COMP1721 Object-Oriented Programming

Coursework 2

1 Introduction

This assignment assesses your ability to create classes that inherit from existing classes.

Your task is to implement a simulation of the card game **Baccarat**—specifically, the simpler 'Punto Banco' variant of the game. To assist you, we have provided four Java classes. Two of these classes, `Card` and `CardCollection`, will form the basis of your solution. The other two are examples that may be useful if you choose to tackle the 'super-advanced' task.

Please note: an absolute requirement of this assignment is that you should not alter the definitions of `Card` and `CardCollection` in any way. The only change allowed is to add a package declaration at the top of each file if you want.

2 Preparation

The essential first step is obvious: learn the rules of the game! Consult the Wikipedia page on Baccarat, which has a very good summary. Note that you only need to read the sections headed 'Punto banco' and 'Tableau of drawing rules'.

We strongly advise spending some time actually playing the game, with real decks of cards. It would be a good idea to do this with other people, so that you can help each other understand the rather complex logic for drawing cards and scoring the game. Feel free to do this during lab sessions!

We have provided them as the Zip archive `cwk2files.zip`, in the Coursework 2 folder.

Next, study the UML diagram in Figure 1. This summarises the features of and relationships between the classes used in this assignment. The four unshaded classes must be implemented as specified here in order to complete the advanced solution.

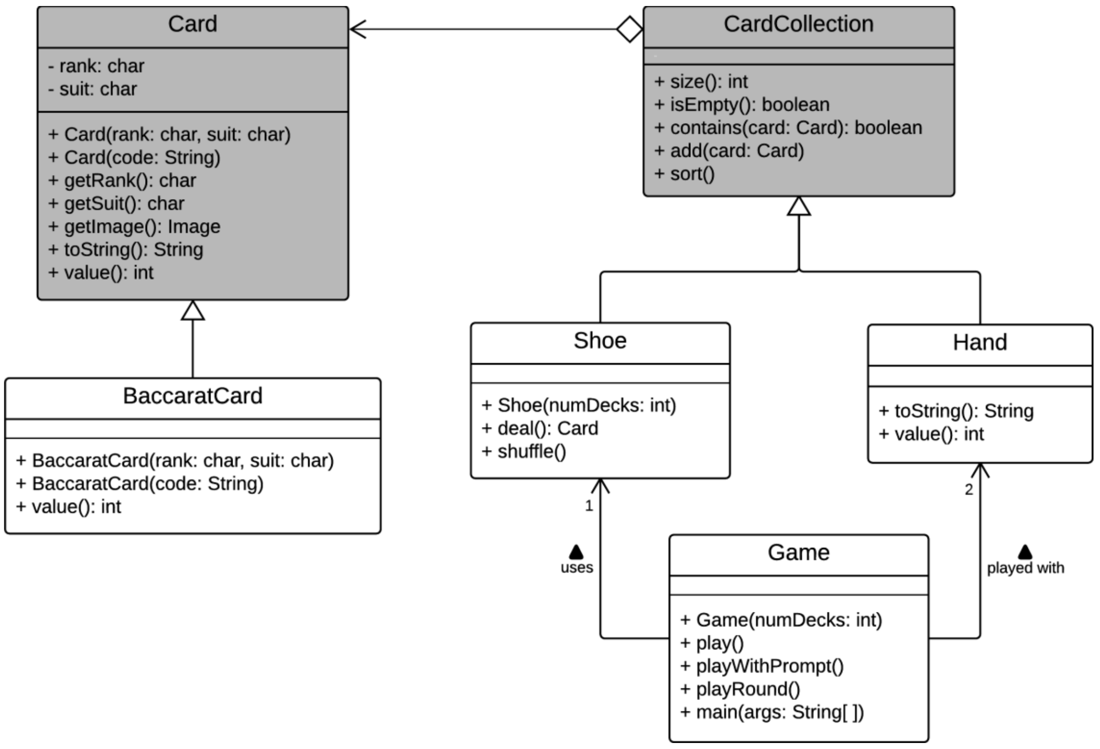

Figure 1: UML diagram for Baccarat simulation

3 Implementation

Please note the following requirements:

- Your implementation must be put in the `cwk2` directory of your repository.

3.1 Basic Solution

1. In a file called `BaccaratCard.java`, create a public class called `BaccaratCard`, inheriting from `Card`. Give this class the two constructors identified in Figure 1. These should simply delegate to the corresponding superclass constructors.
2. Override the `value` method inherited from `Card` so that it returns the points value of a card in Baccarat. (See the Wikipedia article for details of scoring.)
3. In a file called `Game.java`, create a public class called `Game`. Give this class a `main` method. Inside this method, write some code that creates three different instances of `BaccaratCard` and then tests that calling the `value` method returns the expected results.
4. In a file called `Hand.java`, create a public class called `Hand`, inheriting from `CardCollection`. In this class, override the default version of the `toString` method with a new version that returns a string containing two-character representations of each card, separated from each other by a space. For example, a hand containing the Ace of Clubs, Four of Diamonds and Jack of Spades should yield the following:

   ```
   AC 4D JS
   ```

   Note: `Card` defines a `toString` method that does some of this work for you already.
5. Add to `Hand` a method called `value` that returns the total points value of the cards in the hand, according to the rules of Baccarat. (See the Wikipedia article for details of scoring.)
6. Modify the program you implemented in `Game` so that it
   - Creates a `Hand` object
   - Creates two `BaccaratCard` objects with ranks and suits specified via user input
   - Adds those two cards to the hand, then displays the hand's contents and value
   - Applies the "Player's rule" described on the Wikipedia page for Baccarat, with the third card (if needed) being specified via user input
   - Displays the hand contents and value again if a third card was added

   Use the program to test that you are scoring Baccarat correctly.

