COSC 3P91 – Design Patterns – 6402176, 6186076

MICHAEL WISNEWSKI, AIDAN LAROCK, Brock University, Canada

This paper is to breakdown and discuss the implementation of design patterns in the java code base for the Traffic-Sim game. There are several design patterns which have been implemented within the code in order to improve the efficiency and readability as well as improve the overall implementation of current code.

ACM Reference Format:
Michael Wisnewski, Aidan Larock. 2022. COSC 3P91 – Design Patterns – 6402176, 6186076. 1, 1

1 TRAFFICSIM

1.1 Classes and Files Added

As per instructions multiple files were created, which implements the following design patterns.

Table 1. New Classes/Files and Descriptions of Each

Package	Class	Description
Assets/	`Map.xsd`	XSD file which contains the rules of the xml file and defines what map files are allowed using the rules.
Assets/	`Map.xml`	XML file which contains map data in a specified format, this data is then read via a parser into the game.
Map/	`Parser.java`	Reads the map xml file into the game and uses the xsd to confirm that the file is of valid type.
Map/	`Map.java`	Interface of the composite design pattern for the graph component and Road Segment leaf.
Map/	`RoadSegment.java`	Leaf node of the composite pattern.
Map/	`Graph.Java`	Composite which contains RoadSegment leaf nodes.
Players/	`PlayerFactory.java`	Creates and returns newly created player objects using strings player name and player type as parameters and calls vehicle factory as well.
Vehicle/	`VehicleFactory.java`	Creates and returns newly created vehicle objects using string vehicle type as parameters.
Main/	`TrafficMain.java`	Singleton class which is the main engine of the game creating and calling objects of the game.
Main/	`TrafficDemo.java`	Tests the TrafficMain singleton class design pattern.
Main/	`FactoryDemo.java`	Tests the PlayerFactory and VehicleFactory design patterns.

Author's address: Michael Wisnewski, Aidan Larock, Brock University, 1812 Sir Isaac Brock Way, St. Catharines, ON, L2S 3A1, Canada.

2 DESIGN PATTERN BREAKDOWN

Design patterns are useful for verifying that code conforms to high quality standards as well as making sure the code is as efficient and readable as possible, by following these patterns, we have refactored and updated our game to contain these elements wherever possible. The `MVC`, `Factory` , `Singleton Creational` , `Composite Structural`, and `XML/XSD` patterns and files will be explained in further details below.

2.1 MVC

Model View controller is used many times through the entirety of this assignment however, the most noteworthy use is in `Players/Player.java`, we can see that `Player.java` clearly resembles an MVC design pattern. This is because we can assign a model (of `Vehicle/Vehicle.java`) to the view (of type `Graph.java` or `RoadSegment.java`) thus creating a Model View controller. Each player is considered as a controller that is able to control the model vehicle that they are driving and view the information of the road segments through the player model view controller (MVC).

2.2 Factory Design Patterns

The factory design pattern allows for the creation of objects to be simplified. In the code, this design pattern is used for vehicle and player creation. The `Players/PlayerFactory.java` and `Vehicle/VehicleFactory.java` allow for the creation of vehicle and player types within the code. The PlayerFactory takes in a tuple (player type, player name, vehicle). By comparing the player type to a player or AI, the factory creates a user controlled player object or an AI player object using the player class and anonymous class. The factory assigns the player name to the player and calls upon the VehicleFactory to create a vehicle object for the player using a single parameter of vehicle to create a new vehicle of that type, that type is then returned to the PlayerFactory where it is assigned to the newly created player, which is then returned to `Main/TrafficMain.java`.

2.3 Singleton Creational Design Pattern

The Singleton design pattern as the name suggests is a creational design pattern. This type of pattern only creates a single instance of an object. For our case, we refactored `TrafficMain.java` to be used as a singleton. Inside `TrafficMain.java`, it is clear to see that we created a static instance of the object. Then, with a getter method `getInstance()` we can retrieve the instance of the object globally. As seen in `Main/TrafficDemo.java` we have a main function which simply retrieves the instance of the object, from which we can run the main game.

2.4 Composite Structural Design Pattern

The composite design pattern is comprised of `RoadSegments.java` and `Graph.java` using the interface `Map.java` we store the types as strings. Thus, when we create our composition of Graph using a array list of road segments we can easy identify the type from the interface. Ultimately, this is useful for the future expansion of a GUI because we can iterate through multiple graphs (assuming there will be an online aspect) we can easily tell which graph belongs to which player and what road segments they have vehicles on.

2.5 XML and XSD

Originally the map was read in as a text file. This can cause problems when implementing new maps as these text files can have any format or any text inside of it. Using XML and XSD files, we can confirm that the data inside the map asset file is a map of correct format.

2.5.1 XSD. The XSD file "Map.XSD" defines the XML file. The XSD file is made of elements and attributes which can be used to create XML files and alter XML files. By using the XSD file for map creation, one can simply follow the rules set out to create more maps to be used in future implementations of the game such that the XML created is in an acceptable format for the parser. The XSD file contains "Row" elements of which there must be at least 2 rows, and a maximum of 512 rows (so that maps at least 2x2 and maximum 512x512 to not be too large or too small). There is also an int element with similar minimum and maximum occurrences in each row, these int elements represent the size of road segments and the number of int elements per row represent the number of roads in the map.

2.5.2 XML. The Map.xml file is used to load a map into `Map/Parser.java`, this is where we read in the `assets/Map.xml`. We use the document builder library from the java xml parser, this reads the file line provided as a string (i.e "assets/Map.xml") and begins parsing. We create a Node list based on the document element "Row" this creates a list of all elements with the tag "Row". We calculate the length of this list and begin parsing. We assume that the length represents the amount of integers to be parsed for each row. We select each key from the element tag "int" and place it into a linked list. This is way we can dynamically forward to a fixed array and create our map. The class has getter methods that returns the size of the row which is assumed to be the same as column length and the linked list containing each key. This can be forwarded into our graph creation and build the map required to run the simulation.

3 WRAP UP

Using proper design patterns and ideas, the Traffic-Sim game has been updated to be more efficient, readable, and conform to a high quality standard of code. The code can be found in a `ZIP` file, and the changes can be seen throughout the files inside.