Interpet the metrics offered by the static analyzer

Uy Quang Nguyen

Hangman - Java

INF3121 - Project Assignment 1

March 14, 2016

Contents

Abstract

For this project assignment, I've chosen a project that utilises Java. This is because I'm inexperienced with C++ and Csharp, with Java being my "strongest" side. I ended up with choosing Hangman, programmed in Java to analyse and test. PS. If something is too small to be read, just zoom in. This might be necessary when reading the tables or the pictures I've attached.

1 Introduction

For this project, I've decided to work with the Hangman, programmed in java project. I'm usually running OSX on my MacBook, so this means that I'll have to somehow run Source Monitor on my computer.

This assignment was done on / with

- MacBook Pro running Bootcamp (Windows 10 Pro, newest updates as of 03/03/16)
- i7 - 4750HQ @ 2.00ghz
- IDE: IntelliJ 15.0.4
- Java SE Development Kit 8u73
- Source Monitor, Version 3.5.0.306

I first started out with installing Windows with Bootcamp on my MacBook. After this, I updated Windows, and installed IntelliJ, together with JDK 8u73, and finally Source Monitor.

1.1 Background

The Java courses I've had on University Of Oslo:

- INF1000 - "Grunnkurs i objektorientert programmering", 2013 Autumn
- INF1010 - "Objektorientert programmering", 2014 Spring
- INF2220 - " Algoritmer og datastrukturer", 2014 Autumn

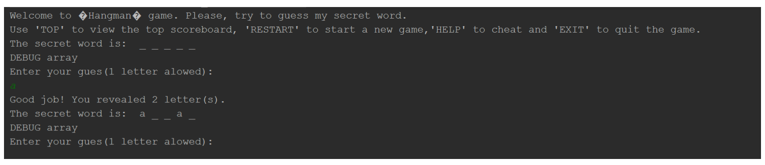
It's been over a year since I've used Java, so this'll be a very exciting assignment. I have no other experience with Java.

1.2 Motive

The objective of this task is to discover what our source code is supposed to do, and to manually test it and analyze it with a static analyzer. But why a static analyzer?
A static analyzer can be an invaluable tool. It can analyze a tool, which gives us metrics, which we can use to improve the performance, maintainability, and the readability of a software. There are very many static analyzer out there, but for this task, we'll use source monitor.

2 Requirement 1(5p)

Make a brief description of the program that you use for your project assignment.
Hangman is a guessing game with a text-based interface.The user can guess a word that is randomly chosen by the game. The word will be represented by a row of dashes, each of the dashes representing a letter of the word. The player can then enter a single letter, and if the letter is a substring of the chosen word, it'll replace the dash(es). The player wins when there are no more dashes. The program consists of 5 java files.



```
Welcome to �Hangman� game. Please, try to guess my secret word.
Use 'TOP' to view the top scoreboard, 'RESTART' to start a new game,'HELP' to cheat and 'EXIT' to quit the game.
The secret word is:  _ _ _ _ _
DEBUG array
Enter your gues(1 letter alowed):
a
Good job! You revealed 2 letter(s).
The secret word is:  a _ _ a _
DEBUG array
Enter your gues(1 letter alowed):
```

(a)

Figure 1: Application opened in IntelliJ (Debug enabled)

Make an analysis of the testable parts of the program that you are using for the project assignment. Design the manual tests that you would perform for this program, in order to test its functionality (system tests). Specify if you used a particular test design technique in designing your tests.

Table 1: Analysis of testable parts of software

Type	**Functionality**	**Description**
Functional	Input	Testable inputs: - Single letter - Multiple letters (More than one letter at once) - Special characters (æ,ø,å,@,0-9,tab, space, etc.)
Functional	Output	Testable output with these inputs: - Correct input - Incorrect input - Letter guessed / Not guessed - Already entered input - Special functions (Help, cheat & restart)
Functional	Functionality	- Will guessing all letters correctly end the game? - Does the application choose a different word when starting a new game?
Functional	Extra Functions	- Does the cheat function work properly? - Does the highscore function work properly? - Does the restart function work properly?
Non-Functional	Usability	- Is it easy to learn how to use the application? - Is it hard to remember how to operate the software?
Non-Functional	Maintainability	- Is it easy to understand the code of the application?

