbers where if you have many subclasses, you would have to change them one-by-one. Also, specific exceptions are not handled along with code with no comments. Small and Big Passenger Trains are highly coupled with Train as they implements/extends it, therefore changing trains would need considerations for small passenger trains.