SWEN30006 Software Modelling and Design

Semester 1 2017

Project B – Metro Madness

Design Analysis

Group 73

Members :

Darren Yeoh Cheang Leng – 715863

Marco Vermaak - 760718

Ziqian Qiao - 770758

General responsibility assignment software patterns also known as GRASP, are guidelines used in assigning responsibilities to classes and objects in object oriented software development.

In this design analysis of Shoddy Software Development Company's metro train simulation, we will be using a few of these patterns as a guideline to criticize and improve upon their design.

CONTROLLER

Starting off with the controller pattern, this pattern delegates the responsibility to class which handles and coordinates a system operation. This is usually the class that handles the software behind the scenes of the UI layer for the output to be displayed. In the metro train simulation, a clear example of a class that obeys the controller pattern is the Simulation class, as it sits between the UI class, MetroMadness, which acts as an adapter to the External Graphic Library and the system components – it contains a list of all the system components (lines, stations and trains) and is responsible for the coordination of system events (updating), while the rendering functionality is delegated to the UI adapter.

CREATOR

Next, we take a look at the creator pattern, which assigns the responsibility of creating other classes based on the relationship between objects. Shoddy Development Company has done a good job in designing most of their classes based on this pattern, for example, the Passenger class, which creates Cargo. This is a good design because, passengers will be the ones carrying the cargo so it would make sense that the class is responsible for creating the Cargo if it exists. The only strife with the Passenger class is that the Cargo class is defined within the Passenger class itself, and in my opinion, should be separated into its own class. Another example is the simulation, creating all the objects needed for the simulation.

COHESION

Moving on to cohesion, this pattern keeps objects focused on their task and is a measure of how strongly or functionally related and focused the responsibilities of an element are.

An example of the metro train simulation not adhering to rules of this pattern is the Station class, which has a generatePassenger function which should instead, probably be handled by the PassengerGenerator class. PassengerGenerator does have a function with the exact same name, but still relies on the function in Station to generate passengers. This is bad design and is very confusing as they both take the different arguments and perform differently on top of having the same name. But apart from that, the rest of the classes seem to display some degree of cohesion, as their functions are all related to their responsibilities, e.g. the train classes having functions only related to their class responsibilities, like moving the train, updating the state of the train, rendering it and other similar things.

COUPLING

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. Usually low coupling is favoured as a pattern when assigning responsibilities to classes. Shoddy development has done a bad job in this context as many of their classes are highly coupled.

In this design, Simulation is coupled with MapReader – it contains an instance of MapReader and calls methods of MapReader. It provides a hard-coded file which MapReader parses and calls the methods of MapReader to populate the system-component ArrayLists. If the way in which MapReader gets its data (e.g. from a database instead of file) changes or the process in which it parses data changes, code would have to be changed in Simulation. Simulation is really only interested in the system components, and not how they came about – this is MapReader's reponsibility, which is a candidate for the Pure Fabrication

Pattern – Simulation should not know about the process with which MapReader populates the Simulation data, only a reference to Simulation should be passed to MapReader so that it can add lines/stations/trains to Simulation using mutator methods – MapReader is the Expert and Simulation is the Controller – this improves cohesion and decouples the two classes.

Similarly, PassengerGenerator has been coupled tightly with the Station and Active Station components, with its functionality spread between these classes, decreasing cohesion and increasing coupling. It also forces Station to keep track of lines so that a random destination can be picked for the generated Passenger – this increases coupling between Station and Line – Station does not need to know about Line. It would make more sense to delegate instantiation of PassengerGenerator to the MapReader class, which is an Expert on all system components, since it has the data required to generate them.

Since PassengerGenerator is coupled to ActiveStation, it is inflexible in terms of different behaviours depending on the type of Station – if different types of Stations were introduced that required different algorithms for Passenger generation (e.g. via the Strategy Pattern), the current coupling would make such a change difficult. To solve this we might have to couple it to the Station class, so that different implementations of the PassengerGenerator can be assigned to different stations via inheritance

Furthermore, in the Station class, the canEnter() function takes Line as argument, but does not use this argument at all in its logic – it only needs to know the number of trains currently inside the station. The canEnter() function should be taking something else like train type as an argument, to allow for the fact that there might be different rules corresponding to different trains attempting to enter a certain type of station. It is unnecessarily coupling Station to Line.

Train is unnecessarily coupled to the Track class – to determine if the train can enter a track, it can query the line that it is currently on, giving it the current Station that the Train is in to derive the forward track, and can query the next track provided by getter methods of Line to determine if it can move onto it. Again, these increases coupling between Train and Track, which is not necessary.

The Passenger class contains an inner class Cargo, and generation of cargo for a passenger is required. The two classes are very tightly coupled, and does not allow for the possibility that a Passenger does not have cargo, unless we represent it with cargo of weight 0.

INFORMATION EXPERT

Apart from that, another pattern is the information expert pattern which dictates that the class with the most information required to carried out a responsibility, should then be assigned that responsibility. For the most part, Shoddy Development has gotten this part right, except for a few cases such as the PassengerGenerator needing to use generatePassenger in the Station class, which instead should be in the PassengerGenerator class. One example is that the Train class has
Station and Track which it uses to know where it is and which track they take before
their next station. But their design does not come without repercussions as this causes many of the classes to be highly coupled.

From a more general Object-Oriented best-practice standpoint, there are some obvious issues with this design. Most of the attributes of the classes are public, and there are almost no getter/setter methods in the classes – this violates encapsulation and means than classes can access the fields of other classes without constraints i.e. class A can set class B's attributes to invalid types – this is very poor OO design. These attributes should be made private, and accessor/mutator methods provided.

There are some issues with the Representational Gap between the Domain and Design Models. The Domain Model shows that a Station can only belong to one line; it does not account for Stations which lay at intersections between lines, which is accounted for in the Design Model, thus introducing Representational Gap. Further, in the Domain Model a Line is shown to have only one train travelling on it, but in the Design Model a Line can have zero to many Trains travelling on it, introducing further RG between the models. Moreover, the domain model doesn't account for different types of Stations and shows Station as always having at least 1 Passenger departing / having that station as its destination; some Stations do not have passengers.

Currently the design only allow for simulation of two types of trains, BigPassengerTrain and SmallPassengerTrain, which is inflexible. It would be better if a variable to set size of Passengers is added in Train class, and delete BigPassengerTrain and BigPassengerTrain.