INF3121 – Project Assignment 1

Interpret the metrics offered by the static analyser

Requirement 1
The assignment we chose to work on was the second one, Minesweeper-Java, as Java is the language we feel most comfortable around when it comes to coding. The Minesweeper program is a basic console printing application that lets you play the classic game Minesweeper. We are using Atom/Sublime to modify the project.

Testing the functionality of the program. We are using test cases to specify what parts of the program we want to test. This is to ensure that the program works as specified in the requirements of the application (we assume that it is supposed to be flawless and work just like any other minesweeper game).

#	Test-case	input	Expected result	Actual result
1	Testing integers outside the row/column domain to see that it does not accept such values	5 10 -1 -1 -1 20	"Invalid input" "Invalid input" "Invalid input"	"Invalid input" "Invalid input" "Invalid input"
2	Testing integers for slots that you've already revealed, to see that revealed slots cannot be replaced.	0 0 0 0	Correct or boom/end "You stepped in already revealed area!"	Correct "You stepped in allready revealed area!"
3	Make sure that "Invalid input" does not count as a turn.	Enter 4 5 4 5 4 5	Turn = 0 Turn = 1 Turn = 1 Turn = 1	Pressed enter without any input multiple times, this did not affect the turn integer Tried input row 4 column 5 several times, this did not affect the turn integer
4	Make sure that when you type "exit" or "restart" with 0 turns, the program will exit without registering your name and score as this is useless.	Exit Restart	End Restart	The program registered our score even though we put no effort into obtaining a score when exiting/restarting, this should be changed in the code
5	Testing "top"-command with no results	Top	"Still no results"	"Still no results"

6	Does the program end correctly when you have finished the board?	All slots	"Congratulations you WON the game!"	//Commented out the function "boom();" at line 145 in MineField.java, allowing us to finish the board without it ending from a boom :) We won the game FAIRLY and it showed with the correct score of 35. Result: "Congratulations you WON the game!"

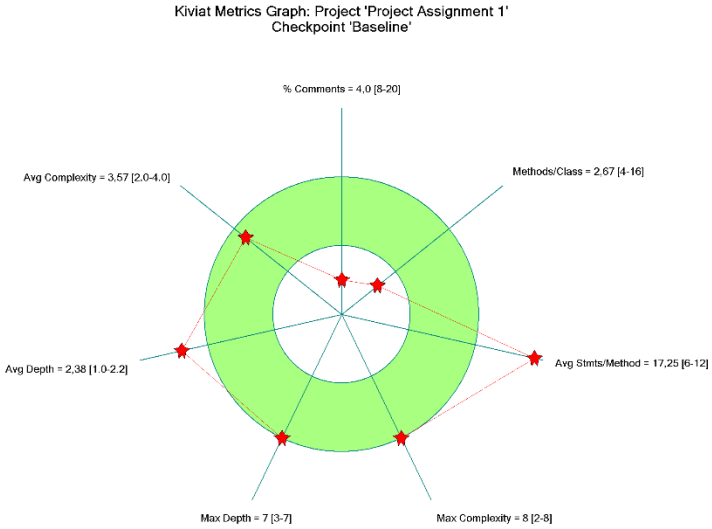
It would make sense to run some non-functional tests on the source code. A load test by spamming the enter key or running the program repeatedly could reveal problems with bottlenecks or too high CPU usage. Another test is maintainability testing, for example changing the program too having more rows, columns or mines. The column numbers are hard-coded when printed, and fixing the printing causes the column-line to look ugly

Our previous knowledge is miniscule and therefore we struggled to come up with ideas for testing in this case.

Requirement 2

Metrics at project level analysis

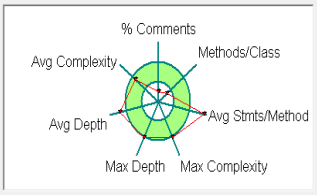
Brief description of the metric and values presented to us



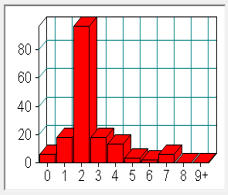
Metrics Summary For Checkpoint 'Baseline'

