Concordia University
Engineering and Computer Science

SOEN 6611

Software Measurement

Summer 2019

IEEE Report

Professor : Jinqiu Yang

June 25, 2019

Name	Student ID	Email
Sahaja Gottipati	40093560	sahajagottipati@gmail.com
Rajasekhar Reddy Guntaka	40094479	rajasekhar.grr@gmail.com
Sai Santhosh Sathwik Ganta	40091433	sathwik.g@hotmail.com
Sai Charan Duduka	40103928	charan140494@gmail.com
Koteswara Rao Kothamasu	40070848	rao.kothamasu@ymail.com

Link to the replication package in GitHub:

https://github.com/soen-6611-teaml/SOEN_6611_TEAM-L

Sahaja Gottipati
Dept. Of Computer Science
Concordia University
Montreal, Canada
sahajagottipati@gmail.com

Rajasekhar Reddy Guntaka
Dept. Of Software Engineering
Concordia University
Montreal, Canada
rajasekhar.grr@gmail.com

Sai Santhosh Sathwik Ganta
Dept. Of Software Engineering
Concordia University
Montreal, Canada
sathwik.g@hotmail.com

KoteswaraRao Kothamasu
Dept. Of Software Engineering
Concordia University
Montreal, Canada
rao.kothamasu@ymail.com

Sai Charan Duduka
Dept. Of Software Engineering
Concordia University
Montreal, Canada
charan140494@gmail.com

***Abstract*—In order to measure the effectiveness of the software, we need to consider subjective measurements. In this work, we considered six distinct metrics and correlate them to determine how effective and relative the projects are defined. We selected 5 open source projects such as Apache Commons Math, JFreeChart, Apache Commons Lang, Apache Commons Collections, Apache Commons DbUtils. We have assessed the impact of metric on a product framework with the results gathered from this study. Also, we have considered the Spearman correlation coefficient to measure the correlation between the entities.**

Keywords— Statement Coverage, Branch Coverage, Mutation Testing, Cyclomatic Complexity, Maintainability Index, Defect Density.

I. INTRODUCTION

Software has transformed in many ways in the present IT world . In the past, work which requires more of human effort has been changed and automated using the software systems. As the software is gaining more demand day by day there is a need in measuring the effectiveness of systems. Software metrics are used to make sure all the important measurements of a software such as complexity, coverage, maintainability, bug fixing, and more can be measured.

Measurement lies in everything around us like economic measurements consists cost calculations, measurements in flight radar systems enable us to detect aircrafts or airplanes when direct vision is obtained. In medical systems, measurements helps doctors to treat and diagnose specific diseases based on the issue and without that there will practically be no means of knowing how effective and efficient the software system is. Measurement is not solely the domain of professional technologists and helps us to understand our world by all means and helps us interact with our surroundings and improve our lives

Many expertise found out various ways to measure or calculate the software metrics and among them, there are few well known metrics. In this paper, we discussed about 6 such metrics to calculate how different projects are effective and efficient.

1) Statement and branch coverage to calculate code coverage.

2) PIT testing for testing the test suit effectiveness.

3) Cyclomatic complexity using McCabe cyclomatic complexity to calculate project complexity.

4) Maintainability index to measure the software maintenance effort.

5) Post-release defect density to calculate the software quality.

The paper will also discuss about the correlations between these metrics are formed, the hypothesis, the findings, and the conclusion. Correlation is a relative analysis that measures the relativity and strength of association between two variables and the direction of the relationship. We have implemented Spearman correlation to find and draw the correlation between these metrics.

II. PROJECTS SELECTED

A. Apache Commons Math

Size:186,000 LOC
Technologies Underneath:JAVA
Project link: http://issues.apache.org/jira/browse/MATH
Description: Commons Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang.

B. JFreeChart

Size:317,000 LOC
Technologies Underneath:JAVA
Project link: https://github.com/jfree/jfreechart/issues
Description: JFreeChart is a chart library for the Java platform that supports a wide range of charts including pie charts (2D and 3D), bar charts (horizontal and vertical, regular or stacked, with optional 3D-effects), line charts, XY plots, scatter plots, time series charts, high/low/open/close charts, candlestick plots, Gantt charts, Pareto charts, combination charts, and more. It is suitable for use in applications, applets, servlets, and JSP.

