The University of Melbourne
School of Computing and Information Systems

COMP90041
Programming and Software Development
Second Semester, 2018

Third Assessed Exercise (lab3)

Submission due Saturday, 15 September 2018, 5:00PM

These exercises are to be assessed, and so **must be done by you alone**. Sophisticated similarity checking software will be used to look for students whose submissions are similar to one another.

For this project, we will turn the tables. Instead of you writing code to implement a specification and me testing it, I will give you a specification and an implementation, and ask *you* to test it. You will be assessed on how thoroughly you test that the implementation meets its specification.

The class you will be given, `BackgammonBoard`, represents a game in the ancient Persian board game of Backgammon, as depicted below.

The 24 triangles on the board, called *points*, can each hold any number of disk-shaped game pieces, called *men*, as long as all the men on a point are the same colour. We number the points from 0 (lower right) through 11 (lower left) to 12 (upper left) through to 23 (upper right). There are two colours of men; actual colours vary depending on the Backgammon set, but we shall call the colours *black* and *white*.

The two players face each other across the board; the player on the near side moves the black pieces, and the facing player moves white. The two players take turns rolling dice which determine how far they are allowed to move how many of their men in that turn. White men always move from lower to higher numbered points (clockwise), and black men move from higher to lower numbered points (anti-clockwise). Each move consists of the player selecting one of his or her pieces and moving it one of the numbers of steps indicated by the dice. A piece may be moved to an empty point, or to a point occupied by his or her own men, or to a point occupied by exactly one of the opponent's men. In the latter case, the opponent's man is moved from the destination point to the *bar* running from top to bottom down the middle of the above picture, and the player's man becomes the sole occupant of that point. It is not permitted to move a piece onto a point occupied by two or more of the opponent's men. For example, one of the men in the lower left of the above picture (on point 11) would be allowed to move to the empty point 13, or to point 16, because it is occupied by men of the same colour, but not to point 12, because that is occupied by more than one of the opponent's men. It is also not permitted to move a man directly from point 11 to point 18, because that is more than 6 points away.

The `BackgammonBoard` class will have the following public methods:

`BackgammonBoard()` (constructor) Creates a fresh `BackgammonBoard` with no men on it.

`int getPointCount(int point)` Returns the number of men on the point.

`boolean getPointBlack(int point)` Returns `true` if the men on the specified point are black, or `false` if they are white. If there are no men on that point, the result may be either `true` or `false`.

`void setPoint(int point, int count, boolean black)` Sets the number of men on the specified point to the specified count, and sets their colour to black if `black` is true, or white if it is false.

`int getBarBlackCount()` Returns the number of black men on the bar.

`int getBarWhiteCount()` Returns the number of white men on the bar.

`void move(int fromPoint, int toPoint)` Moves one men from the specified `fromPoint` to the specified `toPoint`, if the move is legal; if it is illegal, this method does nothing.

For `setPoint` and `move` methods, if a specified point number is not between 0 and 23, the method should do nothing. For `getPointCount` and `getPointBlack`, if the specified point is out of bounds, the value returned may be any valid value of the appropriate type.

As mentioned above, you do not need to implement the `BackgammonBoard` class; an implementation will be provided. You must implement a class called `BackgammonTest` with a public main method that will thoroughly test that the `move` method of the `BackgammonBoard` class behaves correctly. You may assume that the other methods listed above behave exactly as specified.

For this exercise, you will be assessed on what proportion of the ten specific bugs in the `move` method I have selected (from a multitude of possible bugs) are detected by your `BackgammonTest` main method, without reporting a bug for a correct `BackgammonBoard` class. If your code reports a bug when there is none, you will receive no marks for this lab. Your `BackgammonTest` program should print out "`BUG`" as a single line (without the quotes) if it detects a bug, or "`CORRECT`" as a single line if it does not detect a bug. It should not print out anything else (but of course the line should end with a newline).

You will be supplied with a `BackgammonBoard.class` file that correctly implements the `BackgammonBoard` class. You will also receive a `Backgammon.jar` file which also contains the `BackgammonBoard` class, but you will find it easier to use if you program with an IDE. Note that you do not need to submit either of these files with your code.

To develop your code in an IDE, you will need to add the `Backgammon.jar` file so the IDE can find it. If you are using Eclipse, at the far left of the window, in the Package Explorer, right click on the project you will use for this lab and select "Properties". In the dialog that pops up, select "Java Build Path" on the left, and select the Libraries tab at the top. On the right side of the dialog, select "Add External JARs...". In the file selection dialog that pops up, navigate to where you placed the `Backgammon.jar` file, select it, click "Open", and then click "OK" back in the Properties dialog. Finally back in the Eclipse window, right click your project again and select "Refresh". Now you are ready to develop and test your `BackgammonTest` class.

If you are using NetBeans, you should look at the far left of the NetBeans window in the Projects tab and find the project you are using for this lab. Under this project, you will see a Libraries label. Right click on this label and select "Add JAR/Folder...". In the file selection dialog, navigate to where you placed the `Backgammon.jar` file, select it, and click "Choose". Now you are ready to develop and test your `BackgammonTest` class.

If you are working without an IDE, or when you are testing on the department server, you should put the `BackgammonBoard.class` file in the same directory (folder) as your `BackgammonTest.java` file, and you will be able to compile your class.

When you run your testing code, it should print out "`CORRECT`", indicating that it found no bugs in the `BackgammonBoard` class. On submission, the `BackgammonBoard` will be made to exhibit ten different bugs, and your `BackgammonTest` class will be expected to detect them. It is up to you to be thorough in your testing.

As usual, when you verify your submission, you will see how well you have done detecting bugs. If you have missed some, look closely at the spec for the `move` method, and ensure you test every way you can think of for this method to misbehave. You can submit as many times as you like.

Hint: Be sure to test that the `move` method behaves as specified for *incorrect* inputs (error cases) as well as correct inputs.

Hint: It is best to create a fresh `BackgammonBoard` object for each test.

Hint: If your code is not able to find all the bugs, return to the above specification and re-read it carefully for some aspect you have not tested.

Submission

You must submit your project from any one of the student unix servers. Make sure the version of your program source files you wish to submit is on these machines (your files are shared between all of them, so any one will do), then `cd` to the directory holding your source code and issue the command:

```
submit COMP90041 lab3 BackgammonTest.java
```

Important: you must wait a minute or two (or more if the servers are busy) after submitting, and then issue the command

```
verify COMP90041 lab3 | less
```

This will show you the test results and the marks from your submission, as well as the file(s) you submitted. If the test results show any problems, correct them and submit again. You may submit as often as you like; only your final submission will be assessed.

If you wish to (re-)submit after the project deadline, you may do so by adding "`.late`" to the end of the project name (*i.e.,* `lab3.late`) in the `submit` and `verify` commands. But note that a penalty, described below, will apply to late submissions, so you should weigh the points you will lose for a late submission against the points you expect to gain by revising your program and submitting again. **It is your responsibility to verify your submission.**

Late Penalties

Late submissions will incur a penalty of 1% of the possible value of that submission per hour late, including evening and weekend hours. This means that a perfect project that is a little more than 2 days late will lose half the marks. These lab exercises are frequent and of low point value, and your lowest lab mark will be dropped. Except in unusual circumstances, I will not grant extensions for lab submissions.

Academic Honesty

This lab submission is part of your final assessment, so cheating is not acceptable. Any form of material exchange between students, 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.