Proceedings of the 9th IASTED International Conference
Software Engineering and Applications (SEA 2008)
November 16 - 18, 2008 Orlando, Florida, USA

USING A COMBINATION OF MEASUREMENT TOOLS TO EXTRACT METRICS FROM OPEN SOURCE PROJECTS

Normi Sham Awang Abu Bakar, Clive Boughton
Department of Computer Science
Australian National University
Australia
normi@cs.anu.edu.au, clive.boughton@anu.edu.au

ABSTRACT
Software measurement can play a major role in ensuring the quality and reliability of software products. The measurement activities require appropriate tools to collect relevant metric data. Currently, there are several such tools available for software measurement. The main objective of this paper is to provide some guidelines in using a combination of multiple measurement tools especially for products built using object-oriented techniques and languages. In this paper, we highlight three tools for collecting metric data, in our case from several Java-based open source projects. Our research is currently based on the work of Card and Glass, who argue that design complexity measures (data complexity and structural complexity) are indicators/predictors of procedural/cyclomatic complexity (decision counts) and errors (discovered from system tests). Their work was centered on structured design and our work is with object-oriented designs and the metrics we use parallel those of Card and Glass, being, Henry and Kafura's Information Flow Metrics, McCabe's Cyclomatic Complexity, and Chidamber and Kemerer Object-oriented Metrics.

KEY WORDS
Complexity, CBO, fanout, parameters, post-delivery defects

1. Introduction

One of the most important objectives of software engineering is to improve the quality of software products. The quality of software can be defined in different ways but one of the most common definitions is the number of defects that arise in the final product [1], be it functional defects or programming defects, that can cause problems to users. Such 'quality' measures should be determined as early as possible during development, by using predictors of ultimate quality.

To establish measures that can predict quality (in this case post-delivery defects), it is important to undertake careful data collection [1]. Data collection is a challenging task especially when done across a diverse set of projects. Thus, the data collection process has to be done using a systematic plan to ensure that measures are defined unambiguously, that collection is consistent and complete, and that data integrity is protected.

This paper presents three available tools that can be used to collect metrics data from software products. We are currently using these tools to extract metrics data from several open source projects which are written in Java. The tools are: Resource Standard Metrics (RSM), JStyle and Chidamber and Kemerer Java Metrics (CKJM). The details of each tool will be discussed in Section 3 of this paper.

The open source systems that we are investigating have been downloaded from SourceForge.net. These systems have been divided into four different categories based on functionality, as well as the success of the systems in terms of numbers of downloads, development status and activity percentile. A few of these systems are listed in the most active project list in SourceForge.net [2].

2. Background

The open source software development community has grown enormously over the past few years. Open source systems are commonly accepted and successfully adopted/adapted into many organizations, and some of these systems have been used for mission critical purposes [1]. Therefore, it is very important to not only assess and validate the reliability and performance of these systems to help ensure that they fulfill their purpose, but also to provide developers with simple measures that will help them determine quality.

A study by Zhou and Davis [3] demonstrated that open source projects show similar reliability growth patterns to that of proprietary software projects. This potentially means that even though open source development methodologies are usually seen as different from the proprietary software development methodologies, they have similar properties that can be used as indicators of software quality. Paulson et al. [4] have conducted a study to compare several aspects of system development between open source and closed-source projects. They have found that creativity is more widespread in open source projects and defects are found and fixed more rapidly compared to closed-source projects. Another study conducted by Mockus et al. [5]

investigated the claim that open source style software development has the capability to complete successfully and in most cases, even displace traditional commercial development methods. They had looked into the aspects of developer participation, core team size, code ownership, productivity, defect density and problem resolution intervals in order to understand the methods used for software development in open source projects. In their paper, Refenc et al. [6] discuss a framework called "Colombus" which they used to calculate the object-oriented metrics for illustrating how fault-proneness detection from the Mozilla open source web and e-mail suite can be done.

Other researchers [7], [8], [9], [10], have discussed the usage of tools to support software measurement programs. In their work, Tian et al. [10] used several tools to carry out their software measurement, analysis and improvement activities. For data gathering, they used: IDSS, CMVC, TestLog and SlaveDriver, and for analysis and presentation, they used S-PLUS. Kempkens et al.[8] used several tools, such as COSMOS, MOODKIT and WISE to automate metrics data collection. In their paper, AlGhamdi et al. [7] presented three existing tools: Brook's and Buell's tool, a "Tool" for analyzing C++ code (TAC++), and an object-oriented metrics gathering tool (OOMetDaGa) and compared them with a tool which was developed by themselves.

