CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Comparative Case Study on Tools for Internal Software Quality Measures

Bachelor of Science Thesis in Software Engineering and Management

MAYRA NILSSON

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

{A Comparative Case Study on Tools for Internal Software Quality Measures}

{MAYRA G. NILSSON }

Supervisor: {LUCAS GREN}{VARD ANTINYAN}
Examiner: {JENIFFER HORKOFF}

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

[Cover:
Generated image based on keywords used in this thesis.]

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

A Comparative Case Study on Tools for Internal Software Quality Measures

Mayra Nilsson
The Gothenburg University
Department of Computer Science
and Engineering
Software Engineering Division
Sweden
gussolma@student.gu.se

***Abstract* — Internal software quality is measured using quality metrics, which are implemented in static software analysis tools. There is no current research on which tool is the best suited to improve internal software quality, i.e. implements scientifically validated metrics, has sufficient features and consistent measurement results. The approach to solve this problem was to find academic papers that have validated software metrics and then find tools that support these metrics, additionally these tools were evaluated for consistency of results and other user relevant characteristics. An evaluation of the criteria above resulted in a recommendation for the Java/C/C++ tool Understand and the C/C++ tool QAC.**

Keywords — software metrics tools, static analysis tools, metrics, attributes.

I. INTRODUCTION

Software quality has been a major concern for as long as software has existed [1]. Billing errors and medical fatalities can be traced to the issue of software quality [2]. The ISO/IE 9126 standard defines quality as "*the totality of characteristics of an entity that bears on its ability to satisfy stated and implied needs*" [3]. This standard categorizes software into internal and external quality where internal quality is related to maintainability, flexibility, testability, re-usability and understandability and external quality is related to robustness, reliability, adaptability and usability of the software artefact. In other words, external quality is concerned with what the end user will experience, and internal quality is related to the development phase, which ultimately is the ability to modify the code safely [38]. One might argue that the customers point of view is the most relevant, but since software inevitably needs to evolve and adapt to an ever-changing environment internal quality is essential. Unadaptable code can mean high maintenance costs and could in extreme cases cause major rework [39]. The focus of this thesis is internal software quality metrics and the tools used to measure them, specifically which validated metrics are implemented in the tools, whether the measurements for these metrics are consistent and if these tools have enough support and integration capabilities to be used daily.

While much research has been conducted on internal software quality metrics in the form of empirical studies, mapping studies and systematic literature reviews [4] [5] [6], very little research has been done on the tools that implement these measures regarding their capabilities and limitations. Lincke, Lundberg and Löwe [7] conducted a study on software metric tools, which concludes that there are variations regarding the output from different tools for the same metric on the same software source. This indicates that the implementation of a given metric varies from tool to tool. The limitation of their study is that the metrics were selected based on which metrics are generally available in commonly used tools. The fact that the metrics are not necessary scientifically validated limits its usefulness, since practitioners cannot be certain that the metric actually relates to internal software quality. Scientifically validated means that an empirical study has been conducted that concludes that a given metric can predict an external software quality attribute, where an external attribute can for example be maintainability, fault proneness or testability. Empirical validation is done by studying one or several metrics on iterations of source code and using statistical analysis methods to determine if there is a significant relationship between a metric and an external attribute. Basili et al [4] conducted such a study on 8 separate groups of students developing a system based on the same requirements. For each iteration of the software the metrics were studied to see if they could predict the faults that were found by independent testing.

Briand et al [35] define empirical validation of a metric as "The measure has been used in an empirical validation investigating its causal relationship on an external quality attribute". An external quality attribute is a quality or property of a software product that cannot be measured solely in terms of the product itself [35]. For instance, to measure maintainability of a product, measurement of maintenance activities on the product will be required in addition to measurement of the product itself [35]. This is only possible once the product is close to completion. Internal quality metrics are used to measure internal quality

attributes like complexity or cohesion, which can be measured on the code itself at an early stage in a project. The value of validating an internal quality metric in regard to an external attribute is that they can then be used to predict the external attribute at an early stage in the project.

Many different static software metric tools are used for commercial purposes, but the choice of which tools to use is not based on the scientific validity of the measures but rather on how popular these measures are and whether they are recommended by external standards, for instance MISRA or ISO 9126. To the best of the author's knowledge there is no scientific study which investigates the existing tools and provides knowledge on their adequacy of use in terms of validity of measures, coverage of programming languages, supported operating systems, integration capabilities, documentation and ease of adoption and use. The aim of this thesis is therefore to identify studies that validate internal software metrics and provide an overview of the tools that support validated internal quality measures in order to support decision making regarding which tool or combination of tools would be suitable for a given situation. To make this information accessible, a checklist was developed where the identified tools are classified according to the metrics that they support. Additionally, knowledge is provided regarding the consistency of the measurements in the selected tools. The consistency is evaluated based on the measurement results from using these tools on different sets of open source code projects. To address the research problem the following research question was formulated:

Which are the key internal code quality measures in available tools that could help practitioners to improve internal quality?

In order to answer the above stated question, the following sub questions were answered:

- RQ1 *Which are the most validated internal quality measures according to existing scientific studies?*
- RQ 2 *Which are the tools that support these measures and also have high availability in terms of cost, coverage of programming languages, user interface, supported operating systems, integration capabilities and available documentation?*
- RQ 3 *To what extent are these tools consistent in conducting measurements on a set of open source projects?*

II. Literature Review

Internal software quality is related to the structure of the software itself as opposed to external software quality which is concerned with the behaviour of the software when it is in use. The end user of the software will obviously be concerned with how well the software works when it is in use. The structure of the software is not visible to the end user but is still of immense importance since it is commonly believed that there is a relationship between internal attributes (e.g., size, complexity cohesion) and external attributes (e.g., maintainability, understandability) [8]. In addition, the availability of software testing is not the same for external and internal attributes. External quality is limited to the final stages of software development, whereas testing for internal quality is possible from the early stages of the development cycle, hence internal quality attributes have an important role to play in the improvement of software quality. The internal quality attributes are measured by means of internal quality metrics [9]. According to Lanza and Marinescu [10] software metrics are created by mapping a particular characteristic of a measured entity to a numerical value or by assigning it a categorical value. Over the last past 40 years, a significant number of software metrics have been proposed in order to improve internal software quality. Unfortunately, it is difficult to analyse the quality of these metrics because of a lack of agreement upon a validation framework, however this has not stopped researchers from analysing and evaluating metrics [11]. There are a significant number of metrics available to assess software products, for instance a mapping study on source code metrics by Nuñez-Varela et al. [12] shows that there are currently 300 metrics based on the 226 papers that were studied.

Metrics can be valid for all programming languages, but some apply only to specific programming paradigms and the majority can be classified as Traditional or Object Oriented Metrics (OO) [13] [14]. Considering the popularity of object oriented metrics, it is not surprising that most of the validation studies concentrate on OO [15] [16]. Basili *et al.* [4], conducted an experimental investigation on OO design metrics introduced by Chidamber & Kemerer to find out whether or not these metrics can be used as predictors for fault-prone classes. The results showed that WMC (Weighted Method Count), DIT (Depth of Inheritance of a class), NOC (Number of Children of a Class), CBO (Coupling Between Objects), RFC (Response for a Class) and LCOM (Lack of Cohesion of Methods) are useful to predict class fault-proneness in early development phases. The same results were obtained by Krishnan *et al.* [17]. In 2012 Yeresime [18] performed a theoretical and empirical evaluation on a subset of the traditional metrics and object oriented metrics used to estimate a systems reliability, testing effort and complexity. The paper explored source code metrics such as cyclomatic complexity, size, comment percentage and CK Metrics (WMC, DIT, NOC, CBO, RFC LCOM). Yeresime's studies concluded that the aforementioned traditional and object oriented metrics provide relevant information to practitioners in regard to fault prediction while at the same time provide a basis for software quality assessment. Jabangwe *et al*. [19] in their systematic literature review which focused mainly on empirical evaluations of measures used on object oriented programs concluded that the link from metrics to reliability and maintainability across studies is the strongest for: LOC (Lines of Code), WMC McCabe (Weighted Method Count), RFC (Response for a Class) and CBO (Coupling Between Objects). This topic was later also studied by Ludwig *et al.* [20] and Li *et al.* [21]. Antinyan, *et al.* [22] proved in their empirical study on complexity that complexity metrics such as McCabe cyclomatic complexity [23], Halstead measures [24], Fan-Out, Fan-In, Coupling Measures of Henry & Kafura [25], Chidamber & Kemerer OO measures [26] Size

measure [27] and Readability measures [28] [29] correlate strongly to maintenance time. They also suggested that more work is required to understand how software engineers can effectively use existing metrics to reduce maintenance effort. In 2017 Alzahrani and Melton [30] defined and validated client-based cohesion metrics for OO classes, they performed a multivariate regression analysis on fourteen cohesion metrics applying the backwards selection process to find the best combination of cohesion metrics that can be used together to predict testing effort, the results revealed that LCOM1 (Lack of Cohesion of Methods 1) LCOM2 (Lack of Cohesion of Methods 2), LCOM3 (Lack of Cohesion of Methods 3) and CCC (Client Class Cohesion) are significant predictors for testing effort in classes [31]. The empirical validation of OO metrics on open source software for fault prediction carried out by Gyimothy *et al.* [16] on Mozilla and its bug database Bugzilla shows that CBO (Coupling between Objects), LOC (Lines of Code) and LCOM (Lack of Cohesion on Methods) metrics can predict fault-proneness of classes. The empirical validation of nine OO class complexity metrics and their ability to predict error-prone classes in iterative software development performed by Olague *et al.* [32] has shown that WMC, WMC McCabe among others can be used over several iterations of highly iterative or agile software products to predict fault-prone classes.

In 2010 Al Dallal [33] mathematically validated sixteen class cohesion metrics using class cohesion properties, as a result only TCC (Tight Class Cohesion), LCC (Loose Class Cohesion) [34], DC(D) (Degree of Cohesion-Direct), DC(I) (Degree of Cohesion-Indirect), COH (Briand Cohesion) [35] and ICBMC (Improve Cohesion Based on Member Connectivity) [36, 37] were considered valid from a theoretical perspective, he concluded that all the other metrics studied need to be revised otherwise their use as cohesion indicators is questionable.

III. Research Method

In order to answer research question 1 a review of previous work on software metrics validation was done. The main goal was to elicit the validated internal quality measures based on scientific studies. There are two types of validation, theoretical and empirical [35]. For the following sections only empirical studies will be considered, since this is considered to be the most relevant form of validation [35]. After selecting the empirically validated metrics, tools were found that support these metrics and they were tested for consistency on an open source code bases.

Step 1: Searching and identification of relevant papers

To perform the search of relevant papers related to the topic the online database SCOPUS[1] was used to identify relevant research papers, the subject area was restricted to Engineering and Computer Science, the string below was built based on keywords as well as synonyms defined for the study. Since the main purpose was to find reliable scientific text and metadata only digital libraries and international publishers of scientific journals such as Google Scholars[2], IEEE Digital Library[3], Science Direct[4], Springer[5] and Engineering Village[6] were used as sources.

validated OR *verification of internal quality* OR *code quality* OR *software quality* AND *internal metric* OR *metrics* OR *software metrics* OR *code metrics* OR *measure* OR *measuring* AND *tools* OR *metrics tools*

After the search 567 articles were found (Fig. 1) many of which were irrelevant for the purpose of this paper.

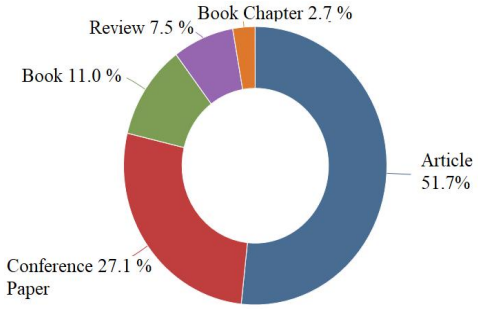


Fig 1 Pie chart showing types and percentage of papers found

A second search was done to narrow down the search and this time the following string was used:

validated AND evaluation *AND* internal *AND* quality *OR* code *AND* quality *OR* software *AND* quality *OR* internal *AND* metric *OR* metrics *OR* software *AND* metrics *OR* code *AND* metrics *OR* measure *OR* measuring *OR* tools *OR* m etrics *AND* tools *AND (EXCLUDE (SUBJAREA* , "MEDI" *) OR EXCLUDE (SUBJAREA* , "BIOC" *) OR EXCLUDE (SUBJAREA* , "ENVI" *) OR EXCLUDE (SUBJAREA* , "CHEM" *) OR EXCLUDE (SUBJAREA* , "AGRI" *) OR EXCLUDE (SUBJAREA,* "PHYS" *) OR EXCLUDE (SUBJAREA* , "SOCI" *))*

The result of the second search was 292 papers related to the topic.

Step 2: The analysis of papers

[1] https://www.scopus.com

[2] https://scholar.google.se/

[3] http://www.springer.com

[4] http://ieeexplore.ieee.org

[5] http://www.sciencedirect.com

[6] https://www.engineeringvillage.com

The 292 scientific papers and articles found in step 1 were assessed according the following criteria:

Inclusion Criteria	
I1	Papers published in a Journal or Conference
I2	Papers that present studies on empirical validation or verification of internal quality or software metrics.
Exclusion Criteria	
E1	Papers that are not written in English
E2	Papers that do not have internal metrics context and do not provide scientific validation of internal quality metrics.

