INF3121 - Assignment 1

by Henrik Lilleengen (henrilil), Andreas Thompson (andrerth) and Simon Pettersen (simonpe)

Requirement 1

Brief description of the program

For our project assignment we chose to work with Hangman written in Java, we chose this because we all have experience in java and we were all familiar with Hangman.

Hangman is a classic childs game where the premise lies in guessing the secret word one letter at a time. There are two parts to the game, the gamemaster and the player, usually the gamemaster chooses the secret word and draws the amount of letters as underscores ‘ _ ’. The player then has to guess the secret word one letter at a time, if the player chooses a correct letter the correlating underscore becomes said letter. If the player on the other hand chooses an incorrect letter, the man gets one step closer to getting hung. The game ends when either the man is fully hung or the word is fully written out.

In the program version the premise is almost the same, there are only two major differences, the computer choosing the word instead of another ‘player’. The player chooses one letter at a time, same as in real life. The other flaw/difference is that the program does not handle losses, you get as many tries as you want, and it only informs you how many mistakes you did.

Analysis of the testable parts of the program

Since this is a very informal test process with very little documentation, there are not a lot of parts to analyze. Mainly you have the menu, input and output, using different pre/post conditions. We've based these conclusions on the fact that this assignment resembles what you could expect to experience when continuing on a project another team has made, for example the only documentation provided are a couple hungarian comments.

Some simple manual tests are:

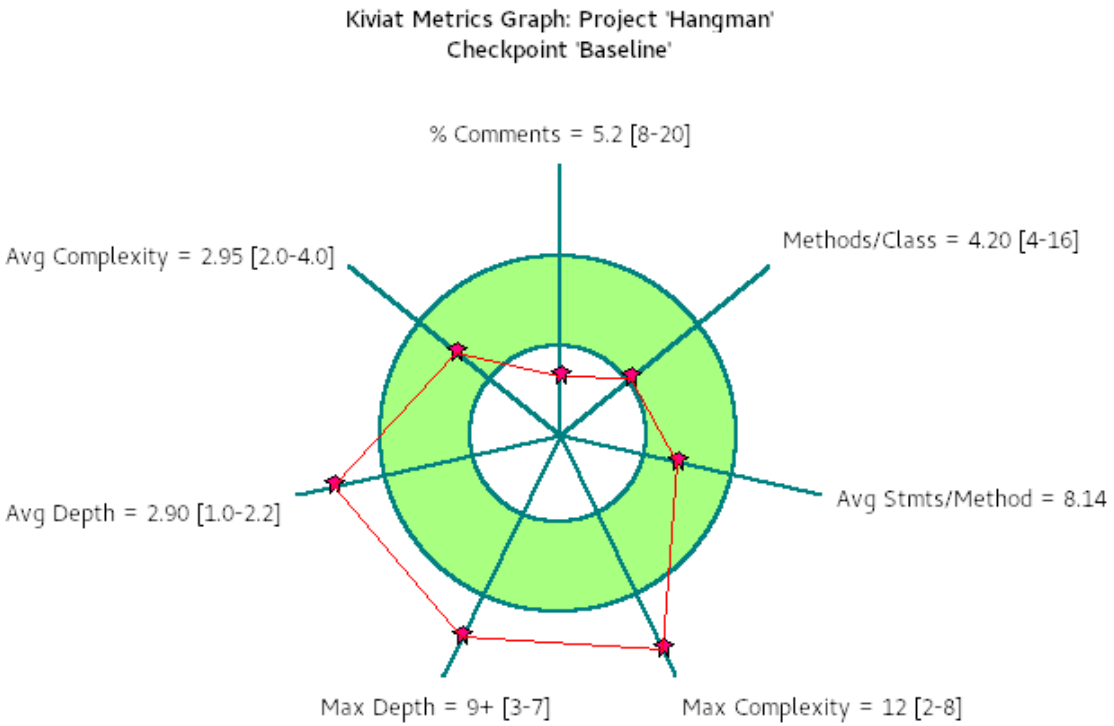
1. Testing Menu Options: Start the program, proceed to test that each of the different menu options does its Intend job, and without any hiccups: TOP, RESTART, HELP, EXIT. Also do excessive testing to reveal bugs that lie within repeating the same command several times.
2. Successful completion: Start the program and input letters until the word is complete, check for errors or faults.
3. Repeating the same mistake: Start the program, proceed to enter the same incorrect letter several times, then complete the word if possible, check for errors or faults.
4. Repeating the same correct letter: Start the program, proceed to enter 1 correct letter several times before finishing the word if possible, check for errors or faults.
5. Entering several letters at once: Start the program, proceed to enter two or more letters, complete the word if possible, check for errors or faults. These tests were created with black box testing and use case testing in mind.

