The University of Melbourne
School of Computing and Information Systems
COMP90041 Programming and Software Development
Lecturer: Prof. Rui Zhang
Semester 1, 2019

Project B
Due: 3pm, Monday 6 May, 2019

1 Introduction

This project (Project B), and the next one (Project C), will continue with the same theme introduced in Project A - namely, an implementation of the game of Nim. Please refer to the Project A specification for a description of the game and its rules.

In Project A, a simple Java program was created to handle Nim's core game play mechanics. In the next two projects, the objective is to design and implement a more complete version of Nim, making full use of Java's object-oriented paradigm.

Key Knowledge Points Covered in Project B:

1. Design of the class structure for the project requires the knowledge of UML diagrams (taught in Week 5).
2. Implementation requires understanding of *Classes* (taught in Week 4 - 6) and *Arrays* (taught in Week 7).

You may start working on the project right away except the array part. If you would like to start to work on the array part before week 7, you may learn basics of arrays by yourself.

2 Requirements

In this project, we introduce a third class NimGame. The game playing process is delegated from Nimsys to NimGame. Since only one game will be active at any given time, only a single NimGame instance is required at any time by Nimsys. Nimsys should also maintain a collection of players. Initially, this collection will be empty - players will need to be added in order to play a game.

A NimGame instance needs to have the following information associated with it:

- The current stone count
- An upper bound on stone removal

- Two players

The system should allow for games of Nim to be played, with the rules of the game, and the players, specified by the user.

A player, as described by the NimPlayer class, needs to have the following information associated with it:

- A username
- A given name
- A family name
- Number of games played
- Number of games won

The system should allow players to be added. It should also allow for players to be deleted, or for their details to be edited. Players should not be able to directly edit their game statistics, although they should be able to reset them.

The system is a text based interactive program that reads and executes commands from standard input (the keyboard) until an 'exit' command is issued, which will terminate the program. If a command produces output, it should be printed to standard output (the terminal).

When Nimsys is first executed, it should display a welcome message, **followed by a blank line**. A command prompt (a 'dollar' sign, i.e., `$`) should then be displayed.

In following description, all command line displays are put in a box. This is only for easier understanding the format. **The box should NOT be printed out by your program, only the contents in the box should be printed.** The command prompt is illustrated below:

```
Welcome to Nim

$
```

At any given time, the system can be in one of two states - either a game is in progress, or no game is in progress. Hereafter, these will be referred to as the 'game' and 'idle' states, respectively. (Note: the states are just used to explain the mechanism of Nimsys. You don't need to create a variable called 'state' in your code).

When in the idle state, the system should accept the following commands. These commands are entered at the Nimsys command prompt. If a command produces output, it should be printed immediately below the line where the command was issued. After the command has executed, a new command prompt should be displayed. This new command prompt should be separated from the previous command (and its output, if any) by a single blank line.

Note that in the syntax descriptions below, a term enclosed in square brackets indicates an optional parameter. The input is assumed to be always *valid*, but not always *correct*. Valid input suggests that entered data have the same type of the corresponding variables, e.g., `String` data are entered for `String` variables, `integer` data are entered for `int` variables. Correct input suggests that the entered data can be correctly processed by the corresponding command, e.g., adding an existing user and removing a nonexistent user are incorrect input. **Unless otherwise stated, you are NOT required to check validness, but you ARE required to check the correctness of the input, as shown in the below examples.**

1. addplayer - Allows new players to be added to the game. If a player with the given **username** already exists, the system should indicate this, as shown in the example execution.

   Syntax: `addplayer username,family_name,given_name`

   Example Execution:

   (a) add a new user:

   ```
   $addplayer lskywalker,Skywalker,Luke

   $
   ```

   (b) add a user who already exists in the system

   ```
   $addplayer lskywalker,Skywalker,Luke
   The player already exists.

   $
   ```

2. removeplayer - Allows players to be removed from the game. The username of the player to be removed is given as an argument to the command. If no username is given, the command should remove all players, but in this case, it should display a confirmation question first. If a username for a non-existent player is given, the system should indicate that the player does not exist. The format of these messages is illustrated in the example execution below.

