“Masters Thesis Software Engineering”

Name:	Sabrina Filemon
Studentnumber:	0346691
Date:	June 22, 2004
Professor:	dr. M.G.J. van den Brand
Institute:	University of Amsterdam

Preface

At the end of this very intensive Masters Programme, I can safely say that I know what I can and cannot do. I have learned a great deal about software, people and life. I would like to thank the people who have helped me achieve this. Especially Mark van den Brand, who has motivated and supported me a great deal.
During this year, Mark van den Brand, Paul Klint and Alban Ponse have made a great effort to make this education a success. This has not gone unnoticed. The other teachers that put their time and effort in this year, Hans van Vliet and Patricia Lago, Daan van den Berg, Tobias Kuipers and Reza Esmaili, are appreciated as well.
Additionally I would like to thank my fellow students with whom I have worked closely most of the time during this year. We hade quite some fun together, of course while studying very hard.
Finally I would like to thank the teachers of the Hogeschool van Amsterdam, Marten Teitsma, Slobodanka Dzebric, Ahmed Nait Aicha and Eric Ravestein and Ruud Roël of the University of Amsterdam, who provided me with an insight in their work in favour of this research.

Alkmaar, June 21, 2004

Table of contents:

1. Summary *1*

2. Introduction *1*

2.1 Automated homework checking **1**

2.1.1 Checking if the code is correct 1

2.1.2 Checking for plagiarism 1

2.1.3 Checking if the code fixes the problem it was meant to fix 2

2.1.4 Checking if the code is structured decently 2

2.1.5 Grading 2

2.2 Eclipse **3**

2.2.1 Popularity 3

2.2.2 Eclipse plugins 4

3. Research *8*

3.1 Research question **8**

3.2 Goals **8**

3.3 Research setup **8**

3.3.1 Plan 8

3.3.2 Setups for the specified metrics and grades tests 9

4. Results *10*

4.1 Execution **10**

4.1.1 Pre-test 10

4.1.2 Test 1 11

4.1.3 Test 2 12

4.1.4 Test 3 13

4.1.5 Test 4 17

5. Conclusion *20*

5.1 Relations **20**

5.2 Strength of found relations **21**

5.3 Algorithm for grading **21**

6. Related work *22*

6.1 Related work **22**

6.2 Future work **22**

7. Evaluation *23*

8. References *24*

Appendix A *25*

Appendix B *27*

Automatic homework checking: Utopia?

1. Summary

In this thesis a relationship between code metrics of student assignments and the grade it received is investigated. Perhaps a program with a high cyclomatic complexity receives a low grade. Possibly other code metrics, such as Lines of Code and the number of methods in a program influence the grade as well. If such relationships exist, can they be used to construct an automated homework checker?

2. Introduction

From software evolution practice it is well known that some properties, like low cyclomatic complexity, a low number of attributes per class, and a low number of parameters per method, indicate that the software is well maintainable. One would expect that, when grading student exercises, teachers intuitively give the resulting programs with good metrics high grades. In this thesis we will show whether this is really the case or whether there is no relation between these grades and the derived code metrics.

In personal interviews with teachers of programming practica[1], they explained that their normal duties include:

ú checking if the code is correct;
ú checking for plagiarism;
ú checking if the code fixes the problem it was meant to fix;
ú checking if the code is structured decently; and
ú grading.

Some of these activities can be automated. Since the Eclipse Framework is becoming increasingly popular in schools and universities in Holland, our focus will be on homework checking tools that can be integrated in Eclipse as a plugin.

2.1 Automated homework checking

2.1.1 Checking if the code is correct

All compilers will check if the code is syntactically correct, simply because the code will not build or compile if it is not.

By using Eclipse for testing and running Java code, one can assert that the code is correct. If it is not, the code will not run. Several plugins exist for assisting in testing and debugging code, e.g. JUnit and the debug workspace. Testing results may prove useful in the grading process.

2.1.2 Checking for plagiarism

Teachers usually trust their students. They say that plagiarism is not a common custom in most universities. And even if a student copied his or her homework from someone else, if he or she is clever enough to scramble the program so much that the naked eye of a teacher cannot tell instantly that it is a copy, then this student is not such a bad programmer anyway. Not an honest one, but that is a different discussion. This principle is carried out through the increasingly popular practice of having students explain orally the program that is displayed on their monitor. It does

[1] These interviews were held on the Hogeschool van Amsterdam with 4 different teachers.

not matter how they got the code as long as they can explain in detail what it does.

Some research has been done in detecting plagiarism in submitted programming exercises by students. Several, mostly freeware, tools exist to aid in detecting plagiarism by comparing two different programs with each other, reduce these to abstract syntax trees and compare the trees to find out how much similarity exists between both programs[2]. One tool deducts the longest common substring by breaking up the first file and trying to use these parts to construct the second file[3].

The checking for plagiarism is not supported in Eclipse. There is a comparison tool that compares two versions of the same file to each other. This is a textual comparison, like 'diff' in UNIX-based systems. It can be used to compare different files as well, but it will display individual differences rather than the degree of similarity. This means it will still be almost as much work to check for plagiarism as when checking manually.

The possibility of creating a custom plugin to check for plagiarism has been studied. In order to do so, a parser would have to be constructed that reduces two files to abstract syntax trees[4]. This means that the code has been stripped of variable names and the sequence of statements. But the teachers involved gave plagiarism a very small priority since it hardly ever happens and since an undetectable fraud requires considerable programming skills anyway. Therefore we decided not to investigate this possibility any further.

2.1.3 Checking if the code fixes the problem it was meant to fix

The program under investigation can be of really good quality, without actually solving the problem it was intended to fix. Many believe that fitness for purpose is actually equal to quality, or at least part of its definition. The easiest way to test this special property, fitness for purpose, is to compare the submitted program to a set of characteristics that any solution to the problem has to contain. Another possibility is to compare the submitted program to a standard solution[5]. Needless to say, for every different problem or exercise, the set of solution characteristics or the canonical solution have to be entered anew.

For checking Java programs, no publicly available tools currently exist. Automation for this part of the homework checking process exists, but not as Eclipse plugins.

2.1.4 Checking if the code is structured decently

Most compilers have features for checking the layout and automatic indentation. Eclipse can be extended with plugins that perform these tasks in the Java editor. Plugins exist that aid in checking the code structure; e.g. that assist you by warning when the nesting level of loops and branches gets too high to be understandable and when the call graph indicate circular dependencies among classes. The use of (JavaDoc) comments, the use of whitespace, limiting the number of characters on a single line, no multiple statements on one line and not using GOTO may help improve readability as well[6]. Research has been done on documentation style in relation to grades[7]. Many tools exist to help in checking the code structure. Actually, any tool that can provide you with metrics about code structure, will suffice.

2.1.5 Grading

[2] Violette, 2000
[3] Grune & Huntjens, 1989
[4] Aho et al. 1986
[5] Hext & Winings, 1969, Morris, 2003
[6] McConnell, 1993
[7] Dulal, 2001

2.1.5.1 Quality

This thesis is not a discussion about the definition of quality. Interested readers may want to check “Facts and Fallacies"[8] for that. It does not matter if fitness for purpose is a part of the defintion of quality, or equal to quality, or no part at all of this definition. To determine the grade for a program, both the properties of the program and its fitness for purpose are relevant. In other words, it is irrelevant if the properties of a program and its fitness for purpose compose its quality or are just stand-alone characteristics of a program.

However, for the sake of writing this thesis, we will establish a working definition to enable a common understanding of quality. In this situation, the properties are those relevant to a software program and the values of those properties determine its quality.

Quality = Portability + Reliability + Efficiency + Human Engineering + Understandability + Modifiability + Testability[9]

The program, submitted by the student, should be tested for the extent to which these properties are present to determine its quality. To test for the presence of these properties, one defines metrics and a value from where the property is present or not. These metrics can be automatically retrieved and analyzed. Based on the quality of the program, a grade can be assigned.

The metrics relevant for a program under investigation are the following:
ú Cyclomatic complexity;
ú Lines of Code (LOC);
ú Number of comments;
ú Number of classes;
ú Number of methods;
ú Lines of code per method;
ú Grade given by teacher

2.1.5.2 Metrics

Metrics provide information about the properties of an object, in this case a Java program. If demands exist about what makes a good Java program, these properties will tell if the program at hand is indeed a good Java program or not. In other words, the metrics can give insight in the quality of a program.

A metric is usually expressed as a certain property, e.g. “Lines of Code”, that can assume a value in a certain range, e.g. a natural number, such as 20.
Later in this thesis we will discuss if a relationship exists between the properties of a Java program made by students and the grade it received.

2.2 Eclipse

2.2.1 Popularity

The Eclipse Platform is a free, open source Java compiler and editor[10]. This definition does not do justice to Eclipse’s full range of capabilities, that I have experienced during these three months. We have decided to perform our research using the Eclipse Platform, because it is already a widespread editor in schools. Chances are its popularity will increase even more because the

[8] Glass, 2002
[9] Boehm, 1975
[10] Gallardo et al., 2003

competition is pricing itself out of the school market, with its limited budgets. A second contribution to Eclipse's growing number of users is the increasing rate of improvements made. A growing market means a growing number of volunteers that help improve its quality even faster. My prediction therefore is that Eclipse will soon dominate the school market and that is why it is most suited to perform this research using the Eclipse Platform.