Would it make sense or not to write non-functional tests for the chosen source code?

Yes, to a certain degree. You could do some compliance testing, endurance testing and usability testing to ensure a quality product. But to what degree these testing practices contribute to a program of this size and complexity is questionable.

Concrete test cases

TEST 1 - Startup and menu test	**TEST 2 - Program runthrough completion**	**TEST 3 - Repeating mistakes**	**TEST 4 - Repeating correct answers**	**TEST 5 - Several letters at once**
Preconditions: The program at its default condition, just after launching it. **Test description:** Test the functionality and success of the various commands: Including: TOP RESTART HELP EXIT **Expected result:** All commands should run without error, and complete their designed task. **Actual result:** RESTART and EXIT worked perfectly. HELP had 1 minor fault in which if you used HELP to complete the whole word you would get prompted to input another letter once the whole word was sown instead of the underscores. Using HELP again crashes the program, while inputting a correct letter completes the word with no mistakes. A incorrect input letter completes the word with one more mistake. The command TOP does not work correctly, it prints an error message 'Error during reading of file' **Postconditions:** No real change mostly because the leaderboard doesn't work.	**Preconditions:** The program at its default condition, just after launching it. **Test description:** Go through the whole game to test that it works correctly. **Expected result:** After inputting all the correct letters it will prompt you for your name to the leaderboard, as long as you did not use the command help. **Actual result:** The program works as described, the only error is the leaderboard, the program doesn't correctly handle writing/reading of files. **Postconditions:** No change, again because of the read/write problem.	**Preconditions:** The program at its default condition, just after launching it. **Test description:** Repeat the same wrong letter several times. **Expected result:** The program ignores repeated mistakes. **Actual result:** The program adds one mistake each time you repeat a wrong letter. **Postconditions:** Again, there are no post conditions, because of the r/w problem.	**Preconditions:** The program at its default condition, just after launching it. **Test description:** Repeat the same correct input several times. **Expected result:** Nothing happens, the system just prompts you for another input. **Actual result:** Nothing happens, the system prompts another input, and no mistakes are recorded. **Postconditions:** Again, there are no post conditions, because of the r/w problem.	**Preconditions:** The program at its default condition, just after launching it. **Test description:** Try several combinations of various inputs. **Expected result:** Nothing happens, the system just prompts you for another input. **Actual result:** Nothing happens, the system prompts another input, and no mistakes are recorded. **Postconditions:** Again, there are no post conditions, because of the r/w problem.



Parameter	Value
Project Directory	Z:\home\ithenrik\Projects\School\INF3121\S
Project Name	Hangman
Checkpoint Name	Baseline
Created On	7 Mar 2016, 14:51:47
Files	5
Lines	441
Statements	222
Percent Branch Statements	18.5
Method Call Statements	147
Percent Lines with Comments	5.2
Classes and Interfaces	5
Methods per Class	4.20
Average Statements per Method	8.14
Line Number of Most Complex Method	{undefined}
Name of Most Complex Method	Game.findLetterAndPrintIt()
Maximum Complexity	12
Line Number of Deepest Block	{undefined}
Maximum Block Depth	9+
Average Block Depth	2.90
Average Complexity	2.95

