Software Architecture Design and Implementation
COSC 2391/2401 Semester 1, 2018
Casino Style Dice Game

Assignment Part 1: Console Implementation (25 marks)

NOTE: The provided *Javadoc* and commented interface source code is your main specification, this document only serves as an overview.

This assignment requires you to implement a game engine and console based user interface (logging output only, no user interaction required) for a casino style dice game. The rules are simple, each player places a bet and then rolls two dice before the house rolls against the players .. Highest number (sum of the two dice) wins! A draw is a no contest and the bet is returned to the drawing player.

NOTE: players only play against the house not against each other. Also, do not worry about modelling a real Casino "craps" game with its more complex rules. The focus here is on the implementation using a simple, highest dice sum wins.

<u>HOW TO GET STARTED:</u>

For this assignment you are provided with a skeleton eclipse project (`DiceGame.zip`) that contains a number of interfaces that you must implement to provide the specified behaviour as well as a simple client which will help you get started.

The provided *Javadoc* documentation (load `index.html` from `DiceGame/docs/` into a browser to view), and commented interface source code (in the provided package `model.interfaces`) is your main specification, this document only serves as an overview.

NOTE: You may copy and extend the provided console client code to facilitate testing but must ensure that the original unaltered code can still execute since we will use our own test client to check your code which is strictly based on the interfaces (this is the point of having interfaces after all!)
i.e. DO NOT CHANGE ANY OF THE INTERFACES ETC.

You do not need to provide any console input, all your test data can be hard coded as in the provided `SimpleTestClient.java`

Implementation Specifications

Your primary goal is to implement the provided `GameEngine`, `Player`, `GameEngineCallback` and `DicePair` interfaces, in classes called `GameEngineImpl`, `SimplePlayer`, `GameEngineCallbackImpl` and `DicePairImpl`. You must provide the behaviour specified by the javadoc comments in the various interfaces and the generated javadoc `index.html`. The imports in `SimpleTestClient.java` show you which packages these classes should be placed in.

More specifically, you must provide appropriate *constructors* (these can be determined from `SimpleTestClient.java` and are also documented in the relevant interfaces) and method implementations (from the four interfaces) in order to ensure that your solution can be complied and tested ***without modifying*** the provided `SimpleTestClient.java`[1] (although you can and should extend this class to thoroughly test your code). A sample output trace (`OutputTrace.txt`) is

[1] A common mistake is to change the imports (sometimes accidentally!) Therefore, you MUST NOT change the imports and must place the class implementations in the expected package so that we can test your code with our own testing client.

provided to help you write correct behaviour in the `GameEngineImpl` which in turn calls the `GameEngineCallbackImpl` class to perform the actual logging. You should follow the exact output format although we will not be checking down to individual commas or spaces!

Your client code (`SimpleTestClient.java` and any extended derivatives) should be separate from, and use, your `GameEngineImpl` implementation via the `GameEngine` interface. Furthermore, your client should NOT call methods on any of the other interfaces/classes since these are designed to be called directly from the `GameEngine`[2]

The main implementation classes `GameEngineImpl` and `GameEngineCallbackImpl` are described in more detail below. The `SimplePlayer` and `DicePairImpl` are relatively straightforward data classes and should not need further explanation for their implementation (beyond the comments provided in the respective interfaces).

GameEngineImpl class

This is where the main game functionality is contained. All methods from the client are called through this class (see footnote). Methods in the supporting classes should only be called from `GameEngineImpl`.

The main feature of this class that is likely different to previous code you have written is that the `GameEngineImpl` does not provide any output of its own (i.e. it SHOULD HAVE NO `println()` or `log()` statements). Instead it calls appropriate methods on the `GameEngineCallback` as it runs (see below) which is where all output is logged to the console for assignment part 1.

This provides a nice level of isolation and will allow you to use your `GameEngineImpl` unchanged in assignment 2 when we add a graphical AWT/Swing use interface!

NOTE: To support multiple players, your `GameEngineImpl` must maintain a collection (or array) of `Players` and a collection (or array) of `GameEngineCallbacks`. When a callback method should be called this must be done in a loop iterating through all callbacks. Note that each callback receives the same data so there is no need to distinguish them (i.e. they are all the same and not player specific). `SimpleTestClient.java` gives an example for two players and shows it is trivial to add more (simply increase the array size by adding to the initialiser).

GameEngineCallbackImpl class

The sole purpose of this class is to support the user interface which in assignment part 1 consists of simple console output. Therefore all this class needs to do is log data to the console from the parameters passed to its methods. Apart from implementing the logging (we recommend `String.format()` here) the main thing is to make sure you call the right method at the right time! (see below). You should also as much as possible make use the of the overidden `toString()` methods you will implement in `SimplePlayer` and `DicePairImpl` since this will simplify the logging!

