# INF4121 Assignment 1

Group: Øyvind Julsrud, Olav W. Eide, Torbjørn N. Høiland

## Requirement 1:

The program is a text based minesweeper game written in Java. The generated board is 10 characters long (horizontal) and 5 characters down (vertical) in size. When started the program gives the user some useful commands: restart, exit and top. These commands will start a new game, quit the game and reveal the top five ranked players, respectively. The program will read input, checking for the previously mentioned commands before determining if the user has entered a legal move. A legal move is on the format "row col" where row ranges from 0-9 and col from 0-4. When a legal move is entered the program reveals the "hidden" character as either a number (from 0-8, minesweeper style) or a bomb, portrayed as '*'. In the case of the bomb the game requests the user's name and then it resets.

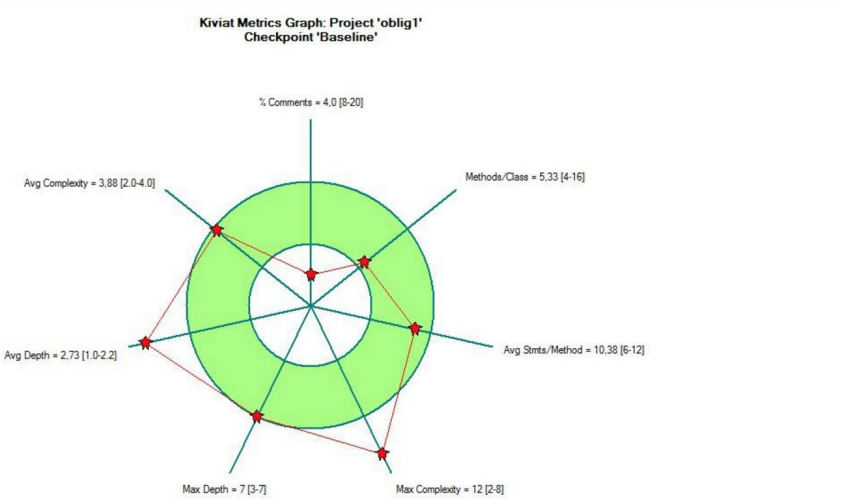
**MineField** - This class is used for generating, drawing and showing the board on which the user is playing minesweeper. It also has methods for checking if the input (rows and cols) are legal and then responds either true or false.
**Ranking** - This class is used to save the user's name and his score. The ranking class stores up to 5 names with their values and when there are more than one name it will sort this, from highest to lowest, based on score.
**MineSweeper** - This is the main class, which creates objects of the above classes. It initializes the game and gives the user some helpful commands (restart, top and exit. See program description).

**Manual test design**
Due to the size of the project we decided to follow an exploratory based fault attack, i.e. to try to make the program crash with input of highly likely user values, and test that the given commands behave as specified, note that these values can also be error prone values, but still highly likely by a user to input. This approach, can also to some extent be seen as a equivalence partition based testing, especially when we test the input of coordinates.
We decided that the areas to be tested are those that feature user input. Following are the commands top, restart, exit and input of coordinates. We define the manual tests by the idea

that they should work as expected when the commands are executed correctly, if not the program should give a suitable error message and let the user continue to give input.
Top, restart and exit should execute as specified when executed when the program is at: *“Please enter your move(row col): “ .* They should give an error message if they are executed with a typo, and not execute other places in the programs, such as when adding a name to the ranking list. Input of coordinates should edit the minefield only when the input is a valid row(0-4) followed by the space character and a column(0-9). If a coordinate is already added, one or both of the coordinates are out of bounds, or the command contains a typo, then the program should give a suitable error message. We have also designed some tests to test that the ranking system works as expected. We expect the ranking to work as follows: we can enter any name in the result list, the result list is sorted with highest results at the top, if a game had worse result than those already listed, it could discard our result.
Our previous knowledge helped us to identify error-prone input that may cause the program to crash.
We chose *not* to do non-functional testing on this project, due to the small scale of the program. Also, we saw no good way to improve the usability of the program without rewriting it with a GUI of some sort.