Whilst there has been much work in measuring various aspects of open source software, as with proprietary software there are few measurements done relating design quality to defects of any sort – especially for the purpose of predicting defect numbers or severity.

Our current work is mainly based on previous work done by Card and Glass [11], who studied eight systems written in FORTRAN (RATFOR) in the Software Engineering Laboratory, and sponsored by NASA Goddard Space Flight Center (GSFC). They hypothesized that the complexity of a system can be broken down into 3 main components, data and structural complexity (established as part of design) and procedural complexity (established as part of implemenetation). They then found that the more complex the design of a particular system, the more errors it possessed, independent of size of system.

3. Measurement Tools

3.1 Chidamber & Kemerer Java Metrics (CKJM)

The program "*ckjm*" calculates Chidamber and Kemerer object-oriented metrics by processing the byte code of compiled Java files. The program calculates the following six metrics proposed by Chidamber and Kemerer, for each class:

- WMC: Weighted methods per class
- DIT: Depth of Inheritance Tree
- NOC: Number of Children
- CBO: Coupling between object classes
- RFC: Response for a Class
- LCOM: Lack of cohesion in methods
- Ca: Afferent couplings
- NPM: Number of public methods

"*ckjm*" [12] is freely available as open source software and the current version is 1.8 (at the time of writing this paper).

3.2 JStyle

JStyle [13] is another tool for collecting software metrics including the Chidamber and Kemerer object-oriented (OO) metrics. This tool supports the measurement of Java software and has four levels of object-oriented metrics: project level, module level, class level and method level. In this paper, we are interested in looking at metrics at the class level as listed below:

- Depth of Inheritance (DIT)
- Number of Children (NOC)
- Response For Class (RFC)
- Lack of Cohesion in Methods (LCOM) Chidamber-Kemerer
- Lack of Cohesion in Methods (LCOM) Li-Henry
- Lack of Cohesion in Methods (LCOM) Henderson-Sellers
- Fan-in (FI)
- Fan-out (FO)
- Intra-Package Fan-In (PFI)
- Intra-Package Fan-out (PFO)
- Inter-Package Fan-in (IFI)
- Inter-Package Fan-out (IFO)

3.3 Resource Standard Metrics (RSM)

The third tool, Resource Standard Metrics [14], is a source code metrics and quality analysis tool for systems written in C, ANSI C++, C# and Java source code across operating systems. This tool can provide measurements at several levels of a system, for example, project level, package level, file level, class level, interface level and method level.

In this paper, we highlight the (RSM) metrics for Java source code, as listed below:

- Total number of classes
- Inheritance Tree
- Number of Base Classes
- Number of Derived Classes
- Maximum and Average Inheritance Depth
- Maximum and Average Number of Child Classes
- Public, private, protected data attributes
- Public, private, protected methods
- LOC Lines of Code
- eLOC (Effective LOC)
- lLOC (Logical Statements LOC)
- Number of Input Parameters
- Number of Return Points
- Interface Complexity (Parameters + Returns)

- Cyclomatic Complexity Logical Branching
- Class Complexity (Interface + Cyclomatic)
- Total Quality Profile

Table 1 is a summary of the main characteristics of the three tools that we have used to produce the essential metrics of interest in our work.

Table 1. Tools Characteristics

Comparison criteria	Tools		
	CKJM	**JStyle**	**RSM**
Supported language	Java	Java	C, ANSI C++, C#, Java
Number of supported metrics	8	66	174

4. Metrics Collection

The measurements we use in our work are obtained from the combination of tools described above. The software base to which we apply these tools consists of 36 open source projects from www.sourceforge.net and all projects chosen are written in Java. The reason we chose to study open source systems is that a random choice can easily be made for the categories of projects of interest. Also it is relatively easy to identify an appropriate set of projects similarly sized (or not) that are actively supported and can provide adequate information regarding the characteristics we wish to examine/evaluate.

The work reported in this paper is based on the work of Card and Glass to determine whether there are similar correlations that they report concerning the relationship between design quality and errors for different styles of language. We are interested in investigating whether there is a general correlation between post-delivery defects and system design complexity, by studying object-oriented measures relating to data, structural and procedural complexity and comparing them with discovered, post-delivery defects.

