Project 1

Aulon Mujaj, Henning Lund-Hanssen, Espen Jones

11th March 2016

1 Requirement 1 - Brief analysis

1.1 Brief description

This program is a simulation of the popular game Minesweeper. In this version of Minesweeper, the player is presented with a map of coordinates with questionmarks. Under each questionmark, there is either a number representing how many mines that are in the vicinity of this particular point on the map, or, a mine. The player is prompted for a set of coordinates, which should correlate to a questionmark on the map that the player does not think has a mine beneath it. The game goes on for as long as the player does not hit a questionmark with a mine beneath. The game ends when the player has identified all of the questionmarks without a mine, or if the player hits a questionmark with a mine.

1.2 Analysis

1.2.1 Defects

This program consists of three files.

- Minesweeper.java - The main class of this program. Calls on the two other parts, MineField and Ranking.
- MineField.java - Represents the map of the minefield
- Ranking.java - Handles the highscores of the players playing this game

1.2.2 Minesweeper.java

This main class on the program has the responsibility for the control flow of the program. When should the minefield be called to make a judgement about whether the game is still going or ended? When should the ranking be upon to calculate the score of the player and show the highscore? It taks input from the terminal and compares it to certain keywords. If the input from the player does not match the input criteria, the game just prompts for new input.

The testable parts in this class consist of handling the input from the user and that the right action is taken accordingly. For example, if the user gives

"top" as input to the program, the ranking class should be called to show the ranking of the players. And if restart is called, you would expect the program to restart your session.

1.2.3 MineField.java

The MineField class contains information about the current coordin

1.2.4 Ranking.java

1.3 Non-functional tests

Non-functional tests should always be tested. Though, this code looks like it was submitted in some introduction course in Java programming. If we look on this as a mandatory assignment in such a course, it should run on every machine at UiO installed with Java. If this were to be a huge game developed by a real company with the goal of making a multiplayer minesweeper with a regional highscore for each country and so on. The amount of non-functional requirements will increase.

We have decided to measure the non-functional requirements to meet what we can extract from the Minesweeper game on Windows 10.

1.3.1 Performance, load and stress

The program should not strain the computer to the point where Minesweeper prevents normal execution of other programs.

1.3.2 Reliability

If the program crashes, it does not save the state, so it is not recoverable after a crash. Though, the original game of Minesweeper does not do this either.

1.3.3 Usability

Playing the game requires the player to have previous knowledge of coordination systems. The instructions on how to play the game is lacking to some degree. It is a single line that instructs the user to give coordinates as input and to not step on a mine. This is forgiving to some degree, as most of us have previous experience with the Minesweeper game.

1.3.4 Efficiency

The program operates instantaneously both when it comes to asking for input and calculating the highscore. Also, the program as a whole does not utilize anything more than it needs to.

1.3.5 Maintainability

If changes is to be made to this system, the programmer has to know the Java programming language. If not, it as to be written from scratch in order for a change to be made. It is not imported or implemented a test library like JUnit for testing the different parts of the program, so it is not easy to just implement new tests without interrupting the data flow of the program.

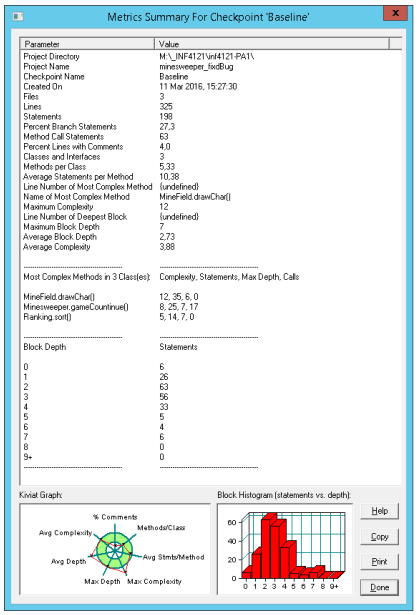
1.3.6 Portability

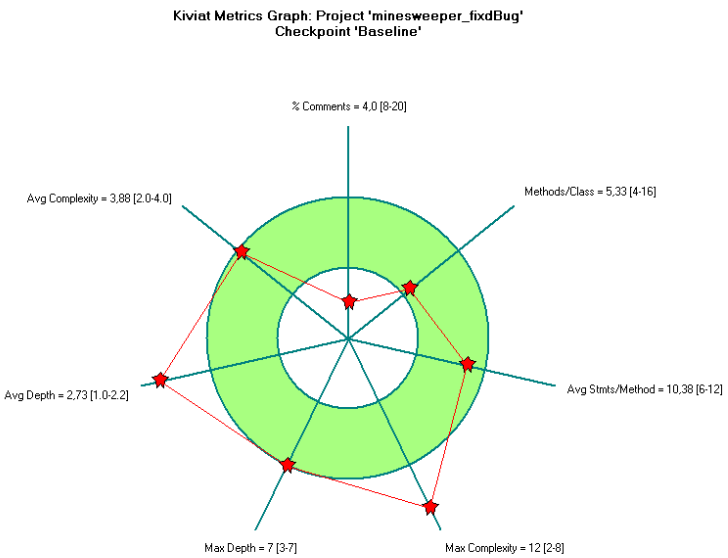
The only requirement for the program to be run is Java. The program can be run automatically with an IDE like IntelliJ or Eclipse. Without an IDE, you have to have some knowledge about the Java Platform. An alternative to just use the Java compiler and platform is to use a software project management system like Maven, Ant or Make. They will shorten and ease up the process of defining goals like compilation, testing and running the system standalone or for example with an application server like Tomcat or Jetty

1.4 Test cases

2 Requirement 2 - In-depth metrics

2.1 Metrics at project level





- What metrics do you spot for the whole project in the window Metrics Summary for Checkpoint? Write a brief description of the metrics. Try to explain their values (below what is expected, as expected or above the

expected level). What metrics do you think need to change?