2.2.2 Eclipse plugins

The Eclipse Platform is a Java interpreter supported by a collection of tools – plugins – that can be arranged at will. If you need a certain feature that is not currently in your workspace you can look for a plugin on the net. If it does not exist, you can write your own plugin.

2.2.2.1 Team in a Box Metrics plugin

For statical analysis and reporting code metrics, several plugins exist. Because of time constraints, we selected the first Eclipse plugin capable of providing the metrics we need for our comparison of student Java programs; the Team in a Box Metrics plugin. It produces several code metrics[11]:

ú Package metrics[12]:
  ú Cyclomatic complexity (CC);
  ú Lines of Code in Method (LOCm);
  ú Number of Levels (NOL);
  ú Number of Parameters (NOP);
  ú Number of Statements (NOS);
  ú Efferent Couplings (Ce);
  ú Number of Fields (NOF);
  ú Weighted Methods per Class (WMC).
ú Type metrics:
  ú Cyclomatic complexity;
  ú Lines of Code in Method;
  ú Number of Levels;
  ú Number of Parameters;
  ú Number of Statements;
  ú Efferent Couplings;
  ú Number of Fields;
  ú Weighted Methods per Class.
ú Method metrics:
  ú Cyclomatic complexity;
  ú Lines of Code in Method;
  ú Number of Levels;
  ú Number of Parameters;
  ú Number of Statements.

Cyclomatic Complexity
McCabe's Cyclomatic Complexity refers to the number of branches one could cross when travelling to a 'code tree'. This number starts off with 1 and is incremented every time a branch like "if", "for", "while" or "case" is encountered.

Efferent Couplings
Efferent Couplings indicate the number of types a class knows about. All types referred to from the current type increment this number by 1.

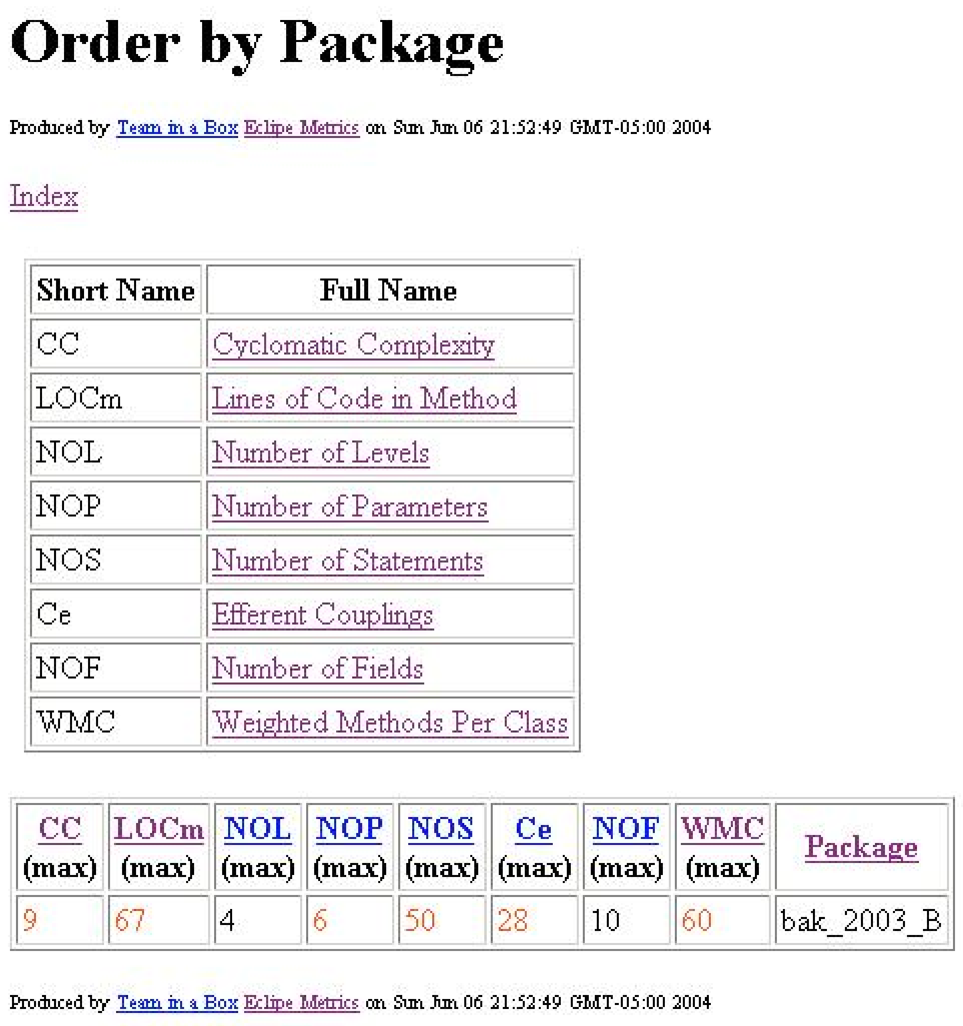
[11] These metrics are defined on the Team in a Box website, www.teaminabox.co.uk.
[12] In the 'package' views, the metrics indicate the highest occurrence found in a class in the package. There is only one value for each metric in the package. In the 'type' views, the metrics also indicate the highest occurrence found in a class, but one result per class, so there will be as much values as there are classes in the package.

Weighted Methods per Class
This metric is the sum of cyclomatic complexities of all methods in a class.

The number of comments could be of interest as well, as mentioned before, but is not supplied. If time allows, this could be deducted as Lines of Code minus the Number of Statements, but it will still include lines of whitespace then.
Metrics that are supported that were not listed as relevant before, will be evaluated as well, if time allows. These include Number of Levels, Number of Parameters, Efferent Couplings, Number of Fields and Weighted Methods per Class.

Somewhat confusing is the fact that this plugin supplies the highest value found instead of the average. So when the cyclomatic complexity per package is presented, this is actually the highest CC found in a type in the package. The CC per type will display the highest CC found in a method in the type. The CC per method will display the CC per method as the name suggests.

The supported metrics can be exported into HTML-files per classfile. A sample HTML-file of the package metrics looks like this:



Order by Package

Produced by Team in a Box Eclipe Metrics on Sun Jun 06 21:52:49 GMT-05:00 2004

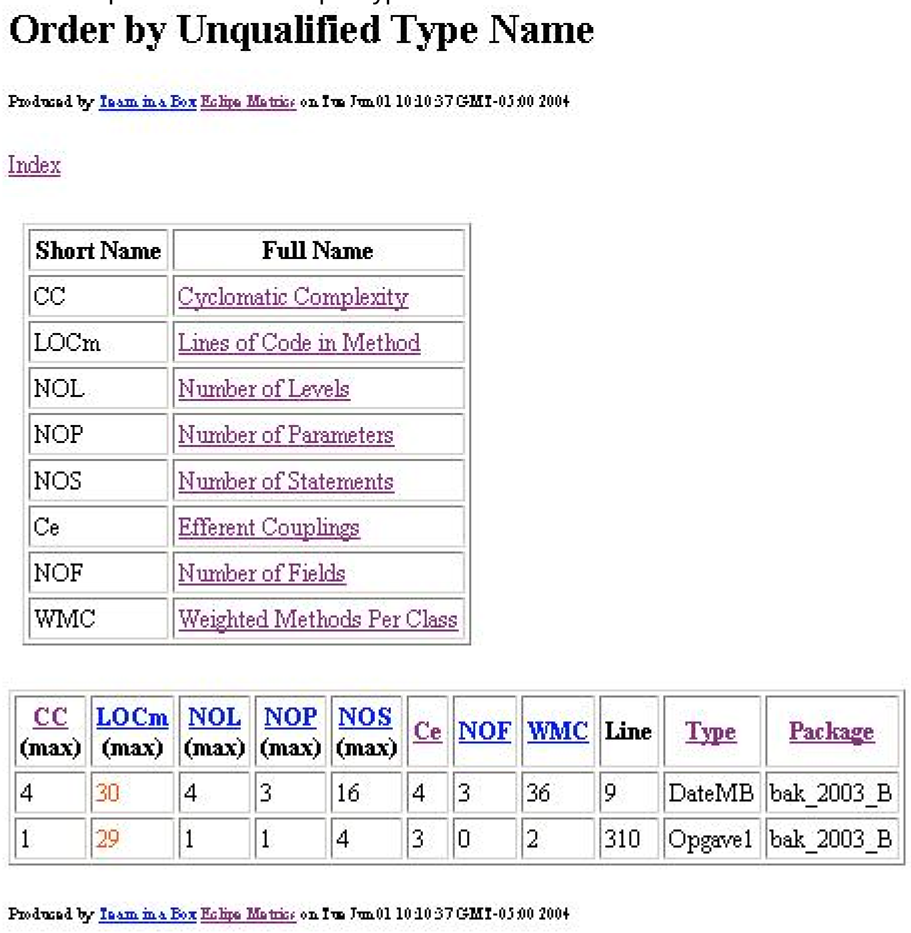
Index

Short Name	Full Name
CC	Cyclomatic Complexity
LOCm	Lines of Code in Method
NOL	Number of Levels
NOP	Number of Parameters
NOS	Number of Statements
Ce	Efferent Couplings
NOF	Number of Fields
WMC	Weighted Methods Per Class

