Object-Oriented Complexity Metrics for the Java Programming Language

Submitted by

Hari Narayanan

In partial fulfillment of the requirements for the Master of Science Degree

under the supervision of

Dr. Curt Cook
Department of Computer Science
Oregon State University

Abstract

Since its introduction in 1995 by Sun Microsystems, the Java programming language has been widely accepted by the software development community. Besides being a natural fit for Internet and World Wide Web (WWW) based applications, Java is also being used in other diverse application areas due to its simplicity, reduced learning-curve, portability, and Object-Oriented features. Given this tremendous potential for Java as a development language, there is a pressing need for software measures or metrics with which to manage the process of software development in Java. A good set of metrics can be very useful especially in the post-coding phases of the software life-cycle, such as testing and maintenance, in identifying those classes which are likely to be hard to test or modify. This research addresses these needs through an investigation into an Object-Oriented metrics suite for Java. A set of metrics at the class-level is proposed, and a tool was developed to automate the collection of the metrics. Two experiments were conducted to determine which of the metrics were effective and useful measures of complexity. The experiments indicate that the number of non-static external references is a good complexity metric for Java.

1.0 Introduction

Software development is a multi-faceted process that is yet to be completely understood and is very difficult to manage. This difficulty in managing software development arises out of the complexity of the application being developed, the ability of the personnel involved, the characteristics of the computer system on which the software system is to be installed, and the nature of the programming language or tool used to develop the software product. These difficulties have at least in part contributed to what has been termed as a *software crisis*, wherein software costs exceeding the estimate, unreliable and buggy software products and schedule slippage are the rule rather than the exception. To cope with this crisis, it has been proposed that the proper use of software metrics, measurement and models be used in the successful management of software development and maintenance [14] to provide greater visibility of these processes. Software metrics are used to characterize the essential features of software quantitatively, so that classification, comparison and mathematical analysis can be applied. Judicious use of these metrics can be a great aid to management in achieving management goals. Java is a programming language developed and marketed by Sun Microsystems, Inc. Java is a new language for network programming: object-oriented; it is secure and portable, which enables users to write new applications for the Internet and Intranets. Java is platform-independent, and comes bundled with a rich set of GUI, networking and other utility classes. For many, Java is known primarily as a tool to create *applets* for the World Wide Web. "Applet" is the term Java uses for a mini-application that runs inside a web page. An applet can perform tasks and interact with the user on their browser page without using resources from the web server after being downloaded. Apart from its obvious value to distributed network environments like the Web, Java is also a powerful general-purpose programming language suitable for building a variety of applications that may or may not necessarily depend on network features. There are also other groups that use Java as a general-purpose programming language where Java's ease of programming and safety features help produce debugged code quickly.

Java has gained broad acceptance by the software industry since its introduction, and is

widely considered a promising programming language for the future to develop secure and portable cross-platform applications for the Internet and the Web, as well as general purpose applications. Many leading software companies such as Apple Computers, Microsoft, Oracle etc. have licensed the Java technology from Sun Microsystems, thus recognizing its importance and potential to the software industry. Many software systems in diverse application areas are currently being developed in Java, and it is clear that Java is poised to be a very important programming language in the future of software development.

Given this tremendous potential for software development using Java, it is very important to develop software metrics that will provide measures of the complexity of Java programs to better manage the process of software development in Java. Software complexity has often be defined as "a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software "[14]. In the context of performing maintenance activities on a software product, "complexity" would be the resources expended in performing testing or modification tasks on the software. One large of class of complexity metrics are those which are measures or combination of measures of software attributes. The idea behind these metrics is that the degree of occurrence of these attributes is related to the complexity or difficulty of performing programming tasks, such as maintenance or testing. Java as a programming language is so new that many questions about programming techniques or standards and their influence on program testing, modification, or understanding are have not been studied. Further investigation of these questions is necessary for the sustained success and acceptance of the language in the software development community in the future. The availability of a set of validated complexity metrics for the language will be very useful in making decisions regarding resource allocation, during the testing and maintenance stages of a software project lifecycle. This research addresses this need, and investigates appropriate software metrics for the Java programming language. These metrics are based on object-oriented program attributes, such as inheritance, coupling etc. and are intended to provide measures of performing programming tasks such as maintenance or testing. We have studied and proposed a suite of complexity metrics for

Java which consist of metrics based on existing Object-Oriented metrics as well as new ones. We have also conducted two experiments to determine which of these metrics are useful indicators of complexity. We have proposed a suite of software complexity metrics for Java and developed a source code analyzer tool that computes the proposed metrics.

2.0 Background

With the increasing adoption of the Object-Oriented programming paradigm as the dominant software development methodology in the past few years, there has been much interest in developing with ways to measures the complexity of Object-Oriented software systems. Software complexity is an area of software engineering concerned with the measurement of factors that affect the cost of developing and maintaining software. "Complexity" is a much overloaded term, and is used so often in so many different contexts in software research that it may be useful to discuss its various connotations. In theoretical contexts, it is common to classify algorithms as to their computational complexity, which refers to the efficiency of the algorithm in its use of machine resources. On the other hand, the perceived complexity of software is often called *psychological complexity* because it is concerned with those characteristics of the software that affect programmer performance in composing, comprehending, and modifying the software. Curtis [14] has suggested a definition that encompasses both types of complexity:

"Complexity is a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software."
This definition implies that complexity is a function of both the software itself and its interactions with other systems including humans. Measurements of software characteristics can be useful throughout the software life cycle. Software metrics are often classified as either *process* metrics or *product* metrics, and are applied to either the development process or the software product developed. In this paper, we will be dealing with product metrics, which are based on the attributes of the software product as opposed to process metrics which quantify attributes of the development process and of the development environment.

2.1 Traditional software complexity metrics

Several metrics have been proposed and used in conjunction with the Procedural programming languages such as C, Fortran, Pascal, COBOL etc. They may be categorized as follows:

Size Metrics: These metrics measure the size characteristics of a program. These metrics are based on the intuition that the "larger" a program, the more complex it is. Typical size metrics include lines of code, function count, token count etc.

Data Structure Metrics: These metrics capture the amount of data input to, processed in, and output from software. These usually include the Amount of Data, the Usage of Data within a Module, the Sharing of Data among Modules etc.

Logic Structure Metrics: These metrics are based on the logic structure or control flow of a program. They are based on the belief that the more the number of different execution paths in a program, the more complex the program. They include Decision Count, Minimum Number of Paths and Reachability Metrics, Nesting Levels, Transfer Usage, Cyclomatic Complexity etc. Of these, probably the best known metric is the Cyclomatic Complexity number proposed by McCabe [21]. This metric was originally designed to measure the number of "linearly independent" paths through a program, which in turn is believed to relate to the testability and maintainability of the program. The Cyclomatic Complexity of a program is defined as the cyclomatic number of its control graph. The nodes of a program control graph represent the statements or basic blocks in the program, and the edges represent the flows of control between the nodes. For a program control graph with e edges and n nodes, cyclomatic complexity $V(G)$ is given by:

$V(G) = e - n + 2.$

Composite Metrics: Composite metrics are based on the premise that software complexity has several dimensions, and can be represented better as a composite of several metrics. These metrics try to assess or provide a measure of the complexity of software by compositing several different metrics (such as the ones described earlier).

The most significant composite metrics were proposed by Halstead, and are known as the *Software Science Composite Metrics*. The Software Science Complexity Metrics are based on counts of four basic *tokens*, which are basic syntactic units distinguishable by a compiler. A computer program is considered in Software Science to a collection of tokens that can be classified as either operators or operands. All Software science metrics are functions of the counts of these tokens. The basic metrics are defined as:

n1 = number of unique operators

n2 = number of unique operands

N1 = total occurrences of operators

N2 = total occurrences of operands

Generally, any symbol or keyword in a program that specifies an action is considered an operator, while a symbol used to represent data is considered an operand. Based on the above four basic counts, several composite metrics such as the Estimated Program Length, the Program Volume, Potential Volume and difficulty etc.

Another related metric is called *function points*, which defines productivity in terms of a weighted sum of delivered functional units. Functional units are defined as the number of inputs, the number of outputs, the number of inquiries, and the number of files. This approach works well in commercial applications in which the functional units are mor or less clearly definable and fairly homogeneous. However, for system programs such as compilers and compilers and for other program types, function units are more difficult to define precisely.

2.2 Extension of traditional metrics to the Object-Oriented paradigm

Quite a few researchers have proposed extensions of traditional software complexity such as the above metrics to object-oriented programs. In this section, we describe some of the work that has been done in extending traditional complexity metrics to the Object-Oriented paradigm.

D. Tegarden and S. Sheetz [1] investigated traditional software metrics such as lines of code, software science, cyclomatic complexity etc. as possible indicators of complexity of object-oriented systems. They also investigated the effects of polymorphism and inheritance on the complexity of object-oriented systems as measured by the traditional metrics. They concluded that traditional metrics are applicable to the measurement of the complexity of object-oriented systems. However, they also stated in their conclusions that additional metrics are needed to fully measure all aspects of OO systems. They suggested that these metrics should include those based on the complexiy of the messages being passed.

C. Coppick and T. Cheatham [2] studied the extension of Halstead's Software Sciences metrics and McCabes's Cyclomatic Complexity metric to objects. They concluded that software metrics can and should be applied within the Object-Oriented paradigm, and that their application of Halstead's software science and McCabe's cyclomatic complexity to objects produced intuitively reasonable results. They also suggested a limit for the cyclomatic complexity of an object.

