The University of Melbourne
Department of Computing and Information Systems
COMP90041 Programming and Software Development
Semester 1, 2016
Project B
Due: 4pm Monday 9th May 2016

1 Background

This project is the second in a series of three, with the ultimate objective of designing and implementing (in Java) the game of Tic Tac Toe; please refer to Project A for a description of the game and its rules.

In Project A, a simple Java program was created to handle Tic Tac Toe's core game-play mechanics, namely: assigning names to the players, playing a game, and determining the winner. In this project, the objective is to implement a more complete version of the game, making full use of Java's object-oriented paradigm. Key knowledge points covered in this project are:

1. Design of the class structure for the project in UML (Week 5).
2. Implementation of Tic Tac Toe using *Classes* (Week 4 - 6) and *Arrays* (Week 7).

2 Your Task

In this project you will build a Tic Tac Toe "game system" with more functionalities than running a single game. The game system will keep record of multiple players, and allow any two to play a game of Tic Tac Toe. The sections below will walk you through the steps you should follow to complete this project.

2.1 UML Class Design (5 Marks in Total)

Before starting to write the code, you will be required to design your solution using the Universal Modelling Language (UML). It will help you to write better code.

Your UML model should detail all the classes you will use in your solution, including their instance variables and methods (with visibility modifiers), and the relationship between the classes. Specifically, your model should at least have the following core classes:

- `TicTacToe` (the main class)
- `Player`
- `PlayerManager`
- `GameManager`

You are free to add more classes to your design. However, *if your design do not have the four core classes above, you need to justify in detail why your design is a good one, or penalties may apply.*

2.1.1 The TicTacToe Class (1 Marks)

The `TicTacToe` class will manage the overall game system. Similar to Project A, your `TicTacToe` class should have a `main` method. This method will create an object of the `TicTacToe` class named `gameSystem` and call a method `run` to run the game system:

```
TicTacToe gameSystem = new TicTacToe();
gameSystem.run();
```

You need to implement the `run` method to **support the functionalities as described in Section 2.2 (NOT just running a game as in Project A)**. Again, you should make `run` an instance method of the `TicTacToe` class, not static.

The `TicTacToe` class should have an instance variable of the `PlayerManager` class as described in Section 2.1.3 to mange the players in the game system; it should also have an instance variable of the `GameManager` class as described in Section 2.1.4 to mange a single game-play between two players. You should also add a proper constructor to initialise these instance variables.

2.1.2 The Player Class (1 Mark)

The `Player` class represents a single player in the game, and should have the following information:

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

You should add instance variables to the `Player` class to store this information. You should also add appropriate accessors, mutators, and constructors to this class.

2.1.3 The PlayerManager Class (1 Mark)

The game system allows creating multiple players (**up to 100** objects of the `Player` class). To manage the players created, you need to have a `PlayerManager` class which has an array to store all the current players in the game system, and methods to access the players in the array, including:

- Add a new player
- Remove one or all players
- Modify the name of a player
- Reset the statistics of one or all players
- Display the information of one or all players
- Rank the players based on winning history

Further details of each of these methods are provided in Section 2.2.

2.1.4 The GameManager Class (1 Mark)

The new game system will use a `GameManager` class to handle a single game-play between two players, *which is similar to the* `TicTacToe` *class you wrote for Project A*.

The `GameManager` class should have the following information associated with it:

- The state of the game board cells (now you should use an array to represent the cells)
- Two players of the game

`GameManager` should at least have a method named `playGame` that takes in two players, runs a game, and updates the game playing statistics of the players. You will need to add extra methods to support the class.

2.1.5 Association between the Classes (1 Mark)

You also need to associate the classes properly and add multiplicity values where appropriate.

2.2 Implementation (15 Marks in Total)

2.2.1 System Initialisation and Exit (2 marks)

When `gameSystem.run()` is called, it should display a welcome message, followed by a blank line. It should then display a command prompt (a "greater than" sign, '`>`'), as shown in the box below (*note: you should type in the text yourself instead of copying from this PDF file because copying from PDF may bring non-English characters into your program which may fail the compilation at submission*).

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

```
Welcome to Tic Tac Toe!

>
```

Next, the `run` method should process user input commands. The user input commands are entered at the command prompt. If a command produces output, it should be printed immediately below the line where the command was issued. After the command has been 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 blank line.