CC (max)	LOCm (max)	NOL (max)	NOP (max)	NOS (max)	Ce (max)	NOF (max)	WMC (max)	Package
9	67	4	6	50	28	10	60	bak_2003_B

Produced by Team in a Box Eclipe Metrics on Sun Jun 06 21:52:49 GMT-05:00 2004

An example of the metrics per type looks like this:



Order by Unqualified Type Name

Produced by Team in a Box Eclipse Metrics on Tue Jun 01 10:10:37 GMT-05:00 2004

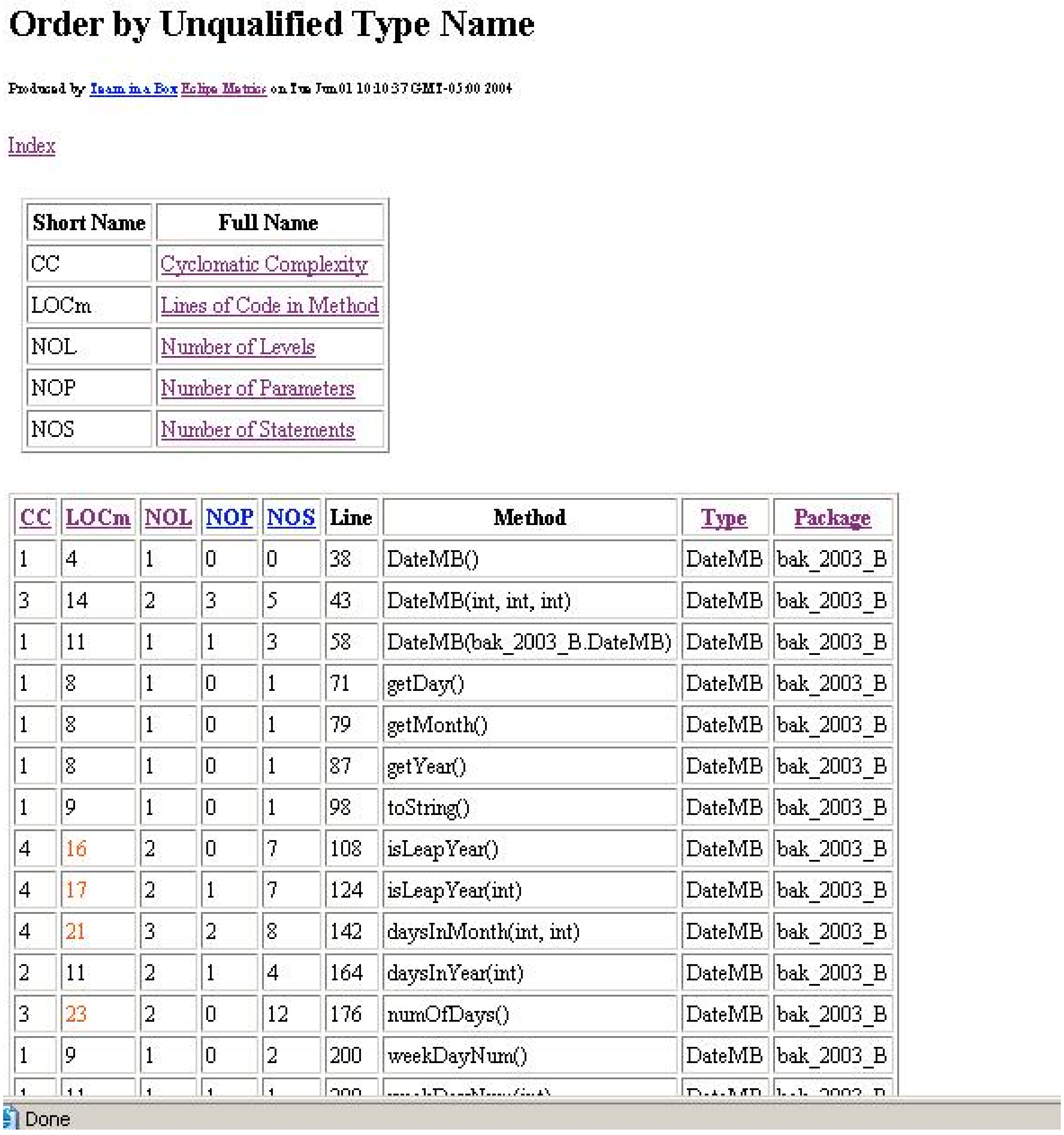
Index

Short Name	Full Name
CC	Cyclomatic Complexity
LOCm	Lines of Code in Method
NOL	Number of Levels
NOP	Number of Parameters
NOS	Number of Statements
Ce	Efferent Couplings
NOF	Number of Fields
WMC	Weighted Methods Per Class

CC (max)	LOCm (max)	NOL (max)	NOP (max)	NOS (max)	Ce	NOF	WMC	Line	Type	Package
4	30	4	3	16	4	3	36	9	DateMB	bak_2003_B
1	29	1	1	4	3	0	2	310	Opgave1	bak_2003_B

Produced by Team in a Box Eclipse Metrics on Tue Jun 01 10:10:37 GMT-05:00 2004

Finally, an example of the metrics per method looks like this:



Order by Unqualified Type Name

Produced by Team in a Box Eclipse Metrics on Tue Jun 01 10:10:37 GMT-05:00 2004

Index

Short Name	Full Name
CC	Cyclomatic Complexity
LOCm	Lines of Code in Method
NOL	Number of Levels
NOP	Number of Parameters
NOS	Number of Statements

CC	LOCm	NOL	NOP	NOS	Line	Method	Type	Package
1	4	1	0	0	38	DateMB()	DateMB	bak_2003_B
3	14	2	3	5	43	DateMB(int, int, int)	DateMB	bak_2003_B
1	11	1	1	3	58	DateMB(bak_2003_B.DateMB)	DateMB	bak_2003_B
1	8	1	0	1	71	getDay()	DateMB	bak_2003_B
1	8	1	0	1	79	getMonth()	DateMB	bak_2003_B
1	8	1	0	1	87	getYear()	DateMB	bak_2003_B
1	9	1	0	1	98	toString()	DateMB	bak_2003_B
4	16	2	0	7	108	isLeapYear()	DateMB	bak_2003_B
4	17	2	1	7	124	isLeapYear(int)	DateMB	bak_2003_B
4	21	3	2	8	142	daysInMonth(int, int)	DateMB	bak_2003_B
2	11	2	1	4	164	daysInYear(int)	DateMB	bak_2003_B
3	23	2	0	12	176	numOfDays()	DateMB	bak_2003_B
1	9	1	0	2	200	weekDayNum()	DateMB	bak_2003_B

Done

In every column, the highest value found in the unit[13] for this metric is displayed. Team in a box defined some advisable limits, e.g., a maximum Cyclomatic Complexity of 8. If the highest score exceeds this limit, it is displayed in red.

[13] class in package, type or method

3. Research

3.1 Research question

Is there a relationship between the metrics of a program and the grade assigned to it? If so, can this relationship be used to construct an automated homework checker to automatically analyze and grade the programs submitted?

Beforehand, we expect Cyclomatic Complexity (CC) and Lines of Code (LOC) to have a negative influence on the grade. Meaning, a higher CC or a higher number of LOC will result in a lower grade. Use of good coding practices that are meant to keep complexity low and the number of lines of code as low as possible, should be rewarded with a higher grade.

3.2 Goals

The goal of this research effort is to prove a relationship exists between the metrics of a software program and the grade given by a teacher, if it does. Furthermore we will describe if and how this relationship can be used to aid in the homework checking and grading process. If it can be molded into an automated homework checker, we will describe how this could be done, initially for Java-programs only.

This research will be restricted to Java-programs because Java is most used in courses.
To analyze the Java-programs we use the Eclipse platform. Tools to retrieve code metrics will therefore be restricted to Eclipse plugins.

This research is limited to a selection of bachelor UvA students in computer science to limit time needed for communications. To make the test results more representative, these tests should be performed on more than one university. Initially, tests had been planned on the HvA as well, but the exercises differed so much from the exercises at the UvA, that they could not be compared.

3.3 Research setup

3.3.1 Plan

To find a relationship between program metrics and the grade it received, programs submitted by students have to be accumulated. However, not all programs will do. For instance, servlets can hardly be analyzed without setting the proper settings and establishing a server and a database if appropriate. These settings will influence the grade as well as the quality of the programming. Furthermore, the pieces of code, sometimes in different programming languages, are scattered throughout the application.

To be able to analyze the programs in a meaningful way, we will therefore acquire only programs that match the following criteria:
- The exercise must be supplied in Java;
- The exercise must be sufficiently large (at least 200 LOC);
- Students should be able to solve the exercise in varying ways, so the metrics will vary as well;
- At least 10 students must submit the exercise;
- The grades assigned to the programs must vary considerably;
- The exercise must have as little as possible dependencies to external settings.

