COSC 3P91 – Assignment 3 – 7023609

AMEYA CHINDARKAR, Brock University, Canada

In this assignment, I developed a time-stepped traffic simulation game utilizing a console-based interface for player interaction. The primary objective was to implement all the specifications in Assignment 2 into a functioning simulation, guided by a provided UML Class Diagram to ensure a unified starting point for all participants. The system is designed to be simple, and divided into 6 different packages, each handling an aspect of the game. In addition, to this, for Assignment 3, I remodeled my code to affirm the MVC pattern.

1 MODEL

The Model represents the state of the application and its underlying data structures and business logic. the following classes are a part of the Model:

- TrafficElement
- Intersection
- Lane
- Point
- RoadSegment
- TrafficNetwork
- Direction
- MovementDecision
- MovementStatus
- Position
- Bus
- Car
- DamageStatus
- Reputation
- Truck
- Vehicle

1.1 Intersection Class updates:

The constructor now takes both the mapPosition and a list of RoadSegment objects, initializing the intersection in a complete state. The roads attribute is now final, which implies that the list of road segments cannot be reassigned after the object is constructed. The getRoads method returns a new ArrayList, which is a defensive copy of the internal list, to prevent the caller from modifying the original list. The setter has been removed to maintain immutability

1.2 Lane Class updates:

Both length and direction are final. The Lane class now includes a constructor that sets these values upon instantiation, and input validation ensures that length is positive. The setters for length and direction have been removed to make the class immutable.

Author's address: Ameya Chindarkar, Brock University, 1812 Sir Isaac Brock Way, St. Catharines, ON, L2S 3A1, Canada.

Update date: 23 March 2024

1.3 Road Segment Class updates:

Both intersections and lanes are declared final. The class now includes a constructor that initializes these collections, and both are initialized with defensive copies of the input lists to protect the internal state. The setters have been removed to make the class immutable. This design ensures that once a RoadSegment is created, its set of intersections and lanes cannot be changed

1.4 Traffic Network updates:

The collections are initialized in the constructor to ensure they are never null. ArrayList has been replaced with List in field declarations and method return types to adhere to the principle of programming to interfaces, not implementations. Methods for adding and removing roads and intersections provide more control over the individual elements in the network, rather than replacing entire collections at once. Methods checkNumberVehiclesAtIntersection and checkNumberVehiclesInIntersection now accept a single Intersection object instead of a list, which seems more logical for the operation being performed.

1.5 Traffic Element updates:

In this enhanced version, a check is added in the constructor to ensure that a null Point is not passed, which helps prevent the creation of invalid TrafficElement instances. This kind of validation is crucial for maintaining the integrity and reliability of the model.

1.6 Movement Status updates:

The class is made immutable by having final fields and setting them via a constructor. Validation is added in the constructor to ensure that the position is not null, the speed is not negative, and the direction is not null. The setters are removed to maintain immutability. This design ensures that the MovementStatus of an entity cannot be changed once it is created, leading to a more robust and predictable system.

1.7 Position updates:

Both trafficElement and coordinate are final and set via the constructor, ensuring that a Position object's state cannot change after it's created. The constructor checks that neither trafficElement nor coordinate is null, throwing an IllegalArgumentException if this is the case to prevent the creation of invalid Position objects. These changes reinforce the integrity and consistency of the Position objects in your application.

1.8 Damage Status updates:

The class now uses List<Double> for sufferedDamageHistory and generatedDamageHistory to generalize the collection type. Added addSufferedDamage and addGeneratedDamage methods to incrementally update the damage history without exposing the entire list for replacement, which enhances encapsulation. Validation checks ensure that negative values cannot be set for the damage status or added to the damage history, preventing illogical states.

1.9 Reputation updates:

The reputationHistory is now a List<Double> to use a more generic type instead of a specific implementation like ArrayList. Removed the setReputationHistory method to prevent external replacement of the entire history, which ensures the integrity of the historical data.