3.2 Intermediate Solution

1. In a file called `Shoe.java`, create a public class `Shoe`, following the specification in Figure 1. Your `Shoe` class should inherit from `CardCollection` and should have a constructor that allows the number of decks to be specified as 4, 6 or 8. The constructor should ensure that cards from the given number of decks are stored in a `Shoe` object, using the inherited list for this purpose.

   Note: you will need to iterate over ranks and suits for this. The static methods `Card.getRanks` and `Card.getSuits` will give you lists that can be used for iteration.
2. Implement methods `deal` and `shuffle` in `Shoe`. The `deal` method should remove the 'top card' from the shoe and return it. The `shuffle` method should randomly shuffle the cards in the shoe. Use Java's `Collections` class to help you with the latter.
3. Modify `Game` so that it has fields to represent the shoe of cards and the two hands used in the game: one for the player, the other for the banker. Give the class the constructor specified in Figure 1. This constructor should initialise the fields and set things up ready for the start of the game.

4. Add to `Game` a method called `playRound`. This method should contain the logic from the `main` method of the basic solution, altered so that the cards are dealt from the shoe rather than being specified by the user. `main` should now create a `Game` object and call `playRound`.

3.3 Advanced Solution

Note: this involves a full simulation of the game, which is more challenging than the partial simulation implemented earlier.

1. Extend the implementation of `playRound` in `Game` so that it plays a realistic round of Baccarat, following the full Punto Banco rules.
2. Add a method called `play` that repeatedly calls `playRound` until the shoe is exhausted. Once the game finished, your `play` method should display summary statistics showing the number of rounds played, number of player wins, number of banker wins and number of ties.
3. Add a method called `playWithPrompt`. This should behave in a similar way to `play`, except that it should end each round by asking the user if they want to play another round, terminating the game if they respond negatively. Note that summary statistics should still be displayed if a game is terminated early.

   Below is an example of what interaction might look like. Your program should behave similarly but does not need to duplicate this format exactly.

   ```
   Round 1
   Player: 6D 3D = 9
   Banker: JS 6S = 6
   Player win!
   Another round? (y/n): y

   Round 2
   Player: 9S 4D = 3
   Banker: 7H 4H = 1
   Dealing third card to player...
   Dealing third card to banker...
   Player: 9S 4D 4C = 7
   Banker: 7H 4H AH = 2
   Player win!
   Another round? (y/n): y

   Round 3
   Player: KH 3D = 3
   Banker: 7C KC = 7
   Dealing third card to player...
   Player: KH 3D 8S = 1
   Banker: 7C KC = 7
   Banker win!
   Another round? (y/n): n

   3 rounds played
   2 player wins
   1 banker wins
   0 ties
   ```

Feel free to introduce additional methods into `Game`, beyond those mentioned above, if it will help to improve the structure and clarity of your solution.

3.4 Super-Advanced Solution

The final part of this assignment is significantly more challenging and will require a considerable investment of time on your part, for the reward of relatively few marks. It is aimed at people who complete the earlier parts fairly quickly and are looking for something to keep them busy over Easter! You should not attempt it if you struggled with the earlier parts.

1. Investigate JavaFX by reading Chapters 14–16 of Liang's book and trying out some of the example programs. You can also consult the official documentation on JavaFX, at

   ```
   http://docs.oracle.com/javase/8/javase-clienttechnologies.htm
   ```

2. Study the files `DisplayDeck.java` and `Animate.java` that were provided for this assignment. These give you further examples of how JavaFX can be used, in ways that may be relevant to this assignment. Try compiling and running them.
3. Use JavaFX to implement a graphical user interface for your Baccarat simulator. This can take any form you like. Be imaginative!

4 Submission

When your implementation is complete, generate a Zip archive of your work by moving into the top-level directory of your local repository and entering

```
zip -r cwk2.zip cwk2
```

Don't forget the `-r` option!

Note that an actual Zip archive is required; you will be penalised if you submit some other archive format (RAR, 7-Zip, tar, gzipped tar, etc).

The deadline for submissions is **10 am on Monday 20 May**. The standard university penalty of 5% of available marks per day will apply to late work, unless an extension has been arranged due to genuine extenuating circumstances.

Note that all submitted code will be scanned by plagiarism detection software.

5 Marking

40 marks are available for this assignment.

A basic solution can earn up to 60% of these marks. This rises to 80% for an intermediate solution. The advanced and super-advanced tasks are each worth an additional 10%.

A detailed breakdown of mark allocation is as follows:

4	`BaccaratCard` class defined properly, with constructors
2	`value` method of `BaccaratCard`
2	`toString` method of `Hand`
2	`Hand` class defined properly
2	`value` method of `Hand`
2	`Game` class defined properly, with a `main` method
3	`Shoe` class defined, with constructor
2	`deal` & `shuffle` methods of `Shoe`
4	Constructor and `playRound` method in `Game`
4	Full game simulation, `play`, `playWithPrompt`
4	JavaFX GUI implemented for super-advanced solution
3	Successful compilation & execution
3	Coding style
3	Comments
40	