C. Apache Commons Collections

Size:121108 LOC
Technologies Underneath:JAVA
Project link:
http://issues.apache.org/jira/browse/COLLECTIONS
Description: The Java Collections Framework was a major addition in JDK 1.2. It added many powerful data structures that accelerate development of most significant Java applications. Since that time it has become the recognised standard for collection handling in Java.

D. **Apache Commons Lang**

Size:90700 LOC
Technologies Underneath:JAVA
Project link: http://issues.apache.org/jira/browse/LANG
Description: Commons Lang, a package of Java utility classes for the classes that are in java.lang's hierarchy, or are considered to be so standard as to justify existence in java.lang.

E. **Apache Commons DbUtils**

Size:8280 LOC
Technologies Underneath:JAVA
Project link: http://issues.apache.org/jira/browse/DBUTILS
Description: The Commons DbUtils library is a small set of classes designed to make working with JDBC easier. JDBC resource cleanup code is mundane, error prone work so these classes abstract out all of the cleanup tasks from your code leaving you with what you really wanted to do with JDBC in the first place: query and update data.

III. METRICS

A. **Statement Coverage:**

Statement coverage is a white box testing technique, which involves testing of all the statements present in the source code at least once. If every statement of the program is executed, then it is 100% statement coverage. It is a metric, which calculates the number of statements executed in the source code. This technique helps in checking whether the source code is performing the required actions. It can also be used to check the quality of code and flow of different paths of the program.

A= Number of statements executed

B= Total number of statements in source code

Statement Coverage = (A/B) * 100

Data Collection:

Plugin Used: JaCoCo

Statement Coverage can be calculated by code coverage tools. For calculating statement coverage we used EclEmma plugin in Eclipse.

Steps for report generation:

- Install EclEmma plugin in Eclipse through **Eclipse->Help->Eclipse-Marketplace->Search for EclEmma-> Install and Restart Eclipse**
- Import the project into Eclipse as Existing maven project through **File->Import->Existing Maven Projects.**
- Add JaCoCo dependency in pom file of the maven project. Run the project as Maven Install.
- Results in HTML, XML and CSV format will be generated in the target folder of the project.

B. **Branch Coverage:**

Branch coverage measures the proportion of branches in the control flow graphs (e.g., if statements and case statements) that are executed by the test suite. Achieving full branch coverage will protect against errors in which some requirements are not met in a certain branch. Test coverage criteria requires enough test cases such that each condition in a decision takes on all possible outcomes at least once, and each point of entry to a program or subroutine is invoked at least once. That is, every branch (decision) taken each way, true and false. It helps in validating all the branches in the code making sure that no branch leads to abnormal behavior of the application.

A= Number of Decision Statements Executed

B= Total Number of Decision Outcomes

Branch Coverage = (A/B) * 100

Data Collection:

Plugin Used: JaCoCo

Branch Coverage can be calculated by code coverage tools. For calculating branch coverage we used EclEmma plugin in Eclipse.

Steps for report generation:

- Install EclEmma plugin in Eclipse through **Eclipse->Help->Eclipse-Marketplace->Search for EclEmma-> Install and Restart Eclipse**
- Import the project into Eclipse as Existing maven project through **File->Import->Existing Maven Projects.**
- Add JaCoCo dependency in pom file of the maven project. Run the project as Maven Install.
- Results in HTML, XML and CSV format will be generated in the target folder of the project.

C. **Test Suite Effectiveness:**

Mutation testing is also referred by some as White box technique, also it is also a coverage technique widely used to evaluate the effectiveness of test suites.Mutation refers to creating small modification of program and creating a mutant of original source code. Idea is to identify the mutant and kill it.For this the code that is mutated and one that is original are compared, if the results are similar the mutant remains alive and if detected the mutant is killed.

