School of Science

Further Programming COSC 2391/2401, S2, 2020
Casino Style Card Game

Assignment Part 1: Console Implementation

	Assessment Type: Individual assignment; no group work. Submit online via Canvas→Assignments→Assignment 1. Marks are awarded for meeting requirements as closely as possible according to section 2 and the supplied rubric. Clarifications/updates may be made via announcements/relevant discussion forums.
	Due date: Due 6:30PM Fri. 28th August 2020. Late submissions are handled as per usual RMIT regulations - 10% deduction (4 marks) per day (Saturday and Sunday count as separate days). You are only allowed to have 5 late days maximum unless special consideration has been granted.
	Weighting: 40 marks (40% of your final semester grade)

1. Overview

NOTE: The separately provided Javadoc and commented interface source code is your main specification, this document serves as a starting point. 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 card game that is loosely based on Black Jack but is for the gambler in a hurry that doesn't want to think too hard and wants to trust in luck alone without having to worry about statistics!

The rules are simple, the player places a bet and then receives a set of cards from the dealer (from a 28 card "half"* deck containing the cards 8, 9, 10, J, Q, K, A) until they bust by exceeding the limit of 42[1] (or reach 42 exactly). The final score is the sum of the cards prior to the final card that caused the bust. *"Half" deck in quotes since it is not exactly half a standard 52 card deck!

The house then deals on their own behalf against the players .. Highest score wins! A draw is a no contest and the bet is returned to the player.

2. Assessment Criteria

This assessment will determine your ability to implement Object-Oriented Java code according to a formal Javadoc specification. In addition to functional correctness (i.e. getting your code to work) you will also be assessed on code quality. Specifically:

- You should aim to provide high cohesion and low coupling.
- You should aim for maximum encapsulation and information hiding.
- You should rigorously avoid code duplication.
- You should comment important sections of your code remembering that clear and readily comprehensible code is preferable to a comment.
- Since this course is concerned with OO design you must not use Java 8+ lambdas which are a functional programming construct.
- 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.
- IF IN DOUBT ASK EARLY!

[1] The number 42 is also an amusing pop-culture reference, do you recognise it?

3. Learning Outcomes

This assessment is relevant to the following Learning Outcomes:

CLO1: **Explain** the purpose of OO design and **apply** the following OO concepts in Java code: inheritance, polymorphism, abstract classes, interfaces and generics.

CLO2: **Describe and Document Diagrammatically** the OO design of the Java Collection Framework (JCF) and **apply** this framework in Java code.

CLO4: **Demonstrate Proficiency** using an integrated development environment such as Eclipse for project management, coding and debugging.

4. Assessment details

Note: Please ensure that you have read sections 1-3 of this document before going further.

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 card game that is loosely based on Black Jack but is for the gambler in a hurry that doesn't want to think too hard and wants to trust in luck alone without having to worry about statistics!

The rules are simple, for each round, the player **places a bet** of a chosen amount up to their maximum available points (see NOTE1 below) and then receives a set of cards from the dealer (from a 28 card "half"* deck containing the cards 8, 9, 10, J, Q, K, A of all suits) until they **bust** by exceeding the limit of 42 (or reach 42 exactly). There is no bust card if the player scores exactly 42 otherwise the final score is the sum of the cards prior to the bust card (see **scoring** below). * "Half" deck in quotes since it is not exactly half a standard 52 card deck!

The house then deals on their own behalf against the players .. Highest score wins! A **draw,** where both the player and house score the same result, is a no contest and the bet is returned to the player. The game then proceeds to the next round where the process of betting and dealing continues.

SCORING
The rules for scoring the cards are also simple with no action required by the player. An Ace is always worth 11 (not 1!), Jack, Queen and King are 10 and the other cards are worth their face value. i.e. an eight of spades is worth 8 points.

NOTE1: Player points are not changed by placing a bet, they are only changed after the House has dealt and the win/loss has been determined.

NOTE2: Players are only competing against the house to win more points, with their win loss determined only by their own and the House's dealt cards, not by the other players. Also, do not worry about modelling a real Casino "Black Jack" game with its more complex rules such as splitting etc. The focus here is on the implementation using a simple, highest card sum wins.

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

For this assignment you are provided with a skeleton eclipse project (`CardGame.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 `CardGame/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 (this is the point of having interfaces 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 output 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 `PlayingCard` interfaces, in classes called `GameEngineImpl`, `SimplePlayer`, `GameEngineCallbackImpl` and `PlayingCardImpl`. 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`[2] (although you can and should extend this class to thoroughly test your code). A sample output trace (`OutputTrace.pdf`) is provided to help you write correct behaviour in the `GameEngineImpl` which in turn calls the `GameEngineCallbackImpl` class to perform the actual logging. You MUST follow the **exact** output format!

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`[3]

The main implementation classes `GameEngineImpl` and `GameEngineCallbackImpl` are described in more detail below. The `SimplePlayer` and `PlayingCardImpl` 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 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 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 (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 overridden `toString()` methods you will implement in `SimplePlayer` and `PlayingCardImpl` since this will simplify the logging!

---

[2] 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.
[3] 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 only class that will call the `GameEngineCallbackImpl` methods is the `GameEngineImpl` class. For example as the `dealPlayer(…)` method is executing in a loop it will call the `nextCard(…)` 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 `CardGame.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/attributes. In fact you might as well implement the methods while you are there since they are straightforward. **HINT**: In my (Caspar's) solution many of the methods of `SimplePlayer` are one liners!
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 *deal* methods below.
5. The deal methods involve the most code but even these are fairly small if well written. 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 `dealPlayer(…)` method and having this call `nextCard(…)` and `bustCard(…)` 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.
7. Once you get this far you have the basic structure underway so you can finish by implementing the `dealHouse()` method (this should be able to share much of its code with `dealPlayer` so using private helper methods to avoid code duplication is the trick here).
8. Finally add in the *logging* calls into the `GameEngineImpl` and implement the delay in the deal 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 :)

5. Referencing and third party code exclusion

- You are free to refer to textbooks or 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.

6. Submission format

The source code for this assignment (i.e. complete compiled **Eclipse project**[4]) should be submitted as a .zip file by the due date. You should use the Eclipse option export->general->archive to create the zip file for submission.

The project should be called **CardGame**, which is the original name of the project in the start-up code.
You can include a README.TXT file in the root of the project providing any details about your project and running your submission although this should not be necessary if you have adhered to this specification!
Once submitted you are advised to ensure you have submitted the correct file by downloading your submission from Canvas and importing your project into a fresh Eclipse workspace. You can use Switch Workspace to achieve this.
Any errors or inconsistencies when importing your project **will** result in a loss of marks.

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

7. Academic integrity and plagiarism (standard RMIT warning)

NOTE: Any discussion of referencing below in the standard RMIT policy is generic and superseded by the third-party code exclusion in section 5.

Your code will be automatically checked for similarity against other students' submission so please make sure your submitted assignment is entirely your own work.

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e. directly copied), summarised, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods,
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from Internet sites.

If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own.
RMIT University treats plagiarism as a very serious offence constituting misconduct. Plagiarism covers a variety of inappropriate behaviours, including:

- Failure to properly document a source
- Copyright material from the internet or databases
- Collusion between students

For further information on our policies and procedures, please refer to the University website.

8. Assessment declaration

When you submit work electronically, you agree to the assessment declaration.

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