COSC 3P91
Assignment 2

Due date: February 28th, 2022 at 23:59 (11:59 pm).
Team: The assignment can be completed in groups of up to two students.
Delivery method: the student needs to delivery the assignment only through Sakai.
Delivery contents: document with answers and [Java, C, C++] codes if applicable (see Submission instructions).
Attention: check the Late Assignment Policy.

Assignment Overview

To implement traffic simulation game using a console as a interface for the interactions of a player. The console is basically System.in/out for I/O communication with the player. Ideally, your design from Assignment 1 will form the basis of your implementation. Please note the UML Class Diagram for this implementation is provided. The diagram is intended to support a equal startup for all students and facilitate the implementation towards the game design for this assignment.

The Game

The **game** is a simplified traffic simulator where vehicles move around in a urban traffic network. The game is a turn-based simulator that updates the position of vehicles each "time step".

The **traffic network**, urban environment, is basically a map coded in the form of a graph, where edges are road segments and nodes are intersections. The map can be defined statically but is kept in memory as objects. A file can define the structure of the graph and be loaded at the beginning of the game (at its startup). Each road segment maybe composed of more than 0 lanes.

The **mobile entities** in the simulator are vehicles, which can be cars, trucks, or buses. As defined as a game, certain vehicles can be controlled by users while other are just controlled by the Game Engine.

The game follows a **time-stepped simulation**, where from time to time, the state of moving entities are updated accordingly, and the user is prompted for a some decision. Therefore, vehicles take movement/routing decisions, such as turning left, turning right, and going straight in intersections, whenever approaching a "decision position" in the map.

As the **main objective** of the game, **users** need to conduct their vehicle properly over the traffic network to avoid collisions.

Assignment Details

The assignment is to implement a system which produces a game of traffic simulation as described above. and present the model using a UML class diagram. Ideally, the design you created for Assignment 1 will form the basis of your implementation. In this case, the complete and correct UML Class diagram for the game is provided.
Note that **there are many feasible implementations and that you will be evaluated based on the merits of your code**. Therefore, your code does not necessarily need to be perfect; it is OK to re-evaluate your design during implementation, but it must demonstrate that you have thought out and planned the overall design of the system using object-oriented principles. Also, keep in mind that you will actually be implementing the system later; thus, a well-planned design will be beneficial in the future.

Your design should facilitate, at minimum, the following behaviours:

- **Looking** – the player should be able to request information about its vehicle's surroundings.

- **Moving** – the player should select a decision about next moving decision: changing lanes (left, right), turning in intersections, or keeping straight.
- **Asserting** – the decision of the player should be verified so that the decision means a consistent state. A problem could trying to enter a road segment in the wrong way or that is full of vehicles.
- **Gambling** – together with asserting, it allows the player to take chances. It gives risky opportunities to the player. Thus, by rolling dices, it infers if the player was lucky or not. For instance, the player decides to turn into a road segment that is full but there is a chance that, in next turn, the segment will vacant a spot.

The provided class diagram is not fully complete and should give hints about the relationships among classes in the code. The diagram is supposed to cover the minimum requirements of the needed funcionalities (Looking, Moving, Asserting, and Gambling) at the design level. The class diagram does not represent all setters and getters needed for guaranteeing access and object encapsulation.

Movement Decisions Several situations in the simulation trigger moving objects/entities for decisions that might force them to change their direction or allow them to keep the same moving trend there were following. Such situations maybe be clustered in two major occasions, spatially represented on the simulation Map: **intersections** and **middle of lanes** (lane changing). The following aspects should be taken in consideration when designing your simulator:

1. **Intersection Decision**. When at an intersection, the vehicle may opt for moving straight ahead, turn to the left or turning to the right. The decisions need to follow the physical constraints of the map: vehicles are not allowed to turn left a some intersections.
2. **Lane Changing**. Vehicles while moving in a road segment, they will have the chance to be change lanes in case there is too much traffic ahead slowing its movement. In case there is a traffic jam, the player will have the chance to change lanes several times while it has not covered the whole road segment yet. I some other situations, the player will be prompted for intersection decisions in case there is low traffic density and the player's vehicle can cover the road segment really fast.
3. **Challenge**. Vehicles at the intersection will compete for their right to go straight or turn. A player may feel lucky and try to move without caring about other vehicles. This action will be lead to a Challenge, which is somehow Gambling, and the Game Engine will roll dices and tell how is the winner in the Challenge.
4. **Reputation**. Each vehicle has an overall reputation in the game. It grows as right, conservatives decisions are made. It decreases as the player looses Challenges or when it gambles. All vehicles controlled by the Game Engine may present the same Reputation during all the simulation; they may only take conservative decisions all times. The Reputation can range from predefined MIN to a MAX values.
5. **Reputation Value**. With higher Reputation, a player has a better chance to win Challenges and win when Gambling.
6. **Damage**. Any movement decision may lead to an accident, which necessarily will damage the vehicle. The damage may be determined probabilistically (rolling dices) and according to several factors: status of vehicle, speed of vehicle, movement situation, etc.

