UNIVERSITÉ Concordia UNIVERSITY

Project Proposal

SOEN-6611 Software Measurements

Submitted to

Jinqiu Yang

1. Team Information: Team J

Full Name	Student ID	Email
Suruthi Raju	40084709	suruthi77@gmail.com
Pranoti Mulay	40129435	opranoti@gmail.com
Avinash Damodaran	40078258	avinashdamu323@gmail.com
Niralkumar Hemantkumar Lad	40080612	niralhlad.concordia@gmail.com
Shalin Rohitkumar Patel	40088004	shalinpatel610@gmail.com

2. Selected Metrics and Correlation analysis:

Metric 1- Branch Coverage

Branch coverage is a requirement that, for each branch in the program (e.g., if statements, loops), each branch has been executed at least once during testing.
That leads to every branch can lead to either tTrue or False which ensures that there is no abnormal behaviour.

Branch Testing = (Number of decisions outcomes tested / Total Number of decision Outcomes) x 100%

Metric 2- Statement Coverage

Statement coverage is a white box testing technique, which involves the execution of all the statements at least once in the source code. It is a metric, which is used to calculate and measure the number of statements in the source code which have been executed. It can also be used to check the quality of the code and the flow of different paths in the program.

Statement coverage = No of statements Executed/Total no of statements in the source code x 100

Metric 3 - Mutation Score

Mutation Testing is a type of software testing where we mutate certain statements in the source code and check if the test cases are able to find the errors. It is a type of White Box Testing which is mainly used for Unit Testing. The changes in the mutant program are kept extremely small, so it does not affect the overall objective of the program.

The goal of Mutation Testing is to assess the quality of the test cases which should be robust enough to fail mutant code. This method is also called a Fault-based testing strategy as it involves creating a fault in the program.

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

If mutation score= 0% the test cases are not written correctly on the contrary mutation score=100% means that all the faults are recognized completely.

Metric 4 - McCabe Cyclomatic complexity

Cyclomatic complexity is a quantitative measure of the number of linearly independent paths through the program's source code. Cyclomatic complexity is used as a benchmark to compare two different source code. The program with high cyclomatic complexity is more error-prone and requires more understanding of testing. It also helps us in determining the number of test cases that will be required for complete branch coverage.

Cyclomatic complexity is calculated with the help of the number of edges(E), the number of nodes(N) and the number of connected points (P).

Cyclomatic Complexity = E – N + 2P

Cyclomatic complexity can also be determined with the help of number of control predicate (D):

Cyclomatic Complexity = D + 1

In general, a McCabe complexity of low is good to have, A high complexity (>10) makes the method more complex.

Metrics 5 - Structural Measures

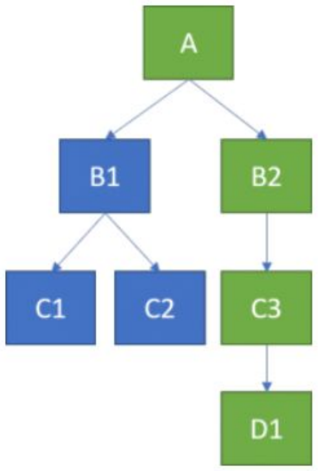
The most common set of metrics for assessing code maintainability is structural measures (SM), including the CK metrics. The subset includes the coupling measure OMMIC (call to methods in an unrelated class), the cohesion measure TCC (tight class cohesion) and the measure of the size of classes WMC1 (number of methods per class; each method has a weight of 1) and depth of inheritance tree (DIT).

Depth of Inheritance Tree (DIT):

The Depth of Inheritance Tree (DIT) metric provides for each class a measure of the inheritance levels from the object hierarchy top. The minimum value of DIT for a class is 1.
A DIT value of 0 indicates a root while a value of 2 and 3 indicates a higher degree of reuse. If there is a majority of DIT values below 2, it may represent poor exploitation of the advantages of OO design and inheritance. It is recommended a maximum DIT value of 5 since deeper trees constitute greater design complexity as more methods and classes are involved.

Method of calculation

Since ObjectScript supports multiple inheritances, for each declaration such as:
Class A Extends (B1, B2)
the Depth of Inheritance Tree is the maximum level reached by walking the dependency tree of A, then B1 and B2, recursively so until an element is encountered which does not Extend anything.
On the next example, the DIT value is 4 (the green hierarchy):



Metrics 6 - Software Defect Density:

Defect Density is defined as the number of defects per size of the software or application area of the software.