   Syntax: `removeplayer [username]`

   Example Execution:

   (a) remove a nonexistent user

   ```
   $removeplayer lskyrunner
   The player does not exist.

   $
   ```

   (b) remove a user

   ```
   $removeplayer lskywalker

   $
   ```

   (c) remove all users

   ```
   $removeplayer
   Are you sure you want to remove all players? (y/n)
   y

   $
   ```

3. editplayer - Allows player details to be edited. Note that the player's username cannot be changed after the player is created. If a username for a non-existent player is given, the system should indicate that the player does not exist, as illustrated in the example execution.

   Syntax: `editplayer username,new_family_name,new_given_name`

   Example Execution:

   (a) edit a nonexistent user

   ```
   $editplayer lskyrunner,Skywalker,Laurence
   The player does not exist.

   $
   ```

(b) edit a user

```
$editplayer lskywalker,Skywalker,Laurence

$
```

4. resetstats - Allows player statistics to be reset. The username of the player whose statistics are to be reset is given as an argument to the command. If no username is given, the command should reset all player statistics, but as with the 'removeplayer' command, a confirmation question should be displayed in this case. If a username for a non-existent player is given, the system should indicate that the player does not exist, as illustrated in the example execution.

   Syntax: `resetstats [username]`

   Example Execution:

   (a) reset a nonexistent user

   ```
   $resetstats lskyrunner
   The player does not exist.

   $
   ```

   (b) reset a user

   ```
   $resetstats lskywalker

   $
   ```

   (c) reset all users

   ```
   $resetstats
   Are you sure you want to reset all player statistics? (y/n)
   y

   $
   ```

5. displayplayer - Displays player information. The username of the player whose information is to be displayed is given as an argument to the command. If no username is given, the command should display information for all players, ordered by username alphabetically. If a username for a non-existent player is given, the system should indicate that the player does not exist, as illustrated in the example execution. **Please note when displaying player, the sequence of syntax is** `username,givenname,familyname,number of games played,number of games won`.

   Syntax: `displayplayer [username]`

   Example Execution:

   (a) display a nonexistent user

   ```
   $displayplayer lskyrunner
   The player does not exist.

   $
   ```

   (b) display a user

   ```
   $displayplayer lskywalker
   lskywalker,Luke,Skywalker,3 games,3 wins

   $
   ```

(c) display all users

```
$displayplayer
dvader,Darth,Vader,7 games,1 wins
hsolo,Han,Solo,4 games,3 wins
lskywalker,Luke,Skywalker,3 games,3 wins

$
```

6. rankings - Outputs a list of player rankings. There are three columns displayed. The first column displays percentage wins or winning ratio, the second column displays the number of games played, and the final column shows the player's full name, that is, first name followed by last name. This command takes the sort order as an argument. The sort order is `desc` or descending by default. That is, if no argument or `desc` is provided, the program should rank the players by the percentage of games they have won in descending order, i.e., players with highest percentage wins should be displayed first. If the user provides `asc` as an argument, the players should be ranked by the percentage of games they have won in ascending order. Round the percentages to the nearest integer value. However, you should use the exact values of winning ratios when comparing and sorting two users' winning ratios. Ties should be resolved by sorting on usernames alphabetically. Only the first 10 players should be displayed, if there are more than 10. The output should be formatted according to the example below. For the purposes of formatting the output, you may assume that no player has played more than 99 games. Note that the vertical lines need to be aligned, with a single space appearing on either side. This means that in the first column you **must** have 5 characters consisting of a number, '%', and spaces. The first column must be left-justified.

