INF3121

Johanpar
Birashat
Tanusanr

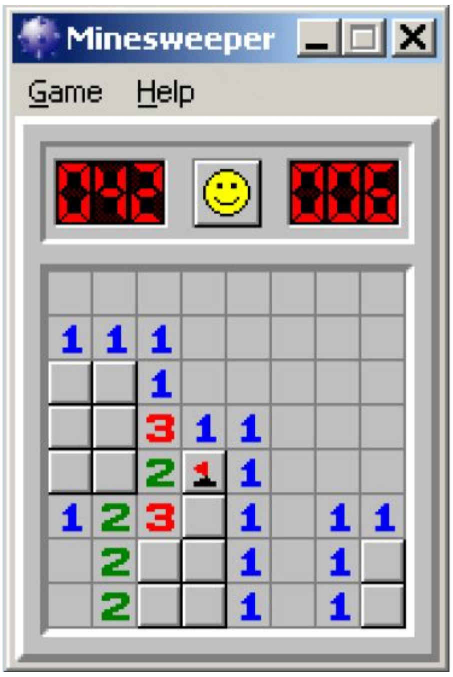
Project Assignment 1: Interpretation of metrics given by a static analysis

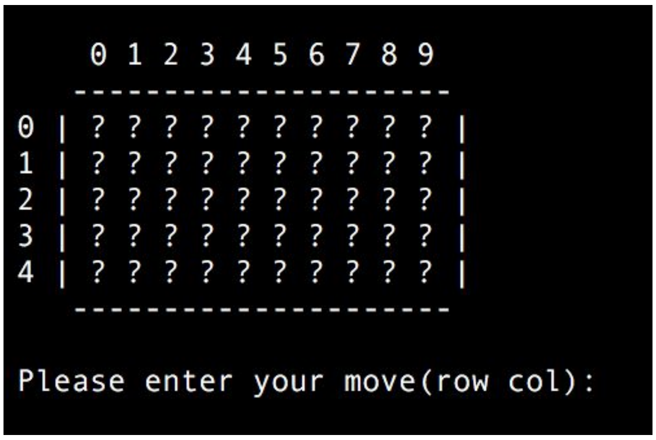
Requirement 1

The program we are going to use is Minesweeper, and we are using java as our programming language. In Minesweeper the object is to find out the spots that are not affected of bombs. The program will randomly generate bombs in some of positions, and the user need to be smart and find all the spots not containing a bomb. If the user is able to find all the spots not containing a bomb, then she/he has won the game. For each spot you find, you will get a number which is the number of bombs around the spot, so the user can calculate of the marked spots where the bombs can be.
The program is build up by three files, Minesweeper being the main file containing the main method. Supported by the MineField file that contains the setup for the game, and the Ranking file that has the player ranking system.

We are going to make tests on the runnable parts of the program, which will be tested by running the program. This will be equivalent to black box testing. The tests will be written below in the table where the tests will be conducted. The test design/technique which will be used will be a IEEE829 test template design, excluded the actual results which we have been instructed to remove by the lecturer. This template which will be used will contain pre-conditions, test cases, input values, expected results and post-conditions. The testable parts of this assignment is when you're playing the game. And they are insertion of the coordinates of the program, inputting your name for the ranking system and the different commands you have which are top(gives the ranking table), restart(finishes the current game and receives the players name, and starts a new game) and at last exit(which receives the players name and exits the game).

Measuring the characteristics of the metrics wouldn't be necessary, for instance to check if the software is user friendly or within a substantial response time. Compared to measuring the characteristics, making tests based on experience is more of an advantage. Most of the test below, are based on previous mistakes we've done in the past, or intuitively thinking of what may go wrong.





```
     0 1 2 3 4 5 6 7 8 9
    ---------------------
0 | ? ? ? ? ? ? ? ? ? ? |
1 | ? ? ? ? ? ? ? ? ? ? |
2 | ? ? ? ? ? ? ? ? ? ? |
3 | ? ? ? ? ? ? ? ? ? ? |
4 | ? ? ? ? ? ? ? ? ? ? |
    ---------------------

Please enter your move(row col):
```