3.3.2 Setups for the specified metrics and grades tests

For this setup, two different sets of students from different classes are needed. A Set 1 (from class A) and a Set 2 (from class B) are selected to filter out differences on the account of different classes, teachers or fellow-students. It will make a more representative selection this way.

Comparisons are made based on the earlier determined criteria. A submission is the program a student created and submitted as a solution to the exercise at hand.

3.3.2.1 Pre-test

The quality and suitability of several plug-ins delivering metrics of a Java-program will be determined by analyzing the metrics of some sample code. A piece of code found on Internet (one simple class only) and some sample code from a different programming course on the UvA.

The quality of the sample code must be determined.

Furthermore, the way to retrieve the metrics and to accumulate and document statistics should be established.

3.3.2.2 Test 1 – High grades

The submissions that received the highest grade for every exercise (Exercise 1 to 6) of both Set 1 and Set 2 will be compared.

3.3.2.3 Test 2 – Low grades

The submissions that received the lowest grade for every exercise (Exercise 1 to 6) of both Set 1 and Set 2 will be compared, to analyze ‘grade-reducing’ properties of a program.

3.3.2.4 Test 3 – Higher or lower than average

The metrics of each submission for Exercises 1, 3 and 5 in Set 1 will be compared to the average value for this metric. At the same time, the grade of this submission will be compared to the average grade for this exercise.

3.3.2.5 Test 4 – Class differences

Test 3 will be repeated for Set 2.

4. Results

4.1 Execution

After the interviews with teachers on the HvA it became clear that the exercises did not fit our needs. There were not enough submissions and the grades would not be available in time. We decided to complete the research with exercises and grades from the UvA.

Exercise 1 should result in a calendar application that can calculate on what day your birthday will be in any given year. It should be able to calculate the number of days between any two dates as well. Furthermore it should be able to count the number of Sundays in a certain month and the number of weeks until the end of the year. In Exercise 2, a cardgame called “Set” has to be created, similar to “Patience” with more players. Exercise 3 was actually a 3-hour exam in which a UNIX user administration system had to be designed and built. Exercises 4, 5 and 6 should result in a Mandelbrot-microscope.[14]

Submissions were acquired from two groups of students, Set 1 and Set 2. Both sets consisted of 13 students. Every student was supposed to submit the answer to 6 exercises. An average of 10 students submitted each exercise in a set.

A total of 128 submissions by 26 different students have been analyzed. Some submissions got corrupted in the metrics gathering process. On opening the file in Eclipse, strings of unreadable characters would appear throughout the code. These files, 21 of them, have been discarded. 2 files have been discarded additionally because they had not been graded.

The values of the metrics and the grade belonging to each submission have been collected and documented per package, type and method. For types and methods, the lowest occurrence, the highest occurrence and the average have been calculated and collected from the metrics plugin. This has been done because some submissions had a very large number of statistics, e.g. when the number of methods exceeded 100. The highest and lowest occurrence and the average were the most interesting. For packages only the highest occurrence found in a type had been noted. This was the only value supplied by the plugin per package, because there was only one package in every exercise in every submission.

The grades were initially administered on a schale of 1 to 15. This grade has been recalculated to a schale of 1 to 10.

An example of gathered metrics of Set 2 – Exercise 1, can be viewed in Appendix A.

4.1.1 Pre-test

Two Eclipse plugins for extracting code metrics exist:
ú Team in a box[15];
ú SourceForge[16].
Since the first plugin by Team in a box provided the necessary metrics, no effort was made to investigate the second option.

Course A does not meet our criteria since the grades are assigned on a scale of 1 to 4 and “1” and “2” occur only sporadically. The variations therefore do not mean enough. However, it can be used to test our own grading machine if relevant.

[14] A more detailed explanation of these exercises can be found at: http://carol.science.uva.nl/~ruudr/Prog_B/pr.html
[15] http://www.teaminabox.co.uk/downloads/metrics/index.html
[16] http://sourceforge.net/projects/metrics

4.1.2 Test 1

Properties of submissions that received an extremely high grade have been examined closely. Extreme values of metrics have been studied as well, to find out if these submissions had extreme metrics values as well. Only if similarities exist in relations in an exercise, it will be mentioned. For example, if the highest grade occurs twice in an exercise, and both programs have a very high cyclomatic complexity, it will be mentioned.

4.1.2.1 Exercise 1

In Set 1 the average grade was 7.53. The highest grade was a 9.33 and occurred twice. The number of types for one of these programs was 6. It was the highest number of types that occurred. Another submission with 6 types got a grade of 6. The other program had a normal number of types. The number of methods for this submission, 16, was below the average of 23.20, but not the lowest which was 15. The lowest number of methods received a grade of 6.67.

In Set 2 the average grade was 7.33. Here, the highest grade was a 9.33 as well and it also occurred twice. The number of types for both these programs was 3, which was close to average. The number of methods for both programs was also close to average.

In Set 1, the CC per package for the first high score program was very high, 18. The average CC per package for all submissions was 12.60. The highest CC was 20 and occurred twice. The grades for these occurrences were a 6.67 and an 8.67. The highest CC per method of this program was also 18, far above the average of all submissions of 12.60. The average CC per method for this submission was 3.63, while the average of all submissions was 2.91. In Set 2, there was also one high value for one high-score program, but the other program had an average value.

The LOCm per package of the first high-score program was very high, 77. One higher occurrence was detected of 145. The average LOCm for all submissions was 57.30. The same numbers appear for the highest LOCm per type and per method for this program. The average LOCm per type was a little above average. In Set 2, a very similar picture existed.

The highest NOP found in a type was only 1, which occured in 2 other submissions, that both got a grade of 6.67. The average highest NOP per type was 2.60.

The highest NOS per package, type and method of 68 was way above the average of 44.30. One submission had a higher NOS of 89. This submission got a 6.67. The average NOS per type for the first program was 39.50. The average for all submissions per type was 26.52. The only higher average found was 50. This submission got a 6.67. The lowest average found was 10. The second high-scored submission had an average of 17, which was the second lowest. The average NOS per method was 12.19. One higher average was found of 14.19. This average belonged to a grade of 8.67.

The second high-score program had an extremely high NOF of 5. The average NOF was 2.30. In Exercise 1 of Set 2, one of the high-score programs also had a very high NOF.

The average Ce of 8 is equal to that of a submission that got a grade of 6.67. No higher occurrences were found. The average Ce of all submissions per type was 5.82.

The average highest WMC for all submissions was 28.60. The first high-score submission had the lowest value of 15, together with another submission that had a grade of 6.67. The second high-score submission had a WMC per package of 19. The lowest score of all submissions was 16. The average WMC per package was 30.90. This high-score program also had the lowest low WMC per type of 2. Another program had a low WMC of 2 as well and got a grade of 8.

From Exercise 1 can be concluded that both submissions with a high score have very different metrics values. In fact, both programs could not be more different in properties.

4.1.2.2 Other exercises

The other exercises of Set 1 and Set 2 do not show much difference in the fact that indisputable relationships between high grades and metrics values do not exist.

The CC did not have a clear correlation with the grade. Grades above the set average seemed to have a higher CC more often than not, but this was only true for about 60 percent of the submissions.

A high NOF means a high grade in Exercise 5. In Exercise 4 this relation is not as clear, but 2 high values are found with 2 of the 6 grades above average. In Exercise 6 high grades seem to go with high NOF as well. All NOF relationships seem clearer in Set 1 than in Set 2.

In Exercise 1 of Set 2 a high WMC value gets a low grade. In Exercise 2, their seems to be no relation whatsoever. In Exercise 3, the high grades go with high WMC values more often than not. In Exercise 4 of Set 1 this is also the case, but in Exercise 4 of Set 2 all WMC values of the highest grade are average. A high WMC per package, type or method gets a high grade in Exercise 5 of Set 1. In Exercise 5 of Set 2, a high grade has very high WMC values. All submissions of Exercise 6 indicate no relation between high grades and WMC values.

4.1.3 Test 2

Test 2 will be performed like Test 1, but this time for examining low grades. Only the notable facts will be mentioned.

4.1.3.1 Exercise 1

In Set 1 the average grade was 7.53. The lowest grade was a 6 and it occurred twice. The number of types for one of these programs was 6. This was the highest number of types and it occurred once more with a grade of 9.33. The other one consisted of only 1 type, which was the lowest number. The average number of types was 3.50. The number of methods for the first low-score program was 5, the lowest. The number of methods for the other low-score program was 47, the highest occurrence.

In Set 2 the average grade was 7.33. The lowest grade was 0. This program had the highest number of types of 5 and the highest number of methods of 31 as well. The average number of types was 2.44 and the average number of methods 16.22.

Both low score programs in Set 1 and the low-score program in Set 2 had a low number of LOCm per type.

One of the low-score programs had a high NOP per package, type and method, the highest occurrence of 5. The average per package and method was 3.20, per type 2.60.

In Set 2, the low-score program had a low highest NOS per type and per method, of 32. It was not the lowest occurrence, which was 15 and belonged to a grade of 8. The average NOS per type was also very low, 17.40, compared to an average of 38.76. Per method, the average NOS for this program was 5.39, the lowest occurrence. Here, the average was 14.28.

