Further Programming COSC 2391/2401
Semester 1, 2019
Casino Style Gaming Wheel

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. You should also regularly follow the Canvas assignment discussion board for assignment related clarifications and discussion.

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 *gaming wheel* that is loosely based on Roulette but with a simplified betting and scoring system.

The rules are simple, each player places a bet with three different bet types, each with the odds described below. The house then spins the wheel and the result is the numbered and colored **SLOT** the ball lands in when it stops.

After the wheel spins the win or loss amount is applied to the players point balance as follows.

Bet Type	Odds	Description	Example
Red	1 to 1	Win on red slot/number	Bet 100 win or lose 100
Black	1 to 1	Win on black slot/number	As above
Zeros	WHEEL_SIZE / 2 – 1 to 1 for win, 1 to 1 for loss	Win on green 0 or 00 slots	Bet 100 win 1800 or lose 100*

Table 1 – Bet types and odds to be implemented

* The wheel size is 38 containing numbers 1-36 and the two green zeros, so the odds for a win when betting on zeros is 18 to 1. See the image in the Appendix that shows the order and coloring of slots on the wheel which must be implemented correctly to show the correct order for a 'spin'.

NOTE1: You do not need to model a real Casino "Roulette" game with its more complex rules, just the three simple bet types in the table above.

NOTE2: Player points are not changed by placing a bet, they are only changed after the wheel has spun and the win/loss has been determined.

HOW TO GET STARTED:

For this assignment you are provided with a skeleton eclipse project (`WheelGame.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. There are also two provided `enum` types in the package `model.enumeration` which need to be completed (in the case of `BetType`) and used appropriately.

The provided *Javadoc* documentation (load `index.html` from `WheelGame/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 add to the provided console client code to facilitate more thorough testing. You must however 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 and provided enum classes (this is the point of having interfaces and a specification after all!)

i.e. DO NOT CHANGE ANY OF THE INTERFACES ETC.

The supplied project also contains code that will validate your interfaces for the main four classes you must write (see implementation specification below), and will warn you if you have failed to implement the required interfaces or have otherwise added any **non-private** methods that break the public interface contract. By default the validator lists all the expected methods as well as your implemented methods so you can use the output of this to find problems if you fail the validation.

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 `Slot` interfaces, in classes called `GameEngineImpl`, `SimplePlayer`, `GameEngineCallbackImpl` and `SlotImpl`. 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. Additionally, you are required to implement the `applyWinLoss()` method overrides in the `BetType` enum in order to implement the win/loss behaviour described in **Table 1**.

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 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 since it is clearly specified and this facilitates automated testing.

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 `SlotImpl` 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`.

---

[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.

[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!

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 other than for debugging and these should be commented or removed prior to submission). 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 good 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: Your `GameEngineImpl` must maintain a collection of `Players` AND a collection 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 three players with each bet type and shows it is trivial to add more (simply increase the array size by adding to the initialiser).

GameEngineCallbackImpl class

The main purpose of this class is to support the user interface (view) which in assignment part 1 consists of simple console/logging 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 `SlotImpl` since this will simplify the logging!

The only class that will call the `GameEngineCallbackImpl` methods is the `GameEngineImpl` class. For example as the `spin(…)` method is executing in a loop it will call the `nextSlot(…)` 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.

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 take a bit of work but the more carefully you read before beginning coding the better prepared you will be!

1. Start by importing the supplied Java project `WheelGame.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. We 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/fields. 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 `setBet()` which is a few more lines since it requires some basic validation.
4. Once the code can compile you are ready to start implementing the `GameEngineImpl`. You can start with the simple methods like `addPlayer(), addGameEngineCallback()` etc.
5. The next functionality to work on is creating the gaming wheel as a `Collection` of Slot which is returned to the client which logs it so you can CAREFULLY check correctness against the `OutputTrace.txt`.
6. The `spin()` method involves the most code but even this is fairly small if well structured. In fact, this assignment doesn't require a lot of lines of code, it is about understanding concepts and putting the pieces together to make it work!
7. When implementing `spin(…)` the main focus is calling `nextSlot(…)` and `result(…)` on the `GameEngineCallBackImpl` (via the `GameEngineCallback` interface). You can ignore the delay for now and use temporary log/println statements and the debugger to help you compare against the `OutputTrace.txt`.
8. Once you get this far you have the basic structure underway so you can finish by implementing the `applyWinLoss()` methods in `BetType` and calling them from `calculateResult()` in the `GameEngine`. Again, use the log output of the client and `OutputTrace.txt` to check for correctness.
9. Finally add in the *logging* calls into the `GameEngineImpl` and implement the delay in the spin method and you are pretty much done!
10. 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 rigorously avoid code duplication.
4. You should comment important sections of your code remembering that clear and readily comprehensible code is preferable to a comment.
5. Since this course is concerned with OO design you should avoid the use of Java 8+ lambdas which are a functional programming construct.
6. 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.
7. IF IN DOUBT ASK EARLY!
8. Marking emphasis will be on the quality of your code as well as your ability to implement the specified 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 or on Canvas; however, the assignment should be your OWN INDIVIDUAL WORK and is NOT a group assignment.
- 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.