According to Card and Glass [11], system complexity metrics are based on the structured design modularity principles of coupling and cohesion. It uses both intramodule and intermodule complexity to arrive at a system complexity metric, for which the initial equation is:

$$Ct = St + Dt$$

where Ct = system complexity
St = structural (intermodule) complexity
Dt = data (intramodule) complexity

In this paper, we want to show the typical results of these complexity analyses for ten open source projects namely: DataCrow (Project 1), Mars (Project 2), HTMLParser (Project 3), SCAM (Project 4), Saxon (Project 5), SchemaSpy (Project 6), Eclipse Checkstyle (Project 7), JasperReports (Project 8), Freemind (Project 9) and Cewolf (Project 10). The metrics that are appropriate to our work are Fanout (FO) and Coupling Between Object Classes (CBO) to represent structural complexity; Average Cyclomatic Complexity to represent procedural complexity; and Average Number of Parameters to represent data complexity.

Average Fanout for each of the ten projects are shown in Figure 1. According to Henry and Kafura [15], Fanout is a number of local flows out of a module plus the number of data structures that are used as output. They stated that modules with low Fanout have low complexity. In our work, we investigated Fanout of classes and tried to see the correlation between Fanout (structural complexity) of the systems with post-delivery defects. Figure 1 shows that three projects: Mars (Project 2), SchemaSpy (Project 5) and Saxon (Project 6) exhibit high Fanout values, which potentially means that there has been inadequate factoring performed in the project.

In Figure 2, the Average Cyclomatic Complexity (procedural complexity) of each of the ten systems are presented. Cyclomatic Complexity is the measure of the number of control flows within a module [16]. According to McCabe, the greater the number of paths through a module, the higher the complexity. McCabe has suggested that, on the basis of empirical evidence, when Average Cyclomatic Complexity per module of a project exceeds 10, the project may be problematic [16]. In our findings, only two projects: DataCrow (Project 1) and Cewolf (Project 10) have Average Cyclomatic Complexity per module less than 10. Hence, according to McCabe, the other eight projects need to be investigated further to find out the reasons for the high values of Average Cyclomatic Complexity. In this work, the formula for Average Cyclomatic Complexity is given as:

Average Cyclomatic Complexity = Total (System) Cyclomatic Complexity/Number of Classes

Figure 3 shows the Average Number of Parameters (Data Complexity) per class for each of the ten projects. The internal complexity (Cyclomatic Complexity) of a module represents the amount of work it must perform. Card and Glass showed that Data Complexity (Number of Parameters) was a predictor of Procedural Complexity (Cyclomatic Complexity). Figure 4 illustrates that such a relationship holds for the ten Java projects we studied.