2 VIEW

The View handles the presentation layer and user interface. It presents data to the user and collects user input. These classes would be a part of the View:

- ConsoleInterface
- UserInterface

2.1 Console Interface updates:

Here's how ConsoleInterface fits into the View component:
Displaying Information: The displayMessage and displayError methods allow the console interface to present information and errors to the user, respectively, which is a core function of the View in MVC. These methods output data to the console, directly interacting with the user.
Gathering User Input: The getInput method provides a way to collect input from the user, another key role of the View. It facilitates user interaction with the application, allowing the user to respond to prompts and make choices.
Decoupling from Business Logic: As an implementation of the UserInterface, ConsoleInterface is decoupled from the business logic and data handling (which are part of the Model and Controller). This separation adheres to the MVC principle of separating concerns, where the View should only be responsible for display and user interaction.
No Direct Data Manipulation: The ConsoleInterface does not manipulate the application's data directly, which aligns with the MVC concept where the View should request data from the Controller to display to the user, and not directly from the Model.

2.2 User Interface updates:

Added clearDisplay() method for clearing the user interface, which might be particularly useful for console interfaces or resetting GUI states. Added closeInterface() method to properly handle the closing of the user interface, allowing for clean-up or save operations before the application closes. These enhancements make the UserInterface more flexible and detailed, better fitting the View aspect of the MVC architecture by clearly defining how user interaction should be handled.

3 CONTROL

The Controller handles user input, updates the model, and changes the views accordingly. You don't have a dedicated controller class listed, so you would need to create one. The following are a part of the Controller:

- GameEngine
- ChallengeHandler
- MovementControl
- Player

3.1 Challenge Hnadler updates:

The ChallengeHandler now takes a TrafficNetwork in its constructor, associating it with a specific network where challenges will occur. The runChallenge method has been simplified to return a boolean or another suitable result type that represents the outcome of the challenge, which aligns more closely with typical game logic where challenges have win/lose outcomes or other effects. The findNeighbours method remains private as it is an internal operation of ChallengeHandler, used to support challenge logic. This structure allows ChallengeHandler to focus on the specifics of managing challenges in the game, which the GameEngine can then utilize in its broader control over game logic and flow.

3.2 Game Controller updates:

Introduced a startGame method that could initiate the game loop, controlling the flow of game updates and rendering. The updateGame, render, and checkEndConditions methods provide a structured way to manage the game loop's main phases: updating the game state, rendering the state to the user, and checking if the game should continue. Direct manipulations of the model (like TrafficNetwork and Player) are handled within the controller, which then communicates necessary information to the view (UserInterface).

4 FACTORY DESIGN PATTERN

For the traffic simulation game, I implemented the Factory design pattern. This choice was driven by the need to instantiate various types of vehicles dynamically based on game conditions and player choices.

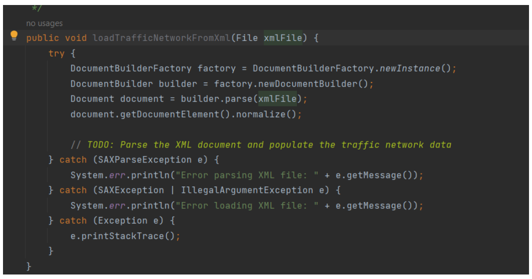
4.1 Location in Code:

The Factory pattern is implemented in the VehicleFactory class within the Vehicle package. This class serves as a centralized point for creating different types of vehicle objects like Car, Bus, and Truck.

4.2 Reason:

The Factory pattern was chosen because it simplifies the creation process of vehicle objects, encapsulating the instantiation logic within a single method. This approach aligns with the principles of object-oriented design by promoting encapsulation and reducing coupling. It allows for easy extension (adding new vehicle types) and modification (changing vehicle creation logic) without affecting the client code.

5 LOADING AND PARSING XML FILES



```
 */
no usages
public void loadTrafficNetworkFromXml(File xmlFile) {
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.parse(xmlFile);
        document.getDocumentElement().normalize();

        // TODO: Parse the XML document and populate the traffic network data
    } catch (SAXParseException e) {
        System.err.println("Error parsing XML file: " + e.getMessage());
    } catch (SAXException | IllegalArgumentException e) {
        System.err.println("Error loading XML file: " + e.getMessage());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

This is the code to parse XML files located in the Traffic Element Section in the code.