Final Project

Vehicle Simulator (40%)

DUE DATE

Session 15

OBJECTIVES

The main objectives of this project are to:

- Interpret specifications and analysis performed.
- Design a solution based on the requirements and specifications.
- Design the logic required for functional solution.
- Translate design documents and algorithms into source code.
- Use debugging tools and error-handling techniques.
- Validate the solution with test data.
- Integrate the knowledge acquired thus far.
- Use the features of Eclipse IDE.
- Demonstrate the use of the Java programming language.
- Apply the various program flow constructs.

DESCRIPTION

Your knowledge on Java Syntax is already good enough to create a small universe. This universe is full of different types of vehicles on the field. Outside of the field is a cliff. Any vehicle that went outside of the field is dead. The field is divided into four different quadrants.

2	1
3	4

In the beginning of the application, vehicles must be randomly populated on one of those quadrants. While the simulation will last for a certain number of turns, vehicles will move slightly each turn depending on their type. When the simulation ends, it will display the location of each vehicle, list of dead vehicles, and which quadrant they are/were in.

Time Required

You will require 30 hours to complete this project. These hours include five in-class sessions plus homework time to complete the application.

Required Material

You will need the following material to complete this project:

- Eclipse 2019-06
- Java SE 8 Development Kit (JDK)

INSTRUCTIONS

This simulation is all about the architecture and functionality not the appearance. Providing GUI is a bonus.

This application requires you to use specific polymorphism structure, composition, inheritance. Certain methods will require you to use exception handling.

When the application starts, the console will prompt the user to input following values:

- Length of each quadrant.
- Number of vehicles.
- Number of each vehicle type.

When the specifications are done, program initialized the field and populate the vehicles (`Car, Truck, SportsCar, Tractor`) of random types in a random position in random quadrants. Coordinate of the center of the field is always (0, 0), each quadrant is always a right triangle and the length of each quadrant is the same.

All vehicles move in their own pattern for each turn. Details will be specified later.

When all the vehicles are dead, User should be able to see the top of three vehicles of each kind that lasted the longest and the details of those vehicles.

Key Functions

The main functions of the simulator may be summarized as follows:

- Use Prompt Input
- Simulation / Display each turn
- Display Results at the end

Detailed Guideline/Requirements

The following is a guideline to implement the simulator.

Object-Oriented design has to be implemented as follows due to the objectives of this course.

Vehicle abstract class

Superclass for all vehicles.

Properties: X coordinate (int), Y coordinate (int), alive (Boolean)

Constructor to initialize all properties.

Move Method:

a) Called for all alive vehicles each turn
b) Increments or decrements either x or y coordinates by 1.

Any type of vehicle should throw an exception if the move method is invoked even when the vehicle is dead.toString method should be provided.

Car class

Inherits from a vehicle. Properties: type (string)

Constructor calling the super constructor and initializing the properties and setting the type to "CAR".

Move method inherited from the vehicle class. toString method.

Overriding from the vehicle class.

Truck class

Inherits from a vehicle class. Properties: type

(string)

Constructor calling the super constructor and initializing the properties and setting the type to "TRUCK"

Move method overrides the Move method in the Vehicle class. Move method:

a) Move method for truck is allowed 2 increment/decrements.
b) Either x or y can increment/decrement, by 2.
c) Both x and y can increment/decrement by 2.
d) Probability for b and c happening should be 50:50

SportsCar class

Inherits from a Vehicle class. Properties: type

(string)

Constructor calling the super constructor and initializing the properties and setting the type to "SPORTSCAR"

Move method overrides the Move method from the Vehicle class. SportsCar is allowed one more life.

Move method:

a) Move method for SportsCar is allowed 3 increment/decrements.

b) Abstract value for movement always has to be 3 and has to move

freely. For example:

(0,0) - > (2, -1)

(2, -1) - > (5, -1)

(5, -1) - > (4, -3)

toString method overriding from the vehicle class.

Tractor class

Inherits from a Vehicle class. Properties: type

(string)

Constructor calling the super constructor and initializing the properties and setting the type to "TRACTOR"

Move method overrides the Move method from the Vehicle class.

