Software Architecture Design and Implementation
COSC 2391/2401 Semester 2, 2018
Casino Style Card Game

Assignment Part 2: AWT/Swing UI Implementation (25 marks)

This assignment requires you to build an AWT/Swing graphical user interface for the Card Game that reuses your code from assignment part 1.

You should build on top of your existing assignment part 1 functionality and should not need to change your existing Game Engine code if it was correct in assignment part 1. You can seek help in the tutelab/consultation to finish assignment part 1 if necessary (since it is a core requirement of this course that you should be able to work with basic OO classes and interfaces). Furthermore, you will require a solid understanding of classes/interfaces to write effective AWT/Swing!

As part of your new implementation you must write a `GameEngineCallbackGUI` class that is added to the `GameEngine` via the existing `addGameEngineCallback()` method. This class will be responsible for managing all of the graphical updates as the game is played. NOTE: this class should not actually implement the UI functionality but instead for cohesion it should call methods on other classes, especially the view. To state another way, it must be the entry point for any UI code resulting from game play, in order to avoid coupling between the `GameEngine` and the UI which is counter to the original specification.

This `GameEngineCallbackGUI` is to be added **in addition to** the console based `GameEngineCallbackImpl` from assignment 1 in order to demonstrate that your `GameEngine` can work with multiple callbacks i.e. both (all) callbacks are called for any relevant operation such as `dealHouse()/dealPlayer()`. NOTE: This is the main reason the assignment part 1 specification required the `GameEngineImpl` to handle multiple callbacks!

AWT/Swing User Interface

You are to develop an AWT/Swing user interface that implements the following basic functionality:

Add players (including name and initial betting points balance)
Place a bet (per player) and display an error message for invalid bets
Player deals (per player) .. each card is shown in real-time as it is dealt
House deals (automatically after all players have bet/dealt) .. each card is shown in real-time as it is dealt
Display results including updated player balances after house deal

NOTE: You should set your delay to 1 second (1000ms) for testing and submission.

It is up to you how to design the layout and appearance of your interface and you should focus on clarity and simplicity rather than elaborate design. However, to demonstrate competency you should include at least one each of the following.

A pull-down menu (like the standard File menu in Eclipse)
A dialog box
A toolbar
A status bar
A switchable per player panel which represents the dealt cards (see UI Implementation/Limitations below)
A summary panel which is always visible which shows player names and their current points balance

Marking emphasis will be on the quality of your code and your ability to implement the required functionality as well as basic usability of the UI (based on the usability attributes described in the lecture notes).

Your code should be structured using the **Model View Controller** pattern (MVC) where the `GameEngineImpl` from assignment part 1 serves as the model, the listeners represent the controllers as separate classes (each in a separate file placed in the `controller` package), whereas the `GameEngineCallbackImpl` and new `GameEngineCallbackGUI` are part of a `view` package (or sub-packages), as are any additional UI classes such as frames, dialogs, components, menus etc. Furthermore, you must NOT use static referencing (e.g. no use of the Singleton pattern) and therefore all references required by different classes must be passed as parameters (e.g. via constructors).

IMPORTANT: All of your GUI code (MVC views including the `GameEngineCallbackGUI` and controllers) should be separate from your `GameEngineImpl` implementation (MVC model). Furthermore, all of your GUI code should call your `GameEngineImpl` implementation via the `GameEngine` interface only. You should not add any UI code to the `GameEngineImpl` with the primary test being that your UI code should work with any `GameEngine` implementation (for example our own implementation we will use for testing). i.e. your UI should not require anything additional from the assignment 1 `GameEngineImpl` if implemented correctly according to the Javadoc and assignment 1 specification. Finally, your `GameEngineImpl` should still pass the Validator checks from assignment 1.

NOTE: It is a core learning outcome of this assignment to demonstrate that encapsulation and programming to interfaces provides complete separation such that code written by independent parties can work together seamlessly without any change!

UI Implementation/Limitations

You must only **display** the UI for a single player at a time, whereby the visible/active player can be directly selected from a list or combo box. The dealt cards are the most important part of part of the player UI.

The card panel must have the appearance of animation but can be as simple as a `JLabel` that is updated for each new intermediate result received from the `GameEngineCallback` methods, but it will likely be more rewarding and impress your friends if you can find some nice creative commons or free for personal use card graphics :) You cannot however simply replicate the logging behaviour of assignment 1 in a `JTextArea` or similar component.

More importantly, since the UI is per player, if a player is currently visible and being dealt cards, when you switch to another player the UI should stop animating the dealt cards and should instead display the last cards dealt to the player you switched to. You can however assume that only one player will be dealt to at a time so you do not have to handle concurrency in the Game Engine.

All this behaviour is readily facilitated by the `GameEngineCallback` which receives a `Player` instance as a parameter, so as long as your GUI keeps track of the current player it can receive updates for all players and display only the current/relevant player in the GUI, while your `GameEngineCallbackImpl` from assignment 1 will log all players to the console.

NOTE: Although threads will be covered towards the end of the semester, AWT/Swing requires all calls to the UI (any methods in AWT/Swing such as creating a component, laying out, or updating) to be done on the AWT Event Dispatch Thread. Furthermore, since the `dealPlayer(…)/dealHouse(…)` methods of `GameEngineImpl` execute in a loop with a delay it is necessary to run them in a separate thread so they do not lock up the UI.

To achieve this you can use the code below. You don't need to know exactly how this works for now (although the API docs are useful here and we will also cover in class) but hopefully you are able to identify how the code is simply using anonymous inner classes to execute some code on a separate thread!

To call a GameEngineImpl method (such as rollPlayer()) on a separate thread.

```
new Thread()
{
   @Override
   public void run()
   {
      //call long running GameEngine methods on separate thread
   }
}.start();
```

To update the GUI from the callback ..

```
SwingUtilities.invokeLater(new Runnable()
{
   @Override
   public void run()
   {
      // do GUI update on UI thread
   }
});
```

Implementation Details

IMPORTANT: As with assignment one you must not change any of the interfaces. You may implement any other helper classes that you need to. A correct implementation should not require any changes to the `GameEngineImpl`, only the addition of new AWT/Swing classes to build the UI, in addition to a new `GameEngineCallbackGUI`.

To demonstrate cohesion and correct OO techniques, all UI code must be written carefully by hand. You can use builder tools (such as NetBeans) to aid prototyping but all final code must be written by hand. You **will lose marks** for if you submit generated code! You will also be breaching the third party code requirement below.

SUMMARY

1. You should use MVC implementation for your system.
2. You should write all your listeners as separate controller classes in the `controller` package.
3. You must not use any static referencing (e.g. no use of the Singleton pattern permitted).
4. You should aim to provide high cohesion and low coupling.
5. You should aim for maximum encapsulation and information hiding.
6. You should rigorously avoid code duplication.
7. You should comment important sections of your code remembering that clear and readily comprehensible code is preferable to a comment.
8. Since this course is concerned with OO design you should avoid the use of Java 8 lambdas which are a functional programming construct.
9. You should use extra threads where necessary (based on the sample code provided above) to ensure smooth UI operation.
10. You should handle all exceptions with a sensible error message but do not need to handle them in the GUI (a console log is sufficient)

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, or any generated code (e.g. UI builders) (i.e. not written by you) in your work.

The source code for this assignment (i.e. complete compiled **Eclipse project**[1]) should be submitted as a .zip file by the due date using the Eclipse option `export->general->archive`.

Due 9:00AM Mon. 15th October 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.

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