A= Dead Mutants

B= Total number of non-equivalent mutants.

Mutation score = A/B

Data Collection:

Plugin Used: PITClipse

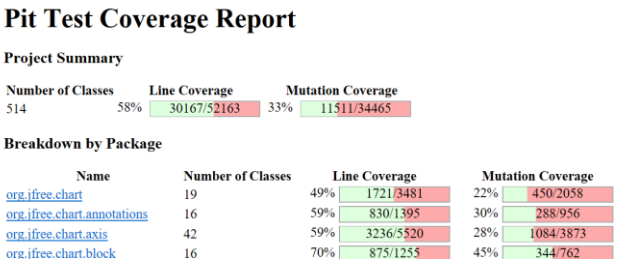
Calculating Mutation Score: Mutation score is defined as percentage of killed mutants with total number of mutants.

- Mutation Score = (Killed Mutants/Total number of Mutants)*100

Steps for report generation:

- Import a maven project through **File -> Import -> Existing Maven Projects.**
- Add Pitest dependency plugin in the pom.xml file of the maven project.
- Run the project as Maven Install.
- After execution the reports are generated in the target folder of the project.

PI Test Coverage Report



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
514	58%	30167/52163	33%	11511/34465

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.jfree.chart	19	49%	1721/3481	22%	450/2058
org.jfree.chart.annotations	16	59%	830/1395	30%	288/956
org.jfree.chart.axis	42	59%	3236/5520	28%	1084/3873
org.jfree.chart.block	16	70%	875/1255	45%	344/762

D. **McCabe Complexity**

Cyclomatic complexity is a metric used to measure the complexity of a program. The procedure's statements are transformed into a graph. Then the cyclomatic complexity is calculated by measuring the linearly independent path in the graph and is represented by a single number. Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths. Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.

$$CC = e - n + 2$$

CC = the cyclomatic complexity of the flow graph G of program.

e = the number of edges in G

n = the number of nodes in G

Data Collection:

Plugin Used: JaCoCo

Calculating McCabe Complexity:

McCabe Complexity can be calculated by code coverage tools. Sci-tools can be used to calculate McCabe Complexity but we used EclEmma plugin in Eclipse for measuring the complexity value.

Steps for report generation:

- Install EclEmma plugin in Eclipse through **Eclipse->Help->Eclipse-Marketplace->Search for EclEmma-> Install and Restart Eclipse**
- Import the project into Eclipse as Existing maven project through **File->Import->Existing Maven Projects.**
- Add JaCoCo dependency in pom file of the maven project. Run the project as Maven Install.
- Results in HTML, XML and CSV format will be generated in the target folder of the project.
- Consider complexity of both missed and covered branches while calculating McCabe complexity from JaCoCo report.

E. Maintainability Index:

Software maintenance involves modification of the product to fix the bugs. This plays an important role in a way that the software does not age. Maintainability Index is a software metric which measures how maintainable (easy to support and change) the source code is. The maintainability index is calculated as a factored formula consisting of Lines Of Code, Cyclomatic Complexity and Halstead volume.

The original formula:

MI = 171 - 5.2 * ln(V) - 0.23 * (G) - 16.2 * ln(LOC)

V = Halstead Volume

G = Cyclomatic Complexity

LOC = count of source Lines Of Code (SLOC)

CM = percent of lines of Comment (optional)

The value of CM is optional as it is not yielding any effective results towards calculating Maintainability Index metric. Thus the insignificant CM calculated values are not considered while arriving at the Maintainability Index metric values for all the projects. Rationale behind this is explained in detail in the following references.[8][9]

Data Collection:

Tools Used: Prest and Sci-tools

Calculating Maintainability Index: We considered 5 Distinct Projects and calculated Maintainability Index. We need to calculate Maintainability Index for each class and average the all classes denotes the Maintainability Index for Specific Project. The Data Collection starts from Using Prest jar file to calculate Halstead Volume; PREST is easy to use as it takes Input files and parse the methods, classes and packages and consider class results. Also, Calculating the Cyclomatic complexity and source lines of code we considered Sci-tools which takes the project files as Input and Analyze the data. We choose the required metrics from the exported files. The results from both the files are combined and a macro was written to get the Maintainability Index for each class and calculated the average to get the Maintainability Index of a project.