Johanpar
Birashat
Tanusanr

Nr	Pre-condition	Test-case	Inputs	Expected results	Post-conditions
1	Program has to be started	Wrote invalid input to see if the program would respond.	"9,5" "fdsfds"	Expected a error message	Program runs normally and gave "invalid input" warning
2	Has played a game	Finished a game and writing the name on scoreboard.	"@åæø'1+=)(!$"	Expected to accept name and run smoothly. + allows UTF-8 characters	Functions normally.
3	5 different people have played and have written their names in the ranking table	Tried to add 10 people in the ranking system, so that those 10 can compete/compare their score.	"Ole" "Elena" "Iren" "Felix" "Ellen" "Per" "Lisa" "Emma" "Celine" "Mary"	Expected the system to be able to add all the players, and at least contain the top 10 best players.	Add all of the 5 players in a scoreboard, and function normal.
4	Program is running	Exiting program	"exit"	Expected the program to end	Program ended.
5	Empty scoreboard	Tried to write an x-coordinate first, which was a higher number than the y-coordinates	"8 ,2"	Expected to run normally, because the normal to write a coordinate is (x,y)	gave "Invalid Input" warning and program runs normally.
6	Played a round	Tested the indentation on the scoreboard	"Barry" "Oliver"	Expected to write the name on the scoreboard under "score" label.	Registers the name and functions normally
7	There are similar scores in the bottom half of the ranking table	What would happen if someone gets a high score and the last and second last have the same score, who would get thrown out?	Plays a round and gets into first place.	Expected the program to do something about it, like asking the player for a opinion.	Youngest player gets removed and the program continues.
8	The program must be running	Is it possible to get anything else than "Invalid input", when you do something wrong?	Writes gibberish a couple of times	Expecting the program to always respond to incorrect input by printing "Invalid input"	The program printed out "Invalid output".