Most Complex Methods in 4 Class(es):	Complexity, Statements, Max Depth, Calls
FileReadWriter.printAndSortScoreBoard()	10, 24, 8, 32
Game.findLetterAndPrintIt()	12, 51, 7, 43
HangmanTets.main()	1, 2, 2, 2
Players.getScores()	1, 1, 2, 0

Block Depth	Statements
0	20
1	31
2	59
3	45
4	32
5	12
6	7
7	6
8	2
9+	8

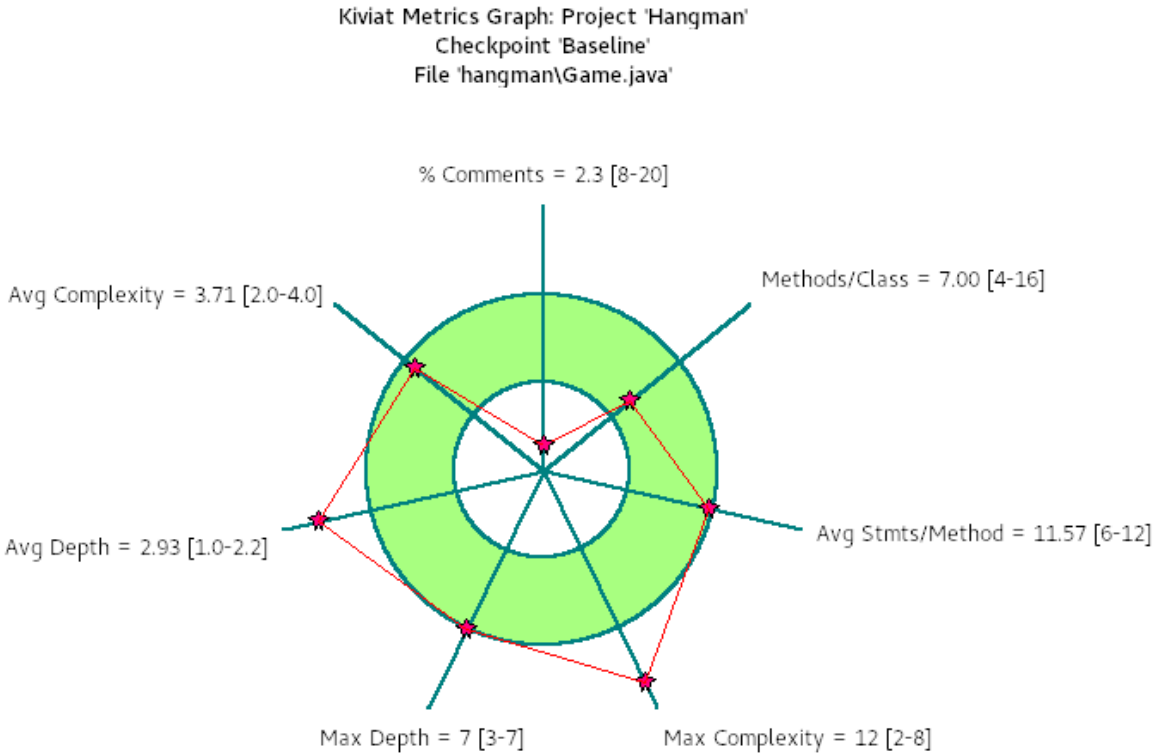
Requirement 2

Metrics at project level

Files is the amount of files in the project.
Lines are how many lines of code there is in the project
Statements are computational statements are terminated with a semicolon character.
Percent branch statements gives the amount of branch statements, such as **if, else, for, do, while, break** in the code as a percentage.
Method call statements is the amount of method call statements
Percent lines with comments show how many percent of the lines have comments - it lies at 5.2% which seems a tad low, would perhaps be better with between 8 and 20 %.