Move method:

a) Move method for Tractor is allowed 2 increment/decrements.

b) Abstract value for movement always has to be 2 and has to move

freely. For example:

(0,0) - > (2, 0)

(2, 0) - > (3, 1)

(3, 1) - > (2, 2)

c) Not move at all.

d) Probability of b and c happening should be 50:50

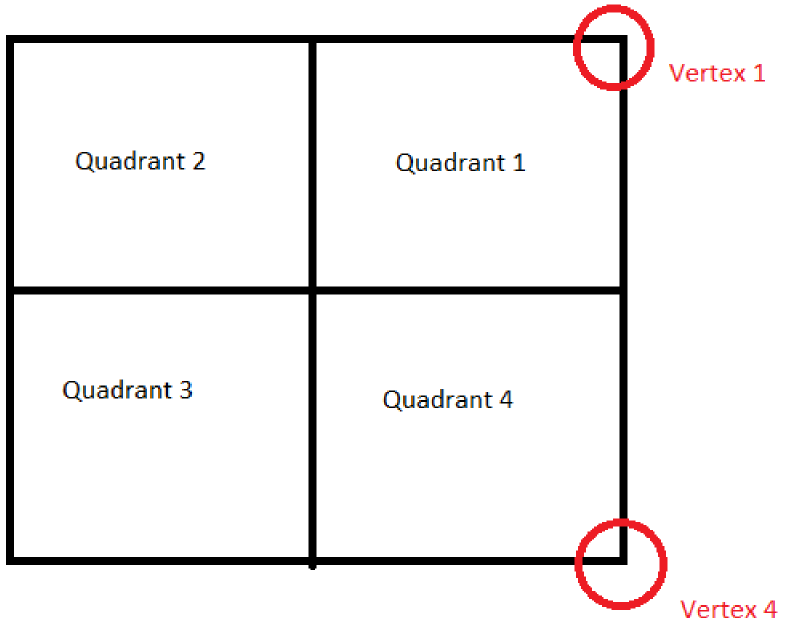
toString method overriding from the vehicle class.

Field class

Includes all the vehicles.

Properties:
4 vertices of the field (array of ints [quadrant number][x, y coordinates (store this value

in whichever form you want)]) Array of vehicles



Constructor:

a) Takes a parameter to determine the length of the quadrant which is always a right triangle.

b) Takes parameters for how many of each vehicle are to be populated. At least four of each vehicle type has to be populated (since the result is to sort out the top three longest survivors).

c) Populating the vehicles, each vehicle is positioned to random positions within a field.

d) Populated vehicles are stored in the array of vehicle class. (Using ArrayList is also possible.)

When a vehicle steps outside of the field, field class should 'kill' the vehicle. You are free to make extra methods that do not violate the guidelines.

Simulator class

Simulator class "has" the Field class.

In each turn, Simulator class should invoke the Move method of each vehicle inside the Field object.

Simulator class should be able to keep track of each vehicle's movements and take a record of it (you are free to store this data anywhere in the application).

"Dead" vehicle's Move methods should not be invoked.

In each turn, console should display the current position of each position.
For example:

```
"Turn 3: SPORTSCAR 1 (DEAD), SPORTSCAR 2 (2, 3), …TRUCK 5
(DEAD)"
```

When all the vehicles are "dead" Simulator class should be able to display the track of each vehicle, how many turns it survived.

For example:

```
"Simulation terminated in 34 turns in the field of 34*34
     Top 3 SPORTSCARS

     1. SPORTSCAR 6 survived 33 turns (0,0) - > (2, 0) - >
        (3, 1) -> (2, 2) … (34, 1)

     2. SPORTSCAR 2 survived 29 turns (0,0) - > (2, 0) - >
        (3, 1) -> (2, 2) … (34, 1)

        …

     Top 3 TRUCKS

     1. …

     "
```

Aside from turns, Simulator class should also be able to measure real-time simulation duration and display it at the end.

For example:

```
"Simulation took total 36 seconds until all vehicles are
terminated."
```

Try-catch block is mandatory to catch the exception from the Move method. Like mentioned on the Vehicle class specification, Move method throws an exception when it's invoked after the vehicle's death.

Simulator class is to include the Run method to start off the simulator.

Main class

This is the class to include the main method. This

class is to "have" the Simulator class.

Main method is only allowed to have three features:

- Prompt User to input values for Simulator configuration:
    - Length of each quadrant.
    - Number of vehicles.
    - Number of each vehicle type.

When the user input is invalid, it needs to request a new value from the user until a valid value is entered.

- Initialize the Simulator class according to user input.
- Invoke Run method in the Simulator class.

Providing GUI using JavaFX is a bonus.

DELIVERABLES:

Part 1 – The Architecture Design:

A key part of this course is to understand Object-Oriented Design. After reading the specifications and requirements, draw a class diagram to fit the requirements. It will work as a blueprint of the project. Write a pseudo code to fit the class diagram. You are free to write pseudo code for certain methods, but not required.