Above is an analysis I've made of the testable parts of the software. I've analyzed the several things you can test, such as input, output, functionality, functionality, extra functions, usability and maintainability. In input for a testable parts are what happens when you input a single letter, multiple letter etc

In table 2 in the next page, I've sat up a table, that consist of the manual tests that I'd perform for the program to test its functionality, together with a priority number, which tells how important they are, ranging from 1 (most important) to 9 (least important).

Table 2: Manual Design - Blackboxing + Experience

Test ID	Objective	Steps	Pre/Post Condition	Expected Results	Actual Results	Priority Number
1	"Test that guessing correct word grants victory"	1. Enter a letter 2. Repeat with different letters until all dashes are gone.	PRE 1. Application is opened POST 1. Victory is granted	1. When all dashes are gone victory should be granted	1. When all dashes are gone victory is granted.	1
2	"Test that invalid inputs dont crash the program"	1. Enter an invalid input (multiple letters, 0-9, special characters etc)	PRE 1. Application is opened POST 1. Application is still running	1. Application should ignore invalid inputs, and handle them without the application crashing.	1. Application handles invalid input, and prompts player to input another letter	2
3	"Test that the 'cheat' function helps the player"	1. Enter 'cheat' as input during game	PRE 1. Application is opened POST 1. A dash is revealed	1. Application should assist the player, and reveal a dash.	1. A dash is revealed, and is converted to a letter	5
4	"Test that the score is saved in highscore after game"	1. Enter a letter 2. Repeat with different letters until all dashes are gone. 3. Enter name that'll be entered into highscore.	PRE 1. Application is opened POST 1. Score and name is saved to file	1. When victory is granted, user should be prompted to type in his name. 2. Name should be saved, together with score in a file.	1. User is prompted to enter his name to save his score 2. A file is created, and stored in a folder.	6
5	"Test that the highscore shows when requesting"	1. Enter 'top' as input during game	PRE 1. Application is opened 2. Highscore file exists POST 1. Highscore is loaded from file	1. Application loads saved file. 2. Application shows contents of saved file as output.	1. Application crashes.	7
6	"Test that the restart function works"	1. Enter 'restart' as input during game	PRE 1. Application is opened POST 1. Application ends current game, and presents a new word covered by dashes	1. Application ends current game. 2. Application presents user with another word to guess.	1. Application ends current game. 2. Application presents user with another word to guess.	4
7	"Test that the game doesn't crash after 5 games"	1. Enter a letter 2. Repeat until all dashes are gone. 3. Repeat 5 times.	PRE 1. Application is opened POST 1. Application is still running	1. Application is still running.	1. Application is still running.	3

When designing these tests, I used two specific test design techniques. First one being **blackboxing**, and second being **experience** design technique. I chose blackboxing, because hangman is a game. It's supposed to work in a specific way, so the question is **does it work?**. So I tested the games functionality, without caring about how the code worked. I tested for example if the function "restart" restarted the game. Experience design technique was used, because me, as a programmer knows that invalid input, repeated inputs, etc, can crash a program if you've not set up handlers and exceptions for it. Users often do invalid inputs by mistake when playing games, so if the program didn't handle them correct, it'd basically crash.

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

Yes, based on what kind of game this is, I've chosen to write some non-functional tests which I believe are essential. Since a round of hangman usually goes very quick, it's important to expect that the player is going to be playing multiple rounds, which means that the stability should be tested. You can't for example have the game crashing after 5 games, because of a full array, or some other errors such as buffer overflow. Testing out the stability is important. Testing usability is also, because as a game, it should be easy to play and use.

