Group 44

Metro Madness Analysis

General Design Principles

In the initial implementation of the simulation all of the instance variables are public, this results in bad encapsulation. As a result it may cause high coupling if the code is extended in the future, as a result it has been fixed in the new implementation. Furthermore, although there was use of polymorphism it has been implemented poorly as the subclass is overriding all methods of the superclass in the **Track** class of the initial implementation (this is further detailed below in our discussion of the **Track** class).

On the other hand, the code does follow the abstraction quite well, there are many examples of so only one will be discussed. The example used will be **ActiveStation** class' *enter()* function. The details and logic of determining whether a train can enter the station are abstracted away when the train enters a station it only needs to call the *enter()* function.

As cohesion and coupling are more principles than design pattern they will also be discussed here. The code in general does have high cohesion within classes and low coupling between classes. An example of high cohesion would be the **Simulation** class, and an example of low coupling in the code is the **PassengerRouter** class, both of which will be discussed below

MapReader

Follows creator pattern well

MapReader is the object in the software that has the responsible of reading input which is the data required to initialize Train, Line, and Station. MapReader follows creator pattern well as it creates the Train, Line, and Station class all of which need data that is retrieved from input. As the MapReader is the object in the software that reads the input, it has all the data that is needed to initalize these classes.

MetroMadness

Follows controller pattern well

MetroMadness deals with all the inputs that the user makes through the *handleInput* function which is then delegated away to the **Camera** class. It also deals with the rendering of the UI through the *render* function which also delegates the actual rendering to the **Camera** class and **Simulation** class. This class does not do much work itself other than delegating which class should do what given a user input. It is the first object beyond the UI layer that receives user input and controls the updating of the UI through delegation. This is essentially the definition of a Controller pattern, which is why **MetroMadness** follows the controller pattern well.

Simulation

Follows information expert pattern well

Simulation is tasked with updating all the **Trains** in the simulation, and also the rendering of the **Lines**, **Train**, and **Station** classes for the UI. This is justified as simulation has instance variables of the previously mentioned classes and as a result it has all the information that is required to implement the methods that were previously discussed (*update* and *render*).

Has high cohesion

The **Simulation** class has high cohesion as the methods that it implements are all directly associated with updating the current state of the simulation and then rendering it. Moreover to further support high cohesion, **Simulation** does not care about when it should be called or where the camera should be centered for the user, as that responsibility is delegated to the **MetroMadness** class, it is just concentrated on rendering the UI via delegation.

Train

Follows information expert pattern well

Train follows the information expert pattern well because the required information achieve the responsibilities are already in the class. An example of this is *update* function that is implemented by the **Train** class. There wasn't a lot of need to access another object and retrieve information, as the **Train** class has most of the required information.

Compare this to if the *update* method of **Train** was implemented by the **Simulation**, then to achieve the same outcome **Simulation** would have to access each of the **Train's** instance variables that are required to update the train such as: departureTimer, track, state, and etc, which would add an additional layer of complexity that isn't needed.

Not good modelling for train sizes

The subclass of train has magic numbers for their maximum capacity, and there isn't a way to specify a new train with a different capacity other than creating a new subclass (meaning lots of code duplication for the *embark* and *render* methods). Moreover, the methods that were overridden (*embark* and *render*) could all be abstracted. Which is why the subclasses were deleted and an instance variable for maximum capacity was added to the *Train* class.

Track

Track follows the Polymorphism design pattern

There are many cases of polymorphism but one case is in the **Train** update function. There is no need to use a conditional statement to determine whether the track was one way or two ways due to the subclassing of **Track**. Moreover, we are able to call the *canEnter* function due to subtyping polymorphism making the code cleaner.

Code quality of the implementation

The implementation of the **Track** polymorphism design pattern isn't good. In the initial implementation, the subclass (**DualTrack**) overrides all of the methods of its superclass (**Track**). The sole purpose of the superclass was to store the instance variables, a better implementation would be to make **Track** an abstract superclass with all abstract methods. The reason for this is that we are able to store the instance variables and our subclasses wouldn't be overriding all the concrete methods of our superclass like in the initial implementation which is bad practice.

PassengerGenerator

PassengerGenerator follows Controller pattern well. It uses two functions to control the creation of passengers in random stations. This allows low coupling between Passenger and Station, since the creation of passengers is separate from Passenger and Station allowing changes to be made in the creation of passengers without affecting Passenger and Station.

Station

The previous design uses ActiveStation class to model a specific type of station that allows passengers entering and leaving. This does not allow combinations of other types of station. For example, depot station that does not allow passengers entering or leaving the station can't be captured by the previous design.

Follows Information Expert pattern well

Station follows the information expert pattern well because the all methods in the class are the responsibility of a train station and all relevant information such as router, trains and lines can be accessed. For example, station needs to check if a train can enter, so Station's *canEnter()* can access the trains stored in the class and check if it exceeds number of platforms.

PassengerRouter

Follows Pure Fabrication pattern (a specialization of Indirection)

If the **PassengerRouter** wasn't created then the **Station** class would have low cohesion as the **Station** class shouldn't be responsible for the routing of passengers. The use of the Pure Fabrication design pattern solves this problem by creating **PassengerRouter** whose sole purpose is to handle the routing of passengers. As result instead of having low cohesion for **Station** it would now have high cohesion as it is able to delegate the routing to another class.