| Test case | Pre-conditions | Input data: | Post-conditions | Expected results | Actual results |
|---|---|---|---|---|---|
| start game | game executed from command prompt | | program at: Please enter your move(row col): | Give info about how to play game and commands to enter, print mine field | To play just input some coordinates and try not to step ont mine :)<br>Usefull commands:<br>restart- Starts a new game.<br>exit- Quits the game.<br>top- Reveals the top scoreboard.<br>Have Fun !<br><br>0 1 2 3 4 5 6 7 8 9<br>0 \| ? ? ? ? ? ? ? ? ? ? \|<br>1 \| ? ? ? ? ? ? ? ? ? ? \|<br>2 \| ? ? ? ? ? ? ? ? ? ? \|<br>3 \| ? ? ? ? ? ? ? ? ? ? \|<br>4 \| ? ? ? ? ? ? ? ? ? ? \| |
| top-command | program at: Please enter your move(row col): (one game finished) | top | program at: Please enter your move(row col): | show ranking with score and names, sorted. | N Name result<br>1 olav 1 |
| top-append bogus | program at: Please enter your move(row col): (one game finished) | top1 asdf | program at: Please enter your move(row col): | Give an error mesage about unvalid input. | "Invalid Input!" |
| top-prepend bogus | program at: Please enter your move(row col): (one game finished) | asdftop | program at: Please enter your move(row col): | Give an error mesage about unvalid input. | "Invalid Input!" |
| top-as name on ranking | program at: Please enter your name - | top | | add top as a name on ranking and print ranking as expected, not executing top-command. | N Name result<br>1 top 0 |
| top-no result yet | program at: Please enter your move(row col): (no game finshed yet) | top | program at: Please enter your move(row col): | Give an message of no results added yet. | "Still no results" |
| restart | program at: Please enter your move(row col): (already added coordinates to mine field) | restart | program at: Please enter your move(row col): | print a undiscovered mine field, as with start game. | need to fill in name for ranking, then prints a new mine field,as with start game |
| restart-append bogus | program at: Please enter your move(row col): (already added coordinates to mine field) | restart1 asdf | program at: Please enter your move(row col): | Give an error mesage about unvalid input. | "Invalid Input!" |
| restart-prepend bogus | program at: Please enter your move(row col): (already added coordinates to mine field) | asdfrestart | program at: Please enter your move(row col): | Give an error mesage about unvalid input. | "Invalid Input!" |
| restart-as name on ranking | program at: Please enter your name - | restart | | add restart as a name on ranking and print ranking as expected, not executing restart-command. | N Name result<br>1 olav 1<br>2 restart 1 |
| exit | program at: Please enter your move(row col): | exit | return to command line | exit game and return to command prompt. | need to fill in name for ranking, then exits with proper exit message |
| exit append bogus | program at: Please enter your move(row col): | exit1 asdf | program at: Please enter your move(row col): | Give an error mesage about unvalid input. | "Invalid Input!" |
| exit prepend bogus | program at: Please enter your move(row col): | asdfexit | program at: Please enter your move(row col): | Give an error mesage about unvalid input. | "Invalid Input!" |
| exit-as name on ranking | program at: Please enter your name - | exit | | add exit as a name on ranking andp print ranking as expected, not executing restart-command. | Please enter your name -exit<br>N Name result<br>1 exit 0 |
| row col | program at: Please enter your move(row col): | int(0-4) int(0-9) | program at: Please enter your move(row col): | show number of mines the area on mine field, give message if the user hit a mine or if the user won the game. | 0 1 2 3 4 5 6 7 8 9<br>0 \| * * - - - - * * * - \|<br>1 \| - - - * - - - - - * * \|<br>2 \| - - - * - - - - - - \|<br>3 \| - - - - * - - - - * \|<br>4 \| - * * - - - - * * - \|<br>Booooooooooooooooooooooooooooom!You stepped on a mine!You survived 0 turns<br>0 1 2 3 4 5 6 7 8 9<br>0 \| ? ? ? ? ? ? ? ? ? ? \|<br>1 \| ? 1 ? ? ? ? ? ? ? \|<br>2 \| ? ? ? ? ? ? ? ? ? ? \|<br>3 \| ? ? ? ? ? ? ? ? ? ? \|<br>4 \| ? ? ? ? ? ? ? ? ? ? \| |
| row col-col out of bounds | program at: Please enter your move(row col): | int(0-4) 10 | program at: Please enter your move(row col): | Give an error mesage about unvalid input. board remains unchanged. | "Invalid Input!" |
| row col-row out of bounds | program at: Please enter your move(row col): | 5 int(0-9) | program at: Please enter your move(row col): | Give an error mesage about unvalid input. board remains unchanged. | "Invalid Input!" |
| row col-no separator | program at: Please enter your move(row col): | int(0-4)int(0-9) | program at: Please enter your move(row col): | Give an error mesage about unvalid input. board remains unchanged. | "Invalid Input!" |
| row col-already added | program at: Please enter your move(row col): y=int(0-4) x=int(0-9) (already added) | y x (same as last) | program at: Please enter your move(row col): | Give an error mesage about already added coordinate. | "You stepped in allready revealed area!" |
| ranking- name and score added | program at: Please enter your name - | olav | program at: Please enter your move(row col): or same same as exit when exit is called. | Show ranking with score and name as added. | N Name result<br>1 olav 2 |
| ranking- long string with different special chars | program at: Please enter your name - | asdfasdf3434,.-@$½¥[[[[]]±± | program at: Please enter your move(row col): or same same as exit when exit is called. | Show ranking with score and name as added, could discard last parts of name if too long | N Name result<br>1 olav 2<br>2 asdfasdf3434,.-@$½¥[[[[]]±± 1 |
| ranking- sort | program at: Please enter your name - (already added one or more names) | olav | program at: Please enter your move(row col): or same same as exit when exit is called. | Show ranking sorted on results, with score and name added | N Name result<br>1 olav 2<br>2 asdfasdf3434,.-@$½¥[[[[]]±± 1<br>3 nils 1<br>4 top 0<br>5 per 0 |
| ranking discard | program at: Please enter your name - has done game with results less than the last entry on ranking. Has 5 results on ranking. | per | program at: Please enter your move(row col): or same same as exit when exit is called. | suitable error message of descarding the name and results since the list contains better results already. | "Sorry you cannot enter top 5 players" |
| win game | user enters several coordinates and does not hit a mine, after number of successful tries(NOTE: set number of mine aka counter2 =0 | int(0-4) int(0-9) | return to command line | Give suitable message to user | 0 1 2 3 4 5 6 7 8 9<br>0 \| 0 0 0 0 0 0 0 0 0 0 \|<br>1 \| 0 0 1 ? 1 ? 0 0 0 0 \|<br>2 \| 0 0 ? ? ? ? ? ? ? ? \|<br>3 \| 0 0 1 ? ? ? ? ? ? 0 \|<br>4 \| 0 0 0 0 0 0 0 0 0 0 \|<br>Please enter your move(row col): 3 8<br>Congratulations you WON the game! |