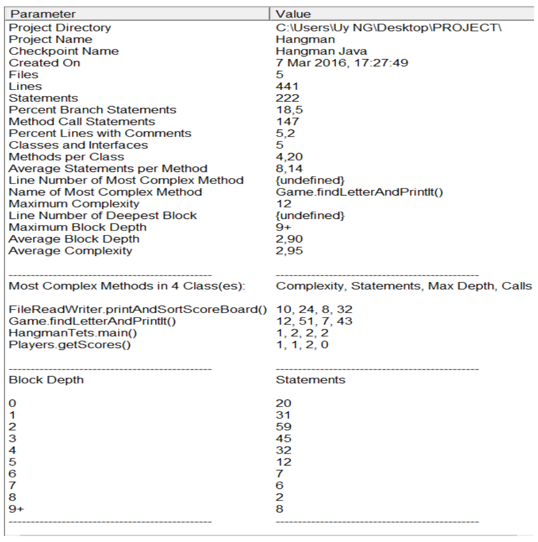
Write a list with the concrete test cases that you obtained, as a result of the analysis and design performed above. The list needs to contain: pre- and post- test conditions (if the case), the test itself, the expected result and the actual result. Order the list of tests based on the importance of the tests. Can you tell something about your how did your previous knowledge / experience helped you in creating this list with tests?

The list with concrete test cases and condition is in table 2 above, together with the importance order and results. My previous experience helped me with creating the test, because I've made tests based on small, easy, mistakes that you can make while programming (such as buffer overflow, not handling invalid inputs etc).

3 Requirement 2(10p)

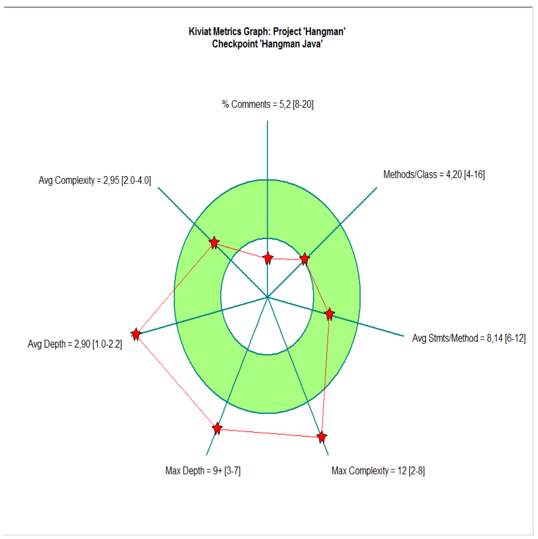
3.1 Metrics at project level

Run the static analyzer on the whole project to be analyzed
Take a screenshot of both the metrics summary and of the kiviat graph.



```
Parameter                                   Value
Project Directory                           C:\Users\Uy NG\Desktop\PROJECT\
Project Name                                Hangman
Checkpoint Name                             Hangman Java
Created On                                  7 Mar 2016, 17:27:49
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
--------------------------------------------  --------------------------------------------
```

(a) Metrics Summary from SourceMonitor



(b) Kiviat Graph from SourceMonitor

Figure 2: Metrics summary and Kiviat graph

Above are the metric summary we get when using SourceMonitor. Table 3 describes the metrics that are relevant to java programming.

