Department of Computing

Assignment 1 Specification

Due date: Monday 28 September, 5 pm

Assignment marks: 20% of overall unit marks.

Objective: To apply your knowledge of JAVA to building a simple application. Practice your skills of class design, and applying the given Design Patterns.

Please note: This assignment specification aims to provide as complete a description of this assessment task as possible. However, as with any specification, there will always be things we should have said that we have left out and areas in which we could have done a better job of explanation. As a result, you are strongly encouraged to ask any questions of clarification you might have, either by raising them during a lecture or by posting them on the iLearn discussion forum devoted to this assignment.

A Simple Game: Nim

Your task in this assignment will be to write a simple version of a version of the game of *Three Pile Nim* described below. Nim is a two player game in which contestants take turns to remove pebbles or counters from one of **three piles** according to the rules set out below. The aim of each player is to force the opponent to take the last pebble. The player which achieves this is declared the winner.

Of course, in order to do this assignment the first thing you will need to do is familiarise yourself with Nim. The game itself is very simple, but describing it in words won't quite give you the full picture. So by far the best way to get the idea is to try playing a few rounds of (a version of) the game, which you can do online. Below is a suggested website to try. Do note however that there are many versions of Nim, with varying rules and set up --- **the set of rules which you must implement for your assignment are listed below under the heading Rules of Nim**.

1. Nim. At this site you can play a number of variations of Nim.

To find out more about the history of the game read and follow the suggested links at Nim section on Wikipedia.

For this assignment, we'll be using the following formal rules of Nim:

Rules of Nim

The game of Nim is played using piles of matches or pebbles. Initially you may assume that there are three piles, each with 5 pebbles, but read carefully the suggested extensions for obtaining a D/HD. Each player takes turns to remove pebbles from exactly one of the piles.

Object of the game

The object of the game is to force the opponent to remove the last pebble. (Note: there are other conventions, but this is the one we will use for this assignment.) Note that one player assumes the colour BLACK and the other one assumes the colour WHITE. Players take it in turn as follows.

Rules of Nim

1. WHITE always plays first.
2. When it's a player's turn he chooses a pile from which to remove pebbles;
3. After choosing a pile he selects a number of pebbles to remove from that pile and then he removes them.
4. Note that the removed pebbles do not take any further part in the game.
5. A player must remove at least one pebble when it is his turn, and he is not allowed to remove more than 3 pebbles in any single turn.
6. Play alternates until there are no more pebbles, with the player who removes the last piece being declared the loser.

Your task

Your task in this assignment will be to implement a very simple Nim game in Java. Your game will be two players, in the sense that two individuals sit at the same terminal and take turns to make moves. In particular to obtain a PC, P or CR, you will not be required to implement the artificial intelligence required of a game in which a human player plays against the computer (but see the requirements for obtaining an HD).

In brief, depending on which level of credit you would like to aim for, your implementation should provide:

- A *game logic* class which manages the state of the game from one move to the next and ensures that the rules of the game are adhered to.
- A win detection facility, as part of that game logic class, which detects when a winning position is reached (and determines which of the players has won).
- A undo / redo facility, allowing players to step backwards and forwards through the moves they've made.
- A more sophisticated graphical user interface by which more interactively minded players can play the game. (Note that if you provide a GUI, then there is no need to provide a text-based user interface.)
- A computer player which implements some AI so that a player can play in "single player mode" against the computer.
- The possibility of setting the number of piles arbitrarily at the beginning.

To give you an idea of what we expect, we've written a sample implementation of the kind of thing we are looking for. You can download (the executable version of) this sample solution from the COMP229 iLearn site. It is contained in a "jar" file called SinglePileNim.jar. To run this application go to the place that you saved the jar file and double click on it (i.e. the file called Nim.jar). Alternatively, you can open your command line utility, change directory to the place that you saved the jar file and type:

```
        java -jar SinglePileNim.jar
```

This is a very simple 1-pile version. Each player can choose to remove up to three images. For example, if the player wants to delete 2 images, he can right click on the second image from the bottom. It also has two buttons which allow moves to be undone or redone. Just for fun, Barbie's face disappears when clicked.

Note that if you choose to run the jar file from the command line you should also see an alternative text display (as well as the GUI). The text display corresponds exactly to the GUI's display, but also includes additional reports (e.g. if the player tries to remove too many pebbles).

The text based user interface displays the game state as a simple column of asterisks (instead of matches) like this:

```
The state of play is

           _
           *
           *
           *
           *
           *
           _
```

Also, please do follow the lectures where we will be discussing how to design games in the OO way, using design patterns to structure the GUI and the Java classes which deal with the game logic. These discussions and related code snippets can be particularly helpful to you when approaching your own design.

More about our implementation

As a hint, here is a little more information about the internals of our implementation above.

Firstly, at the heart of this implementation lies a class called `GameState`. It encapsulates all of the *state* information required to keep track of a game as it progresses, including:

- The configuration of the game board as pebbles are placed;
- A variable to keep track of how many total moves have been made;
- A list of moves taken, through which it manages the processes of undo-ing and redo-ing.