The only class that will call the `GameEngineCallbackImpl` methods is the `GameEngineImpl` class. For example as the `rollPlayer(…)` method is executing in a loop it will call the `intermediateResult(…)` method on the `GameEngineCallbackImpl` (via the `GameEngineCallback` interface). Details of the exact flow and where `GameEngineCallback` methods should be called are provided in the `GameEngineImpl` source code and associated Javadoc.

[2] This is because we will be testing your code with our own client by calling the specified `GameEngine` methods. We will not call methods on any other classes and therefore if your `GameEngineImpl` code expected other methods to be called from the client (rather than calling them itself) it won't work!

IMPORTANT: The main thing to watch out for (i.e. "gotcha") is that this class should not manage any game state or implement any game based functionality which instead belongs in the `GameEngineImpl`. The core test here is that we should be able to replace your `GameEngineCallbackImpl` with our own (which obviously knows nothing about your implementation) and your `GameEngineImpl` code should still work. This is a true test of encapsulation and programming using interfaces (i.e. to a specification) and is one of the main objectives of this assignment!

IF YOU DO NOT FOLLOW THE NOTE ABOVE YOUR CODE WILL NOT EXECUTE PROPERLY WITH OUR TEST HARNESS AND YOU WILL LOSE MARKS! PLEASE DON'T GET CAUGHT OUT .. IF IN DOUBT ASK, WE ARE HAPPY TO HELP :)

IMPLEMENTATION TIPS

Before you start coding make sure you have thoroughly read this overview document and carefully inspected the supplied Java code and Javadoc documentation. It might not all make sense yet but the more carefully you read before beginning coding the better prepared you will be!

1. Start by importing the supplied Java project `DiceGame.zip`. It will not compile yet but this is normal and to be expected.
2. The first step is to get the code to compile by writing a minimal implementation of the required classes. Most of the methods can be left blank at this stage, the idea is satisfy all of the dependencies in `SimpleTestClient.java` that are preventing successful compilation. Eclipse can help automate much of this with the right click *Source ...* context menu but it is a good idea to write at least a few of the classes by hand to make sure you are confident of the relationship between classes and the interfaces that they implement. It will also help familiarise you with the class/method names and their purpose. I have already provided a partial implementation of `GameEngineCallbackImpl` showing the use of the Java logging framework but you will need to complete it by implementing the missing methods.
3. When writing the `SimplePlayer` class you will need a 3 argument constructor for the code to compile. You could leave this blank at this stage but might as well implement it by saving the parameters as instance variables/attributes. In fact you might as well implement the methods while you are there since they are straightforward. **HINT**: In my (Caspar's) solution most of the methods of `SimplePlayer` are one liners, except for `placeBet()` which is a few more lines since it requires an `if` statement!
4. Once the code can compile you are ready to start implementing the `GameEngineImpl`. You can start with the simple methods like `addPlayer()` etc. and then when ready move on to one of *roll* methods below.
5. The roll methods involve the most code but even these are fairly small. In fact this assignment doesn't require a lot of lines of code, it is about understanding concepts and putting them into place!
6. I would suggest focusing first on the `rollPlayer(…)` method and having this call `result(…)` on the `GameEngineCallBackImpl` (via the `GameEngineCallback` interface). You can ignore the delay for now and use log/println statements and the debugger to help you.
7. Once you get this far you have the basic structure underway so you can finish by implementing the `rollHouse()` method (this should be able to share most of its code with `rollPlayer` so using private helper methods to avoid code duplication is the trick here).
8. Finally add in the `intermediateResult(…)` calls into the `GameEngineImpl` and implement the delay in the roll methods and you are pretty much done!
9. Copy `SimpleTestClient` (you can call it `MyTestClient` for example) and update it with some more through testing code, debug as necessary to fix any issues and you are done :)

FINAL POINTS

1. You should aim to provide high cohesion and low coupling.
2. You should aim for maximum encapsulation and information hiding.
3. You should comment important sections of your code.
4. You should CAREFULLY read the instructions and supporting code and documents. This assignment is intended to model the process you would follow writing real industrial code.
5. IF IN DOUBT ASK EARLY!
6. Marking emphasis will be on the quality of your code as well as your ability to implement the required functionality.

Submission Instructions

- You are free to refer to textbooks and notes, and discuss the design issues (and associated general solutions) with your fellow students on Canvas; however, the assignment should be your own individual work.
- You may also use other references, but since you will only be assessed on your own work you should NOT use any third party packages or code (i.e. not written by you) in your work.

The source code for this assignment (i.e. complete compiled **Eclipse project**[3]) should be submitted as a .zip file by the due date. You can either zip up the project folder or use the Eclipse option `export->general->archive`. You will be provided with further submission instructions on Canvas before the deadline.

Due 9:00AM Mon. 16th April 2018 (25%)

Late submissions are handled as per usual RMIT regulations - 10% deduction (2.5 marks) per day. You are only allowed to have 5 late days maximum.

[3] You can develop your system using any IDE but will have to create an Eclipse project using your source code files for submission purposes.