Table 3: Metrics at project level

Metric	**Value**	**Kiviat Expected Threshold**	**Description**	**Thoughts**
Lines	441	N/A	The total number of physical lines in all the files that are in the project. In this case, 5 files. This includes blank lines.	No change needed here. Considering that it's a text based game, 441 looks perfectly fine. And also, there's no "expected" line of code value, because every program is different.
Statement	222	N/A	Statements are lines that are computational, and are terminated with a semicolon. If, for, and while loops together with exception control statements such as try are also statements.	No change needed here. The amount of statement that is acceptable for a code depends on how many methods these statements are divided between. Having the pure value of total statements doesn't tell us much. Again, there's no expected statement amount value.
Percent Branch Statement	18,5%	N/A	Branch statements are the following: If, else, for, do, while, break, continue,switch, case default, together with try, catch and finally. The percent branch statement is based of the amount of statements that are branch statements.	No change needed. It's a value that doesn't really tell much when doing static analysis. It's been mentioned that a lower percent branch statement is good because branches uses more computational power from the processor.
Method Call Statement	147	N/A	Amount of statements and logical expressions that contain method calls. This metric doesn't tell us much that is important for static analysis.	No change needed. The amount of method call statements depends on what kind of software it is, so there's no "exact" acceptable amount of method call statements.
Percent Lines with Comments	5,2%	8-20 %	Amount of lines that contain comments in the project, compared to the total amount of lines in the project. This metric is useful, because it tells us how much of the code is commented. A low number is bad, because that makes it hard to maintain and understand.	I'd definitely say that this needs to be increased. Average/Expected value should be around 8-20%. We can also confirm this by looking at the Kiviat Graph which shows that % comments is too low.
Classes and Interface	5	N/A	Total number of classes and interface combined.	No change needed here. Again, number depends on what kind of project it is, so there's no "expected" value here.
Methods per Class	4,20	4-16	The average amount of methods per class in the project. This metric is useful, it shows if a class is taking on more work than it should, and that can lead to classes that are too big.	According to the Kiviat graph, the amount of methods per class in the project is fine. The accepted boundaries in the graph is between 4-16 methods per class. The low number shows that the class is following good object oriented principles. No change needed here.
Average Statements per Method	8,14	6-12	The average amount of statement per method. This metric is important, because it tells us the average amount of statements per method. If the number is too high, then a method might be taking on more than it should, and the code should be split up into more methods.	According to the Kiviat graph, the average statements per method is within the expected boundaries. No change needed here.
Average Complexity	2,95	2-4	The complexity metric measures the number of execution paths through a function or method. Each function or method has a complexity of 1. This number is plussed with 1 for each branch statement, arithmetic, and logic statements. Switch statement however add count for each exit from a case, and catch/except in a try block each add one count too. The average complexity is a measure of the overall complexity measured for each method in the whole project (checkpoint).	No change needed here, it's within the expected threshold of the Kiviat graph.
Maximum Complexity	12	2-8	The maximum complexity is the complexity value of the most complex method in the project.	Definitely needs a change. The accepted boundaries in the Kiviat graph is 8, but the project's maximum complexity is at 12. There might be a single method that is the cause of this, because the average complexity is much lower.
Average Block Depth	2,90	1-2,2	This metric shows us the average nested block depth, weighted by depth. Because nested blocks are always introduced with branch statements , a code with high depth might be hard to read because of the conditions that are introduced. Average block depth is the weighted average of the block depth of all statements in the project.	The value is a little higher than the accepted boundaries in the Kiviat graph. This might mean that there's some unnecessary long nested statements.
Maximum Block Depth	9+(10)	3-7	This metric shows us the maximum nested block depth level found.	Definitely needs a change, as accepted boundaries are 3-7, and the project's at 9+. The actual value can be seen by enabling an option. The actual value is 10. The value is much higher than the average, so there might be a single file/method that is the cause of this.

Which is the biggest file you have in your project by the number of lines of code?
The file with the highest amount of lines of code is FileReadWriter.java
Which is the file with most branches in your project?
The file with the most branches is FileReadWriter.java. It has a statement count of 106, together with branch statement percent of 20,8.
Which is the file with most complex code? What metric did you choose to answer to this question?
The file with the most complex code is Game.java . Looking at the metrics it's easy to see why. It has the maximum method complexity of all other files, and the highest average complexity too.

3.2 Metrics at file level

Now, do the same with a significant file from your project (choose a file from the projects, select it, go to "View" menu and choose "Display file metrics Kiviat graph"). Take a screenshot of your Kiviat graph.

