Victoria University of Wellington
School of Engineering and Computer Science

SWEN221: Software Development

Assignment 5

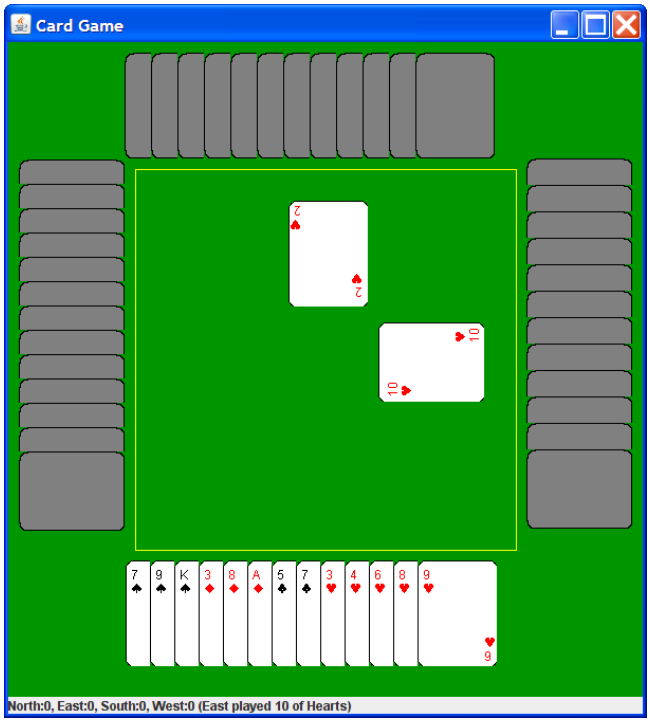
Due: Monday 11th May @ Mid-night

The Card Game

The `CardGame` system is a simple card game written in Java. The game implements some variations on the well-known *trick-taking card game*. For more on this style of game, see this:

`http://en.wikipedia.org/wiki/Trick-taking_game`

There are four players (North, East, South and West) and is dealt exactly 13 cards each (i.e. the whole deck). The game then proceeds in a series of *tricks*. In each trick the *leader* lays a card, and then the next player (following clockwise rotation) plays a card, and so on until four cards have been played. The following illustrates:



Here, we see that North and East have played and, hence, South is next to play. Since North lead with a Heart, and South has a Heart, then he/she must play one of the available hearts.

If a player has a card of the same suit as that played by the leader, *then he/she must follow suit*. The winner of a trick is the player who played the highest card *in the same suit as the leader*. The players sitting opposite to each other form a partnership. The winner of the game is that partnership who, after every card is played, has won the most tricks.

Every round either has a single suit of *trumps* or has *no trumps*. The sequence of trumps is *Hearts*, *Clubs*, *Diamonds*, *Spades*, *No Trumps* and this is repeated for the duration. The current suit of trumps (if applicable) is always the highest suit with respect to the ordering of cards.

Part 1: Card Comparisons (worth 20%)

The `CardGame` system is almost fully functioning! To start off, you should implement the methods `Card.equals()`, `Card.hashCode()` and `Card.compareTo`. The latter method should sort cards by their suit and rank, such that Hearts < Clubs < Diamonds < Spades. In other words, any card in hearts is always less than than any card in clubs, etc. However, the 6 of hearts is greater than the 2 of hearts. For picture cards we have that: Ace > King > Queen > Jack > 10.

There are several JUnit tests provided with the `CardGame` system for this part (`testCardEquals()`, `testCardNotEquals()` and `testCardCompareTo()`). You should ensure that all of these tests now pass correctly. You should also find that, having implemented the required classes and methods, you can now play the game by running the method `cards.Main.main()`, and choose all human players. If the `Card.compareTo()` method is implemented correctly, the hand of each player should be sorted by suit in increasing order, starting with hearts.

Part 2: Illegal Moves (20%)

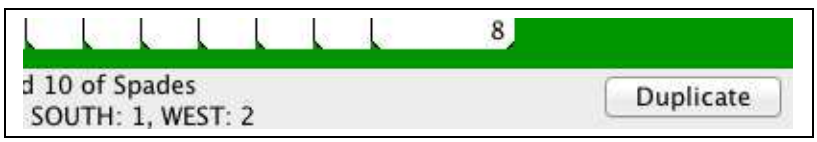
This part is concerned with the method `Trick.play(Player,Card)`. When a player plays a card, this method is called to update the current trick being played. Unfortunately, it is possible for players to play *invalid cards* (e.g. cards not present in their hand, or not following suit), or to try and play *out of sequence*. This happens when a human player does something out of sequence.

You should implement the method `Trick.play(Player,Card)` to ensure that any attempt to play an invalid card, or to play out of sequence results in an `IllegalMove` exception being raised.