The Calculation of Maintainability Index for all five projects is depicted in Table

Projects	Versions	Maintainability Index
Apache Commons DbUtils	1.2 1.3 1.4 1.5 1.6	77.74 77.75 74.03 72.92 70.23
Apache Commons Lang	3.4 3.5 3.6 3.7 3.8	45.73 51.97 51.96 52.50 52.44
Apache Commons Collections	3.0 4.0 4.1 4.2 4.3	69.03 66.52 65.01 76.30 76.29
Apache Commons Math	3.2 3.3 3.4 3.5 3.6	67.58 67.48 67.06 66.95 60.56
JFreeChart	1.0.4 1.0.6 1.0.7	63.82 63.00 56.04

	1.0.8 1.5.0	63.56 62.56

Table 1 : Maintainability Index values for all the projects and different variations are illustrated in the above table.

F. Post Release Defect Density:

The post-release Defect Density is a metric which we can compare across the different versions of the projects as well as various projects. The post-release Defect Density is basically used to indicate the Product Quality. The total number of source lines of code are used by the calculation of the post-release Defect Density.

A formula to measure Defect Density:

Defect Density = Defect count/size of the release(SLOC)

- Defect Count is the total number of post-release defects
- SLOC are the total number of source lines of code

It measures the defects relative to the software size expressed as lines of code or function point, this metric is used in many commercial software systems. The defects that are identified are fixed as soon as possible or at least by the next version. The defect count can increase even in a new version because the defects that were rectified might have caused defects elsewhere (ripple effect). Post release defect density can be formulated Defect Density is the number of defects confirmed in software/module during a specific period of operation or development divided by the size of the software/module.

A= Defect Count

B= Size of release (SLOC)

Density Defect = A/B

Data Collection:

Tools Used: JIRA, LocMetric

In order the collect the bugs related to version we have used widely used tool like JIRA. For the projects listed above we took bugs reports for different versions and based on the bugs and SLOC from LocMetric tool calculated the defect count based on number of confirmed and closed issues mentioned in the Jira reports.

In order to get the defects lists for each version of the individual project, we gone through the project's official website and searched for the defects list and from the link for the external issue tracking system that the specific project have used. We performed this step manually for each project. We collected the bugs along with essential information such as bug ID, affected version, fix version, priority, product, component, status, and summary

Projects	Version	Number of Bugs	Source Lines of Code (in LOC)	Post Release Defect Density
Apache Commons Math	3.6	13	208849	0.00006225
Apache Commons Lang	3.8	3	78050	0.00005125
Apache Commons Collections	4.3	1	62869	0.0000159
Apache Commons DbUtils	1.7	4	6864	0.000582
JFreeChart	1.5.0	12	134119	0.00008947

Table 2 : Post release defect density for all the projects are derived using the bug reports extracted for Git repos are displayed in the above table.

IV. CORRELATIONS

A. Correlation between Statement and Branch Coverage with McCabe Complexity

Correlation was started with a hypothesis that the project with higher complexity will likely have less code coverage.

Project	Statement coverage (1)	Branch Coverage (2)	Average Cyclomatic complexity (3)	Spearman Correlation (1&3)	Spearman Correlation (2&3)
Apache Commons Math	92%	86%	18	-0.625	-0.645
Apache Commons Lang	95%	91%	33	-0.541	0.497
JFreeChart	54%	46%	34	-0.386	0.466
Apache Commons DbUtils	64%	77%	11	-0.288	-0.322
Apache Commons Collections	86%	81%	15	-0.278	0.469

Table 3 : Spearman Correlation between 1 & 3 metrics and 2 & 3 metrics are shown in the above table.

Correlation between Statement and Branch Coverage with McCabe Complexity is observed to be not strongly related.