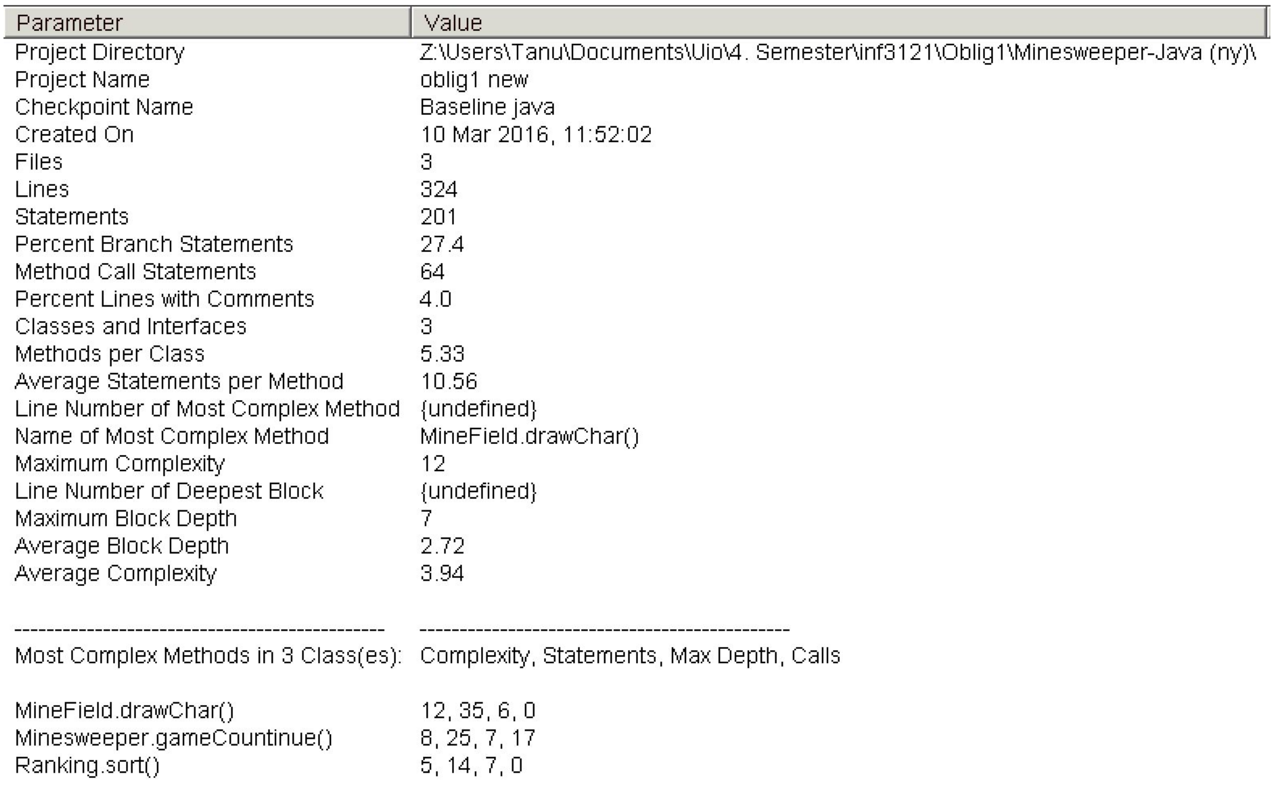
Johanpar
Birashat
Tanusanr

Requirement 2

The metrics we will look at in the Metrics Summary of the checkpoint will be listed below with an brief description.

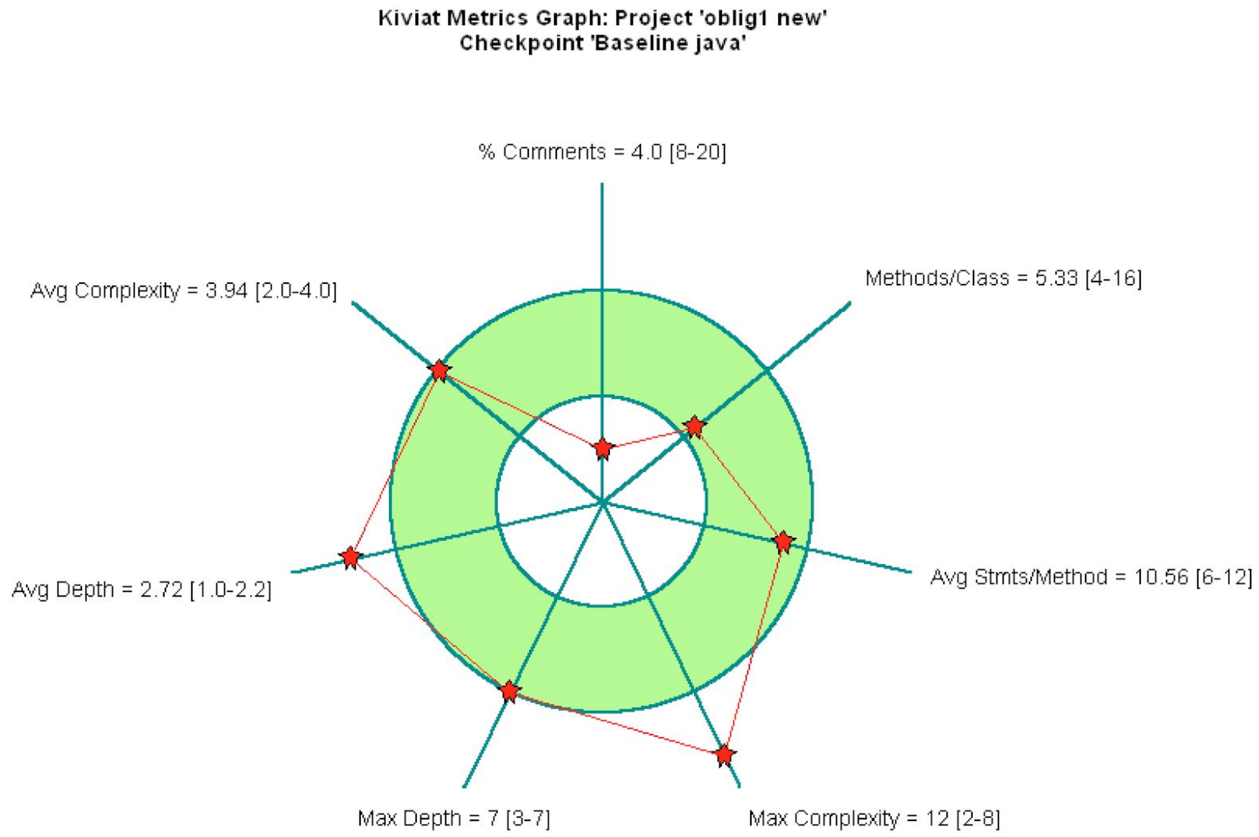
Project Directory - The path of the file (where the file/files can be found)
Project name - Name of the project, in our case “Oblig1”
Checkpoint name - Is the name you have chosen for this checkpoint which contain the metrics, so that when you make another checkpoint later, you will know the difference between the checkpoints.
Created On - When the file was created.
Files - Number of files the project contains.
Lines - Amount of lines in total.
Statements - Number of statements in total.
Percent Branch Statements - Percentage of branched statement, example given an *if*-statement, have both true and false branches. This also applies to else, do, for, while, continue, break, default, case, switch and etc. Generally speaking, whatever that can change the program to another direction which can give another output/exit of the program. Which would give the percentage of the program whom can be either one case or another case.
Method call Statements - The number of methods that have been called in total in the file/files.
Percent Lines with Comments - The percentage of the program which is commented.
Classes and Interfaces - Amount of classes and interfaces.
Methods per Class - The number of methods in the given classes/interfaces.
Average Statements per Method - The average amount of statements in the class/classes, which is the total amount of statements divided by methods.
Line Number of Most Complex Method - Self explanatory, undefined in our case.
Name of Most Complex Method - Self explanatory, in our case MineField.drawChar()
Maximum Complexity - The complexity value of the most complex method in a file. Should be within 2-10 according to the SourceMonitor.
Line Number of Deepest Block - Self explanatory, undefined in our case.
Maximum Block Depth - The maximum amount of bundled code(if, for, while and etc) in our case the maximum is 7 and the number of statements is 6.
Average Block Depth - The average block depth shows what the program usually revolves around when it comes to a problem(the amount of bundled code). Can be low or high depending on the difficulty of the problems in the program.
Average Complexity - The average complexity is computed by running the program a set of times, and then calculating average complexity. Which is the effectivity of the given program/code.

