**OOSE Lab Exam Problem**

**Coursework Overview**

*You are being tasked with implementing several sub-systems for a car racing game. Your work will need to be used by other programmers in the team, so it is essential that you follow good design practice in this assessment. You are going to handle the creation of race cars, their associated behaviours, and the weather in the simulation.*

*You should be aware that this exercise is less structured than other work you have done so far in OOSE – you will be writing a substantial amount of code from scratch and need to make decisions about how to properly follow Object Oriented design principles and what design patterns you need to use to address problems. Getting the work to run and pass basic functionality tests is less than half the battle here, most of the marks are on offer for the quality of your code, and how easily other programmers can work with it.*

**The Racing Game**

You are part of a team making a video game simulating car racing. Car races take place on a racetrack, with a specific set of race cars, each with their own unique behaviours.

There are a range of elements that go into making the videogame simulation including the racetrack, the commentary team, pit stop crews, crowds, etc. Behind the scenes, the game also needs to handle graphics and models and player input, but in this scenario, you are only concerned with 1) managing a small subset of these problems and 2) ensuring that other developers can extend your code to make use of it in their work without modifying it. Your responsibilities are divided into then following main areas:

*RaceWeather*

You should build on the class called RaceWeather. This class will be responsible for storing the state of the weather and updating other, interested classes about changes in the weather. You have a skeleton for the class provided that:

* Represents weather as an enumerated WeatherType: DRY, MISTY, or WET
* Declares a changeWeather() method that changes the status of the weather in a cycle. DRY will become MISTY, MISTY will become WET and WET goes back to DRY.

You should add the following functionality:

* Allow other classes to register their interest in being updated about the weather and unregister it.
* Update classes that are interested in changes in the weather when the weather changes.

Bear in mind that you will be assessed on:

* Whether other programmers can make use of your RaceWeather class without changing its code.
* Whether the class can handle any number of interested instances.
* Whether the class can handle different types of interested class.

Hints:

* Implementation is not the challenge; the hard part is ensuring that other coders on the team can extend your code without modifying it.
* Think about the design patterns you have been taught - you will need to modify the classes’ declaration to implement the correct design pattern.
* You will need to create other classes as part of this process.

*RaceCar*

You should build on the class called RaceCar. This class represents a single Race Car in your game. You have a skeleton for the class provided that

* Stores the name of the RaceCar (String), distance travelled in the race (int initially at 0), and the manufacturer (String).
* getStatus() returns a string documenting the progress of the car in the race.

You should add the following functionality:

* The RaceCar should initially be able to have one of two different types of driving behaviour based on whether the car is driven cautiously or being driven fast. The driveCar() function should return the appropriate int value based on the weather and behaviour taken from the table below:

|  |  |  |
| --- | --- | --- |
| **Weather Condition** | **Cautious Distance on driveCar()** | **Fast Distance on driveCar()** |
| *DRY* | 10 | 20 |
| *MISTY* | 10 | 5 |
| *WET* | 10 | 8 |

* Awareness of the weather – you should enable each instance of RaceCar to be updated when the RaceWeather changes and store the state of the weather in a WeatherType.
* The changeDrivingBehaviour() function should be able to alter the cars driving behaviour at runtime changing it between cautious and fast initially – you will need to determine how this should be done and the parameters that the method takes in.

Bear in mind that you will be assessed on the following areas:

* Other developers should be able to implement their own behaviours for the car based on the weather and use them without having to modify your RaceCar code.

Hints:

* The same hints given for RaceWeather also apply here.
* You’re race cars will need to make use of the RaceWeather instance – make sure you follow the appropriate design pattern!
* You will need to alter the constructor for the RaceCar class.

*Race*

The Race class provides a skeleton for the following:

* The Race class stores racers, an ArrayList of the cars currently racing, an instance of RaceWeather that handles changes in the weather, and a racetrackDistance int that specifies how many meters the cars racing need to travel to complete the race.
* progressWeather() calls the changeWeather() function of the RaceWeather object and should not need to be edited.
* main(String args) creates a race object, makes four cars and, while the race is not over, calls the raceStep() function. When the race completes, it prints out the final outcome taken from raceOutcome.