Classes and interfaces is the amount of classes and interfaces in the project
Methods per class is the amount of methods per class - it lies at 4.20 which seems like a reasonable number.
Average statements per method gives the average statements per method.
Line number of most complex method tells you where you can find the most complex method, here it is undefined as we are looking at more than one file.
Name of most complex method tells us what method is the most complex
Maximum complexity indicates the highest complexity in our project. It lies at 12 which seems over the top. I would very much like to reduce this.
Average block depth is 2.90
Average complexity is 2.95
Both of these averages seem a bit on the high end, so i believe we could reduce them.

File with the most lines: **FileReadWriter.java**
File with the most branches: **FileReadWriter.java**
The file with the most complex code is: **Game.java** - I looked at both avg complexity and max complexity and it was the highest on both these metrics of all the files in the project.



Metrics at file level

Game.java: The metrics here show us that the maximum complexity should be lower and the avg depth should be reduced. There is also a serious lack of comments. As well as fixing this we should also seek to reduce average statements/method, maximum depth and average complexity. This graph is fairly similar with the project graph, this is not shocking since the project consist of only two big files. We would definetly rewrite the function findLetterAndPrintIt, and add comments in every method.

Requirement 3

At project level the metrics we need to alter are max complexity which before any changes to the code lies at 12, we would ideally have this between 2 and 8. We would also like to change avg depth, max depth and comments. Avg depth is at 2.90, and it should be between 1.0-2.2. Max depth is at 9+ and should lie at between 3 and 7. Comments are not too bad, it is at 5.2, and we would like to have it between 8 and 20.

Code examples

Some examples of the code we ended up fixing are changing is the while-loops in findLetterAndPrintIt() from Game.java:

```
        while (!dashBuff.equals(guessWord)) {
            //Print progress
            System.out.println("The secret word is:" + dashBuff.replace("", " "));
            System.out.println("DEBUG " + guessWord);

            letter = input.next();
            //Check input
            if (letter.equals(Command.help.toString())) {
                //Run help
                isHelpUsed = true;
                dashBuff = help(dashBuff);
            //If letter is a single letter
            } else if (letter.matches("[a-z]")) {
                String tmp = checkLetter(letter, dashBuff);

                //If return value from checkLetter is null, increase mistakes
                mistakes += (tmp == null) ? 1 : 0;
                //Else, set dashBuff to the return value
                dashBuff = (tmp != null) ? tmp : dashBuff;
            } else {
                menu(letter);
            }
        }
```

A lot of the code has been moved to the two new functions help() and checkLetter(), and some other minor things have also been cleaned up. In our opinion, a much cleaner code.

We have also opted to remove the functions “private StringBuffer getW(String word)” and “private String printDashes(StringBuffer word)” and avoided StringBuffer all together in Game.java. Our solution is to use strings, and to create a function which replaces the char at the index you choose in a string that you choose with a char that you choose:

```
    private String replaceChar(int i, String inpt, char letter) {
        //Convert input to char-array
        char[] temp = inpt.toCharArray();
        //Replace i with letter
        temp[i] = letter;
        //Return string value of the char-array
        return String.valueOf(temp);
    }
```

This, in our opinion, is a cleaner solution which offers less confusion in the code. The replaceChar()-function could just as easily used StringBuffer, but we opted to go for a simple String -> char[] -> String instead as we found StringBuffer to be a little “bloated” for this task.

As our last code-example we would like to show a snippet from the checkLetter()-function in Game.java:

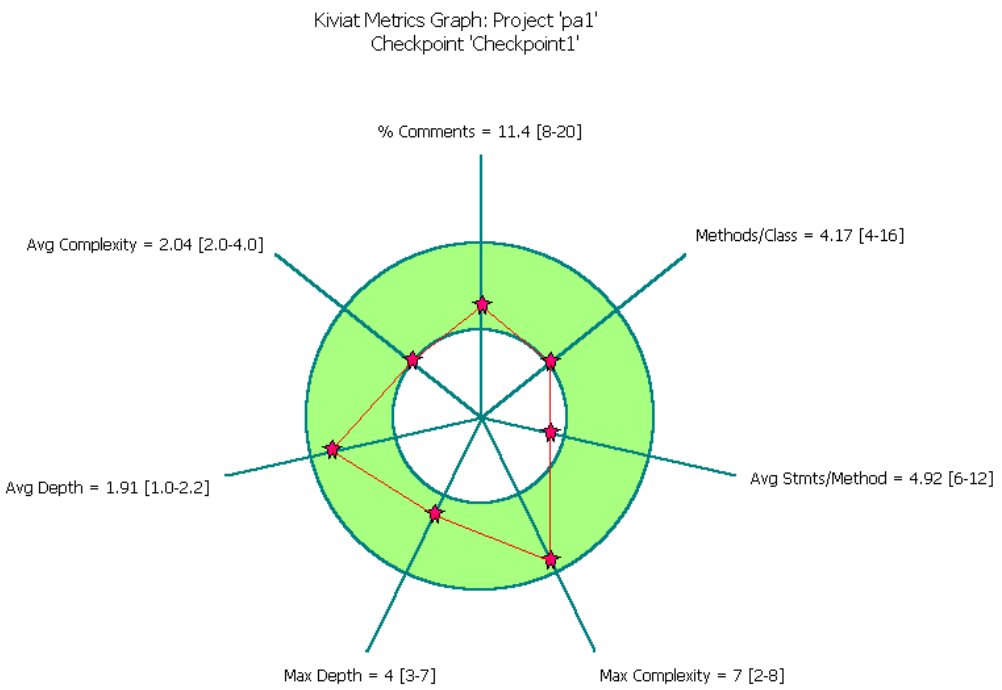
```
        //Set i to the index of the first occurrence of letter in guessWordCopy
        int i = guessWordCopy.indexOf(letter);

        //While the letter is in guessWordCopy
        while(i != -1) {
            //Replace the letter with underscore in guessWordCopy and replace the underscore with letter in dashBuff
            guessWordCopy = replaceChar(i, guessWordCopy, '_');
            dashBuff = replaceChar(i, dashBuff, letter.charAt(0));

            i = guessWordCopy.indexOf(letter);
            ++counter;
        }
```

This code which had the same job in the original program used two while-loops, here with the use of the indexOf()-function in String-class we can use just one while-loop.

We also ended up adding another class to fix writing to the scoreboard-file. As the comments in the file explains, this is not our code, but it belongs to Andreas_D from Stack Overflow (http://stackoverflow.com/a/1195078). As there were nothing in the rules that forbid us from doing so, and the license Stack Overflow have on all answers gave us the permission to reuse it, we found that the use of the code would not be problematic.