Important Notes

1. Given that this is only an initial implementation, do not worry too much about supporting multiple players. We are mainly concerned with having a system which facilitates the overall logic of the game. However, we will, eventually, be adding multiple players in future assignments so do not pigeon-hole the implementation into only supporting a single player.
2. For simplicity, do not employ sophisticated mechanisms to support gambling. However, make it consistent since it is considered a 'basic' functionality in the game.
3. There are plans to implement a graphical user interface (GUI) in later phases of this 'project' (later

assignments), but it is not necessary to implement anything more sophisticated than a console (terminal prompting the player for input).

4. Recall that this course is about advanced object-oriented programming. Thus, you will want to ensure that your implementation makes use of object-oriented constructs, such as interfaces, inheritance, enumerations, and generics, where appropriate. Such constructs should have been identified during design. Recall that you need at least two enumerations, one for the suit and one for the rank.
5. Be sure to include sufficient comments which should consist of, at minimum, an appropriate comment for each file and method. Consider using Javadoc style comments. If you use NetBenas, Javadoc comments can be auto-created in NetBeans by placing the cursor above a method or a class that currently has no Javadoc comment, typing "/**", then pressing Enter. This will create a template Javadoc comment with some information automatically completed for you (e.g., parameters for a method).
    - Your comments, if done correctly, should facilitate writing the document that explains your code. The comments should match with what you have in your description document.
6. Besides the Object Orientation concepts, you must be able to show complete understanding of the following concepts and employ them in your code. **Adequately use them whenever possible and explain them, pointing out in your description file** (in other words, marking will consist checking the existence and proper use of each of these elements):
    (a) Generics (Type Parameters and Wildcards, for example);
    (b) Local and anonymous classes;
    (c) Lambda expressions and method references;
    (d) Exceptions (create customized Exceptions);
    (e) Java-Standard Utility classes (Collections, for example).
    (f) I/O (use of streams generated from collections and lists, for example).

Submission Material

The submission for this assignment will consist of two parts:

1. A ***Description document (PDF)***. A document succinctly describing your design and implementation decisions is necessary. Also, you will find it beneficial to justify your design choices such that the marker does not have to reason about why you have designed your system as you have. Make sure that the description and reasoning of your decisions are consistent with your Java code.
    - ***Latex template - a must for writing your assignment***. For writing your description file, use the Latex template enclosed in this assignment (update it accordingly!). You do not need to install Latex software in your computer. You can write it through Overleaf on your browser (it is a free tool). Just upload the latex template to your Overleaf project; it should compile/render the tex file gracefully.
2. The **Java code** implementing design in previous assignment or the one provided. The code should follow the UML class diagram. The code also must address the OOP concepts listed above. The classes and methods must be documented (commented). The code should compile and run properly. The compilation and execution of your code should not rely on any IDE.
    - ***Compilation***. Provide the command for compiling your source code from the command line.
    - ***Running***. Provide the command for running your compiled code from the command line.

You must guarantee that your code is legible, clear, and succinct. Keep in mind that any questionable implementation decision or copy from any source might have a negative effect on your mark. If you still have any questions regarding which file types are acceptable, please inquire prior to submission. Note that it is not the fault of the marker if they are unable to mark your assignment due to submitting an unreasonable or uncommon file format.

Marking Scheme

Marks will be awarded for completeness and demonstration of understanding of the material. It is important that you fully show your knowledge when providing solutions in a concise manner. Quality and conciseness of solutions are considered when awarding marks. Every code added to the originals should be well commented and explicitly indicated in the Java files; lack of clarity may lead you to loose marks, so keep it simple and clear.

Submission

Submission is to be a PDF (description document), and text Java files (Java code). All the submission should be performed electronically through Sakai.
You must guarantee that your code is legible, clear, and succinct. Keep in mind that any questionable implementation decision or copy from any source might have a negative effect on your mark. If you still have any questions regarding which file types are acceptable, please inquire prior to submission. Note that it is not the fault of the marker if they are unable to mark your assignment due to submitting an unreasonable or uncommon file format.

All content files should be organized and put together in a ZIP for the submission through Sakai.

*** Do not forget to include the names and student IDs of group members.**

Late Assignment Policy

A penalty of 25% will be applied on late assignments. Late assignments are accepted until the Late Assignment Date, three days after the Assignment Due Date. No excuses are accepted for missing deadlines. However, deadline extensions may be granted under extenuating circumstances, such as medical or physical conditions; please note that granting the extension is under the instructor's discretion. However, deadline extensions may be granted under extenuating circumstances, such as medical or physical conditions; please note that granting the extension is under the instructor's discretion.

Plagiarism

Students are expected respect academic integrity and deliver evaluation materials that are only produced by themselves. Any copy of content, text or code, from other students, books, web, or any other source is not tolerated. If there is any indication that an activity contains any part copied from any source, a case will be open and brought to a plagiarism committee's attention. In case plagiarism is determined, the activity will be cancelled, and the author(s) will be subject to the university regulations.
For further information on this sensitive subject, please refer to the document below:

`https://brocku.ca/academic-integrity/`