M.Hamza and B. Lees [3] studied the applicability of traditional metrics to object-oriented software using Quality Function Deployment (QFD) and Case Based Reasoning (CBR). They proposed a model to explore the assurance of quality through a combination of QFD and BDR methods, and thus established a relationship between software quality metrics and the software quality criteria of QFD.

2.3 Complexity metrics for Object-Oriented software

Many research projects investigating complexity metrics for OO software have pointed out that traditional metrics do not capture many aspects of the complexity of object-oriented software. Specifically, since traditional procedural programming metrics concentrate on either data structures or functions separately, they fail to capture the complexity in dealing with *objects and classes*, that encapsulate both data structures and algorithms that operate on those data structures. Hence, many researchers have proposed

new metrics that would be better indicators of the complexity of object-oriented software. This section describes some of the significant work that has been done in that area.

2.3.1 The Chidamber-Kemerer Object-Oriented Metrics Suite:

S. Chidamber and C. Kemerer [3] proposed a suite of six metrics for measuring complexity of object-oriented systems. This metric suite is probably the best-known among object-oriented metrics, and has a generated a lot of interest in the metrics community. The metrics they proposed are:

Weighted Methods per Class (WMC): Chidamber and Kemerer defined WMC of a class C_1 with methods M1, M_2 etc. and their corresponding complexities c1, c2 etc. as

WMC = sum of all c_is, where i = 1, 2...n.

If all method complexities are considered to be unity, then WMC = the number of methods.

The rationale for this metric may be stated as follows:

- The greater the number of methods in a class, the greater the potential impact on children, since children will inherit all the methods defined in the class.
- Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

***Depth in Inheritance Tree (DIT)*:** DIT of a class is the depth of the class in its inheritance tree. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.

The intuitions behind this metric may be stated as follows:

- The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior.
- Deeper trees constitute greater design complexity, since more methods and classes are involved.
- The deeper a particular class is in the hierarchy, the greater the potential use of inherited methods.

Lack of Cohesion in Methods (LCOM): Consider a class C_1 with n methods M1, M2... Mn. Let {Ij} = set of instance variables used by method Mi. There are n such sets {I1}... {In}. Let P = {Ii, Ij} | Ii intersection Ij is empty} and Q = {(Ii, Ij) | Ii intersection Ij is not empty}

LCOM = |P| - |Q|, if |P| >|Q|

= 0 otherwise.

The intuitions behind this metric are as follows:

- 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 subclasses.
- Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Coupling Between Object Classes (CBO): CBO for a class is a count of the number of couples with other classes.

The rationale for this metric follow:

- Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the less its coupling, and the easier it is to reuse it in another application.
- In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult.
- A measure of coupling is useful to determine how complex the testing of various parts of a design is likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

Number of Children (NOC): NOC is the number of immediate sub-classes subordinated to a class in the class hierarchy.

The intuitions behind the metric are:

- The greater the number of children, the greater the reuse, since inheritance is a form of reuse.
- The greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of sub-classing.

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

RS = {M} $U_{all\ i}$ {Ri} where $\{R_i\}$ = set of methods called by method i and {M} = set of all methods in the class.

The intuitions behind the metrics are:

- If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.
- The larger the number of methods that can be invoked from a class, the greater the complexity of the class.
- A worst case value for possible responses will assist in appropriate allocation of testing time.

The authors proposed the six metrics on the premise that metrics must be both theoretically rigorous and practically useful. Each of the six metrics proposed were validated against the metrics evaluation criteria proposed by Weyuker [19]: Non-coarseness, non-uniqueness, monotonicity, non-equivalence of interaction, importance of design details, increase of complexity with interaction, permutation and granularity. We briefly describe some of these criteria here, the interested reader is urged to refer [19]. Non-coarseness is the ability of a metric to differentiate classes based on the value of a metric i.e. a non-coarse metric should be such that not every class will have the same value for the metric, otherwise it would have lost its value as a measure. Non-uniqueness refers to the property of a metric whereby there can exist two distinct classes such that their metric values can be equal. Monotonicity implies that the metric for a combination of two classes can never be less than the metric for either of the component classes.

Granularity requires that there be a finite number of cases where classes have the same metric value. The C-K metrics suite satisfies most of these criteria. These metrics are probably the best known Object-Oriented metrics, and several of the metrics that we propose in this paper have been adapted from this suite specifically for Java.

2.3.3 Other related work in the OO metrics area:

C. Cook and A. Lake [4] studied software complexity measures for C++ programs. Their metrics included numerous object-oriented metrics and also extensions of traditional metrics for C++. They also proposed an approach using factor analysis to reduce the large number of proposed OO metrics to a small number of the complexity domains by identifying collinear metrics.

F. Abreu and R. Carapuca [5] proposed the TAPROOT framework (TAxonomy PRecis for Object-Oriented meTrics) which has two axes, category and granularity. The categories are: design, size, complexity, reuse, productivity, and quality. The granularities are: methods, class and system.

J. Bieman and S. Karunanithi [6] studied a method for deriving candidate reuse metrics. They presented a set of measurable reuse attributes and a metrics suite which quantify these attributes for object-oriented systems. They also introduced the concept of perspectives for reuse.

C. Chung and M. Lee [7] proposed a graph-theoretic metric for measuring the complexity of class hierarchy. They argued that inheritance has a close relationship with object-oriented software complexity. They also presented an algorithm to support this software metric.

S. Henry and W. Li [8] investigated maintenance metrics for the OO paradigm. These metrics primarily targeted Ada design and source code. They built upon Chidamber's and Kemer's work and added a few metrics of their own.

R. Hudli and C. Hoskins [9] defined two kinds of metrics to evaluate the design and implementation of OO software systems. One kind is class-based, and evaluates the design of classes. The second kind measures the class design structure of the program.

T. Korston and J. McGregor [10] suggested a number of criteria for object-oriented class libraries. These criteria are categorised by their support for a set of desirable attributes. The attributes given are: completeness, consistence, ease of learning, ease of use , efficiency, extendability, integrability, intuitiveness, robustness and support.

Y. Lee and B. Liang [11] presented a set of complexity metrics for object-oriented systems based on information flow models and evaluated them using Weyuker's meta-metrics for their validity. The entities measured by this set of metrics consisted of methods, classes, class hierarchies, and programs in an object-oriented system.

H. Sneed [12] proposed a metric called object-points to express the size and complexity of object-oriented software. The main goal of this research was to propose an approach to accurately estimate the costs of developing object-oriented software.

M. Shumway [13] proposed a metric to measure class cohesion in object-oriented system written in Java. The measure counts the proportion of method pairs in a class exhibiting connectedness through the use of one or more common variables in that class.

Thus, all of the work described in this section propose metrics based on the object-oriented attributes of programs. Also, none of the described work attempts to determine the utility of the proposed metrics through experiments or case studies.

2.4 Empirical Studies, Metric Validation and Data Collection

In this section, we describe research that has been done in validating metrics for Object-Oriented programs, case studies in the development of large object-oriented programs, and comparisons of the object-oriented and traditional approaches to software development.

S.Henry, J. Lewis et al [15] conducted an empirical study of the Object-Oriented paradigm and software reuse. This work attempted to validate the claim that the Object-Oriented paradigm promotes software reuse. From the controlled experiment they conducted, the authors generally concluded that the Object-Oriented paradigm does promote software reuse.

S. Henry and M. Humphrey [16] conducted a controlled experiment to evaluate the maintainability of object-oriented software. This experiment compared the maintainability of two functionally equivalent systems, in order to explore the claim that systems developed with Object-Oriented languages are more easily maintained than those programmed with procedural languages. They found supporting evidence that programmers produce more maintainable code with an Object-Oriented language than with a standard procedural language.

J. Walsh [17] studied the data collected during the development of Rational Rose, a large (100 KLOC) program written in C++, and concluded that high product quality can be achieved during a telescoped development schedule through the use of an iterative-development methodology. The author studied the data on defect density and discovery rate gathered on one phase of Rose development and deduced there was a low error rate in code delivered for functional test. According to the author, the data showed that 80% of the defects are found in 20% of the code, and 80% of the defects are critical while 20% are non-critical. The author also reports that there is an association between errors detected during functional testing and the depth of a class in its subsystem hierarchy.

V. Basili et al [18] conducted a study to experimentally investigate the suite of Object-Oriented design metrics introduced by Chidamber and Kemerer. In order to do this, they assessed these metrics as predictors of fault-prone classes. They studied data from eight medium sized project developed in C++, and concluded from their study that the metrics are reasonable indicators of fault-proneness.

3.0 A metrics suite for object-oriented programs in Java

In this chapter, we present a set of object-oriented metrics for Java. Java is a relatively new language, but has already proven very popular in the software industry. It seems very possible now that the Java technology will be deployed in the development of many different kinds of software applications and systems in the future. Given this potential, it is very important to develop software metrics that can help the successful management of Java software development and maintenance. The prudent use of useful complexity metrics for Java can increase the visibility and understanding of the software development process, and can provide intelligent decision support in the allocation of resources to the testing and maintenance stages of the software life cycle.

3.1 Some background on the Java programming language

Java is a programming language developed and marketed by Sun Microsystems, Inc. Java is a new language for network programming: portable, secure and object-oriented, which enables users to write new applications for the Internet and intranets. Java is platform-independent, and comes bundled with a rich set of GUI, networking and other utility classes. For many users, Java is known primarily as a tool to create *applets* for the World Wide Web. "Applet" is the term Java uses for a mini-application that runs inside a web page. An applet can perform tasks and interact with the user on their browser page without using resources from the web server after being downloaded. Apart from its obvious value to distributed network environments like the Web, Java is also a powerful general-purpose programming language suitable for building a variety of applications that may or may not necessarily depend on network features. There are also other groups that use Java as a general-purpose programming language where Java's ease of programming and safety features help produce debugged code quickly.