Table 1 Inclusion and Exclusion criteria

The output from this step resulted on a list of 13 relevant research papers verifying or validating internal software metrics. This metrics were categorized into Traditional (LOC, McCabe, etc) and Object Oriented (OO) (coupling, cohesion and inheritance).

Step 3: Selection of validated metrics

The goal of this step was to select the metrics that have been validated. A total number of 29 metrics that have one or more papers that supported them were selected. In order to reduce the risk that a metric has been incorrectly validated only metrics that have been validated at least twice were considered. Out of the 29 metrics with one or more papers supporting them a subset of 18 metrics were found that have two or more papers supporting them. The complete list of the validated metrics and the subset selected and used is shown in Table 3 and Table 4 respectively.

Step 4: Selection of tools

For the selection of the tools, a free search on internet was conducted. The main criteria was that the tools should calculate any type of static analysis. As a result, 130 tools were found (Appendix A).
After the initial search the tools were chosen according the following criteria:

Criteria	*The tool should be able to*
C1	Support static analysis
C2	Run one or more of the metrics established in Table 4
C3	Be open source, freeware or commercial tool with a trail option
C4	Support programs written in C/C++ or Java
C5	Support system integration to IDEs, Continuous Integration, Version Control or Issue Tracker Tools
C6	Provide documentation such as user manual and installation manual

Table 2 Tools selection criteria

As a result, 8 tools were selected for this thesis: QAC[7], Understan[8], CPPDepend[9], SourceMeter[10], SonarQube[11], Eclipse Metrics Plugin[12], CodeSonar[13] and SourceMonitor[14].

Step 5: Selection of code source

The tools were tested on two different Github open source projects, one written in Java and one written in C/C++. Github provides a large variety of open source software projects written in different programming languages. The following criteria was applied when selecting a source:

- The source needs to be written in one single programming language, either C or Java.
- The source needs to be able to compile in their respective environment.
- Because of the limited licenses of some commercial tools the maximum size needs to be less than 10 000 lines of code

The projects were chosen randomly given the constraints stated above.

Step 6: Consistency of test results

In this step open source code was analysed by the tools selected in step 5 regarding the metrics selected in step 4. During this phase the tools are divided into two groups, firstly Eclipse Metrics, SourceMonitor, SonarQube and Understand are tested using Java and QA-C and CPPDepend using C. The output from this step is a matrix with tool, metric, and the measurement results.

IV. Results

A. Selection of metrics

In this section the results obtained from the search for internal quality metrics is presented. A total number of 292 papers on internal software quality were found. Based on the inclusion and exclusion criteria described in Table 1, Section III, 13 research papers were selected for this study.

After narrowing down the number of scientific papers an in-depth analysis of each was performed and a preliminary table with the 29 metrics found in these papers was created (Table 3).

[7] https://www.qa-systems.com
[8] https://scitools.com
[9] https://www.cppdepend.com
[10] https://www.sourcemeter.com
[11] https://www.sonarqube.org
[12] eclipse-metrics.sourceforge.net
[13] https://www.grammatech.com
[14] www.campwoodsw.com

#	Metric	No. of paper
1	Weight Methods per Class	9
2	Lack of Cohesion on Methods	8
3	Depth of Inheritance	8
4	Response for Classes	8
5	Number of Classes	8
6	Coupling Between Objects	7
7	Tight Class Cohesion	5
8	Loose Class Cohesion	4
9	Lines of Code	4
10	McCabe Complexity	3
11	Lack of Cohesion on Methods 2	3
12	Lack of Cohesion on Methods 3	2
13	Lack of Cohesion on Methods 1	2
14	Degree of Cohesion (Direct)	2
15	Degree of Cohesion (Indirect)	2
16	Fan-Out Fan-In	2
17	Number of Methods	2
18	Weight Methods per Class (MacCabe)	1
19	Standard Deviation Method Complexity	1
20	Average Method Complexity	1
21	Maximum CC of a Single Method of a Class	1
22	Number of Instance Methods	1
23	Number of Trivial Methods	1
24	Number of send Statements defined in a Class	1
25	Number of ADT defined in a Class	1
26	Sensitive Class Cohesion	1
27	Improved Connection Based on Member Connectivity	1
28	Lack of Cohesion on Methods 4	1
29	Number of Attributes	1

Table 3 List of found validated metrics in literature

To reduce the risk of incorrectly validated metrics an additional condition of 2 supporting papers was imposed. This resulted in a final selection of 18 metrics as shown in Table 4.

#	Metric	Attribute	Paper
1	Lack of Cohesion on Methods	Cohesion	[4][15][16][17][18][19][21][30]
2	Depth of Inheritance	Inheritance	[4][5][15][16][17][18][19][21]
3	Response for Classes	Coupling	[4][5][15][16][17][18][19][21]
4	Coupling Between Objects	Coupling	[4][5][15][16][17][18][19]
5	Number of Classes	Inheritance	[4][5][15][16][17][18][19][21]
6	Weight Methods per Class	Complexity	[4][5][15][16][17][18][19][21][32]
7	Lines of Code	Size	[5][15][16][19]
8	Number of Methods	Size	[5][21]
9	McCabe Complexity	Complexity	[5][18][32]
10	LCOM1	Cohesion	[19][30]
11	LCOM2	Cohesion	[5][19][30]
12	LCOM3	Cohesion	[19][30]
13	LCOM4	Cohesion	[30]
14	Loose Class Cohesion	Cohesion	[5][30][33][34]
15	Tight Class Cohesion	Cohesion	[5][19][30][33][34]
16	Fan- Out Fan-In	Coupling	[5][15]
17	Degree of Cohesion (Direct)	Cohesion	[30][33]
18	Degree of Cohesion (Indirect)	Cohesion	[30][33]

Table 4 List of metrics and corresponding attributes

B. Selection of Tools

Research question 2 is concerned with which tools support the validated measures and in addition have other characteristics that make them easy to adopt. In total there are over 130 commercial and non-commercial tools (see Appendix A) that claim to support one or several of the validated metrics in Table 4. No tool was found that supports all of the metrics in table 4, which meant finding tools that support as many of the validated metrics as possible. A preliminary search of the prospects for each tool indicated that several metrics were supported, but a deeper analysis of the technical documentation showed this was this was not always the case, since some metrics were not in the trial versions or were supported, but under a different name than in table 4. The aim of this paper is to aid practitioners to improve the quality of their code, so given that there are several tools that support the same metrics additional criteria can be imposed to find the most useful tools. These criteria are integration capabilities to IDEs, version control systems, continuous integration and issue tracker systems, etc. In addition, the availability and quality of

documentation was also considered. A table representing such information was created (Table 6). An additional limitation was that most of the commercial tools trial versions did not allow for a full evaluation since reports generated by the tools could not be saved, printed or exported and not all metrics or features supported were available. Moreover, some of them required legal binding contracts for the trial as well as written clarification of the purpose and the context in which the tool´s reports will be used. Applying all these constraints on the tools narrowed the selection down to 6 as shown in Table 5. A list with a detailed description of the metrics per tool is presented in Appendix B

Tool	Description
QA-C	Is a commercial static code analysis software tool for the C and C++ language. It performs in-depth analysis on source code without executing programs. It provides analysis and reports on internal software measurements, data flow problems, software defects, languages implementation, errors, inconsistencies, dangerous usage and coding standards violations according to regulations for MISRA, ISO 26262, CWE and CERT. It supports 66 internal metrics divided into File Based Metrics and Function Based Metrics.
Understand	Is a commercial code exploration and metrics tool for Java, C, C++, C#. It supports 102 different standard metrics.
CPPDepend	Is a commercial static analysis tool for C and C++. The tool supports 40 code metrics, allows the visualization of dependencies using directed graphs and dependency matrix. It also performs code base snapshots comparisons, and validation of architectural and quality rules. The metrics are divided into Metrics on Fields, Metrics on Methods, Metrics on Types, Metrics on Namespaces, Metrics on Assemblies and Metrics on Applications.
SonarQube	SonarQube, formerly Sonar, is an open source and commercial platform for continuous inspection of code quality. It performs automatic reviews with static analysis of code to detect bugs, conduct code smells and security vulnerabilities on 20+ programming languages It offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security vulnerabilities. It supports 59 metrics.
Eclipse Metrics Plugin version 1.0.9	Is a free code analysis plugin that calculates various code metrics during build cycles and warns via the problems view of range violations for each metric. This allows for continuous code inspections. It Is able to export metrics to HTML for public display or to CSV format for further analysis. It supports 28 different metrics.
Source Monitor	Is an open source and freeware program for static code analysis it calculates method and function level metrics for C++, C, C#, VB.NET and Java. It displays and prints metrics in tables and charts, including Kiviat diagrams and exports metrics to XML or CSV (comma-separated-value) files. It supports 12 metrics.

Table 5 Description of Selected Tools

Table 6 shows which characteristics are supported by which tool. In this table support is indicated by either 1 or 0, where 1 means that the tool supports this sub-characteristic and 0 means this it is not supported. The total score at the bottom of the table is an arithmetic average of the sub-characteristics per tool.

B. *Selection of Source Code*

The selection of the source code was done according to the criteria set out in the research method section, the following projects were used:

- E-grep [15] project written in C/C++, this is an acronym for *Extended Global Regular Expressions Print*. It is a program which scans a specified file line by line, returning lines that contain a pattern matching a given regular expression.
- Java-DataStructures [16] project written in Java it contains various algorithms for the implementation of the different types of sorting data structures.

Tool characteristics	Tool sub-characteristics	QAC	Understand	CPPdepend	SonarQube	Eclipse Metrics	SourceMonitor
Language	Java (Android)	0	1	1	1	1	1
	C,C++	1	1	1	1	0	1
	Phyton	0	1	0	1	0	0
	C#	0	1	1	1	0	1
	Delphi/Pascal	0	1	0	0	0	1
	Visual Basic	0	1	0	1	0	1
	HTML/ .NET	0	1	1	1	0	1
Availability	Open Source	0	0	0	1	1	0
	Free	0	0	0	0	0	1
	Free-Trial	0	1	1	0	0	0
	Commercial	1	1	1	1	0	0
Interface	Dynamic GUI	1	1	1	1	1	1
	Export Graphs	1	1	1	1	0	1
	Customized Metrics Tables	1	1	1	0	0	0
	Metrics Automation via CMD	1	0	0	0	0	0
Supported operating Systems	Windows	1	1	1	1	1	1
	Mac	1	1	1	1	1	1
	Ubuntu	1	1	1	1	1	1
	Cloud	1	1	1	0	0	0
Integration	IDE	1	1	1	1	1	0
	Continous Integration Tools	1	1	1	1	0	0
	Version Control Tools	1	0	0	1	0	0
	Issue Tracker Tools	1	0	0	1	0	0
Compliance to standards	MISRA	1	1	0	0	0	0
	ISO 26262	1	0	0	0	0	0
	CWE	1	0	0	0	0	0
	CERT	1	0	0	0	0	0
	ISO/IEC 9899:2011	0	0	0	0	0	0
Documentation	Yes	1	1	1	1	1	1
	Tutorials	1	1	1	1	0	0
Metric resolution	Project	1	1	1	1	1	1
	File	1	1	0	1	0	0
	Function/Method	1	1	0	0	0	1
Total Score %		70	73	55	64	27	45

Table 6 Tools Characteristics and Scores

C. *Comparative tests*

Research question 3 concerns to which degree the tools produce consistent results. For this purpose, each of the tools was tested on the selected source and for each metric a measurement was obtained. However, issues regarding naming conventions was a major concern during the testing phase, the names of the metrics vary from tool to tool and they do not necessarily match the names used in the research papers. Out of the 18 validated metrics found only 9 metrics

[15] https://github.com/garyhouston/regexp.old

[16] https://github.com/TheAlgorithms/Java/

were identified and tested. The tools were selected because they stated in their prospects that they support all 18 metrics, but during testing of the trail versions and analysis of the technical documentation it became apparent that only 9 were actually available. This could either be due to incorrect documentation or limitations in the trail versions. Table 7 shows the measurement results from the four selected tools for Java. Source code metrics can typically be measured on entities such as project, file or function/method. For the purpose of this thesis the project entity was selected since it would otherwise be impossible to present any results. Table 7 would at a file level have been a matrix of 648 cells (9 metrics * 18 files * 4 tools = 648). In addition, not all of the tools support representation on file or method level, at least not for the trial versions used for this thesis.

JAVA	Eclipse	Source Monitor	Understand	SonarQube
LCOM	NA	NA	2.38	NA
DIT	1.34	2.09	1.53	NA
RFC	NA	NA	1.12	NA
CBO	NA	NA	2	NA
NOC	7	NA	8	NA
LOC	1310	1310	1310	1328
NOM	1.14	2.82	NA	NA
CC	2.75	2.43	2.43	8.4
FI-FO	NA	3	2.6	NA

Table 7 Average Number of validated Metrics on the Tools for Java

In the same way for C/C++ project out of the 18 validated metrics selected only 2 were found, LOC and CC. See Table 8

C	QA-C	CPPDepend
LOC	2680 (7462)	1183
CC	10.6 (10.6)	9.61

Table 8 Average Number of validated Metrics on the Tools for C
LOC 7642 includes the compiler files.

V. Discussion

If software development departments could to a larger degree base their testing on scientifically validated metrics and only acquire tools that are easy to adopt and use, then an increase in internal software quality could most likely be achieved. The aim of this thesis is therefore to find validated internal measures and tools that support these measures in a consistent manner, while meeting availability criteria such as coverage of programming languages, user interface, supported operating systems, integration capabilities and available documentation. The academic community has proposed a large number of metrics and several of these have also been validated, however, the academic studies on these metrics is somewhat unevenly distributed, some metrics have received much more attention than others. Metrics such as WMC have been studied in 9 different papers, followed by LOCM, DIT, RFC and NOC with 8 and CBO with 7. The other 23 metrics have been studied to a lesser extent.