Metrics at project level

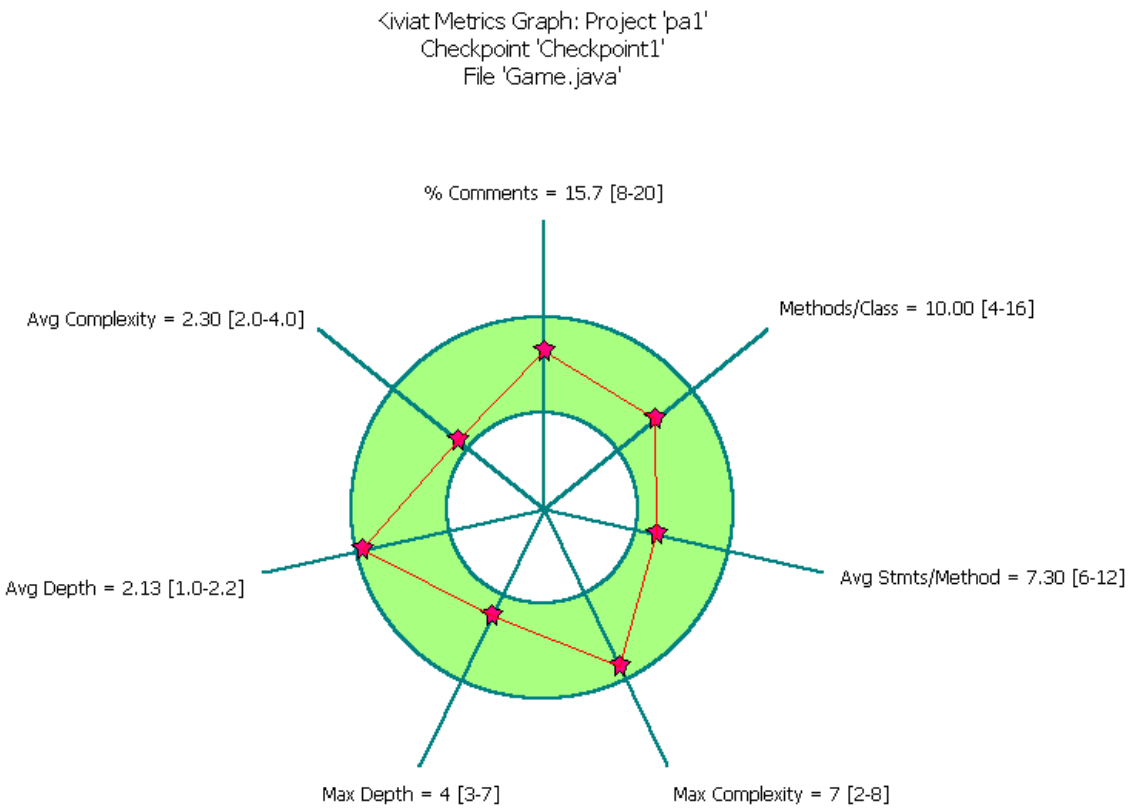
This is the kaviat graph for the whole project after our refactoring. We feel we managed to do what we had hoped, which was to reduce max complexity, max depth and avg depth as well as add more comments. Now almost all the metrics are within the green zone, except for avg stmts/method, which is not far away, so we won't be doing any more refactoring.



Parameter	Value
Project Directory	Z:\home\ithenrik\Projects\School\INF3121\pa1\
Project Name	pa1
Checkpoint Name	Checkpoint1
Created On	10 Mar 2016, 15:37:43
Files	6
Lines	334
Statements	178
Percent Branch Statements	13.5
Method Call Statements	103
Percent Lines with Comments	11.4
Classes and Interfaces	6
Methods per Class	4.17
Average Statements per Method	4.92
Line Number of Most Complex Method	{undefined}
Name of Most Complex Method	Game.findLetterAndPrintIt()
Maximum Complexity	7
Line Number of Deepest Block	{undefined}
Maximum Block Depth	4
Average Block Depth	1.91
Average Complexity	2.04
---	---
Most Complex Methods in 5 Class(es):	Complexity, Statements, Max Depth, Calls
AppendingObjectOutputStream.writeStreamHeader()	1, 1, 2, 1
FileReadWriter.readRecords()	6, 11, 4, 8
Game.findLetterAndPrintIt()	7, 24, 4, 18
HangmanTest.main()	1, 1, 2, 1
Player.getScore()	1, 1, 2, 0
---	---
Block Depth	Statements
0	21
1	34
2	74
3	40
4	10
5	0
6	0
7	0
8	0
9+	0
---	---

Metrics at file level

Game.java: The file looks much better now. Percent of comments is increased, average depth is reduced, maximum complexity is reduced. We have also managed to reduce average statments/method, maximum depth and average complexity. Maximum complexity and average depth is still a little high, but we are highly satisfied with our results.



Github link

https://github.com/Lilleengen/INF3121-hangman-java

Afterwords

We found the code pretty easy to maintain with the help of a IDE (to help us clean up the brackets). As the program was fairly small, there was no code we were not able to understand, but in a larger project this type of coding would be very hard to maintain. A thing we might have done differently, if we had to do this over again would have been to not be so fixated on the kiviat graphs, but rather go for the best code and see where that got us with the graph.