Java has gained broad acceptance by the software industry since its introduction, and is widely considered a promising programming language for the future to develop secure

and portable cross-platform applications for the Internet and the Web, as well as general purpose applications. Many leading software companies such as Apple Computers, Microsoft, Oracle etc. have licensed the Java technology from Sun Microsystems, thus recognizing its importance and potential to the software industry.

The main reasons for this confidence in Java's potential are:

- Java's architecture neutrality, which makes it possible to write applications once and run them anywhere without recompilation.
- Availability of a robust exception-handling mechanism, built-in support for multithreading, garbage collection etc. facilitate quick and bug-free development of modern network based and graphical user interface-based applications.

Many software systems in diverse application areas are currently being developed in Java, and it is clear that Java is poised to be a very important programming language in the future of software development. The Sentry Group, an Information Systems consulting and market research company estimates that currently about 40% of the Fortune 1000 companies deploy Java in their application development, and this number is expected to double by the year 2000.

3.1.1 Salient features of Java

Java is an object-oriented language from the ground up, and derives most of its syntax and form from C++. Some of the salient features of Java are:

- Java is an interpreted language; the code (called bytecodes) generated by the Java compiler is targeted towards a hypothetical machine called the Java Virtual Machine (JVM), which has been implemented on top of all major platforms. The Java interpreter written for a particular machine then interprets these "bytecodes" to run on that specific machine. This is how Java achieves its cross-platform compatibility.
- One advantage that can derived directly from the above feature is that code can be "written once, run anywhere".

- Java is a pure object-oriented language; functions, or methods as they are called in Java, can only be defined classes. There is no concept of "stand-alone" methods in Java.
- There are no pointers in Java. Hence programmers cannot manage memory directly. Java performs automated garbage collection, thereby relieving the programmer of a lot of effort which would otherwise be needed.
- The basic data types of Java, such as *int, char* etc. are architecture-neutral. This again goes towards promoting the cross-platform nature of Java. Java comes bundled with a rich set of classes, thereby drastically reducing development time for companies.

As mentioned earlier, Java derives most of its syntax from C++, but the following are some important differences between Java and C++:

- Java does not allow multiple inheritance, while C++ does.
- Java is entirely object-oriented in that everything in Java is a class. On the other hand, C++ allows stand-alone functions as well as classes.
- In Java, method dispatch is performed dynamically, at runtime, except for static method invocations. In C++, method dispatch can be both dynamic and static.

3.2 Potential Uses for the Software Complexity Metrics for Java

In this section we address the question of how software complexity metrics based on program attributes can be useful in the software life-cycle and maintenance.

3.2.1 Use of Complexity Metrics to Produce Less Complex Programs

This particular use of metrics views metrics as a feedback tool; a historic collection of complexity metrics could be fine-tuned to reflect the complexity properties of programs, and can then be used to improve program quality in future projects. In this context, "complexity" could be the difficulty in modifying or enhancing a program, testing a program, etc. Thus, an organization might use historic data collected over a period of time to establish some kind of a complexity threshold. This complexity threshold should be established based on historical data that shows problems with maintenance and

modification activities if complexity exceeds those threshold levels. If the complexity measurement for a particular class exceeds that threshold, then the programmer might take some actions to keep the complexity of the class under control. The following are some possible scenarios:

- Seeing that the complexity measurements for a particular class exceed preset limits, a programmer might consider an alternative approach or algorithm to solve the problem.
- The programmer could consider further dividing up the work among more classes etc.

However, it should be recognized that it may not always be possible to keep the complexity of classes under pre-established limits. Some problems and their solutions may be necessarily complex. In this case, complexity metrics could be used to identify classes that need more attention in terms of documentation, comments etc. Another important point to note is that standards of complexity or quality are highly dependent on the organization that develops the software, as well as the software itself. Hence, instead of coming up with some magic numbers for complexity, it is our hope that an organization should consider all possible relevant factors and should establish local criteria for complexity.

3.3.2 Use of Metrics in the Maintenance Phase

Software metrics can also play a great role in managing the maintenance phase of the software life cycle. Metrics can be used to evaluate classes in terms of difficulty of understanding modification, to make estimates of the time and effort that may be needed to make modifications to a class etc. Often a software development organization would be better off rewriting a whole class rather than attempting to modify it, because the class is too complex. Metrics could be used as a tool to identify such classes. This kind of maintenance activity is often called *preemptive rewriting*. Another use of metrics in the maintenance phase could be in the allocation of classes to be maintained to programmers in such a fashion as to ensure an equitable distribution in terms of the complexity

encountered by each maintenance staff member, or to identify and assign more complex pieces of the software to more experienced people.

3.3.3 Use of Metrics to Allocate Testing Resources

Complexity metrics can be used as a tool to guide the allocation of resources for testing software. Typically, a large percentage of errors in software systems can be located in a small portion of the code. It is often said that 80% of the errors in a program can be found in 20% of the code. Judicious use of software complexity metrics can be a great aid in identifying and isolating portions of code which might need more rigorous testing in comparison with other portions of the program.

Thus, we have described some of the possible uses of metrics. However, as we cautioned earlier, metrics should be viewed as one of several tools used to manage the software development process. Under all circumstances, common sense should be used to determine if the application of metrics seems reasonable. Another important fact to be remembered is that metrics will work best when applied to a large set of programs characteristic or representative of the organization using the metrics. Applying metrics in isolation to one or two classes may not always be sufficiently generalizable. Any activity concerning human creativity like software development cannot always be predicted with 100% accuracy. This should always be kept in mind while deploying software complexity metrics.

3.3 A taxonomy of the metrics

Influenced by other metrics studies in OO languages, we propose the following classification for product metrics for Java:

- Complexity metrics at the project level
- Complexity metrics at the inheritance tree level
- Complexity metrics at the class level

This research focuses on metrics at the class-level, and suggests a few size-related metrics at the project and inheritance tree levels.

3.3.1 Complexity metrics at the project level

As we mentioned earlier, Java is a purely object-oriented language, meaning that everything in Java is a class, and there are no stand alone functions as in C++.
The following is a suggested list of size-related metrics that might provide useful information regarding the complexity of a program at the project level:

Number of classes:
Intuitively, we would expect a program with a greater number of classes to be more complex than a program a fewer number of classes. Hence this number could be used as an index of the complexity of a program.

Number of trees in the inheritance forest:
This number would give us the number of classes in the program that other classes may inherit from. This information might be crucial to understanding the program, because the greater the number of distinct classes that need to be understood in order to comprehend the program, the greater the effort to understand the program.

Average depth of the inheritance forest:
This is another metric that could provide useful information about the complexity of a Java program. The inheritance forest would comprise all the inheritance trees in the given compilation unit, not taking into account the fact that all the Java classes derive from one common class, the Object. The intuition here is that the deeper the inheritance structure for the program, the greater the interdependence among the classes belonging to the inheritance tree. Hence, the average of the depths of all inheritance trees in the forest would be a measure of the overall complexity of the program, from inheritance perspective.

Maximum depth of the inheritance forest:
This metric is similar in spirit to the above, and provides a "worst case" perspective on inheritance for a forest of inheritance trees.

3.3.2 Complexity metrics at the inheritance tree level

One of the most compelling features of a modern OO programming language like Java is code reuse. The great advantage about code reuse in Java is that it is possible to reuse the code (classes) developed and debugged by other programmers without changing the code. This is made possible in Java through a mechanism called *inheritance*. Inheritance involves taking the code from an existing class and adding code to it, without modifying the existing class. Inheritance is considered one of the cornerstones of the OOP paradigm and has several important implications for a metrics suite attempting to capture the complexity or quality of a Java program. Unlike C++, where a class could potentially inherit from any number of classes, in Java, a class can only inherit from one other class. The class from which other classes *inherit* or *derive* is called a *parent* class or *super* class. The class that inherits, on the other hand, is referred to as a *child* class, *derived* class or *sub*class. We will use these terms interchangeably, and to denote the same thing throughout this paper.

Since a subclass in Java can only have one super class (potentially), inheritance in Java programs gives rise to *tree structures.* A Java program could have several independently rooted inheritance trees, giving rise to an *inheritance forest*. These inheritance trees and forests have several important implications in terms of software complexity: these structures could greatly influence how easy or difficult a program is to understand, maintain or make enhancements to. We have designed a set of metrics that are very likely to provide valuable insights into the complexity of a program due to this feature:

Number of classes in the inheritance tree:

Looking at one inheritance tree in a Java program, the intuition behind this metric is obvious. The greater the number of classes in an inheritance tree, the greater the effort required to understand, and hence to maintain it.

The maximum depth of the inheritance tree:

The intuitions behind this metric are several. For one, the deeper a class is in the inheritance tree, the greater the number of methods it is likely to inherit, hence making it more complex to predict its behavior. Deeper trees might constitute greater design

complexity, since more methods and classes are involved. The deeper a class is in the hierarchy, the greater the potential reuse of inherited methods, and so on.