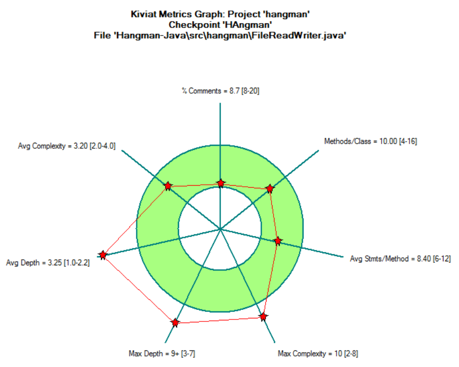


Figure 3: Kiviat Graph of FileReadWriter.java

How do you interpret the metrics applied on your file? How are they different the metrics you obtained on the whole project, compared with the metrics on this file?

Table 4: Comparison between prject and file metrics.

Metrics	**Project**	**FileReadWriter.java**	**Thoughts**
% Comments	5,2%	8,7%	FileReadWriter is actually within the accepted thresholds, which means that one of the other files in the project must have code that are less commented than it.
Methods/Class	4,20	10,0	Both are within the accepted thresholds of the Kiviat Graph, although it's worth mentioning that the FileReadWriter has much more methods per class than the average file. If it weren't for the FileReadWriter, the project metric would've been outside the accepted threshold.
Average Statements per method	8,14	8,40	Both are very similar, and they're also in the accepted thresholds. It's worth noting that FileReadWriter stands for the most of the methods, which means that it might be the file pushing this value up.
Average Complexity	2,95	3,20	Both are similar, and they're also in the accepted threshold. Not much to say here.
Average Block Depth	2,90	3,25	Both are similar, and they both have a value over the accepted threshold. FileReadWriter actually has a higher value, so reducing it's block depth will reduce the average block depth, and might bring us to the accepted treshold.
Max Complexity	12	10	Both are similar, and they both also have a value over the accepted threshold. The max complexity that is accepted is 2-8. Reducing the max complexity in FileReadWriter won't affect the max complexity in project, so we'll have to find the file with the max complexity and reduce it too, if we want our project to be within the accepted threshold.
Max Block Depth	9+(10)	9+(10)	Both are identical and above the accepted thresholds of the Kiviat Graph. There's no other file with as high block dept such as FileReadWriter. Which means it's the culprit, and reducing the block depth in FileReadWriter will bring the project down to accepted thresholds.

Would you refactor (re-write) any of the methods you have in this file?
Yes, there are some methods in FileReadWriter I'd definitely change.

- void nop()
  The first one is the method call nop, which is basically a method that contains many brackets containing the command "System.out.println(true)". This is one of the methods that are increasing our average depth metric. The most notable thing of all is that this method isn't used even once. Removing it completely takes our file metrics max depth to 8, and average depth to 2,82.

- void oldReadRecord()?
  The second one is the method oldReadRecord, which basically calls readRecord() five times in a row. The issue is that this method is not used anywhere either.

Removing these two methods does not affect the software in any way.

4 Requirement 3(10p)

Improve the code, based on the metrics you have obtained with this analyzer. Analyze the improved code using SourceMonitor again.

-Identify at project level the metrics that you need to improve
There are four metrics that we need to improve, based on the Kiviat Graph. We have to improve: Percent of comments, Average Depth, Max Depth, and Max Complexity. These are the four metrics that are outside of the accepted threshold in the graph. We might also have to adjust the method per class, because it's very close to being outside of the boundaries, which might happen when we fix the other metrics.

Provide in the delivery examples of improved/refactored code.

4.1 FileReadWriter.java

Earlier when analyzing FileReadWriter, I found out that it's the file responsible for the average depth increase, and the max block depth, so I'll try to focus on that. It's also responsible for the increase in average complexity. The purpose of this file is to read and save user scores to/from a file.