One of the low-score programs of Set 1 had a low Ce score per package and type, of 4. This value occurred once more with a grade of 8. The average Ce per package and per type was 8.50.

One of the low-score programs of Set 1 had no fields, the lowest NOF value.

In Set 2, the low-score program had the lowest WMC per package of 21, compared to an average of 34.11.

From investigating the low grades of Exercise 1 can be concluded that a low NOS will result in a low grade. A low Number of Fields will result in a low grade as well. Finally, a low grade will be received when the WMC is low.

4.1.3.2 Other exercises

The number of LOC per method did not have a clear correlation with the grade either. However, a very low or high number of LOC per method seemed to go with a low grade more often than with a high grade.

A low “highest average of NOS found in a method per type” seems to make for a very low grade in Exercises 1, 3 and 5, but the other exercises do not show any relationship here.

A high “average Number of Parameters per method” seems to match with a low grade in Exercises 1 and 2 and possibly 4, but Exercises 5 and 6 show no relationship in this case. Exercise 3 actually claims the opposite.

4.1.4 Test 3

Exercise 1, 3 and 5 have been analyzed by examining the grades matching higher than average or lower than average occurrences of values of the metrics. 10 submissions have been analyzed for Exercise 1. 3 of the 13 submissions of Exercise 3 were corrupted, leaving 10 to be analyzed. For Exercise 5, 9 out of 11 submissions have been analyzed; the other 2 were corrupted.

Every value of every submission was compared to the average value of all submissions of this exercise. If it was above average, it was compared to its grade. For every metric, a chart with four quadrants could be constructed, the four quadrants being:
- Value above average and grade above average;
- Value below average and grade above average;
- Value above average and grade below average;
- Value below average and grade below average.

In the charts, squares consisting of four quadrants, a correlation between a metric and the grades can be distinguished. For every metric such a square was constructed. Per set of students, 49 squares were constructed. The value of a metric of a single submission (program) is depicted as a dot in one of the quadrants:
- high value of the metric and high grade;
- low value of the metric and high grade;
- high value of the metric and low grade;
- low value of the metric and low grade.

Every exercise is displayed in a different color of dots.

Set 1: ○

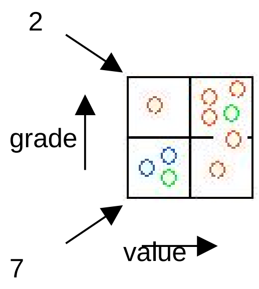
Exercise 1 (Ex1): ○

Exercise 3 (Ex3): ○

Exercise 5 (Ex5): ○

The horizontal middle line in the square represents the average grade. The vertical middle line represents the average value for the metric. Dots that indicate a value of exactly average are placed on the vertical middle line. These dots have not been counted with the total on either one of the diagonals. In the same fashion, grades of exactly average have been placed on the horizontal middle line and have not been counted with the totals either. A value that is extremely high compared to the average (3 times as high) is placed on the right outer border of the square. Values of 0 are placed on the left outer border, except when the average is 0 as well; in this case the dots are placed on the vertical middle line. Grades of 0, which onfortunately occurred, are placed on the lower horizontal border of the square. Other dots have been placed at random locations in the relevant quadrant.

If most dots are on a diagonal line, a positive or negative correlation exists:

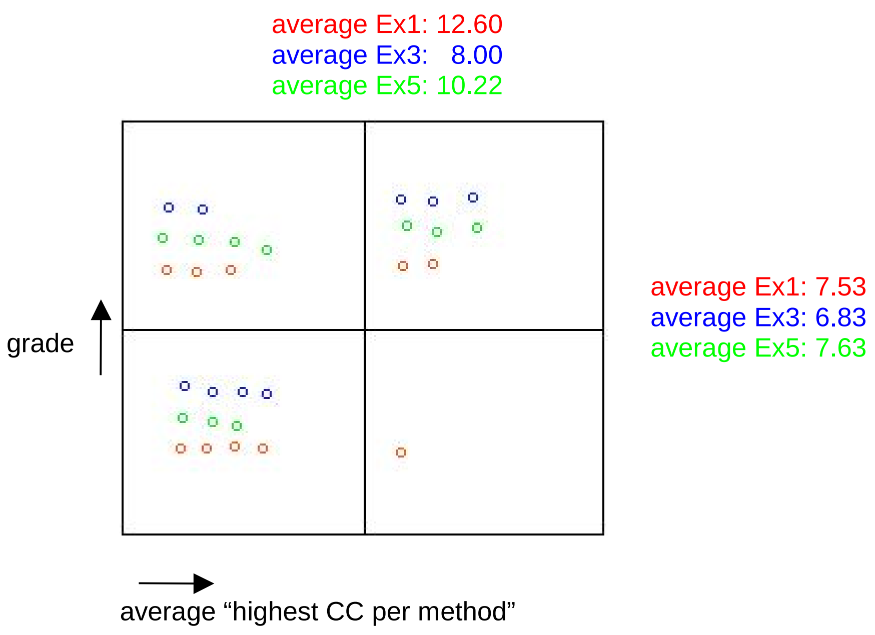


The difference between the number of dots on the diagonals has to be large enough for the relationship to be meaningful enough. Furthermore, the total of counted dots has to be larger than 25, meaning that no more than half the dots can be on the horizontal or vertical middle line. The dots have to be spread equally, or close to equal, over the two quadrants on the diagonal. An upward diagonal means that a low value results in a low grade and vice versa. A downward diagonal means that a low value results in a high grade and vice versa. Only charts that indicate a clear relationship like this are displayed below.

4.1.4.1 Charts

The 5 relationships that show the largest difference here, larger than 7, are displayed below.

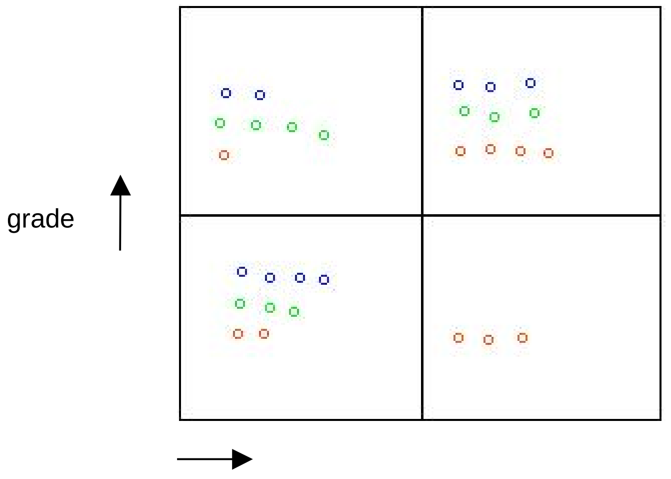
The average “highest CC per method”:



This square shows that a low cyclomatic complexity will get a low grade and vice versa, in two thirds of the cases.

The average "highest LOCm per method":

average Ex1: 13.19
average Ex3: 7.95
average Ex5: 8.40

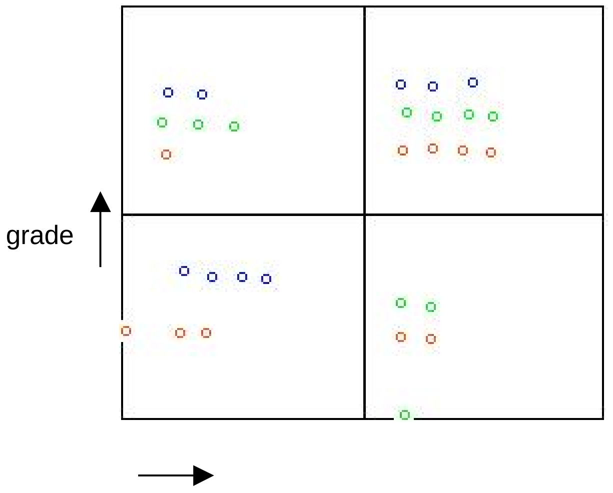


average Ex1: 7.53
average Ex3: 6.83
average Ex5: 7.63

This square shows that a low number of Lines of Code in Method will get a low grade and vice versa, in two thirds of the cases.