3.3.3 Complexity metrics at the class level

As mentioned earlier, a class is the unit of development in Java. Hence it becomes very important to have a good set of metrics that would let us evaluate the quality or complexity of a Java class. The following sections describe a list of metrics that capture the complexity level of a Java program at the class level. Out of these metrics, NOC and Depth are directly adapted from the Chidamber-Kemerer metrics suite. Method Count (MC) is a special case of WMC (Weighted Methods Per Class) in the C-K suite, where all methods are assumed to be have weights of unity. The External Reference metrics (NOR_NS, UNOR_NS, NOR_TOT and UNOR_TOT) are also derived from the CBO (Coupling among Object Classes) in the C-K suite, where coupling is defined in terms of the number of external methods invoked (or messages passed, in OO parlance) by a class.

Number of children/subclasses for a class (NOC):

The greater the number of children or subclasses that inherit from a particular class, the greater its complexity. The intuition for this belief is that since a greater number of classes inherit from this class, this particular class probably encapsulates the data and behavior for a wide range of classes. Hence everything else remaining the same, we would expect this class to be more complex than one which has fewer children. In some cases, it might even be said that if a class has a large number of children, it could be a case of misuse of subclassing.

Method Count for a class (MC):

In Object-Oriented programming languages, the behavior of a class is defined by the methods of a class. Hence, the number of methods defined in a class (Method Count) is a good indicator of the complexity of a class because a greater number of methods means that the class has a wider range of behavior and hence is more complex.

Constructor Count for a class (CC):

Constructors are special methods which are used to initialize the state of an object when it is first brought to life. From a client's point of view, a greater number of constructors represents more ways in which the object can be initialized and subsequently used. Hence the number of constructors could provide useful information regarding the perceived complexity of a class.

Data Count for a class (DC):

The member data for a class defines the state of the class, and hence the greater the data count for a class, the greater its complexity potentially.

Depth of a class in the inheritance tree (Depth):

The intuition for this metric is quite obvious. The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior. Also, the deeper a particular class is in the inheritance hierarchy, the greater the potential reuse of inherited methods.

External Reference Metrics

Classes in Java communicate with other classes mainly through method invocations. Classes in Java can invoke methods defined in other classes, usually called message-passing in Object-Oriented terminology. We refer to such method (function) invocations as "external references". There are two different ways in which these method invocations are executed: they can be either bound at run-time (dynamic binding), or at compile-time (static binding). Static binding is the approach taken in Procedural programming languages, and Object-Oriented languages need dynamic binding to support *polymorphism,* a property that allows a variable to hold different object at run-time. Polymorphism enables a variable of a given class or type to hold either objects of its own class or type, or of any subclass or extended class. Java allows a subclass or extended class to *override* or replace the superclass's implementation of a method with one of its own. When a method is invoked on a polymorphic variable holding an object, the *actual type* of the object at run-time, as opposed to its static declared type, governs which

implementation is used. This is the reason why method invocations in Java are bound at run-time. The following example illustrates this concept.

Let us consider an example of class called *Shape*, which represents the abstract idea of a geometric shape. This class has methods such as draw(), paint() etc. which are characteristic behaviors of a geometric shape. Concrete geometric shapes such as *Circle*, *Triangle* etc. can then be defined as subclasses of *Shape*. Since shape is still an abstract concept at the time that we define the class, usually the methods draw() paint() etc. will not have any implementation. The subclasses *Circle*, *Triangle* etc. inherit these abstract methods and override them to provide their own specific implementation. Let us consider the following declaration of class *Shape*:

Shape aShape;

This variable aShape is polymorphic, and can hold any object that is an instance of Shape or any subclass of Shape. Hence, aShape might very well hold a Triangle or a Circle at some point in time. A method invocation on aShape, such as aShape.draw(), will then either call the draw() method of a Triangle or a Circle, depending on what object aShape happens to hold at that point in time.

Java also has per-class (as opposed to per-instance) methods or static methods, invocations to which are bound at the time the Java classes or loaded in the virtual machine. As an example, the Math class in the JDK (Java Development Kit) core library has several static methods such as cos, sine, etc. that provide a useful collection of math-related methods. Such methods are invoked by prefixing the method name with the class name, for example, Math.sine(angle) is an invocation of the sine method defined in the Math class. These method calls are not bound dynamically , and hence are very similar to function calls in procedural programming languages.

Since objects in Java are primarily coupled among themselves through these method invocations, this gives us a way to measure coupling. Also, the differences in the two types of method invocations have an important effect on program understanding, because in the case of non-static method invocations, method selection happens at run-time, and can be one of several possible choices. Hence understanding non-static method calls can be more difficult than understanding non-static method invocations. Based on this, we propose the following external reference metrics for a class :

Number of References (NOR_TOT): This metric is a count of the total number of external references, both static and non-static.

Unique Number of References (UNOR_TOT): This metric is a count of the *unique* number of references, both static and non-static included.

Number of Non-Static References (NOR_NS): This metric is a count of the total number of non-static external references.

Unique Number of Non-Static References (NOR_NS): This metric is a count of the *unique* number of non-static external references.

4.0 The Architecture and Design of JavaMetrics

4.1 The need for an automated metrics collection system

The process of collecting metrics on large amounts Java source code over a period of time can become a very time-consuming task. Also, manual gathering of metrics can be quite error-prone and very expensive. For these reasons, the metrics collection effort needs the support of automated tools. As a step in this direction, we have designed and implemented *JavaMetrics* , a static metrics analyzer for Java.

4.2 An overview of JavaMetrics

JavaMetrics is a static metrics analyzer tool for Java which parses Java source code statically and computes all the class level metrics proposed in the earlier section. JavaMetrics has three essential functions: (i) parsing the input Java source file and extracting the required information; (ii) storing the extracted information in a suitable format; (iii) and processing this information to compute the desired metrics. JavaMetrics has been written completely in Java, so any platform which supports Java should be able to run JavaMetrics without any recompilation.

4.3 The Architectural Design and Implementation of JavaMetrics

Figure 1 shows the architecture of JavaMetrics. JavaMetrics is composed of the following five classes:

- *Parser*
- *MainTable*
- *TableRow*
- *Organizer*
- *JavaMetrics*

The following subsection describes the responsibilities of each of the five classes, and how they collaborate and interact with each other to compute the specified set of metrics for a given Java source program.

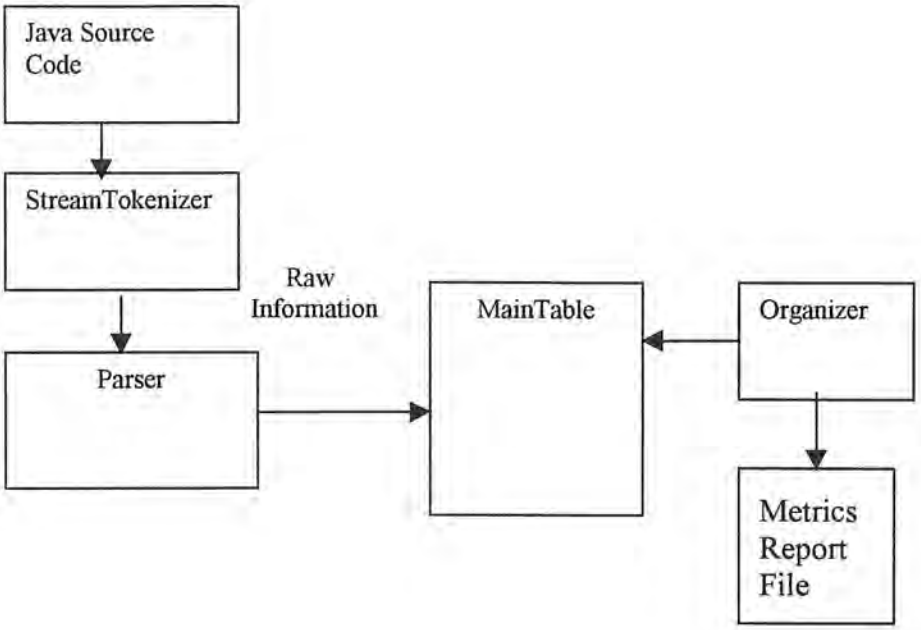

Figure 1.

4.3.1 MainTable class

This class serves as the central repository for the raw information collected from the Java source file by the *Parser* class, and accessed later by the *Organizer* and *JavaMetrics* classes. This class as such, is very crucial to the whole design, and in an effort to keep the system as scalable as possible, all the classes external to this class can only access the data stored in this class through the appropriate accessor and mutator methods provided by the class. More specifically, MainTable has a row corresponding to every Java class in the entire Java program, and has the following fields for each row in the table:

1. A string containing the name of the class. (This is the primary key to the table).
2. A string containing the name of the class's parent, if any.

3. An integer for the method count for the class.
4. An integer for the constructor count for the class.
5. An integer for the data count for the class.
6. A boolean field indicating whether the current class is a root of an inheritance tree.
7. A boolean field indicating whether the parent of the class, if there is one, has been defined in the current program or if it is a library class.
8. An integer for the number of direct children for the class.
9. An integer for the depth of the current class in its inheritance tree.
10. A vector of strings holding all the method names for the current class.
11. A hash table of strings for all the external references in the current class. In this table, the key is the string representing the external reference, and the value is the number of times this particular reference appears for the class.
12. A hash table of strings for all the static references in the class. Again, the key is the string representing the external reference, and the value is the number of times this particular reference appears for the class.