- void nop()
  Completely removed the void method nop(). It wasn't used any where, and it increased our total depth to 10, and our average depth. Removing it did not affect the software in anyway. Even if it was used, it'd just print system out print 'true' several times, making no sense. Removing this brought the average depth, max depth closer to the accepted threshold of the Kiviat Graph.

```java
        private void nop()
        {
                System.out.println(true);
                {
                        System.out.println(true);
                        {
                                System.out.println(true);
                                {
                                        System.out.println(true);
                                        {
                                        ...........
                                        }
                                }
                        }
                }
        }
```

- void oldReadRecord()?
  Completely removed the void method oldReadRecord(). It wasn't used anywhere either. Removing it gave us a lower Line of Code metric, thus increasing the percentage of comments.

```java
        private void oldReadRecords()
        {
                readRecords();
                readRecords();
                readRecords();
                readRecords();
                readRecords();
        }
```

- Unreachable code

```java
        boolean evaluate=false;//new Evaluator().Asses();
        if(evaluate){
                Players temp1;
                int n1 = myArr.size();
                for (int pass = 1; pass < n1; pass++) {
                                for (int i = 0; i < n1 - pass; i++) {
                                if (myArr.get(i).getScores() > myArr.get(i + 1).getScores()) {
                                                temp1 = myArr.get(i);
                                        {
                                                myArr.set(i, myArr.get(i + 1));
                                                {
                                                        myArr.set(i + 1, temp1);
                                                }
                                        }
                                }
                        }
                }
                        System.out.println("Scoreboard:");
                    for (int i = 0; i < myArr.size(); i++) {
                                System.out.printf("%d. %s ----> %d", i, myArr.get(i).getName(),
                                        myArr.get(i).getScores());
                }
        }
```

  This whole part is basically unreachable. The variable evaluate is declared to false the moment it's initialized, which means that everything within that if loop wont be reached. Not to mention all the unnecessary brackets.

- Unnecessary brackets and blank lines
  There are very many unecessary brackets throughout the code which I remove, as these contribute to increasing the average block depth without actually doing anything.

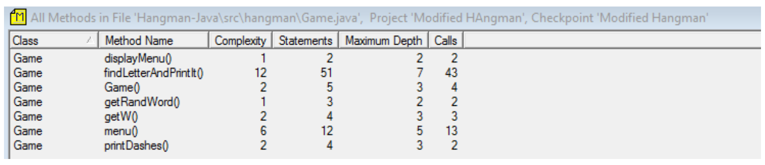
```java
 {
                myArr.set(i, myArr.get(i + 1));
                        {
                                myArr.set(i + 1, temp);
                        }
}
```

  Above is an example of one of these cases. I've also removed plenty of the blank lines.

I've done more changes to the file, but it'll be too much to list here, so check out github later.

4.2 Game.java

Although FileReadWriter.java was the cause of the high block depth, the Game.java on the other hand is the cause of the high maximum complextiy. Game.java handles the game while it's running. It's the object that fetches what word to guess, and keeps the game running with loops until the player guesses the worth. If we use SourceMonitor, we can see that it's method "findLetterAndPrintIt" is actually the cause of the high complexity.



All Methods in File 'Hangman-Java\src\hangman\Game.java', Project 'Modified HAngman', Checkpoint 'Modified Hangman'

Class	Method Name	Complexity	Statements	Maximum Depth	Calls
Game	displayMenu()	1	2	2	2
Game	findLetterAndPrintIt()	12	51	7	43
Game	Game()	2	5	3	4
Game	getRandWord()	1	3	2	2
Game	getW()	2	4	3	3
Game	menu()	6	12	5	13
Game	printDashes()	2	4	3	2

(a)

Figure 4: Source of high max complexity value

- Blank Lines and brackets
  I removed blank lines and unnecessary brackets.

- Comments
  Low amounts of comments, so I added some comments explaining the file. This increased the percentage of comments metrics.