   Syntax: `rankings [asc|desc]`

   Example Execution:

   (a) rank all users in descending order

```
$rankings
100% | 03 games | Luke Skywalker
75%  | 04 games | Han Solo
14%  | 07 games | Darth Vader

$
```

   (b) rank all users in descending order

```
$rankings desc
100% | 03 games | Luke Skywalker
75%  | 04 games | Han Solo
14%  | 07 games | Darth Vader

$
```

   (c) rank all users in ascending order

```
$rankings asc
14%  | 07 games | Darth Vader
75%  | 04 games | Han Solo
100% | 03 games | Luke Skywalker

$
```

7. startgame - Creates and commences a game of Nim. The game's rules, and the usernames of the two players, are provided as arguments. You may assume that the initial stones and upperbound arguments are valid **and correct**. However, if at least one (i.e. one or two) of the

usernames doesn't correspond to an actual player, the system should indicate this by the output "`One of the players does not exist.`", and the game should not commence.

Otherwise, the 'startgame' command will commence a game, i.e., after executing it, the system is in the game state. When a game is in progress, the system should proceed according to the game play mechanics discussed in Project A, i.e., players should, in an alternating fashion, be asked to enter the number of stones they would like to remove, with the game state being updated accordingly. In this project, bounds on stone removal should be enforced. That is, players should only be allowed to remove between 1 and N stones inclusive, where N is the upper bound or the number of stones remaining, whichever is smaller. Once all the stones are gone, a winner should be announced, and the statistics for the two players should be updated accordingly. The system should then return to the idle state, and a command prompt should be displayed again.

Syntax: `startgame initialstones,upperbound,username1,username2`

Example Execution:

(a) start game with a non-existent user

```
$startgame 10,3,lskyrunner,hsolo
One of the players does not exist.

$
```

(b) start a game

```
$startgame 10,3,lskywalker,hsolo

Initial stone count: 10
Maximum stone removal: 3
Player 1: Luke Skywalker
Player 2: Han Solo

10 stones left: * * * * * * * * * *
Luke's turn - remove how many?
3

7 stones left: * * * * * * *
Han's turn - remove how many?
4

Invalid move. You must remove between 1 and 3 stones.

7 stones left: * * * * * * *
Han's turn - remove how many?
3

4 stones left: * * * *
Luke's turn - remove how many?
3

1 stones left: *
Han's turn - remove how many?
0

Invalid move. You must remove between 1 and 1 stones.

1 stones left: *
Han's turn - remove how many?
1

Game Over
Luke Skywalker wins!

$
```

8. exit - Exits the Nimsys program.

   Syntax: `exit`

   Note that before you call the exit method in Java using `System.exit(0);` , you must print a blank line first.

   (a) exit the system

```
$exit
```

As was described earlier, it is important that your design makes good use of object-oriented design principles. This is particularly relevant when it comes to implementing the actual gameplay. **In a real game, game proceeds with each player performing the action of removing some number of stones from the game. Your design should reflect this structure.**

Don't worry about changing your output for singular/plural entities. Simply always use plural entities, i.e., you should output '1 games', '1 wins', '1 stones', etc.