Parameter	Value
Project Directory	C:\Users\kaimyk\Downloads\source-code-pa1\Source Code PA1\2.Minesweeper-Java\Minesweeper-Java\
Project Name	Project Assignment 1
Checkpoint Name	Baseline
Created On	7 Mar 2016, 14:23:42
Files	3
Lines	326
Statements	162
Percent Branch Statements	10,5
Method Call Statements	92
Percent Lines with Comments	4,0
Classes and Interfaces	3
Methods per Class	2,67
Average Statements per Method	17,25
Line Number of Most Complex Method	{undefined}
Name of Most Complex Method	Minesweeper.gameCountinue()
Maximum Complexity	8
Line Number of Deepest Block	{undefined}
Maximum Block Depth	7
Average Block Depth	2,38
Average Complexity	3,57
---	---
Most Complex Methods in 2 Class(es):	Complexity, Statements, Max Depth, Calls
Minesweeper.gameCountinue()	8, 25, 7, 17
Ranking.sort()	5, 14, 7, 0
---	---
Block Depth	Statements
0	6
1	18
2	96
3	18
4	13
5	3
6	2
7	6
8	0
9+	0
---	---

Kiviat Graph:



Block Histogram (statements vs. depth):



Help

Copy

Print

Done

Project directory: What folder the project is located in
Project name: name of the project being summarized
Checkpoint name: what iteration of the project you're currently summarizing, currently the first
Created on: the date the project analysis of this iteration was created, march 9th 2016
Files: number of files included in the summary, three .class files
Lines: total number of lines across all the included files, 326 in total
Statements: the total number of actions that are called in the program, 162 in total
Percent branch statements: statements in the code that break the currently running sequential code, such as if, else, break, return etc. only 10.5%
Method call statements: the total number of method calls, 92 in this project, should try to cut down on the statements in each method
Percent lines with comments: percentage of commented lines, 4% seems very low, and this includes code in the comments. Will be changed in all files
Classes and interfaces: total number of classes/interfaces made in the project, 3 is fine for this project
Methods per class: average number of methods per class, 2.67 per class, can easily be pumped up if we cut down on statements per method
Average statements per method: the average complexity value of all the methods in the project, too high as described above
Line number of the most complex method: undefined because of multiple files
Name of the most complex method: gameContinue in Minesweeper handles most aspects of the game.
Maximum complexity: it seems gameContinue has the highest complexity
Line number of deepest block: undefined because of multiple files
Maximum block depth: Depth of the most complex method. This is found in gameContinue, with a value of 7.
Average block depth: depth of the average method. 2.38 on a project level. Kiviat graph indicates that this number is slightly too high.
Average complexity: the average complexity value of all the methods in the project. 3.57 on a project level. This is within the green zone on the kiviat graph, which is good.

Most complex methods
In our case, that is gameContinue in Minesweeper and sort in Ranking
Their values are the following:

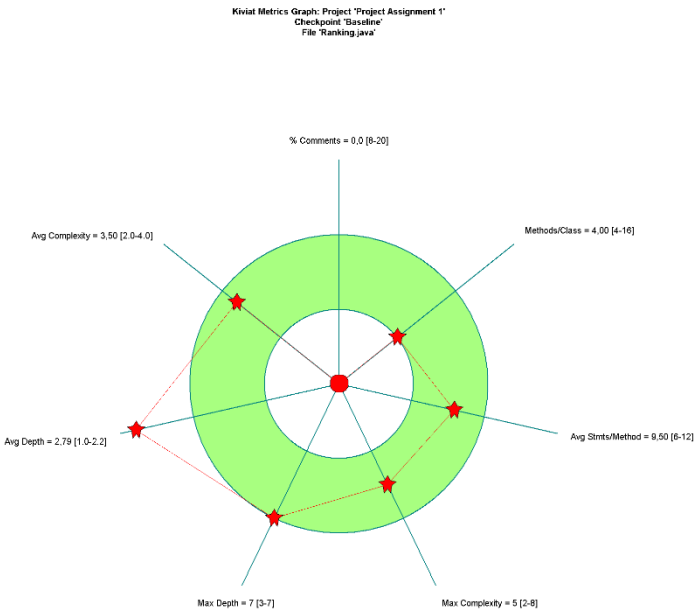
	gameContinue	Sort
Complexity	8	5
Statements	25	14
Max depth	7	7
Calls	17	0

As seen here, gameContinue is more complex as it has more statements and calls during its execution. Sort in Ranking.java will be looked at during the file level analysis.
Last is an overview of the number of statements a specific block depth has. This will be useful for when we are tweaking the code.