Part 2 -Interpret the Design to Write Source Code

Refer to your analysis and design work from part 1 and create the classes accordingly and wire them as designed. Your code must show the same flow as that which you designed in part 1. For example, if you design an architecture to use a while loop to cycle through an array to extract a specific data element, your code should include the while loop as well with the same logic.

Error Management: Test the Client Prototype application

1. You should test as you build to avoid unnecessary delays.
2. Make sure you add appropriate data input validation, and error-handling (HINT: every time you capture a piece of data, make sure it is valid and matches the type of data expected)

SCHEDULE

Note, this schedule is a guideline only. You may require more or less time during each session.

Session 11: Part 1 Analysis and Design of the work to be done

- Analyze the project specifications, and the feature requirements.
- Create your application design. You must use at least two types of documents for the design component. Suggestion, you may want to use a UML class diagram to design your program's overall architecture, and then use flow chart for logics inside the methods, and or pseudocode for each individual feature (depending how you decide to create the program).
- Validate your logic and architecture with your instructor. Make sure you are able to explain the logic to your instructor and he or she approves it before moving on to the coding phase.
- Typically, you should be able to complete this phase in one session. Otherwise, you should complete it as homework and show it to the instructor at the beginning of the next session.

Session 12: Logic Required by the Processes

- Upon approval of your initial design, you can begin detailing each process. At this point each process should have sufficient detail to match the requirements. You can review the details of each and determine if there are any elements that you need to address based on your instructor feedback.
- Once everything is finalized, you may begin coding your program. Create your project structure in Visual Studio and ensure that all your required files are properly setup.
- Begin coding your application according to your logic documentation. It is essential that you follow your plan.

Sessions 13 to 15: Apply Analysis and Design to Coding the Application

- Apply your problem solving skills.
- Code the various processes of your program according to specs.
- Add appropriate data input validation and error-handling.
- When coding your application, you may need to revise the logic (iterative development). Make sure you update your design documentation as required.
- Document any changes to your logic and design explaining why you decided to modify your approach. This must be submitted as part of your project submission.
- Validate the solution using test cases.

- Add any missing input validation or error-handling.
- Update the documentation.
- Submit the project.

IF YOU HAVE TIME

This project is very basic and does not include any complex logic. However, you may find that you did not require all the time allocated to creating the solution. If you have any time left over after having fulfilled all the basic requirements, you may want to attempt to add some additional features to this program. Some examples are listed below.

- ☐ Run multiple simulations in one go and make the computer improve the vehicle movements so that the simulation lasts longer (Research Genetic Algorithms).
- ☐ Provide GUI using JavaFX so that the user can see vehicles moving in real-time.
- ☐ Let the user position the vehicles themselves in the beginning of the application.

These additional options are not mandatory. You do not have to implement any of them. They are offered as suggestions only for enhancement purposes. and allow you the opportunity to experiment with adding additional functionality.

SUBMISSION INSTRUCTIONS

Your project must include the following:

- All source code, as well as all other files required for the proper functioning of your application.
- A project report containing:
  - ♦ a title page with your name, the submission date, and your instructor's name
  - ♦ a table of contents
  - ♦ the project specifications
  - ♦ all relevant documentation including diagrams, pseudocodes
  - ♦ the user manual (instruction guide for your instructor - this must include any test data that you use to test your application)
  - ♦ explanations of any additions that you made (where applicable)

We strongly recommend that you verify, using anti-virus software, that your submission contains no viruses.

Penalties

- A penalty of 5% will be deducted from your mark for each day your project is late.
- Any project submitted more than three calendar days late will receive a maximum mark of 60%.

- Any project containing a virus must be resubmitted and will receive a maximum mark of 60%.

Work must be submitted in the correct file type and be properly labelled as per the College naming convention:
NAME_COURSE_ASSIGNMENT. E.g. XuXiaLing_FM50D_A01.

GRADING CRITERIA

Assignment Value: 40**%**

Evaluation Elements	**% of mark**
Analysis and Design Part 1 with complete documentation showing appropriate architecture design required by the processes/methods (pseudocode and class diagrams) approved by instructor.	30
Design requirements specified in the DETAILED GUIDELINE/REQUIREMENTS	25
Functional requirements specified in the DETAILED GUIDELINE/REQUIREMENTS	20
Documentation preparation and update (if you modify your architecture or logic, it must be documented)	5
Input validation and error handling using controls and code	15
Error free and bug free application (application properly debugged)	5
TOTAL	**100**