As well as the state, there are also a number of methods by which game moves may be made or state inspected, and these impose the rules of Nim by disallowing incorrect moves. Most notably, these methods include:

- `reset()` initialises the board to remove all pebbles, resets the game history and zeroes the move counter.
- `gameIsOver()` returns a boolean to flag when the game has ended.
- `make()` makes a move by updating the game state once the current player has decided how he wants to make the move. Notice how you can use the ideas of encapsulating the data with the methods that can update the data to structure the program so that the

data required to make the move is not required as input to the move (although naturally that information must be preserved somewhere)

- `undo()` steps back one move in the game history, updating the board to reflect the fact that the last move has been rescinded.
- `redo()` steps forward one move in the game history, updating the board to reinstate the last move rescinded by a call to `undo()`.
- `print()` prints the board; this is used in the non-GUI version of the game.

A considerable amount of coding went into interpreting the input and handling the errors that might arise from that. In our implementation we used an extension of Java's exception class to do the error analysis for us. We were also able to tidy up the design by a careful nesting of classes to avoid repeating references.

One aspect of our implementation for the GUI version was our use of the *observer pattern* to ensure that the displayed board was redrawn whenever the game state changed. Of course, you needn't take this approach in your code if you're not going for a D or HD, but if you would like to you can find out lots of information about this programming pattern by consulting its Wikipedia entry Observer_pattern or by having a look at the following freely available online book on Java design patterns The Design Patterns Java Companion by James W. Cooper. It's also covered in Head First Design Patterns.

Please note that we will be covering the basic observer pattern in lectures in case you choose this method to design your program.

What you have to do

This assignment is structured to allow you to decide how much effort you want to expend for the return in marks that you might hope for. You can choose which bits of the functionality of the full application you feel confident about implementing and then only write those parts for a proportion of the maximum possible marks. So you can decide upfront whether you are shooting for a pass or a high distinction and know exactly how much work will be required to obtain that mark.

Here is what is required to obtain marks in one of the performance bands for this assignment:

- **Pass** (P, 50%-65%) an implementation of the basic game logic class (not including undo / redo), but including the determination of the winner, along with a simple GUI interface as described above to display the game state. You must at the very least have thought about how the Model View Controller Design Pattern discussed in lectures applies to your design, and have separate classes for the View/Controller combined with the Game logic class; you must provide a brief description (as comments) of how they are used in your design (you do not need to implement this design). You will also need to produce a set of JUnit tests to test the functionality of the Game logic class .
- **Credit** (Cr, 65%-75%) the P level implementation + the undo and redo facility + saving of game states to files / reloading of game states from files.
- **Distinction** (D, 75%-85%) the Cr level implementation + some additional functionality, such as a single-player mode which incorporates the AI for a computer player to play against a single player, or a mode where the number of piles may be set at the beginning of the game.

- **High Distinction** (HD, 85%-100%) the D level implementation + using the Observer Pattern to implement your MVC design + other cool stuff you can think of.

Notice that under this scheme, the application we've provided only really lies somewhere between Pass and Credit (CR) bracket, since it doesn't allow users to save / restore game states. (It does however use the Observer Pattern!) Remember that high distinctions (HD) are generally rewarded for work that shows particular insight, so to gain an HD it is necessary to do a particularly good job with the design and implementation of your application and to extend this specification in the ways suggested.

One thing you **must not** do is to implement your entire project as a single Java class --- your application should make use of the structuring features of Java to break the problem into classes. As a rule of thumb, remember if you notice that certain methods tend to use the same pieces of data again and again, then you should probably package up the methods with the data.

For example, our sample application consists of several classes, called things like `GameState`, `Move`, `EasyPuzzleGUI`, `GameStateObservable` and so forth.

Expected effort.

In general, a 3cp unit of study at Macquarie is rated as about 150 hours of work (including lectures attendance, exam revision, assignment and tutorial work...) scattered throughout the semester (17 weeks). This assignment is worth 20% of the overall unit marks and so it is probably worth expending about 20% of your total COMP229 study time on. That means that you should expect to spend around a total of 30 hours on writing your code.

You are strongly advised not to attempt to solve all of this assignment in one go. Rather, you should solve each part and get it working before you go on to the next part. It is also important to start working on this assignment *right now*, even if you only devote an hour or three over the first 10-14 days to thinking carefully about what this task might entail and marshalling the resources you will need to complete it.

What you must hand in

1. All of the Java source files you have written bundled together in a ZIP file. Our preference is for zipped archives of Eclipse projects. If you are not using Eclipse then you should make sure that your code comes with a `README.txt` file which *clearly* describes how to compile and run your application using the Java command line tools.
2. Correctly document the program with JavaDoc. Use the JavaDoc syntax and conventions which are dominant in the Java programming community (and we taught you) to document the program.
3. Provide adequate documentation comments. Briefly describe each class you have written, explaining the functionality of the most important methods of that class. Include only enough detail to allow a reader to understand the structure of your code base and how the various components in your code collaborate together to achieve the various functionalities of your application.

   For example, you should explain the role of each individual class in your overall design, but for each class you should provide brief specifications of your methods. A description of how you handled possible errors, and what tests you carried out on your classes and methods. You must also describe how you used the MVC design pattern, and identify the model and the view/controller.

4. Provide adequate comments for testing. Note that just testing one or two simple aspects of this class is not enough for full marks. You should try to demonstrate that you have used a properly representative set of test cases to be confident that you have covered all the bases. You could simply describe what you did to check that your classes work properly.

The ZIP archive of your source code should be submitted via the Assignment 1 link on the COMP229 iLearn site.

Mark allocation

The marks for this assignment will be allocated as follows:

- 55% for implementation quality and functionality.
- 45% in style points, for aspects like good in-code documentation, nice design, interesting functionality etc.

Copyright & Site information