There might be errors in the commands. Your program needs to handle them. We will list all the commands and possible errors your program needs to handle in the following discussion. **You do not need to consider any errors that have not been listed below.**

1. **exit**: Exit the program.

   Note that you need to print a blank line before you call the exit method "`System.exit(0);`".

   Syntax: `exit`

   ```
   >exit
   ```

2.2.2 Player Modification (7 marks)

Recall that the `TicTacToe` class has an instance variable which is an object of the `PlayerManager` class ("the `PlayerManager` object" for short). Your `run` method will call methods of this `PlayerManager` object to handle the following player management commands.

2. **addplayer**: Allow new players to be added to the game by calling the `addPlayer` method of the `PlayerManager` object. If a player with the given `username` already exists, the system should indicate this error, as shown in the example execution.

   Note that the player username, family name, and given name are separated by comma. There is no space in between. You may assume that the username, family name, and given name only contains letters (e.g., no space or comma characters).

   Syntax: `addplayer username,family_name,given_name`

   (a) add a new user:

   ```
   >addplayer rbukater,Bukater,Rose

   >
   ```

   (b) add a user with an existing username

```
>addplayer rbukater,Bukater,Ramsey
The username has been used already.

>
```

3. **removeplayer**: Allow players to be removed from the game system by calling the `removePlayer` method of the `PlayerManager` object. 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, your program should prompt for confirmation 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]`

   Note here "`[]`" indicates an optional command argument. Same applies to the following commands.

   (a) remove a non-existent user

   ```
   >removeplayer lbukater
   The player does not exist.

   >
   ```

   (b) remove a user

   ```
   >removeplayer rbukater

   >
   ```

   (c) remove all users

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

   >
   ```

4. **editplayer**: Allow player family and given names to be edited by calling the `editPlayer` method of the `PlayerManager` object. Note that a 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`

   (a) edit a non-existent user

   ```
   >editplayer lbukater,Bukater,Ramsey
   The player does not exist.

   >
   ```

   (b) edit a user

   ```
   >editplayer rbukater,Bukater,Ramsey

   >
   ```

5. **resetstats**: Allow player statistics to be reset (that is, setting the numbers of games played/won/drawn to be 0) by calling the `resetStats` method of the `PlayerManager` object. 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, your program should prompt for confirmation first. 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]`

(a) reset a non-existent user

```
>resetstats lbukater
The player does not exist.

>
```

(b) reset a user

```
>resetstats rbukater

>
```

(c) reset all users

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

>
```

6. **displayplayer**: Display player information by calling the `displayPlayer` method of the `PlayerManager` object. 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.

   Syntax: `displayplayer [username]`

   (a) display a non-existent user

   ```
   >displayplayer lbukater
   The player does not exist.

   >
   ```

   (b) display a user

   ```
   >displayplayer rbukater
   rbukater,Bukater,Rose,3 games,2 wins,1 draws

   >
   ```

   (c) display all users

   ```
   >displayplayer
   jdawson,Dawson,Jack,3 games,0 wins,1 draws
   rbukater,Bukater,Rose,3 games,2 wins,1 draws

   >
   ```

   Note that the output syntax for a player is:

   `username,family_name,given_name,games_played,games_won,games_drawn`

   You may always use plural forms for the game numbers (e.g., `1 draws`).

2.2.3 Player Ranking (3 Marks)

7. **rankings**: Outputs the top-10 players ranked based on winning ratio from the highest to the lowest by calling the `displayRanking` method of the `PlayerManager` object. Ties are broken first by drawing ratio and then by username alphabetically. A player that has played 0 games should have a 0% winning/drawing ratio. If there are fewer than 10 players, then output all the existing players.

   For the purposes of formatting, you may assume that no players have played for more than 99 games. The output columns need to be aligned. The first two columns should have 6 characters each, consisting

of a space, a rounded integer, a percentage sign '%', and another space. The third column has 6 characters consisting of a space, an integer, and spaces. The fourth column starts with a space followed by a player username. There is **no** size limit or space at the end of this column.