- More methods
  Since the complexity was high, and the number of methods was low, I split up the findLetterANdPrintIt method into several smaller methods (MistakeChecker, CheatChecker, CheckIfCorrect,saveToFile and cheat). This reduced the maximum complexity, and average block depth into acceptable Kiviat thresholds. I also modified the menu method, and compressed the else and if's to else if, to further decrease average block depth.

4.3 HangmanTets.java

There wasn't much to modify in the HangmanTets file, considering it's just the main file which'll create the game objects consisting of only some few lines.

- Blank Lines
  I removed the some blank lines, and there were alot of them. Removing these can have an impact on the percentage of comments metrics, depending on if you chose to ignore blank linkes or not in SourceMonitor.
- Comments
  Absolutely no comments, so I added some comments explaining the file. This increased the percentage of comments metrics.
- Added method to start game
  I added a method to start the game, to make it more object oriented, and to increase the method/class metric, since it was very close to being too low.

4.4 Players.java

There wasn't much either to modify in players.java file. It's just a file that contains the class Players, which is used to construct player objects for the highscore.

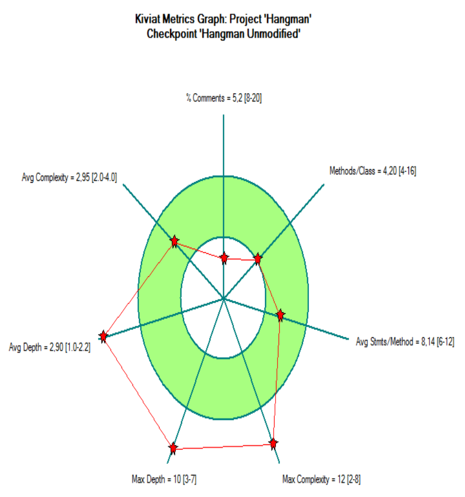
- Blank Lines
  I removed some blank lines to make it more readable, and again, to improve the percentage of comments metric.
- Comments
  Absolutely no comments, so I added some comments explaining the file. This increased the percentage of comments metrics.

4.5 Command.java

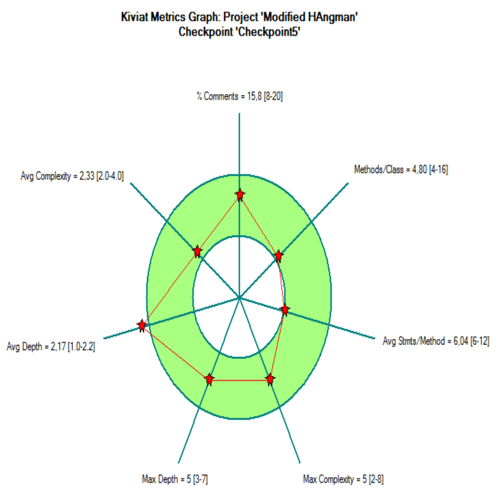
I merged this file with the game class. The only thing that Command.java did was create an enum type that contained restart, top, exit, and help. In my opinion, having a file for just one enum type is a waste, and just complicating. It also makes much more sense that command is implemented in the Game.java file, because that's the "game" itself, which makes it more object oriented.

4.6 Improved metrics

What are the results of running the analyzer on the improved code? Compare the newly obtained metrics with the old ones, both at file level and at project level. Comment on the newly obtained metrics



(a) Old project metrics



(b) Improved project metrics

Figure 5: Metrics summary and Kiviat graph

Table 5: Metrics before and after (Project level)