The highest "highest NOF per type":

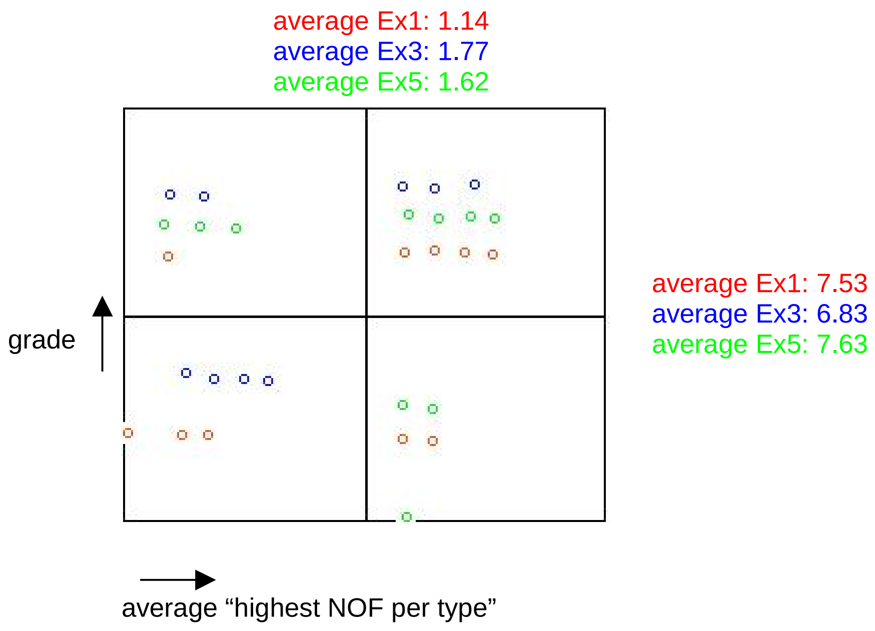
average Ex1: 2.30
average Ex3: 5.22
average Ex5: 4.56



average Ex1: 7.53
average Ex3: 6.83
average Ex5: 7.63

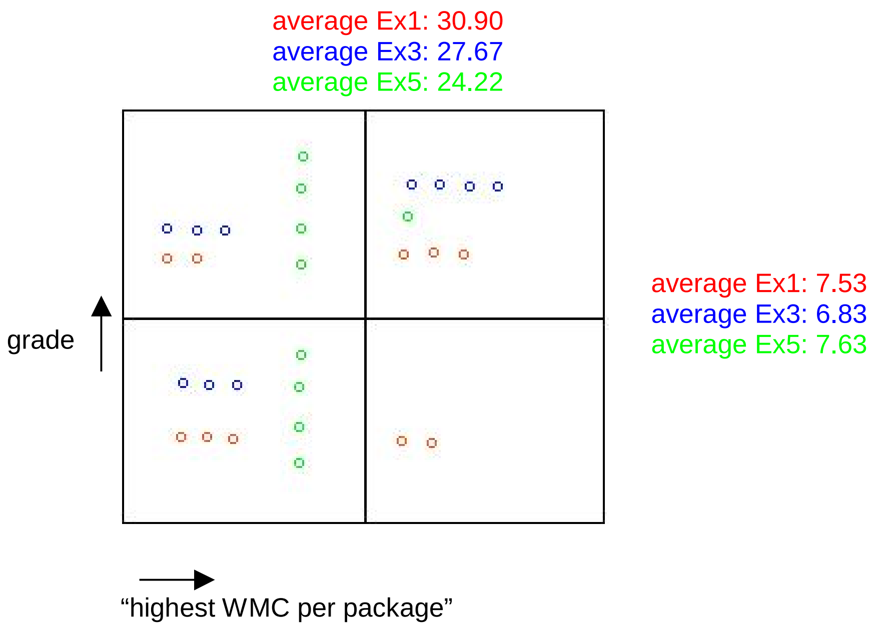
This square shows that a low Number of Fields per type will get a low grade and vice versa in 62 % of the cases. However, this relationship is faint since most dots are placed in the upper right quadrant. This way, it just means that most students used a lot of fields in their code and got a high grade. It says very little about the relationship between the two facts.

The average “highest NOF per type”:



This square shows the same as in the square above.

The “highest WMC” per package:



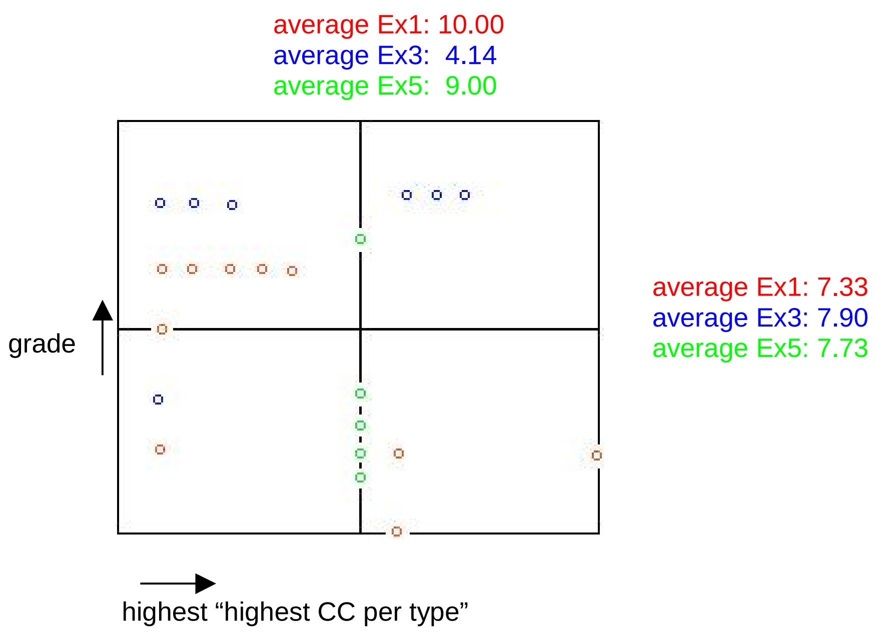
This square shows that a low number of weighted methods per class will receive a low grade in 62 % of the cases.

4.1.5 Test 4

Test 3 has been repeated for Exercise 1, 3 and 5 in Set 2, because of its proven usability in the previous setting. In this group of students, different charts show relationships. Fewer relationships appeared that were as strong as in Set 1 in the previous test. Only 3 charts were found where the number of dots on both diagonals had a difference of at least 7.
For Exercise 1, 10 submissions have been analyzed. 10 students submitted Exercise 3, but 3 were corrupted, leaving 7 to be analyzed. 10 students submitted Exercise 5 as well, but half the files were corrupted.

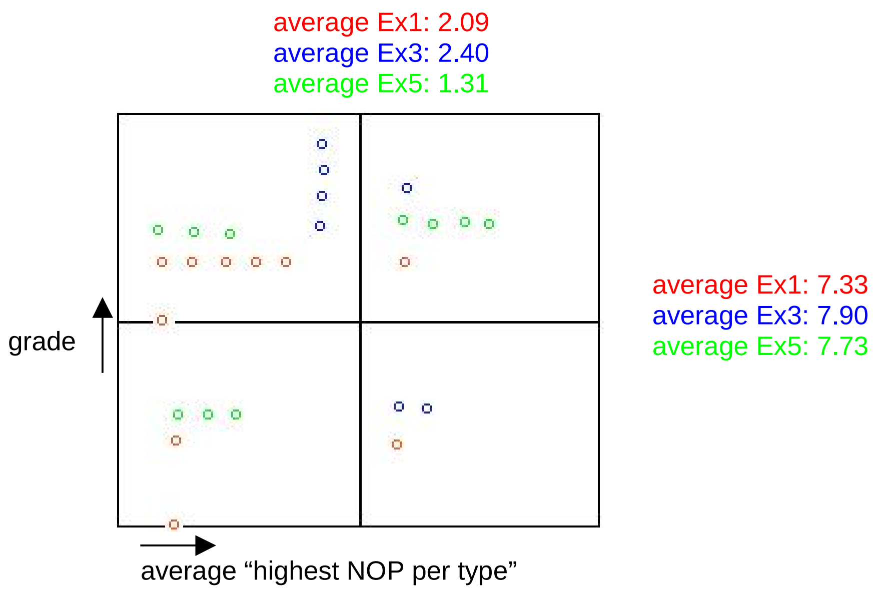
4.1.5.1 Charts

The highest "highest CC per type":



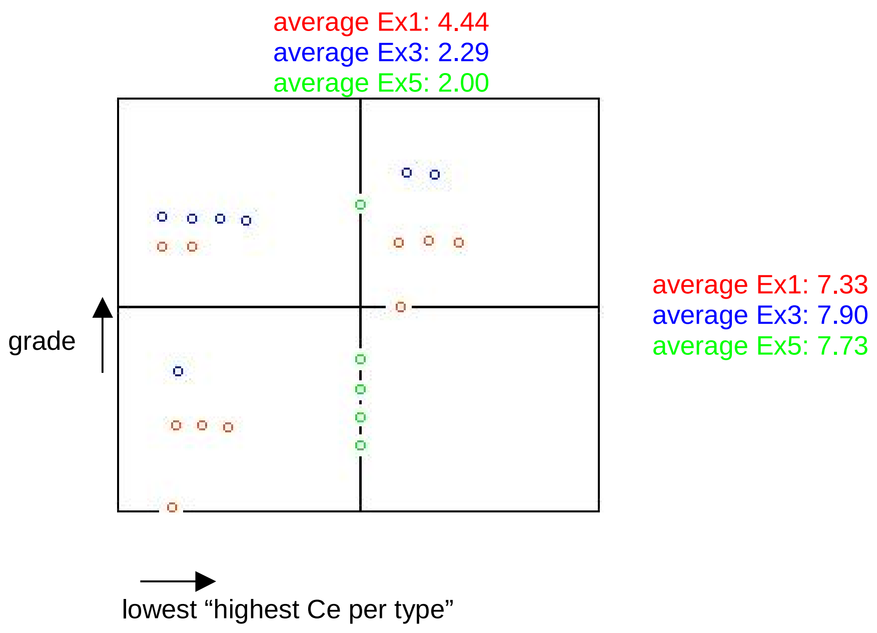
This square shows a negative correlation between the highest cyclomatic complexity per type and the grade in 59 % of the cases that count. A low cyclomatic complexity will get a high grade. However, 6 dots cannot be counted because they are on the middle lines. Most student submissions had a low cyclomatic complexity and got a high grade. But the opposite, a high CC will get a low grade, is not quite proven this way.