There are several JUnit tests provided with the `CardGame` system for this part (`testInvalidPlay_1()`, ..., `testInvalidPlay_5()`. You should ensure that all of these tests now pass correctly. You should also find that, when playing the game, trying to play an invalid card does not work (and, instead, an error is reported on the status bar).

Part 3: Cloning (20%)

This part is concerned with the method `CardGame.clone()` and its implementation. The "duplicate" button in the Graphical User Interface employs this method to duplicate the current game:



Unfortunately, this method is not currently implemented correctly. In particular, this method is implemented in `AbstractCardGame` using a *shallow clone*. However, in order to properly duplicate

a game a *deep clone* is required. Therefore, you need to replace and/or override the method in `AbstractCardGame` with appropriate implementations in its subclass(es).

There are several JUnit tests provided with the `CardGame` system for this part (i.e. `testValidClone_1()`, ..., `testValidClone_5()`). You should ensure that all of these tests now pass correctly.

Part 4: Artificial Intelligence (30%)

This part is concerned with the class `SimpleComputerPlayer`, which is currently mostly unimplemented. This player chooses what card to play based on the following rules:

- If the AI player can *potentially win* the trick, then it plays the *highest eligible card.*
- If the AI player *cannot win* the trick, then it *discards* the lowest eligible card.
- In the special case that the AI player *must win* the trick, then it conservatively plays the *least card* needed to win.

An important concept for understanding these rules is the *ordering of eligible cards.* A card is eligible if it may be played according to the rules of the game (e.g. if the AI player can follow suit then it must, etc). The highest eligible card is then the card with the *highest rank and suit*, where the current suit of trumps (if applicable) is always the highest suit. In the case of two equally ranked cards of non-trump suit, then the underlying ordering implied by `Card.compareTo()` (as discussed above) should be used to chose.

There are several JUnit tests provided with the `CardGame` system for this part (`testSimpleAI_1()`, ..., `testSimpleAI_19()`). You should ensure that all of these tests now pass correctly. You should also find that you now be able to play the game against computer players.

Submission

Your program code should be submitted electronically via the *online submission system*, linked from the course homepage. You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. See the export-to-jar tutorial linked from the course homepage for more on how to do this. *Note, the jar file does not need to be executable.*
2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code.*
3. **All JUnit test files supplied for the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. This does not prohibit you from adding new tests, as you can still create additional JUnit test files. *This is to ensure the automatic marking script can test your code.*
4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in you getting zero marks for the assignment.

Assessment

This assignment will be marked as a letter grade (A+ ... E), based primarily on the following criteria:

- **Correctness of Part 1 (20%)** — does submission adhere to specification given for Part 1.
- **Correctness of Part 2 (20%)** — does submission adhere to specification given for Part 2.
- **Correctness of Part 3 (20%)** — does submission adhere to specification given for Part 3.
- **Correctness of Part 4 (30%)** — does submission adhere to specification given for Part 4.
- **Style (10%)** — does the submitted code follow the style guide and have appropriate comments (inc. Javadoc)

As indicated above, part of the assessment for the coding assignments in SWEN221 involves a qualitative mark for style, given by a tutor. Whilst this is worth only a small percentage of your final grade, it is worth considering that good programmers have good style.

The qualitative marks for style are given for the following points:

- **Division of Concepts into Classes**. This refers to how *coherent* your classes are. That is, whether a given class is responsible for single specific task (coherent), or for many unrelated tasks (incoherent). In particular, big classes with lots of functionality should be avoided.
- **Division of Work into Methods**. This refers to how well a given task is split across methods. That is, whether a given task is broken down into many small methods (good) or implemented as one large method (bad). The approach of dividing a task into multiple small methods is commonly referred to as *divide-and-conquer*.
- **Use of Naming**. This refers to the choice of names for the classes, fields, methods and variables in your program. Firstly, naming should be consistent and follow the recommended Java Coding Standards (see `http://g.oswego.edu/dl/html/javaCodingStd.html`). Secondly, names of items should be descriptive and reflect their purpose in the program.
- **JavaDoc Comments**. This refers to the use of JavaDoc comments on classes, fields and methods. We certainly expect all `public` and `protected` items to be properly documented. For example, when documenting a method, an appropriate description should be given, as well as for its parameters and return value. Good style also dictates that `private` items are documented as well.
- **Other Comments**. This refers to the use of commenting within a given method. Generally speaking, comments should be used to explain what is happening, rather than simply repeating what is evident from the source code.
- **Overall Consistency**. This refers to the consistent use of indentation and other conventions. Generally speaking, code must be properly indented and make consistent use of conventions for e.g. curly braces.

Finally, in addition to a mark, you should expect some written feedback highlighting the good and bad points of your solution.