Unfortunately, the currently available tools either do not support all of the validated metrics or they use names which do not match the ones used in academic papers. This situation is confusing and could indeed slow down the adoption of metric testing. A practitioner that is not academically inclined may well select a tool and start using it and only later find that adapting the code based on measurements from these metrics does little or nothing to improve the quality of software, which may cause them to abandon this type of testing. The tools themselves leave a lot to be desired regarding basic user friendliness. During the testing phase the author faced a considerable number of technical issues and the documentation is often questionable. It requires a lot of time to set up the tool environment and get them to working correctly. Most of them had specific technical requirements for the pieces of code that are to be tested, for instance, some tools were not able to start the analysis without a Build, Cmake or Visual studio project file. Several tools required a specific hardware in order to use their servers to run the static code analysis, however none of this is explicitly mentioned in their documentation. SourceMeter had to be excluded because it was not able to execute on the demonstration code that was included with the installation files, despite following every instruction in detail. Some of the tools require a working build chain in order to function and some do not, which can lead to issues if for instance one source requires VisualStudio10 and another requires VisualStudio15 and they cannot co-exist on the same machine. In summary none of these tools are easy to use and this is a real hurdle to overcome if these tools are to be adopted. There are also big differences between the commercial and free tools, where the commercial tools offer an overwhelming level of detail and the free tools can be somewhat less detailed reports. See Appendix C.

Another issue with the tools is that they do not always support reporting results on the same level. The measurements can be reported on entities such as project, file or function/method level, but not all tools support this. The most relevant level would normally be function/method level since this level can be assigned to a developer or a team for tracking and improvement. To compare the metrics across the tools a project level view had to be adopted since this was the smallest possible denominator. On the positive side the metrics that can be compared. i.e. the metrics that are supported by more than one of the tools showed a fairly good consistency as shown in Table 9 and 10. One exception is Cyclomatic Complexity, where SonarQube has project level complexity of 8.4 and the other tools calculate a complexity of about 2.5. Possibly this is related to how these averages are calculated. For Eclipse, SourceMonitor and Understand the complexity is calculated by the tools.

For SonarQube the average was calculated manually by adding the complexity of each file and dividing by the number of files. It is not clear how the other tools have calculated their complexity. In order to get a better understanding of the differences CC was analysed on a file level and even here there were still differences between the tools, but not as substantial. The maximum complexity for Sonarqube was 16 and the minimum was 1 For Eclipse 12/2, Source Monitor 12/1 and Understand 12/1. This indicates that the average calculation for SonarQube differs from the other tools in some way and that the difference is not mainly caused by different definitions of complexity. The other exception is LOC for CCPDepend, which calculates 1183 lines of code and QAC calculated 2680. A count of the actual lines of code in a text editor showed that the correct LOC is 2680 and not 1183.

JAVA	Average	Standard deviation
DIT	1,715	0,375
NOC	7,500	0,500
LOC	1314,500	7,794
NOM	1,98	0,840
CC	2,8	0,200

Table 9 Average and standard deviation for Java

C	Average	Standard deviation
LOC	1931	748.5
CC	10,105	0,495

Table 10 Average and standard deviation for c

Of the tools tested Understand covers the most metrics, has sufficient documentation, supports both C/C++ and Java and is easy to use. QAC offers the most detailed reports, has good documentation and excellent support, but only supports C/C++. Both of these tools also support project, file and function level views and offer high levels of integration. The results from these two tools are also consistent with each other. The objective score for characteristics presented in table 6 also indicates that these are the two best tools. SourceMonitor is a third option for practitioners that do not need the integration capabilities of Understand and QAC or are not interested in using a commercial tool. In summary QAC and Understand are the two tools that can be highly recommended to practitioners. There is however still room for improvement in both of these tools, since only a portion of the validated metrics are actually supported. Potentially there is a market gap for at tool that actually focuses on metrics that have proper scientific backing. Of the tools that were not recommended CPPdepend has insufficient metric resolution, SonarQube lacks metric support and Eclipse Metrics Plugin lacks metric resolution and integration capabilities. These tools need to address these issues if they tools are to be relevant for practitioners.

VI. Threats of Validity

When performing a comparative case study, validity issues might arise in the collected data whereby certain assumptions made do not stand as true, compromising and possibly invalidating the data. As such, this must be avoided. During this study's data collection process, the following limitations have been considered and addressed as discussed in this section.

A. Internal Validity

Error in underlying papers, the validity of a metric is established in other papers. In theory these results could be incorrect, which could influence the result of this thesis. The threat to validity for a specific metric can be assumed to be lower the more independent validation studies have been conducted. This threat is mitigated by the fact that 80 % of the metrics are supported by 2 or more papers.

Error on the search process, the searching was based on a single indexing system (SCOPUS) where abstract, title and keywords only were considered which could lead to the omission or repetition of papers. This kind of limitation is particularly difficult to tackle, the step taken in this case study to challenge this threat is to ensure the search by using two different search strings at the same time.

Omission of relevant papers, as stated in the research method section during the initial search 567 papers were found, but many of these were not relevant to this thesis as they also included papers about medicine, biochemistry, environmental science, chemistry, agriculture, physics or social science. The reason for these papers being found by the search is presumably that the keywords "metrics", "software" and "validation" are common to many scientific papers. In the second search the subject areas above were excluded and as a result 292 papers were found. After examining the abstracts 13 papers were actually found to be relevant to this thesis. Theoretically there could be a paper were a researcher has looked into validation of a software metric in for instance the chemical industry, but in that case, it would be fair to assume that the author in that case should have marked his research as "SOFT" instead of "CHEM" for SCOPUS. It is also possible that a researcher did validation work on metrics in the software field but omitted this from the abstract. This can be considered to be unlikely. It is also possible that the author missed a paper while looking through the 292 abstracts. There is also a risk that the search strings were incorrectly defined.

B. External Validity

Non-representative source code, if the code selected for this study is not representative of the main population of source codes then the results from this thesis would not be valid in a wider context. This threat is mitigated by choosing a large open source code base. The assumption being that a large source will contain more variation than a small source and should therefore provide a more representative result. Using open source code means that other researchers can

check the results if they were so inclined. The source code size was limited to 100 000 lines due to trial limitations of certain tools. It is theoretically possible that very large and typically commercial source would have given different results.

Bias regarding code selection. In theory there could be a difference in the results between sources from different areas. For example, code written for the military or for medical use might differ from open source code. These differences cannot be evaluated, since no such sources are available.

Bias regarding naming conventions. Unfortunately, each tool can use names for metrics that do not match the names used in academic papers, which leads to a mapping problem, which if done incorrectly could be a threat to validity.

VII. CONCLUSIONS

There are several internal software quality metrics proposed by the research community for facilitating a better design of software. These metrics are supported in a variety of available internal quality measurement tools. While the metrics and their validity are relatively well-documented in the literature, there is little research on which tools are suitable for measurements in terms of cost, availability, integrity, system support, and measurement consistency. This thesis identified validated metrics in the literature, selected a range of tools that support these metrics and tested these tools for the properties stated above. Of the tools tested Understand covers most metrics, has sufficient documentation, supports both C/C++ and Java and is easy to use. QA-C offers the most detailed reports, has good documentation and excellent support, but only supports C/C++. Both of these tools support project, file and function level views and offer high levels of integration. The results from these two tools are consistent with each other. These are the two tools that can be recommended to practitioners. The other tools that were not recommended had various issues.

CPPdepend has insufficient metric resolution, SonarQube lacks metric support and Eclipse Metrics Plugin lacks metric resolution and integration capabilities. These tools need to address these issues if they tools are to be relevant for practitioners. SourceMonitor needs better integration options, but it could still be of interest for practitioners that do not need the integration capabilities of QA-C or Understand and do not want to use a commercial tool. A topic for further study would be to verify the metrics used by the tools in table 6 against the validated metrics in table 4. The tools do not always use the same names for metrics as found in academic papers, which means that the mathematical definitions need to be compared in order to define the number of supported metrics per tool.

REFERENCES

[1] G.G Schulmeyer, J.I. McManus *Handbook of Software Quality Assurance* (2nd ed.). 1992. . Van Nostrand Reinhold Co., New York, NY, USA.

[2] N. G. Leveson and C. S. Turner. 1993. An Investigation of the Therac-25 Accidents. *Computer* 26, 7 (July 1993), 18-4.

[3] ISO/IEC 9126-1:2001 Software engineering - Product quality. Web https://www.iso.org/standard/22749.html

[4] V. R. Basili, L. C. Briand and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," in *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751-761, Oct 1996.

[5] M. Santos, P. Afonso, P. H. Bermejo and H. Costa, "Metrics and statistical techniques used to evaluate internal quality of object-oriented software: A systematic mapping," *2016 35th International Conference of the Chilean Computer Science Society (SCCC)*, Valparaíso, 2016, pp. 1-11.

[6] A. B. Carrillo, P. R. Mateo and M. R. Monje, "Metrics to evaluate functional quality: A systematic review," *7th Iberian Conference on Information Systems and Technologies (CISTI 2012)*, Madrid, 2012, pp. 1-6.

[7] R. Lincke, J. Lundberg, and W. Löwe. 2008. Comparing software metrics tools. In *Proceedings of the 2008 international symposium on Software testing and analysis* (ISSTA '08).

[8] L. C. Briand, S. Morasca and V. R. Basili, "Property-based software engineering measurement," in *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68-86, Jan 1996.

[9] MJ. Ordonez, H.M. Haddad." The State of Metrics in Software Industry". Fifth International Conference on Information Technology: New Generations, April 2008 Page(s):453 - 458

[10] M. Lanza, Marinescu, R., 2016. "Object Oriented Metrics in Practice". Springer Berlin Heidelberg, Berlin, Heidelberg.

[11] A. Nunez-Varela, H. Perez-Gonzales, J.C. Cuevas-Trello, Soubervielle-Montalvo, "A methodology for Obtaining Universal Software Code Metrics". The 2013 Iberoamerican Conference on Electronics Engineering and Computer Science. Procedia Technology 7(2013)336-343.

[12] A. Nuñez-Varela, Pérez-Gonzalez, Héctor G., Martínez-Perez, Francisco E., Soubervielle-Montalvo, Carlos, "Source code metrics: A systematic mapping study", Journal of Systems and Software 1281641972017 2017/06/01/ 0164-1212.

[13] Shepperd, M. J. & Ince, D., 1993. Derivation and Validation of Software Metrics. Clarendon Press, Oxford, UK.

[14] N. Fenton, S. L. Pfleeger, 1977. Software Metrics, A Rigorous and Practical Approach. 2nd ed. International Thomson Computer Press.

[15] Saraiva, J. de A.G, de França, Micael S., Soares, Sérgio C.B., Filho, Fernando J.C.L., Souza, Renata M.C.R., 2015. "Classifying metrics for assessing Object-Oriented Software Maintainability: A family of metrics catalogs". Journal of Systems and Software Vol 13, Pages 85-101. Informatics Center, Federal University of Pernambuco, Brasil.

[16] T. Gyimothy, R. Ferenc and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," in *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897-910, Oct. 2005.

[17] M. S. Krishnan, R. Subramanyam, "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects," in *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297-310, April 2003.

[18] S. Yeresime,, J. Pati,, S. Rath,, "Effectiveness of Software Metrics for Object-oriented System", Procedia Technology 6-420- 427- 2012-2012/01/01/- 2nd International Conference on Communication, Computing & Security [ICCCS-2012]- 2212-0173.

[19] S. Jabangwe, J. Börstler, D. Šmite,. et al. Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review (2015) 20: 640. https://doi.org/10.1007/s10664-013-9291-7.

[20] J. Ludwig, S. Xu and F. Webber, "Compiling static software metrics for reliability and maintainability from GitHub repositories," *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Banff, AB, 2017, pp. 5-9.

[21] W. Li, S. Henry. "Object-oriented metrics that predict maintainability". Journal of Systems and Software, Volume 23, Issue 2, 1993. Pages 111-122. ISSN 0164-1212.

[22] Antinyan, V., Staron, M., Sandberg, A., "Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time",Empirical Software Engineering, 2017, Dec 01. Volume 22, 6. Pages 3057── 3087.

[23] T. J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Softw. Eng.* 2, 4 (July 1976), 308-320.

[24] Halstead MH (1977) Elements of Software Science (Operating and programming systems series). Elsevier Science Inc.

[25] Henry S, Kafura D (1981) Software structure metrics based on information flow. IEEE Trans Softw Eng 5:510–518

[26] Chidamber SR, Kemerer CF (1994) A metrics suite for object-oriented design. IEEE Trans Softw Eng 20(6):476–493

[27] Antinyan V et al. (2014) Identifying risky areas of software code in Agile/Lean software development: An industrial experience report. 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering, (CSMR-WCRE), IEEE.

[28] Tenny T (1988) Program readability: Procedures versus comments. IEEE Trans Softw Eng 14(9):1271–1279

[29] Buse RP, Weimer WR (2010) Learning a metric for code readability. IEEE Trans Softw Eng 36(4):546–558

[30] J. Al Dallal and L. C. Briand, "A Precise Method-Method Interaction Based Cohesion Metric for Object-Oriented Classes," ACM Trans. Softw. Eng. Methodol., vol. 21, no. 2, pp. 1–34, 2012.