The MainTable abstraction encapsulates this storage and access to the data, and serves the data on request through accessor and mutator methods. The actual manner in which MainTable is implemented is completely insulated from the user, hence MainTable can change its implementation sometime in the future without having to modify any of the other classes which use MainTable. At this point, MainTable has an instance of the Hashtable class which comes with the Java API. If it should be decided to store the data in a commercial Relational Database Management System (RDBMS), for example, this can be done by localizing the changes only to the MainTable class, without impacting the rest of the class in any way. Thus the object-oriented design of JavaMetrics provides for a very flexible and open framework for future modifications and enhancements.

4.3.2 TableRow class

This class is a wrapper class for the various fields stored by MainTable, and provides access to them through get and set methods. This class is necessary because the class Hashtable provided by Java's standard library allows the storage of key-value pairs, and

in order to associate a particular class with all the information belonging to the class, we need to have a composite class which stores all the fields mentioned earlier.

4.3.3 Parser class

This class is the work horse class of JavaMetrics and does the important function of parsing the input Java source file to extract information from a given Java source file, and stores it in MainTable. This class makes use of a class provided by Java's standard library for lexical analysis, viz. *StreamTokenizer*, to parse the Java source file.
The following is the main loop which does the parsing:

```
while there are more classes in the file
{
    add the class name to MainTable;
    get the information for the class;
    store the information in the TableRow in MainTable corresponding to this class.
}
```

The following are some interesting notes and observations regarding the parsing process:

- The parser goes through three passes on the input source file before it finishes parsing. This is necessary due to the following reasons:
    1. Java, as a typical Object-Oriented programming language, permits programmers to define new data types which can be declared and used just as though they were primitive data types provided by the language. For example, consider the data type *int*, which is a primitive data type provided by the Java language. To declare a variable of type *int*, the programmer just needs to say

        int i;

        and can then go on and use this variable *i* in his/her program. However, now suppose that the programmer needs a data type *complex* which would allow him/her to operate on complex numbers. There is no such primitive data type provided by the Java language, but this is where the power of an OO programming

language comes in: the programmer can now define a new data type called *complex*, and go on and use this just as if it were provided by the language. The philosophy behind this approach is that instead of trying to foresee all possible data structures that may be needed by the programmer and provide them as part of the language, provide the programmer with the facility to define new data types and use them freely.

This factor has a very important implication for our metrics analysis purposes, since one of the metrics we compute for a class is the number of member data that it has. Since in Java it is possible for the programmer himself to define new data types and use them in other classes, there is no way for us to know beforehand what the set of data types is. Hence, in JavaMetrics, we go through the entire program in the first pass and and collect all information about the new data types defined by the programmer, and then during subsequent passes, identify the data members for a particular class.

2. For a very similar reason, a second pass through the source code is needed to collect the *external references.* Since all the information about the methods defined in a particular class is necessary before we can start analyzing the external references for the classes, information about all the methods defined for a particular class is collected during the second phase. The actual metrics collection for *external references* is done in the third pass.

   To make this more clear, let us consider a simple example. Let us suppose that there are two classes, class A and class B in a Java program being analyzed by JavaMetrics. Let us suppose that A has a method foo() defined in it, which is referenced from B. Now when we actually locate the external reference from B to A,

through the method foo(), we need to know where the method foo() is defined (class A in this case). Information regarding where (or in which class) a method has been defined is essential before the *external references* metrics can be computed, so during the second pass, we create a database of all method definitions for each class in the project. Using this database during the third pass we can external refernces in a particular class, and hence compute our metrics.

4.3.4 Organizer Class

The Organizer class performs the function of analyzing and organizing the "raw information" collected from the Java source and stored into the MainTable. The information collected by the Parser class by parsing the input source does not contain the information needed to compute the metrics, so the Organizer class performs the additional computation needed to compute all the metrics, by accessing the information available in the MainTable. In particular, the Organizer class sifts through the information collected during the parse stage and identifies the inheritance tree structures for the given source program being analyzed. The following algorithm is used for this process by the Organizer class:

- **Algorithm for computation of metrics involving inheritance tree structures:**

  One interesting problem that needed to be solved while computing the metrics suite is that of identifying the inheritance tree structure in a given Java source program to be analyzed by JavaMetrics. The situation in Java is less complicated compared with a language like C++ because, unlike C++, Java does not allow multiple inheritance, meaning that a class cannot be a subclass of more than one other class. Hence, inheritance structures in Java are trees, and not graphs, as would have been the case with a language allowing multiple inheritance.

  We use the following algorithm to identify the inheritance structures in a given Java source program:

Input: The MainTable containing the "raw information" regarding the classes in the given Java source file. This data structure is the output of the Parser. We call this "raw information", because it is information that can be directly gleaned from the source without any additional processing, such as the name of a class, the name of the parent of a class etc.

Algorithm:

1. Walk through the MainTable and identify all "roots" in the program, i.e. all classes which do not derive from any other class. For the purpose of computing inheritance trees, we classify the classes as follows:
   - classes which do not inherit from anything else; These are the "roots" of the inheritance trees in the Java program .
   - classes which inherit from other library classes which are not part of the current Java program bering analyzed. JavaMetrics always considers those classes that are not available for analysis, mostly library classes supplied by vendors, as black boxes. Hence, classes which derive from library classes are also regarded as "roots".
   - classes which derive from some other class which is part of the Java program being analyzed, and hence a "non-root".
2. For each root identified in the previous step:

```
{
Push the root into a stack.
while the stack is not empty {
        Pop the top class off the stack.
        Determine all classes which have this particular class as their
        immediate parent and push them into the stack.
        }
}
```

The above algorithm is constructs the tree in a breadth-first fashion.

4.3.5 Class JavaMetrics

The JavaMetrics class is the starting point for the whole metrics analysis process, and is the manager of all the other classes and modules in the system. It also implements a simple graphical user interface (GUI) through which the user can select files for analysis, initiate the metrics analysis process and quit the metrics analysis system. When the user clicks the appropriate buttons and chooses a Java source file for analysis, the class JavaMetrics first activates the class Parser, and when parsing is done and the MainTable has all the "raw information", activates the Organizer class which then performs the additional processing described earlier. Once everything is done, the JavaMetrics class again takes over and writes the results of the analysis into a file which can be opened and read by the user. The files storing the metrics analysis results are generated by adding a .metrics suffix to the source file name. For example, for a Java source file with the name JavaMetricsSource.java submitted for analysis to JavaMetrics, a file named JavaMetricsSource.java.metrics is generated with the results of the metrics analysis. The following page shows a sample metrics report generated by JavaMetrics.

5.0 Experimental Validation of the Class-Level Metrics for Java

In this chapter, we describe two experiments that were conducted to determine the utility of the class-level metrics proposed for Java as part of this research.

5.1 The need for a scientific validation process

The main goal of the experimental validation is to determine which, if any, of the proposed metrics are good indicators of the complexity of Java programs. This allows us to

- Determine whether our intuitions regarding the complexity about Java programs are experimentally supported.
- Filter out the metrics which are experimentally suggested as being useful from those that are not. These metrics could then be used to identify classes most likely to be difficult to test or to modify.

5.2 The Experiments

With these goals in mind, two experiments were done to validate the proposed set of class level metrics for Java. In order to validate our metrics suite, we considered two different possible approaches: (1) small-scale controlled experiments using small or contrived Java programs, (2) real-life large scale industrial case studies. In this project, we took a hybrid approach, in that the programs that were studied were real applications, and not contrived or trivial. However, the programs are not exceptionally large; two of the programs studied in experiment I are parts of a commercial software system but these programs are not currently being shipped to customers. Experiment II was conducted entirely in a combined graduate/undergraduate Computer Science course in compiler construction at Oregon State University

The design of the experiments, the data collected and the results from the experiments are discussed in the following subsections.

5.2.1 Experiment I

Background.

In this experiment, the subjects are professional Java programmers. They were asked to select a Java program that they have worked with, either as developers or testers, or in any other capacity which gave them sufficient familiarity with the programs so they could evaluate the complexity of the Java classes. Then, the candidates were requested to rate the complexity for each class - difficulty in performing both testing and modification tasks for the classes in the programs on a 4 point scale, with the following meaning assigned for each level of difficulty:

1 - very easy/trivial
2 - easy
3 - difficult
4 - very difficult

There were two main stages in the experiment:

- Data Collection stage, where we collect metrics and subject rating data for a set of classes
- Analysis stage, where we use statistical methods to analyze the metrics using the data collected in the previous stage.

Data Collection

The main goals of this step in the experiment were:

- to gather metrics data for the sample of Java classes
- to collect complexity level ratings for the different from a "reliable agent". The reliable agent could be the programmer who actually wrote the code, or any other person who is in a legitimate position to evaluate the complexity of the code, like testers etc.

In the following subsections, we provide some background on each of three subjects and their programs, and the data collected from each subject.

Subject A:

Subject A is a software engineer in a software development company with one year of experience in the field. The subject chose a Java program developed entirely by himself for the purpose of the evaluation. This program is a Java applet and is the front-end to a commercial Object-Oriented Database Management System (OODBMS). The main goal of this program was to demonstrate the web and Java access features of the OODBMS. His ratings and the metrics values computed for all the ten classes are shown in Table 1.

Subject B:

The subject is a senior software engineer in a software development company with over 7 years experience in the field. The subject holds a BS degree in computer science.
The program chosen by the subject is a part of a Java library that provides Java proxies for a recently released commercial Object-Oriented Database Management System (OODBMS). These classes act as proxies for the multimedia class library provided by the OODBMS, and as such reflect the same inheritance hierarchy as the original classes in the database. The various metrics computed by the tool for the programs, and the subject's ratings of the complexity of the classes in the program on a scale of 1-4, is as shown in table 2.