Checklist For Solution

- **Error handling issues**
  Only the following errors need to be handled (you may choose to handle more if you wish):
  - ☐ Adding a new player with an existing username.
    Error message: The player already exists.
    Follow-up operation: Print '$' and prompt for user input again.
  - ☐ Removing a player with a non-existing username.
    Error message: The player does not exist.
    Follow-up operation: Print '$' and prompt for user input again.
  - ☐ Resetting the statistics of a player with a non-existing username.
    Error message: The player does not exist.
    Follow-up operation: Print '$' and prompt for user input again.
  - ☐ Displaying a player with a non-existing username.
    Error message: The player does not exist.
    Follow-up operation: Print '$' and prompt for user input again.
  - ☐ Starting a game where at least one of the player's username does not exist.
    Error message: One of the players does not exist.
    Follow-up operation: Print '$' and prompt for user input again.
  - ☐ Removing stone outside of the range [1, `stoneRemovalUpperBound`], where `stoneRemovalUpperBound` is the maximum number of stones allowed to be removed in one turn.
    Error message: Invalid move. You must remove between 1 and `stoneRemovalUpperBound` stones (do not need to consider the singular or plural form of "stone").
    Follow-up operation: Prompt for the same player to remove stone again.
- **Blank line and whitespace related issues.**
  - ☐ Make sure that between the output of last command (including indication for nonexistent users, confirmation for all-user operations, display results, and game results) and the next command prompt, there is one blank line.
  - ☐ Make sure that there is a whitespace between `(y/n)` and `Are you sure...` sentence.
  - ☐ Make sure that there is NO whitespace after each comma, e.g., when displaying users, it should be `davader,Darth` instead of `davader, Darth`.
  - ☐ Make sure that in game state, there is one blank line before the `Initial stone count...` and one blank line after the `Player 2:...`.
  - ☐ Make sure that in game state, if a user input an invalid move, there is one blank line between the user-input move and the indication sentence `Invalid move...`.
  - ☐ Make sure that an extra blank line is printed out before calling `System.exit(0);` when command `exit` is entered.
- **Ranking display related issues.**
  - ☐ Make sure that the ranking is in a descending order of winning ratio by default or when "desc" is provided as an argument.

- ☐ Make sure that the ranking is in a ascending order of winning ratio when "asc" is provided as an argument.
- ☐ Make sure that winning ratio is rounded to the nearest integer value when displaying the winning ratio. However, please note that exact values are still used when comparing and sorting two users winning ratios.
- ☐ Make sure that users with the same winning ratio are sorted according to the alphabetical order of the user name irrespective of the argument supplied to the rankings command, e.g., if username ethan and username tom have the same winning ratio, you should rank ethan the higher place.
- ☐ Make sure that the first and the second column strictly have 5 and 10 characters, respectively.

- **Display player related issues.**
  - ☐ Make sure that the displayed list is sorted in an alphabetical order of the username.
- **Game related issues.**
  - ☐ Make sure to correctly update the statistics for players when a game ends.
  - ☐ Make sure to check valid move by comparing the move to the smaller one between the current number of stones and the upper bound for one move.
  - ☐ Make sure that after an invalid move, it is still the turn of the player who made the invalid move.
- **Command line prompt related issues.**
  - ☐ Make sure that the command prompt appears again after each command is issued (except the `exit` command).
  - ☐ Make sure that only one command prompt is displayed after a game is over and the system returns to the idle state.
- **Other issues.**
  - ☐ The maximum number of players can be set as 100.
  - ☐ The branching statement `switch` may not support `Strings` on the submission server.
  - ☐ The boxes enclosing the above example executions are NOT to be printed out, they are just for better illustration.

Important Notes About Submission

Immediately after you make a submission using the "submit" command, computer automatic test will be conducted on your program by automatically compiling, running, and comparing your outputs for several test cases with generated expected outputs. The automatic test will deem your output wrong if your output does not match the expected output, even if the difference is just having an extra space or missing a comma. Therefore it is crucial that **your output follows exactly the same format shown in the examples above.**

The keyword `import` is available for you to use standard java packages. However, please **DO NOT** use the `package` keyword to organise your source files into packages. The automatic test system cannot deal with packages. If you are using Netbeans as the IDE, please be aware that the project name may automatically be used as the package name. Please remove the line like

```
package ProjB;
```

at the beginning of the source files before you submit them to the system.

Please use **ONLY ONE** Scanner object throughout your program. Otherwise the automatic test will cause your program to generate exceptions and terminate. The reason is that in the automatic test, multiple lines of test inputs are sent all together to the program. As the program receives the inputs, it will pass them all to the currently active Scanner object, leaving the rest Scanner objects nothing to read and hence cause run-time exception. Therefore it is crucial that **your program has only one Scanner object.** Arguments such as "It runs correctly when I do manual test, but fails under automatic test" will not be accepted.

The test cases used to mark your submissions will be **different** from the sample tests given. You should test your program extensively to ensure it is correct for other input values with the same format as the sample tests.

