A PROGRAM ANALYSIS TOOL

Project ID: 18-055

Project Proposal Report

Gamaarachchi G.A.C.Y	IT15111548
M.A.N.S.U.K. Uvindasiri	IT15413802
K.G.D.R Perera	IT15112538
P.K.H Palihakkara	IT15113900

Bachelor of Science (Honors) Degree in Information Technology

Department of IT / SE / ISE

Sri Lanka Institute of Information Technology

April 2018

A PROGRAM ANALYSIS TOOL

Project Id: 18-055

Project Proposal Report

(Proposal documentation submitted in partial fulfillment of the requirement for the Degree of Bachelor of Science (Honors) in Information Technology)

Bachelor of Science (Honors) Degree in Information Technology

Department of IT / SE

Sri Lanka Institute of Information Technology

April 2018

DECLARATION

We declare that this is our own work and this proposal does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other university or Institute of higher learning and to the best of our knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Student ID	Name	Signature
IT15112538	K.G.D.R Perera	
IT15113900	P.K.H Palihakkara	
IT15413802	M.A.N.S.U.K. Uvindasiri	
IT15111548	Gamaarachchi G.A.C.Y	

The above candidates are carrying out research for the undergraduate Dissertation under my supervision.

Signature of the supervisor: Date:

………………………

Mr. Dilshan De Silva

ABSTRACT

With the existing tools we can measure the quality of the source code and warnings/errors produced. Even, most of the warnings are spurious and the developers are not paying attention to the output. This is why many teams require that code performance very clearly. Through the application the research will provide a wide range of users, from students to lecturers to industry personal, an efficient and convenient manner in which to measure the complexity of both small and large Java programs, allowing them to quickly and easily remove code bottlenecks thus reducing maintenance and testing phase costs and efforts significantly through the Software measuring tool will provide a wide range of users, from students to lecturers to industry personal, an efficient and convenient manner in which to measure the complexity of both small and large Java programs, allowing them to quickly and easily remove code bottlenecks thus reducing maintenance and testing phase costs and efforts significantly.

It’s a major requirement to determine the capacity and performance of Software developers assigning to a project in terms of gaining the maximum productivity of projects in commercial scale. Hence we are focusing to develop a more interactive Software measuring tool to analyze the source code quality and the performance of Software developers assigned to a certain project.

Table of Contents

ABSTRACT	4
LIST OF FIGURES	6
LIST OF TABLES	6
1 INTRODUCTION	7
1.1 Background	7
1.2.1 Literature review	8
1.3 Research Gap and Research Problem	16
2 OBJECTIVES	19
2.2 Specific Objectives	19
3.2 System Overview	22
3.2.1 Assigning Software developers to projects	22
3.2.2 Model Construction	22
3.2.4 Results Interpretation	35
3.3 Tools and Technologies	42
3.4 Gantt Chart	43
3.5 Work breakdown Structure	44
4 DESCRIPTION OF PERSONAL AND FACILITIES	45
5 REFERENCES	49

LIST OF FIGURES

Figure 1.2.2.1: Logo of SonarQube 12
Figure 1.2.2.2: Logo of JArchitect 13
Figure 1.2.2.3: Logo of FindBugs 14
Figure 1.2.2.4: Logo of Targetprocess 15
Figure 3.1: High Level System Architecture 21
Figure 3.2.2.2: ANLTR™ logo 24
Figure 3.2.3.1.1: Indication of nodes and edges in a control flow graph 26
Figure 3.2.3.1.2: Pseudo code of insertion sort algorithm 26
Figure 3.2.3.3: Control flow graph for insertion sort 27
Figure 3.2.4: Burn down Chart 41
Figure 3.4.1: Gantt chart 43
Figure 3.13.1: Work breakdown Structure 44

LIST OF TABLES

Table 1.3 1: The Overview of the results of software metric tools review 17
Table 3.2.3 1 Standard Values of Cyclomatic Complexity 29
Table 3.2.3 2 Indicators to check Halstead complexity of a module 30
Table 3.2.3.2 1 Mathematical Formulas related to Halstead Complexity 30
Table 3.2.4 1 Composite complexity of a method 36
Table 3.2.4 2 Composite complexity of a class 36
Table 3.2.4 3 Composite complexity of the system 37
Table 3.2.4 4 Type of a statements (executable and non-executable) for a class 37
Table 3.2.4 5 Length based measures for the system 38
Table 3.2.4 6 List of operators for a given programming statement, method, class or system 38
Table 3.2.4 7 List of operands for a given programming statement, method, class or system 38
Table 3.2.4 8 The type and nesting levels for a given method, class or system 39
Table 3.2.4 9 The inheritance related information of the classes belonging to the system 39
Table 3.2.4 10 Number of decisional statements of a system 39
Table 3.2.4 11 Total execution time of a system 40
Table 3.2.4 12 Memory consumption of a system 40
Table 3.2.4 13 Communications between methods, classes and a system 41

1 INTRODUCTION

1.1 Background

Measurement is an essential part of any activity. Fundamentally, it allows any onlooker, or even the action doer themselves, to judge the success of the activity. Beyond simply measuring the degree of success, measurement enables comparison between activities, accurate prediction of the success of an activity, accurate assessment of the effort required to conduct an activity and improvement of activities based risk, effort and resources required, to name a few. The same is true for any software project or code. Simply put, “You can’t manage what you can’t measure” [1].

If you were able to measure code, you could judge how large your program is, you could compare it with other programs, you could put a price tag on your code, you could even predict the effort needed to maintain or rewrite your code. The list goes on and on. But none of that means measuring code is easy to do. As the code grows in complexity, so does the degree of difficulty in measuring the code.

The first and most fundamental part of any measurement is establishing suitable metrics. Metrics have to be meaningful, easily quantifiable, reliable and accurate. They can range from simple units such as Centimeters and meters to broader metrics such as time taken or “man days”, the sole objective being to make work done measurable. In IT, code has always been difficult to quantify. Initially the sheer volume of code was considered alone (Lines of code) and then expanded on to include variants of this metric such as commented lines of code or blank lines of code [2]. Lines of Code as a metric alone falls short of giving an accurate assessment of the size of a project or code. A code segment with 50 lines of code could still be more complex than a different code with 100 lines of code. In light of this software engineers endeavored to find more meaningful metrics. Eventually, in 1976, Thomas McCabe introduced Cyclomatic Complexity and in 1977, the Halstead Metrics were introduced by Maurice Halstead. While there have been other metrics introduced over the years, these two metric suits remain the cornerstone of all significant code complexity measurement techniques. It should be noted that measuring code is considered as measuring the complexity of code, thus the approach used by both these engineers in developing metrics based on the complexity of code.

The reasons to measure how complex a program or code is can be broken down into two main aspects. That is, the degree of difficulty to maintain the code, and the degree of difficulty to test the code. Both maintenance and testing are integral parts of any software life cycle and remain active phases long after the design and implementation phases have completed. Thus, both represent the two most costly phases of any projects life cycle. It goes without saying, the more complex a code is, the harder it is to test and maintain. Having recognized this, many organizations give measuring code complexity a prominent place in the hope of reducing the losses of maintaining and testing complex code. Some of these organizations term this cost as “Technical Debt”, and measure it in “man days” [3]. Technical debt, though hard to quantify, is a useful measure for identifying and reducing bugs, improving the quality and readability of code.

1.2 Literature Survey

We have being survey books, scholarly articles, and any other sources relevant to code complexity, static source code analysis and code analysis tools. This section provides a brief description, summary, and critical evaluation of these works in relation to the development of Software productivity measuring tool with team allocation which is being investigated.

1.2.1 Literature review

- De Oliveira, C.D., De Oliveira, C.D., Fong, E.N. and Black, P.E., 2017. *Impact of Code Complexity on Software Analysis*. US Department of Commerce, National Institute of Standards and Technology:

  In order to determine the impact of code complexity on software analysis, during investigations they have studied thousands of warnings from static analyzers. Accordingly, they saw that tools have difficulty in distinguishing between the absence of a weakness and the presence of a weakness that is buried in otherwise irrelevant code elements, which we call “code complexities” were considered as a part of test cases generation strategy when evaluating static analyzers. As the key factors, the benefits of using code complexity was given as the development of coding guidelines, boosting diversification of test cases. [9]

- Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A. and Love, T., 1979. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on software engineering*, (2), pp.96-104:

  Three software complexity measures (Halstead's E, McCabe's u(G), and the length as measured by number of statements) were compared to programmer performance on two software maintenance tasks. In an experiment on understanding, length and u(G) correlated with the percent of statements correctly recalled. In an experiment on modification, most significant correlations were obtained with metrics computed on modified rather than unmodified code. All three metrics correlated with both the accuracy of the modification and the time to completion. Relationships in both experiments occurred primarily in unstructured rather than structured code, and in code with no comments. The metrics were also most predictive of performance for less experienced programmers. Thus, these metrics appear to assess psychological complexity primarily where programming practices do not provide assistance in understanding the code. [10]

- Kan, S.H., 2002. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc...

  It presents examples of quality metrics for the three categories of metrics associated with the software life-cycle: end-product, in-process, and maintenance. It describes the metrics programs of several large software companies and discusses software engineering data collection and the application of the basic statistical tools for quality control, known as Ishikawa's seven basic tools, in software development. The potentials and challenges of applying the control chart in software environments are also discussed. It also mentioned about several quality management models that cover the entire development cycle. In-process metrics and reports that support the models are shown and discussed. It focuses on the metrics for software testing and other essential key areas under metrics and models in software quality engineering. [11]

- Sanjay Misra, "A complexity Weight based on cognitive weights", International journal, Volume 01 Number 01 (2006), pp. 1-10:

  Their main target was to develop a method to calculate the complexity of a code in terms of cognitive weights and Basic control structure. This method is fairly simple and also provides an in depth view of the contents of a program. The underlying concept is to come up with a numerical value for the complexity value of a program by applying weights for cognitive informatics and Basic control structures of a program. [15]

- Magel, K., Kluczny, R.M., Harrison, W.A. and Dekock, A.R., 1982. Applying software complexity metrics to program maintenance:

  According their estimates about 40-70% of annual software expenditures involve maintenance of existing systems. Hence they suggested that if complexities could somehow be identified, then programmers could adjust maintenance procedures accordingly. A clear specification is given about the existing complexity measures, factors affecting code maintenance such as program size, data structures, data flow, and flow of control. [12]

- Bandi, R.K., Vaishnavi, V.K. and Turk, D.E., 2003. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Transactions on Software Engineering*, *29*(1), pp.77-87:

  The purpose of this research is to empirically explore the validation of three existing OO design complexity metrics and, specifically, to assess their ability to predict maintenance time. This research reports the results of validating three metrics, Interaction Level (IL), Interface Size (IS), and Operation Argument Complexity (OAC). A controlled experiment was conducted to investigate the effect of design complexity (as measured by the above metrics) on maintenance time. Each of the three metrics by itself was found to be useful in the experiment in predicting maintenance performance. [13]

- Girish H. Subramanian, Parag C. Pendharkar and Mary Wallace, "An empirical study of the effect of complexity, platform, and program type on software development effort of business applications", Empir Software Eng (2006), 17 October, 2006:

  This particular research aims at calculating the impact of platform, program and software complexity on software development effort. Through the use of data on 666 programs from 15 software projects the study goes on to show the significant impact that all three adjustment variables have on the software effort. In relation to our own research topic, this study shows in particular that programs that are more complex require significantly more effort than programs rated with low or average complexity. [14]

- Boehm, B.W., Brown, J.R. and Lipow, M., 1976, October. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering* (pp. 592-605). IEEE Computer Society Press

  Through this research investigations, they establishes a conceptual framework and some key initial results in the analysis of the characteristics of software quality. Its main results and conclusions were explicit attention to characteristics of software quality that can lead to significant savings in software life-cycle costs, current software state-of-the-art which imposes specific limitations on ability to automatically and quantitatively evaluate the quality of software, a definitive hierarchy of well-defined, well-differentiated characteristics of software quality is developed. Its higher-level structure reflects the actual uses to which software quality evaluation would be put; its lower-level characteristics are closely correlated with actual software metric evaluations which can be performed. In additions, a large number of software quality-evaluation metrics have been defined, classified, and evaluated with respect to their potential benefits, quantifiability, and ease of automation and particular software life-cycle activities have been identified which have significant leverage on software quality. [22]

- Usha Chhillar, Shuchita Bhasin, “A New Weighted Composite Complexity Measure for Object-Oriented Systems”, International Journal of Information and Communication Technology Research, Volume 1 No. 3, Department of Computer Science, Kurukshetra University, Kurukshetra, Haryana, India, July 2011:

  Evaluating and controlling software complexity is one of most important objectives of software development because it directly affects many nonfunctional requirements of a software such as Readability and Maintainability as well as functional requirements. This study focuses on improving the accuracy of complexity values by applying weights to aspects of complexity. [23]

- Jones, C.C., 1997. *Software quality: Analysis and guidelines for success*. Thomson Learning:
  Based on the author's empirical observations of nearly 7000 software projects derived from close to 600 companies and government agencies, Software Quality: Analysis and Guidelines for Success examines the effects of about 75 major software quality factors on the quality level of actual software applications. This book also compares the quality methods used by six major software sub industries, including commercial and systems software, management information systems and outsources. Software Quality: Analysis and Guidelines for Success brings together the major issues in software quality that impacts the enterprise, including cost benefit and analysis that shows costs of using quality standards and the higher cost of systems failures and ineffectiveness when they are not used. It demonstrates the need for a multi-faceted approach to achieve high-levels of software quality, utilizing quality measurements, protest inspections and testing by trained testing experts as tools to achieve success. [24]

1.2.2 Existing Tools

1. SonarQube

Figure 1.2.2.1: Logo of SonarQube

SonarQube collects and analyzes source code, measuring quality and providing reports for your projects. It combines static and dynamic analysis tools and enables quality to be measured continuously over time. Everything that affects our code base, from minor styling details to critical design errors, is inspected and evaluated by SonarQube, thereby enabling developers to access and track code analysis data ranging from styling errors, potential bugs, and code defects to design inefficiencies, code duplication, lack of test coverage, and excess complexity. [25]

Features:

- SonarQube doesn't just show you what's wrong. It also offers quality-management tools to actively help you put it right
- SonarQube commercial competitors seem to focus their definition of quality mainly on bugs and complexity, whereas SonarQube offerings span what its creators call the Seven Axes of Quality
- SonarQube addresses not just bugs but also coding rules, test coverage, duplications, API documentation, complexity, and architecture, providing all these details in a dashboard
- It gives you a moment-in-time snapshot of your code quality today, as well as trends of lagging (what's already gone wrong) and leading (what's likely to go wrong in the future) quality indicators
    - It provides you with metrics to help you take right decision. In nearly every industry, serious leaders track metrics. Whether it's manufacturing defects and waste, sales and revenue, or baseball hits and RBIs, there are metrics that tell you how you're doing: if you're doing well overall, or whether you're getting better or worse.

Limitations:

- Upgrading the version of the server is a bit cumbersome and could be made slightly easier. Allowing admin users to upgrade the software through the front-end would make upgrading easier.

- Another improvement is with false positives. Sometimes the tool can say there is an issue in your code but, really, you have to do things in a certain way due to external dependencies, and it's very hard to indicate this is the case. There is a way to mark the code/method with the issue number, but having to add comments/annotations in your code for your static analysis tool gives a bad user experience in UI/UX perspective.
- Unable to have different groups or projects within a single SonarQube server for different environments in a project development life cycle.
- A better design of the interface and add some new rules.
- When we have a thousand products published over it, we expect it to be more efficient in terms of serving requests from the browser.
- Ease of use/interface.
- It requires advanced heuristics to recognize more complex constructs that could be disregarded as issues.
- There is need for support for the additional languages and ease of use in adding new rules for detecting issues.

- **Scalability Issues**
  With the upgrade to version 6, they have moved the processing of the stats from outside the server to inside the server. As a result the machines running SonarQube are using a lot more resources, as the processing is done server side. This means that we need to increase the resources allocated to the machine. If an application is running this in the cloud, it would be easy, as if we would create a larger instance for the service. But if we have this running on a physical machine, it's limited to what we can allocate.

2. JArchitect

Figure 1.2.2.2: Logo of JArchitect

JArchitect is a static analysis tool for Java code. This tool supports a large number of code metrics, allows for visualization of dependencies using directed graphs and dependency matrix. The tools also performs code base snapshots comparison, and validation of architectural and quality rules. User-defined rules can be written using LINQ queries. This possibility is named CQLinq. The tool also comes with a large number of predefined CQLinq code rules. [26]

Features:

- Dependency Visualization (using dependency graphs, and dependency matrix)
- Software metrics (JArchitect currently supports 82 code metrics: Cyclomatic complexity; Afferent and Efferent Coupling; Relational Cohesion; Percentage of code covered by tests, etc.).
- Declarative code rule over LINQ query (CQLinq).
- JArchitect can tell you what has been changed between 2 builds.

Limitations:

- Documentation issues.
- Cost for small developers.

3. FindBugs

Figure 1.2.2.3: Logo of FindBugs

FindBugs is an open source static code analyzer created by Bill Pugh and David However which detects possible bugs in Java programs. The analyzer got itself a successor: Spot Bugs. Potential errors are classified in four ranks:

(i) Scariest,

(ii) Scary,

(iii) Troubling and

(iv) Of concern.

This is a hint to the developer about their possible impact or severity. FindBugs operates on Java bytecode, rather than source code. The software is distributed as a stand-alone GUI application. There are also plug-ins available for Eclipse, NetBeans, IntelliJ IDEA, Gradle, Hudson, Maven, Bamboo and Jenkins. [27]

Features:

- Uses Static Analysis to Locate Bugs
- Supports Plug-In Architecture

Limitations:

- FindBugs can report false warnings, which are warnings that do not indicate real errors.
- Hard to write code and maintain.

4. Targetprocess

targetprocess

SEE. CHANGE.

Figure 1.2.2.4: Logo of Targetprocess

Targetprocess is a project management software solution to help you visualize and manage Agile projects and other complex work. It provides full support for Kanban, Scrum, SAFe, NEXUS, SoS, or customized Agile methods as well as other frameworks and approaches, allowing you to reduce clutter and focus on important workflows and processes. It gives enhanced visualization functionality to help you get visibility across teams, projects, and the entire company. As with most project management solutions, Targetprocess has a big-picture and drill-down dashboard feature and uses a hierarchical structure to organize tasks and subtasks.

Targetprocess is fully customizable , flexible, and can adapt to your organization’s structure and company’s management style. Although it is designed for companies in any industry with 50 to 5,000 IT employees, it can also be used to manage non-IT projects and processes like recruiting, marketing, and education, providing you transparency and visibility over data and workflows. [28]

1.3 Research Gap and Research Problem

Software metric can be defined as measure that reflects some property of a software product or its specification. Software metric value can also be related to only one unit of a software product. There are numerous categorizations of software metrics but considering the measurement, target metrics could be divided in three main categories: product metrics, process metrics and project metrics [17]. In this point, we shall deal with the product metrics and especially code metrics as its sub-category. Although metric values could be calculated manually, nowadays software metric tools are being used for calculation of metric values and for their further processing and analysis. Software metrics and software metric tools are wide research areas and improvements in these fields may bring higher success of software projects in general. However, the state of the art in the field shows that there is no wider acceptance of techniques and therefore still no significant improvements. A new software metrics tool with advanced features would play important role in these improvements.

Motivations behind designing a new tool lay in numerous reports on weaknesses of existing tools both from practice and from academic world (e.g. [18], [19], [20], [21]) of which we enumerate some:

• Software metrics tools are generally not independent on programming language and/or underlying platform. Therefore, different tools are often used for different projects, for different components of one project, or even within a single software component.

• Different tools sometimes provide inconsistent results. Therefore, usage of different tools for different software components is not only hard but also often unusable.

• Tools usually compute only a selection of possible metrics. They also rarely combine them to gain higher measure quality and also rarely store the code/metrics values to track changes over time.

• Tools rarely display values in ‘user-friendly’ way to a non-specialist (e.g., graphically) and rarely interpret the meaning of computed numerical results and their correlations. They almost never suggest what typical actions should be taken in order to improve the quality of the code.

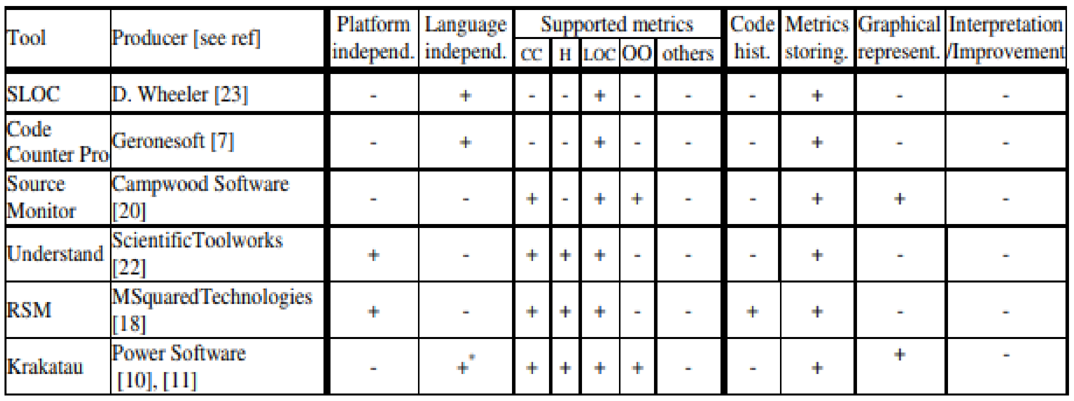
• Tools are typically insensitive to the existence of additional, useless and duplicate code that can be present for tracking, testing and debugging purposes.

• Tools usually do not deal with attempts to ‘cheat’ the metrics algorithm. Cheating is possible if the internal characteristics of the employed techniques/tools are known

- in such cases programmers can spend more time adjusting the code to the tool, rather than reaching the real program quality.

• Sometimes it is not clear which specific software metric has to be applied to accomplish the specific goal. Frequently, the reason for this confusion lies in the gap between the real quality parameters for

Table 1.3 1: The Overview of the results of software metric tools review



Tool	Producer [see ref]	Platform independ.	Language independ.	Supported metrics					Code hist.	Metrics storing.	Graphical represent.	Interpretation /Improvement
				CC	H	LOC	OO	others				
SLOC	D. Wheeler [23]	-	+	-	-	+	-	-	-	+	-	-
Code Counter Pro	Geronesoft [7]	-	+	-	-	+	-	-	-	+	-	-
Source Monitor	Campwood Software [20]	-	-	+	-	+	+	-	-	+	+	-
Understand	ScientificToolworks [22]	+	-	+	+	+	-	-	-	+	-	-
RSM	MSquaredTechnologies [18]	+	-	+	+	+	-	-	+	+	-	-
Krakatau	Power Software [10], [11]	-	+*	+	+	+	+	-	-	+	+	-

• Cyclomatic Complexity (CC) that reflects structure complexity based on control-flow structures in the program.

• Halstead Metrics (H) that reflects complexity of the program based on number of operators and operands.

• Lines of Code (LOC) that represents length of the source code expressed in number of the lines of the source code. It is common to make difference between number of the lines of comment (CLOC), source code (SLOC), etc. In this analysis, if some tool calculates any of LOC metric, than corresponding cell contains "+" symbol.

• Object Oriented Metrics (OO) – big family of metrics related to object-orientation of the program. If some tool supports any of OO metrics then corresponding cell contains "+" symbol.

• Others – if any metric that does not belong to any of listed categories is supported.

The most important conclusions of the analysis follow:

• Analyzed tools could be divided in two categories. The first category includes tools that calculate only simple metrics as are metrics from LOC family, but for wide set of programming languages. The second category of tools is characterized with wide range of metrics, but limited to a small set of programming languages. There are attempts to bridge the gap between these categories, but without final success. This is the big limitation not only for reasons noted in the introductory section, but also

because there are many legacy software systems written in ‘ancient’ languages to which modern metrics tools cannot be applied uniformly.
• Even if tools support some object-oriented metrics, the amount of supported OO metrics is fairly small. Especially comparing to the broad application of the object-oriented approach in current software development.
• Many techniques/tools compute numerical results with no real interpretation of their meaning. The only interpretation of numerical results which can be found is graphical. These results possess little or no value to practitioners who need suggestion or advice how to improve their project based on metrics' results.

And our new software metrics tool is being constructed to remedy some of the flaws by achieving the following goals:

- Ability to assign (suggest) the best developers (both at an individual level and as teams) that will best suit the particular project.
- Measures source code quality of a Java code based on static analysis and machine learning
- Identify the areas, each developer need to improve
- Storing the source code history
- Storing the calculated metrics values
- Interpretation of metrics results to the end user
- Rate the performance of Software Developers
- Identifying there strong and weak points
- Improvement recommendations to the user

2 OBJECTIVES

2.1 Main Objectives

Objective 1: To create a comprehensive tool to analyze and increase the maintainability of the software, measure and effectively display complexity of code using complexity metrics, clearly identifying complex code segments and plan the productivity and maintainability of the software product.

Objective 2: To create a comprehensive tool that will check the quality of a program to industry or company specific quality standards. The standards will be selectable by the user through an interactive interface and developers rank according their productivity.

2.2 Specific Objectives

- Extensive review of existing complexity matrices and established metrics.
- Review of existing code analysis tools and their advantages and disadvantages.
- Establishing metric thresholds through literature review and surveys.
- Extensive review of code parsing techniques and the use of grammars.
- Selection of a suitable grammar.
- Design and construction of a well-structured framework of complexity metrics for the analysis of program code in terms of program structure.
- Implementation of a user interface allowing the user to input their program code.
- Implementation of a customizable user interface allowing the user to select the quality standards they wish to check their code for adherence of.
- Extraction of raw metric data from the program code through the use of the chosen parser.
- Measuring the raw data according to the established framework.
- Representation of the calculated metric values in numeric.
- Interpretation of the numerical values into a more user-understandable format.
- Graphical representation of the interpreted results.
- Output visual structure of the program with annotated complexity metrics.
- Output based on Coupling derived from program structure.

- Output based on Exceptions derived from program structure.
- Output based on Control Structures derived from program structure.
- Output based on Inheritance derived from program structure.
- Output should include suggestions on how to improve the quality of code and reduce complexity.
- Output should present the original code with high complexity areas highlighted.
- Output should present degree of adherence to quality standards as input.

3 METHODOLOGY

This section outlines the logical processes and its associated components that will be utilized to develop a Software productivity measuring tool with team allocation which determine the performance of software developers based on their code quality using software metrics, code complexity measuring techniques and user defined quality standards.

3.1 System Architecture

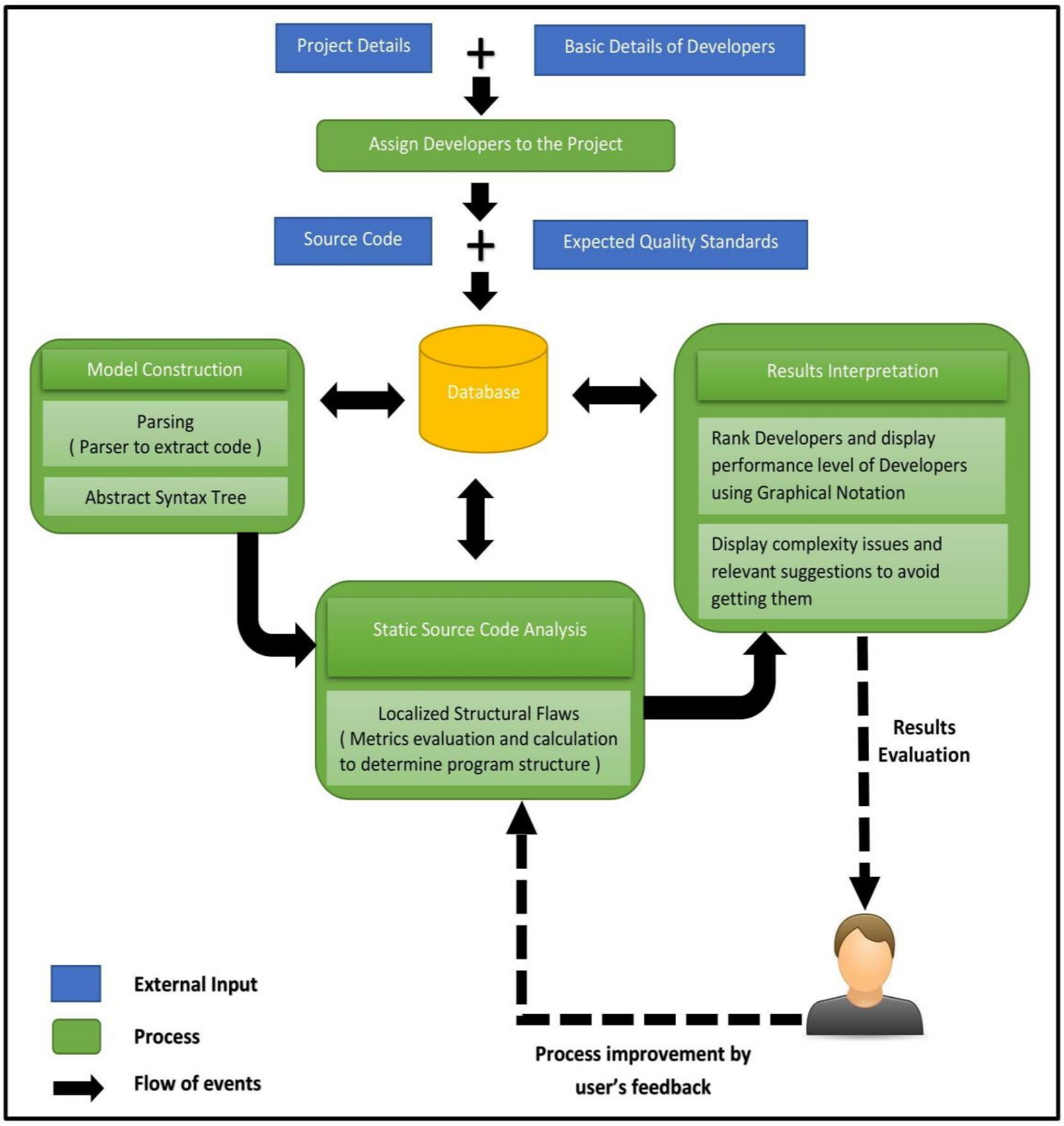

Project Details
+
Basic Details of Developers
Assign Developers to the Project
Source Code
+
Expected Quality Standards
Model Construction
Parsing
(Parser to extract code)
Abstract Syntax Tree
Database
Results Interpretation
Rank Developers and display performance level of Developers using Graphical Notation
Display complexity issues and relevant suggestions to avoid getting them
Static Source Code Analysis
Localized Structural Flaws
(Metrics evaluation and calculation to determine program structure)
Results Evaluation
Process improvement by user's feedback
External Input
Process
Flow of events

Figure 3.1.High Level System Architecture

3.2 System Overview

This specify the core functionalities of the Software productivity measuring tool with team allocation that will be developed as a desktop application. With Parallel to the development of the tool, a website will be implemented as a user reference/guide including facts about how to use the tool, system installation requirements and its features etc. Main functionalities of the Software productivity measuring tool with team allocation includes assigning Software developers to projects, model construction, static source code analysis using algorithms and results interpretation.

3.2.1 Assigning Software developers to projects

Initially user has to input basic information of software developers working in the company and the project details. The user is capable of assigning suitable software developers as per project requirements by using previously tracked performance level (rank achieved by the software developer) pertained in the system. This process can be automated or customized by the user. Then the user can input the source code and expected quality standards to analyze. Model construction phase will be initiated when user input the source code and expected quality standards.

3.2.2 Model Construction

This functionality is concerning about parsing the source code and expected quality standards input by the user to analyze the source code. Code parsing is a part of analysis performed by the compiler where the character stream of the source code breaks up into constituent pieces and imposes a grammatical structure on them. Analysis operates as a sequence of phases such as lexical analyzer, syntax analyzer (parser) and semantic analyzer. [29]
The lexical analyzer reads the stream of characters making up the source code and group the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form:
<token-name, attribute-value>

Example:

Suppose a source code contains the assignment statement as follows.

a = b + c;

The representation of this assignment statement after lexical analysis as a sequence of tokens <id, 1> <=> <id, 2> <+> <id, 3>
Where, id is an abstract symbol standing for identifier.

In a token, the first component is an abstract symbol that is used during syntax analysis, and the second component points to an entry in the symbol table.

The second phase is syntax analysis or parsing. Parsing is the process of conversion of token stream into a syntax tree. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree like intermediate representation that depicts the grammatical structure of the token stream. [30]

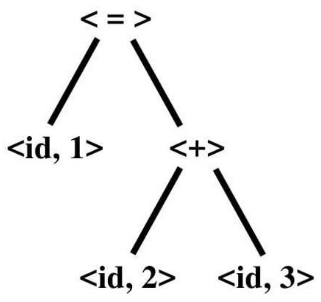


Figure 3.2.2.1: The abstract syntax tree for the sequence of tokens<id, 1> <=> <id, 2> <+> <id,3>

In this approach, Natural language processing (NLP) can be used for code parsing. NLP is the ability of a computer program to understand human language as it is spoken. NLP is a component of artificial intelligence. Rather than developing a parser from scratch with the use of NLP, existing parsing tools were reviewed for possible integration.

For the implementation of Software productivity measuring tool with team allocation, Eclipse IDE was chosen. Hence the parser tool had to be able to integrate with Eclipse IDE. Accordingly, ANLTR (Another Tool for Language
Recognition) can be used for the parsing code. ANTLR is limited to structured data only, but as we will be parsing Java code (which is extremely well structured) this is a non-existent drawback for this particular research

ANTLR is limited to structured data only, but as we will be parsing Java code (which is extremely well structured) this is a non-existent drawback for this particular research. ANTLR offers many features that would prove useful for the development of the complexity tool.

Significant features of ANLTR are as follows.

1. Automatic building of parser trees.

   This feature could be used to construct the visual representation of the program structure in form of a tree.

2. Automatically generated tree walkers.

   This feature could allow users to visit nodes in the tree, which would essentially be classes, and expand upon the node and see detailed descriptions of complexity such as coupling level or control structure types and nesting.

3. Customized grammar development environment.
4. Open source free software.
5. Can be used as a plugin with Eclipse IDE

Because of the above mentioned features, we decided to use ANLTR to parse the code.

ANTLR

Figure 3.2.2.2: ANLTR™ logo

3.2.3 Static source code analysis

After parsing the code and generating an abstract syntax tree, static source code analysis is carry out based on the program structure. We are focusing on following factors of an object oriented program.

- Polymorphism
  - Method overriding
  - Method overloading
- Encapsulation
- Tokens
  - Number and type of the operators used – Arithmetic, relational, bitwise, logical, assignment, decisional, and dot operators
  - Number and type of the operands used – Identifier, return type of a method, string, numerical values

- Control structures

    - Number of control structures
    - Number of conditions checked by a single control structure
    - The types of control structure
    - The nesting levels of control structures

- Physical length
    - LOC: count of all the lines in a program except for blank and commented lines
    - lLOC: count of all the lines in a program that are ending with semicolons
    - eLOC: Count of all the lines in a program except for blank lines, commented lines and lines with only brackets
    - Commented lines
      Number of executable and non-executables statements
- Inheritance
    - Inheritance level of each class
    - A count and names of immediate parent class(es)
    - A count and names of immediate child class(es)
    - A count and names of ancestor class(es)
    - A count and names of descendant class(es)

- Coupling
    - Call to a regular method in the same class
    - Call to a regular method in a different class
    - Call to a recursive method in a same class
    - Call to a recursive method in a different class
    - Referencing a data member of the same class
    - Referencing a data memberr in a different class
    - Calling from another method in the same class
    - Calling from another method in a different class
    - Call to a recursive method in a same class
    - Referencing of a data member in the current class by a method of a different class
    - Coupling types – Data, stamp, control, global and context

- Variable types
- Exception handling

Using complexity metrics we suppose to determine the blank lines, commented lines and lines with only brackets (parenthesis) in the program, the size of a method, class or system in terms of lines of code (count of all the lines in a program except for blank and commented lines), the size of a method, class or system in terms of effective lines of code (count of all the lines in a program except for blank lines, commented lines and lines with only brackets), the size of a method, class or system in terms of logical lines of code (count of all the lines in a program that are ending with a semicolon), obtain the executable code of the program by filtering the non-executable codes, the number of decisional statements in a method, class and system and the following for each executable statement (Composite Complexity measure):

a. Operators and operands
b. Type of control structures and their nesting levels
c. Inheritance level of each statement.

Software complexity analysis can be considered as a key issue essential to improve the code quality, reduce the maintenance cost, increase the robustness and meet the architecture standards.

Measuring software complexity increases the ability to predict the effort and code efficiency, reduces the risk of introducing defects into production, preserves the quality of software and extends its lifetime and optimize the galloping cost associated with software maintenance. Typically, reducing model complexity has a significant impact on maintenance activities. A lot of metrics have been used to measure the complexity of source code such as Halstead, McCabe Cyclomatic, Lines of Code, and Maintainability Index, etc. In our approach, we propose a hybrid module which consists of two theories such as Halstead and McCabe. Both theories will be used to analyze a code written in Java.

3.2.3.1 McCabe Complexity Measures

McCabe Complexity (Cyclomatic complexity) was developed by Thomas J. McCabe in 1976. This software metric, measures independent paths through program source code. An 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 and it is based on a control flow representation of the program. Control flow depicts a program as a graph which consists of Nodes and Edges.

In the graph, Nodes represent processing tasks while edges represent control flow between the nodes. [31]

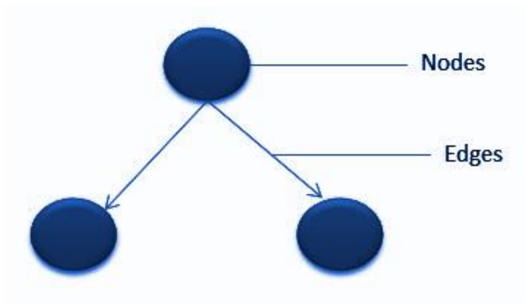


Figure3.2.3.1.1: Indication of nodes and edges in a control flow graph

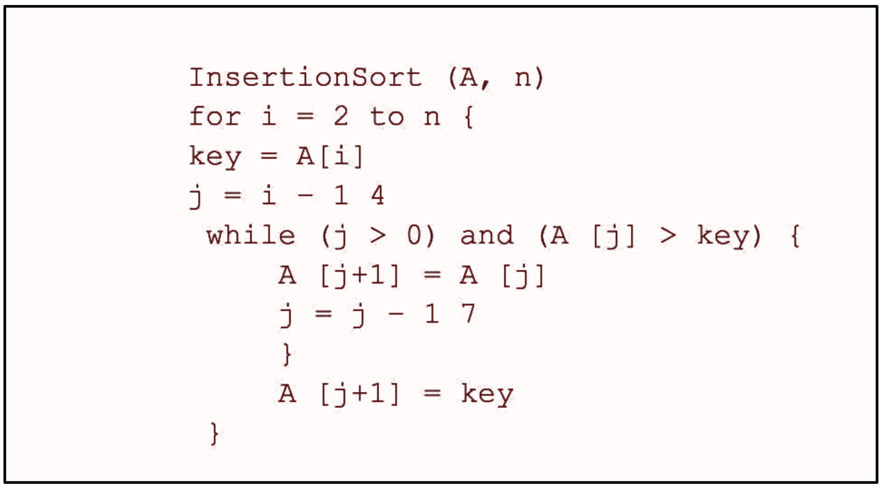
3.2.3.1.1 Mathematical Representation of McCabe Complexity

The complexity of the program can be defined as:

$$V(G) = E - N + 2$$

Where,
E - Number of edges
N - Number of Nodes

Example:



```
InsertionSort (A, n)
for i = 2 to n {
key = A[i]
j = i - 1 4
 while (j > 0) and (A [j] > key) {
     A [j+1] = A [j]
     j = j - 1 7
     }
     A [j+1] = key
 }
```

Figure 3.2.3.1.2: Pseudo code of insertion sort algorithm

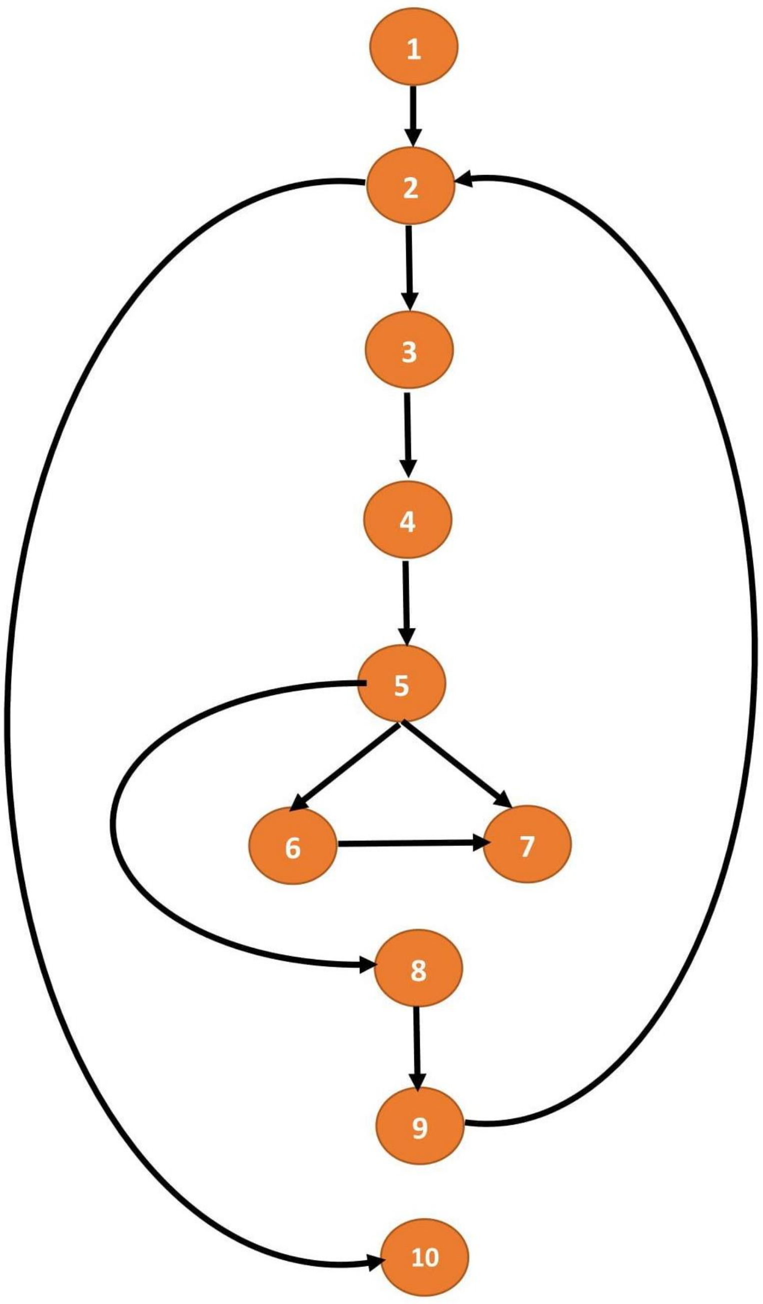


Figure 3.2.3.3: Control flow graph for insertion sort

Figure 3.2.3.3 shows that E = 11 and N = 10, therefore Cyclomatic complexity is computed as: V (G) = 11 – 10 + 2 = 3.

This means that there are 3 independent paths through the method. Furthermore it implies that we need at least 3 different test cases to test all the different paths through the code.

Cyclomatic Complexity also is useful in determining the testability of a program. Often, the higher the value, the more difficult and risky the program is to test and maintain.
Standard values of Cyclomatic Complexity are shown in Table 3.2.3.1.

Table 3.2.3 1: Standard Values of Cyclomatic Complexity

Complexity Number	Risk Complexity
1-10	Structured and well written code High Testability Cost and Effort is less
11-20	Complex Code Medium Testability Cost and effort is Medium
21-40	Very complex Code Low Testability Cost and Effort are high
>40	Not at all testable Very high Cost and Effort

3.2.3.2 Halstead Complexity Measures

Halstead complexity metrics were developed by Maurice Howard Halstead in 1977. Halstead's goal was to identify measurable properties of software, and the relations between them by determining a quantitative measure of complexity directly from the operators and operands in the module to measure a program module's complexity directly from source code.
Halstead metrics is based on interpreting the source code as a sequence of tokens and classifying each token to be an operator or an operand. [32]

Table 3.2.3 2: Indicators to check Halstead complexity of a module

Parameter	Definition
n1	Number of unique operators
n2	Number of unique operands
N1	Number of total occurrence of operators
N2	Number of total occurrence of operands

3.2.3.2.1 Mathematical Representation of Halstead Complexity

Mathematical representation of Halstead Complexity will be as follows.

Table 3.2.3.2 1: Mathematical Formulas related to Halstead Complexity

Metric	**Definition**	**Mathematical Formula**
n	Vocabulary	n1 + n2
N	Size	N1 + N2
V	Volume	Length * Log2 Vocabulary
D	Difficulty	(n1/2) * (N1/n2)
E	Efforts	Difficulty * Volume
B	Errors	Volume / 3000

T	Testing time	Time = Efforts / S, where S=18 seconds.

In this approach, we'll be analyze the source code using McCabe and Halstead Complexity measures and suppose to generate a common result using both techniques.

3.2.3.3 Source Code Quality Standards

Quality Standards (Coding conventions) are a set of guidelines that are developed by development teams and include the recommendations for the programming style, practices, and methods for each aspect of a code that is written within the company or certain project. These conventions are usually specific for every programming language and cover file organization, indentation, comments, declarations, statements, white spaces, naming conventions, programming practices and principles, programming rules, architectural best practices, etc. The main advantage of defined standards is that every piece of code looks and feels familiar. It makes it more readable and helps programmers understand code written by another programmer.

If the coding standards are followed by and applied consistently throughout the development process, in future, it will be easier to maintain and extend the code, refactor it, and resolve integration conflicts. The standard itself matters much less than adherence to it.

Code conventions are important to programmers for a number of reasons:

- 40% – 80% of the lifetime cost of software goes to maintenance.
- Hardly any software is maintained for its whole life by its author.
- Code conventions improve the readability of the software, allowing programmers to understand the new code more quickly.

In addition to the code analysis using complexity metrics, our expectation is to analyze the source code input by the user relevant to expected quality standards that the user required, providing the ability to customize the code with project-specific rule extensions. To analyze the source code with respect to the expected quality standards, process and project metrics are being used.

Process metrics are standard measurements that are used to evaluate and benchmark the performance of project management and development processes. It is common for operational processes to be heavily optimized in a cycle of measurement, improvement and measurement. [33]

The process metrics which are concerning in the development of Software productivity measuring tool with team allocation are as follows.

- Cost of quality: It is a measure of the performance of quality initiatives in an organization. It's expressed in monetary terms.

  Cost of quality = (review + testing + verification review + verification testing + QA + configuration management + measurement + training + rework review + rework testing)/ total effort x 100

- Cost of poor quality: It is the cost of implementing imperfect processes and products.

  Cost of poor quality = rework effort/ total effort x 100.

- Defect density: It is the number of defects detected in the software during development divided by the size of the software (typically in KLOC or FP)

  Defect density for a project = Total number of defects/ project size in KLOC or FP

- Review efficiency: defined as the efficiency in harnessing/ detecting review defects in the verification stage.
  Review efficiency = (number of defects caught in review)/ total number of defects caught) x 100
- Testing Efficiency:
  Testing efficiency = 1 – ((defects found in acceptance)/ total no. of testing defects) x 100

- Defect removal efficiency: Quantifies the efficiency with which defects were detected and prevented from reaching the customer.

  Defect removal = (1 – (total defects caught by customer/ total no of defects)) x 100

  Efficiency
- Residual defect density = (total number of defects found by customer)/ (Total number of defects including customer found defects) x 100

Project metrics are metrics that pertain to Project Quality. They are used to quantify defects, cost, schedule, productivity and estimation of various project resources and deliverables. In this approach, following project metrics are used in evaluating quality standards. [33]

- Schedule Variance: Any difference between the scheduled completion of an activity and the actual completion is known as Schedule Variance.

  Schedule variance = ((Actual calendar days – Planned calendar days) + Start variance)/ Planned calendar days x 100

- Effort Variance: Difference between the planned outlined effort and the effort required to actually undertake the task is called Effort variance.

  Effort variance = (Actual Effort – Planned Effort)/ Planned Effort x 100

- Size Variance: Difference between the estimated size of the project and the actual size of the project (normally in KLOC or FP).

  Size variance = (Actual size – Estimated size)/ Estimated size x 100
- Requirement Stability Index: Provides visibility to the magnitude and impact of requirements changes.

RSI = 1- ((No of changed + No of deleted + No of added) / Total no of Initial requirements) x100.

- Productivity (Project): It is a measure of output from a related process for a unit of input.

  Project Productivity = Actual Project Size / Actual effort expended in the project
- Productivity (for test case preparation) = Actual no of test cases/ Actual effort expended in test case preparation.
- Productivity (for test case execution) = Actual number of test cases / actual effort expended in testing.
- Productivity (defect detection) = Actual number of defects (review + testing) / actual effort spent on (review + testing).
- Productivity (defect fixation) = actual no of defects fixed/ actual effort spent on defect fixation.
- Schedule variance for a phase: The deviation between planned and actual schedules for the phases within a project.
- Schedule variance for a phase = (Actual Calendar days for a phase – Planned calendar days for a phase + Start variance for a phase)/ (Planned calendar days for a phase) x 100
- Effort variance for a phase: The deviation between planned and actual effort for various phases within the project.
- Effort variance for a phase = (Actual effort for a phase – planned effort for a phase)/ (planned effort for a phase) x 100

3.2.4 Results Interpretation

Upon Static code analysis and user feedback, results will be interpreted. The following content describes the way that we proposed to display results. In addition to providing a detail analysis regarding a program, the proposed to will display the complexity (based on Cyclomatic Complexity measure (CC) and Composite Complexity measure (CCM)) and the number of lines of code (LOC) in a package, class or method for the user to gain a much clear understanding about a certain program. complexity values and the LOCof a package, class or method would be displayed as follows
Example:

Package Test → If package Test is selected, can display the following:
↳ LOC = 358, CC = 68, CCM = 667

Class DD Name → If class DD selected, can display the following:
↳ LOC = 56, CC = 20, CCM = 153

Method A → If method A is selected, can display the following:
↳ LOC = 20, CC = 5, CCM = 24

Method B

Method C

Class SS Name

Method P

Method Q

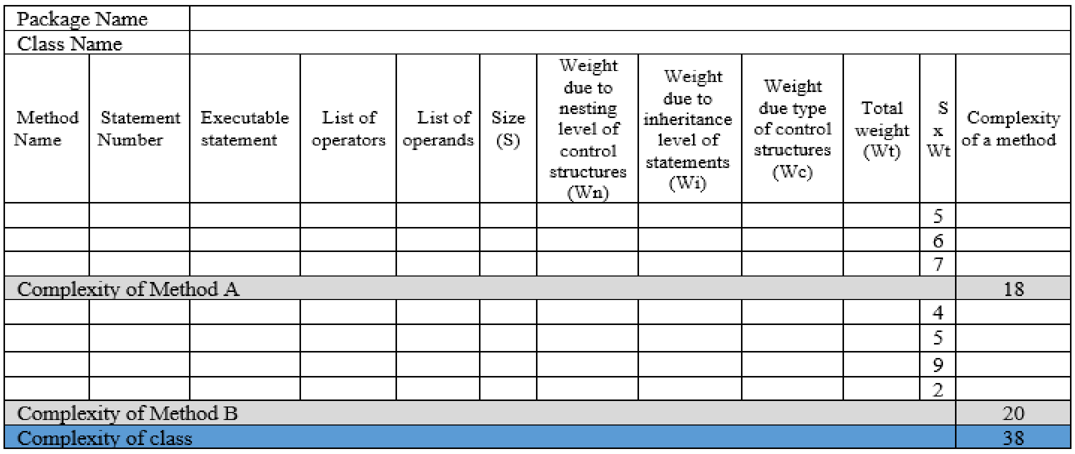
Method R

The composite complexity of a method, class and system based on user's requirement will be interpreted as follows.

Table 3.2.4 1: Composite complexity of a method

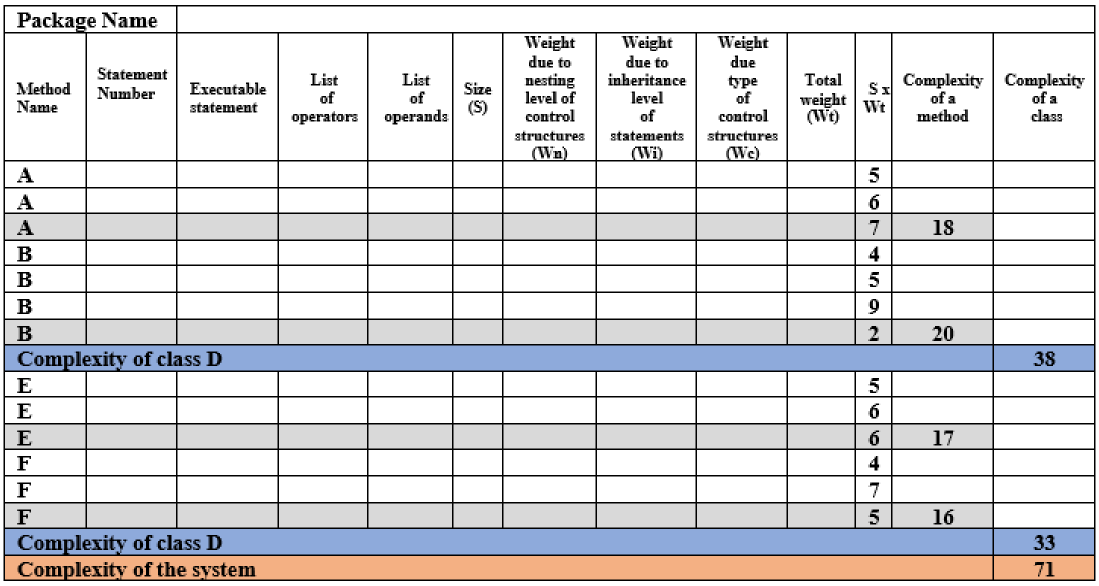
Package Name									
Class Name									
Statement Number	Executable statement	List of operators	List of operands	Size (S)	Weight due to nesting level of control structures (Wn)	Weight due to inheritance level of statements (Wi)	Weight due type of control structures (Wc)	Total weight (Wt)	S x Wt
Complexity of method									

Table 3.2.4 2 Composite complexity of a class



Package Name											
Class Name											
Method Name	Statement Number	Executable statement	List of operators	List of operands	Size (S)	Weight due to nesting level of control structures (Wn)	Weight due to inheritance level of statements (Wi)	Weight due type of control structures (Wc)	Total weight (Wt)	S x Wt	Complexity of a method
										5	
										6	
										7	
Complexity of Method A											18
										4	
										5	
										9	
										2	
Complexity of Method B											20
Complexity of class											38

Table 3.2.4 3: Composite complexity of the system

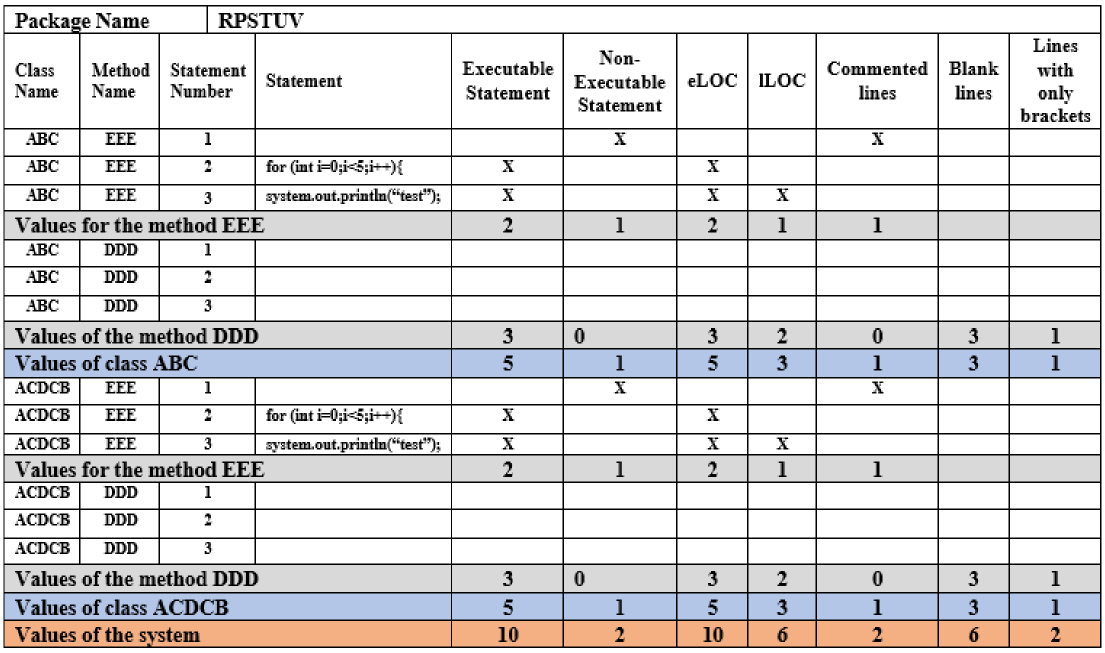


Package Name												
Method Name	Statement Number	Executable statement	List of operators	List of operands	Size (S)	Weight due to nesting level of control structures (Wn)	Weight due to inheritance level of statements (Wi)	Weight due type of control structures (Wc)	Total weight (Wt)	S x Wt	Complexity of a method	Complexity of a class
A										5		
A										6		
A										7	18	
B										4		
B										5		
B										9		
B										2	20	
Complexity of class D												38
E										5		
E										6		
E										6	17	
F										4		
F										7		
F										5	16	
Complexity of class D												33
Complexity of the system												71

Table 3.2.4 4: Cyclomatic complexity of a system

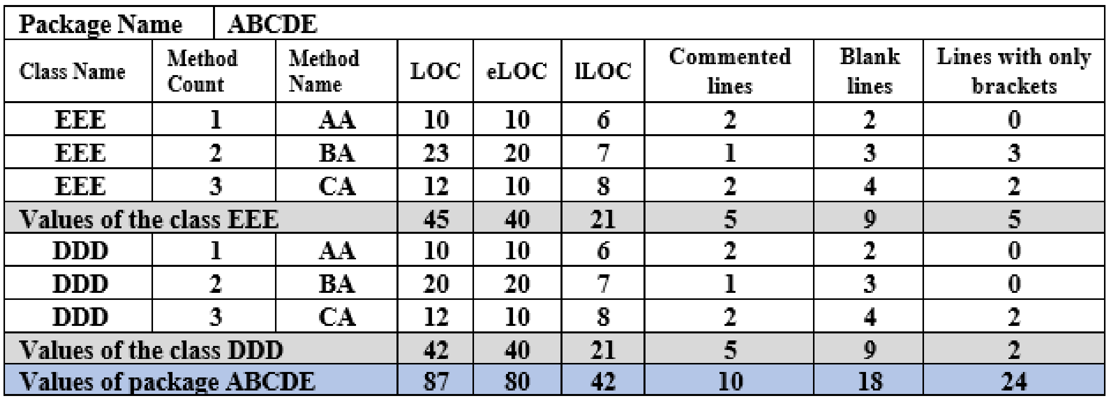
Package Name		ABCDE						
Class Name	Method Count	Method Name	If	For	While	Do-while	Switch - case	Total
EEE	1	AA	1	2	0	0	0	3
EEE	2	BA	0	1	1	0	2	4
EEE	3	CA	0	2	0	1	1	4
Total decisional statements of class EEE								11
DDD	1	AA	1	2	0	0	0	3
DDD	2	BA	0	0	1	0	1	2
DDD	3	CA	0	1	0	1	1	3
DDD	3	DA	1	1	1	1	1	5
Total decisional statements of class DDD								13
Total decisional statements of package ABCDE								24

Table 3.2.4 5: Type of a statements (executable and non-executable) for a class



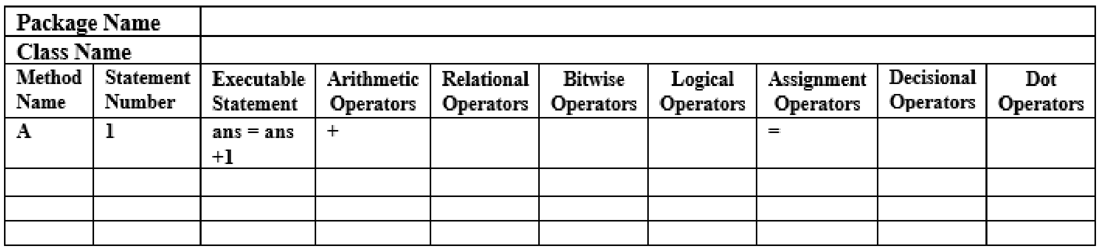
Package Name	RPSTUV									
Class Name	Method Name	Statement Number	Statement	Executable Statement	Non-Executable Statement	eLOC	lLOC	Commented lines	Blank lines	Lines with only brackets
ABC	EEE	1			X			X		
ABC	EEE	2	for (int i=0;i<5;i++){	X		X				
ABC	EEE	3	system.out.println(“test”);	X		X	X			
Values for the method EEE				2	1	2	1	1		
ABC	DDD	1								
ABC	DDD	2								
ABC	DDD	3								
Values of the method DDD				3	0	3	2	0	3	1
Values of class ABC				5	1	5	3	1	3	1
ACDCB	EEE	1			X			X		
ACDCB	EEE	2	for (int i=0;i<5;i++){	X		X				
ACDCB	EEE	3	system.out.println(“test”);	X		X	X			
Values for the method EEE				2	1	2	1	1		
ACDCB	DDD	1								
ACDCB	DDD	2								
ACDCB	DDD	3								
Values of the method DDD				3	0	3	2	0	3	1
Values of class ACDCB				5	1	5	3	1	3	1
Values of the system				10	2	10	6	2	6	2

Table 3.2.4 6: Length based measures for the system



Package Name	ABCDE							
Class Name	Method Count	Method Name	LOC	eLOC	lLOC	Commented lines	Blank lines	Lines with only brackets
EEE	1	AA	10	10	6	2	2	0
EEE	2	BA	23	20	7	1	3	3
EEE	3	CA	12	10	8	2	4	2
Values of the class EEE			45	40	21	5	9	5
DDD	1	AA	10	10	6	2	2	0
DDD	2	BA	20	20	7	1	3	0
DDD	3	CA	12	10	8	2	4	2
Values of the class DDD			42	40	21	5	9	2
Values of package ABCDE			87	80	42	10	18	24

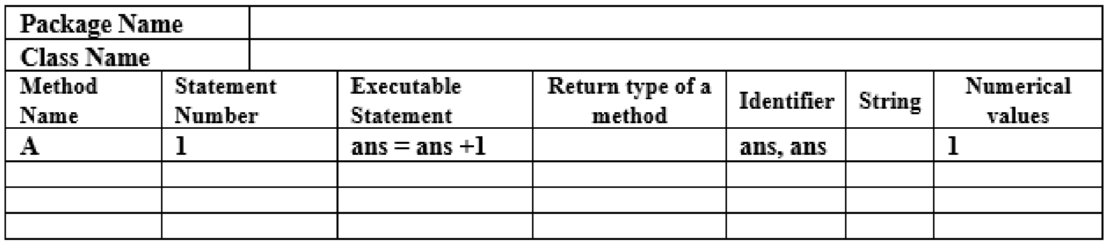
Table 3.2.4 7: List of operators for a given programming statement, method, class or system



Package Name									
Class Name									
Method Name	Statement Number	Executable Statement	Arithmetic Operators	Relational Operators	Bitwise Operators	Logical Operators	Assignment Operators	Decisional Operators	Dot Operators
A	1	ans = ans +1	+				=		

- Example for decisional operators :
    - if ( ), for( ), while( ), do while( ), switch ( ), case :, ?: (ternary operator)
- The semicolons in a 'for' loop is not considered as operators.
- The square brackets of an array is not considered as operators.

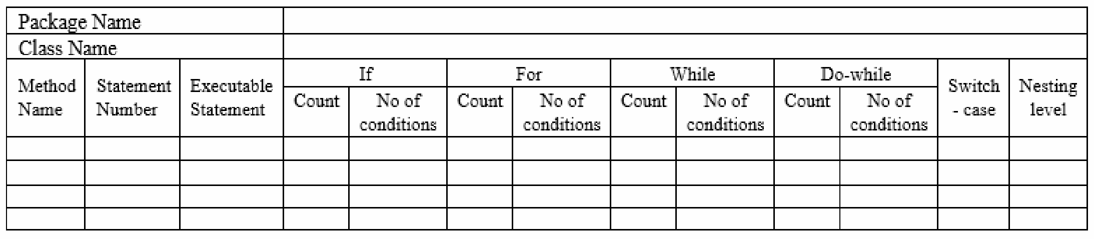
Table 3.2.4 8: List of operands for a given programming statement, method, class or system



Package Name						
Class Name						
Method Name	Statement Number	Executable Statement	Return type of a method	Identifier	String	Numerical values
A	1	ans = ans +1		ans, ans		1

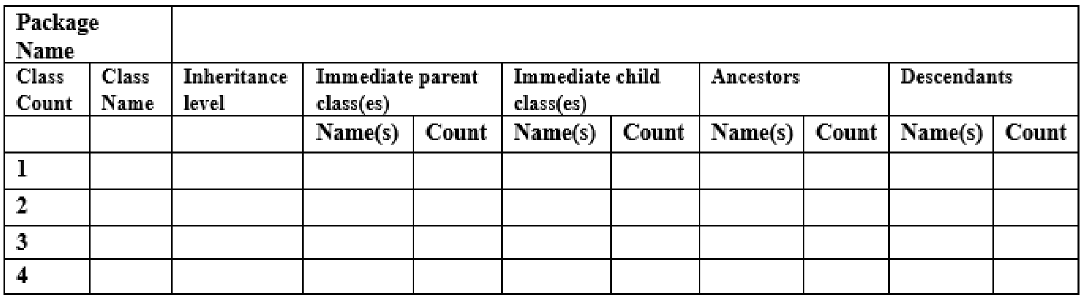
- Examples for identifiers :
    - Class names, method names (along with the brackets, parameter names and their data types), object names, variable names etc.
- Any programming statement that appears in between a double quote is considered a one String.
    - Example : "enter value"

Table 3.2.4 9: The type and nesting level of control structures for a given method, class or system



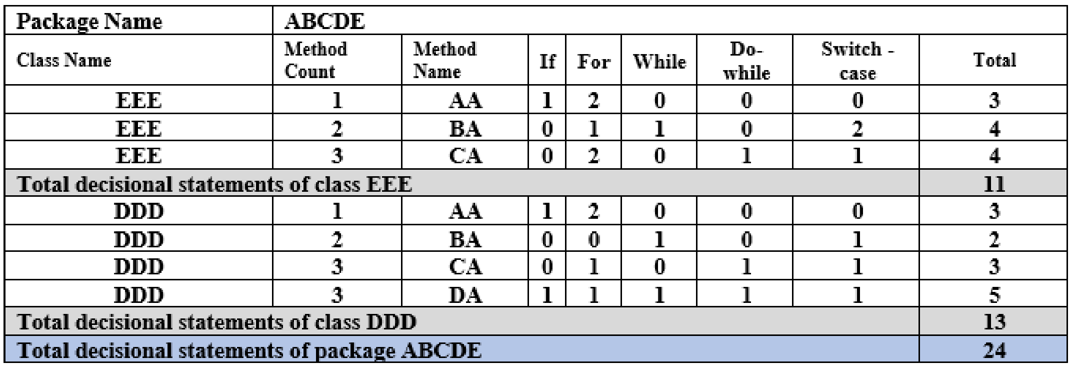
Package Name												
Class Name												
Method Name	Statement Number	Executable Statement	If		For		While		Do-while		Switch - case	Nesting level
			Count	No of conditions	Count	No of conditions	Count	No of conditions	Count	No of conditions		

Table 3.2.4 10: The inheritance related information of the classes belonging to the system



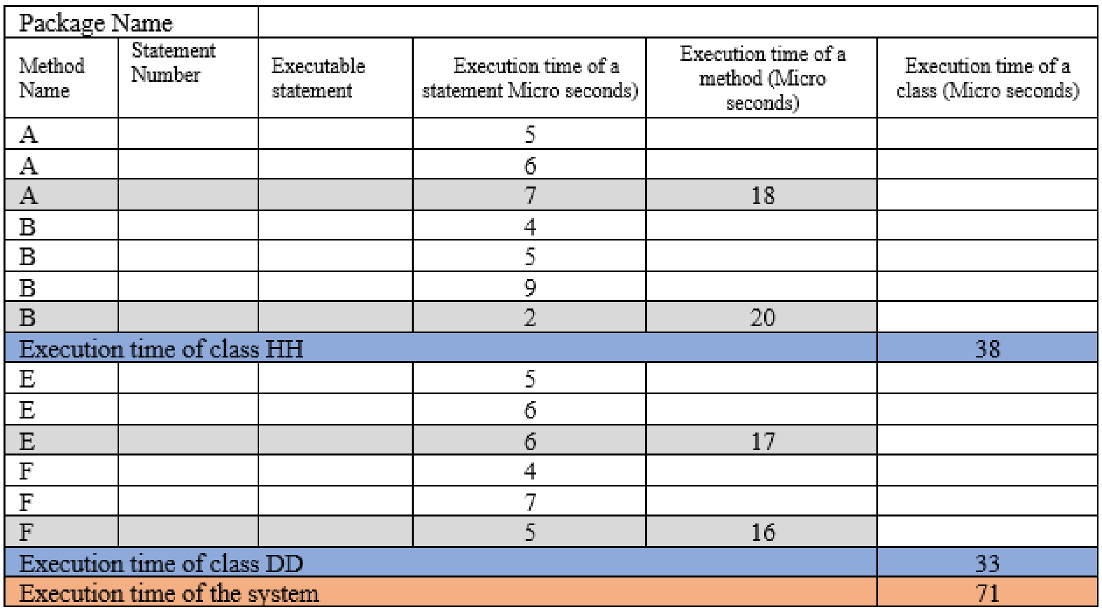
Package Name										
Class Count	**Class Name**	**Inheritance level**	**Immediate parent class(es)**		**Immediate child class(es)**		**Ancestors**		**Descendants**	
			Name(s)	**Count**	**Name(s)**	**Count**	**Name(s)**	**Count**	**Name(s)**	**Count**
1										
2										
3										
4										

Table 3.2.4 11: Number of decisional statements of a system



Package Name	**ABCDE**							
Class Name	**Method Count**	**Method Name**	**If**	**For**	**While**	**Do-while**	**Switch - case**	**Total**
EEE	**1**	**AA**	**1**	**2**	**0**	**0**	**0**	**3**
EEE	**2**	**BA**	**0**	**1**	**1**	**0**	**2**	**4**
EEE	**3**	**CA**	**0**	**2**	**0**	**1**	**1**	**4**
Total decisional statements of class EEE								**11**
DDD	**1**	**AA**	**1**	**2**	**0**	**0**	**0**	**3**
DDD	**2**	**BA**	**0**	**0**	**1**	**0**	**1**	**2**
DDD	**3**	**CA**	**0**	**1**	**0**	**1**	**1**	**3**
DDD	**3**	**DA**	**1**	**1**	**1**	**1**	**1**	**5**
Total decisional statements of class DDD								**13**
Total decisional statements of package ABCDE								**24**

Table 3.2.4 12: Total execution time of a system



Package Name					
Method Name	Statement Number	Executable statement	Execution time of a statement Micro seconds)	Execution time of a method (Micro seconds)	Execution time of a class (Micro seconds)
A			5		
A			6		
A			7	18	
B			4		
B			5		
B			9		
B			2	20	
Execution time of class HH					38
E			5		
E			6		
E			6	17	
F			4		
F			7		
F			5	16	
Execution time of class DD					33
Execution time of the system					71

Table 3.2.4 13: Memory consumption of a system

Package Name						
Class Name	Method Name	Statement Number	Executable statement	Static memory allocation	Dynamic memory allocation	Memory consumption of a statement (Kilo bytes)
HH	A			X		5
HH	A				X	6
HH	A			X		7
Values for method A				2	1	18
HH	B			X		4
HH	B			X		5
HH	B			X		9
HH	B				X	2
Values for method B				3	1	20
Values for class HH				5	2	38
DD	E			X		5
DD	E				X	6
DD	E			X		6
Values for method E				2	2	17
DD	F			X		4
DD	F			X		7
DD	F			X		5
Values for method F				3	0	16
Values for class DD				5	2	33
Values for the system				10	4	71

Table 3.2.4 14: Communications between methods, classes and a system

Package Name												
			Fan-out						Fan-in			
			Method				Variable		Method		Variable	
Class Name	Method Name	Executable Statement	Call to a regular method in the same class (internal messaging)	Call to a regular method in a different class (external messaging)	Call to a recursive method in the same class (internal messaging)	Call to a recursive method in a different class (external messaging)	Referencing a data member of the same class	Referencing a data member in a different class	Calling from another method in the same class	Calling from another method in a different class	A method in a different class referencing a data member in the current class	Abstract data types	Coupling Type
CDE	AB		X										
CDE	AB			X									
CDE	AB									X			
Counts for method AB			1	1	0	0	0	0	0	1	0		
CDE	CD			X									
CDE	CD								X				
Counts for method CD			0	1	0	0	0	0	1	0	0		
Counts for class CDE			1	2	0	0	0	0	1	1	0		
PQR	EF						X						
PQR	EF							X					
Counts for method EF			0	0	0	0	1	1	0	0	0		
Counts for class PQR			0	0	0	0	1	1	0	0	0		
Counts for the system			1	2	0	0	1	1	1	1	0		

In our approach, we'll be ranking the developers based on source code quality as well as task allocated for each developer in projects by considering the completion of each tasks, deadlines and workbench. Accordingly, the performance of each developer will be evaluated for each and every project and their continuous progress will be tracked. In here, a burn down chart will be generated based on the task allocation for team of developers in a project as follows.

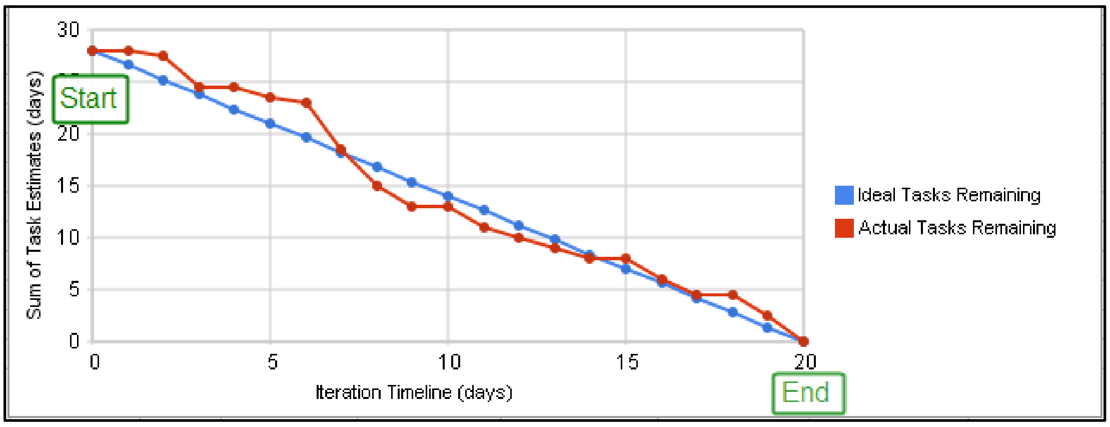


Figure 3.2.4 :Burn down Chart

3.3 Tools and Technologies

Tools:

- Eclipse IDE
- ANLTR
- Notepad++
- Xampp
- POSTMAN
- ANLTR (Eclipse Plugin)

Technologies:

- MySQL
- Java
- HTML
- PHP5
- CSS
- AngularJS
- Node.js

3.4 Gantt Chart

Table 3.4: Tabular representation of Gantt Chart

Task Name	Start	Finish	Duration (days)
Forming Group	12/22/17	12/29/17	7
Identify research topic	12/30/17	01/09/18	10
General research about topic	01/05/18	01/22/18	17
Project topic assessment	01/23/18	02/14/18	22
Project charter preparation and submission	02/15/18	02/23/18	8
Deciding research methodology	02/24/18	03/08/18	12
Project proposal report writing and submission	03/09/18	04/01/18	23
Project proposal presentation	03/19/18	04/04/18	16
(SRS)/Design document preparation and submission	04/09/18	05/07/18	28
Project Status document - SRS preparation	04/10/18	05/08/18	28
Progress presentation - I	05/09/18	06/25/18	47
Research paper	07/11/18	08/06/18	26
Prototype development	04/29/18	07/16/18	78
Progress presentation - II	07/03/18	08/07/18	35
Final report (Draft)	08/20/18	09/01/18	12
Final report (Draft) feedback submission	09/05/18	09/20/18	15
Website Assessment	09/18/18	10/04/18	16
Final Report (Soft Bound)	09/12/18	10/06/18	24
Final Submissions – CD with deliverables	10/03/18	10/28/18	25
Final Presentation	10/23/18	11/12/18	20
Final Report (Group) Hard Bound	10/30/18	11/21/18	22

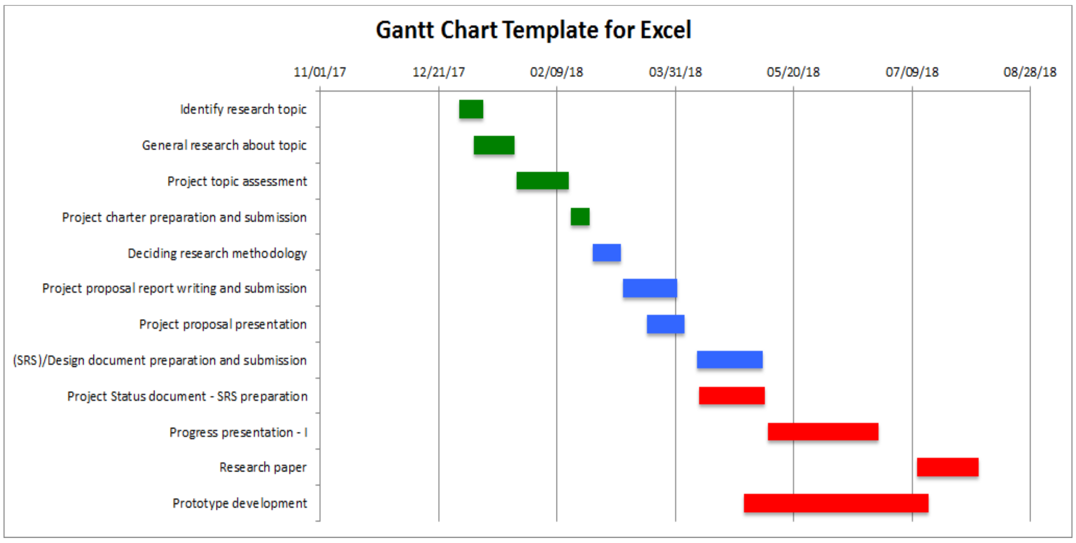
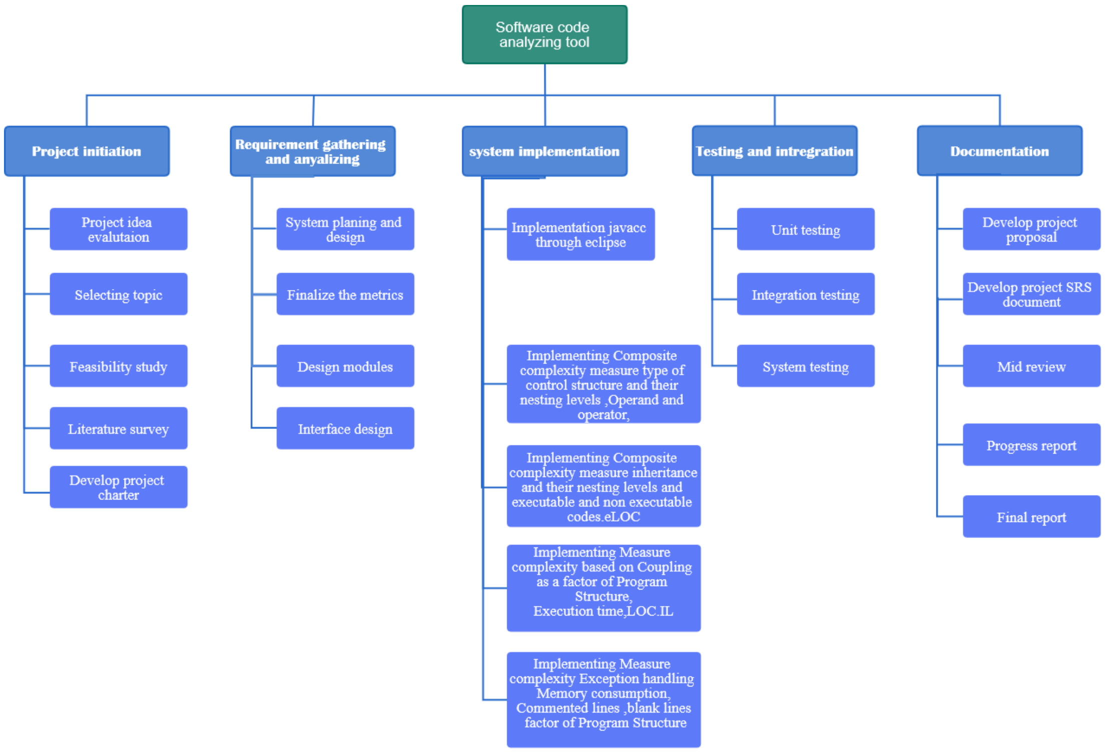


Figure 3.4.1: Gantt chart

3.5 Work breakdown Structure



Software code analyzing tool
Project initiation
Project idea evalutaion
Selecting topic
Feasibility study
Literature survey
Develop project charter
Requirement gathering and anyalizing
System planing and design
Finalize the metrics
Design modules
Interface design
system implementation
Implementation javacc through eclipse
Implementing Composite complexity measure type of control structure and their nesting levels ,Operand and operator,
Implementing Composite complexity measure inheritance and their nesting levels and executable and non executable codes.eLOC
Implementing Measure complexity based on Coupling as a factor of Program Structure, Execution time,LOC.IL
Implementing Measure complexity Exception handling Memory consumption, Commented lines ,blank lines factor of Program Structure
Testing and intregration
Unit testing
Integration testing
System testing
Documentation
Develop project proposal
Develop project SRS document
Mid review
Progress report
Final report

Figure 3.13.1: Work breakdown Structure

4 DESCRIPTION OF PERSONAL AND FACILITIES

Member	Component	Task
IT15413802 M.A.N.S.U.K. Uvindasiri	Composite complexity measure type of control structure and their nesting levels ,Operand and operator,	1. Literature Review on impact of control structures on complexity operand and operator. 2. Implementing ANLTR as a parser with Eclipse. 3. Extraction of raw metric data from the program code through the use of ANLTR 4. Output program structure with complexity based on control structure complexity. 5. Output operand and operator complexity. 6. Graphical representation of the interpreted results through graph and table. 7. Write website content 8. Documenting. 9. Testing individual component
IT15113900 P.K.H Palihakkara	Composite complexity measure inheritance and	1. Literature Review on impact of inheritance on complexity.

	their nesting levels and executable and non executable codes, eLOC	2. Implementing ANLTR as a parser with Eclipse. 3. Extraction of raw metric data from the program code through the use of ANTLR 4. Output program structure with complexity based on inheritance. 5. Output program executable and non-executable code complexity 6. Graphical representation of the interpreted results through graph and table. 7. Implementing input interface 8. Documenting 9. Testing individual component
IT15112538 K.G.D.R Perera	Measure complexity based on Coupling as a factor of Program Structure,Execution time,LOC.ILOC	1. Literature Review on impact of coupling and execution time on complexity. 2. Implementing ANLTR as a parser with Eclipse. 3. Extraction of raw metric data from the program code through

		the use of ANLTR 4. Output program structure with complexity based on coupling. 5. Output the execution time on the code complexity. 6. Graphical representation of the interpreted results through graph and table. 7. Documenting 8. Testing individual
IT15111548 Gamaarachchi G.A.C.Y	Measure complexity Exception handling , Memory consumption, Commented lines ,blank lines factor of Program Structure	1. Literature Review on impact of exception handling and memory consumption on complexity. 2. Implementing ANTLR as a parser with Eclipse. 3. Extraction of raw metric data from the program code through the use of ANTLR. 4. Output program structure with complexity based on exception handling. 5. Output program structure with complexity based on memory consumption. . 6. Graphical

		representation of the interpreted results through graph and table. 7. Implementing input interface. 8. Documenting. 9. Testing individual component.

5 REFERENCES

[1]Arnold, K., Gosling, J., Holmes, D., & Holmes, D. The Java programming language (Volume 2). Reading: Addison-wesley. 2000

[2] Sanchez, S. M., & Lucas, T. W. Exploring the world of agent-based simulations: simple models, complex analyses: exploring the world of agent-based simulations: simple models, complex analyses. In Proceedings of the 34th conference on Winter simulation: exploring new frontiers. 2002. Pages 116-126

[3] Shrivastava, S. V., & Shrivastava, V. Impact of metrics based refactoring on the software quality: A case study. In TENCON 2008- 2008 IEEE Region 10 Conference. 2008. Pages 1-6.

[4] Davis, J.S., & LeBlanc, R.J. A Study of the Applicability of Complexity Measures. IEEE Transactions on Software Engineering, Volume 14. Number 9. 1988

[5] Sheppard, S. B., Curtis, B., Milliman, P., Borst, M. A., & Love, T. First year results from a research program on human factors in software engineering. In afips (p. 1021). IEEE. 1899

[6] Prabhu, Jeevan. Complexity Analysis of Simulink Models to Improve the Quality of Outsourcing in an Automotive Company. Manipal University. 2010

[7] Olszewska, Marga. Simulink-Specific Design Quality Metrics. TUCS Technical Report 1002. Turku Centre for Computer Science. 2011

[8] Tourlakis G. J. Computability, Reston, Virginia. Volume 12. 1984. Pages 39-42.

[9] De Oliveira, C.D., De Oliveira, C.D., Fong, E.N. and Black, P.E., 2017. *Impact of Code Complexity on Software Analysis*. US Department of Commerce, National Institute of Standards and Technology

[10] Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A. and Love, T., 1979. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on software engineering*, (2), pp.96-104

[11] Kan, S.H., 2002. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc..

[12]Magel, K., Kluczny, R.M., Harrison, W.A. and Dekock, A.R., 1982. Applying software complexity metrics to program maintenance

[13]Bandi, R.K., Vaishnavi, V.K. and Turk, D.E., 2003. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Transactions on Software Engineering*, *29*(1), pp.77-87

[14]Girish H. Subramanian, Parag C. Pendharkar and Mary Wallace, “An empirical study of the effect of complexity, platform, and program type on software development effort of business applications”, Empir Software Eng (2006), 17 October, 2006

[15]Sanjay Misra, “A complexity Weight based on cognitive weights”, International journal, Volume 01 Number 01 (2006), pp. 1-10

[16]Boehm, B.W., Brown, J.R. and Lipow, M., 1976, October. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering* (pp. 592-605). IEEE Computer Society Press

[17]S. Kan, Metrics and Models in Software Quality Engineering – Second Edition, Addison-Wesley, Boston, 2003, ISBN 0-201- 72915-6

[18]M.Lanza, R.Marinescu, Object-Oriented Metrics in Practice – Using Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, Springer – Verlag, Berlin, Heidelberg, New York, Germany, 2006, ISBN 978-3-540-24429-5

[19]R. Lincke, J. Lundberg, W. Löwe, Comparing software metrics tools, Proc. of the 2008 international symposium on Software testing and analysis ISSTA '08, pp. 131-142

[20]J. Novak, G. Rakić, Comparison of Software Metrics Tools for :NET, Proc. 13th International Multiconference Information Society - IS 2010, vol. A, pp. 231-234

[21]Rakić G., Budimac Z., Bothe K., Towards a ‘Universal’ Software Metrics Tool-Motivation, Process and a Prototype, Proceedings of the 5th International Conference on Software and Data Technologies(ICSOFT), 2010, pp.263-266

[22] Boehm, B.W., Brown, J.R. and Lipow, M., 1976, October. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering* (pp. 592-605). IEEE Computer Society Press

[23]Usha Chhillar, Shuchita Bhasin, “A New Weighted Composite Complexity Measure for Object-Oriented Systems”, International Journal of Information and Communication Technology Research, Volume 1 No. 3, Department of Computer Science, Kurukshetra University, Kurukshetra, Haryana, India, July 2011

[24]Jones, C.C., 1997. Software quality: Analysis and guidelines for success. Thomson Learning

[25]Blog.sonarsource.com. (2018). *SonarSource Blog*. [online] Available at: https://blog.sonarsource.com/executable_lines [Accessed 29 Mar. 2018]

[26]Grobmeier, C. (2018). *Checking out JArchitect*. [online] Grobmeier.solutions. Available at: https://grobmeier.solutions/checking-out-jarchitect-05082013.html [Accessed 29 Mar. 2018]

[27]profile, V. (2018). *The FindBugs Blog*. [online] Findbugs.blogspot.com. Available at: http://findbugs.blogspot.com/ [Accessed 2 Apr. 2018]

[28]Gurina, I., Okolotovich, A., Seriaga, O. and Malahosky, N. (2018). *Targetprocess | Product Blog*. [online] Targetprocess. Available at: https://www.targetprocess.com/blog/ [Accessed 2 Apr. 2018]

[29]Voutilainen, A., 1995, March. A syntax-based part-of-speech analyser. In *Proceedings of the seventh conference on European chapter of the Association for Computational Linguistics* (pp. 157-164). Morgan Kaufmann Publishers Inc..

[30]Neamtiu, I., Foster, J.S. and Hicks, M., 2005. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, *30*(4), pp.1-5

[31]Zuse, H., 1991. Software complexity. *NY, USA: Walter de Cruyter*

[32]Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A. and Love, T., 1979. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on software engineering*, (2), pp.96-104

[33]Vanderfeesten, I., Cardoso, J., Mendling, J., Reijers, H.A. and van der Aalst, W., 2007. Quality metrics for business process models. *BPM and Workflow handbook*, *144*, pp.179-190