Subject C:

The subject is a MS graduate in Computer science. The subject has more than two years' experience programming in Java. The program chosen by the subject is a game application which can be played over the web as a Java applet. As such, the program to a great extent, consists of code that deals with user interface. The metrics for the program, and the rating of the complexities of the various classes in the project are shown in Table 3.

	Depth	D.C.	M.C	C.C	NOC	NOR_TOT	UNOR_TOT	NOR_NS	UNOR_NS	Rating
Acknowledment	0	2	6	3	0	6	5	6	5	1
AddTest	0	7	7	3	0	27	21	26	20	2
ImageCanvas	0	7	4	1	0	5	5	4	4	1
InvalidID	0	2	5	0	0	5	5	5	5	1
MainFrame	0	15	12	2	0	60	53	58	52	2
PatFrame	0	7	9	2	0	64	57	62	56	2
PatInfoFrame	0	51	14	2	0	273	147	267	146	4
PhysicianInfo	0	9	5	2	0	26	26	26	26	3
TestDetail	0	9	7	3	0	23	22	21	21	3
TestShort	0	8	8	2	0	39	37	36	36	3

Table 1.

Legend:

Depth: Depth of a class in inheritance tree
DC: Data Count
MC: Method Count
CC: Constructor Count
NOC: Number of Children
NOR_TOT: Total number of References (non-unique)
UNOR_TOT: total number of References (unique)
NOR_TOT: Total number of Non-static References (non-unique)
UNOR_TOT: total number of Non-static References (unique)

	Depth	D.C.	M.C	C.C	NOC	NOR_TOT	UNOR_TOT	NOR_NS	UNOR_NS	Rating
MMData	3	5	1	1	1	0	0	0	0	1
MMFile	4	9	10	2	2	44	15	32	14	3
MMImageFile	5	0	1	1	4	0	0	0	0	1
MMJavaAudioFile	7	0	2	2	0	0	0	0	0	1
MmjavaImageFile	6	3	7	4	0	18	8	11	7	4
MMPixmapFile	6	3	5	4	0	7	6	5	5	4
MMProperty	0	5	1	1	0	0	0	0	0	1
MMSoundFile	5	0	1	1	1	0	0	0	0	1
MMSunAudioFile	6	3	4	3	1	6	3	2	2	4
MMTiffFile	6	3	5	4	0	14	7	9	6	4
Mmedia	2	0	1	1	1	0	0	0	0	1
Video	6	0	0	0	0	0	0	0	0	3
ImageViewer	0	0	12	3	0	26	17	25	16	3
JpRoot	0	3	0	0	1	0	0	0	0	2
RootClass	0	4	8	1	1	6	5	4	4	2

Table 2.

	Depth	D.C	M.C	C.C	NOC	NOR_TOT	UNOR_TOT	NOR_NS	UNOR_NS	Rating
DropEvent	0	0	2	1	0	0	0	0	0	3
Grid	0	10	10	1	2	11	6	11	6	3
ShipGrid	1	6	11	1	0	12	8	11	7	3
FireGrid	1	4	8	1	0	8	8	8	8	2
InfoDialog	0	1	2	1	0	7	7	7	7	2
LabelCanvas	0	6	7	1	0	7	6	7	5	2
InfoFrame	0	1	3	2	0	8	7	8	7	2
Login	0	7	5	1	0	33	23	33	23	2
Ship	0	6	6	1	5	4	4	4	4	2
BattleShip	1	3	1	1	0	0	0	0	0	2
Destroyer	1	3	1	1	0	0	0	0	0	2
Submarine	1	3	1	1	0	0	0	0	0	2
CannonCruiser	1	3	1	1	0	0	0	0	0	2
SeaHarrier	1	3	1	1	0	0	0	0	0	2
ToolbarButton	0	2	11	1	0	10	8	10	8	3
ToolbarPanel	0	5	6	1	0	20	12	20	12	3
LogoPanel	0	1	2	1	0	5	5	5	5	2
BattleConnection	0	2	1	0	0	0	0	0	0	1
BattleApplet	0	36	24	0	0	109	62	89	60	4
BattleClientImp	0	1	9	2	0	7	7	7	7	2
BattleServer	0	5	8	3	0	45	24	37	22	3
ChallengeDialog	0	3	2	1	0	7	7	7	7	2

Table 3.

Data Analysis.

The subject ratings indicate the perceived difficulty in performing testing or modification tasks. We used Spearman's Rank Correlation coefficient to investigate the relationship between each of the metrics and the subject ratings.

Spearman's Rank Correlation Coefficient [20] is a number between -1 and +1, and quantifies the similarity between two different sets of ranks. Spearman's rank correlation coefficient takes into account only the rankings of entities on the basis of the values of some property of that entity, and not the numerical value of the property itself. Ties in ranks are resolved by assigning the means in rank values to the entities that have the ties. When the two sets of rankings being compared match perfectly, the value of Spearman's rank correlation is +1 and there is perfect positive correlation between the two. On the other end of the spectrum, if the two sets of ranking are the exact opposite of each other, there is perfect negative correlation between the two sets of rankings and the value of Spearman's rank correlation coefficient is –1. If the value of Spearman's rank correlation is near 0, there is no relationship between the two sets of rankings.

For each subject, the different classes that the subject rated were ranked according to the subject ratings and according to the different metrics. For each metric, Spearman's rank correlation coefficient was computed between the set of rankings obtained using the subject ratings and the set of rankings based on the metric. This was done for all the three subjects. Table 4 summarizes the results of this study:

	Depth	DC	MC	CC	NOC	NOR_ TOT	UNOR _TOT	NOR_ NS	UNOR _NS
Subject A	N/A	.827**	.570	.252	N/A	.663*	.745*	.682*	.739*
Subject B	.384	.185	.520*	.659*	-.360	.767**	.754**	.754**	.754**
Subject C	-.199	.356	.669**	.036	.104	.620**	.533**	.620**	.493**

Table 4.

*Correlation is significant at the 5% level (2-tailed)

**Correlation is significant level at the 1% level (2-tailed).

N/A Not applicable because metric is 0 throughout

A discussion of the results:

As can be observed from table 4 of the above study, the following metrics have significant correlations with subjective ratings of complexity, for all of the subjects:

- NOR_TOT
- UNOR_TOT
- NOR_NS
- UNOR_NS

In addition, the metrics CC, MC and DC have significant correlations with subjective impressions of complexity with at least one subject, but not all of them . It might also be observed that the metrics NOR_TOT and UNOR_TOT highly correlated (0.954 for subject A, 0.995 for subject B and 0.962 for subject C). Similarly, NOR_NS and UNOR_NS are highly correlated between themselves (0.976 for subject A, 0.996 for subject B, and 0.946 for subject C) and do not differ very greatly from each other in their correlations with the ratings. Hence we use one of the two metrics in each case for analysis purposes instead of considering both.

5.3.1 Experiment II

Experiment Goal.

The goal of this experiment was to investigate the relation between subjective ratings of difficulty in performing testing or modification tasks, and metrics values in a larger group of subjects.

Subjects and Materials.

In this experiment, the subjects consisted of a set of 27 undergraduate (CS480) and 17 (CS580) graduate students taking a senior/graduate level course in compilers at Oregon State University. The students developed a compiler for a small programming language using Java. The compiler development was done in several stages, and this experiment was conducted in the middle of that development process. At the time this experiment was conducted, the software system consisted of 33 Java classes, and was partly functional. The two main classes developed entirely by the students included the Lexer and the Parser. The Lexer class provides a stream of tokens on demand by the Parser. The Parser performs syntax analysis, and uses the SymbolTable related classes, Symbol related classes and Type related classes. These supporting classes were written by the professor, and provided to the students for use in developing their Lexer and Parser. The students had to read and understand the supporting classes, and use them in their code.

Experiment Design and Data collection.

The 44 subjects were asked to rate the 33 Java classes on a 1-4 scale (1- very easy, 2- easy, 3- difficult, 4- very difficult), for each of the two tasks i.e. testing and modification. Tables 5 and 6 show the frequency of each rating level for the modification task and for the testing task.

	very easy			easy			Difficult			very difficult		
	cs580	cs480	Comb.	cs580	cs480	Comb.	cs580	cs480	comb.	cs580	cs480	comb.
Symbol	3	5	8	13	13	26	1	7	8	0	2	2
ConstantSymbol	4	4	8	10	18	28	3	4	7	0	1	1
TypedSymbol	3	0	3	11	20	31	3	6	9	0	1	1
TypeSymbol	3	0	3	10	20	30	4	6	10	0	1	1
GlobalSymbol	3	0	3	11	16	27	3	10	13	0	1	1
OffsetSymbol	2	0	2	10	14	24	5	12	17	0	1	1
FunctionSymbol	3	0	3	8	13	21	6	13	19	0	1	1
NestedFunctionSymbol	2	0	2	6	11	17	7	14	21	2	2	4
MethodSymbol	2	0	2	8	14	22	6	10	16	1	3	4
SimpleSymbolTable	2	0	2	9	10	19	5	14	19	1	3	4
RecordSymbolTable	0	1	1	7	11	18	10	12	22	0	3	3
ArgumentSymbolTable	1	0	1	7	11	18	7	11	18	3	4	7
NestedSymbolTable	1	0	1	8	9	17	7	13	20	2	4	6
FunctionSymbolTable	1	0	1	5	9	14	9	14	23	2	4	6
ClassSymbolTable	2	1	3	8	9	17	4	13	17	2	5	7
SimpleType	4	4	8	12	14	26	1	8	9	0	1	1
PrimitiveType	3	3	6	13	13	26	1	9	10	0	2	2
PointerType	2	1	3	7	11	18	6	11	17	2	4	6
AddressType	3	2	5	4	12	16	9	9	18	1	4	5
RecordType	3	1	4	6	12	18	7	11	18	1	3	4
ArrayType	3	1	4	6	13	19	7	10	17	1	3	4
Functiontype	3	2	5	6	12	18	8	12	20	0	1	1
ClassType	2	1	3	11	13	24	4	11	15	0	2	2
Ast	5	5	10	8	6	14	3	12	15	1	4	5
FramePointer	3	2	5	7	6	13	7	14	21	0	5	5
UnaryNode	5	1	6	10	10	20	2	14	16	0	2	2
BinaryNode	4	3	7	6	12	18	7	12	19	0	0	0
GlobalNode	3	1	4	9	17	26	4	7	11	1	2	3
IntegerNode	3	6	9	13	16	29	1	5	6	0	0	0
RealNode	2	6	8	12	15	27	3	6	9	0	0	0
StringNode	4	6	10	9	15	24	4	5	9	0	1	1
Lexer	3	3	6	9	17	26	4	3	7	1	4	5
Parser	2	2	4	4	4	8	6	15	21	5	6	11