# Requirement 2:




Metrics Summary For Checkpoint 'Baseline'

| Parameter | Value |
|---|---|
| Files | 3 |
| Lines | 329 |
| Statements | 198 |
| Percent Branch Statements | 27,3 |
| Method Call Statements | 63 |
| Percent Lines with Comments | 4,0 |
| Classes and Interfaces | 3 |
| Methods per Class | 5,33 |
| Average Statements per Method | 10,38 |
| 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 | 2,73 |
| Average Complexity | 3,88 |

| 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 |

| Block Depth | Statements |
|---|---|
| 0 | 6 |
| 1 | 26 |
| 2 | 63 |
| 3 | 56 |
| 4 | 33 |
| 5 | 5 |
| 6 | 4 |
| 7 | 6 |
| 8 | 0 |
| 9+ | 0 |

A kiviat chart visually displays a set of metrics. The metric values are then placed inside a circle with a minimum threshold at the inner circle and a maximum threshold at the outer circle. The inner circle corresponds to the values below the given range of what is to be expected, values

outside the outer circle corresponds to values above the expected range, in both cases a refactoring might be needed.

**Metrics at project level**

**Statements:** Code terminated with a semicolon, if, for, while, attributes and exception control statements are also counted as statements. Total statements at project level gives you a general clue on how big your project is in terms of executable code.

**Percent branch statements:** In Source Monitor, statements that cause a break in the sequential execution is counted separately. A branch is the outcome of a decision, i.e if, else, do, while etc.

**Method call statements:** A total count of how many method calls there is in our project

**Percent lines with comments:** Comment lines are counted and compared to the total amount of lines. The percent lines with comments value for the whole project is 4. Based on the kiviat chart at project level, we can clearly see that this value is below the recommended range [8-20] which indicates that we might need to add more comments. For further investigation, we have to analyze the kiviat charts of every class to see if some classes are more exposed than others.