[31] M. Alzahrani and A. Melton, "Defining and Validating a Client-Based Cohesion Metric for Object-Oriented Classes," *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Turin, 2017, pp. 91-96.

[32] Olague, H. M., Etzkorn, L. H., Messimer, S. L. and Delugach, H. S. (2008), An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile, software: a case study. J. Softw. Maint. Evol.: Res. Pract., 20: 171-197.

[33] J Al Dallal. (2010) Mathematical validation of object-oriented class cohesion metrics. International Journal of Computers, 4 (2) (2010), pp. 45-52 .

[34] J. M. Bieman and B. Kang, Cohesion and reuse in an object-oriented system, Proceedings of the 1995 Symposium on Software reusability, Seattle, Washington, United States, pp. 259-262, 1995

[35] L. C. Briand, J. Daly, and J. Wuest, A unified framework for cohesion measurement in object-oriented systems, Empirical Software Engineering - An International Journal, Vol. 3, No. 1, 1998, pp. 65117.

[36] Y. Zhou, B. Xu, J. Zhao, and H. Yang, ICBMC: An improved cohesion measure for classes, Proc. of International Conference on Software Maintenance, 2002, pp. 44-53.

[37] J. Alghamdi, Measuring software coupling, Proceedings of the 6th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems, p.6-12, February 16-19, 2007, Corfu Island, Greece.

[38] D. Nicolette, (2015). *Software development metrics*. Page 90.

[39] S. Freeman, N. Pryce. 2009. *Growing Object-Oriented Software, Guided by Tests* (1st ed.). Addison-Wesley Professional. Page 10.

Appendix A

Language	Tools
Multi Language (48)	APPscreener,Application Inspector, Axivion Bauhaus Suite, CAST, Checkmarx,Cigital , CM evolveIT, Code Dx , Compuware, ConQAT, Coverity , DefenseCode ThunderScan, Micro Focus, Gamma, GrammaTech, IBM Security AppScan, Facebook Infer , Imagix 4D, Kiuwan, Klocwork, LDRA Testbed, MALPAS, Moose, Parasoft, Copy/Paste Detector (CPD), Polyspace, Pretty Diff, Protecode, PVS-Studio, RSM, Rogue Wave Software, Semmle, SideCI , Silverthread, SnappyTick (SAST), SofCheck Inspector, Sonargraph, SonarQube, Sotoarc, SourceMeter, SQuORE, SPARROW, Understand, Veracode, Yasca, Application Analyzer, CodeMR.
.NET (9)	.NETCompilerPlatform, CodeIt.Right, CodePush, Designite, FXCop, NDepend, Parasoft, Sonargraph, StyleCop
Ada (8)	sPARK Toolset, AdaConstrol, CodePeer, Fluctuat, LDRA Testbed, Polypace, SoftCheck Inspector
C, C++ (25)	AdLint, Astreé, Axivion Bauhaus Suite, BLAST, Cppcheck, cpplint, Clang, Coccinelle, Coverity, Cppdepend, ECLAIR, Eclipse, Flawfinder, Fluctuat, Frama-C, Goanna, Infer, Lint, PC-Lint, Polyspace, PRQA QA C, SLAMproject, Sparse, Splint, Visual Studio
Java (16)	Checkstyle, ErrorProne, Findbugs, Infer, Intellij IDEA, Jarchitect, Jtest,PMD,SemmleCode, Sonargraph, Sonargraph Explorer, Soot, Spoon, Squale, SourceMeter, ThreadSafe, Xanitizer
JAvaScript (6)	DeepScan, StandrdJS, ESLint, Google Closure Compiler, JSHint, JSLint
Perl (5)	Perl-Critic, Devel:Cover, PerlTidy, Padre, Kritika
PHP (4)	Progpilot, PHPPMD, RIPS, Phlint
Phyton (5)	Bandit, PyCharm, PyChecker, Pyflackes, Pylint
Ruby (4)	Flay, Flog, Reek, RuboCop

Appendix B

SourceMonitor	
Measures Name	**Measure definition**
Number Of Files	Number of Files
Number of Lines of Code	Number of code lines of the method, with or without including empty lines (Specificatins are done when creating the project)
Number of Statements	Number of statement of the code
Porcentage of Branches	Number of Branches of the code
Number of Calls	Number of calls performed in the code

Number of Classes	Number of classes defined
Number of Methods/Class	Number of methods and classes
Average Statements/ Methods	Average number of statements and methods
Max Complexity	Maximal complexity
Max Depth	Maximal depth of a branch
Average Depth	Average depth of a branch
Average Complexity	Average Complexity

UNDERSTAND		
Measure ID	**Measures Name**	**Measure definition**
AltAvgLineBlank	Average Number of Blank Lines (Include Inactive)	Average number of blank lines for all nested functions or methods, including inactive regions.
AltAvgLineCode	Average Number of Lines of Code (Include Inactive)	Average number of lines containing source code for all nested functions or methods, including inactive regions.
AltAvgLineComment	Average Number of Lines with Comments (Include Inactive)	Average number of lines containing comment for all nested functions or methods, including inactive regions.
AltCountLineBlank	Blank Lines of Code (Include Inactive)	Number of blank lines, including inactive regions.
AltCountLineCode	Lines of Code (Include Inactive)	Number of lines containing source code, including inactive regions.
AltCountLineComment	Lines with Comments (Include Inactive)	Number of lines containing comment, including inactive regions.
AvgCyclomatic	Average Cyclomatic Complexity	Average cyclomatic complexity for all nested functions or methods.
AvgCyclomaticModified	Average Modified Cyclomatic Complexity	Average modified cyclomatic complexity for all nested functions or methods.
AvgCyclomaticStrict	Average Strict Cyclomatic Complexity	Average strict cyclomatic complexity for all nested functions or methods.
AvgEssential	Average Essential Cyclomatic Complexity	Average Essential complexity for all nested functions or methods.
AvgEssentialStrictModified	Average Essential Strict Modified Complexity	Average strict modified essential complexity for all nested functions or methods.
AvgLine	Average Number of Lines	Average number of lines for all nested functions or methods.
AvgLineBlank	Average Number of Blank Lines	Average number of blank for all nested functions or methods.
AvgLineCode	Average Number of Lines of Code	Average number of lines containing source code for all nested functions or methods.
AvgLineComment	Average Number of Lines with Comments	Average number of lines containing comment for all nested functions or methods.
CountClassBase	Base Classes	Number of immediate base classes. [aka IFANIN]
CountClassCoupled	Coupling Between Objects	Number of other classes coupled to. [aka CBO (coupling between object classes)]
CountClassDerived	Number of Children	Number of immediate subclasses. [aka NOC (number of children)]
CountDeclClass	Classes	Number of classes.
CountDeclClassMethod	Class Methods	Number of class methods.
CountDeclClassVariable	Class Variables	Number of class variables.
CountDeclFile	Number of Files	Number of files.
CountDeclFunction	Function	Number of functions.
CountDeclInstanceMethod	Instance Methods	Number of instance methods. [aka NIM]
CountDeclInstanceVariable	Instance Variables	Number of instance variables. [aka NIV]
CountDeclInstanceVariableInternal	Internal Instance Variables	Number of internal instance variables.
CountDeclInstanceVariablePrivate	Private Instance Variables	Number of private instance variables.
CountDeclInstanceVariableProtected	Protected Instance Variables	Number of protected instance variables.

CountDeclInstanceVariableProtectedInternal	Protected Internal Instance Variables	Number of protected internal instance variables.
CountDeclInstanceVariablePublic	Public Instance Variables	Number of public instance variables.
CountDeclMethod	Local Methods	Number of local methods.
CountDeclMethodAll	Methods	Number of methods, including inherited ones. [aka RFC (response for class)]
CountDeclMethodConst	Local Const Methods	Number of local const methods.
CountDeclMethodDefault	Local Default Visibility Methods	Number of local default methods.
CountDeclMethodFriend	Friend Methods	Number of local friend methods. [aka NFM]
CountDeclMethodInternal	Local Internal Methods	Number of local internal methods.
CountDeclMethodPrivate	Private Methods	Number of local private methods. [aka NPM]
CountDeclMethodProtected	Protected Methods	Number of local protected methods.
CountDeclMethodProtectedInternal	Local Protected Internal Methods	Number of local protected internal methods.
CountDeclMethodPublic	Public Methods	Number of local public methods. [aka NPRM]
CountDeclMethodStrictPrivate	Local strict private methods	Number of local strict private methods.
CountDeclMethodStrictPublished	Local strict published methods	Number of local strict published methods.
CountDeclModule	Modules	Number of modules.
CountDeclProgUnit	Program Units	Number of non-nested modules, block data units, and subprograms.
CountDeclProperty	Properties	Number of properties.
CountDeclPropertyAuto	Auto Implemented Properties	Number of auto-implemented properties.
CountDeclSubprogram	Subprograms	Number of subprograms.
CountInput	Inputs	Number of calling subprograms plus global variables read. [aka FANIN]
CountLine	Physical Lines	Number of all lines. [aka NL]
CountLineBlank	Blank Lines of Code	Number of blank lines. [aka BLOC]
CountLineBlank_Html	Blank html lines	Number of blank html lines.
CountLineBlank_Javascript	Blank javascript lines	Number of blank javascript lines.
CountLineBlank_Php	Blank php lines	Number of blank php lines.
CountLineCode	Source Lines of Code	Number of lines containing source code. [aka LOC]
CountLineCodeDecl	Declarative Lines of Code	Number of lines containing declarative source code.
CountLineCodeExe	Executable Lines of Code	Number of lines containing executable source code.
CountLineCode_Javascript	Javascript source code lines	Number of javascript lines containing source code.
CountLineCode_Php	PHP Source Code Lines	Number of php lines containing source code.
CountLineComment	Lines with Comments	Number of lines containing comment. [aka CLOC]
CountLineComment_Html	HTML Comment Lines	Number of html lines containing comment.
CountLineComment_Javascript	Javascript Comment Lines	Number of javascript lines containing comment.
CountLineComment_Php	PHP Comment Lines	Number of php lines containing comment.
CountLineInactive	Inactive Lines	Number of inactive lines.
CountLinePreprocessor	Preprocessor Lines	Number of preprocessor lines.
CountLine_Html	HTMLLines	Number of all html lines.
CountLine_Javascript	Javascript Lines	Number of all javascript lines.
CountLine_Php	PHP Lines	Number of all php lines.
CountOutput	Outputs	Number of called subprograms plus global variables set. [aka FANOUT]
CountPackageCoupled	Coupled Packages	Number of other packages coupled to.
CountPath	Paths	Number of possible paths, not counting abnormal exits or gotos. [aka NPATH]

CountPathLog	Paths (Log10x)	Log10, truncated to an integer value, of the metric CountPath
CountSemicolon	Semicolons	Number of semicolons.
CountStmt	Statements	Number of statements.
CountStmtDecl	Declarative Statements	Number of declarative statements.
CountStmtDecl_Javascript	Javascript Declarative Statements	Number of javascript declarative statements.
CountStmtDecl_Php	PHP Declarative Statements	Number of php declarative statements.
CountStmtEmpty	Empty Statements	Number of empty statements.
CountStmtExe	Executable Statements	Number of executable statements.
CountStmtExe_Javascript	Javascript Executable Statements	Number of javascript executable statements.
CountStmtExe_Php	PHP Executable Statements	Number of php executable statements.
Cyclomatic	Cyclomatic Complexity	Cyclomatic complexity.
CyclomaticModified	Modified Cyclomatic Complexity	Modified cyclomatic complexity.
CyclomaticStrict	Strict Cyclomatic Complexity	Strict cyclomatic complexity.
Essential	Essential Complexity	Essential complexity. [aka Ev(G)]
EssentialStrictModified	Essential Strict Modified Complexity	Strict Modified Essential complexity.
Knots	Knots	Measure of overlapping jumps.
MaxCyclomatic	Max Cyclomatic Complexity	Maximum cyclomatic complexity of all nested functions or methods.
MaxCyclomaticModified	Max Modified Cyclomatic Complexity	Maximum modified cyclomatic complexity of nested functions or methods.
MaxCyclomaticStrict	Max Strict Cyclomatic Complexity	Maximum strict cyclomatic complexity of nested functions or methods.
MaxEssential	Max Essential Complexity	Maximum essential complexity of all nested functions or methods.
MaxEssentialKnots	Max Knots	Maximum Knots after structured programming constructs have been removed.
MaxEssentialStrictModified	Max Essential Strict Modified Complexity	Maximum strict modified essential complexity of all nested functions or methods.
MaxInheritanceTree	Depth of Inheritance Tree	Maximum depth of class in inheritance tree. [aka DIT]
MaxNesting	Nesting	Maximum nesting level of control constructs.
MinEssentialKnots	Minimum Knots	Minimum Knots after structured programming constructs have been removed.
PercentLackOfCohesion	Lack of Cohesion in Methods	100% minus the average cohesion for package entities. [aka LCOM, LOCM]
RatioCommentToCode	Comment to Code Ratio	Ratio of comment lines to code lines.
SumCyclomatic	Sum Cyclomatic Complexity	Sum of cyclomatic complexity of all nested functions or methods. [aka WMC]
SumCyclomaticModified	Sum Modified Cyclomatic Complexity	Sum of modified cyclomatic complexity of all nested functions or methods.
SumCyclomaticStrict	Sum Strict Cyclomatic Complexity	Sum of strict cyclomatic complexity of all nested functions or methods.
SumEssential	Sum Essential Complexity	Sum of essential complexity of all nested functions or methods.
SumEssentialStrictModified	Sum Essential Strict Modified Complexity	Sum of strict modified essential complexity of all nested functions or methods.