The average “highest NOP per type”:



In 59 % of the cases, a low number of parameters will get a high grade. However, the opposite is not proven substantially because most dots are in one quadrant.

The lowest “highest Ce per type”:

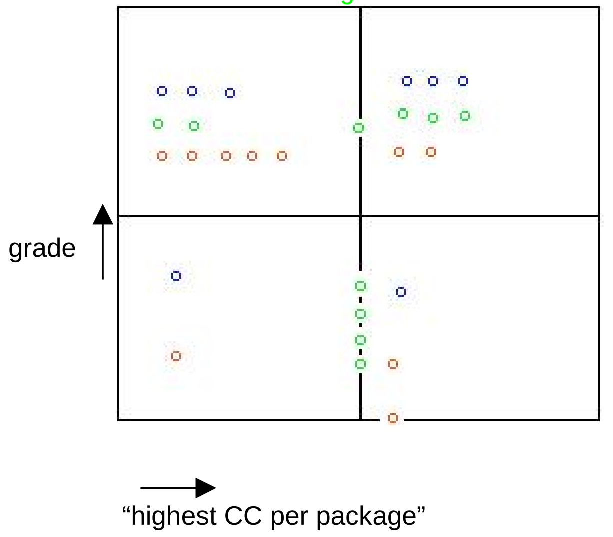


This relationship is based on very little occurences that count. It shows that in 10 out of 16 cases, a low number of efferent couplings will get a low grade and vice versa.

The only relationships with a difference in number of dots on the diagonals greater than 5 that were the same ones as in Set 1, were CC per package and WMC per package:

The “highest CC” per package:

average Ex1: 12.44
average Ex3: 4.14
average Ex5: 9.00

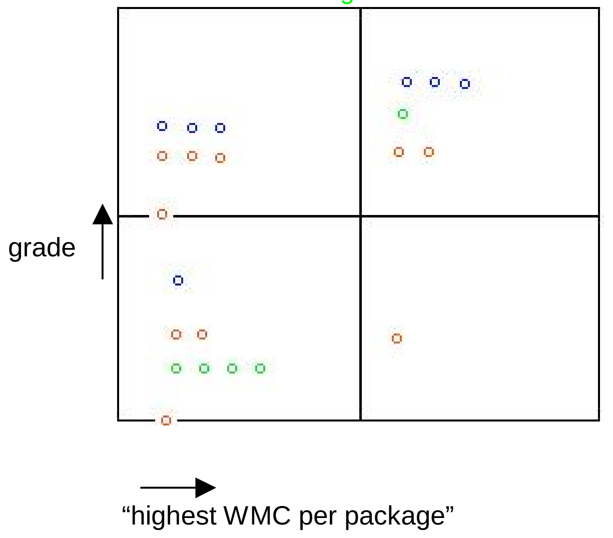


average Ex1: 7.33
average Ex3: 7.90
average Ex5: 7.73

This square shows that a low cyclomatic complexity gets a high grade and vice versa. However, most student programs had a low cyclomatic complexity and got a high grade. The opposite (a high CC with a low grade) did not occur as often, and can therefore not be proven.

The “highest WMC” per package:

average Ex1: 34.11
average Ex3: 27.14
average Ex5:27.20



average Ex1: 7.33
average Ex3: 7.90
average Ex5: 7.73

This square shows a relationship between the weighted methods per class of a program and the grade it receives. A low number of WMC gets a low grade and a high number of WMC a high grade in two thirds of the cases.

5. Conclusion

5.1 Relations

The following can be concluded from the tests:

- If the lowest “highest Number of Statements” per type is low, the grade will be lower. The opposite cannot be concluded.
- If the average “highest Number of Fields” per type is high, the grade will be high and vice versa.
- If the highest “highest Number of Fields” per type is high, the grade will be high and vice versa.
- If the highest number of “Weighted Methods per Class” per package is high, the grade will be high, and vice versa.
- If the average number of “Weighted Methods per Class” per type is high, the grade will be high, and vice versa.
- If the average “highest Lines of Code in Method” per method is high, the grade will be high and vice versa.
- If the average “highest NOP” per type is low, the grade will be high. The opposite cannot be concluded.
- If the lowest “highest Ce” per type is high, the grade will be high and vice versa.
- It seems that a high CC is more often than not ‘rewarded’ with a high grade, but test results conflict in this matter[17].

It should be noted that all relations above are at best proven in 67 % of the cases.

Coding practices could explain some of these results.
A low Number of Statements per type may indicate that a good design took out the need for unnecessary coding, leaving only really useful lines of code. However, this is no more than an educated guess, since the Number of Statements used in a type can be influenced by just about anything.
A high Number of Fields may suggest sensible use of public and private variables. This should be rewarded with a higher grade.
A high number of Weighted Methods per Class suggests that methods have an equal Number of Statements; not one very large method surrounded by many very small methods. Opinions differ in this area, but most agree that a higher grade is legitimate with a high number of Weighted Methods per Class.

Surprisingly, a relation between grade and McCabe’s Cyclomatic Complexity cannot be established with certainty. It appears that a high CC gets a high grade. Another positive correlation exists between the number of Lines of Code and the grade, unlike expected at the beginning of this research project. Both these results cannot be explained by good coding practices, which they seem to contradict. Possibly the use of much more testing material would yield different results.

Another explanation for the absence of clear relations between grades and code metrics could be that other factors are involved in the grading process, such as the motivation of students and their skills in teamwork and communication.

[17] Test 1 and 3 versus Test 4

5.2 Strength of found relations

The full-automatic homework checker only exists in utopia. However, such a tool can prove to be quite efficient in addition to the traditional manual checking.

This research has falsified the assumption that there is a clear relation between good grades and good metrics. The testresults show no indisputable correlations between metrics and grades. The clearest relationships exist between the grades and the “Weighted methods per Class”. This relationship appeared in al four tests. A low WMC-value of the program will result in a lower grade.

Unlike what was expected at the beginning of this research, programs with high grades have a high CC, at least more often than not. However, in Test 4 the opposite seems to be true. The number of Lines of Code seems to have some minor influence on the grade, but only in Test 3. More lines of code result in a higher grade here.

Possible explanations of the absence of clear relationships between code metrics and grades could be that the collection of exercises was not big enough. Or student exercises are too small to get realistic or useful metrics. Maybe a different set of metrics will yield different results.

5.3 Algorithm for grading

Since the relations found are not very strong, an automaton could be constructed by only adding or subtracting a fraction of a standard grade. Or by only modifying the grade in memory if the values are very extreme. E.g. if a high number of Weighted Methods per Class exist, the grade will be incremented with 1.

The following algorithm can be established for an automated homework checker based on earlier findings:
1. First, all average values of all submissions to the exercise should be calculated.
2. If a similarity of 80 percent or higher[18] exists between the current submission and another, plagiarism may have been detected and manual checking should take place. The submission should be excluded from the grading process.
3. If a low (lower than average) number of Lines of Code is discovered, the grade will be incremented with a low factor. Low, because this relationship was relatively weak. If the program has a high number of Lines of Code, the grade will be decremented with the same factor.
4. If a high Number of Fields is discovered, the grade should be incremented with a medium factor. The opposite does not count, because this has not been proven conclusively.
5. If a low Number of Parameters is found, the grade should be incremented with a low factor. The opposite does not count.
6. If a high number of Efferent Couplings is found, the grade should be incremented with a medium factor, and vice versa.
7. If a high number of Weighted Methods per Class is discovered, the grade should be incremented with a high factor. If this number is low, the grade should be decremented with the same factor.

However, some of these steps do not comply with good coding practices. Clearly more research should be done in order to construct an automated homework checker.

[18] Grune & Huntjens, 1989

6. Related work

6.1 Related work

Applications that automatically check and grade student programming assignments generally compare the assignment under investigation with a canonical answer. The grade then depends on the discrepancies between the assignment and that answer. Another way is to test for the presence of a set of characteristics that no solution to the problem can do without.

Existing automatic homework checkers use one of these techniques to analyze and grade the submitted program. Examples are:

- BOSS[19]
  Features include automatic testing against data sets, plagiarism detection and code metrics. BOSS uses code metrics in addition to comparison based on a canonical answer. It is the only tool available that allows for code metrics analysis specifically for educational purposes, but it does not grade on metrics. The code metrics it uses are among others:
  - CBO (Coupling Between Object classes);
  - Complexity Weighting;
  - Documentation;
  - LOCM (Lack Of Cohesion Metric); and
  - RFC (Response For a Class).

  This tool comes closest to the goal of this research. The metrics used here are all different from the ones used in this research.
- Ganesh[20]
  This seems to be a full-automatic grading tool, based on a canonical answer. In this research, code metrics analysis is the goal. Ganesh is not deployed for this.
- Ceilidh-CourseMaster[21]
  This tool does not do grading but it is more of a database with a plagiarism detection feature. It does not use code metrics for analysis or grading.
- SIM[22]
  The similarity checker used on the VU checks for similarities between two files, in order to detect plagiarism. This is a part of the homework checking process. This tool does not analyze code metrics nor does it grade student assignments. The similarity checking can be done for natural language and a multitude of programming languages, including Java, Modula2 and C.

6.2 Future work

The relationships found between code metric and grades are frail. To be conclusive in proving or denying these relationships exist, more measurements should be performed, preferably on different schools. Perhaps other code metrics exist that do show a relationship to grades. The existence of another tool, BOSS, used for grading that uses completely different code metrics seems to suggest this[23].

The relationships found between metrics and grades, though faint, can be used to construct an algorithm for grading. This algorithm or prototype should be tested to see if the grades it supplies are similar to the grades given by a teacher.

[19] http://www.dcs.warwick.ac.uk/boss/
[20] http://www.ncc.up.pt/~zp/ganesh
[21] http://www.cs.nott.ac.uk/CourseMarker/cm_com/html/More_Info.html
[22] Grune & Huntjens, 1989, http://www.cs.vu.nl/~dick/sim.html
[23] http://www.dcs.warwick.ac.uk/boss/

The homework checker could be used to investigate differences in metrics between male and female programmers. Will they show the same statistics according to the homework checker?

7. Evaluation

In the beginning, a lot more time and effort was planned to research plagiarism detection, but it became more and more apparent that this had been done already much more thoroughly than we would be able to achieve in this limited timeframe.
The setting up of the tests and deciding how to retrieve the most accurate and meaningful results took a lot of time. Much more time than was available. This caused some tests to be dropped completely and the pre-testing to be very limited. Test 3 and 4 were not planned out with charts beforehand, but along the way the charts proved very useful and efficient.
A weakness in this testing approach has been that both courses, course A and B, had been tutored and graded by one teacher only. In this point of view, the grading may have been subjective. The number of student submissions of exercises that have been examined has been substantially large, but in Test 3 and Test 4, more test results might have strengthened found conclusions.

8. References

Aho, Alfred V., Sethi, Ravi & Ullman & Ullman, Jeffrey D., “Compilers: Principles, Techniques and Tools”, 1986, 769 p., Addison-Wesley

Gallardo, David, Burnette, Ed & McGovern, Robert, “Eclipse in Action”, 2003, 283 p., Manning Publications

Glass, Robert L., “Facts and Fallacies of Software Engineering”, 2002, 240 p., Addison-Wesley

Grune, Dick & Huntjens, Matty, “Het detecteren van kopieën bij informatica-practica”, Informatie no. 31, pp. 864-867, 1989

Heemstra, Fred J., Kusters, Rob J., Trienekens, Jos J.M., “Softwarekwaliteit:: Op weg naar betere software”, 2001, 341 p., Ten Hagen Stam Uitgevers

Hext, J.B. & Winings, J.W., “An automatic grading scheme for simple programming exercises”, Communications of the ACM Vol. 12 No. 5, 1969, p.p. 272-275,

Hunt, James J., “Delta Algorithms: An Empirical Analysis”, ACM Transactions on Software Engineering and Methodology, Vol. 7 No. 2, 1998, pp. 192-214

Kar, Dulal C., “Automatic characterization of computer programming assignments for style and documentation”, International Conference on Education and Technology, 2001, p.p. 1-3

McConnell, Steve, “Code Complete”, 1993, 776 p., Microsoft Press

Morris, Derek S., “Automatic grading of student’s programming assignments: an interactive process and suite of programs, 33rd ASEE/IEEE Frontiers in Education Conference, 2003, Session S3F, p.p. 1-6

Violette, A., “Generating syntactical Deltas from Java Source Code”, 2000, p.p. 1-6

Appendix A

Course “Programmeren B” – Set 1 – Exercise 1

Set 1
Exercise 1
Submissions: 10

			Student 1	Student 2	Student 3	Student 4	Student 5
Number of Types			3,00	6,00	3,00	5,00	3,00
Number of Methods			23,00	47,00	15,00	34,00	24,00
Cyclomatic Complexity							
	per Package		20,00	13,00	7,00	16,00	8,00
	per Type						
		Lowest	3,00	2,00	3,00	2,00	3,00
		Highest	20,00	13,00	6,00	16,00	8,00
		Average	11,50	5,67	4,50	7,00	5,33
	per Method						
		Lowest	1,00	1,00	1,00	1,00	1,00
		Highest	20,00	13,00	7,00	16,00	8,00
		Average	2,57	2,38	2,53	2,15	2,46
Lines of Code in Method							
	per Package		145,00	39,00	44,00	55,00	57,00
	per Type						
		Lowest	13,00	10,00	13,00	7,00	13,00
		Highest	143,00	39,00	44,00	55,00	57,00
		Average	78,00	18,67	28,50	31,40	38,67
	per Method						
		Lowest	3,00	1,00	1,00	6,00	1,00
		Highest	143,00	39,00	44,00	55,00	57,00
		Average	16,78	6,77	13,07	13,29	14,46
Number of Lines							
	per Package		5,00	3,00	3,00	4,00	4,00
	per Type						
		Lowest	3,00	2,00	3,00	1,00	3,00
		Highest	5,00	3,00	3,00	4,00	4,00
		Average	4,00	2,33	3,00	2,20	3,33
	per Method						
		Lowest	1,00	1,00	1,00	1,00	1,00
		Highest	5,00	3,00	3,00	4,00	4,00
		Average	1,61	1,57	2,00	1,41	2,17
Number of Parameters							

	per Package		3,00	3,00	3,00	3,00	3,00
	per Type						
		Lowest	1,00	1,00	1,00	1,00	1,00
		Highest	1,00	3,00	1,00	3,00	3,00
		Average	1,00	1,33	1,00	1,40	1,67
	per Method						
		Lowest	0,00	0,00	0,00	0,00	0,00
		Highest	3,00	3,00	3,00	3,00	3,00
		Average	0,52	0,49	0,47	0,65	0,46
Number of Statements							
	per Package		89,00	49,00	37,00	46,00	44,00
	per Type						
		Lowest	11,00	6,00	11,00	2,00	11,00
		Highest	89,00	49,00	37,00	46,00	44,00
		Average	50,00	23,83	24,00	20,20	29,00
	per Method						
		Lowest	0,00	0,00	1,00	1,00	0,00
		Highest	89,00	49,00	37,00	46,00	44,00
		Average	8,13	8,04	8,73	6,24	8,21
Efferent Couplings							
	per Package		10,00	10,00	10,00	7,00	10,00
	per Type						
		Lowest	6,00	3,00	5,00	2,00	4,00
		Highest	10,00	10,00	10,00	7,00	10,00
		Average	8,00	5,17	7,50	4,20	6,33
Number of Fields							
	per Package		6,00	3,00	1,00	3,00	3,00
	per Type						
		Lowest	0,00	0,00	0,00	0,00	0,00
		Highest	1,00	3,00	1,00	3,00	3,00
		Average	0,50	2,17	0,50	1,20	1,33
Weighted Methods per Class							
	per Package		24,00	31,00	16,00	26,00	38,00
	per Type						
		Lowest	15,00	6,00	7,00	7,00	6,00
		Highest	20,00	31,00	15,00	26,00	38,00
		Average	17,50	18,67	11,00	14,60	19,67
			0,67	0,67	0,67	0,67	0,67
			10	9	10	12	10
Grade			**6,67**	**6,00**	**6,67**	**8,00**	**6,67**

To save space, 5 submissions have been left out.

Appendix B

Self-assessment

Quality of the research results

The research itself has been performed with excruciating detail and precision. Furthermore the choice of testing methods and displaying of results is, if I may say so myself, inventive and creative. It was a very error-prone and time-consuming process to document and interpret the test results, so errors may have been entered nonetheless. What would have improved the quality of the test results would be increasing the quantity of the data, but this was not possible in favour of the deadline.

Grade: 9

Quality of the thesis

This thesis started out promising to end well within the limit of 11 pages. Somehow this number doubled only in the last two days before this deadline. But quality and quantity are not necessarily related. What could have been better is having a clearer 'storyline'. This thesis may reflect the switch of focus that took place halfway in this project, causing a 'messy' appearance. What could not be better is the use of language.

Grade: 7.5

Difficulty of the research question

This research project initially started out with a different focus, that of detection of plagiarism. The theorethical introduction to parsers and compiler construction of the first focus, plagiarism detection, was extremely complicated but very interesting as well. Due to the heavy time constraints and the poor research upfront, it was not discovered in time that our research question had already been solved in detail. Therefore we changed the focus to an automated homework checker where plagiarism detection is still, but only partially, relevant. Only half the time for this project was left. This change caused that most of the initial reading had become irrelevant and more reading had to be done to cover this new topic. This was the major drawback of this research. The second focus was less complicating, but especially Eclipse was a completely new topic for yours truly. But the research question of the relation between code metrics and grade was easier.

Grade: 7.5

Relevance of the courses of the Masters Software Engineering

Not all courses were relevant, but this would not have been possible. Most relevant were Software Evolution as in code metrics and Software Testing as in quality of software. Of course the relevance of the research question was not entirely up to us.

Grade: 8