- Which is the biggest file you have in your project by the number of lines of code?
  MineField.java
- Which is the file with most branches in your project?
  MineField.java
- Which is the file with most complex code? What metric did you choose to answer to this question?
  MineField.java

Files: This project that we are testing and analyzing has a total of three files

Lines: There are in total 325 lines of code throughout the three files.
Statements: Statements are terminated with a semicolon character in Java. Those lines of code are not the only ones that are statements, but branches suchs as if, for and while are also counted as statements. At the same time, exception control statemens try, catch and finally are also counted as statements, and throw statements. Our project that we chose has in total 162 statements.

Percent Branch statements: There are multiple statements that are regarded as branch statements. These are; if, else, for do, while, break, continue, switch, case and default, but we also count exception block statements such as try, catch, finally and throw as branch statements. The percent branch statements is how much percent of all the statements are branch statements in our project. This number in our project is 10.5% which means that out of 162 statements, 16 of those are branch statements

Method call statements: All calls are counted, both in statements and in logical expressions. Our porject has 92 method call statements.

Percent lines with comments: Off all the lines throughout the three files, the percentage that are comments is 4%. Which means that from 326 lines of code, 13 lines are comments. This isn't that much really, but if the code is easy to read and understand, that it does not really matter. If the code is unreadable with obscure variable names and complex logic, then it needs more comments.

Classes and Interfaces: SourceMonitor checks for the name "class <class name>" or "interface <interface name>" and extracts the class names. Both interfaces and classes are counted together. Our project has only three classes and no interfaces.

Methods per class: Total method count divided by the total class count. In our project this is 5.33 methods throughout three files.

Average Statements per Method: Total number of statements found inside of methods divided by methods found in the file. 10.38

Name of Most complex methods: The method that is the most complex one; MineField.drawChar()

Maximum Complexity: 12 Maximum Block Depth: Start of each file is level zero, each indent is one depth. The maximum depth here is 7.

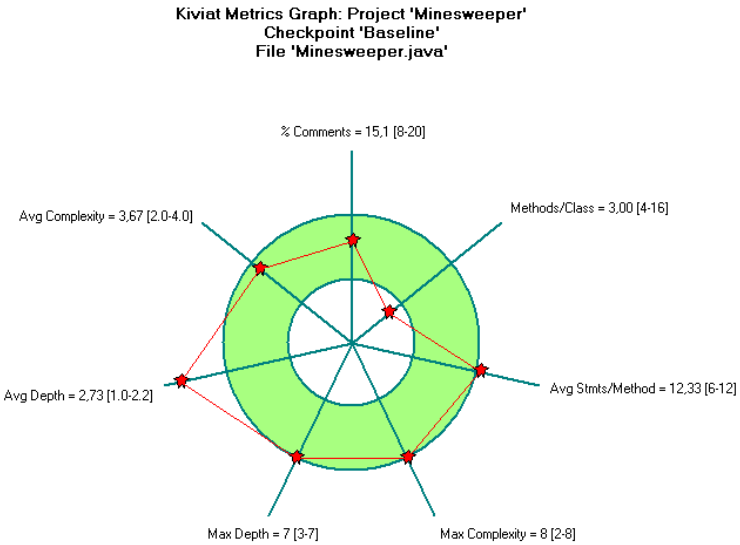
Average Block Depth: Weighted average of the block depth of all statements of the project. 2.73 is the average block depth in our project.

Average Complexity: Measure of the overall complexity for each method in a file or checkpoint. For our whole porject, the average complexity is 3.88

Most complext methods (complexity, statements, max depth, calls): Shows the most complex methods of the whole project, and shows the different numbers for complexity, statements, max depth and calls that it does.

Statements on each block depth: This shows how many statemetents each block depth has, and we can see from our Metrics Summary picture that there are 63 statements on block depth 2, then comes block depth 3 with 56 statements in total.

2.2 Metrics at file level



We decided to check the metrict of minesweeper.java file. Here we can see that the average complexity is at about 3.67, while the whole project is at 3.88. We can conclude from this that they are at the same point on the metric scale, but it is still a little high. What we would like to have is at about 2 on the scale.

Average depth is the same on both the whole project and in the specified file. The depth is too high and we should refactor it so that it goes down to about 1-2.2.

Max depth is at about 7, which we find is too high and we want refactor it to about 2-4, it is the same for the whole project. If the max depth starts to be higher than 4, then the complexity of the program goes up, and that becomes a problem when you want to debug or add more functionality to the code.

The whole project has 4% comments in total from all the lines, while minesweeper.java is at about 15%. This is ideal for the specific file, but not with the whole project, it should be at about the same as the specific file. On the other hand, if the code is easy to understand and does not have any complex code, then it shouldn't become a problem.

The methods/class is at 3, when it should be between 4 and 16, while the whole project is at 5.33. Some of the methods should be divided into smaller methods, to spread the different functions of the overall code. It will be easier to understand the code, then have everything in one method.

The average statements/method is why we should refactor the code into more methods, because then the average statements/method is lower. At the moment it is at about 12.33, while the project is at 10.38, and we would like to have it between 6 and 12.

The max complexity on the whole project is at 12, which is pretty high, and on our specific file the max complexity is 8, which is also high. We should refactor the code, so that the complexity is under 8, hopefully at 4.

- Would you refactor (re-write) any of the methods you have in this file?

3 Requirement 3 - Code improvement

3.1 Identification of metrics

3.2 Results from changes

3.3 Final remarks