You should edit the class to implement the following functionality:

* raceStep() should move the race forward one “step” by calling the driveCar() method of all cars racing and printing out their status.
* isFinished() should return true if a RaceCar has driven beyond the racetrackDistance and false otherwise.
* getOutcome() should return a string of the format:

“WINNER: ” + Winning car name + “, “ + Winning car make

In the event more than one car has finished the race, the winner is the one that drove the furthest. If two or more cars have driven the same distance, return the string “TIE”.

* makeFourCars() should populate the racers ArrayList with the following four cars:

|  |  |  |
| --- | --- | --- |
| **Car Name** | **Make** | **Driving Type** |
| *Car 1* | *Fjord* | *Cautious* |
| *Car 2* | *Fjord* | *Fast* |
| *Car 3* | *Furbi* | *Cautious* |
| *Car 4* | *Furbi* | *Fast* |

* changeCarBehaviour should take in an integer corresponding to the position of a RaceCar instance in the racers ArrayList and alter its behaviour. The method can take additional parameters as you deem necessary.

*Final challenge*

The final challenge for this work is to encapsulate the use of the new keyword when creating RaceCars in Race. Think about the correct approach to solving this problem with a design pattern if you knew that your code was going to be extended with many different makes of car and driving types going forward.

Implement a solution to this in the function makeFiveCars() – the solution should not need to have any new keywords in it and instead encapsulate those calls outside of the Race class.

**Submission details**

Your final submission should be in the form of a GitLab upload.

* Include all the source files.
* Include a diagram showing how your classes interact with each other.
* Provide an explanation of the design patterns you have followed and how you have used them (this should be under 200 words).

**Marking Scheme**

Marks are given as a percentage of the final grade for the Lab assignment. Note that this is a software engineering assessment - if code cannot be understood by the marker, then this is an issue on your part and will cost you marks.

|  |  |  |  |
| --- | --- | --- | --- |
| **Part One: Functionality –** Marks are awarded for successfully making the program function, regardless of how this has been done | | | *25%* |
| *5%* | *Weather Update* | When race’s progressWeather method is called, each instance of RaceCar in the race should be updated about the change in weather conditions. | |
| *5%* | *Drive Fast and Drive Cautious* | When each car’s drive method is called, it should move the correct distance in meters given the prevailing weather condition for the race. | |
| *5%* | *Changing Driving* | When the changeDrivingBehaviour method is called, the drive behaviour of a car instance should be altered. | |
| *10%* | *Correct Outcome* | After the main method of the Race class is called (and its attendant calls to makeCars, looping based on the isFinished method, and raceStep) the correct car should win the race. | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Part Two: Code Quality –** Marks are awarded for how cleanly you have written code in line with guidance in class – you lose 1% for each violation of conventions identified in each area. | | | *15%* |
| *5%* | *Encapsulation* | Good use of public and private variables and appropriate use of getter and setter methods. | |
| *5%* | *Clean Code* | Spelling mistakes free code implementation. | |
| *5%* | *Variable Naming* | You follow good variable naming conventions – you do not use single letters or names that are not descriptive in your code. | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Part Three: Design Quality –** Marks are awarded based on how easily another developer can make use of your code or how easily you could go back and handle changes to the code. | | | *60%* |
| *20%* | *Design Pattern Selection* | You have selected appropriate design patterns in the code, and you have named them in your submission readme. | |
| *20%* | *Design Pattern Operation* | Your code should make use of the design patterns in the correct way that has been described and not, for example, state you use a design pattern while achieving the intended functionality without following it. | |
| *20%* | *Code Extensibility* | Where the problem highlights changes will be made to the system you implement going forward, another developer can easily make those changes. They do not need to edit code in RaceWeather, RaceCar or Race (except for the main method in Race) and do not need to be aware of the inner workings of those classes. | |