Johanpar
Birashat
Tanusanr



Parameter	Value
Project Directory	Z:\Users\Tanu\Documents\Uio\4. Semester\inf3121\Oblig1\Minesweeper-Java (ny)\
Project Name	oblig1 new
Checkpoint Name	Baseline java
Created On	10 Mar 2016, 11:52:02
Files	3
Lines	324
Statements	201
Percent Branch Statements	27.4
Method Call Statements	64
Percent Lines with Comments	4.0
Classes and Interfaces	3
Methods per Class	5.33
Average Statements per Method	10.56
Line Number of Most Complex Method	{undefined}
Name of Most Complex Method	MineField.drawChar()
Maximum Complexity	12
Line Number of Deepest Block	{undefined}
Maximum Block Depth	7
Average Block Depth	2.72
Average Complexity	3.94
--	--
Most Complex Methods in 3 Class(es):	Complexity, Statements, Max Depth, Calls
MineField.drawChar()	12, 35, 6, 0
Minesweeper.gameCountinue()	8, 25, 7, 17
Ranking.sort()	5, 14, 7, 0

(Figure 1)



(Figure 2)

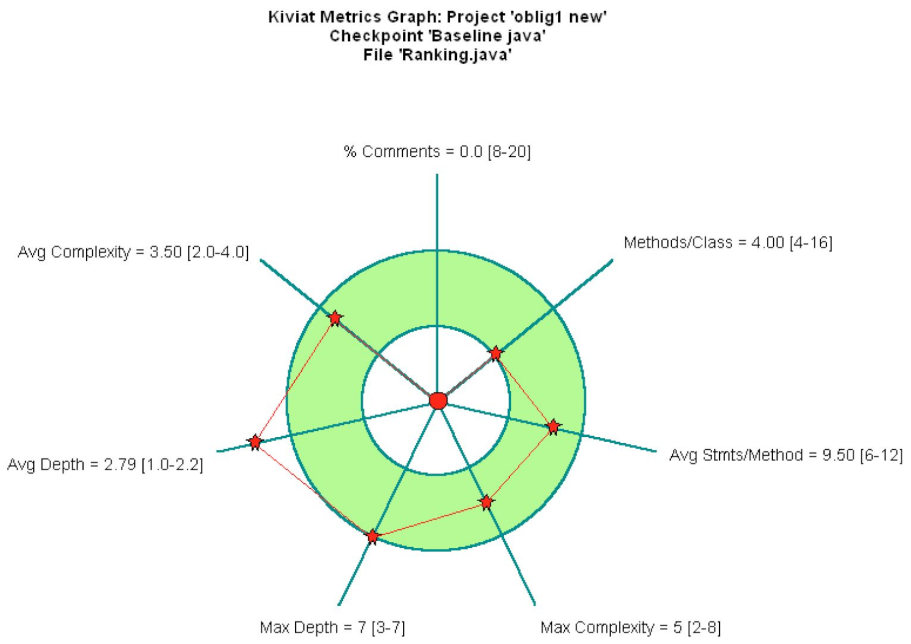
Johanpar
Birashat
Tanusanr

Project level

Looking at the kiviat graph, the comments should be between 8-20 percent of the code, however the percentage is 4 in our case. This if of course not a big problem, because the amount of commenting does not reflect the actual essence of coding, in other words quality is better than quantity. Method/Class has a limit of 4-16 and our program gave a 5,33 on this scale, which is within the limit. The max depth is exactly within the kiviat graph, which has a limit from 3-7. Our program scored a 7 on this metric. Average statements/methods are 10.56, which is a little bit high, however within a substantial range. The max complexity range should be between 2-8, however ends up being 12, which shows potential for improvement.

By the metrics, we can see that we are on the low on the comments and it is stretching it on the complexity mark. We could easily fix the comments section of the kiviat graph, by adding some lines of comments. But to be able to change the complexity of the code, would require the developer to change the code. Other than that, the developer should focus on the depth, with that we mean code wise how many statements that are built up on each other which results in a bundled code. Which is a far more intricate change than the comments, but that would result in a really good improvement to the program.

Minefield.java is the biggest file with a total of 169 lines of code. The file with the most branches is Minefield, which has a percentage of branch statements of 34.9, this shows how many different possibilities there are in Minefield. The file with the most complex code is also Minefield, this is because it does a lot of different things, contrary to the other files whom simply does what their class name implies. The metrics used to answer these questions are the metric summary of baseline(a combined summary of the three files) and the metric summary of Minefield.



(Figure 3)