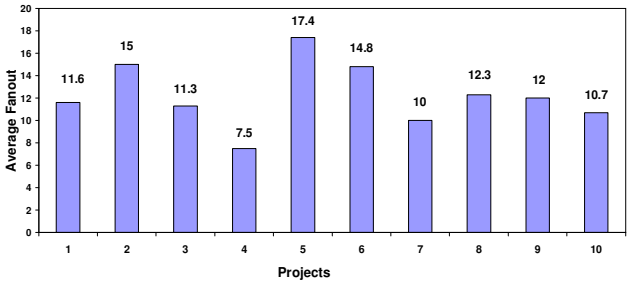

Figure 1. Average Fanout

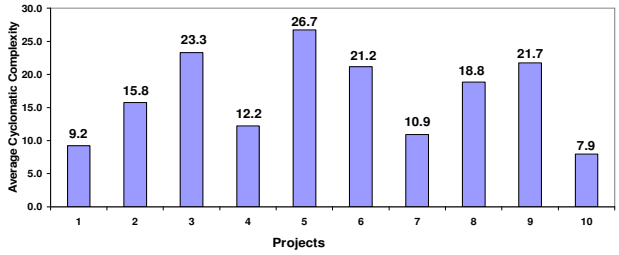

Figure 2. Average Cyclomatic Complexity

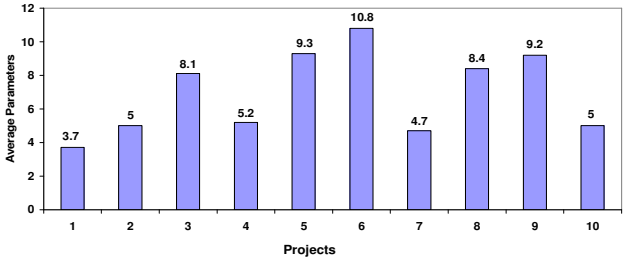

Figure 3. Average Parameters

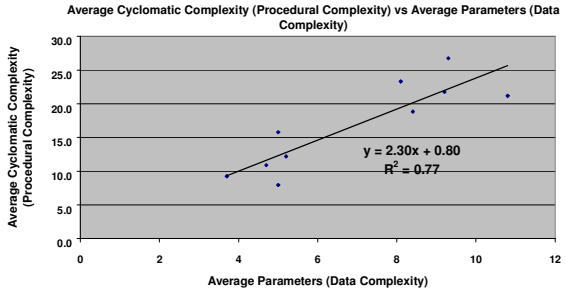

Figure 4. Procedural Complexity vs Data Complexity

In Figure 4, the linear fit for the ten systems is represented by the equation: *Decisions per Class = 2.25 x D + 1.03*. This means that for each unit of data complexity (number of parameters), 2.25 decisions must be made to implement the required function and in addition, the average class includes a base of 1 decision (perhaps) not directly related to the data function of the class. This relationship closely aligns with the findings of Card and Glass [11], so that if a design is created that contains little in the way of detail for the internals of each class (in this case for Java) but does show expected parameters for each class (indicating data complexity), the likely degree of procedural complexity can be predicted and additionally the potential effort required to implement each class. Such a measure can help to produce an effective design before too much detail is added.

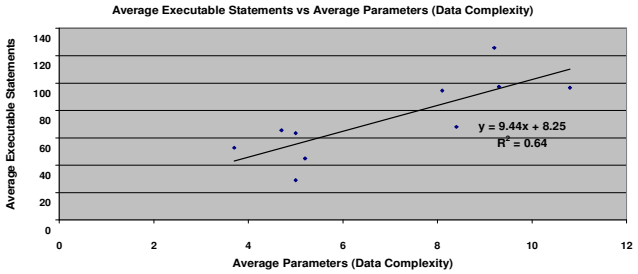

Figure 5. Executable Statements vs Data Complexity

Figure 5 demonstrates that data complexity effectively predicts the size of a class in terms of executable statements. The linear fit for the ten systems is: *Executable Statements = 9.44 x D + 8.25*, which means that an increase of one unit of data complexity increases program size by around nine executable statements. The correlations between these two variables is good (R^2 = 0.64). This result is consistent with the findings of Card and Glass [11] for RATFOR (4.1 D + 24.5), who show that data complexity is a good predictor of subroutine size. This result also provides insight into an estimate of effort to create each module/class.

5. Tools Comparison

In this paper, we compare the results of metrics of particular interest to us as produced from three different tools. Our original intention in using the three tools was based on a belief that the three particular metrics in which we were interested could easily be verified as accurate provided there existed some direct relationship between the possible varying ways in which two or more of the tools might produce the metrics. For some other metrics, such as Number of Classes, the three tools have produced the same results. However, for the metrics of interest to us that relate to data complexity, procedural complexity and structural complexity as defined by Card and Glass [11], it is difficult to obtain any such verification.

In order to measure the metrics relevant to our work, we need to use the three tools and combine the results to arrive at substantial conclusions empirically. Because we are analyzing systems constructed using object-oriented languages, one of the key metrics is Coupling Between Objects (CBO) which we believe has parallels to Card and Glass's Fanout. Only one tool (CJKM) can provide this metric. However Jstyle does purport to provide actual Fanout.

Since we are measuring object-oriented systems, we thought CBO could be used as a measure of structural complexity. It was also thought that we could validate the CBO metrics with the actual Fanout obtained from JStyle. The plot of Average CBO and Average Fanout is shown in Figure 6 below. From the plot, the linear fit for the systems is *CBO = 0.31 x Fanout + 3.18* which means that CBO is about a quarter of the Fanout. If Fanout is zero, we obtain a constant CBO of (approx.) 3 which indicates that CBO is more complex than Fanout because it includes more variables like method calls, inheritance, arguments, return types and exceptions, while Fanout is a simple measure of (use) dependency between modules/classes. We had expected to see a close relationship between CBO and Fanout, but we found that the correlation is quite weak (R^2=0.1) and thus needs further investigation.

However, when we plot Total CBO against Total Fanout for the projects, we find that the correlation between the two variables is very strong (R^2 = 0.97) and the linear fit is *CBO = 0.64 Fanout*. This is shown in Figure 7 below and we believe that the poor correlation

that was shown in Figure 6 might be influenced by other properties of the individual systems, i.e. high use of inheritance, amount of interface code etc. The degree to which such properties influence the correlation between CBO and Fanout is still being studied.