Eclipse Metrics Plugin	
Measures Name	**Measure definition**
Number of Classes	Total number of classes in the selected scope
Number of Children	Total number of direct subclasses of a class. A class implementing an interface counts as a direct child of that interface
Number of Interfaces	Total number of interfaces in the selected scope
Depth of Inheritance Tree (DIT)	Distance from class Object in the inheritance hierarchy.

Number of Overridden Methods (NORM)	Total number of methods in the selected scope that are overridden from an ancestor class. Here you can control whether to count abstract methods, methods that call the inherited implementation (through use of super.[same-method] call). Certain methods that are supposed to be overridden can be excluded explicitly (like toString, equals and hashCode).
Number of Methods (NOM)	Total number of methods defined in the selected scope
Number of Fields	Total number of fields defined in the selected scope
Lines of Code	since version 1.3.6 Lines of code has been changed and separated into: TLOC: Total lines of code that will counts non-blank and non-comment lines in a compilation unit. usefull for thoses interested in computed KLOC. MLOC: Method lines of code will counts and sum non-blank and non-comment lines inside method bodies
Specialization Index	Average of the specialization index, defined as NORM * DIT / NOM. This is a class level metric
McCabe Cyclomatic Complexity	Counts the number of flows through a piece of code. Each time a branch occurs (if, for, while, do, case, catch and the ?: ternary operator, as well as the && and || conditional logic operators in expressions) this metric is incremented by one. Calculated for methods only. For a full treatment of this metric see McCabe.
Weighted Methods per Class (WMC)	Sum of the McCabe Cyclomatic Complexity for all methods in a class
Lack of Cohesion of Methods (LCOM*)	A measure for the Cohesiveness of a class. Calculated with the Henderson-Sellers method (LCOM*, see page 147). If (m(A) is the number of methods accessing an attribute A, calculate the average of m(A) for all attributes, subtract the number of methods m and divide the result by (1-m). A low value indicates a cohesive class and a value close to 1 indicates a lack of cohesion and suggests the class might better be split into a number of (sub)classes. I'm unsure of the usefullness of this metric in Java since it penalizes the proper use of getters and setters as the only methods that directly access an attribute and the other methods using the gettter/setter methods. Perhaps I could alter the implementation to take this into account, assuming standard JavaBean naming conventions.
Afferent Coupling (Ca)	The number of classes outside a package that depend on classes inside the package.
Efferent Coupling (Ce)	The number of classes inside a package that depend on classes outside the package.
Instability (I)	Ce / (Ca + Ce)
Abstractness (A)	The number of abstract classes (and interfaces) divided by the total number of types in a package
Normalized Distance from Main Sequence (Dn	| A + I - 1 |, this number should be small, close to zero for good packaging design.
Design Size | Design Size in Class	Total number of source classes.
Hierarchies | Number of Hierarchies (NOH)	A count of the number of class hierarchies in the design.
Abstraction | Average Number of Ancestors (ANA)	The average number of classes from which each class inherits information.
Encapsulation | Data Access Metrics (DAM)	The ratio of the number of private (protected) attributes to the total number of attributes declared in the class. Interpreted as the average across all design classes with at least one attribute, of the ratio of non-public to total attributes in a class.
Coupling | Direct Class Coupling (DCC)	A count of the different number of classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods. Interpreted as an average over all classes when applied to a design as a whole; a count of the number of distinct user-defined classes a class is coupled to by method parameter or attribute type. The java.util.Collection classes are counted as user-defined classes if they represent a collection of a user-defined class.
Cohesion | Cohesion Among Methods in Class (CAM)	Represents the relatedness among methods of a class, computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class. Constructors and static methods are excluded.
Composition | Measure of Aggregation (MOA)	A count of the number of data declarations whose types are user-defined classes. Interpreted as the average value across all design classes. We define 'user defined classes' as non-primitive types that are not included in the Java standard libraries and collections of user-defined classes from the java.util.collections package.
Inheritance | Measure of Functional Abstraction (MFA)	A ratio of the number of methods inherited by a class to the number of methods accessible by member methods of the class. Interpreted as the average across all classes in a design of the ratio of the number of methods inherited by a class to the total number of methods available to that class, i.e. inherited and defined methods.
Polymorphism | Number of Polymorphic Methods (NOP)	The count of the number of the methods that can exhibit polymorphic behaviour. Interpreted as the average across all classes, where a method can exhibit polymorphic behaviour if it is overridden by one or more descendent classes.
Messaging | Class Interface Size (CIS)	A count of the number of public methods in a class. Interpreted as the average across all classes in a design.

Complexity | Number of Methods (NOM)	A count of all the methods defined in a class. Interpreted as the average across all classes in a design.

SonarQube	
Measures Name	**Measure definition**
Complexity	It is the complexity calculated based on the number of paths through the code. Whenever the control flow of a function splits, the complexity counter gets incremented by one. Each function has a minimum complexity of 1. This calculation varies slightly by language because keywords and functionalities do.
Cognitive Complexity	How hard it is to understand the code's control flow. See https://www.sonarsource.com/resources/white-papers/cognitive-complexity.html for complete description of the mathematical model applied to compute this measure.
Duplicated blocks	Number of duplicated blocks of lines. For a block of code to be considered as duplicated:Non-Java projects:There should be at least 100 successive and duplicated tokens. 30 lines of code for COBOL 10 lines of code for other languages Java projects: There should be at least 10 successive and duplicated statements whatever the number of tokens and lines. Differences in indentation as well as in string literals are ignored while detecting duplications.
Duplicated files	Number of files involved in duplications.
Duplicated lines	Number of lines involved in duplications.
Duplicated lines (%)	Density of duplication = Duplicated lines / Lines * 100
New issues	Number of new issues.
False positive issues	Number of false positive issues
Open issues	Number of issues whose status is Open
Confirmed issues	Number of issues whose status is Confirmed
Reopened issues	Number of issues whose status is Reopened
Code Smells	Number of code smells.
New Code Smells	Number of new code smells.
Maintainability Rating (formerly SQALE Rating)	Rating given to your project related to the value of your Technical Debt Ratio. The default Maintainability Rating grid is:A=0-0.05, B=0.06-0.1, C=0.11-0.20, D=0.21-0.5, E=0.51-1 The Maintainability Rating scale can be alternately stated by saying that if the outstanding remediation cost is: <=5% of the time that has already gone into the application, the rating is A between 6 to 10% the rating is a B between 11 to 20% the rating is a C between 21 to 50% the rating is a D anything over 50% is an
Technical Debt	Effort to fix all maintainability issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days.
Technical Debt on new code	Technical Debt of new code
Technical Debt Ratio	Ratio between the cost to develop the software and the cost to fix it. The Technical Debt Ratio formula is: Remediation cost / Development cost Which can be restated as: Remediation cost / (Cost to develop 1 line of code * Number of lines of code) The value of the cost to develop a line of code is 0.06 days.
Technical Debt Ratio on new code	Ratio between the cost to develop the code changed in the leak period and the cost of the issues linked to it.
Bugs	Number of bugs.
New Bugs	Number of new bugs. A = 0 Bug B = at least 1 Minor Bug C = at least 1 Major Bug D = at least 1 Critical Bug E = at least 1 Blocker Bug
Reliability remediation effort	Effort to fix all bug issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days.
Reliability remediation effort on new code	Same as *Reliability remediation effort* by on the code changed in the leak period.
New Vulnerabilities	Number of new vulnerabilities.
Security Rating	A = 0 Vulnerability B = at least 1 Minor Vulnerability C = at least 1 Major Vulnerability D = at least 1 Critical Vulnerability E = at least 1 Blocker Vulnerability
Security remediation effort	Effort to fix all vulnerability issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days.
Security remediation effort on new code	Same as Security remediation effort by on the code changed in the leak period.
Comments (%)	Density of comment lines = Comment lines / (Lines of code + Comment lines) * 100
Directories	Number of directories.
Files	Number of files.
Lines	Number of physical lines (number of carriage returns).

Lines of code	Number of physical lines that contain at least one character which is neither a whitespace nor a tabulation nor part of a comment.
Lines of code per language	Non Commenting Lines of Code Distributed By Language
Functions	Number of functions. Depending on the language, a function is either a function or a method or a paragraph.
Projects	Number of projects in a view.
Statements	Number of statements.
Classes	Number of classes (including nested classes, interfaces, enums and annotations).
Comment lines	Number of lines containing either comment or commented-out code. Non-significant comment lines (empty comment lines, comment lines containing only special characters, etc.) do not increase the number of comment lines. The following piece of code contains 9 comment lines:
Condition coverage	Condition coverage = (CT + CF) / (2*B) whereCT = conditions that have been evaluated to 'true' at least onceCF = conditions that have been evaluated to 'false' at least onceB = total number of conditions
Condition coverage on new code	Identical to Condition coverage but restricted to new / updated source code.
Condition coverage hits	List of covered conditions.
Conditions by line	Number of conditions by line.
Covered conditions by line	Number of covered conditions by line.
Coverage	Coverage = (CT + CF + LC)/(2*B + EL) where CT = conditions that have been evaluated to 'true' at least once CF = conditions that have been evaluated to 'false' at least once LC = covered lines = lines_to_cover - uncovered_lines B = total number of conditions EL = total number of executable lines (lines_to_cover)
Coverage on new code	Identical to Coverage but restricted to new / updated source code.
Line coverage	Line coverage = LC / EL where LC = covered lines (lines_to_cover - uncovered_lines) EL = total number of executable lines (lines_to_cover)
Line coverage on new code	Identical to Line coverage but restricted to new / updated source code.
Line coverage hits	List of covered lines.
Lines to cover	Number of lines of code which could be covered by unit tests (for example, blank lines or full comments lines are not considered as lines to cover).
Lines to cover on new code	Identical to Lines to cover but restricted to new / updated source code.
Skipped unit tests	Number of skipped unit tests.
Uncovered conditions	Number of conditions which are not covered by unit tests.
Uncovered conditions on new code	Identical to Uncovered conditions but restricted to new / updated source code.
Uncovered lines	Number of lines of code which are not covered by unit tests.
Uncovered lines on new code	Identical to Uncovered lines but restricted to new / updated source code.
Unit tests	Number of unit tests.
Unit tests duration	Time required to execute all the unit tests.
Unit test errors	Number of unit tests that have failed.
Unit test failures	Number of unit tests that have failed with an unexpected exception.
Unit test success density (%)	Test success density = (Unit tests - (Unit test errors + Unit test failures)) / Unit tests * 100

QA-C

Measure ID	Measure Name	Measure Description
STAKI	Akiyama's Criterion	This metric is the sum of the cyclomatic complexity (STCYC) and the number of function calls (STSUB). Although this is not an independent metric, it is included on account of its use in documented case histories. See Akiyama10 and Shooman11 for more details. The metric is calculated as: STAKI = STCYC + STSUB
STAV1	Average Size of Statement in Function (variant 1)	These metrics (STAV1, STAV2, and STAV3) measure the average number of operands and operators per statement in the body of the function. They are calculated as follows: STAVx = (N1 + N2) / number of statements in the function where N1 is Halstead's number of operator occurrences. N2 is Halstead's number of operand occurrences. The STAVx metrics are computed using STST1, STST2 and STST3 to represent the number of statements in a function. Hence there are three variants: STAV1, STAV2 and STAV3, relating to the respective statement count metrics. This metric is used to detect components with long statements. Statements comprising a large number of textual elements (operators and operands) require more effort by the reader in order to understand them. This metric is a good indicator of the program's readability. Metric values are computed as follows: STAV1 = (STFN1 + STFN2)