3 Your Tasks

1. Draw a UML class diagram for Nimsys based on the above information. For each class you need to identify all its instance variables and methods (including public and private modifiers) along with their corresponding data types and list of parameters. You should also identify relationships between classes. You only need to identify the relationships taught in the lecture (i.e., association), if you include other relationships not taught in the lecture, it is your responsibility to make sure your UML relationships are right to avoid any possible deductions. (**5 marks**)
2. Implement Nimsys in Java according to the above specification and in accordance with your UML class diagram. (**15 marks**)

4 Assessment

This project is worth 20% of the total marks for the subject. Project C will build on this project. In order to pass the hurdle (20/40) for all projects, it is very important that you do well in this project first.

The deadline for the project is **3pm, Monday 6 May, 2019**. **There is a 20% penalty per day for late submissions. Suppose your project gets a mark of 10 but is submitted within 1 day after the deadline, then you get 20% penalty and the mark will be 8 after penalty. There will be no mark for submissions after 3pm 10 May, 2019**.

For the UML diagrams, you may submit a preliminary version before **3pm, Monday 15 April, 2019** to get feedback before the final submission. This step is not compulsory and also not marked, but is

highly encouraged. Only the final UML submission will be marked, due at **3pm, Monday 6 May, 2019**. Multiple submissions for the UML diagrams are allowed, yet only the last submission will be assessed. Concerning the submission steps, please refer to Section 5.

Your UML diagram will be marked based on how well it reflects the program structure as described in this project specification. Your Java program will be assessed based on correctness of the output as well as quality of code implementation. See LMS for a detailed marking scheme.

5 Submission

For the UML diagrams, please submit your files in the format of **jpg** or **png** on LMS. The detailed steps are as follows

1. log on to the LMS and click Projects section;
2. below the projects table there is a link named "ProjB UML Diagram Submission", click it;
3. the submission window will pop out, in the "Attach File" section, please choose "Browse My Computer" and select your UML diagram files on computer;
4. click the "submit" button and you shall see that the submission is successful; whenever you click the "ProjB UML Diagram Submission" again, you can browse the submission history and start a new submission

For the implementation, your program should be contained within a number of well structured and documented Java classes. The entry point of your program should be in a class called Nimsys (in a file called Nimsys.java). Thus, your program will be invoked via: `java Nimsys` (you do not need to type this command).

Your Java classes should be stored together in their own directory under your home directory on the student server. Then, you can submit your work using the following command:

```
submit COMP90041 projB *.java
```

Note that you must submit all your Java files, not just `Nimsys.java`. If you submit you code multiple times, the later submission will overwrite the previous one. If you submit all your java source codes and then modify one source code, you need to submit all of your source codes agian, not just the modified one.

You should then verify your submission using the following command. This will store the verification information in the file '`feedback.txt`', which you can then view:

```
verify COMP90041 projB > feedback.txt
```

You should issue the above commands from within the same directory as where your project files are stored (to get there you may need to use the `cd` 'Change Directory' command). Note that you can submit as many times as you like before the deadline.

How you edit, compile and run your Java program is up to you. You are free to use any editor or development environment. However, **you need to ensure that your program compiles and runs correctly on the student servers**. Submit your program to the student servers **a couple of days before the deadline** to ensure that they work (you can still improve your program). **"I can't get my code to work on the student server but it worked on my Windows machine" is not an acceptable excuse for late submissions.**

6 Individual Work

Note well that this project is part of your final assessment, so cheating is not acceptable. Any form of material exchange, whether written, electronic or any other medium is considered cheating, and so is the soliciting of help from electronic newsgroups. Providing undue assistance is considered as serious as receiving it, and in the case of similarities that indicate exchange of more than basic ideas, formal disciplinary action will be taken for all involved parties. A sophisticated program that undertakes deep structural analysis of Java code identifying regions of similarity will be run over all submissions in "compare every pair" mode.