File level

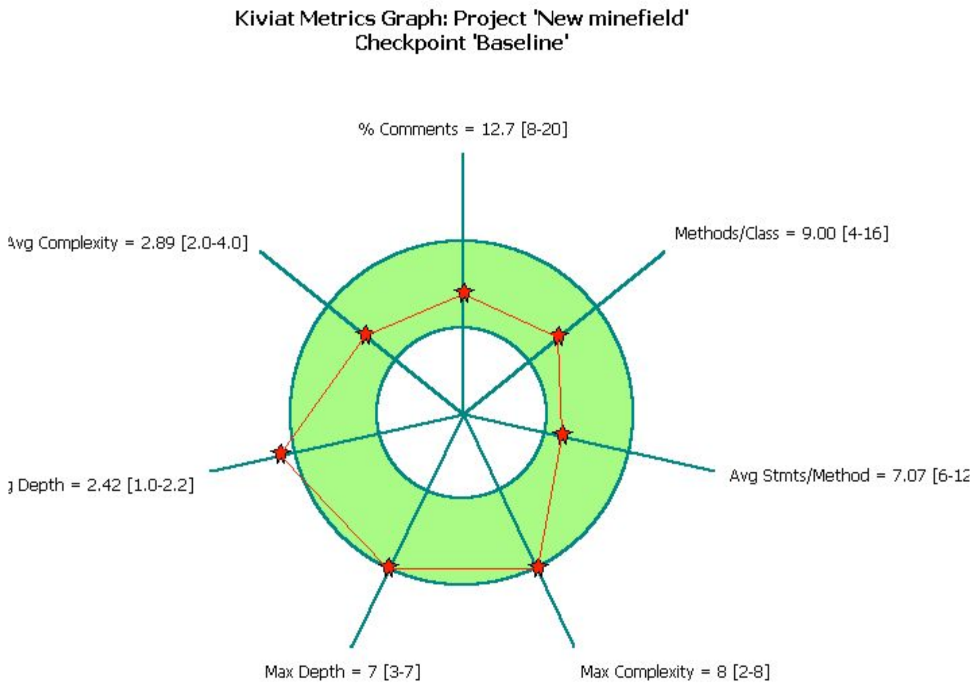
Johanpar
Birashat
Tanusanr

The metrics applied in our file are all the metrics on figure 3, the file we choose was Ranking. How we interpreted comments is that our file did not have any commenting since the number was 0 %. Methods/class basically means how many classes or methods the file contains, in our case the file contained 4 methods/constructors/classes in total. Average statements per methods is 10,56, which would mean we have a total of 10,56 statements inside of the methods found in our file divided by the number of methods we have. Max depth = 7, this means that there are a maximum 7 blocks of bundled code in our file. Average complexity is at 3,50 which would mean that our file has an overall complexity of 3,50. Average depth is off the chart with 2,79, this means the average nested blocks have a depth of 2,79. Most of the metrics are within the limit and are mostly equal to the metrics on project level. The only metric that have a large difference from the file we chose and the whole project is max complexity, which means the method with most complexity. On file level the complexity is 5(*sort*-method), and on project level it is 12(MineField.drawChar-method). In other words, this is the least complex file in the project. The rest of the metrics on file level, shows in essence the same values.

If we would have changed anything, we would have refactored the “sort” method by for instance splitting it up to smaller methods. Code wise this is the heaviest part of the file, however this doesn’t necessary mean that this is a complicated code, but compared to the other parts of the file it might be.

Requirement 3

The metrics that we would need to improve on project level are the max complexity, average block depth and comments. In order to do this, we have to take a look at the code and see where the most complex methods are. The most complex method in overall is the Minefield.drawChart() with an complexity of 12. The other files has a complexity that’s within the boundary. There are two methods with an average block depth of 7, Minesweeper.gameCountinue() and Ranking.sort(). Commenting is lacking in both Minesweeper and Ranking. The complexity of Minesweeper is way over the estimated boundary, and in addition to the complexity the average depth is also very high.



(Figure 4, after picture)

We have now changed the code accordingly to the things we were thinking of in requirement 2, with the change we will now see the difference. To see the difference compare Figure 1 against Figure 4.

Johanpar
Birashat
Tanusanr

What we have changed for now is the comment part, which has improved from 4.0% to 12,7**%**. Which takes it to the acceptable part of the kiviat graph which is 8-20%. We also made the max complexity decreased from 12 to 8. Average depth decreased from 2,7 to 2,42. At file level only one of the files got lower max complexity which was minefield, that went from 12 to 7. The Minesweeper.java file increased from 15,9 % to 17,6**%** in commenting and from 2,73 to 2,40 in average depth. The Ranking.java file improved from 0 to 14,1% in commenting and from 2,79 to 2,26. Minefield.java enhanced from 0 to 9,0% in commenting and from 2,69 to 2,50 in average depth. This shows that small improvements can heighten the kiviat graph quite easily. In general, the best way to solve this problem is to either change the data structure, to a more efficient one or split up the methods, to smaller methods. So far we have improved the kiviat graph by a good margin, but it still don't look quite perfect. So the developer has to evaluate code over if it is worth the effort to fix the given problems more, or to let it be because the improvement won't be so much better or the time it would take would not make it worth it.

A thing the developer could have done is to have commented better, in the case of all the files there were no comments that described how the code was going to work. With some use of regular comments, javadoc or etc the developer could have easily made comments which shows the progression of the program. Which would help rewriting the parts of code making it inefficient.

Github link: https://github.com/brish100/Assignment1.git