Metric	**Kiviat Accepted Treshold**	**Old**	**Improved**	**Thoughts**
Methods/Class	4-16	4,20	4,80	Due to the high complexity and depth of some methods, I splitted some methods into smaller ones. This gave a higher method per class. Still within accepted boundaries.
Average statements per method	6-12	8,14	6,04	I removed multiple unused methods, and made some methods smaller, which is why average statement per method decreased. Still within boundaries.
Average Complexity	2.0 - 4.0	2,95	2,38	Average Complexity decreased, because I splitted methods into smaller ones, and also decreased the maximum method complexity. Still within accepted boundaries.
Average Depth	1.0 - 2.2	2,90	2,17	Average Depth is lowered by removing unnecessary brackets, and some unused methods that had alot of depth. Still not within accepted boundaries, but very close.
Max Complexity	2 - 8	12	6	Maximum Method Complexity was reduced by splitting the causing methods into smaller methods, or completely removing some unecessary parts, such as unreachable code etc. Now within accepted boundaries.
Max Depth	3 - 7	10	5	Max Depth was reduced by removing unreachable code, removing unecessary brackets etc. Now within accepted boundaries.

Table 6: Metric before and after(File level / FileReadWriter)

Metric	**Kiviat Accepted Treshold**	**Old**	**Improved**	**Thoughts**
Methods/Class	4-16	10	7	Removing the unused methods or unnecessary ones in FileReadWriter contributed to reducing the methods/class, although it was already in the accepted boundaries.
Average statements per method	6-12	8,4	7	This metric was reduced from 8,4 to 7, because the long unused methods were erased.
Average Complexity	2.0 - 4.0	3,2	3,29	Average Complexity actually increased a little, because the unused methods had complexity of 1. Removing this increased the metric, but still within accepted boundary.
Average Depth	1.0 - 2.2	3,25	2,21	Average depth was reduced when the unused method nop() was removed. It had a depth of 10.
Max Complexity	2 - 8	10	6	Maximum Method Complexity was reduced by erasing unnecessary brackets from the method printAndSortScoreBoard and several others.
Max Depth	3 - 7	10	5	Reduced to accepted boundaries by removing unused method nop() which had a depth of 10.

Table 7: Metrics before and after(File level / Game

Metric	**Kiviat Accepted Treshold**	**Old**	**Improved**	**Thoughts**
% Comment	8-20%	2,3%	10,7%	Blank lines removal + Commenting made this metric better. Now in the accepted boundaries.
Methods/Class	4-16	7	6	I actually added many methods, but this number still decreased. The reason being that I merged Command.java and Game.java. Since there basically are 2 classes in Game.java, the average method/class is reduced. Still within accepted boundaries.
Average statements per method	6-12	11,57	7,42	This metric was reduced from 11,57 to 7,42 due to me splitting the method findLetterAndPrintIt to smaller methods. Now the metric is much closer to the middle.
Average Complexity	2.0 - 4.0	3,71	2,42	Reducing findLetterAndPrintIt reduced the average complexity alot. It had alot of uneccessary brackets, and was generally too large.
Average Depth	1.0 - 2.2	2,93	2,32	Average depth was reduced when findLetterAndPrint it was reduced, and multiple methods were made from it which had much smaller depth.
Max Complexity	2 - 8	12	4	Maximum Method Complexity was splitting findLetterAndPrintIt. Now in acceptable boundaries.
Max Depth	3 - 7	7	5	Again, findLetterAndPrintit was reduced, and max depth reduced.

Upload the modified project in Github. In your project assignment, provide the web link to your modified code.
https://github.com/UyNG/INF3121

In my opinion, modifying the code in order to improve its **metrics** was easy. It's a simple job checking which method has too high complexity / depth.The hardest part was probably reducing the average depth. I didn't make it (got very close), as I didn't want to make even more methods than necessary, which will basically ruin the part of java being object oriented. I wouldn't have done anything different.

4.7 Afterword

I'd definitely say that doing this task learned me much about static analysis. Both its pros and cons. Using source monitor made it easy to find useful metrics such as percentage of comment, unnecessary large methods etc. But it also showed that you'll need dynamic testing to fully test an application. Even if my metrics now are in the acceptable boundaries of the Kiviat Graph, there are some errors you can only find with dynamic testing, such as the highscore not working, or that typing help when there's only 1 dash left will crash the code etc.