**Classes and interfaces:** The total amount of classes and interfaces in the project.

**Average statements per method:** The number of statements inside a method, divided by the total amount of methods in a file or a checkpoint. It gives you an indication of how big your average method is. In case of values outside the range, you might need to consider refactoring some methods. Our value was measured at 10,38 which is as expected [6-12]. Closer investigation revealed that the class MineSweeper had an value of 12.33, while the other two classes was inside the range. Refactoring methods in MineSweeper will might be necessary.

**Maximum method complexity:** Is measured by a complexity of one plus one for every branch statement in a method. Logical expressions are counted as well. According to Source Monitor, their complex metric is counted approximately as defined by Steve McConnell in his book Code Complete, Microsoft Press, 1993, p.395. Our maximum complexity values was measured at 12, which is above expected level, [2-8]. Further static analysis revealed that the class MineField had an Maximum complexity value at 12, while the other classes was inside the accepted range.

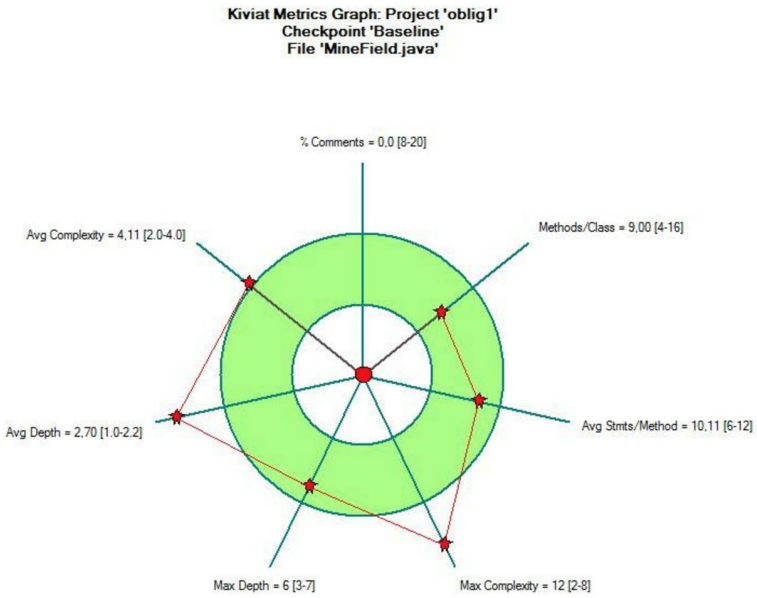
A refactoring of the method drawChar() in MineField will most likely be necessary due to its high complexity.

**Average complexity:** The average method complexity is an overall complexity measurement for each method, in this case, the whole project. The average complexity was measured at 3,88 in the range of [2.0 - 4.0]. This is as expected and changes to this metric will not be necessary, although refactoring of some methods might make the metric estimation to decrease.

**Maximum block depth:** At the start of each file the block level is zero, and then incremented with one according to how many block levels there is in the class. Depths up to 9 are recorded and all statements at deeper levels are counted as depth 9, this is indicated by the “9+” for the deepest level. Block depth gives us an insight on how complex the method is. More levels makes the code more difficult to read and can lead to errors in program logic, because with each new nested depth level, more conditions must be evaluated. If the method has too many levels you should consider splitting the method into several parts. Maximum block depth is the maximum depth level found in the file or the checkpoint. In this case we have analyzed the project level, resulting in finding the method with maximum depth level, which is gameCountinue() in the MineSweeper class, with a value at 7 in the range [2-8] which is a bit high, but in the acceptable range. A closer investigation of the method is needed.
**Average block depth:** Is the weighted average of the block depth of all the statements in the checkpoint. The average block depth in our project level is 2,73 in the range of [1.0-2.2]. That is above the expected level, which indicates that our code can be difficult to read, and it will be harder to find bugs in further dynamic testing later in the testing cycle, since the code is too complex and nested in too many levels.

The biggest file in our project is MineField with 165 lines. The file with the most branches is MineField, with a value of 34,9%. MineSweeper was the file with the most complex code, with an average complexity value of 3,67 and the maximum method complexity measured at 8 (MineSweeper.gameContinue()), which is also the most complex method among all the methods in the project.

