Victoria University of Wellington
School of Engineering and Computer Science

SWEN221: Software Development

Assignment 3

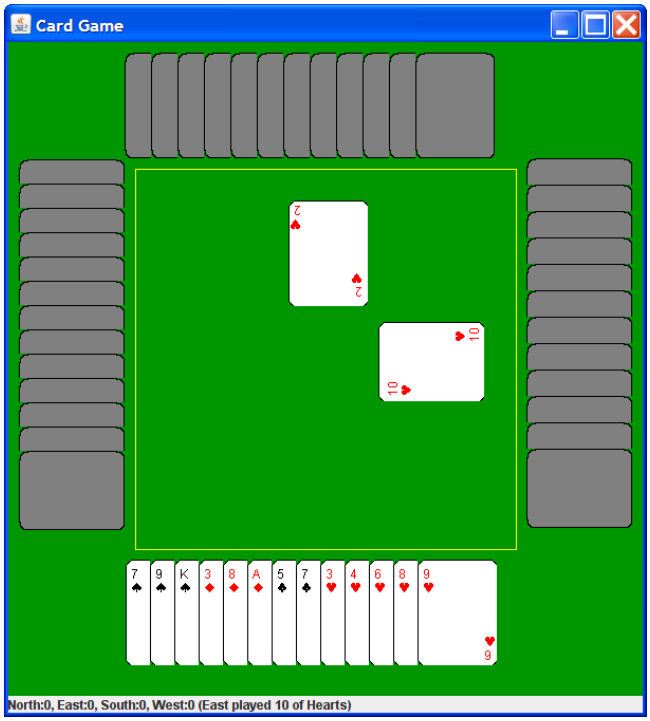
Due: 23:59pm Sunday 14th May

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, i.e. play a card in the same suit as the leader*. Otherwise, he/she can play any card.

Each trick 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 is always the highest suit with respect to the ordering of cards, but a trumps suit card can only be played if the player does not have a card that has the same suit as the card played by the leader.

The winner of a trick is determined by comparing cards as follows.

1. At first, the *leader* is the current winner;
2. If a player plays a card in the same suit as the current winner and has a higher rank, then the player becomes the current winner;
3. If a player plays a card in the current suit of trumps (if there is), and the current winner's card is not in the current suit of trumps, then the player becomes the current winner.

The winner of a trick will become the leader of the next trick. The winner of the game is the player who, after every card is played, has won the most tricks.

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 `Card.compareTo()` 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. For the cards in the same suite, the card with a higher rank is greater than the card with a lower rank. For example, 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%)

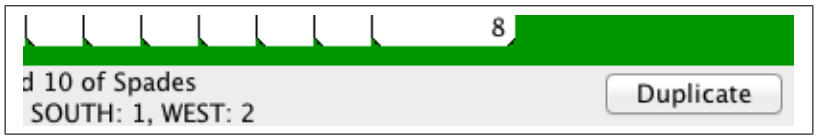
This part is concerned with the method `Trick.play(Player,Card)`. When a player plays a card, this method is called to update the current trick being played. Unfortunately, it is possible for players to play *invalid cards* (e.g. cards not present in their hand, or not following suit), or to try and play *out of sequence* (e.g. South tries to play before East). 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 `clone()` method in the subclass(es) of `AbstractCardGame` with appropriate implementations.

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 (for example, when it is the last player of the trick, and already knows the cards played by all the other players), then it conservatively plays the *lowest eligible card needed to win*.

An important concept for understanding these rules is the *ordering of eligible cards*.

First, the *eligibility* of a card is determined by the rules of the game. Specifically,

- If the AI player is the *leader* of a trick, then *all the cards* at hand are eligible.
- If the AI player is not the leader, and it has cards with the same suit as the card played by the leader, then *only the cards with the same suit as the card played by the leader* are eligible.
- If the AI player is not the leader, and it has *no* card with the same suit as the card played by the leader, then all *the cards* at hand are eligible.

After getting all the eligible cards, the *order of eligible cards* is determined as follows.

- If card A has the *current suit of trumps*, and card B has not, then A is higher than B.
- If both cards A and B have the current suit of trumps, and A has a *higher rank* than B, then A is higher than B.
- If both cards A and B do not have the current suit of trumps, and A has a *higher rank* than B, then A is higher than B.
- If both cards A and B do not have the current suit of trumps, A has the *same rank* as B, and A has a *higher suit* than B, then A is higher than B.

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 source files should be submitted electronically via the *online submission system*, linked from the course homepage. The minimal set of required files is:

```
swen221/cardgame/cards/Main.java
swen221/cardgame/cards/core/Card.java
swen221/cardgame/cards/core/CardGame.java
swen221/cardgame/cards/core/Hand.java
swen221/cardgame/cards/core/IllegalMove.java
swen221/cardgame/cards/core/Player.java
swen221/cardgame/cards/core/Trick.java
swen221/cardgame/cards/tests/Part1.java
swen221/cardgame/cards/tests/Part2.java
swen221/cardgame/cards/tests/Part3.java
swen221/cardgame/cards/tests/Part4.java
swen221/cardgame/cards/util/AbstractCardGame.java
swen221/cardgame/cards/util/AbstractComputerPlayer.java
swen221/cardgame/cards/util/SimpleComputerPlayer.java
swen221/cardgame/cards/variations/ClassicWhist.java
swen221/cardgame/cards/variations/KnockOutWhist.java
swen221/cardgame/cards/variations/SingleHandWhist.java
```

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**. *Note, the jar file does not need to be executable.* See the following Eclipse tutorials for more on this:

   `http://ecs.victoria.ac.nz/Support/TechNoteEclipseTutorials`

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 testing mechanism supplied with the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. However, this does not prohibit you from adding new tests. *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 your submission being reject by the submission system and/or zero marks being awarded.

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 SWEN 221 involves a qualitative mark for coding style. Coding style is an important aspect of coding since all code should be expected to be revisited again, i.e., it is important that you are someone else can make sense of the code it in the future and change it without accidentally introducing errors.

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

- **Division of Functionality into Classes**. This refers to how *cohesive* your classes are. That is, whether a given class is responsible for single specific task (has high cohesion), or for many unrelated tasks (has low cohesion). In particular, big classes with lots of weakly related 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 reusable methods (good) or implemented as one large monolithic method (bad).
- **Self-Explanatory Naming**. This refers to the choice of names for the classes, fields, methods and variables in your program. First, 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 code.
- **JavaDoc Comments**. This refers to the use of JavaDoc comments on classes, fields and methods. We expect all `public` and `protected` items to be properly documented. For example, when documenting a method, an appropriate description should be given, including despriptions for (if present) its parameters and return value. Good style dictates that `private` features are documented as well.
- **Code Comments**. This refers to the use of commenting within a given method. Comments should be used to elaborate the purpose of the code, rather than simply repeating what is evident from the code already.
- **Layout**. This refers to the consistent use of indentation and other layout conventions. Code must be properly indented and make consistent use of conventions e.g. for placing curly braces.

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