IMPORTANT: SUBMISSIONS OF A ROULETTE OR ANY OTHER GAME WHICH DO NOT IN ANY WAY ADHERE TO THE SPECIFICATION AND PROVIDED CODE WILL RECEIVE A **ZERO MARK**

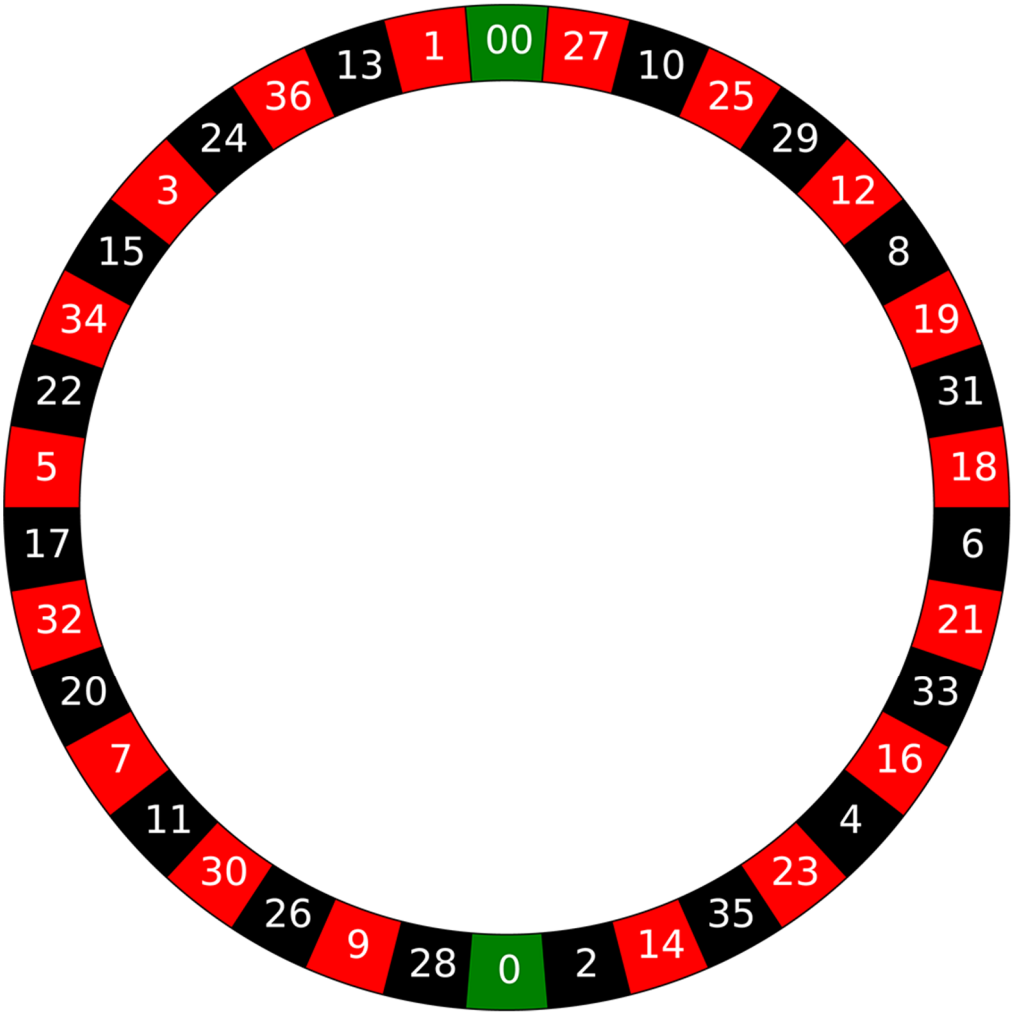
Due 6:00PM Fri. 12th April 2019 (25%)

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.

APPENDIX

Layout of the gaming wheel as per the logs of *OutputTrace.txt*

NOTE: You do not need to implement any graphics or UI for assignment 1, this will done according to a precise specification in assignment 2.



[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.

ATTRIBUTION:

This wheel image is based on
https://commons.wikimedia.org/wiki/File:Basic_roulette_wheel.svg
The original was licensed under Creative Commons CC0 1.0 Universal (CC0 1.0) Public Domain Dedication

The provided file Basic_roulette_wheel_1024x1024.png is a modification by Caspar to replace the duplicated 35 with a 25 in the correct position and is the file that should used for this assignment!

It is also provided under the original license