Table 5:
Modification Task : Subject Responses

	very easy			easy			Difficult			very difficult		
	cs580	cs480	Comb.	cs580	cs480	Comb.	cs580	cs480	comb.	cs580	cs480	comb.
Symbol	3	2	5	10	10	20	4	14	18	0	1	1
ConstantSymbol	4	2	6	11	15	26	2	9	11	0	1	1
TypedSymbol	2	1	3	12	15	27	3	9	12	0	2	2
TypeSymbol	1	1	2	11	15	26	5	9	14	0	2	2
GlobalSymbol	3	1	4	10	12	22	4	14	18	0	0	0
OffsetSymbol	1	1	2	11	8	19	5	17	22	0	1	1
FunctionSymbol	1	1	2	8	6	14	8	14	22	0	6	6
NestedFunctionSymbol	1	2	3	4	4	8	10	14	24	2	7	9
MethodSymbol	1	2	3	9	6	15	5	16	21	2	3	5
SimpleSymbolTable	4	1	5	6	6	12	5	16	21	2	4	6
RecordSymbolTable	1	1	2	9	5	14	6	17	23	1	4	5
ArgumentSymbolTable	2	2	4	6	4	10	6	18	24	3	3	6
NestedSymbolTable	2	1	3	6	5	11	7	15	22	3	5	8
FunctionSymbolTable	1	1	2	5	5	10	9	15	24	2	6	8
ClassSymbolTable	1	1	2	6	6	12	8	14	22	2	6	8
SimpleType	2	4	6	12	15	27	3	8	11	0	0	0
PrimitiveType	2	3	5	13	15	28	2	9	11	0	0	0
PointerType	2	2	4	10	10	20	4	10	14	1	5	6
AddressType	1	3	4	10	8	18	5	10	15	1	6	7
RecordType	1	3	4	9	5	14	6	18	24	1	1	2
ArrayType	2	2	4	7	9	16	7	13	20	1	3	4
Functiontype	2	3	5	11	5	16	7	16	23	0	0	0
ClassType	2	3	5	8	7	15	8	16	24	0	0	0
Ast	6	5	11	8	8	16	2	11	13	1	3	4
FramePointer	2	3	5	8	8	16	6	15	21	1	1	2
UnaryNode	3	5	8	9	10	19	5	10	15	1	1	2
BinaryNode	1	5	6	9	10	19	6	10	17	1	1	2
GlobalNode	2	4	6	11	14	25	3	7	10	1	2	3
IntegerNode	2	7	9	13	13	26	1	7	8	1	0	1
RealNode	1	7	8	13	13	26	2	7	9	1	0	1
StringNode	4	7	11	11	13	24	1	6	7	1	1	2
Lexer	3	3	6	9	13	22	4	9	13	1	2	3
Parser	3	2	5	5	4	9	6	13	19	3	8	11

Table 6:
Testing Task: Subject Responses

In addition to collecting the responses from the students, the metrics were also computed for each class. The following table shows the metrics for the different classes:

	Depth	DC	MC	CC	NOC	NOR_tot	UNOR_tot	NOR_NS	UNOR_I
SimpleType	0	0	5	1	5	0	0	0	0
PrimitiveType	1	5	3	1	2	0	0	0	0
PointerType	2	1	5	1	1	4	4	4	4
AddressType	3	0	2	1	0	1	1	1	1
RecordType	1	1	3	1	0	2	2	2	2
ArrayType	1	3	4	1	0	3	3	3	3
Functiontype	1	2	4	1	0	1	1	1	1
ClassType	1	1	4	1	0	1	1	1	1
SimpleSymbolTable	0	2	12	1	2	8	6	5	5
RecordSymbolTable	1	1	5	2	1	2	2	1	1
ArgumentSymbolTable	3	0	3	1	0	5	5	4	4
NestedSymbolTable	2	2	3	1	3	4	4	3	3
FunctionSymbolTable	3	2	4	1	0	4	4	4	4
ClassSymbolTable	3	1	5	2	0	5	5	5	5
Symbol	0	1	4	1	2	0	0	0	0
ConstantSymbol	1	1	2	1	0	0	0	0	0
TypedSymbol	1	1	2	1	4	0	0	0	0
TypeSymbol	2	0	1	1	0	0	0	0	0
GlobalSymbol	2	1	3	1	0	0	0	0	0
OffsetSymbol	2	1	2	1	0	0	0	0	0
FunctionSymbol	2	1	2	1	2	0	0	0	0
NestedFunctionSymbol	3	0	2	1	0	1	1	1	1
Ast	0	1	4	1	8	2	2	2	2
FramePointer	1	0	2	1	0	1	1	0	0
UnaryNode	1	6	2	1	0	6	2	1	1
BinaryNode	1	12	2	1	0	15	3	2	1
GlobalNode	1	1	2	1	0	1	1	0	0
IntegerNode	1	1	3	2	0	2	2	1	1
RealNode	1	1	3	2	0	2	2	1	1
StringNode	1	1	2	1	0	1	1	0	0
Lexer	0	9	7	1	0	38	8	30	6
Parser	0	2	29	1	0	241	36	229	35

Table 7: Metrics Data

Analyzing the Data.

The goal of conducting this experiment was to investigate the relationship between the perceptions of difficulty in performing testing and modification tasks by the subjects, compared with the different metrics. Again, we use Spearman's Rank Correlation Coefficient to investigate the relationship between the two.

For the purpose of analyzing the data, we computed a *weighted difficulty index* for each class, for the testing and for the modification task. This *weighted difficulty index* for a class is computed by multiplying the number of respondents in each of the four rating levels by the corresponding weights for that level: 1 for very easy, 2 for easy, 3 for difficult and 4 for very difficult. For example, if a class X had 5, 10, 15 and 14 respondents rating X as very easy, easy, difficult and very difficult respectively, the *weighted difficulty index* would be computed as $1*5 + 2*10 + 3*15 + 4*14 = 126$.

The *weighted difficulty indices* for each class for each task, provided a basis for ranking the classes that reflects a a consensus among the subjects. The classes were also ranked according to the metric values. For each metric, we then computed Spearman's rank correlation coefficient between the ranking for that metric and the ranking of the classes using the *weighted difficulty*. Table 8 shows Spearman's correlation coefficient values obtained for the different metrics for each group of students (cs480 and cs580), and all students for both the modification and testing tasks.

Modification Task									
	Depth	DC	MC	CC	NOC	NOR_TOT	UNOR_TOT	NOR_NS	UNOR_NS
CS580	.454**	-.12	.247	.024	-0.21	.508**	.599**	.641**	.646**
CS480	.376*	.06	.251	-.112	-.17	.496**	.536**	.566**	.581**
Combined	.435*	-.003	.280	-.073	-.066	.486**	.552**	.603**	.161**
Testing Task:									
	Depth	DC	MC	CC	NOC	NOR_TOT	UNOR_TOT	NOR_NS	UNOR_NS
CS580	.528**	.060	.081	-.049	-.303	.481**	.510**	.556**	.548**
CS480	.451**	-.034	.269	.152	.012	.256	.338	.469**	.489**
Combined	.493**	-.045	.212	-.005	.-.054	.328	.395*	.495**	.504**

Table 8

*correlation significant at 5% level in two-tailed test
**correlation significant at 1% level in two-tailed test

A discussion of the results

From an examination of the above results, the following observations can be made:

- It is clear that three of the metrics, Depth, NOR_NS and UNOR_NS have significant correlations with subjective rankings, for both the testing and modification tasks. These three metrics are significantly correlated with the rankings of cs580 students, cs480 students, and all students, for both testing and modification tasks. Also, NOR_NS and UNOR_NS are strongly correlated between themselves (Spearman's correlation coeffiecient 0.994) and do not differ from each other greatly in their correlations with subject rankings.
- The metrics NOR_TOT and UNOR_TOT have significant correlations for the modification task with subjective rankings. But for testing, there is significant correlation only with the ranking of cs580 students.

- In table 8, the CS580 graduate students' rankings have the largest number of significant correlations (significant at the 1% level), with the various metrics (Depth, NOR_TOT, UNOR_TOT, NOR_NS and UNOR_NS) for both modification and testing tasks.