Defect Density = Total number of defects / Total Number of Volume

Or

Defect Density = Total no. of defects/KLOC

The average number of defects in a section or per KLOC of a software application is bug or defect density. The higher the bug density, the poorer the Quality.

Defect Density calculation:

1: Collect the raw material: You are going to need the total no. of defects (for a release/build/cycle).

2: Calculate the average no. of defects/Functional area or KLOC

Defect density Formula with calculation example:

Example 1: For a particular test cycle there are 30 defects in 5 modules (or components). The density would be:

Total no. of defects/Total no. of modules = 30/5 = 6. DD per module is 6.

Example 2: A different perspective would be, say, there are 30 defects for 15KLOC. It would then be:

Total no. of defects/KLOC = 30/15 = 0.5 = Density is 1 Defect for every 2 KLOC.

3. Related Work:

Metrics 1 and Metrics 2:

There are various open source tools available to compute the branch and statement coverage of a project. This includes Cobertura, Hansel, EMMA, Gretel, JCOV for JAVA, Coverage.py for python and many others. Apart from the open source tools, many commercial tools including but not limited to Atlassian Clover for JAVA and Bullseye Coverage for C++ are being used in the market.

Metrics 3:

Mutation testing, basically, is testing the test cases with minor changes in the program to check the robustness of the unit test. Therefore, the score of mutation of the program depends on how well the test cases have been written.

Metrics 4:

Cyclomatic Complexity, generally, referred to as McCabe’s Complexity, helps to improve the code coverage. Many tools are available to determine this for a project which are OCLint for C related languages and Gmetrics for Java.

Metrics 5:
Testing of the software costs more if the Depth of Inheritance Tree is more.

Metrics 6:
Defect density elects whether the software should be released or not.

4. Selected Open source systems:

Projects	Version	Lines of Code
Apache Struts	2.5.22	1,037,544
Apache XML Graphics Commons	2.4	39028
Apache Cordova	7.1.0	11308
PuTTY	0.73	131,000
log4j	2	191,000

5. Resource Planning:

Sl.No	Work Assigned	Designated person
1	Metrics 1 & 2	Avinash, Shalin
2	Metrics 3 & 4	Shalin, Avinash
3	Metrics 5 & 6	Suruthi Raju, Pranoti
4	Correlation	Pranoti, Niral
5	Related Work	Niral, Suruthi Raju

References:

1) Cyclomatic complexity [Online]. Available: https://en.wikipedia.org/wiki/Cyclomatic_complexity [Accessed: 15-Feb-2020].
2) Mccabe's Cyclomatic Complexity: Calculate with Flow Graph (Example) [Online]. Available: https://www.guru99.com/cyclomatic-complexity.html [Accessed: 15-Feb-2020].
3) M. M. S. Sarwar, S. Shahzad and I. Ahmad, "Cyclomatic complexity: The nesting problem," *IEEE,* 2014.
4) Mutation Testing in Software Testing: Mutant Score & Analysis Example [Online]. Available: https://www.guru99.com/mutation-testing.html [Accessed: 15-Feb-2020].
5) ThinkSys, “34 Software Testing Metrics & KPIs: Complete Guide,” *ThinkSys Inc*, 11-Oct-2019. [Online]. Available: https://www.thinksys.com/qa-testing/software-testing-metrics-kpis/. [Accessed: 16-Feb-2020].
6) J. Arnold, “64 Essential Software Quality Testing Metrics For Measuring Success,” *QASymphony*, 07-Jan-2019. [Online]. Available: https://www.qasymphony.com/blog/64-test-metrics/. [Accessed: 16-Feb-2020].
7) “SOFTWARE MAINTENANCE METRICS AND ITS IMPORTANCE FOR ...” [Online]. Available: http://iaeme.com/MasterAdmin/uploadfolder/SOFTWARE MAINTENANCE METRICS AND ITS IMPORTANCE FOR DERIVING IMPROVEMENT IN SOFTWARE MAINTENANCE PROJECT AN EMPIRICAL APPROACH-2-3-4/SOFTWARE MAINTENANCE METRICS AND ITS IMPORTANCE FOR DERIVING IMPROVEMENT IN SOFTWARE MAINTENANCE PROJECT AN EMPIRICAL APPROACH-2-3-4.pdf. [Accessed: 14-Feb-2020
8) “Depth of Inheritance Tree,” *cachéQuality*. [Online]. Available: https://www.cachequality.com/docs/metrics/depth-inheritance-tree. [Accessed: 16-Feb-2020].