**Metrics at file level**




Metrics Details For File 'MineField.java'

| Parameter | Value |
|---|---|
| Statements | 106 |
| Percent Branch Statements | 34,9 |
| Method Call Statements | 24 |
| Percent Lines with Comments | 0,0 |
| Classes and Interfaces | 1 |
| Methods per Class | 9,00 |
| Average Statements per Method | 10,11 |
| Line Number of Most Complex Method | 64 |
| Name of Most Complex Method | MineField.drawChar() |
| Maximum Complexity | 12 |
| Line Number of Deepest Block | 71 |
| Maximum Block Depth | 6 |
| Average Block Depth | 2,70 |
| Average Complexity | 4,11 |

| Most Complex Methods in 1 Class(es): | Complexity, Statements, Max Depth, Calls |
|---|---|
| MineField.boom() | 4, 6, 5, 1 |
| MineField.drawChar() | 12, 35, 6, 0 |
| MineField.getBoom() | 1, 1, 2, 0 |
| MineField.initMap() | 3, 4, 4, 0 |
| MineField.legalMoveString() | 5, 15, 4, 6 |
| MineField.legalMoveValue() | 4, 9, 3, 2 |
| MineField.MineField() | 2, 8, 4, 8 |
| MineField.show() | 3, 8, 4, 7 |
| MineField.trymove() | 3, 5, 3, 0 |

| Block Depth | Statements |
|---|---|
| 0 | 2 |
| 1 | 13 |
| 2 | 30 |
| 3 | 38 |
| 4 | 20 |
| 5 | 2 |
| 6 | 2 |
| 7 | 0 |

The most significant file from the project we chose is MineField.java, this choice is based on that MineField is the biggest file with most methods and carries the main functionality. In addition, it has the most complex method together with second highest depth of the project and the highest percentage of branches. As the kiviat graph suggests the file has a good number of statements per method and methods per class. Although,  this number could be higher with no problems. Possibly, some of the functionality could be factored out in own private methods. Which would lower the average statements per method, and increase the number of methods a bit.

Both average and maximum complexity and depth appears to be at noticable, if not alarming levels. We first consider the complexity of this class, and recall that the complexity is the number of branch statements, that is keywords in the code that causes branching, that is **if, else, for, while switch, default** etc. Most of the methods has a maximum of complexity 5, except *drawChar()* which has 12. We also notice that this method has the highest depth of 6, which is more than the double amount of the average depth of all methods in the class. If we could change this method and factor out some of the code to own methods it would greatly decrease the two depth-measures and the two complexity-measures.
By just looking at the method, most developers would think about factor out the work done in several methods. At first glance we notice the counting of mines, this two nested for-loops could be in a separate method, this also goes for the switch-case, maybe one could do this in a different fashion. We also notice a less effective structure of the if-sentences. More on the actual change on this class will follow in req. 3. Worth to mention is also the total absence of comments in the code.
Both the metrics for project level and for file level of MineField.java shows the same trends. Average depth and Maximum complexity are the metrics that are out of bounds. Comments as with the whole project is absent. The reason for these two metrics to be so equal could be the size of this project, which only has three files, and in total 329 lines of code.

## Requirement 3

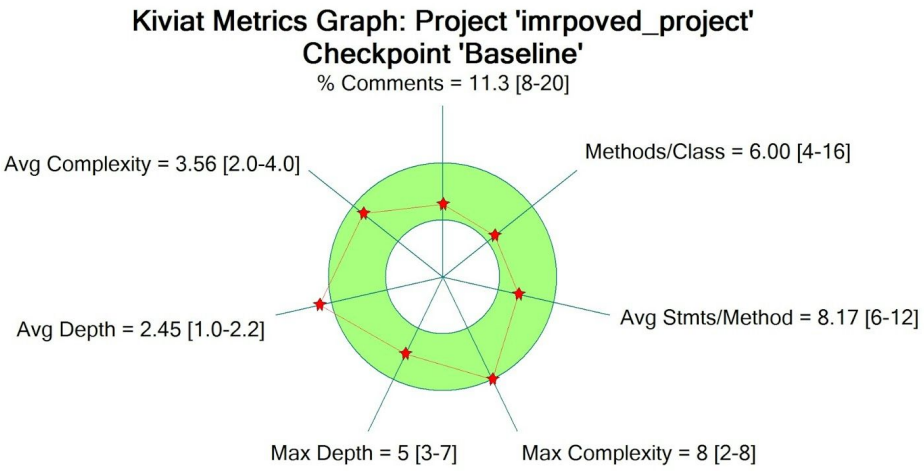
As you can see from the metrics at project level, under Requirement 2, we will need to improve the average depth of the methods and the complexity of some. The metrics also tells us that there are too few comments in the project.
We only did some minor improvements in the Minesweeper class, as in, we removed some excessive curly brackets, and removed the commented methods. We also moved the creation of a new scanner object outside the while loop in the gameContinue() method.