Modification task		
	NOR_NS	Depth
Combined	6/9(67%)	6/9(67%)
CS580	6/9(67%)	7/9(78%)
CS480	6/9(67%)	6/9(67%)
Testing task:		
	NOR_NS	Depth
Combined	5/9(56%)	6/9(67%)
CS580	5/9(56%)	6/9(67%)
CS480	6/9(67%)	5/9(56%)

Table 9

- Looking at the top few most difficult classes as picked by the subjects for the testing and modification tasks, it would be interesting see how many of those classes are also picked by the metrics. In the ideal case, there would be a perfect match between the set of classes picked by the metrics and that picked by the subjects. Table 9 shows the results of this line of inquiry for the 9 classes rated most difficult for the metrics NOR_NS and Depth. A practical use of metrics is identifying and then allocating resources to classes that are very likely to be hard to test or modify. Of the 9 classes identified as most complex by the metrics, about two-thirds of them were among the rated as most difficult to test or modify by the subjects.

- The metrics seem to do a slightly better job of identifying the most difficult classes for modification than for testing. More specifically, if we look at the classes that are identified as the most difficult by the NOR_NS and UNOR_NS metrics, 3 classes consistently fail to appear in the list suggested by the weighted difficulty index. These classes are : Lexer, ArrayType and SimpleSymbolTable. The class Lexer has the second highest value for NOR_NS and UNOR_NS, but a Depth of 0. This low depth, coupled with the fact that Lexer was the first Java class to be developed by the subjects, and hence probably quite familiar to the subjects at the time they answered

the questionnaire, suggests a reason why Lexer is not being perceived to be difficult for, testing or modification tasks by the subjects. The classes SimpleSymbolTable and ArrayType have low depths 0 and 1 respectively, which could be a reason why they are not considered to be difficult by the subjects.

Since the metric NOR_NS had significant correlations with subject rankings of complexity in the first experiment, we compared the top few most difficult classes as selected by the subjects, with the classes picked by NOR_NS. Table 10 shows the results. Column 3 in table 10 is the number of classes rated very difficult by the subject that were also ranked among the most difficult by NOR_NS. Column 5 in the table shows the number of classes that were ranked either difficult or very difficult by the subjects and also picked as the most difficult by NOR_NS. For example, subject B rated seven classes as being very difficult or difficult to test and modify, and two of these classes were also picked as most difficult by NOR_NS. As can be observed from the table, the percentage of agreement is often about 60-70%, sometimes as high as 85%. Hence NOR_NS seems to do a reasonable job of identifying those classes that are likely to be difficult to modify or test.

	No. of classes rated very difficult by subject	NOR_NS rated	No. of classes rated difficult or very difficult by subject	NOR_NS Rated
Subject A	1	1	4	2
Subject B	4	2	7	6
Subject C	1	1	7	6

Table 10

6.0 Conclusions and Future Directions

6.1 Summary

In this research, we have investigated Objected-Oriented metrics for the Java programming language and proposed a set of metrics at the class level. A static metrics analyzer tool was developed to automate the metrics collection process. To validate the metrics, two experiments were conducted. The results of the experiments showed that NOR_NS is a good metric. NOR_NS also proved to be useful in identifying those classes which are likely to be hard to modify or test.

6.2 Conclusions

From the results of the two experiments conducted to validate the metrics suite proposed for Java, suggest the following:

- There is strong evidence that the external references metric (NOR_NS), which provides a measure of the degree of coupling or interaction a class has with other classes, is a good indicator of difficulty in testing or modifying a Java class. In the Object-Oriented paradigm, computation is done primarily through these interactions between classes, and that could be the reason why this metric seems to work well.
- The Depth of a class in its inheritance tree is another metric which received strong support from the data collected in the second experiment. The programs considered in the first experiment did not make heavy use of inheritance, and hence could not provide evidence in support of this metric. So again, it would seem that this metric can be useful in picking classes in a Java program that could be difficult to test or maintain.
- There are certain other metrics such as DC (member data count), MC (method count) etc. which showed significant correlation with subjective impressions of complexity for at least one subject. However, since these metrics do not have consistent support

in the experiments conducted, it may be safe to assume that these metrics are not very useful unless proved otherwise.

- The results of the experiments suggest that metrics can be used as predictors of difficulty in performing testing or modification tasks on Java classes. Hence metrics can be potentially very useful in the testing and maintenance stages of the software life-cycle.

6.3 Suggestions for Future Work

6.3.1 Further improvements to the Metrics Analyzer Tool.

- *Modifying the tool to handle multiple-file inputs.* Currently, the tool can only parse and generate metrics report one file at a time. In the future, the tool can be enhanced to support multiple-file inputs.
- *Improving the User-Interface of the tool.* Currently, the tool has a simple Graphical User Interface implemented using Java Development Kit (JDK) version 1.0.2. The user interface can be enhanced and made more sophisticated using the new JDK version 1.1 or other later versions.
- *Storing the metrics data in a Relational Database Management System.* Currently, all the metrics data obtained by parsing a Java source program is held and operated on in the main memory. This does not affect the performance of the tool for small programs. But with large programs, this could adversely affect the performance of the tool. Hence it would be a good idea to store this information in a Relational Database Management System (RDBMS) with sophisticated query facilities. In an industrial setting, the long-term collection of historic metrics data will definitely need a scalable and robust technology such as an RDMBS.

6.3.2 Further Investigation and Refinement of the Metrics Suite

As the validation experiments indicate, the external references metrics seem to be very good indicators of the complexity of Java programs. Hence, it would be rewarding to further investigate and refine these metrics. Since these metrics provide a measure of the degree of interaction or coupling that a class has with other classes, it would be very useful to explore other metrics that provide a more comprehensive measure of this metric. As an example, the current metrics only capture the *fan-out*, or the number of references of messages that go out of a class. For each class, it would also be interesting to know the number of incoming messages or references, the *fan-in*.

The experiments also indicate that an inheritance metric, the Depth of a class in its inheritance tree, has strong significant correlations with subjective impressions of complexity. Hence, it would be interesting to further investigate composite metrics based on inheritance and external references.

Finally, in this research, we only conducted experiments to validate the class-level metrics for Java. Further work could involve defining and experimentally validating metrics at the inheritance hierarchy level or project level.

References

[1] D. Tegarden, S. Sheetz and D. Monarchi, "Effectiveness of Traditional Software Metrics for Object-Oriented Systems", Proceedings of the 25th HICSS, 1992.

[2] A. Lake and C. Cook, " Use of factor analysis to develop OOP software complexity metrics", in Proc. 6th Annual Oregon Workshop on Software Metrics. Silver Falls, Oregon, 1994.

[3] M. Hamza, B. Lees et al., "The Extension of Software Metrics in Object-Oriented Development", in Proceedings of the 3rd International Conference of Software Quality Management, Seville, Spain, 1995.

[5] F. Abreu and R. Carapuca, "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework", Journal of Systems and Software, 26, pp87-96, 1994.

[6] J. Bieman and S. Karunanidhi, "Candidate Reuse Metrics for Object-Oriented and Ada Software", IEEE Transactions on Software Engineering, pp120-128, 1993.

[7] C. Chung and M. Lee, "Inheritance-Based metric for complexity analysis in object-oriented design", Journal of Information Science and Engineering, 8(3), pp-431-447, 1992.

[8] S. Henry and W. Li, "Maintenance metrics for the Object-Oriented Paradigm", in Proc. First International Software Metrics Symposium, Baltimore, MD, IEEE Computer Science press,,1993.

[9] R. Hudli, C. Hopkins et al., "Software Metrics for Object-Oriented Design", in Proc. IEEE international Conference on Computer Design, VLSI in Computers and Processors, Los Alamitos, CA,, USA, IEEE, 1994.

[10] T. Korson and J. McGregor, "Technical criteria for the specification and evaluation of object-oriented libraries.", Software Engineering Journal, 7(3), pp85-94, 1992.

[11] Y. Lee, B. Liang, et al., "Some Complexity Metrics for Object-Oriented Programs based on Information Flow: a Study of C++ Programs", Journal of Information and Software Engineering, 1994 (10), pp21-50, 1994.

[12] H. Sneed, "Estimating the Costs of Object-Oriented Software", in Proc. Evolving Systems, Durham 1995, 9th European Workshop on Software Maintenance, Durham, UK, 1995.

[13] M. Shumway, "Measuring Class Cohesion in Java", TR CS-97-113, Colorado State University, Colorado, USA, 1997.

[14] B. Curtis, "The Measurement of Software Quality and Complexity", " Software Metrics: An Analysis and Evaluation", Cambridge, MA: The MIT Press, 1981.

[15] J. Lewis, S. Henry, D. Kafura,"An Empirical Study of Object-oriented Paradigm and Software Reuse", in proc. OOPSLA, pp. 184-196, 1991.

[16] S. Henry, M. Humphrey "A Controlled Experiment to Evaluate Maintainability of Object-Oriented Software", IEEE, 1990

[17] J. Walsh, "Preliminary Defect Data from the Iterative Development of a Large C++ Program", in proc. OOPSLA 1992, pp. 178-183

[18] V. Basili, L. Briand, W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", CS-TR 3443.0824, University of Maryland, Maryland, USA

[19] E. Weyuker, "Evaluating Software Complexity Measures", IEEE Transactions on Software Engineering, 14, 9, pp.1357-1365, 1988.

[20] S. Conte, H. Dunsmore, V. Shen, "Software Engineering Metrics and Models", Benjamin Cummins Publishing Company, 1986.

[21] T. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, SE-2, 4, pp. 308-320, 1976.