STAV2	Average Size of Statement in Function (variant 2)	See above STAV2 = (STFN1 + STFN2) / STST2
STAV3	Average Size of Statement in Function (variant 3)	See above STST2 STAV3 = (STFN1 + STFN2) / STST3
STBAK	Number of Backward Jumps	Jumps are never recommended and backward jumps are particularly undesirable. If possible, the code should be redesigned to use structured control constructs such as while or for instead of goto. It differs from the metric STSUB, in that only distinct functions are counted (multiple instances of calls to a particular function are counted as one call), and also that functions called via pointers are not counted
STCAL	Number of Functions Called from Function	This metric counts the number of function calls in a function. It differs from the metric STSUB, in that only distinct functions are counted (multiple instances of calls to a particular function are counted as one call), and also that functions called via pointers are not counted
STCYC	Cyclomatic Complexity	Cyclomatic complexity is calculated as the number of decisions plus 1. High cyclomatic complexity indicates inadequate modularization or too much logic in one function. Software metric research has indicated that functions with a cyclomatic complexity greater than 10 tend to have problems related to their complexity. Some metrication tools include use of the ternary operator ? : when calculating cyclomatic complexity. It could also be argued that use of the && and ||operators should be included. Instead, STCYC calculation is based on statements alone.
STELF	Number of Dangling Else-Ifs	This is the number of if-else-if constructs that do not end in an else clause. This metric is calculated by counting all if statements that do not have a corresponding else and for which QA·C issues a warning 2004. STELF provides a quick reference allowing monitoring of these warnings.
STFDN	Number of distinct operands in a Function	This metric is Halstead's distinct operand count on a function basis (DN). STFDN is related to STFN2, STOPN and STM20: all of these metrics count 'operands', the difference is summarized as: STFN2 Counts ALL operands in the function body STFDN Counts DISTINCT operands in the function body STM20 Counts ALL operands in the file STOPN Counts DISTINCT operands in the file
STFDT	Number of distinct operators in a Function	ThismetricisHalstead'sdistinctoperatorcountonafunctionbasis(DT).STFDTisrelated to STFN1, STOPT and STM21: all of these metrics count 'operators', the difference is summarized as: STFN1 Counts ALL operators in the function body STFDT Counts DISTINCT operators in the function body STM21 Counts ALL operators in the file STOPT Counts DISTINCT operators in the file
STFN1	Number of Operator Occurrences in Function	This metric is Halstead's operator count on a function basis (N1). STFN1 is related to STFDT, STOPT and STM21: all of these metrics count 'operators', the difference is summarized as: STFN1 Counts ALL operators in the function body STFDT Counts DISTINCT operators in the function body STM21 Counts ALL operators in the file STOPT Counts DISTINCT operators in the file
STFN2	Number of Operand Occurrences in Function	This metric is Halstead's operand count on a function basis (N2). STFN2 is related to STFDN, STOPN and STM20: all of these metrics count 'operands', the difference is summarized as: STFN2 Counts ALL operands in the function body STFDN Counts DISTINCT operands in the function body STM20 Counts ALL operands in the file STOPN Counts DISTINCT operands in the file
STGTO	Number of Goto statements	Some occurrences of goto simplify error handling. However, they should be avoided whenever possible. According to the Plum Hall Guidelines, goto should not be used.
STKDN	Knot Densit	This is the number of knots per executable line of code. The metric is calculated as: STKDN = STKNT / STXLN The value is computed as zero when STXLN is zero.
STKNT	Knot Count	This is the number of knots in a function. A knot is a crossing of control structures, caused by an explicit jump out of a control structure either by break, continue, goto, or return. STKNT is undefined for functions with unreachable code. This metric measures knots, not by counting control structure crossings, but by counting the following keywords: • goto statements, • continue statements within loop statements, • break statements within loop or switch statements, except those at top switch level, • all return statements except those at top function level.
STLCT	Number of Local Variables Declared	This is the number of local variables of storage class auto, register, or static declared in a function. Thesearevariablesthatthathavenolinkage
STLIN	Number of Code Lines	This is the total number of lines, including blank and comment lines, in a function definition between (but excluding) the opening and closing brace of the function body. It is computed on raw code. STLIN is undefined for functions which have #include'd code or macros which include braces in their definition. Long functions are difficult to read, as they do not fit on one screen or one listing page. An upper limit of 200 is recommended.
STLOP	Number of Logical Operators	This is the total number of logical operators (&&, ||) in the conditions of do-while, for, if, switch, orwhile statementsinafunction
STM07	Essential Cyclomatic Complexity	The essential cyclomatic complexity is obtained in the same way as the cyclomatic complexity but is based on a 'reduced' control flow graph. The purpose of reducing a graph is to check that the component complies with the rules of structured programming. A control graph that can be reduced to a graph whose cyclomatic complexity is 1 is said to be structured. Otherwise reduction will show elements of the control graph which do not comply with the rules of structured programming. The principle of control graph reduction is to simplify the most deeply nested control subgraphsinto asingle reducedsubgraph. A subgraph isa sequenceof nodeson thecontrol flowgraphwhichhasonlyoneentryandexitpoint. FourcasesareidentifiedbyMcCabe13 which result in an unstructured control graph. These are: a branch into a decision structure, • a branch from inside a decision structure, • a branch into a loop structure, • a branch from inside a loop structure. However, if a subgraph possesses multiple entry or exit points then it cannot be reduced. The use of multiple entry and exit points breaks the most fundamental rule of structured programming.
STM19	Number of Exit Points	This metric is a measure of the number of exit points in a software component and is calculated by counting the number of return statements. A function that has no return statements will have an STM19 value of zero even though it will exit when falling through the last statement. This is regardless of whether the function is declared to have a return value or not (i.e. returns void). Calls to non-returning functions such as exit() or abort() are ignored by this metric.
STM29	Number of Functions Calling this Function	This metric is defined as the number of functions calling the designated function. The number of calls to a function is an indicator of criticality. The more a function is called, the more critical it is and, therefore, the more reliable it should be.

STMCC	Myer's Interval	This is an extension to the cyclomatic complexity metric. It is expressed as a pair of numbers, conventionally separated by a colon. Myer's Interval is defined as STCYC : STCYC + L. Cyclomatic complexity (STCYC) is a measure of the number of decisions in the control flow of a function. L is the value of the QA·C STLOP metric which is a measure of the number of logical operators (&&, ||) in the conditional expressions of a function. A high value of L indicates that there are many compound decisions, which makes the code more difficult to understand. A Myer's interval of 10 is considered very high. When exporting metric values or displaying in the Metrics Browser, rather than attempting to display a value pair, the value of L is chosen for STMCC.
STMIF	Deepest Level of Nesting	This metric is a measure of the maximum control flow nesting in your source code. You can reduce the value of this metric by turning your nesting into separate functions. This will improve the readability of the code by reducing both the nesting and the average cyclomatic complexity per function. STMIFisincrementedinswitch,do,while,if andfor statements. Thenestinglevelofcode is not always visually apparent from the indentation of the code. In particular, an else if construct increases the level of nesting in the control flow structure, but is conventionally written without additional indentation.
STPAR	Number of Function Parameters	This metric counts the number of declared parameters in the function argument list. Note that ellipsis parameters are ignored.
STPBG	Residual Bugs (STPTH-based est.)	Hopkins, in Hatton & Hopkins14 investigated software with a known audit history and observed a correlation between Static Path Count (STPTH) and the number of bugs that had been found. This relationship is expressed as STPBG. STPBG = log10 (STPTH)
STPDN	Path Density	This is a measure of the number of paths relative to the number of executable lines of code. STPDN = STPTH / STXLN. STPDN is computed as zero when STXLN is zero.
STPTH	Estimated Static Program Paths	This is similar to Nejmeh's NPATH statistic and gives an upper bound on the number of possible paths in the control flow of a function. It is the number of non-cyclic execution paths in a function. The NPATH value for a sequence of statements at the same nesting level is the product of the NPATH values for each statement and for the nested structures. NPATH is the product of: NPATH(sequence of non control statements) = 1 • NPATH(if) = NPATH(body of then) + NPATH(body of else) • NPATH(while) = NPATH(body of while) + 1 • NPATH(do while) = NPATH(body of while) + 1 • NPATH(for) = NPATH(body of for) + 1 • NPATH(switch) = Sum(NPATH(body of case 1) ... NPATH(body of case n)) Note: else and default are counted whether they are present or not. In switch statements, multiple case options on the same branch of the switch statement body are counted once for each independent branch only. The true path count through a function usually obeys the inequality: cyclomatic complexity ≤ true path count ≤ static path count
STRET	Number of Return Points in Function	STRET is the count of the reachable return statements in the function, plus one if there exists a reachable implicit return at the } that terminates the function. Structured Programming requires that every function should have exactly one entry and one exit. This is indicated by a STRET value of 1. STRET is useful when the programmer wants to concentrate on functions that do not follow the Structured Programming paradigm, for example those with switch statements with returns in many or every branch. This metric is computed during data flow analysis, see Data flow-Dependent Metric Values
STST1	Number of Statements in Function (variant 1)	These metrics count the number of statements in the function body. There are 3 variants on the metric: STST1 is the base definition and counts all statements. See table to the right. This metric indicates the maintainability of the function. Number of statements also correlates with most of the metrics defined by Halstead. The greater the number of statements containedinafunction, thegreaterthenumberofoperandsandoperators, andhencethe greater the effort required to understand the function. Functions with high statement counts should be limited. Restructuring into smaller sub-functions is often appropriate
STST2	Number of Statements in Function (variant 2)	STST2 is STST1 except block, empty statements and labels are not counted.
STST3	Number of Statements in Function (variant 3)	STST3 is STST2 except declarations are not counted.
STSUB	Number of Function Calls	The number of function calls within a function. Functions with a large number of function calls are more difficult to understand because their functionality is spread across several components. Note that the calculation of STSUB is based on the number of function calls and not the number of distinct functions that are called, see STCAL. A large STSUB value may be an indication of poor design; for example, a calling tree that spreads too rapidly. See Brandl (1990)16 for a discussion of design complexity and how it is highlighted by the shape of the calling tree.
STUNR	Number of Unreachable Statements	This metric is the count of all statements within the function body that are guaranteed never to be executed. STUNR uses the same method for identifying statements as metric STST1. Hence STUNR counts the following as statements if unreachable. See table to the right the colum Statement Kind Counted. This metric is computed during data flow analysis, see Data flow-Dependent Metric Values
STUNV	Unused or Non-Reused Variables	An unused variable is one that has been defined, but which is never referenced. A nonreused variable is a variable that has a value by assignment, but which is never used subsequently. Such variables are generally clutter and are often evidence of "software ageing", which is the effect of a number of programmers making changes.
STXLN	Number of Executable Lines	This is a count of lines in a function body that have code tokens. Comments, braces, and all tokens of declarations are not treated as code tokens. The function below has an STXLN value of 9. This metric is used in the computation of the STKDN and STPDN metrics
STBME	Embedded Programmer Months	The estimate the number of programmer-months required to create the source code in the embedded environment is: STBME = 3.6 * (STTPP / 1000) 1.20
STBMO	Organic Programmer Months	The estimate the number of programmer-months required to create the source code in the organic environment is: STBMO = 2.4 * (STTPP / 1000) 1.05
STBMS	Semi-detached Programmer Months	The estimate the number of programmer-months required to create the source code in the semi-detached environment is: STBMS = 3.0 * (STTPP / 1000) 1.12

STBUG	Residual Bugs (token-based estimate)	This is an estimate of the number of bugs in the file, based on the number of estimated tokens. Its value would normally be lower than the sum of the function-based STPBG values. For a more detailed discussion of software bug estimates,see Hatton and Hopkins. STBUG = 0.001 * STEFF 2/3
STCDN	Comment to Code Ratio	This metric is defined to be the number of visible characters in comments, divided by the number of visible characters outside comments. Comment delimiters are ignored. Whitespace characters in strings are treated as visible characters. A large value of STCDN indicates that there may be too many comments, which can make a module difficult to read. A small value indicates that there may not be enough comments, which can make a module difficult to understand. The value of STCDN is affected by how QA·C counts the comments. QA·C can count comments in three possible ways: all comments, (a), • all comments except for those from headers, (n), • inlineorinternalcomments(i). Thesearecommentswithinfunctionsandcomments that annotate a line of code (comments that are on the same line as code at file scope). You can determine which counting method is used in Comment Count by setting the -co option on the command line.
STDEV	Estimated Development (programmer-days)	This is an estimate of the number of programmer days required to develop the source file. Unlike COCOMO statistics, which are based solely on the number of lines of code, this estimate is derived from the file's difficulty factor. It is a more accurate measure of the development time, especially after the scaling factor has been adjusted for a particular software environment. STDEV = STEFF / dev_scaling where dev_scaling is a scaling factor defined in -prodoption development::scaling. The default is 6000.
STDIF	Program Difficulty	This is a measure of the difficulty of a translation unit. An average C program has a difficulty of around 12. Anything significantly above this has a rich vocabulary and is potentially difficult to understand. STDIF = STVOL / ((2 + STVAR) * log2 (2 + STVAR))
STECT	Number of External Variables Declared	This is a measure of the number of data objects (not including functions) declared with external linkage. It is an indication of the amount of global data being passed between modules. It is always desirable to reduce dependence on global data to a minimum.
STEFF	Program Effort	This metric is a measure of the programmer effort involved in the production of a translation unit. It is used to produce a development time estimate. STEFF = STVOL * STDIF
STFCO	Estimated Function Coupling	Since the actual value of Brandl's metric requires a full, well-structured calling tree, STFCO can only be an estimate. A high figure indicates a large change of complexity between levels of the calling tree. The metric is computed from STFNC and the STSUB values of the component functions in the translation unit: STFCO = Σ(STSUB) - STFNC + 1
STFNC	Number of Functions in File	This metric is a count of the number of function definitions in the file.
STHAL	Halstead Prediction of STTOT	This metric and also STZIP are predictions (derived from the vocabulary analysis metrics STOPN and STOPT) of what the value of STTOT should be. If they differ from STTOT by more than a factor of 2, it is an indication of an unusual vocabulary. This usually means that either the source code contains sections of rather repetitive code or it has an unusually rich vocabulary. The two metrics are computed as follows: STZIP = (STOPN + STOPT) * (0.5772 + ln (STOPN + STOPT)) STHAL = STOPT * log2 (STOPT) + STOPN * log2 (STOPN)
STM20	Number of Operand Occurrences	ThismetricisthenumberofoperandsinasoftwarecomponentandisoneoftheHalstead vocabulary analysis metrics. Halstead considered that a component is a series of tokens that can be defined as either operators or operands. Unlike STOPN, this metric is the count of every instance of an operand in a file, regardlessofwhetherornotitis distinct. STOPNonlycountstheoperandsthatare distinct.
STM21	Number of Operator Occurrences	ThismetricisthenumberofoperatorsinasoftwarecomponentandisoneoftheHalstead vocabulary analysis metrics. Halstead considered that a component is a series of tokens that can be defined as either operators or operands. Unlike STOPT, this metric is the count of every instance of an operator in a file, regardless of whether or not it is distinct. STOPT only counts the operators that are distinct.
STM22	Number of Statements	This metric is the number of statements in a software component. This is a count of semicolons in a file except for the following instances: within for expressions, • within struct or union declarations/definitions, • within comments, • within literals, • within preprocessor directives, • within old-style C function parameter lists
STM28	Number of Non-Header Comments	This metric is a count of the occurrences of C or C++ style comments in a source file, except for those that are within the header of a file. A file header is defined as tokens preceding the first code token or preprocessor directive token. STM28 is based on the method used to compute STCDN but differs from STCDN in that STCDN counts the visible characters within comments whereas STM28 counts the occurrences of comments.
STM33	Number of Internal Comments	This metric is a count of C style or C++ comments in a source file that are within functions or annotate a line of code at file scope. Comments within functions are all comments at block scope. Comments that annotate code are ones that start or end on the same line as code. STM33 is based on the method used to compute STCDN but differs from STCDN in that STCDN counts the visible characters within comments whereas STM33 counts the occurrences of comments.
STOPN	Number of Distinct Operands	This is the number of distinct operands used in the file. Distinct operands are defined as unique identifiers and each occurrence of a literal. Most literals, except 0 and 1, are usually distinct within a program. Since macros are usually used for fixed success and failure values (such as TRUE and FALSE), the differences in counting strategies are fairly minimal.
STOPT	Number of Distinct Operators	This covers any source code tokens not supplied by the user, such as keywords, operators, and punctuation. STOPT is used in the calculation of a number of other metrics.
STSCT	Number of Static Variables Declared	This metric is computed as the number of variables and functions declared static at file scope.
STSHN	Shannon Information Content	Also known as the "entropy" H, this metric is a widely recognized algorithm for estimating the program space required to encode the functions in a source file. STSHN is measured in bits and is calculated as follows: STSHN = STZIP * log2 (√(STOPN + STOPT) + ln (STOPN + STOPT))
STTDE	Embedded Total Months	The estimate the elapsed time in months required to develop source code in an embedded environment is: STTDE = 2.5 * STBME to the porwer of 0.32