In the Ranking class we started by formatting the code, removing unnecessary whitespaces and out of order methods. In addition to formatting the code we also removed unnecessary nesting of statements which only caused the average block depth to grow, and made the code more difficult to read. If we were to further develop Minesweeper, a high average block depth could potentially cause bugs later on in the cycle. Unnecessary “return;” statements were also removed. A solid documentation of the class and methods were also added
In the Minefield class there were some complexity that needed to be improved. Here we in addition to code style improvement and added comments, refactored the *drawChar()* method by

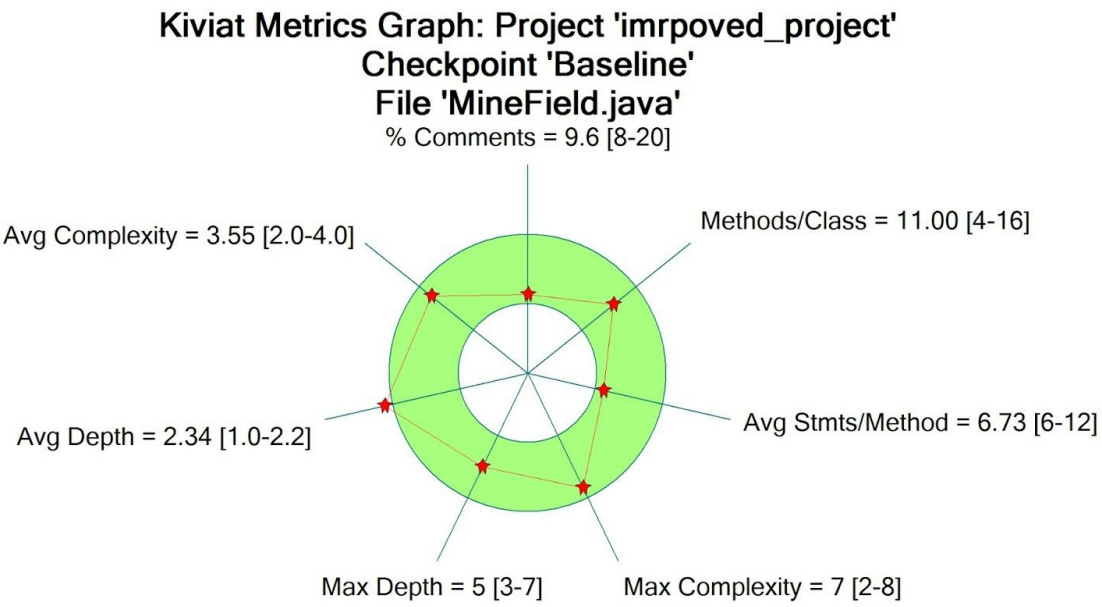
adding the counting of mines to an own method, and the translation from the counted mines to char to another method. We also changed the code of the translation to one if statement, instead of a switch-case sentence. Examples of that can be found in the code, but we have also included one here:

```
//changed switch to if. Prints the number of mines counted, else x
private char printNumberOfMines() {
	if(count>=0 && count<9) return Character.forDigit(count, 9);
	return 'X';
}
```

We again look at the kiviat graph at project level:




Compared to earlier we have added a whole lot of comments, we have also factored out some of the code into own methods, which has increased the number of methods a bit. Furthermore, we now have a maximum depth of 5, compared to an earlier 7. We still have quite a high average depth, this is because we have some methods like *countMines()* and *gameContinue(),* which both has quite high depth and pushes the average depth quite high. As a result of refactoring of methods we also see a fall in both measures of complexity, these are now within acceptable range. If we now turn to the MineField.java, the file we more thoroughly analysed in req 2. We examine the following metrics:




We now have a high level of methods per class and a decreased level of avg. stmts per method. Maximum depth is also dropped one level, together with avg. complexity and depth. Avg. depth is still quite high due to *countMines()* as mentioned above together with the show method that also carries quite high depth. Comments are now in around 10%, this could be considered low, but the class is quite self-explaining and the comments only serve as some extra information to guide of developers.

The code was relatively easy to maintain due to the small amount of code. However, the classes could be organized differently so that parameters such as size of MineField, number of mines etc. could be set in the constructor and the execution of the game could happen in an separate Main-class. This would make it easier in case the programmer wishes to expand the game by having different difficulties and sizes of the board.