B. Correlation between Statement and Branch Coverage with Test Suite Effectiveness

Correlation was started with a hypothesis that code coverage is strongly correlated to mutation score and increases linearly with increase in mutation score.

Project	Spearman Correlation Coefficient between Mutation Score and Statement Coverage
Commons Collection	0.962
Commons Math	0.487
Apache Commons Lang	-0.220
JFreeChart	-0.104
Apache Commons DbUtils	-0.315

Table-4 : Correlation between Code Coverage and Test Suite Effectiveness.

Correlation between Statement and Branch Coverage with Test Suite Effectiveness is observed to be not strongly correlated , that is with the increase in coverage statement coverage doesn't necessarily ensure high test suite effectiveness.

C. Correlation between Statement and Branch Coverage with Post release defect density

Correlation between projects was started with a hypothesis that the project with low code coverage will contain more bugs.

Project	Statement Coverage	Branch Coverage	Post Release Defect Density
Apache Commons DbUtils	64	77	0.000582
Apache Commons Lang	95	91	0.00005125
Apache Commons Collections	86	81	0.0000159
Apache Commons Math	92	86	0.00006225
JFreeChart	54	46	0.00008947

Table-5 : Correlation between Statement coverage and Post release defect density are shown in the above table.

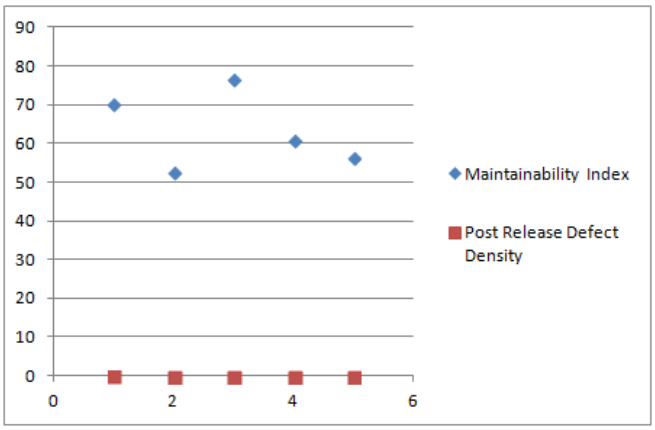
Correlation between Statement and Branch Coverage with Post release defect density appears to be strongly correlated with rho value of 0.7 ; that is with the increase in Statement coverage the post release defect density tends to decrease.

D. Correlation between Maintainability Index and Post release defect density

Correlation was started with a hypothesis that with high maintainability index we will have low post release defect density value.

Projects	Version	Maintainability Index	Post-release Defect density	Spearman coefficient
Apache Commons DbUtils	1.2 1.3 1.4 1.5 1.6	77.74 77.75 74.03 72.92 70.23	0.00262 0.00051 0.00187 0.00092 0.00058	0.14
Apache Commons Lang	3.4 3.5 3.6 3.7 3.8	45.73 51.97 51.96 52.5 52.44	0.00045 0.00017 0.00021 0.00005 0.00003	-0.9
Apache Commons Collections	3.0 4.0 4.1 4.2 4.3	69.03 66.52 65.01 76.30 76.29	0.00106 0.00095 0.00071 0.00026 0.00001	0.3
Apache Commons Math	3.2 3.3 3.4 3.5 3.6	67.58 67.48 67.06 66.95 60.56	0.00023 0.00018 0.00005 0.00001 0.00006	0.7
JFreeChart	1.0.4 1.0.6 1.0.7 1.0.8 1.5.0	63.82 63.00 56.04 63.56 62.56	0.000479 0.000089 0.000158 0.000158 0.000894	0.2

Table 6 : Correlation between MI(Maintainability Index) and Post release defect density are calculated for all the projects and the same has been included in the table above.



Correlation between Maintainability Index and Post release defect density is found to be negatively correlated with rho value of -0.1 thus implying with the increase in Maintainability Index Post release defect density tends to decrease.