STTDO	Organic Total Months	The estimate the elapsed time in months required to develop source code in an organic environment is: STTDO = 2.5 * STBMO to the power of 0.38
STTDS	Semi-detached Total Months	The estimate the elapsed time in months required to develop source code in a semidetached environment is: STTDS = 2.5 * STBMS to power of 0.35
STTLN	Total Preprocessed Source Lines	Thismetricisacountofthetotalamountoflinesinthetranslationunitafterpre-processing. The pre-processed file will reflect the processing of include files, pre-processor directives and the stripping of comment lines.
STTOT	Total Number of Tokens Used	This metric is the total number of tokens, not distinct tokens, in the source file.
STTPP	Total Unpreprocessed Code Lines	This metric is a count of the total number of source lines in the file before pre-processing.
STVAR	Total Number of Variables	This metric represents the total number of distinct identifiers
STVOL	Program Volume	This is a measure of the number of bits required for a uniform binary encoding of the program text. It is used to calculate various Halstead vocabulary metrics. The following is the calculation for the program volume: STVOL = STTOT * log2 (STOPN + STOPT)
STZIP	Zipf Prediction of STTOT	STZIP = (STOPN + STOPT) * (0.5772 + ln (STOPN + STOPT)) See STHAL.

CPPDepend	
Measures Name	**Measure definition**
NbLinesOfCode	This metric (known as LOC) can be computed only if PDB files are present. NDepend computes this metric directly from the info provided in PDB files. The LOC for a method is equals to the number of sequence point found for this method in the PDB file. A sequence point is used to mark a spot in the IL code that corresponds to a specific location in the original source. More info about sequence points here.Notice that sequence points which correspond to C# braces'{' and '}' are not taken account.
NbLinesOfComment	(Only available for C# code, a VB.NET version is currently under development). This metric can be computed only if PDB files are present and if corresponding source files can be found. The number of lines of comment is computed as follow: For a method, it is the number of lines of comment that can be found in its body. In C# the body of a method begins with a '{' and ends with a '}'. If a method contains an anonymous method, lines of comment defined in the anonymous method are not counted for the outer method but are counted for the anonymous method. -For a type, it is the sum of the number of lines of comment that can be found in each of its partial definition. In C#, each partial definition of a type begins with a '{ and ends with a '}'. -For a namespace, it is the sum of the number of lines of comment that can be found in each of its partial definition. In C# each partial definition of a namespace begins with a '{ and ends with a '}'. -For an assembly, it is the sum of the number of lines of comment that can be found in each of its source file. otice that this metric is not an additive metric (i.e for example, the number of lines of comment of a namespace can be greater than the number of lines of comment over all its types). Recommendations: This metric is not helpful to asses the quality of source code. We prefer to use the metric PercentageComment.
PercentageComment	(Only available for C# code, a VB.NET version is currently under development) This metric is computed with the following formula: PercentageComment = 100*NbLinesOfComment / (NbLinesOfComment + NbLinesOfCode) Recommendations: Code where the percentage of comment is lower than 20% should be more commented. However overly commented code (>40%) is not necessarily a blessing as it can be considered as an insult to the intelligence of the reader.
NbILInstructions	Notice that the number of IL instructions can vary depending if your assemblies are compiled in debug or in release mode. Indeed compiler's optimizations can modify the number of IL instructions. For example a compiler can add some nop IL instructions in debug mode to handle Edit and Continue and to allow attach an IL instruction to a curly brace. Notice that IL instructions of third-party assemblies are not taken account. Recommendations: Methods where NbILInstructions is higher than 100 are hard to understand and maintain. Methods where NbILInstructions is higher than 200 are extremely complex and should be split in smaller methods (except if they are automatically generated by a tool).
NbAssemblies	Only application assemblies are taken into account.
NbNamespaces	The number of namespaces. The anonymous namespace counts as one. If a namespace is defined over N assemblies, it will count as N. Namespaces declared in third-party assemblies are not taken account.
NbTypes	The number of types. A type can be an abstract or a concrete class, a structure, an enumeration, a delegate class or an interface. Types declared in third-party assemblies are not taken account.
NbMethods	The number of methods. A method can be an abstract, virtual or non-virtual method, a method declared in an interface, a constructor, a class constructor, a finalizer, a property/indexer getter or setter, an event adder or remover. Methods declared in third-party assemblies are not taken account. Recommendations: Types where NbMethods > 20 might be hard to understand and maintain but there might be cases where it is relevant to have a high value for NbMethods. For example, the System.Windows.Forms.DataGridView third-party class has more than 1000 methods.
NbFields	The number of fields. A field can be a regular field, an enumeration's value or a readonly or a const field. Fields declared in third-party assemblies are not taken account. Recommendations: Types that are not enumeration and where NbFields is higher 20 might be hard to understand and maintain but there might be cases where it is relevant to have a high value for NbFields. For example, the System.Windows.Forms.Control third-party class has more than 200 fields.
PercentageCoverage	The percentage of code coverage by tests. Code coverage data are imported from coverage files. If you are using the uncoverable attribute feature on a method for example, if all sibling methods are 100% covered, then the parent type will be considered as 100% covered. Coverage metrics are not available if the metric NbLinesOfCode is not available.
NbLinesOfCodeCovered	The number of lines of code covered by tests.
NbLinesOfCodeNotCovered	The number of lines of code not covered by tests.
Afferent coupling (Ca)	The number of types outside this assembly that depend on types within this assembly. High afferent coupling indicates that the concerned assemblies have many responsibilities.
Efferent coupling (Ce)	The number of types outside this assembly used by child types of this assembly. High efferent coupling indicates that the concerned assembly is dependant. Notice that types declared in third-party assemblies are taken into account.
Relational Cohesion (H)	Average number of internal relationships per type. Let R be the number of type relationships that are internal to this assembly (i.e that do not connect to types outside the assembly). Let N be the number of types within the assembly. H = (R + 1)/ N. The extra 1 in the formula prevents H=0 when N=1. The relational cohesion represents the relationship that this assembly has to all its types. Recommendations: As classes inside an assembly should be strongly related, the cohesion should be high. On the other hand, too high values may indicate over-coupling. A good range for RelationalCohesion is 1.5 to 4.0. Assemblies where RelationalCohesion < 1.5 or RelationalCohesion > 4.0 might be problematic.

Instability (I)	The ratio of efferent coupling (Ce) to total coupling. I = Ce / (Ce + Ca). This metric is an indicator of the assembly's resilience to change. The range for this metric is 0 to 1, with I=0 indicating a completely stable assembly and I=1 indicating a completely instable assembly.
Abstractness (A)	The ratio of the number of internal abstract types (i.e abstract classes and interfaces) to the number of internal types. The range for this metric is 0 to 1, with A=0 indicating a completely concrete assembly and A=1 indicating a completely abstract assembly.
Distance from main sequence (D)	The perpendicular normalized distance of an assembly from the idealized line A + I = 1 (called main sequence). This metric is an indicator of the assembly's balance between abstractness and stability. An assembly squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal assemblies are either completely abstract and stable (I=0, A=1) or completely concrete and instable (I=1, A=0). The range for this metric is 0 to 1, with D=0 indicating an assembly that is coincident with the main sequence and D=1 indicating an assembly that is as far from the main sequence as possible. The picture in the report reveals if an assembly is in the zone of pain (I and A both close to 0) or in the zone of uselessness (I and A both close to 1). Recommendations: Assemblies where NormDistFromMainSeq is higher than 0.7 might be problematic. However, in the real world it is very hard to avoid such assemblies.
Afferent coupling at namespace level (NamespaceCa)	The Afferent Coupling for a particular namespace is the number of namespaces that depends directly on it. Related Link:: Code metrics on Coupling, Dead Code, Design flaws and Re-engineering
Efferent coupling at namespace level (NamespaceCe	The Efferent Coupling for a particular namespace is the number of namespaces it directly depends on. Notice that namespaces declared in third-party assemblies are taken into account. Related Link:: Code metrics on Coupling, Dead Code, Design flaws and Re-engineering. Related Link:: Layering, the Level metric and the Discourse of Method
Level	(defined for assemblies, namespaces, types, methods) The Level value for a namespace is defined as follow: Level = 0 : if the namespace doesn't use any other namespace. Level metric definitions for assemblies, types and methods are inferred from the above definition. This metric has been first defined by John Lakos in his book Large-Scale C++ Software Design. Level = 1 + (Max Level over namespace it uses direcly) Level = N/A : if the namespace is involved in a dependency cycle or uses directly or indirectly a namespace involved in a dependency cycle. Recommendations: This metric helps objectively classify the assemblies, namespaces, types and methods as high level,mid level or low level. There is no particular recommendation for high or small values. This metric is also useful to discover dependency cycles in your application. For instance if some namespaces are matched by the following CQLinq query, it means that there is some dependency cycles between the namespaces of your application: from n in Application.Namespaces where n.Level == null select n
Type rank	TypeRank values are computed by applying the Google PageRank algorithm on the graph of types' dependencies. A homothety of center 0.15 is applied to make it so that the average of TypeRank is 1. Recommendations: Types with high TypeRank should be more carefully tested because bugs in such types will likely be more catastrophic.
Efferent Coupling at type level (Ce)	The Efferent Coupling for a particular type is the number of types it directly depends on. Notice that types declared in third-party assemblies are taken into account. Recommendations: Types where TypeCe > 50 are types that depends on too many other types. They are complex and have more than one responsibility. They are good candidate for refactoring. Related Link:: Code metrics on Coupling, Dead Code, Design flaws and Re-engineering.
Lack of Cohesion Of Methods (LCOM)	The single responsibility principle states that a class should not have more than one reason to change. Such a class is said to be cohesive. A high LCOM value generally pinpoints a poorly cohesive class. There are several LCOM metrics. The LCOM takes its values in the range [0-1]. The LCOM HS (HS stands for Henderson-Sellers) takes its values in the range [0-2]. A LCOM HS value higher than 1 should be considered alarming. Here are algorithms used by NDepend to compute LCOM metrics: LCOM = 1 – (sum(MF)/M*F), LCOM HS = (M – sum(MF)/F)(M-1) Where: M is the number of methods in class (both static and instance methods are counted, it includes also constructors, properties getters/setters, events add/remove methods). F is the number of instance fields in the class. MF is the number of methods of the class accessing a particular instance field. Sum(MF) is the sum of MF over all instance fields of the class. The underlying idea behind these formulas can be stated as follow: a class is utterly cohesive if all its methods use all its instance fields, which means that sum(MF)=M*F and then LCOM = 0 and LCOMHS = Recommendations: Types where LCOM > 0.8 and NbFields > 10 and NbMethods >10 might be problematic. However, it is very hard to avoid such non-cohesive types. Types where LCOMHS > 1.0 and NbFields > 10 and NbMethods >10 should be avoided. Note that this constraint is stronger (and thus easier to satisfy) than the constraint types where LCOM > 0.8 and NbFields > 10 and NbMethods >10.
cyclomatic complexity	(defined for types, methods) *(Only available for C# code, a VB.NET version is currently under development)* Cyclomatic complexity is a popular procedural software metric equal to the number of decisions that can be taken in a procedure. Concretely, in C# the CC of a method is 1 + {the number of following expressions found in the body of the method}: if| while | for | foreach | case | default | continue | goto | && | || | catch | ternary operator ?: | ?? Following expressions are not counted for CC computation: else | do | switch | try | using | throw | finally | return | object creation | method call | field access. The Cyclomatic Complexity metric is defined on methods. Adapted to the OO world, this metric is also defined for classes and structures as the sum of its methods CC. Notice that the CC of an anonymous method is not counted when computing the CC of its outer method. Recommendations: Methods where CC is higher than 15 are hard to understand and maintain. Methods where CC is higher than 30 are extremely complex and should be split into smaller methods (except if they are automatically generated by a tool).
IL Cyclomatic Complexity (ILCC)	The CC metric is language dependent. Thus, NDepend provides the ILCC which is language independent because it is computed from IL as 1 + {the number of different offsets targeted by a jump/branch IL instruction}. Experience shows that NDepend CC is a bit larger than the CC computed in C# or VB.NET. Indeed, a C# 'if' expression yields one IL jump. A C# 'for' loop yields two different offsets targeted by a branch IL instruction while a foreach C# loop yields three. Recommendations: Methods where ILCyclomaticComplexity is higher than 20 are hard to understand and maintain. Methods where ILCyclomaticComplexity is higher than 40 are extremely complex and should be split into smaller methods (except if they are automatically generated by a tool).
Size of instance	*(defined for instance fields and types)* The size of instances of an instance field is defined as the size, in bytes, of instances of its type. The size of instance of a static field is equal to 0. The size of instances of a class or a structure is defined as the sum of size of instances of its fields plus the size of instances of its base class. Fields of reference types (class, interface, delegate…) always count for 4 bytes while the footprint of fields of value types (structure, int, byte, double…) might vary. Size of instances of an enumeration is equal to the size of instances of the underlying numeric primitive type. It is computed from the *value__* instance field (all enumerations have such a field when compiled in IL). Size of instances of generic types might be erroneous because we can't statically know the footprint of parameter types (except when they have the class constraint). Recommendations: Types where SizeOfInst is higher than 64 might degrade performance (depending on the number of instances created at runtime) and might be hard to maintain. However it is not a rule since sometime there is no alternative (the size of instances of the System.Net.NetworkInformation.SystemIcmpV6Statistics third-party class is 2064 bytes). Non-static and non-generic types where SizeOfInst is equal to 0 indicate stateless types that might eventually be turned into static classes.
NbInterfacesImplemented	The number of interfaces implemented. This metric is available for interfaces, in this case the value is the number of interface extended, directly or indirectly. For derived class, this metric also count the sum of interfaces implemented by base class(es).
Association Between Class (ABC)	The Association Between Classes metric for a particular class or structure is the number of members of others types it directly uses in the body of its methods.
Number of Children (NOC)	The number of children for a class is the number of sub-classes (whatever their positions in the sub branch of the inheritance tree). The number of children for an interface is the number of types that implement it. In both cases the computation of this metric only count types declared in the application code and thus, doesn't take account of types declared in third-party assemblies.
Depth of Inheritance Tree (DIT)	The Depth of Inheritance Tree for a class or a structure is its number of base classes (including the SystRecommendations: Types where DepthOfInheritance is higher or equal than 6 might be hard to maintain. However it is not a rule since sometimes your classes might inherit from third-party classes which have a high value for depth of inheritance. For example, the average depth of inheritance for third-party classes which derive from System.*Windows.Forms.Control is 5.3.em.Object* class thus DIT >= 1).

Method rank	MethodRank values are computed by applying the Google PageRank algorithm on the graph of methods' dependencies. A homothety of center 0.15 is applied to make it so that the average of MethodRank is 1. Recommendations: Methods with high MethodRank should be more carefully tested because bugs in such methods will likely be more catastrophic. Related Link:: Code metrics on Coupling, Dead Code, Design flaws and Re-engineering.
Afferent coupling at method level (MethodCa)	The Afferent Coupling for a particular method is the number of methods that depends directly on it. Related Link:: Code metrics on Coupling, Dead Code, Design flaws and Re-engineering
Efferent coupling at method level (MethodCe)	The Efferent Coupling for a particular method is the number of methods it directly depends on. Notice that methods declared in third-party assemblies are taken into account. Related Link:: Code metrics on Coupling, Dead Code, Design flaws and Re-engineering
IL Nesting Depth	The metric Nesting Depth for a method is the maximum number of encapsulated scopes inside the body of the method. The metric IL Nesting Depth is computed from the IL code. Values computed are very similar to what we would expect by computing them from the C# or VB.NET source code. When you have a testing condition with N conditions, such as if(i > 9 && i < 12) then it is considered as N scopes because it is possible to decompose such conditions into N atomic conditions. When a method has a large number of case statements corresponding to a switch, the C# and VB.NET compiler generally produce optimizations while generating the IL. In such case, the IL Nesting Depth corresponding value might be slightly higher to what you would expect. Recommendations: Methods where ILNestingDepth is higher than 4 are hard to understand and maintain. Methods where ILNestingDepth is higher than 8 are extremely complex and should be split in smaller methods (except if they are automatically generated by a tool).
NbParameters	The number of parameters of a method. *Ref* and *Out* are also counted. The this reference passed to instance methods in IL is not counted as a parameter. Recommendations: Methods where NbParameters is higher than 5 might be painful to call and might degrade performance. You should prefer using additional properties/fields to the declaring type to handle numerous states. Another alternative is to provide a class or structure dedicated to handle arguments passing (for example see the class System.Diagnostics.ProcessStartInfo and the method System.Diagnostics.Process.Start(ProcessStartInfo)).
NbVariables	The number of variables declared in the body of a method. Recommendations: Methods where NbVariables is higher than 8 are hard to understand and maintain. Methods where NbVariables is higher than 15 are extremely complex and should be split in smaller methods (except if they are automatically generated by a tool)
NbOverloads	The number of overloads of a method. . If a method is not overloaded, its NbOverloads value is equals to 1. This metric is also applicable to constructors. Recommendations: Methods where NbOverloads is higher than 6 might be a problem to maintain and provoke higher coupling than necessary. This might also reveal a potential misused of the C# and VB.NET language that since C#3 and VB9 support object initialization. This feature helps reducing the number of constructors of a class.

Appendix C

PRQA Code Review Report

Note: This report is generated using the software (PRQA Framework) of Programming Research Limited and is the Intellectual Property of Programming Research Limited.

Project : C:/06-MayraTERM6/Thesis/CodeToTest/fastgrep2/fastgrep
Status at: 18 May, 2018 at 14:00:16

1. Files
2. Classes
3. Functions
4. Analysis Status
5. Analysis Settings

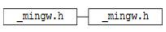
Number of Files	25
Lines of Code (source files only)	7462
Total preprocessed code line	929
Diagnostic Count	2382

Files

File: egrep.c

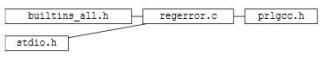
Metric	STVAR	STDIF	STBME	STFCO	STBMO	STHAL	STZIP	STBMS	STECT
Value	175	25.97	3.122	10	2.119	3564	2894	2.626	48
Metric	STSCT	STTPP	STM33	STTOT	STDEV	STOPN	STVOL	STEFF	STTDS
Value	0	888	77	3917	148.63	376	34332	891751	3.505
Metric	STOPT	STBUG	STCDN	STTLN	STSHN	STFNC	STM21	STM20	STM22
Value	59	9	0.472	47	13750	1	2633	1279	372
Metric	STTDE	STM28	STTDO						
Value	3.599	101	3.325						
QA-C	Rule 1	Rule 3	Rule 2	Rule 5	Rule 7	Rule 6	Rule 8		
Violations	2	1	276	1	1	6	1		
Density	0.0023	0.0011	0.3108	0.0011	0.0011	0.0068	0.0011		
RCMA	Rule 2								
Violations	85								
Density	0.0957								

File: _mingw.h



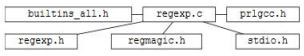
File: regerror.c

Metric	STVAR	STDIF	STBME	STFCO	STBMO	STHAL	STZIP	STBMS	STECT
Value	4	8.36	0.027	10	0.033	70	71	0.031	0
Metric	STSCT	STTPP	STM33	STTOT	STDEV	STOPN	STVOL	STEFF	STTDS
Value	0	17	3	30	0.18	5	130	1084	0.743
Metric	STOPT	STBUG	STCDN	STTLN	STSHN	STFNC	STM21	STM20	STM22
Value	15	0	0.811	6	206	1	22	7	2
Metric	STTDE	STM28	STTDO						
Value	0.788	3	0.686						
QA-C	Rule 3	Rule 2							
Violations	1	6							
Density	0.0588	0.3529							



File: regexp.c

Metric	STVAR	STDIF	STBME	STFCO	STBMO	STHAL	STZIP	STBMS	STECT
Value	126	42.98	4.552	104	2.947	3504	2848	3.735	0
Metric	STSCT	STTPP	STM33	STTOT	STDEV	STOPN	STVOL	STEFF	STTDS
Value	22	1216	108	4404	275.89	370	38512	1655350	3.965
Metric	STOPT	STBUG	STCDN	STTLN	STSHN	STFNC	STM21	STM20	STM22
Value	59	14	0.693	649	13507	15	2981	1420	445
Metric	STTDE	STM28	STTDO						
Value	4.060	144	3.770						
QA-C	Rule 1	Rule 0	Rule 3	Rule 2	Rule 5	Rule 4	Rule 7	Rule 6	Rule 8
Violations	71	5	88	1080	96	10	1	4	3
Density	0.0584	0.0041	0.0724	0.8882	0.0789	0.0082	0.0008	0.0033	0.0025
RCMA	Rule 2								
Violations	6								
Density	0.0049								



File: regexp.h

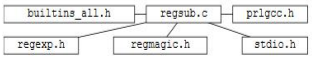
QA-C	Rule 2	Rule 6
Violations	17	1
Density	0.8095	0.0476

File: regmagic.h

QA-C	Rule 2
Violations	3
Density	0.6000

File: regsub.c

Metric									
Metric	STVAR	STDIF	STBME	STFCO	STBMO	STHAL	STZIP	STBMS	STECT
Value	20	21.59	0.174	14	0.169	399	373	0.177	0
Metric	STSCT	STTPP	STM33	STTOT	STDEV	STOPN	STVOL	STEFF	STTDS
Value	0	80	3	339	7.62	39	2118	45727	1.364
Metric	STOPT	STBUG	STCDN	STTLN	STSHN	STFNC	STM21	STM20	STM22
Value	37	1	0.120	45	1382	1	234	102	23
Metric	STTDE	STM28	STTDO						
Value	1.428	4	1.273						
QA-C	Rule 1	Rule 3	Rule 2	Rule 5	Rule 4	Rule 6			
Violations	10	2	94	17	1	1			
Density	0.1250	0.0250	1.1750	0.2125	0.0125	0.0125			
RCMA	Rule 2								
Violations	1								
Density	0.0125								



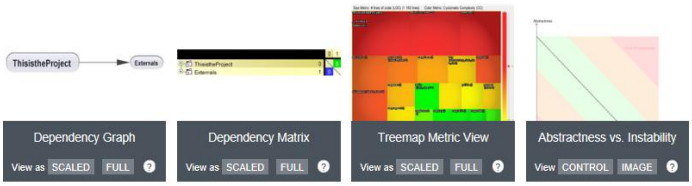
cppdepend report summary

GET STARTED. QUICK TIPS. BACK TO CPPDEPEND.

The present HTML report is a summary of data gathered by the analysis. It is recommended to use the CppDepend interactive UI capabilities to make the most of CppDepend by mastering all aspects of your code.

application name	LastOne
report build date	05/20/2018 01:22:15
analysis duration	00:00:58
cppdepend version	2018.1.0.95 Evaluation 9 days left
baseline for comparison	Baseline is same code base snapshot.
code coverage data	Not Defined. To import Code Coverage Data, please read this online documentation.

Diagrams



Dependency Graph View as SCALED FULL

Dependency Matrix View as SCALED FULL

Treemap Metric View View as SCALED FULL

Abstractness vs. Instability View CONTROL IMAGE

Name	Trend	Baseline Value	Value	Group
Percentage Code Coverage			N/A because no coverage data	Project Rules \ Quality Gates
Percentage Coverage on New Code			N/A because no coverage data	Project Rules \ Quality Gates
Percentage Coverage on Refactored Code			N/A because no coverage data	Project Rules \ Quality Gates
Blocker Issues		0 issues	0 issues	Project Rules \ Quality Gates
Critical Issues		0 issues	0 issues	Project Rules \ Quality Gates
New Blocker / Critical / High Issues			0 issues	Project Rules \ Quality Gates
Critical Rules Violated		1 rules	1 rules	Project Rules \ Quality Gates
Percentage Debt		7.31 %	7.31 %	Project Rules \ Quality Gates
New Debt since Baseline			0 man-days	Project Rules \ Quality Gates
Debt Rating per Namespace		0 namespaces	0 namespaces	Project Rules \ Quality Gates
New Annual Interest since Baseline			0 man-days	Project Rules \ Quality Gates

Showing 1 to 11 of 11 entries

Application Metrics

Note: Further Application Statistics are available.

 # Lines of Code
1 183 no diff
0 (NotMyCode) no diff
Estimated Dev Effort 24d no diff

 # Types
2 no diff
1 Projects no diff
1 Namespaces no diff
49 Methods no diff
80 Fields no diff
10 Source Files no diff
20 Third-Party Elements no diff

Comment
22.07% no diff
335 Lines of Comment no diff

Debt
7.31% no diff
Rating B 4h 22min effort to reach A
Debt 1d 5h no diff
Annual Interest 3h 44min no diff
Breaking Point 3y no diff

Coverage
N/A because no coverage data specified

Method Complexity
67 Max no diff
9.61 Average no diff

Quality Gates
Fail 1
Warn 0
Pass 7

Rules
Critical 1
Violated 14
Ok 274

Issues
All 60
Blocker 0
Critical 0
High 4
Medium 23
Low 33

Quality Gates summary

10 0 1