Syntax: `rankings`

```
>rankings
 WIN   | DRAW | GAME | USERNAME
 100% |   0% | 99   | alphaTicTacToe
  67% |  33% |  3   | rbukater
   0% |  33% |  3   | jdawson
   0% |   0% |  0   | jqi
   0% |   0% | 99   | jsmith

>
```

2.2.4 Game Play (3 Marks)

Recall that the `TicTacToe` class has an instance variable which is an object of the `GameManager` class ("the `GameManager` object" for short). Your `run` method will call the `playGame` method of this `GameManager` object to handle a game between two players.

8. **playgame**: Play a game of Tic Tac Toe by calling the `playGame` method of the `GameManager` object, with the usernames of the two players provided as arguments. If either usernames does not correspond to an actual player, the system should indicate this by the output "`Player does not exist.`", and the game should not commence. Otherwise, the `playgame` command will commence a game.

   When a game is in progress, the two players should alternate in placing an '`O`' or '`X`' on the game board, until a player has a sequence of three and wins the game, or the board is full and there is a draw. The players may make two types of invalid moves: (i) placing at a cell that is already occupied, or (ii) placing at a cell beyond the boundary of the game board. Your program should print an error message in either case, and prompt for the player to make a move again.

   Once the game is over, the game statistics for the two players should be updated. The system control should then be given back to the `run` method, and a command prompt should be displayed again.

   Syntax: `playgame username1,username2`

   (a) start game with a non-existent user

   ```
   >playgame lbukater,jdawson
   Player does not exist.

   >
   ```

   (b) start a game

   ```
   >playgame rbukater,jdawson
    | |
   -----
    | |
   -----
    | |
   Rose's move:
   1 1
    | |
   -----
    |O|
   -----
    | |
   Jack's move:
   1 1
   Invalid move. The cell has been occupied.
   Jack's move:
   ```

```
3 3
Invalid move. You must place at a cell within {0,1,2} {0,1,2}.
Jack's move:
2 2
 | |
-----
 |O|
-----
 | |X
Rose's move:
1 2
 | |
-----
 |O|O
-----
 | |X
Jack's move:
2 1
 | |
-----
 |O|O
-----
 |X|X
Rose's move:
1 0
 | |
-----
O|O|O
-----
 |X|X
Game over. Rose won!

>
```

Note that if the game ends with a draw, then print out:

```
Game over. It was a draw!

>
```

2.3 Checklist for the Implementation

- **Error handling**
  Only the follow 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 username has been used already.`
  Follow-up operation: Prompt for user input again.

  ☐ Removing a player with a non-existing username.
  Error message: `The player does not exist.`
  Follow-up operation: 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: Prompt for user input again.

  ☐ Displaying a player with a non-existing username.
  Error message: `The player does not exist.`
  Follow-up operation: Prompt for user input again.

  ☐ Starting a game where at least one of the player's username does not exist.
  Error message: `Player does not exist.`
  Follow-up operation: Prompt for user input again.

- ☐ Placing a mark at an already occupied cell.
  Error message: `Invalid move. The cell has been unoccupied.`
  Follow-up operation: Prompt for the same player to place a mark again.
- ☐ Placing a mark at a cell outside the valid game board range.
  Error message: `Invalid move. You must place at a cell within {0,1,2} {0,1,2}.`
  Follow-up operation: Prompt for the same player to place a mark again.

- **Blank line and whitespace**
  - ☐ Make sure that there is one blank line between the output of the last command (including indication for non-existent users, confirmation for all-user operations, display results, and game results) and the next command prompt.
  - ☐ Make sure that there is a whitespace between "`(y/n)`" and the "`Are you sure...`" sentence.
  - ☐ Make sure that there is NO whitespace after any comma, e.g., when displaying users, it should be "`Dawson,Jack`" instead of "`Dawson, Jack`".
  - ☐ Make sure that in game play, there is NO blank line between the board display and the prompt for the next move.
  - ☐ Make sure that if a user attempts an invalid move in the game, there is NO blank line between the user input move and the indication sentence "`Invalid move...`."
  - ☐ Make sure that a blank line is printed out before calling "`System.exit(0);`" when the `exit` command is processed.
- **Player ranking**
  - ☐ Make sure that the ranking is in the descending order of winning ratio. Ties are broken first by drawing ratio and then by username alphabetically.
  - ☐ Make sure that the displayed winning/drawing ratio is rounded to the nearest integer value. However, the winning/drawing ratio used to rank the players should be in `double`.
  - ☐ Make sure that the first two columns have 6 characters each.
- **Game play**
  - ☐ Make sure to update the game statistics for the two players after each game.
  - ☐ Make sure to check for invalid moves by checking if the cell is within range, and unoccupied.
  - ☐ Make sure that after an invalid move, it is still the turn of the player who made the invalid move.
- **Command line prompt**
  - ☐ Make sure that the command prompt appears again after each command is executed (except for `exit`).
  - ☐ Make sure that only one command prompt is displayed after a game is over.
- **Other issues**
  - ☐ The maximum number of players is 100.
  - ☐ The boxes enclosing the above example executions are NOT to be printed out, they are purely for illustration.

Please note that:

- You will be given a sample test input file `test0.txt` and the corresponding sample output file `test0-output.txt`. When you run your program by the following command in a terminal (or Windows command line):

  ```
  java TicTacToe < test0.txt > my-output.txt
  ```

  your program should produce a file name `my-output.txt` which should be exactly the same as `test0-output.txt`. In this command, "`< test0.txt`" and "`> my-output.txt`" are called "input redirection" and "output redirection." They use the content in `test0.txt` as the command line input, and print the program output into `my-output.txt`.

- Tests will be conducted on your program by automatically compiling, running, and comparing your outputs for several test cases with generated expected outputs. **These test cases used will be different from the sample test file given**. 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**. Therefore, it is crucial that **you generate your own test files and test your program extensively**.
- The syntax `import` is available for you to use standard Java packages. However, please **DO NOT** use the `package` syntax to customize your source files. The automatic test system cannot deal with customized 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 any lines of the form

  ```
  package ProjA;
  ```

  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 any remaining Scanner objects with nothing to read, causing a 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.

  You may declare an object `keyboard` of the `Scanner` class as a `public static` variable of the `TicTacToe` class. Then you can use this object throughout your program by "`TicTacToe.keyboard`."

3 Assessment

This project is worth 20% of the total marks for the subject. It is very important that you do well in order to pass the project hurdle (20/40). The breakdown of marks is as follows:

1. Draw a UML class diagram based on the above specification. 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 all relationships between classes. (**5 marks**)
2. Implement the game system in Java according to the above specification and in accordance with your UML class diagram. (**15 marks**)

For the UML diagrams, you may submit a preliminary version before **6pm Friday 29th April 2016** to get feedback before the final due date.

Your UML diagram will be assessed 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. *You Java program should follow a good practice of encapsulation and information hiding.* See LMS for a detailed marking scheme.

4 Submission

4.1 UML Diagram Submission

For the UML diagrams, please submit your files in the format of **jpg** or **png** on LMS:

1. Log on to LMS and navigate to the Projects page.
2. Click on the "ProjB UML Diagram Submission" link below the Projects table.
3. The submission window will pop out. In the "Attach File" section, please choose "Browse My Computer" and select your UML diagram file on computer.
4. Click the "submit" button and you shall see that the submission is successful. You can click on "ProjB UML Diagram Submission" again to browse your submission history or to start a new submission.

Multiple submission for the UML diagram are allowed, but **ONLY THE LAST** submission will be assessed.

4.2 Java Program Submission

Your Java 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 named `TicTacToe` (in a file named `TicTacToe.java`). You should upload all the relevant Java code files to the Engineering School student server. Then, you can submit your work using the following command:

```
submit 90041 B *.java
```

Note that this command will submit all your Java files under the current directory, not just `TicTacToe.java`. If you have changed any of your Java source files after a submission, you need to submit **all** of your source files again, not just the modified one.

You should then verify your submission using the following command:

```
verify 90041 B > feedback.txt
```

This will store the verification information in the file `feedback.txt`, which you can then view using the following command:

```
more feedback.txt
```

You should issue the above commands from within the same directory as where the 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 Engineering School student server**, using **build 1.8.0** of Oracle's (as Sun Microsystems has been acquired by Oracle in 2010) Java Compiler and Runtime Environment, i.e., `javac` and `java` programs. Submit your program to the server 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.**

The deadline for the project is **4pm Monday 9th May 2016**. The allowed time is more than enough for completing the project. Penalties will apply on late submissions at 4 marks per day. Late submissions after 4pm Thursday 12th May 2016 will NOT be accepted.

5 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.