V. TOOLS AND CALCULATIONS USED

A. JaCoCo:

JaCoCo is a free code coverage tool used for calculating Statement, Branch Coverage and McCabe Complexity. It is popular Java code coverage tool that can generate coverage reports using Bytecode instrumentation technique. EclEmma plugin in Eclipse is a free code coverage tool for generating coverage reports. This plugin helps in getting code coverage directly into Eclipse workspace.

B. Sci-Tools:

We have used Sci-Tools to extract SLOC, Cyclomatic complexity, and other information. Below are the steps followed to extract the needed data, which we have used in our analysis

Method used to fetch reports from PREST & Understand tools :

- Upon Installing SCI-Understand tool , project source file is imported into the tool to convert into the Understand format, then the report is fetched by selecting Sum Cyclomatic and count Line parameters from the tool and to generate the report at the class level .
- From the PREST tool, project source file is imported and required parameters are selected to fetch the report. Report generated calculates operators and operands and the corresponding Halstead values at class level are later used to calculate the Maintainability index.

C. Calculating Spearman Correlation:

Spearman's Rank correlation coefficient is one of the most-prominent techniques which can be used to find out the strength and correlation between two variables.

Method used to calculate the Spearman correlation

- Create a table from your data and get the ordered pairs of two variables.
- Rank the two data sets. Ranking is achieved by giving the ranking '1' to the biggest number in a column, '2' to the second biggest value and so on. The smallest value in the column will get the lowest ranking. This should be done for both sets of measurements or the variables used to find the correlation for.
- Tied scores are given the mean (average) rank.
- Find the difference in the ranks (d).
- Square the differences (d^2) To remove negative values and then sum them
- Calculate the coefficient (R_s) using the formula mentioned below.

When written in mathematical notation the Spearman Rank formula looks like this:

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)}$$

Here,
ρ= Spearman rank correlation
di= the difference between the ranks of corresponding variables
n= number of observations

D. PITest Tool: PITClipse

PIT stands for Performance Improvement Team. Basic idea of PIT testing tool is to run unit tests against automatically modified versions of application source code. It should produce different results and cause the unit tests to fail when there is even a slight change. If a unit test pass in this situation, it is said that test suite is not effective. We have used PITClipse plugin available in Eclipse as PIT testing tool as we are comfortable to use Eclipse for Java Projects.

VI. RELATED WORK

- We have calculated various metrics to find the correlation among them for the projects chosen. Every metric has a different means to calculate and there were existing set of tools available to measure these metrics.

- JaCoCo-based EclEmma, which is an eclipse plugin has been used to calculate the code coverage for the projects. The tool gave a detailed analysis on the results at the very granular class level. The tool also allowed us to export the reports in the desired formats including a HTML report for visual ease and CSV files for calculation ease.

- Pitclipse, which is yet another powerful Eclipse plugin has been used to calculate the mutation scores for the test suits. Just like EclEmma, this tool also provides data in both HTML as well as CSV files.

- We have used the very traditional way to calculate McCabe Cyclomatic complexity, and maintainability index which depends on the number of linearly independent paths in a program. We have the results class wise, thanks to the tools (EclEmma, and Sci-Tools) which made our job easy.

- Post release defect density is based on the number of bugs found in the system. We have calculated that on a

version-wise basis, of whose' source code has been extracted from Git repos.

VII. SUMMARY AND CONCLUDING REMARKS

We have taken five different open source projects to figure out the various correlations among the chosen metrics. After careful consideration and based on our analysis, we believe, it would be safe to conclude that a particular metric cannot give an absolute estimate about the project. Instead, metrics give valuable insights, however, there might be various other aspects to consider such as the size of the project, knowledge of the persons performing the tests, coverage, number of bugs, interval between the project release, and many more. All the independent findings and results of the correlations we made have already been stated in the 'results and observations' section of Correlation analysis. However, metrics do give a reliable means of measuring the software in their own respective manner.

VIII. REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. *(references)*

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[8] Variations of Maintainability Values formula http://www.virtualmachinery.com/sidebar4.htm

[9] http://www.projectcodemeter.com/cost_estimation/help/GL_maintainability.htm