Remaining pinpoints from the second requirement:
The file containing the most lines of code is Minefield.java.
The file that contains the most branches percentage wise is Ranking.java, though it is close to Minesweeper.java, with 18.2.

Therefore, knowing that Minesweeper has more lines of code than Ranking, I am willing to bet that number wise, Minesweeper has more branches in total.
The file with the most complex code is Minesweeper. We base this on the fact that this file contains the most complex method, gameContinue

Metrics at file level analysis



This part of the analysis looks at standalone files. These metrics are different from the project analysis, because here we do not compare metrics across files. This is useful to find files that contribute with non-satisfying metrics, metrics we want to decrease or increase. In the particular file we are looking at, Ranking.java, we see that the average depth is outside the green circle. While this does not automatically make certain methods in Ranking.java too complex, we will definitely look at it. We will also take into consideration the low percentage of comments, and improve on this in the file that we pick for the last requirement.

Requirement 3

Identify at project level the metrics that you need to improve:

The comments in the project were poor, as shown in the kiviat graph. We added comments to the code throughout the whole project, and the kiviat graph looked a lot better. Taking the minesweeper class as an example, all methods where commented but in the gameContinue method also the if-sentences were commented to give clarification for what they did.

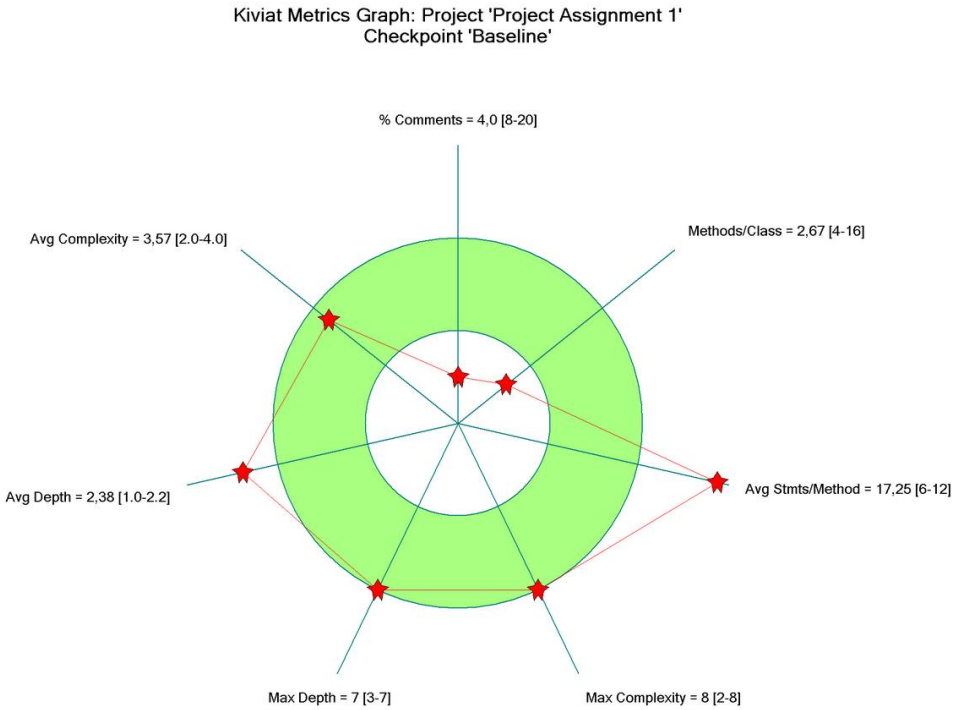
Average Statements per method was too high, and we fixed it by changing the Ranking class, more specifically the sort method in this class. We changed the whole sorting method too, using the Collections.sort() method and Comparator. We also made a new class Record in Ranking.java to save the top five rankings. Average Depth was a bit high, but the value looked significantly better after changing Ranking.

Comparisons of values in the kiviat graph for Ranking.java:

	Value before	Value after
% comments	0	17.6(green zone)
Avg. complexity	3.5(outer green zone)	2.17(inner green zone)
Avg. depth	2.79(far outside green zone)	1.9(outer green zone)
Max depth	7(outer green zone)	5(mid green zone)
Max complexity	5	5
Avg. stmts/method	9.5(green zone)	4.43
Methods/class	4(inner green zone	3.5(inner white circle)

Here is a comparison of the kiviat graphs before and after modifying the project:

Before:



After: