METRICS EVALUATION USING OPEN SOURCE SOFTWARE

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Technology
in
Computer Science and Applications

Submitted By
Shiva Dogra
(Roll No. 601103022)

Under the supervision of
Vineeta Bassi
Assistant Professor
(SMCA)

SCHOOL OF MATHEMATICS AND COMPUTER APPLICATIONS
THAPAR UNIVERSITY
PATIALA – 147004

June 2013

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Metrics Evaluation Using Open Source Software"**, in partial fulfillment of the requirements for the award of degree of Master of Technology in Computer Science and Applications submitted in School of Mathematics and Computer Applications, Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Vineeta Bassi** and refers other researcher's work which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for award of any other degree of this or any other University.

Shiva

(Shiva Dogra)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Vineeta Bassi

(Vineeta Bassi)
Assistant Professor
SMCA

Countersigned by:

(Dr. Rajesh Kumar)
Head
School of Mathematics and Computer Applications
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

First of all I would like to thank the Almighty, who has always guided me to work on the right path of the life.

This work would not have been possible without the encouragement and able guidance of my supervisor **Vineeta Bassi**. I thank my supervisor for their time, patience, discussions and valuable comments. Their enthusiasm and optimism made this experience both rewarding and enjoyable.

I am equally grateful to **Dr. Rajesh Kumar**, Associate Professor and Head, School of Mathematics and Computer Applications, for motivation and inspiration that triggered me for the thesis work.

I will be failing in my duty if I don't express my gratitude to **Dr. S. K. Mohapatra,** Senior Professor and Dean of Academic Affairs the University, for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field.

I am also thankful to the entire faculty and staff members of School of Mathematics and Computer Applications Department for their direct-indirect help, cooperation, love and affection, which made my stay at Thapar University memorable.

Last but not least, I would like to thank my parents for their wonderful love and encouragement, without their blessings none of this would have been possible. I would also like to thank my brother, since he insisted that I should do so. I would also like to thank my close friends for their constant support.

Abstract

Software metrics play an crucial role in the administration of the software projects. Metrics used to track development process, quantify restructuring impact and to calculate code quality. Open source software is a software product whose source code is freely available to all its intended users. Open source software during the last decade has got very remarkable success. This thesis thus been escorted concentrating on gradual directional change especially leading to more advanced form of open source software and software metrics.

In this research, we analyze various software metrics at two levels that is at class level and at method level. Metrics analyze at class level are number of methods, Lack of Cohesion of Method, Average Cyclomatic Complexity, Number of Java Statements, Halstead Bug, Halstead Effort, Unweighted class size, Total Instance Variables declared, Total Packages Imported, Response for Class, Coupling between Objects, Maintainability Index, Total Number of Comment Lines in the class and Total Line of code. Metrics analyze at method level are Complexity, Number of Comment Lines, Number of Java Statements, Halstead Length, Halstead Vocabulary, Halstead Effort, Halstead Predicted Bug, Number of Classes Referenced, Number of Methods External to class called by method, Number of Methods local to class called by method and Total Line of Code in the method.

Four java-based open source softwares are analyzed by assuming that the number of download indicates the success of these softwares. Tool used to find the value of metrics of these open source software is JHawk which is also a java-based open source framework which can be enclosed in any java application.

Table of Contents

List of Figures

List of Tables

Abbreviations

AVCC	Average Cyclomatic Complexity
CBO	Coupling Between Objects
CCML	Comment Lines in the Class
COMP	Cyclomatic Complexity
CREF	Number of Classes Referenced
CSV	Comma Separated Values
HBUG	Halstead's Bug
HEFF	Halstead's Effort
HLTH	Halstead's Length
HTML	Hypertext Markup Language
HVOC	Halstead's Vocabulary
INST	Instance Variables Declared
LCOM	Lack of Cohesion in Methods
LMET	Methods Local to Class Called by Method
MI	Maintainability Index
NLOC	Number of Line of Code
NOCL	Number of Comment Lines
NOS	Number of Java Statements
OSS	Open Source Software
PACK	Packages Imported
RFC	Response for Class
UWCS	Unweighted Class Size
XMET	Methods external to Class Called by Method
XML	eXentensible Markup Language

Chapter 1

INTRODUCTION

Open source software during the last decade has got very remarkable success but people are still reserved to choose open source products. Software metrics is used to calculate the quality of code. This thesis has escorted concentrating on the gradual directional change especially leading to more advanced form of study of open source software and software metrics. The goal of thesis is to study the relationship between different metrics of open source software. The work includes examining the software metrics value to increase its maintainability and reliability using the concept of modularity and defect prediction.

1.1 Open Source Software

The term “open source” is bring into practice during the foundation of open source initiative in 1998. OSS is computer software that has its source code made available under open source definition based license [1]. OSS is software whose license allow user to run the program for any purpose, modify according to their needs, to improve the program and release the modified version of the program and to redistribute its copies at zero cost [2]. OSS licenses grant licensees the right to copy, modify and redistribute code. Example of free software/open source software includes Apache License, BSD License, GNU General Public License, GNU Lesser Public License, MIT Public License and Mozilla License [3].

Advantages of Open Source Software

- Development speed of the open source software is high.
- User involvement makes it more useful.
- OSS sometimes may have advantages in the area of total cost of ownership [4].

Disadvantages of Open Source Software

- OSS does not have even single source that can be tapped for support.
- OSS does not guarantee the update regularly since nobody is bounded to do so.
- Most of the OSS is incompatible with the present day devices so needs higher installation cost.
- Technical support is costlier as compared to commercial software [4].

1.2 Software Metrics

Software metrics is a measure of some property of a piece of software or its specifications. Since quantitative measurements are essential in all sciences, there is a continuous effort by computer science people to bring similar approaches to the software development. Software metrics measures blee of the software product or the process. Categories of software metrics include:

- Management: Cost, schedule, progress and computer resource utilization.
- Requirements: Completeness, traceability, consistency and stability.
- Design: Size, complexity, modularity, coupling and cohesiveness.
- Code: Fault density, problem report analysis, and standard compliance.
- Test: Coverage, sufficiency, failure rate and mean time to failure [5].

1.2.1 Use of Metrics

Software metrics are used to obtain objective reproducible measurements that can be used for quality assurance, performance, debugging, management and estimating costs. Finding defects in the code, predicting defective code, predicting project success, and predicting project risk. There is still some debate around which metrics matter and what they mean, the utility of the metrics is limited to quantifying one of the following goals: schedule of a software project, size/complexity of development involved, cost of project, and quality of software [6].

Advantages of Software Metrics

- Metrics help to identify, prioritize, track and communicate project issues at all levels.
- Metrics can accurately describe the status of software project processes and product.
- Metrics can be used as a resource to anticipate problems and to avoid being forced into a reactive fix.
- Metrics provide an effective rationale for selecting the best alternatives [6].

Disadvantages of Software Metrics

- There can be a strong tendency for professionals to display over-optimism and over-confidence.
- There are arguments about the effect that software metrics have on the developer's productivity and well being [6].

1.3 JHawk

JHawk is java based open source framework which can be incorporated in any java application for performance testing. Latest version of JHawk is JHawk 5.1. JHawk carry out the modules of the input and generate the results in a graphical form. JHawk focuses on the intrinsic quality of the software. The process of taking input in the form of code and analyzing it is called static analysis. JHawk helps to take sensible decisions based on the result of analysis of code. JHawk allow analysis of software metrics value of any code written in java at class level and at method level.

JHawk is a static code analysis tool that is it takes code as a input and calculates metrics based on number of views of the code like volume, complexity, relationship between classes and packages. A number of metrics produced by the JHawk relate directly to the potential design defects.

The professional version of JHawk allows to create own metrics. This can allow the creation of more intense metrics according to the type of application being analyze. These metrics do not need to be related to the code artifact, it can use data from defect databases, code repository and anything that can be accessed via java code.

This allows creating an integrated view of those factors that may affect the quality of software.

The core of JHawk is the standalone code analyzer which analyzes java code and prepares result to be viewed with application in which JHawk is incorporated or exported to HTML, CSV and XML formats. Using JHawk interchange XML format information can be stored about java file set and can be reviewed any time [7].

JHawk licensing is simple. There are two categories of license:

- **Commercial License**

  It includes one year free updates of the software after that renewable fee is charged for each year of update. It comes in four flavor and they are personal, professional, site and corporate.

- **Academic License**

  Academic license for the purpose of research can only be purchased by academic institutions and have number of conditions attached to their purchase. It includes three years of free update. The research license is a full commercial license at reduced rate for recognized academic institutions [8].

1.4 Open Source Software Used as Input

To study the relationship between different metrics four open source software is given as an input to JHawk tool. These softwares are open source code coverage tools.

Quilt

Quilt is a Java software development tool that measures coverage**,** the extent to which unit testing exercises the software under test. It is optimized with JUnit unit test package, the Ant java build facility and the Maven project management tool kit. The simplest measure of coverage is statement coverage [10].

EMMA

EMMA is also an open source toolkit for measuring and reporting java code coverage. EMMA is different from other tools because it uses a unique feature combination that is support for large scale enterprise software development while keeping individual developer's work fast and iterative [11].

Nat

Nat is software which allows seeing how good JUnit tests are and generates a report from code to graphically show how many of project's method are being tested and how well [12].

JVMDI Code Coverage Analyzer

This small utility is shared library which when loaded into a Java VM (1.4+) which supports JVMDI will record all the lines of code executed [13].

1.5 Thesis Objectives

The aim of this thesis is to present a method that will help in measuring the value of software metric of open source software using one java based open source software i.e. JHawk. The values are calculated and then studied to find the relationship between them. The focus of this thesis is to find the values of software metrics and check whether they lie in valid range or not. If value does not lie in valid range it shows parameter which needs to focus in code.

The specific goals of this thesis are:

- To find the values of metrics using tool.
- To statistically and graphically study the values and relationship between metrics using SPSS.
- To correlate the value of metrics with each other metrics using SPSS.

1.6 Thesis Outline

This thesis is divided into six chapters, including its introduction.

Chapter 1 is introduction.

Chapter 2 examines various research papers that are related to Open Source Software and software metrics. It discusses previous works that form the roots of subsequent research in both areas.

Chapter 3 indicates the problem definition and justification to the problem statement.

Chapter 4 presents how work is done in a systematic way.

Chapter 5 presents the results from the research taken.

Chapter 6 contains conclusion and future work.

Chapter 2

LITERATURE SURVEY

Open source software projects nowadays are gaining momentum worldwide and are being focused not only from large corporations, but also from researchers. Some large corporations such as IBM, Sun Microsystems etc are supporting large OSS projects such as Java, Eclipse and such more projects. Many research work are also being conducted using OSS projects with objectives like finding their key success factors, problem areas, license study, development community and many more [14].

Many studies have been conducted to identify the key success factors of OSS Projects and they can be categorized into three approaches. First approach of the study is by studying several successful OSS Projects. The second approach is trying to find the similarity in the processes in many successful OSS Projects. The third approach focuses on process aspects such as team communication across. All of these approaches have the same weakness since they involve only relatively small number of OSS Projects, leading to the consequence that they may not give good representation of other OSS Project that already numbered in hundreds of thousands. Some other researchers are able to identify some alarming potential problems in the OSS Projects and their communities. In certain phase of the projects, the software system will increase in complexity that makes it more difficult to be managed by the communities [14].

The literature review is intended as an introductory guide to help understanding in the areas of open source software and software metrics. Also theoretical background relating to OSS projects and software metrics is reviewed.

2.1 Open Source Software

The term "Open Source Software" has been gaining much popularity in both the fields of research and software industry. There are two widely-used terms for the concept and they are “Free Software” and “Open Source Software”.

Free software allows all users to run, copy, distribute, study, change, and improve software. Also users of free software have freedom to do following things:

- Run the program, for any purpose.
- Study how the program works and change it to make it do what user wish. Access to source code is precondition.
- Redistribute copies so to help others.
- Distribute copies of modified versions to others. By doing this help whole community to get benefit from changes. Access to source code is a precondition for this [15].

Main point to be note is that not all OSS is cost free and not all cost free software is open source [16].

Open source software technically can be defined as software whose code is available to the users and functionally it can be defined as software which has following features:

- Usually free or cheap to acquire and use.
- Primarily developed by volunteers.
- Anyone can modify and customize.
- Users have direct input into development [16].

Antonym of OSS is proprietary software and closed software. In many cases, OSS approaches have the potential to increase the functionality, quality, flexibility, while lowering cost and development time.

There are some myths related to OSS like:

- **Myth : OSS unsupported**
  Businesses support OSS like Red Hat, Novell, HP, Sun, IBM, DMSolutions, SorceLabs, OpenLogic, Carahsoft etc.
- **Myth : Only programmers care about software licenses**
  Bob young said "Would you buy a car with hood welded shut? We demand the ability to open the hood because it gives us, the consumer, the control over what we have bought, overcharges us, would not fix the problem, or refuses to install would be happy to have our business".
- **Myth : Developers are inexperienced**
  According to BCG study average OSS developers are 30 years old with 11 years experience.

- **Myth : OSS is no cost**

  Training, support, transition etc are not free of cost and competition often produces lower total cost of ownership and higher return of investment for OSS [15].

Some common examples of OSS who are familiar, recently emerging and widely used listed below.

TYPE	NAME
Operating systems	LINUX
Web and email servers	Apache and Sendmail/Postfix/Qmail
Web and db languages	PHP/Perl and MySQL/PostgreSQL
Web content management	Drupal, Plone, Bricolage
Virus and spam protection	ClamAV, SpamAssassin
Desktop Applications	Open Office, Mozilla, gAIM
Security	GPG

Table 2.1 Examples of OSS [15]

There are a lot of definitions for the concept of open source software development, according to OSD software which has ability to distribute freely with available source code through the Internet and using unpaid people that can modify the code freely, is open source software [17]. In other words, OSSD is a kind of distributed software development that has a large amount of contributors and because of using internet and make sharing freely it is so successful and useful that developers can communicate over the distance. Due to having variety of contributors in OSS projects and story of knowledge sharing among them the project can be more powerful and this might lead to improve even the position of contributors. For example they can move to the developers group from users or even in developers group move to the trusted developers group [18].

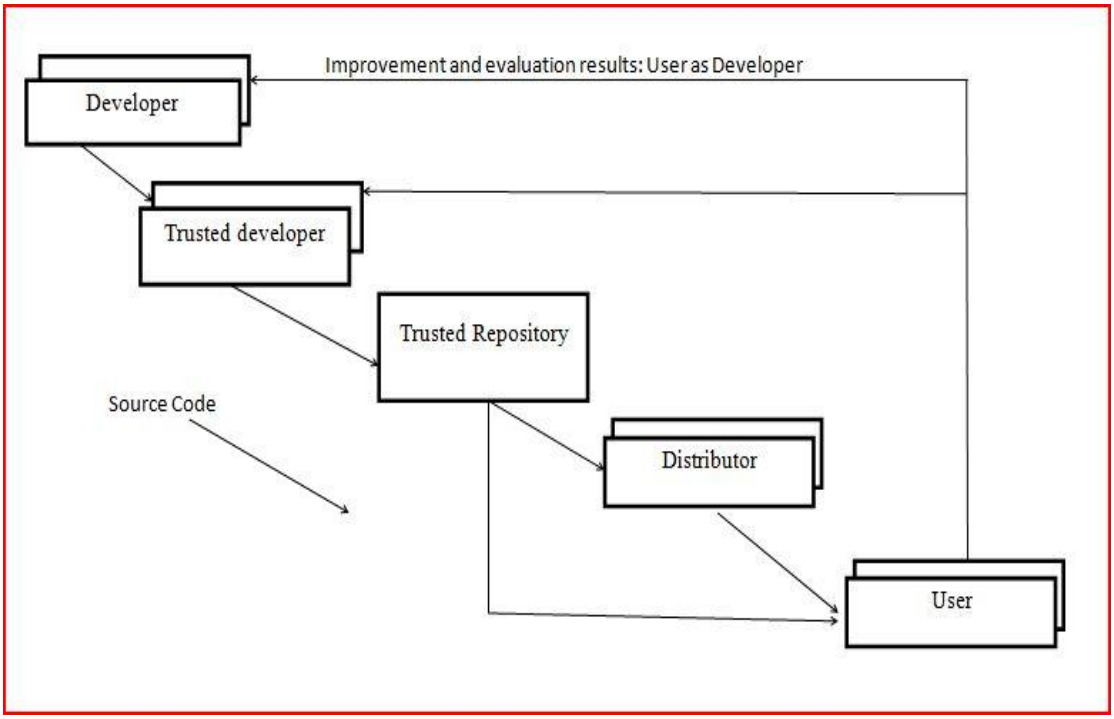


Figure 2.1 OSS Development Model [15]

OSS users typically use software without paying licensing fees. They only pay for training and support. Users are responsible for paying/developing new improvements and any evaluations that they need and often cooperate with others to do so. Goal of this model is active development community (like consortium) [15].

2.2 Software Metrics

It has been estimated that more than half of work force will rely on computers and software to do its daily work. As computer hardware costs continue to decline, the demand for new applications software continues to increase at a rapid rate. The existing inventory of software continues to grow, and the effort required maintaining it continues to increase as well. If software development scene is characterized by poor quality software, productivity rate that is increasing more slowly than demand for software and cost estimate that are grossly inaccurate. This situation has often been referred to software crisis [19].

The 2009 Standish Group CHAOS Report states that 24% of all software projects fail. This means they are cancelled prior to completion or delivered and never used. One of the contributing factors is that modern software is almost never completely developed

from scratch, but is rather extended and modified using existing code and often includes third party source code. This can lead to poor overall maintainability, difficult extensibility and high complexity. To better understand the impact of code changes and track complexity issues as well as code quality software metrics are frequently used in the software development life cycle. Ideally, software metrics should be computed continuously during the development process to enable the best possible tracking. Moreover, software metrics should be definable by development teams to not only cover general factors, but to measure company, project or team specific goals [20].

Definitions of Software Metrics

Software metrics defined by Lord Kelvin, a physicist:

"When you can measure what you are speaking about, and can express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of meager and unsatisfactory kind: it may be beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science" [20].

Another definition given by George Miller, a psychologist:

"In truth, a good case could be made that if your knowledge is meager and unsatisfactory, the last thing in the world you should do is make measurements, and the chance is negligible that you will measure the right things accidentally".

These above two definitions discussed two things. First is that metrics are measurements to measure your progress. Second is measurement of how well you have achieved the goal [20].

Kitchenham Assumptions

Three assumptions related to software metrics stated by Dr. Barbara Kitchenham are:

- We can accurately measure some property of software or process.
- A relationship exists between what we can measure and what we want to know.
- This relationship is understood, has been validated, and can be expressed in terms of a formula or model.

12 Steps to Useful Software Metrics

- **Identify Metrics Customers**

  First step is to find who need the information and who's going to use metrics? This is very important step because this step will give direction to move further.

- **Target Goals**

  There are mainly three types of goals:

  - Organizational goals include developers should be the low cost provider and meet the projected revenue targets.
  - Project goals include deliver the product by deadline and finish the project within budget.
  - Tasks goals include effectively inspect software module and obtain 100% statement coverage during testing.

- **Ask Questions**

  Main goal behind asking questions is to maintain a high level of customer satisfaction. What is our current level of customer satisfaction, what attributes of our products and services are most important to our customers and how do we compare with our competition?

  There are two levels of goals, higher level of goal consider are all known defects corrected where as lower level ensure all known defects are corrected before shipment.

- **Select Metrics**

  In this step we have to choose appropriate metrics that provide information to help answer the questions. Metrics do not solve problems but actually people solve it and metrics only provide information so that people can make better decisions.

  Metrics objective statement is to understand, evaluate, control and predict the attribute of the entity in order to obtain goals for example metric to find percentage defects corrected.

  In this metric objective statement will become to evaluate the percentage defects found and corrected during testing in order to ensure all known defects are corrected before shipment. This is the benefit of this step.

- **Standardize Definition**

  Select definitions from the literature that matches your organizational goals. Use them as a basis for creating your own definitions. Apply them consistently and include them in an appendix to each metrics report.

- **Choose a Model**

  Models for code inspection metrics are metric primitive model. These models provide raw data physical attributes of the software that are later used as inputs for computed metrics. For example lines of code inspected, hours spent preparing, hours spent inspecting, discovered defects etc.

  Other metrics models are preparation rate, inspection rate, defect detection rate etc.

- **Establish Counting Criteria**

  For any organization it is necessary to select standard for any process for example if anybody wants to count the line of code there is variation in counting by two or more than two organisations result because no industry accepted the standard. SEI guidelines states check sheets for criteria.

- **Decide on Decision Criteria**

  Establish baselines for current values like problem report backlog and defect prone modules also for statistical analysis like defect density, fix response time, cycle time, variance from budget etc.

- **Define Reporting Mechanism**

  There are different ways to report metrics for example using pie chart, bar chart or some other way. In this step timing and delivery of metric report is also of main concern.

- **Determine Additional Qualifiers**

  A good metric is a generic metric. There are some additional qualifiers which provide demographic information. Allow detailed analysis at multiple levels and define additional data requirements. Examples of additional qualifier are reporting customer/customer group, root cause, phase found/phase introduced, release/product/product line and severity.

- **Collect Data**

  Metric primitive and additional qualifiers are the data which is to be collect and it should be collected by the data owner like engineer is the owner of data which contains time spent per task, inspection data including defects found and root cause of defects.

- **Consider Human Factors**

  The people side of the metrics equation is how measures affect people and how people affect measures. These above two facts summarize relationship between metrics and people [21].

2.3 Tool Support

There are so many tools developed for software metrics evaluation/analysis purpose. These tools are further characterized in two types and they are static metrics-based evaluation and analysis tool and runtime or dynamic metric-based evaluation and analysis tool.

Static tools study the behavior of software before runtime by processing the source code and calculating static metrics values whereas runtime or dynamic tools analyze the behavior of software at runtime [22].

NAME	TYPE	DESCRIPTION
Borland Together Control Centre(TCC)	Static	Computes several metrics on source code in different languages.
CodeCrawler	Static	Reverse engineering tool which combines metrics and software visualization.
Function Point Workbench	Static	Software tool supporting the function point analysis technique for sizing and evaluating software.
JMetric	Static	Open source static metric collection and analysis tool for java.
McCabe Tool Set	Static	Testing tools based around the McCabe metrics
Metrics	Static	Open Source Metrics Collection tool for java.
Moose	Static	Reverse engineering and analysis environment.
OOMeter	Static	Software quality assurance tool that work over static metrics. It supports Java and C#.
SDMetrics	Static	Computes number of metrics in XML files.
BoundsChecker	Dynamic	Run-time error detection and debugging tool for C++ developers.
Caffeine	Dynamic	Dynamic analysis tool that uses the Java platform debug architecture to generate a trace, i.e., an execution history, and a Prolog engine to perform queries over the trace.
*J	Dynamic	System for gathering, computing and presenting dynamic metrics from java programs.
Shimba	Dynamic	Environment for reverse engineering java software system.
Rational Purify	Dynamic	Performs memory corruption and memory leak detection functions and is available for both Windows and Linux platforms.
Fjalar	Dynamic	Framework that facilitates the extraction of dynamic analysis data from programs written in C.

Table 2.2 Metrics Tool [22]

JHawk

JHawk is a Java based open source framework which can be incorporated in any java application for performance testing. The idea is the developer has to define module and its tasks (essentially a java method) inside the application and register the same with JHawk. JHawk executes the modules and generates a graphical performance report which can be analyzed to find performance bottleneck of your application.

JHawk concentrate on the inherent quality of software. The process of taking code and analyzing it is called 'static analysis'. A number of metrics have been developed over the years that can give a view of aspects of software. The problem is that there are quite a lot of them - many of them either fully or partially duplicate others, others have the same name but may be calculated in a number of different ways, some contradict others and most of them are poorly understood by those who use them. There is a wide variety of information about these various metrics but it tends to be widely scattered and in some cases conflicting. JHawk aims to draw these strands together to allow you to make sensible decisions based on the results of analysis of your code. After a bit of experience in a particular language we can tell good code from bad.

For example consider the following two codes:

Code 1:

```
public static double zzz(double x, double z)
{
 z+=(z*x/100);
 return z;
}
```

Code 2:

```
public static double addInterestToBalance(double interestRate, double balance)

 {
 /* Add interest to balance */
 balance+=(balance*interestRate/100);
 return balance;
}
```

Most competent Java programmers will accept that the second example, while functionally identical to the first, is a far better piece of code. Reason for this are:
One thing is that the second piece of code is easier to read and appears to tell you what it is doing that is the second piece has meaningful comments and meaningful identifiers but what if this line is found in a class called ManageHeartRate.
if (heartRate < 70) stimulantDose =
MedicalUtilities.addInterestToBalance(10,stimulantDose);

This code does not make a sense here, so a lot of the things that are view as 'good' in a piece of code are contextual. This means that developers are going to be limited in what can sensibly analyze using software. Let's say to perform an analysis of the pieces of code above - one of the things might choose to count could be the number of comments. Analyst would then see that there no comments in the first piece of code. However could add a comment-say "/*HGHGHJGHJGJGHJGJGHJGJHGHJGJGJG JGJHG*/" - that would add nothing to understanding of the code. Similarly analyst could measure the length of the variable names - but anyone could change X to XDGSADGFDS and while it would be longer it would still be meaningless.The trouble is that no one can easily tell a machine how to make the kinds of distinctions that anyone can make in just a single glance.
So, having seen that some things are not easy to measure objectively, what are the kinds of things that can usefully measure by taking a piece of code, breaking it down into its constituent parts and measuring the numbers of these parts and some of their aspects?
To do this takes a look at the measurements that can take at Method, Class and package and System level. Define the System level as all of the code in the project that choose to examine as a collective entity. JHawk would evaluate all the files or Eclipse projects that we select for analysis [7].
There are many metrics that are able to be collected using JHawk tool. JHawk evaluates metrics at both method and class level. Metrics evaluated by tool are given below:

Complexity Metrics (COMP): Software complexity is the term that encompasses numerous properties of a piece of software, all of which affect internal interactions. It is proposed by McCabe. This metrics measures the number of decisions caused by conditional statements in the source code [23].

Number of Comment Lines and Java Statements: Number of comment lines (NOCL) includes white spaces, empty lines and comments in the source code and statements includes number of java statements (NOS) ended by semicolon.

Halstead's Length, Vocabulary, Effort and Bug: Program P is considered to be collection of tokens which is either operand or operator.

N1 = total occurrence of operator.

N2 = total occurrence of operands.

n1 = number of unique operators.

n2 = number of unique operands.

Halstead's length (HLTH) is sum of N1 and N2 whereas vocabulary is sum of n1 and n2. Halstead's effort (HEFF) is given by a formula which is E = ((n1/2)*(N2/n2)*Nlogn) where N is addition of n1logn1 and n2logn2. Halstead's bug (HBUG) is an estimate for the number of errors in the implementation [24].

Lack of Cohesion in Methods (LCOM): If a class C has n methods $M_1, \ldots, M_n$ then I_j is the set of instance variables used by method Mj. Let $|P|$ be the number of null intersections between instance variables sets. Let $|Q|$ be the number of non-empty intersections between instance variable sets.

Then:

LCOM = |P| − |Q| if |P| > |0| OR 0 otherwise.

LCOM measures the amount of method pairs which don't access the same instance variable minus the amount of method pairs which do access the same instance variable. Cohesiveness of methods within a class is desirable, since it promotes encapsulation. Lack of cohesion implies classes should probably be split into two or more classes [25].

Response for Class (RFC): RFC is defined as |RS| where RS is the response set for a class. The response set for a class can be expressed as:

$$RS = \{M\} \bigcup \{Ri\}$$

Where {Ri} = set of methods called by method I and {M} = set of all methods in the class.

The response set of a class is a set of methods that can be potentially executed in a response to a message received by an object of that class. Membership of the response set is defined only up to the first level of nesting of method calls due to the practical considerations involved in collection of the metric. If a large number of methods can be invoked in a response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding from the tester [25].

Coupling between Object (CBO): CBO is defined for a class as a count of the number of other classes to which it is coupled. A class is said to be coupled with another class if either one accessed the others methods, or variables. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application [25].

Unweighted Class Size (UWCS): This is calculated from the number of methods plus the number of attributes of a class. Smaller class sizes usually indicate a better designed system reflecting better distributed responsibilities. In other words you didn't just stuff all the functionality into one big class. It's difficult to set hard and fast rules about this but you should look carefully at classes where UWCS is above 100 [7].

Maintainability Index (MI): The MI is a composite number, based on several unrelated metrics for a software system. It is based on the Halstead Volume (HV) metric, the Cyclomatic Complexity (CC) metric, the average number of lines of code per module (LOC), and optionally the percentage of comment lines per module (COM). Halstead Volume, in turn, is a composite metric based on the number of distinct operators and distinct operands in source code. The complete fitting function is:

171-5.2*ln(HV)-0.23*CC-16.2*ln(LOC)+50*sin(sqrt(2.4 * COM))

The higher the MI, the more maintainable a system is deemed to be [26].

2.4 Statistical Analysis Using Correlation Coefficient

Correlation coefficient is a measure of strength of the relationship between x and y for the specific equation of best fit. For instance if the equation of best fit is linear, a correlation coefficient close to 1 or -1 suggests that x and y have a stronger relationship.

Its value always lies in the range of 1 and -1. When you consider correlation coefficient for a specific data set and best fit line, it tells you two things:

- The sign of correlation coefficient tells you something about the behavior of y. More specifically if the sign of r (Pearson's correlation coefficient) is positive the value of y increases when x is increasing. If sign is negative, then y decreases as x increases.
- The absolute value of coefficient tells you how strong the relationship between x and y. If the size is 1, data is perfectly linear. The closer value to zero indicates weaker linear relationship between x and y and the vice versa [27].

Assume that the data are an n*m matrix where n is the number of instances and m is the number of attributes of an instance. Let x and y is instances that contain m attributes. Mathematically, the Pearson correlation coefficient, $r_{x,y}$ between two instances x and y is defined as:

$$r_{x,y} = \quad r_{x,y} = \frac{\sum_{i=1}^{m} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{m} (x_i - \bar{x})} \sqrt{\sum_{i=1}^{m} (y_i - \bar{y})}}$$

where

$$\bar{x} = 1/m \sum_{i=1}^{m} x_i$$

And

$$\bar{y} = 1/m \sum_{i=1}^{m} y_i$$

The Pearson correlation coefficient is a measure of how two instances are linearly related. The value of r ranges from -1 to 1. It is closed to zero if two instances are uncorrelated. When it is positive, x and y is correlated. The higher value indicates the stronger correlation. If the value of $r_{x,y}$ is negative, then x and y are negatively correlated [28].

Chapter 3

PROBLEM SATEMENT

Modularity and maintainability have been identified by many researchers as success factors of Open Source Software (OSS) Projects. Modularity depends upon size, complexity, cohesion, and coupling software metrics [14] where as maintainability is influenced by LOC, Halstead's volume and cyclomatic complexity [30].

In order to accomplish the aim of this thesis, first problem arise is to select the appropriate software project. Java based OSS projects are selected since they are among the most popular object oriented programming for developing Open Source Softwares. Size of these projects is small to medium and they are selected on the basis of number of downloads.

Quantitative value that is producing numbers to characterize properties of source code. Second problem is selection of metrics to be evaluated. There are so many metrics that can be evaluated with tools available in the market nowadays.

Third and main problem is to select a technique to analyze the values of metrics and show the relationship among two metrics. There are many statistical terms available to show the relationship between two variables like Spearman's correlation, Pearson correlation, ANOVA techniques and so many others.

Chapter 4

WORK DONE

4.1 Data Collection Process

There are several considerations and assumptions to select which OSS project to be analysed and they are:

- Java based OSS projects are evaluated because they are among most popular object oriented programming for developing Open Source Software.
- The number of downloads may indicate the success of the project, projects with very large number of downloads are considered.
- Size of the software is from small to medium.
- Source code of the software is free from errors and in compile-able form.
- Four input softwares are Quilt, EMMA, Nat and JVMDI Code Analyser. These are Open Source code coverage tools.

4.1.1 Quilt

Quilt is a Java software development tool that measures coverage**,** the extent to which unit testing exercises the software under test. It is optimized with JUnit unit test package, the Ant java build facility and the Maven project management tool kit. The simplest measure of coverage is statement coverage. For example, if there are 100 lines in the program and only 75 are actually used when tests are being run, then the coverage may be said to be 75%. Quilt's coverage tool does not help to write better tests or better code but will show whether or not code is being tested [9]. It has a Apache software License of OSS [10].

4.1.2 EMMA

EMMA is also an open source toolkit for measuring and reporting java code coverage. EMMA is different from other tools because it uses a unique feature combination that is support for large scale enterprise software development while keeping individual developer's work fast and iterative. Using this tool every developer can now get code coverage for free and fast. It has Common Public License of OSS [11].

4.1.3 Nat

Nat is software which allows seeing how good JUnit tests are. It generates a report from code to graphically show how many of project's method are being tested and how well. It has GNU general public license of OSS [12].

4.1.4 JVMDI Code Coverage Analyzer

This small utility is shared library which when loaded into a Java VM (1.4+) which supports JVMDI will record all the lines of code executed. This is relatively unpolished method, but good enough for many purposes [13].

4.2 Tool Used

JHawk was released in 1999 and has become leader in provision of software metrics to java developer. JHawk is a Java based open source framework which can be integrated in any java application. It is a static code analysis tool. It operates at all levels of customers whether it is small customer, through consultancies and multinationals. JHawk 5.1 is latest release [7].

JHawk compute values of metrics at both method and class level. New version of JHawk has two features differentiating it from other versions and they are snapshot comparison view and dashboard tables. Other versions and this new version provide both Standalone and Eclipse plug-in. It allows to create own metrics integrated with JHawk.

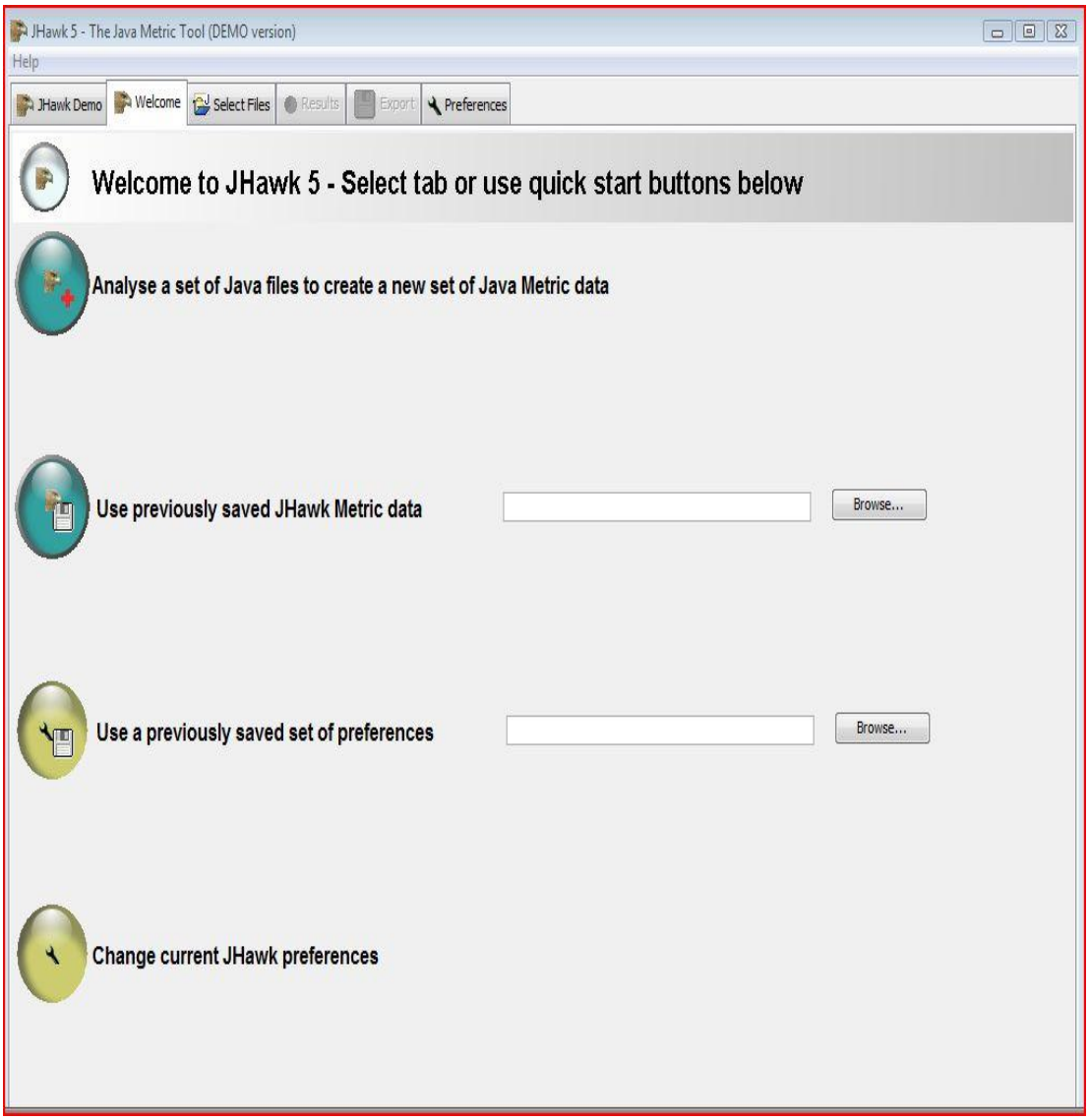


Figure 4.1 Welcome Tab

This window provides access to number of features in the product. This is very first tab when we open JHawk tool. This window allows us to access previously saved data by just browsing the file from its location.

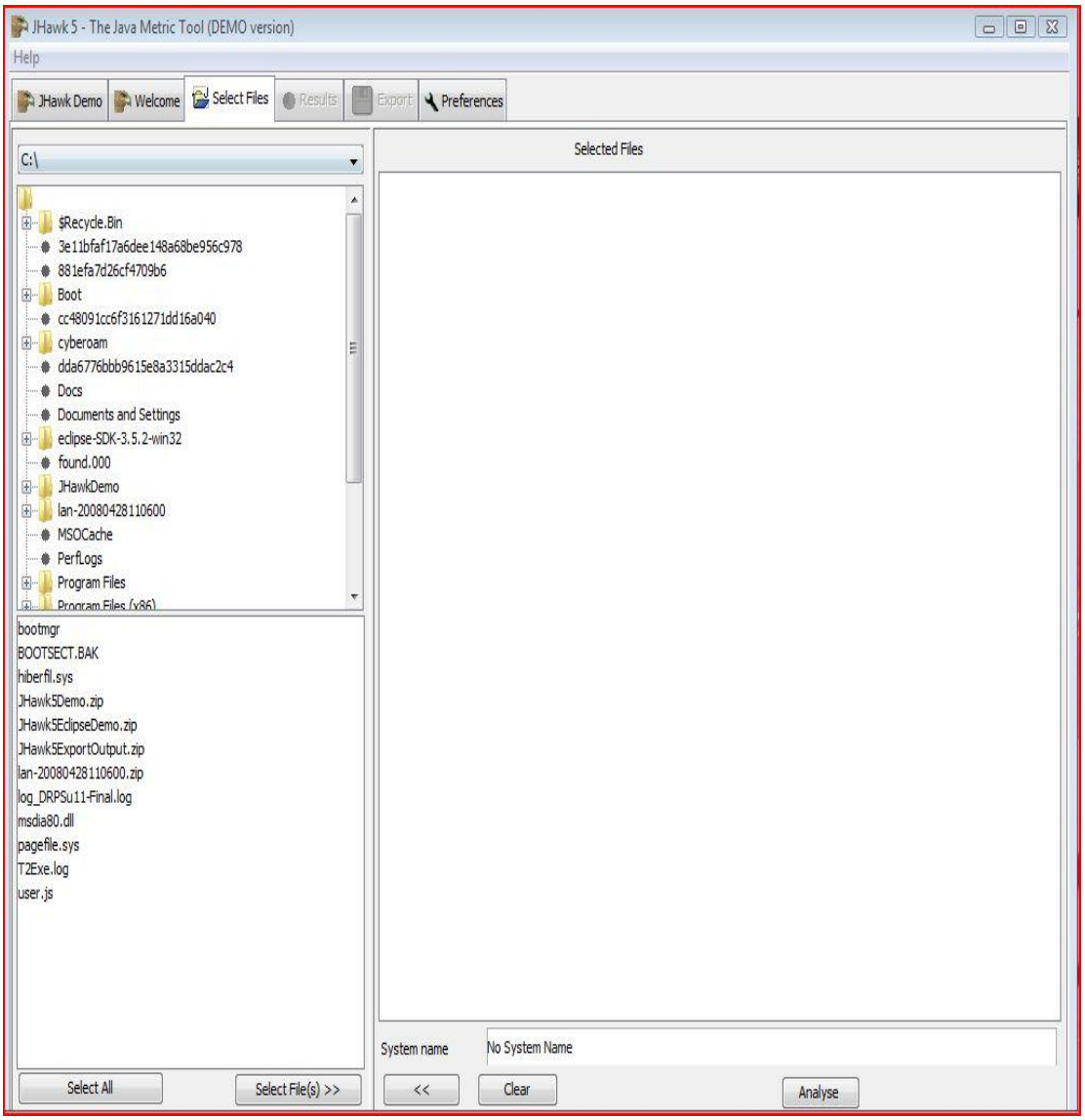


Figure 4.2 Select Tab

This figure shows the tab which allows selection of java files to be evaluated. Here we select files written in java and saved with .java extension. We can select file from any drive in computer, means tool does not limit the browsing space. We can also check the value of metrics for system files which are mostly stored in c drive.

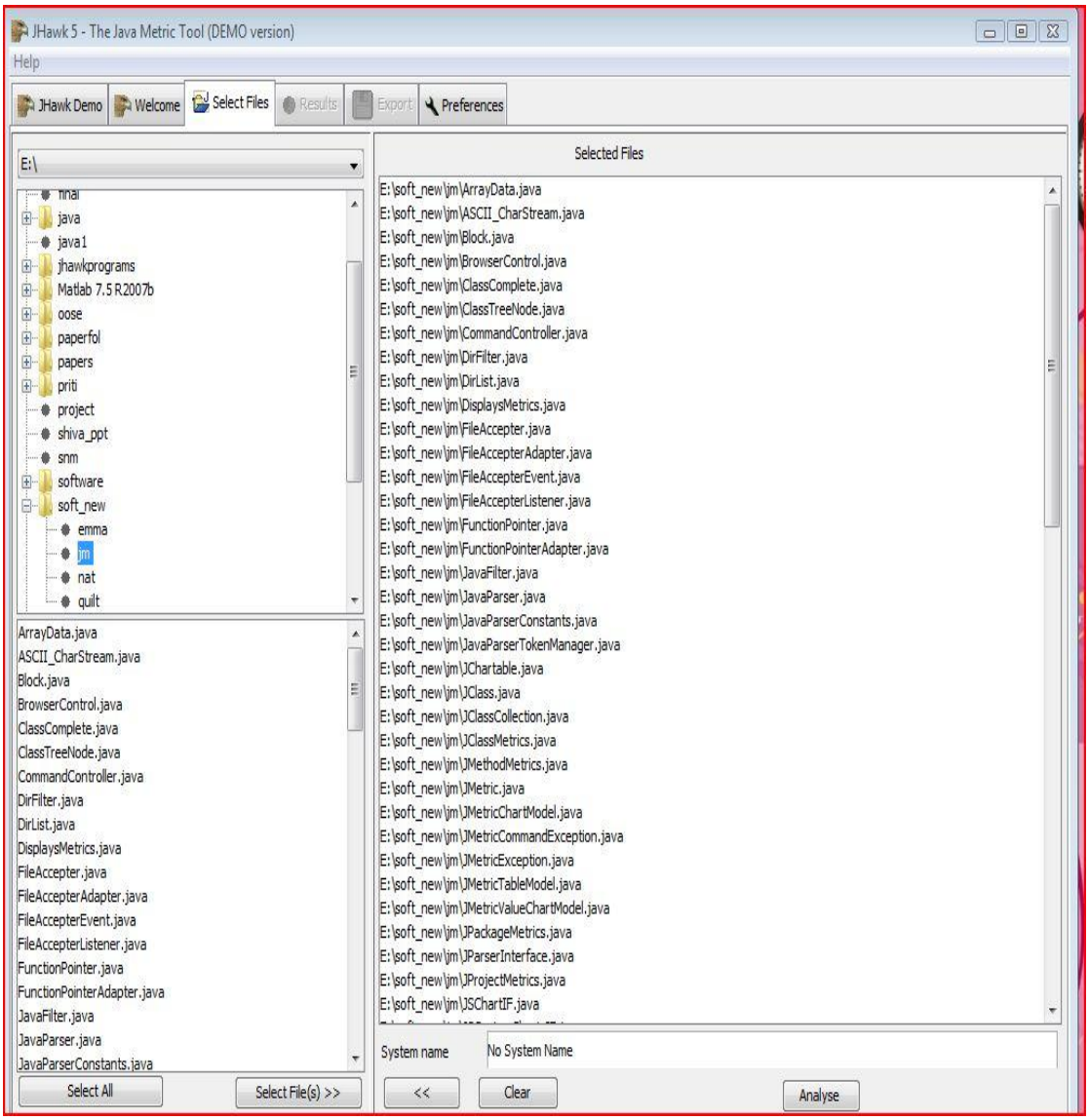


Figure 4.3 Selection of Files

This figure shows the selection of files from the E drive of computer. Files of JVMDI Code Analyzer are being selected from soft new folder. All the files with .java extension are selected and unless we select files and then press analyse, tabs of results and export are inactive.

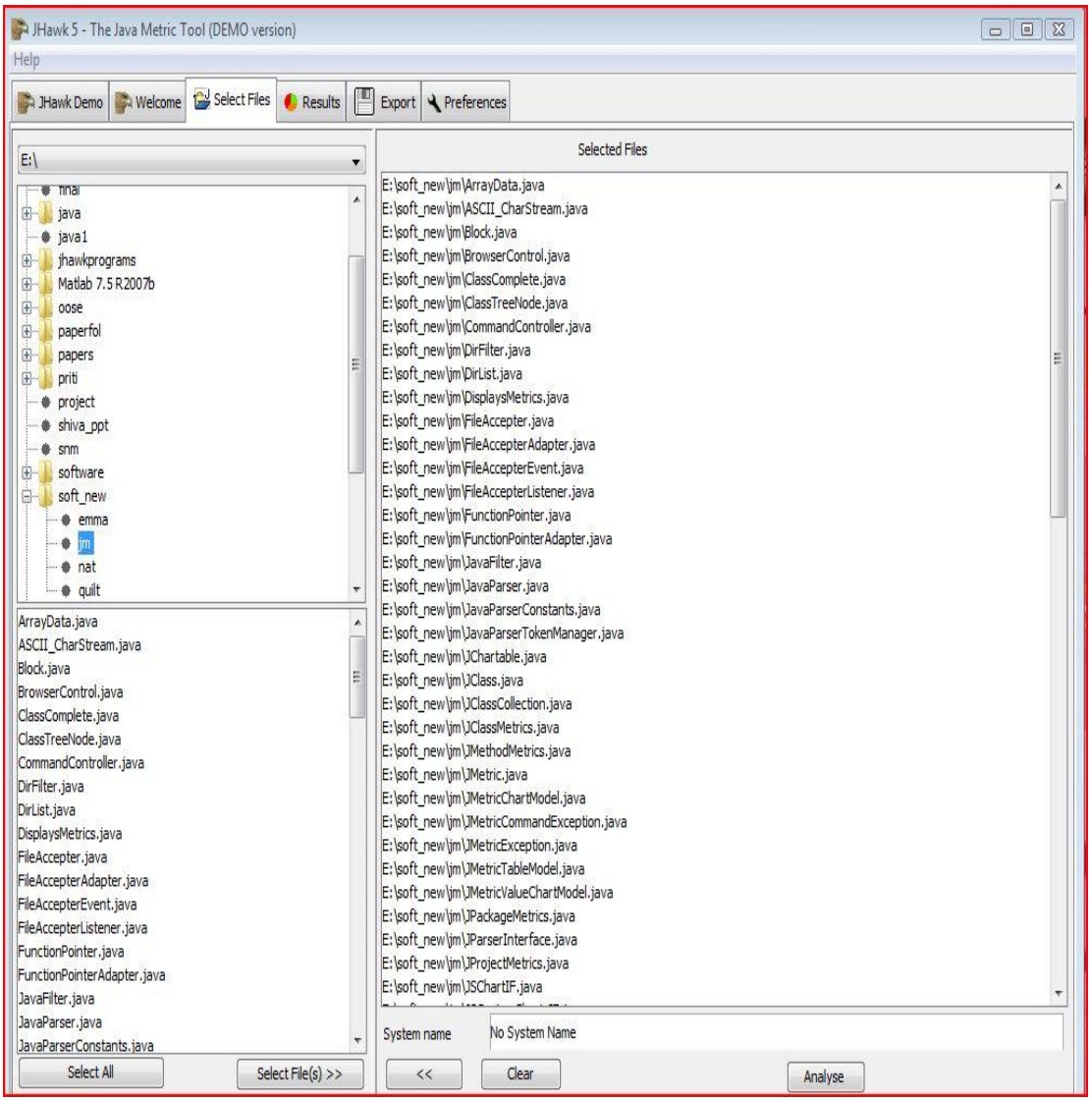


Figure 4.4 Selections of Files With Active Result Tab

This figure shows the same thing which is already discussed in the previous figure but here we can see the result and export tabs in active state which is result of selection of files and then selection of analyse button which is there at the right bottom of the screen.

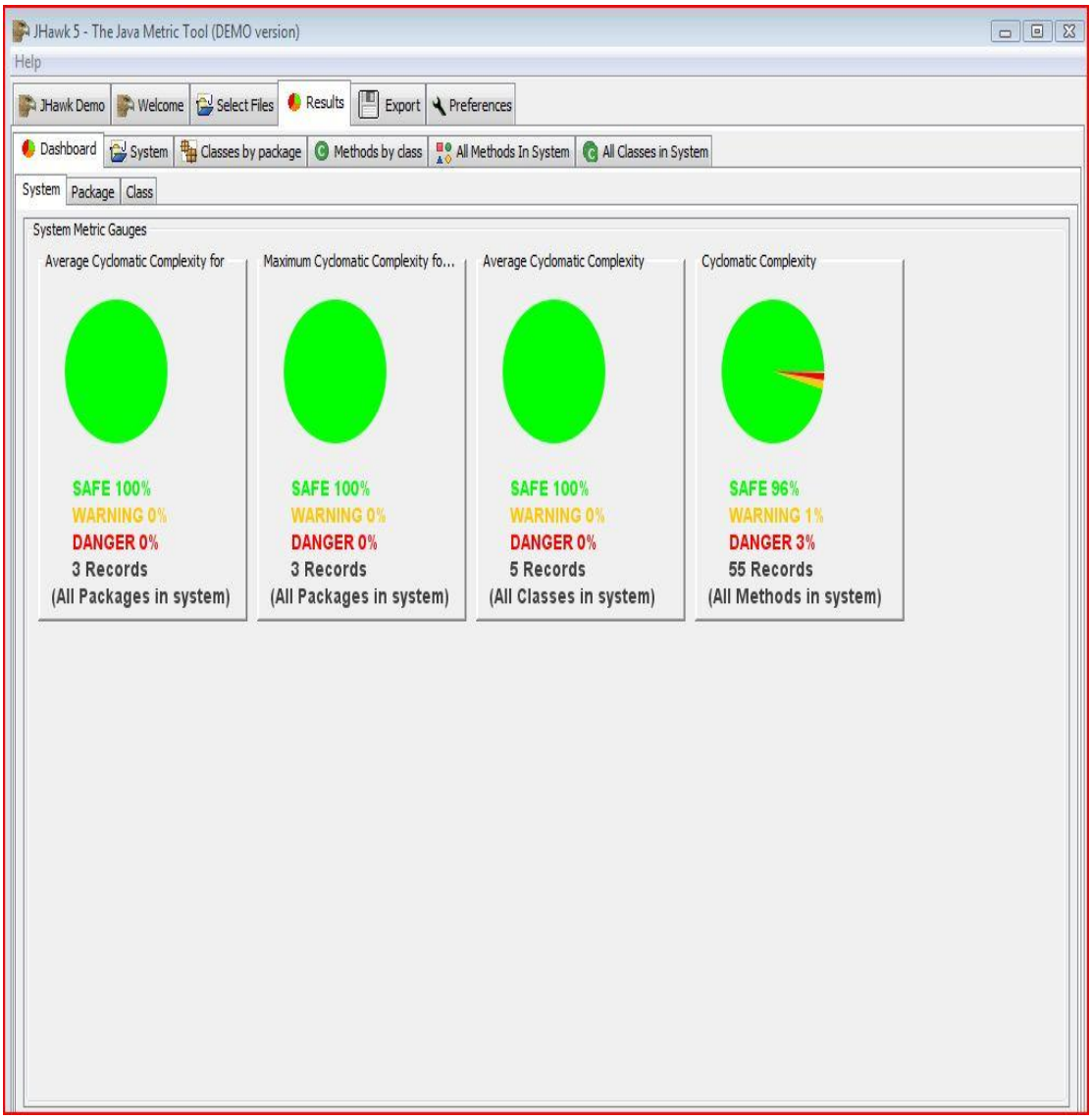


Figure 4.5 Diagrammatical Representations of Cyclomatic Complexity Levels

This figure shows how JHawk diagrammatically represent the various levels of the cyclomatic complexity. Above figure indicates the safe cyclomatic complexity at package and class level but orange and red colour shows the warning and danger of cyclomatic complexity at method level.

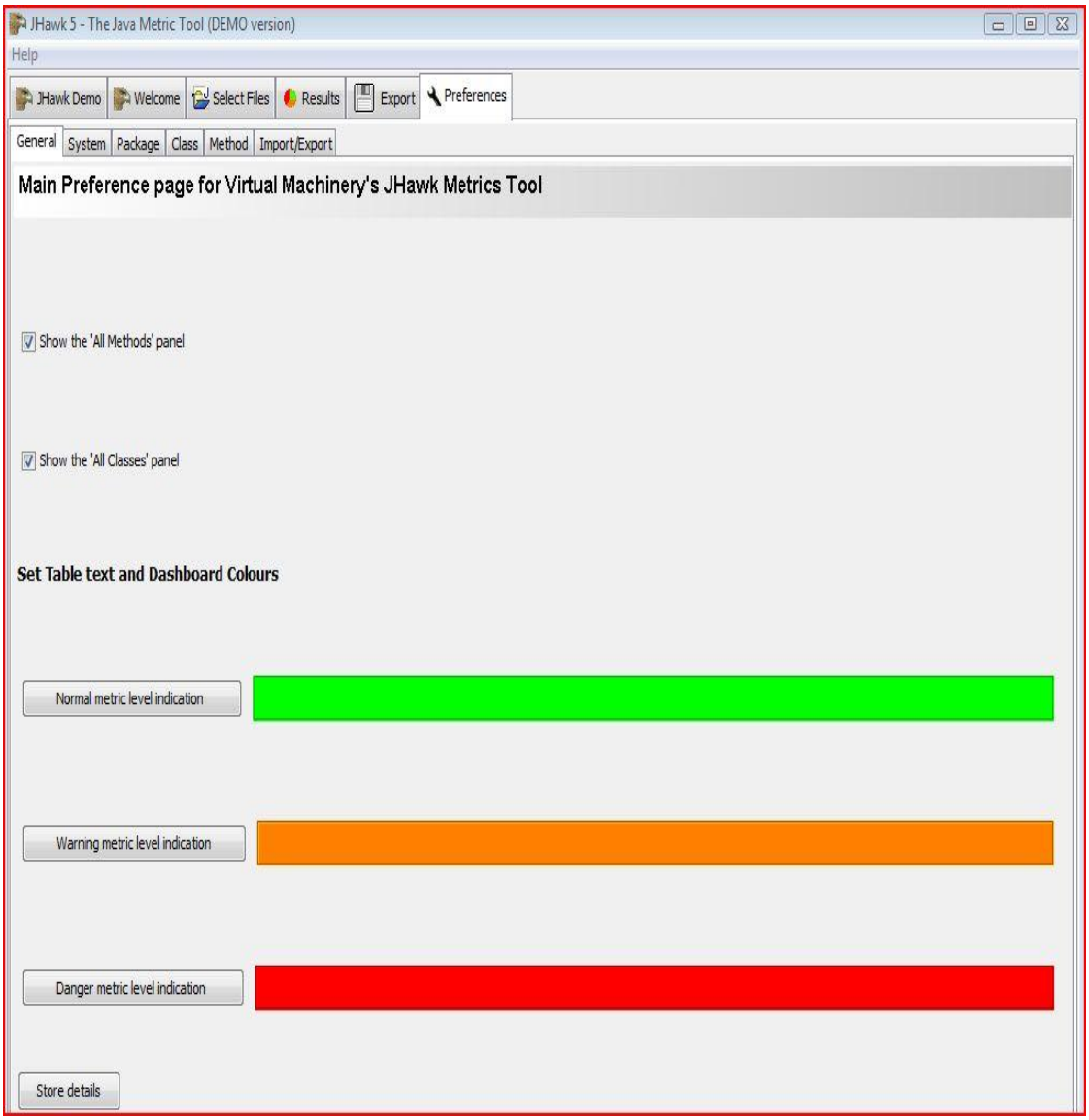


Figure 4.6 Metric Level Indications

This figure shows the tab which contains information of colour indication used by JHawk to depict the levels of danger in metrics. Red colour is used to show danger level, orange to indicate warning level whereas green to show everything is fine with code.

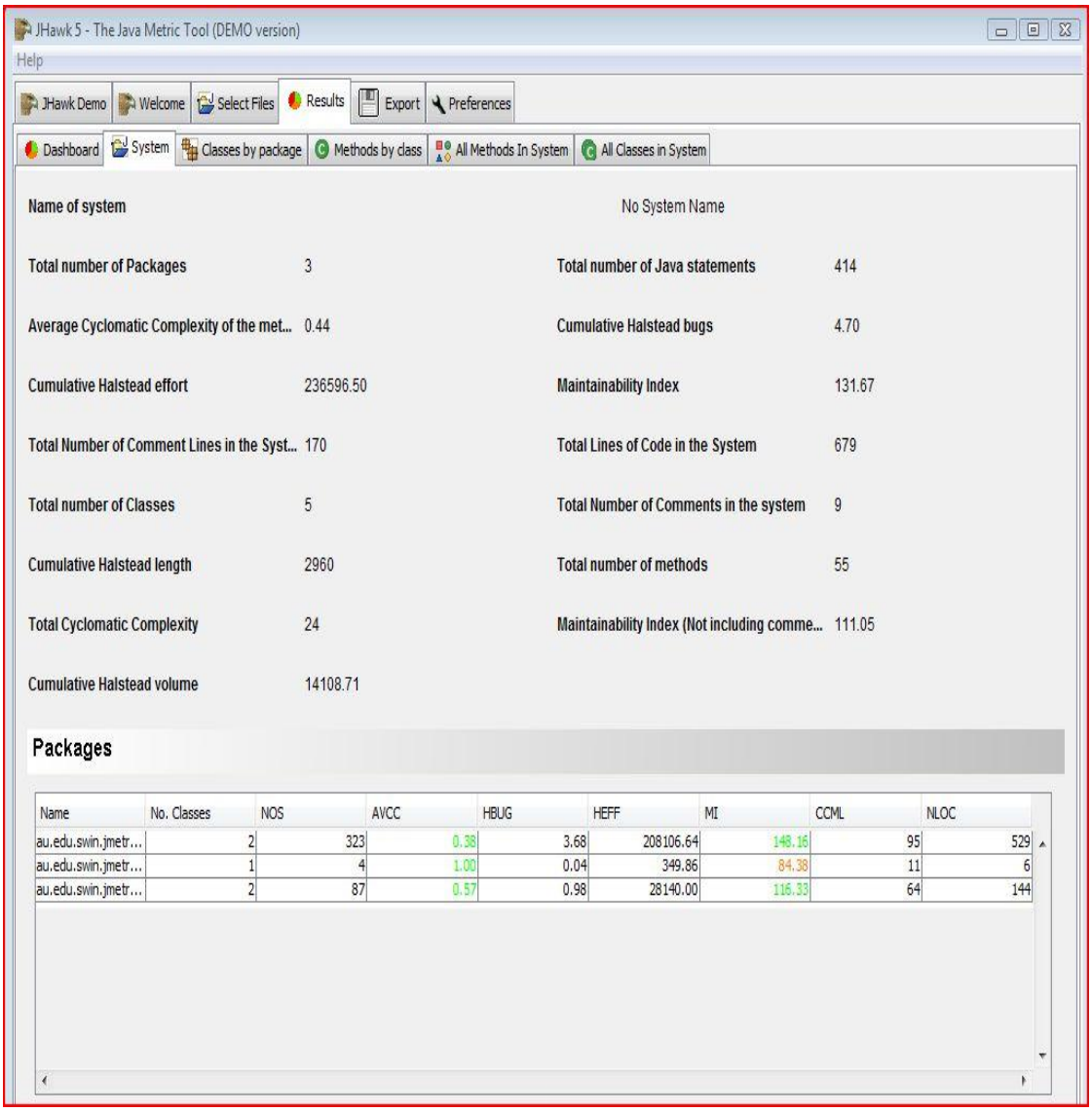
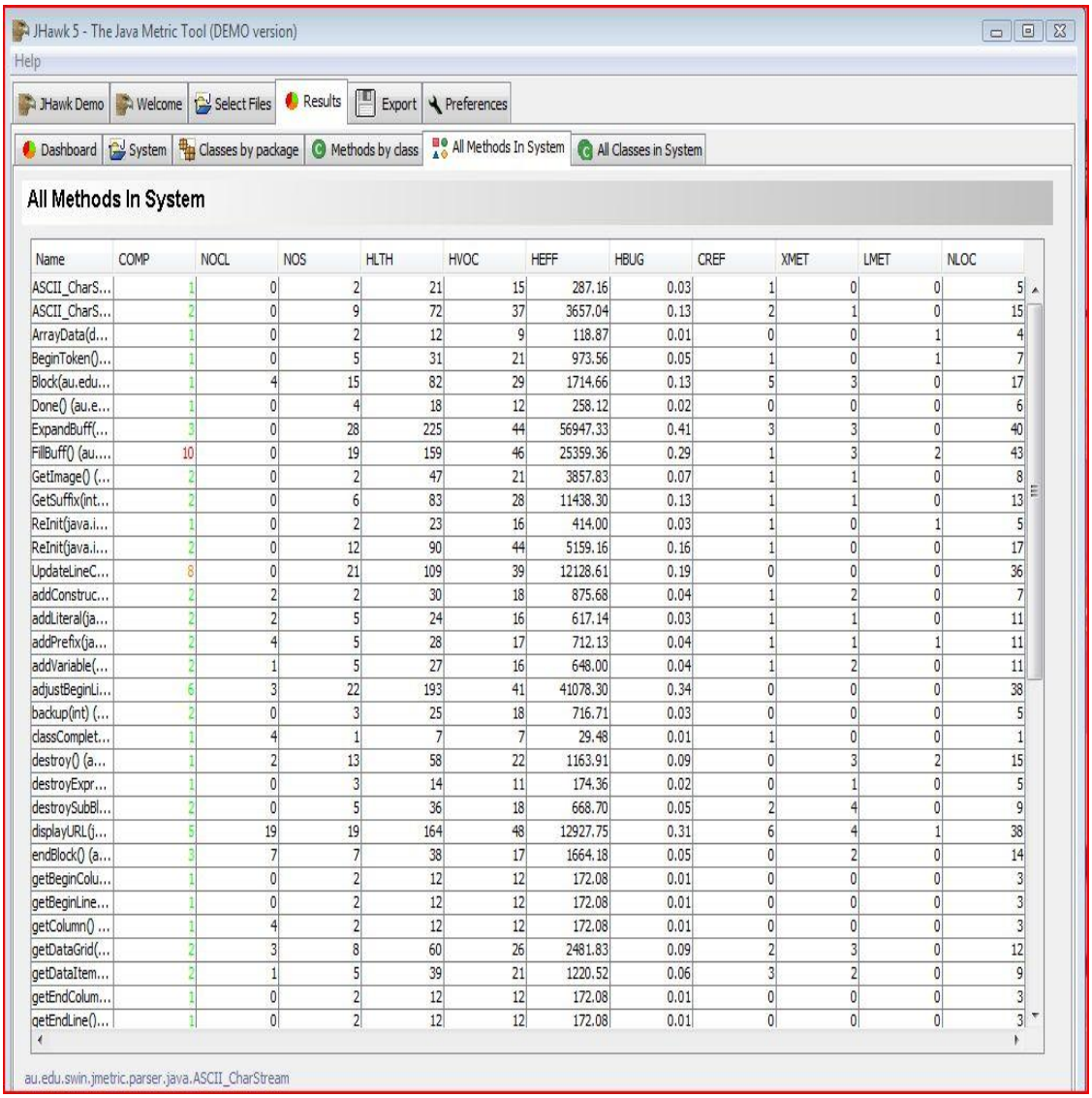


Figure 4.7 System Level Metrics

This figure is used to some metrics at system level like Cumulative Halstead Effort, Total Number of Comment Lines in the System, Total Number of Classes, Total Number of Methods, Cumulative Halstead Volume, Total Number of Package, Maintainability Index(including comments) and Maintainability Index.



Name	COMP	NOCL	NOS	HLTH	HVOC	HEFF	HBUG	CREF	XMET	LMET	NLOC
ASCII_CharS...	1	0	2	21	15	287.16	0.03	1	0	0	5
ASCII_CharS...	2	0	9	72	37	3657.04	0.13	2	1	0	15
ArrayData(d...	1	0	2	12	9	118.87	0.01	0	0	1	4
BeginToken()...	1	0	5	31	21	973.56	0.05	1	0	1	7
Block(au.edu...	1	4	15	82	29	1714.66	0.13	5	3	0	17
Done() (au.e...	1	0	4	18	12	258.12	0.02	0	0	0	6
ExpandBuff(...	3	0	28	225	44	56947.33	0.41	3	3	0	40
FillBuff() (au....	10	0	19	159	46	25359.36	0.29	1	3	2	43
GetImage() (...	2	0	2	47	21	3857.83	0.07	1	1	0	8
GetSuffix(int...	2	0	6	83	28	11438.30	0.13	1	1	0	13
ReInit(java.i...	1	0	2	23	16	414.00	0.03	1	0	1	5
ReInit(java.i...	2	0	12	90	44	5159.16	0.16	1	0	0	17
UpdateLineC...	8	0	21	109	39	12128.61	0.19	0	0	0	36
addConstruc...	2	2	2	30	18	875.68	0.04	1	2	0	7
addLiteral(ja...	2	2	5	24	16	617.14	0.03	1	1	0	11
addPrefix(ja...	2	4	5	28	17	712.13	0.04	1	1	1	11
addVariable(...	2	1	5	27	16	648.00	0.04	1	2	0	11
adjustBeginLi...	6	3	22	193	41	41078.30	0.34	0	0	0	38
backup(int) (...	2	0	3	25	18	716.71	0.03	0	0	0	5
classComplet...	1	4	1	7	7	29.48	0.01	1	0	0	1
destroy() (a...	1	2	13	58	22	1163.91	0.09	0	3	2	15
destroyExpr...	1	0	3	14	11	174.36	0.02	0	1	0	5
destroySubBl...	2	0	5	36	18	668.70	0.05	2	4	0	9
displayURL(j...	5	19	19	164	48	12927.75	0.31	6	4	1	38
endBlock() (a...	3	7	7	38	17	1664.18	0.05	0	2	0	14
getBeginColu...	1	0	2	12	12	172.08	0.01	0	0	0	3
getBeginLine...	1	0	2	12	12	172.08	0.01	0	0	0	3
getColumn() ...	1	4	2	12	12	172.08	0.01	0	0	0	3
getDataGrid(...	2	3	8	60	26	2481.83	0.09	2	3	0	12
getDataItem...	2	1	5	39	21	1220.52	0.06	3	2	0	9
getEndColum...	1	0	2	12	12	172.08	0.01	0	0	0	3
getEndLine()...	1	0	2	12	12	172.08	0.01	0	0	0	3

Figure 4.8 Method Level Metrics

This figure shows how JHawk shows the value of various metrics at method level. Method level metrics includes Complexity, Number of Comment Lines, Number of Java Statements, Halstead Length, Halstead Vocabulary, Halstead effort, Halstead Bug, Classes Referenced, Number of Methods External to this Class called by this Method and Number of Methods Local to this Class called by this Method.

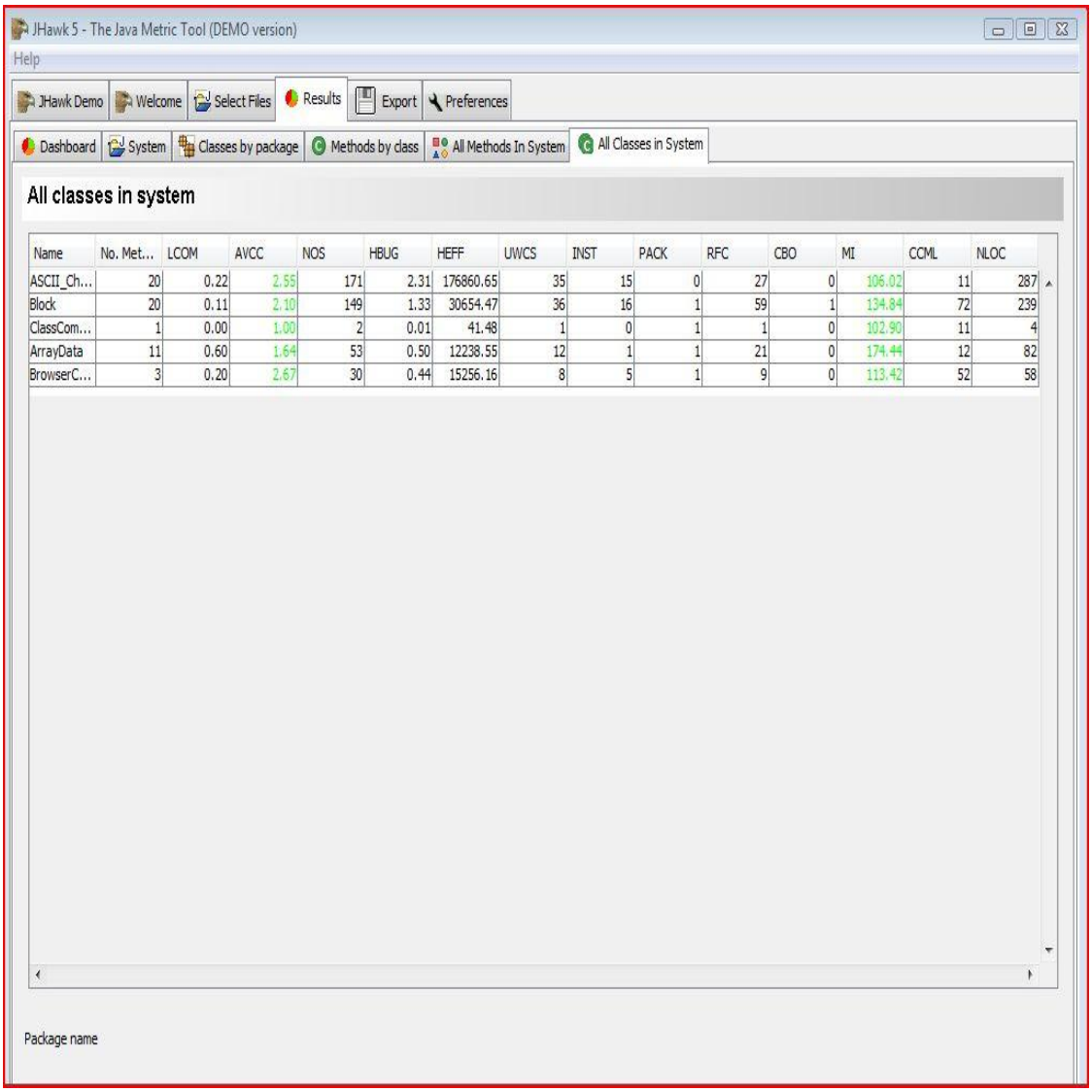


Figure 4.9 Class Level Metrics

This figure shows how JHawk shows the value of various metrics at class level. Class level metrics includes Number of Methods, Lack of Cohesion of Method, Average Cyclomatic Complexity, Number of Java Statements, Halstead Bug, Halstead Effort, Unweighted Class Size, Total Number of Instance Variables declared, Total Response for Class, Total Line of Code, Coupling Between Objects and Maintainability Index.

4.3 Metrics Used

As discussed before in this thesis, JHawk evaluate values of metrics at both method and class level. In this section we will study basics behind the metrics and the way used by tool to find the values.

Starting with metrics at method level, first and very basic metric is number of line of code and number statement. JHawk instead of LOC consider approach to the number of java statements. JHawk reads java statement terminated by semicolon. Number of comments is also valuable but a matter of contention. Comments are relief to complex code is a general view. We cannot parse comments to get their original meaning but writing comment that is appropriate will help understanding code of one person by another. Writing appropriate description about code at suitable place should be considered [7]. For example:

Code 1: public double calculateBalanceWithInterest(double rate, double minForInterest) {

```
        //Only calculate interest if the balance is over the minimum amount
        if (balance > minForInterest) {
                double interest = balance*rate;
                balance = balance +interest;
        }
return balance;
}
```

Code 2: public double calculateBalanceWithInterest(double rate, double minForInterest) {

```
        //wash your clothes
        if (balance > minForInterest) {
                double interest = balance*rate;
                balance = balance +interest;
        }
return balance;
}
```

Code 1 is with appropriate comment and useful but code 2 does not have any significant comment in it and is useless. Expression is also useful in java programming. Expression is something that returns a value. Expression is also related to complexity as we increase average number of expression to the statement it is directly proportional to complexity [7].

Halstead metrics is dependent upon the number of operator and operands. Operator is the one who carries out the action and operand is one who participates in such actions. Example of operators are +, - and such other things whereas operands in java programming can be X, Y, 5 etc. JHawk does not only count the number of operator and operands but only counts unique operators and operands [31]. Comments are never included in the Halstead metrics. Halstead metrics calculated at method level by this tool are Halstead's length, Halstead Volume, Halstead Vocabulary, Halstead Bugs, Halstead's Effort, and Halstead's Difficulty. Formula used by JHawk is same which is standard and has been discussed before in this thesis.

Maintainability Index is a set of polynomial metrics developed at the University of Idaho, using Halstead's effort and McCabe's cyclomatic complexity, plus some other factors relating to the number of lines of code and the percentage of comments. The MI is used primarily to determine if code has a high, medium or low degree of difficulty to maintain. It is language independent. MI less than 65 is difficult to maintain, 65-85 is reasonable maintainable and value greater than 85 have greater maintainability. JHawk gives separate figure for MI which takes account comments and one which does not take account of comments. Two reasons behind these two computations is firstly formula used to find the comments is not standardized and comments are subjective and it is better to compute the value of maintainability index without comments [7].

Other metrics is number of external methods called and the more external methods that a class calls the more tightly bound that class is to other classes. If you feel that a class is too tightly bound then this column will help you to identify the methods that are responsible for external calls. You can see the actual methods called and the number of times that they are called by double clicking on the selected method record to bring up the method details dialog.

Some other metrics at method level are number of variables declared and number of variables referenced. If value of these both metrics are larger than this implies that method is doing so much. Max nesting is also a part of metrics at method level and if

the value of this metrics is large than it indicates huge complexity and we can reduce the depth of nesting by splitting the functionality of method. There are three types of method to split methods into smaller chunks and they are sequential chunks, conditional chunks and nested chunks depending upon the structure of the program [7].

Metrics at class level are not just metrics which measures the aspects of the class but metrics which give us information on the interaction between classes. At basic level metrics we are interested in metrics accumulated from method level metrics example number of methods and statements in the class. We are interested in average, total and maximum cyclomatic complexity and maintainability index at this level.

Lack of Cohesion in Method metric measures the correlation between the methods and the local instance variables of a class. High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. It is calculated as the ratio of methods in a class that do not access a specific data field, averaged over all data fields in the class. The LCOM version that we use in JHawk is LCOM* as defined by Henderson-Sellars. This algorithm should produce results in the range 0 to 1 with zero representing perfect cohesion; however we have noticed that some values exceed 1. This occurs in following cases:

- When there are instance attributes defined in the class.
- Where there are no references to instance attributes but there are instance attributes defined.
- Where the number of references to attributes is less than the number of attributes [7].

Unweighted class size is calculated from the number of methods plus the number of attributes of a class. Smaller class sizes usually indicate a better designed system reflecting better distributed responsibilities. In other words you didn't just stuff all the functionality into one big class. It's difficult to set hard and fast rules about this but you should look carefully at classes where UWCS is above 100 [7].

Response for class is one measure of coupling. This measures the complexity of the class in terms of method calls. It is calculated by adding the number of methods in the class (not including inherited methods) plus the number of distinct method calls made by the methods in the class (each method call is counted only once even if it is called

from different methods). JHawk follows this definition. There is only one other common alternative approach to RFC and that is a version which includes the full call graph of any method called from the originating method i.e. method calls are totaled recursively until the end of the call stack is reached. This introduces difficulties about deciding which methods should be included [7].

Message Passing Coupling metric measures the numbers of messages passing among objects of the class. A larger number indicates increased coupling between this class and other classes in the system. This makes the classes more dependent on each other which increases the overall complexity of the system and makes the class more difficult to change [7].

Fan Out is defined as the number of other classes referenced by a class. Most differences in interpretation hang on the definition of references. This is often looser than the tight definition outlined in our initial paragraph above. Fan In is the number of other classes that reference a class [7].

Efferent Coupling is viewed as equivalent to Fan out and Afferent coupling to Fan In. Definitions of Afferent and Efferent Coupling tend to be stricter for those for Fan In and Fan Out [7].

JHawk provides Fan In and Fan Out measures and takes the view that CBO is equivalent to Fan Out.

The primary reason for taking the view that FanOut is equivalent to CBO is that when we list metrics we are viewing them from the class perspective and we will take action based on the metrics that we see. If we measure coupling based on the total of the classes referenced by and the classes that reference a class then it may be possible that a high CBO figure could be a result of a large number of classes referencing it. If you are the programmer responsible for that class but not the referencing classes then there is nothing that you can do to modify the class and reduce the CBO figure. Using fan-out shows those classes that make a large number of class references and allows you to tackle the issue by changing the number of classes that the class references. There is also a view expressed that high fan-outs represent class coupling to other classes/objects and thus are undesirable while high fan-ins represent good object designs and high level of reuse - taking this view would suggest that metrics that combine these in an additive relationship are meaningless as the two measures effectively negate each other [7].

Reuse Ratio is calculated as number of super classes above this class in the class hierarchy divided by total number of classes in the class hierarchy. Specialization Ratio is calculated as number of subclasses below this class in the class hierarchy divided by number of super classes above this class in the class hierarchy [7].

Number of External Methods Called, Number of Methods Called in class hierarchy and Number of local methods called are some other metrics and these give some idea of the level and complexity of interaction between the class and other classes (both in the class hierarchy and external to it). These figures are used in calculating metrics such as RFC, LCOM, MPC, Fan In and Fan Out. No. of instance variables, No. of modifiers, No. of interfaces implemented and No. of packages imported. These give additional information about the class level of semantic complexity. Large values for these can suggest that a class is doing too much [7].

Chapter 5

RESULTS AND DISCUSSION

Using JHawk tool four open source software has been evaluated and the value of their metrics is studied in this thesis. Four open source software which are given as an input are quilt, EMMA, Nat, and JVMDI Code Analyzer and all these softwares are java code coverage tools. We have statistically and graphically studied these values and the relationship among two variables.

5.1 Statistical Results

Pearson's correlation is used to measure the strength of the relationship between x and y for the specific equation of best fit. For instance if the equation of best fit is linear, a correlation coefficient is close to 1 or -1, which suggests that x and y have stronger linear relationship. Pearson product-moment correlation coefficient sometimes referred to as the PPMCC or PCC or Pearson's r is developed by Karl Pearson.

Correlation coefficient is always a value such that $-1 \leq r \leq 1$. When we consider correlation coefficient for a specific data set and best fit line, it tells two things:

- The sign of correlation coefficient tells you something about the behaviour of y. More specifically if the sign of r is positive the value of y increases when x is increasing. If the sign is negative, then y decreases as x increases.
- The absolute value of the coefficient (its size ignoring the sign) tells you how strong the relationship between x and y. If the size is one, data is perfectly linear. The closer the value is to zero, the weaker the linear relationship between x and y. The closer to one the stronger the relationship.

Pearson's correlation coefficient between two variables is defined as the covariance of the two variables divided by the product of their standard deviations.

The form of the definition involves a product moment, that is, the mean (the first moment about the origin) of the product of the mean-adjusted random variables, hence the modifier product moment in the name.

Range of correlation coefficient is used to indicate specific kind of relationship between two variables which is indicated in table on next page. This table is used to analyze the relationship between two metrics of OSS.

Range of r	Relationship
+0.70 to +1.00	Very strong relationship
+0.40 to +0.69	Strong positive relationship
+0.30 to +0.39	Moderate positive relationship
+0.20 to +0.29	Weak positive relationship
+0.01 to +0.19	No or negligible relationship
-0.01 to -0.19	No or negligible relationship
-0.20 to -0.29	Weak negative relationship
-0.30 to -0.39	Moderate negative relationship
-0.40 to -0.49	Strong negative relationship
-0.70 to -1.00	Very strong negative relationship

Table 5.1 Range and Relationship of Pearson Correlation Coefficient

	N	RANGE	MIN	MAX	MEAN	STD. DEVIATION	VARIANCE	SKEWNESS	KURTOSIS
NOM	6.00	43.00	2.00	45.00	13.66	16.45	270.66	1.798	3.398
LCOM	6.00	0.33	.00	.33	.0717	.1301	.017	2.183	4.818
AVCC	6.00	1.23	1.00	2.23	1.566	.5544	.307	.125	-2.542
NOS	6.00	268.00	6.00	274.00	77.00	102.71	10551.2	1.854	3.862
HBUG	6.00	3.41	.03	3.44	.8617	1.312	1.712	2.068	4.489
HEFF	6.00	173632.71	297.52	173930.23	37358.04	67924.47	4.614E9	2.291	5.373
UWCS	6.00	68.00	2.00	70.00	20.166	25.84	668.16	1.908	3.791
INST	6.00	25.00	.00	25.00	6.500	9.481	89.90	2.027	4.178
PACK	6.00	9.00	7.00	16.00	11.500	3.674	13.50	.490	-1.467
RFC	6.00	105.00	2.00	107.00	32.667	40.411	1633.06	1.538	2.330
CBO	6.00	1.00	.00	1.00	.333	.5167	.267	.968	-1.875
MI	6.00	64.83	71.26	136.09	107.39	29.90	894.27	-.319	-2.506
CCML	6.00	327.00	.00	327.00	87.66	128.25	16448.26	1.671	2.705
NLOC	6.00	380.00	9.00	389.00	1.08.55	146.13	21354.700	1.862	3.654

Table 5.2 EMMA Class Statistical Result

	NoM	LCOM	AVCC	NOS	HBUG	HEFF	UWCS	INST	PACK	RFC	CBO	MI	CCML	NLOC
NoM	1.00													
LCOM	.448	1.00												
AVCC	.824	.627	1.00											
NOS	.928	.530	.928	1.00										
HBUG	.986	.441	.812	.943	1.00									
HEFF	.986	.441	.812	.943	1.00	1.00								
UWCS	.986	.441	.812	.886	.943	.943	1.00							
INST	.986	.441	.812	.886	.943	.943	1.00	1.00						
PACK	-.110	-.318	-.141	-.062	-.185	-.185	-.031	-.031	1.00					
RFC	.928	.530	.928	1.00	.943	.943	.886	.886	-.062	1.00				
CBO	.630	.320	.840	.828	.621	.621	.621	.621	.335	.828	1.00			
MI	-.899	-.177	-.725	-.886	-.943	-.943	-.829	-.829	1.85	-.886	-.621	1.00		
CCML	.955	.188	.770	.880	.941	.941	.941	.941	-.033	.880	.660	-.941	1.00	
NLOC	.928	.530	.928	1.00	.943	.943	.886	.886	-.062	1.00	.828	-.886	.880	1.00

Table 5.3 EMMA Class Correlation Result

	N	RANGE	MIN	MAX	MEAN	STD. DEVIATON	VARIANCE	SKEWNESS	KURTOSIS
COMP	82.00	17.00	1.00	18.00	1.975	2.3039	5.308	4.836	29.669
NOCL	82.00	20.00	.00	20.00	5.341	3.552	12.621	1.338	4.803
HLTH	82.00	497.00	6.00	503.00	34.353	61.301	3757.86	5.932	42.841
NOS	82.00	58.00	1.00	59.00	5.061	7.590	57.614	5.007	31.836
HVOC	82.00	93.00	6.00	99.00	17.00	12.921	166.998	3.735	19.738
HEFF	82.00	88668.45	31.02	88699.47	2510.489	10261.77	1.053E8	7.958	63.192
HBUG	82.00	1.10	.01	1.11	.0554	.13204	.017	6.636	51.233
CREF	82.00	13.00	0.00	13.00	1.622	2.0284	4.115	3.198	13.204
XMET	82.00	26.00	0.00	26.00	1.414	3.2959	10.863	5.538	38.684
LMET	82.00	5.00	.00	5.00	.7683	1.269	1.612	1.896	3.171
NLOC	82.00	89.00	1.00	90.00	7.304	11.118	123.62	5.507	38.192

Table 5.4 EMMA Method Statistical Result

	COMP	NOCL	HLTH	HBUG	XMET	LMET	HVOC	NOS	HEFF	CREF	NLOC
COMP	1.00										
NOCL	.248	1.00									
HLTH	.822	.279	1.00								
HBUG	.846	.264	.950	1.00							
XMET	.737	.324	.813	.810	1.00						
LMET	.585	.039	.619	.645	.428	1.00					
HVOC	.779	.303	.971	.940	.795	.593	1.00				
NOS	.869	.128	.902	.904	.801	.657	.858	1.00			
HEFF	.817	.263	.982	.914	.789	.566	.964	.883	1.00		
CREF	.527	.174	.659	.655	.679	.466	.673	.598	.635	1.00	
NLOC	.885	.170	.914	.903	.791	.658	.873	.980	.898	.599	1.00

Table 5.5 EMMA Method Correlation Result

	N	RANGE	MIN	MAX	MEAN	STD. DEVIATION	VARIANCE	SKEWNESS	KURTOSIS
NOM	5.00	19.00	1.00	20.00	11.00	9.0277	81.550	-.031	-2.878
LCOM	5.00	.60	.00	.60	.2260	.22645	.051	1.406	2.627
AVCC	5.00	1.67	1.00	2.67	1.9920	.68751	.473	-.684	-.771
NOS	5.00	169.00	2.00	171.00	81.00	74.749	5587.50	.395	-2.662
HBUG	5.00	2.30	.01	2.31	.9180	.9131	.834	1.005	.174
HEFF	5.00	176819.17	41.48	176860.65	47010.262	73403.161	5.388E9	2.115	4.561
UWCS	5.00	35.00	1.00	36.00	18.400	16.102	259.300	.336	-2.852
INST	5.00	16.00	.00	16.00	7.400	7.653	58.33	.369	-3.029
PACK	5.00	1.00	.00	1.00	.800	.44721	.200	-2.236	5.000
RFC	5.00	58.00	1.00	59.00	23.400	22.33	498.800	1.157	1.552
CBO	5.00	1.00	.00	1.00	.200	.44721	.200	2.236	5.000
MI	5.00	71.54	102.90	174.44	126.32	29.64	878.78	1.418	1.492
CCML	5.00	61.00	11.00	72.00	31.60	28.64	820.30	.876	-1.786
NLOC	5.00	283.00	4.00	287.00	134.00	122.28	14953.50	.447	-2.433

Table 5.6 JVMDI Code Coverage Analyzer Class Statistical Result

	NoM	LCOM	AVCC	NOS	HBUG	HEFF	UWCS	INST	PACK	RFC	CBO	MI	CCML	NLOC
NoM	1.00													
LCOM	.410	1.00												
AVCC	.359	.300	1.00											
NOS	.975	.500	.400	1.00										
HBUG	.975	.500	.400	1.00	1.00									
HEFF	.872	.300	.700	.900	.900	1.00								
UWCS	.975	.300	.300	.900	.900	.800	1.00							
INST	.872	.100	.600	.800	.800	.900	.900	1.00						
PACK	-.544	-.354	-.354	-.707	-.707	-.707	-.354	-.354	1.00					
RFC	.975	.300	.300	.900	.900	.800	1.00	.900	-.354	1.00				
CBO	.544	.354	.000	.354	.354	.354	.707	.707	.250	.707	1.00			
MI	.410	.600	.100	.300	.300	.100	.500	.300	.354	.500	.354	1.00		
CCML	.289	-.051	.359	.103	.103	.205	.462	.564	.544	.462	.725	.667	1.00	
NLOC	.975	.500	.400	1.00	1.00	.900	.900	.800	-.707	.900	-.354	.300	.103	1.00

Table 5.7 JVMDI Code Coverage Analyzer Class Correlation Result

	N	RANGE	MIN	MAX	MEAN	STD. DEVIATION	VARIANCE	SKEWNESS	KURTOSIS
COMP	55.00	9.00	1.00	10.00	2.1818	1.775	3.152	2.579	7.853
NOCL	55.00	19.00	0.00	19.00	2.2182	3.353	11.248	2.884	11.243
NOS	55.00	27.00	1.00	28.00	6.600	6.184	38.244	1.643	2.273
HLTH	55.00	218.00	7.00	275.00	47.163	47.913	2295.69	1.997	4.141
HVOC	55.00	41.00	7.00	48.00	20.618	10.955	120.01	.995	.120
HEFF	55.00	56917.85	29.48	56947.33	4136.920	9939.06	9.879E7	4.053	17.74
HBUG	55.00	0.40	0.01	0.41	0.0755	0.0891	0.008	2.096	4.463
CREF	55.00	6.00	0.00	6.00	1.1818	1.2923	1.670	1.678	3.482
XMET	55.00	8.00	0.00	8.00	1.4000	1.6514	2.726	1.477	3.142
LMET	55.00	3.00	0.00	3.00	0.3273	0.6681	0.446	2.216	4.804
NLOC	55.00	42.00	1.00	43.00	11.272	10.1511	103.054	1.868	3.041

Table 5.8 JVMDI Code Coverage Analyzer Method Statistical Result

	COMP	NOCL	NOS	HLTH	HVOC	HEFF	HBUG	CREF	XMET	LMET	NLOC
COMP	1.00										
NOCL	.256	1.00									
NOS	.784	.190	1.00								
HLTH	.821	.097	.923	1.00							
HVOC	.794	.084	.896	.979	1.00						
HEFF	.837	.053	.889	.974	.951	1.00					
HBUG	.817	.112	.923	.990	.970	.962	1.00				
CREF	.409	.245	.442	.523	.534	.453	.554	1.00			
XMET	.563	.411	.618	.609	.560	.568	.604	.648	1.00		
LMET	.117	.126	.200	.204	.199	.140	.210	.118	.086	1.00	
NLOC	.821	.204	.955	.940	.901	.905	.951	.487	.644	.236	1.00

Table 5.9 JVMDI Code Coverage Analyzer Method Correlation Result

	N	RANGE	MIN	MAX	MEAN	STD. DEVIATION	VARIANCE	SKEWNESS	KURTOSIS
NOM	5.00	6.00	1.00	7.00	4.00	2.1213	4.500	.000	2.000
LCOM	5.00	.12	.00	.12	.024	.0563	.003	2.236	5.000
AVCC	5.00	.75	1.00	1.75	1.4080	.28030	.079	-.530	.449
NOS	5.00	10.00	12.00	22.00	15.60	4.615	21.30	.808	-1.958
HBUG	5.00	.13	.11	.24	.1620	.05975	.004	.573	-2.392
HEFF	5.00	8608.58	1656.99	10265.57	4258.91	3545.844	1.257E7	1.696	3.002
UWCS	5.00	10.00	1.00	11.00	4.800	3.701	13.700	1.495	3.238
INST	5.00	4.00	0.00	4.00	.8000	1.7888	3.200	2.236	5.000
PACK	5.00	3.00	.00	3.00	1.200	1.095	1.200	1.293	2.917
RFC	5.00	13.00	5.00	18.00	8.800	5.761	33.200	1.362	.860
CBO	5.00	3.00	.00	3.00	1.200	1.095	1.200	1.293	2.917
MI	5.00	51.38	81.35	132.73	115.194	23.723	562.82	-.902	-1.600
CCML	5.00	32.00	15.00	47.00	23.800	13.084	171.20	2.134	4.679
NLOC	5.00	11.00	22.00	33.00	25.800	4.266	18.20	1.646	3.007

Table 5.10 Nat Class Statistical Result

	NoM	LCOM	AVCC	NOS	HBUG	HEFF	UWCS	INST	PACK	RFC	CBO	MI	CCML	NLOC
NoM	1.00													
LCOM	.791	1.00												
AVCC	.229	.363	1.00											
NOS	.229	.725	.400	1.00										
HBUG	-.229	.363	-.684	.895	1.00									
HEFF	-.224	.354	-.667	.872	.975	1.00								
UWCS	1.00	.792	.229	.229	-.229	-.224	1.00							
INST	.791	1.00	-.363	.725	.363	.354	.791	1.00						
PACK	-1.00	-.791	-.229	-.229	.229	.224	1.00	-.791	1.00					
RFC	-.250	.395	-.918	.803	.918	.894	-.250	.395	.250	1.00				
CBO	-1.00	-.791	-.229	-.229	.229	.224	-1.00	-.791	1.00	.250	1.00			
MI	.224	-.354	.667	-.872	-.975	-1.00	.224	-.354	-.224	-.894	-.224	1.00		
CCML	1.00	.791	.229	.229	-.229	-.224	1.00	.791	-1.00	-.250	-1.00	.224	1.00	
NLOC	.918	.725	.368	.368	-.053	-.051	.918	.725	-.918	-.229	-.918	.051	.918	1.00

Table 5.11 Nat Class Correlation Result

	N	RANGE	MIN	MAX	MEAN	STD. DEVIATION	VARIANCE	SKEWNESS	KURTOSIS
COMP	20.00	2.00	1.00	3.00	1.450	0.6863	0.471	1.288	0.542
NOCL	20.00	7.00	3.00	10.00	4.250	2.099	4.408	1.718	2.118
NOS	20.00	17.00	1.00	18.00	3.450	3.734	13.94	3.440	13.183
HLTH	20.00	141.00	4.00	145.00	26.300	31.310	980.32	3.215	11.595
HVOC	20.00	28.00	4.00	32.00	15.450	7.423	55.103	0.799	0.197
HEFF	20.00	10241.57	12.00	10253.57	1040.16	2313.417	5351898.805	3.713	14.702
HBUG	20.00	0.24	0.00	0.24	0.0380	0.0533	0.003	3.209	11.500
CREF	20.00	18.00	0.00	18.00	2.6500	3.84263	14.766	3.661	14.920
XMET	20.00	17.00	0.00	17.00	1.350	3.7595	14.134	4.193	18.168
LMET	20.00	1.00	0.00	1.00	0.250	0.4444	0.197	1.251	-0.497
NLOC	20.00	18.00	2.00	20.00	5.600	4.52362	20.463	2.080	4.461

Table 5.12 Nat Method Statistical Result

	COMP	NOCL	NOS	HLTH	HVOC	HEFF	HBUG	CREF	XMET	LMET	NLOC
COMP	1.00										
NOCL	-.213	1.00									
NOS	.559	.273	1.00								
HLTH	.722	.076	.814	1.00							
HVOC	.722	.036	.814	1.00	1.00						
HEFF	.722	.046	.814	1.00	1.00	1.00					
HBUG	.709	.171	.828	.984	.981	.984	1.00				
CREF	.285	.204	.755	.726	.727	.726	.725	1.00			
XMET	.244	.185	.719	.650	.658	.650	.661	.736	1.00		
LMET	.226	.059	-.24	.191	.181	.191	.225	-.323	-.304	1.00	
NLOC	.671	-.054	.847	.788	.788	.788	.733	.497	.410	-.084	1.00

Table 5.13 Nat Method Correlation Result

	N	RANGE	MIN	MAX	MEAN	STD. DEVIATION	VARIANCE	SKEWNESS	KURTOSIS
NOM	6.00	16.00	1.00	17.00	6.167	6.9689	48.567	1.095	-1.035
LCOM	6.00	1.00	.00	1.00	.1917	.3997	.158	2.403	5.818
AVCC	6.00	1.35	1.00	2.35	1.416	.5105	.261	1.466	2.242
NOS	6.00	126.00	4.00	130.00	31.833	49.58	2458.56	2.160	4.712
HBUG	6.00	1.47	.02	1.49	.3267	.5804	.337	2.268	5.218
HEFF	6.00	77218.20	177.12	77395.32	14172.86	31.025	9.626E8	2.431	5.928
UWCS	6.00	23.00	2.00	25.00	9.333	11.021	121.46	.995	-1.683
INST	6.00	9.00	0.00	9.00	3.1667	4.167	17.367	.943	-1.727
PACK	6.00	17.00	.00	17.00	7.50	8.04363	64.700	.431	-2.399
RFC	6.00	57.00	1.00	58.00	14.00	22.244	494.80	2.152	4.678
CBO	6.00	3.00	.00	3.00	1.200	1.095	1.200	1.293	2.917
MI	6.00	111.12	68.18	179.30	132.97	38.29	1466.31	-.799	1.226
CCML	6.00	127.00	0.00	127.00	28.00	50.259	2526.00	2.116	4.512
NLOC	6.00	171.00	5.00	176.00	42.333	67.14	4507.86	2.208	4.937

Table 5.14 Quilt Class Statistical Result

	NoM	LCOM	AVCC	NOS	HBUG	HEFF	UWCS	INST	PACK	RFC	CBO	MI	CCML	NLOC
NoM	1.00													
LCOM	-.548	1.00												
AVCC	.657	.657	1.00											
NOS	-.941	.759	.618	1.00										
HBUG	.955	.770	.716	.986	1.00									
HEFF	.941	.759	.794	.943	.986	1.00								
UWCS	.871	.806	.657	.941	.955	.941	1.00							
INST	.594	.594	.273	.677	.687	.677	.844	1.00						
PACK	.563	.938	.500	.794	.761	.706	.844	.682	1.00					
RFC	.941	.516	.441	.943	.899	.829	.820	.559	.618	1.00				
CBO	.544	.354	.000	.354	.354	.354	.707	.707	.250	.707	1.00			
MI	-.091	-.516	-.441	-.257	-.232	-.200	-.334	-.088	-.618	-.143	.354	1.00		
CCML	.871	.290	.657	.698	.770	.820	.742	.594	.281	.698	.725	.030	1.00	
NLOC	.941	.759	.618	1.00	.986	.943	.941	.677	.794	.943	-.354	-.257	.698	1.00

Table 5.15 Quilt Class Correlation Result

	N	RANGE	MIN	MAX	MEAN	STD. DEVIATION	VARIANCE	SKEWNESS	KURTOSIS
COMP	37.00	9.00	1.00	10.00	1.7297	1.6774	2.814	3.962	17.541
NOCL	37.00	46.00	0.00	46.00	3.5135	7.8832	62.146	4.721	24.726
NOS	37.00	51.00	1.00	52.00	4.4595	8.6621	75.033	4.971	26.783
HLTH	37.00	348.00	6.00	354.00	29.189	59.012	3482.435	5.019	27.134
HVOC	37.00	67.00	6.00	73.00	14.513	12.095	146.312	3.746	16.060
HEFF	37.00	58179.29	29.48	58208.77	2194.735	9635.8875	9.285E7	5.782	34.233
HBUG	37.00	0.72	0.01	0.73	0.0465	0.12241	0.015	5.212	28.847
CREF	37.00	8.00	0.00	8.00	1.6757	1.7488	3.059	2.736	8.207
XMET	37.00	21.00	0.00	21.00	1.3784	3.60055	12.964	4.846	25.899
LMET	37.00	3.00	0.00	3.00	0.1622	0.55345	0.306	4.240	19.945
NLOC	37.00	71.00	1.00	72.00	6.0270	11.9756	143.416	5.042	27.280

Table 5.16 Quilt Method Statistical Result

	COMP	NOCL	NOS	HLTH	HVOC	HEFF	HBUG	CREF	XMET	LMET	NLOC
COMP	1.00										
NOCL	.211	1.00									
NOS	.848	.067	1.00								
HLTH	.834	.056	.943	1.00							
HVOC	.841	.045	.920	.984	1.00						
HEFF	.824	-.001	.962	.978	.963	1.00					
HBUG	.877	.216	.874	.918	.906	.896	1.00				
CREF	.271	.403	.225	.414	.395	.308	.471	1.00			
XMET	.795	.143	.848	.861	.840	.806	.806	.338	1.00		
LMET	-.009	.00	.164	.195	.100	.183	.186	.382	.083	1.00	
NLOC	.883	.088	.992	.952	.927	.958	.876	.220	.843	.135	1.00

Table 5.17 Quilt Method Correlation Result

5.2 Graphical Results

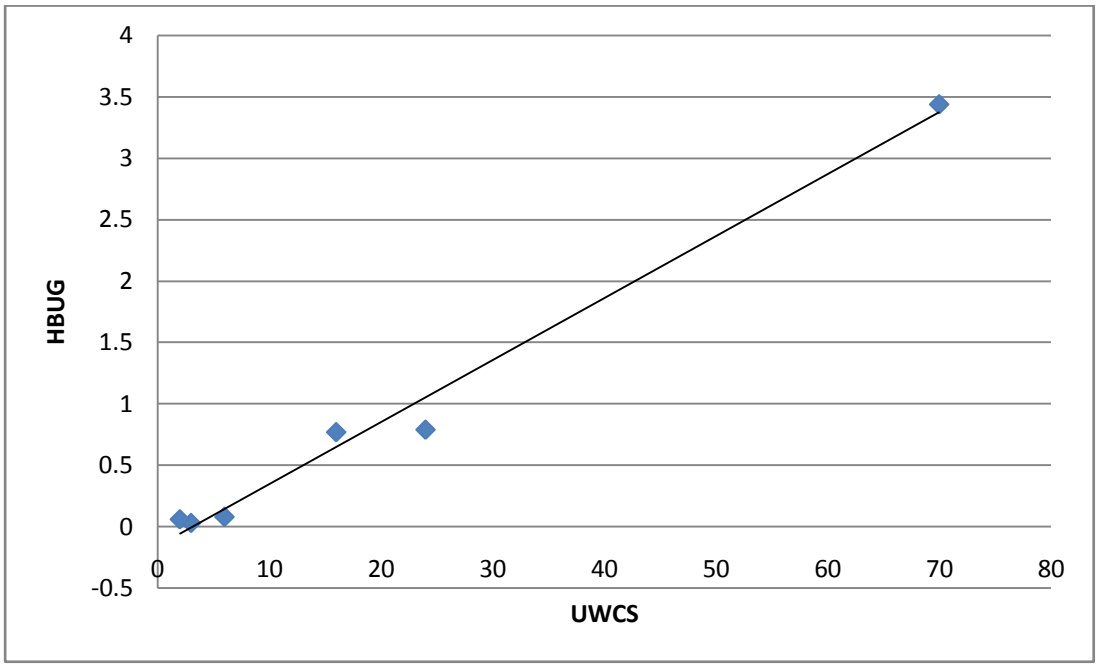


Figure 5.1 Relationship between HBUG and UWCS

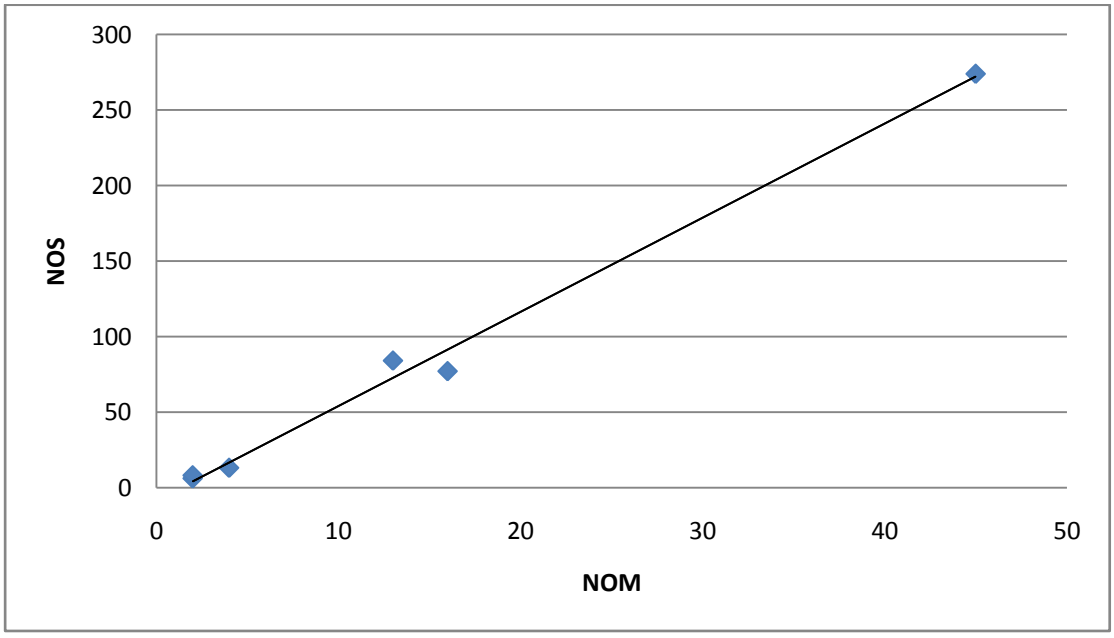


Figure 5.2 Relationship between NOM and NOS

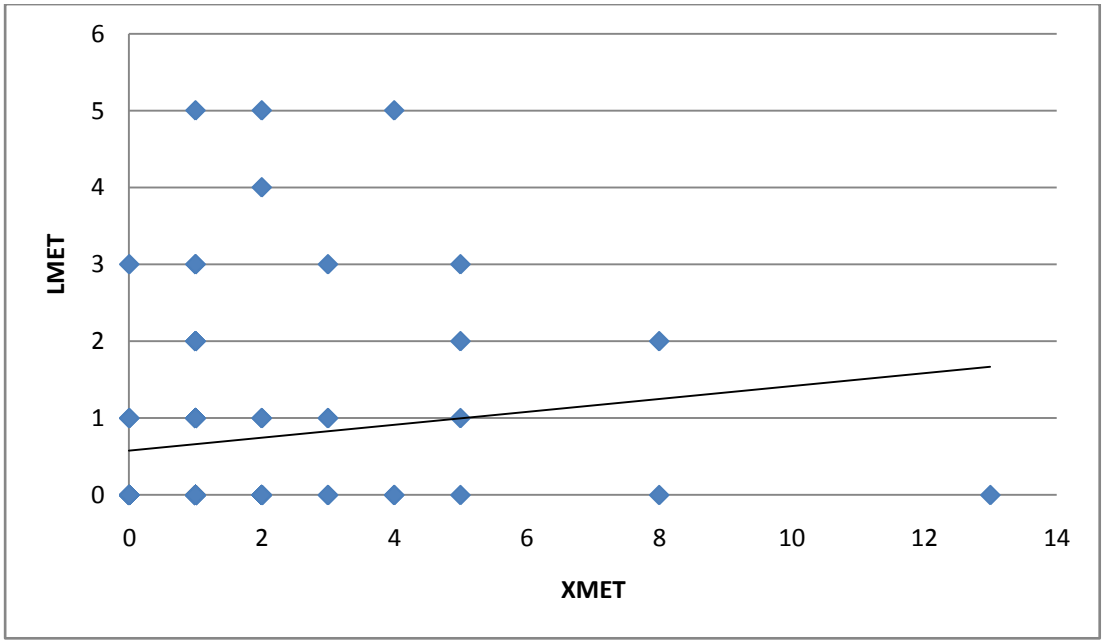


Figure 5.3 Relationship between XMET and LMET

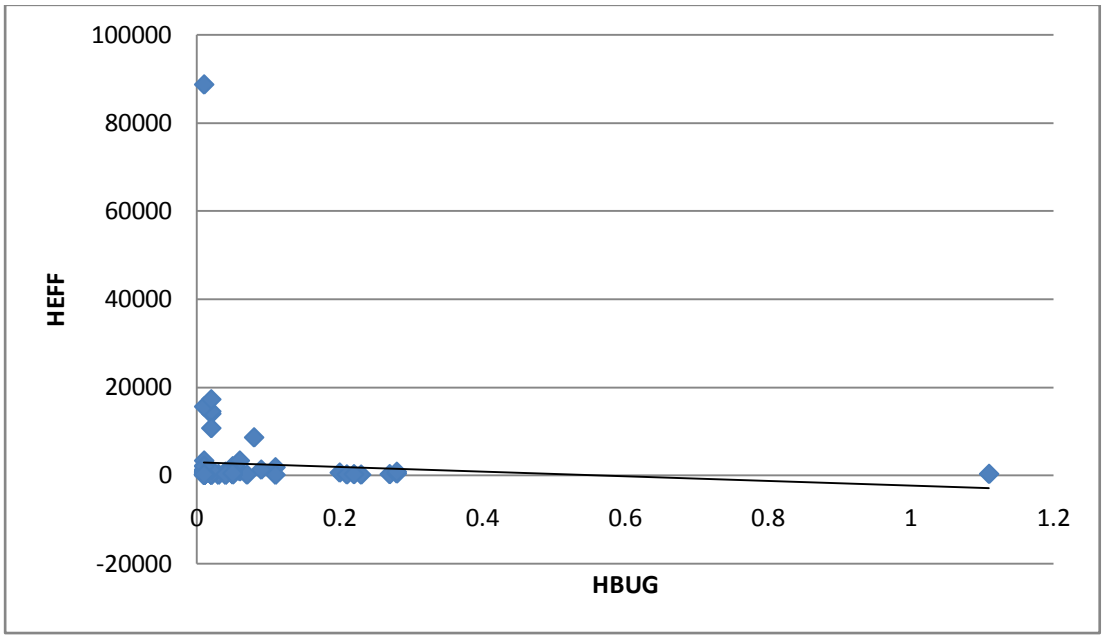


Figure 5.4 Relationship between HBUG and HEFF

These above result tables and graphs which shows the statistical value of the metrics for OSS represent the minimum value, maximum value, mean of the values, standard deviation, variance, skewness and kurtosis of the values obtained with the help of JHawk tool whereas Correlation tables shows the value of Pearson correlation coefficient.

We computed Pearson's correlation and the statistical significance of correlation. Presented metrics are significant at r value < 0.001 using correlation analysis i.e. associated metrics are correlated with each other in some manner. We found that relationship between metrics should be studied at both class and method level.

Starting with the results of class level, we first noticed the fair positive correlation between the NOM and LCOM because their value lies between 0.41 and 0.791 in all cases of input.

NOM and AVCC also correlate fairly because we can see its value between 0.22 and 0.82.

NOS, NLOC and CCML are strongly correlated with each other. AVCC and RFC are strongly correlated with each other. Correlation among NOS, NLOC, CCML and AVCC, RFC indicates the good level of modularity which is a indication of a good design of code.

HBUG and HEFF is strongly correlated with NOS, UWCS, INST and RFC whereas in some cases it is slightly negative correlated with MI, CCML and NLOC.

INST is strongly correlated with NOM, HBUG, HEFF and CCML whereas RFC is correlated with AVCC, HBUG and HEFF.

Now moving to metrics at method level, we noticed the strong correlation of complexity with HLTH, HBUG, HVOC, NOS and HEFF. There is strong correlation between all metrics of Halstead's like HBUG is positively correlated with HVOC, HLTH and HEFF, HEFF is positively related to HBUG, HVOC and HLTH and so on.

There is interesting fact noticed at method level and it is XMET is closely related to all other metrics as compared to LMET in case of all inputs but LMET is also highly positive correlated with other metrics.

We also noticed the very low correlation of NOCL with COMP and HLTH. HVOC is also very less correlated with LMET and NOCL. CREF is correlated to NOCL with very low values and same is the case with XMET and LMET. In case of XMET and LMET we have also seen negative correlation. This low correlation can help us in predicting defects and faults in the code.

Graphical representation helps understanding relationship only by means of visualizing. These graphs are made from the value of metrics obtained using JHawk and this representation also depicts the same relation between two metrics which we have obtained from the Pearson's correlation coefficient. All the graphical representation show positive correlation between metrics but figure 5.3 shows the low positive correlation between XMET and LMET.

Tables with statistical results gives the idea of values present in the data on which all the process of evaluation is carried on. Skewness is a measure of symmetry, or more precisely, the lack of symmetry. A distribution, or data set, is symmetric if it looks the same to the left and right of the centre point. Kurtosis is a parameter that describes the shape of a random variable's probability distribution. We have not used these distributions in our approach directly but the results of these help in making decision about some facts.

Combining approach of studying statistical and graphical results helps to find the precise kind of relationship between any two metrics.

Chapter 6

CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

Software metrics have a very important role in software development. An intelligent selection of metrics for OSS can result in improved software. In this thesis exploratory study is made on the software metrics and discussion has also been made on the open source outlook. A direct evaluation approach is selected where various metrics for four OSS is studied. These softwares are downloaded from SOURCEFORGE.NET and Java-Source.net. The size of these projects is small. Selection of softwares is based on number of downloads. There are some cautions that should be considered when they are put to use these results in a wide scope. These discoveries may be applicable to the small sized projects. The application of the results for other object oriented programming such as C++ may require some changes and adjustments. Their properties are analyzed using Pearson r product moment correlation, graphs and statistical results. There are some interesting findings from the analysis. There are pair of metrics with poor correlation among them and they are XMET and LMET, HVOC and LMET, CREF and NOCL and so on these values can help us in predicting defects and faults in the code where as the strongly correlated metrics pair like RFC and AVCC, NOS and NLOC, NOM and LCOM, AVCC and NOM and some others helps in predicting modularity of the software.

6.2 Future Work

This work can be used as a guideline of OSS developers in developing their projects so that chance of success of their project is higher. The future work can involve conducting additional empirical studies with data from software projects which is of large size.

Since the existing metrics are useful for experienced developers there is a need of simple metrics based on the publicly available data for the users of OSS which help them to take adoption decision. This work can be first step in this direction.

In this work we analysed the value of metrics with the tool which is a static analysis tool we can also combine the dynamic tool which works on the value of metrics at run time. This approach of combining dynamic analysis tool with the static analysis of code can result in a better precision of result.

References

[1] A. Brown and G. Booch, "Revising Open Source Software and Practices: The Impact of Open Source on Commercial Vendors", Software Reuse: Methods, Techniques, and Tools, Proceedings, Vol. 2319, pp. 123-136, 2002.

[2] M. Elliott, M.S. Ackerman, W. Scacchi, "Knowledge Work Artifacts: Kernel Cousins for Free/Open Source Software Development", Proceedings of the Intenational ACM Conference on Supporting Group Work, November 2007.

[3] A. Beard and H. Kim, "A Survey on Open Source Software Licenses ", Journal of Computing Sciences in Colleges, Vol. 22, No. 4, April 2007.

[4] R. Roets, M. Minnar, and K. Wright, "Towards Successful Systems Development Projects in Developing Countries", Proceedings of the Ninth International Conference on Social Implications of Computers in Developing Countries, Sao Paulo, Brazil, May 2010.

[5] G. D. Hoffman, "Early Introduction of Software Metrics", Proceedings of the IEEE Conference, pp. 559-563, 1989.

[6] A. Boughton, "Software Metrics", www.cs.colorado.edu /~Kena/ classes/5828/s12 / presentation-materials / boughtonalexandra.pdf.

[7] "JHawk", http://www.virtualmachinery.com/jhawkprod.htm.

[8] "JHawk Licensing", http://www.virtualmachinery.com/jhawklicensing.htm.

[9] "SOURCEFORGE.NET", http://www.quilt.sourceforge.net.

[10] "Java-Source.net" ,http://www.java-source.net/ open-source/ code coverage / quilt.

[11] "Java-Source.net",http://www.java-source.net/open-source/code-coverage /emma.

[12] "Java-Source.net",http://www.java-source.net/open-source/code- coverage/nat.

[13] "Java-Source.net",http://www.java-source.net/open-source/code-coverage/jvmdi-code-coverage- analyzer.

[14] A. W. R. Emanuel, R. Wardoyo, J. E. Istiyanto, and K. Mustofa, "Statistical Analysis on Software Metrics Affecting Modularity in Open Source Software", IJCSIT, Vol. 3, No. 3, pp. 105-118, June 2011.

[15] Dr. Dravid and A. Wheeler, "Open Source Software (OSS or FLOSS) Basics: An Introduction", Institute of Defence Analyses, July 2010.

[16] A. Brenstein, "Introduction to Open Source Software: How it Works, Why its Free, and How it Might Fit the Needs of Nonprofits", Tech Underground and Electric Embers Zachary Mutrux, CompuMentor, May 2004.

[17] C. Gacek and B. Arief , “The Many Meaning of Open Source”, Software, IEEE, Vol. 21, pp. 39-40, 2004.

[18] A. Khanjani and R. Sulaiman, “The Process of Quality Assurance under Open Source Software Development”, Proceedings of the IEEE Symposium on Computers and Informatics, pp. 548-552, 2011.

[19] E. E. Mills, “Software Metrics”, SEI Curriculum Module SEI-CM-12-1.1, December 1988.

[20] A. Vogelsang, A. Fehnker, R. Huuck, and W. Reif, “Software Metrics in Static Program Analysis”, Australian Research Council Journal, 2010.

[21] G. Ramesh, Managing Global Software Projects, Tata McGraw Hill, 2005.

[22] Linda Westfall, “12 Steps to Useful Software Metrics”, The Westfall Team, 2010.

[23] P. Singh and H. Singh, “DynaMetrics: A Runtime Metric-Based Analysis Tool for Object-Oriented Software Systems ”, SIGSOFT, Software Engineering Notes, Vol. 33, pp. 1-6, November 2008.

[24] McCabe T., “A Complexity Measure”, IEEE Transactions on Software Engineering, Vol. 2, No. 4, pp. 308-320, December 1976.

[25] K. Kaur, K. Minhas, N. Mehan, and N. Kakkar, “Static and Dynamic Complexity Analysis of Software Metrics”, World Academy of Science, Engineering and Technology 32, pp. 159-161, 2009.

[26] M. Scotto, A. Sillitti, G. Succi, and T. Vernazza, “Dealing with Software Metrics Collection and Analysis- A Relational Approach”, Studia Informatica Universalias, pp. 343-366, 2004.

[27] T. Kuipers and J. Visser, “Maintainability Index Revisited”, IEEE Transactions on Software Engineering, Vol. 2, No. 5, pp. 308-311, December 1995.

[28] J. L. Rodgers and W. A Nicewander, “Thirteen Ways to Look at Correlation Coefficient”, The American Staticians, Vol. 42, No. 1, pp. 59-66, February 1988.

[29] E. Kijsipongse and S. U. Ruekolan, C. Ngawphiw, and S. Tongsima, “Efficient Large Pearson Correlation Matrix Computing Using Hybrid MPI/CUDA”, Proceedings of the Eight International Joint Conference on Computer Science and Software Engineering, pp. 237-241, 2011.

[30] S. Batra, “Measuring Maintainablity of Open Source Software Using Metrics”, M.Tech Thesis, Thapar University, Computer Science and Applications, July 2012

[31] “JHawk”, http://java-source.net/open-source/testing-tools/j-hawk.