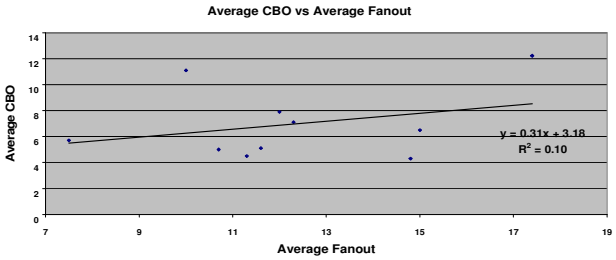

Figure 6. Average CBO vs Average Fanout

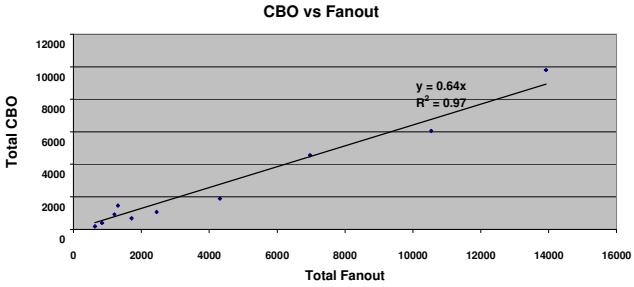

Figure 7. Total CBO vs Total Fanout

Another interesting finding is that System Complexity seems to be a good predictor of Total Post-Delivery Defects in the systems. This is depicted in Figure 8 below. The linear fit of these systems is: *Total Defects = 33.85 System Complexity – 315.09*. We have found that the correlation between these two variables is good (R^2=0.64) and tends to support the hypothesis that System Complexity can predict the Total Post-Delivery Defects in object-oriented systems. This relationship corresponds to the type of relationship found by Card and Glass [11] (R^2=0.83) and Error Rate = 0.4 Complexity – 5.2 (for pre-delivery defects).

Figure 9 demonstrates that unlike Card and Glass's results, Average Class Size may have some effect on Defect Rate, as shown by the correlation (R^2 = 0.51) for the parabola ($0.02x^2 - 2.71x + 95.19$). From the plot in Figure 9, it seems that smaller systems have higher defect rates than larger systems. As the system size rises, Defects/KLOC drop gradually before rising again, indicating that there maybe some optimal class size for lowest defect rates.

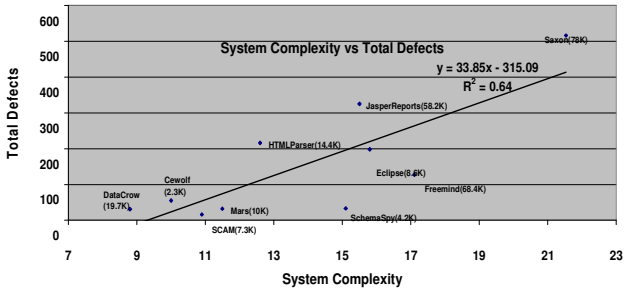

Figure 8. System Complexity vs Total Defects

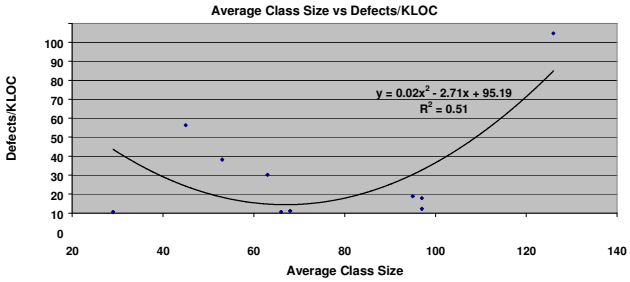

Figure 9. Average Class Size vs Defects/KLOC

6. Tools Validation

Using a combination of multiple tools to extract metrics from software projects has its own challenges, especially to ensure that data collected is correct across all tools. This section will discuss the process of validating the data collected using the three tools: CKJM, JStyle and RSM.

1) Validating similar metrics
Since some tools can produce similar metrics, the first step is to check whether such metrics obtained by each tool have the same values. For example all tools can produce Number of Classes, therefore, it is important to validate that all tools produce the same results for this measure. The same process goes for other metrics that are common between the tools.

2) Further analysis
We have managed to obtain similar results for metrics such as Number of Classes, Number of Methods and Number of Parameters, so we are confident that the tools have followed the same algorithm or process for at least three metrics. However, this is not the case for Cyclomatic Complexity. There are slight differences in the results for Cyclomatic Complexity produced by RSM and JStyle. After a thorough checking process, we managed to confirm that the results produced by JStyle adhere to the definitions given by McCabe [16], whereas the ones produced by RSM have additional properties that were not included in McCabe's definition. However, after removing the effects of the additional properties, the results were the same.

3) Other properties

We have explored the relationship between CBO and Fanout to see whether CBO can represent structural complexity in object-oriented systems as opposed to Fanout (typically used in structured design systems). Referring to Figure 6, the correlation between CBO and Fanout seems quite poor (R^2 = 0.1) and to find explanations to this unexpected result, we have decided to do further investigations on other properties of the systems under study. Currently, we are undertaking further analysis on system properties such as comparing CBO and Fanout for different types of systems; frequency graphs of CBO and Fanout distribution across a system; the proportion of 'inheritance' to 'use' relationships; the proportion of GUI and other interface code; and several other properties. It is hoped that once the analysis of these properties is completed, we will have a better understanding of the validity of using CBO and Fanout depending on system properties.

7. Conclusion and Further Work

The three tools (CJKM, RSM and JStyle) can be used to collect a variety of metrics to suit the needs of different measurement objectives. In our work, we used the tools to gather data from several open source projects and we found that all tools have their own strengths and weaknesses but no one tool produces all the metrics that we need in our work. In other words, although these tools can collect different sets of metrics, they must be used to complement each other in producing the particular metrics required for our work.

The main contribution of this paper is to provide insights in using a combination of tools to extract metrics from software systems and the processes needed to ensure the results obtained from (at least) these tools are valid and trustworthy. Not only it is possible to encounter different 'implementations' of metrics among tools, but also there can be apparent inconsistency confounding and/or unexpected relationships among what seem to be similar metrics.

Further work is needed to investigate the relationship between CBO and Fanout to get a better understanding of the two metrics. We will continue to analyze the metrics obtained through these tools and explore further the properties of the projects under study. Moreover, in order to determine the consistency of our results further, snapshots of infrequent system releases is necessary. We will also look into the possibility of using cumulative defects of the systems as a measure of system defects and not only consider the defects reported for a particular release. In addition, the relationship between defect rate and average class size will be investigated further.

References

[1] N. E. Fenton and S. L. Pfleeger, *Software metrics: A rigorous and practical approach* (Boston, MA: PWS Publishing Company, 1997).

[2] SourceForge: www.sourceforge.net

[3] Y. Zhou and J. Davis, Open source software reliability model: An empirical approach, In: *Proceedings of Open Source Application Spaces: Fifth Workshop on Open Source Software Engineering (5-WOSSE)*, St. Louis, MO, USA, 2005.

[4] J. W. Paulson, G. Succi and A. Eberlein, An empirical study of open-source and closed-source software products, *IEEE Transaction on Software Engineering*, *30*(4), April 2004, 246-256

[5] A. Mockus, R.T. Fielding and J. D. Herbsleb, Two case studies of open source software development: Apache and Mozilla, *ACM Transaction on Software Engineering and Methodology*, *11*(3), July 2002, 309-346

[6] R. Ferenc, I.Siket and T. Gyimothi, Extracting facts from open source software, In Proceedings, *20th International Conference on Software Maintenance* (*ICSM 2004*), Chicago Illinois, USA, 2004

[7] J. AlGhamdi, M. Elish and M. Ahmed, A tool for measuring inheritance coupling in object-oriented systems, *Information Sciences*, *140*(2002), 2002, 217-227.

[8] R. Kempkens, P. Rosch, L. Scott and J. Zettel, Instrumenting measurement programs with tools, *LNCS 1840*, *Springer-Verlag Berlin Heidelberg 2000*, 2000, 353-375

[9] F. G. Wilkie and T.J. Harmer, Tool support for measuring complexity in heterogeneous object-oriented software, In Proceedings, *Proceedings of International Conference on Software Maintenance (ICSM'02)* , 2002

[10] J. Tian, J. Troster and J. Palma, Tool support for software measurement, analysis and improvement, *Journal of Systems Software*, *39*, 1997, 165-178

[11] D. N. Card and R. L. Glass, *Measuring software design quality* (New Jersey, USA: Prentice Hall, 1990).

[12] CKJM: www.spinellis.gr/sw/ckjm/

[13] JStyle: www.mmsindia.com/jstyle.html

[14] RSM: www.msquaredtechnologies.com/

[15] D. G. Kafura and S. M. Henry, Software quality metrics based on interconnectivity, *Journal of Systems and Software*, *3* (1982), 1982, 121-131.

[16] T. J. McCabe, A complexity measure, *IEEE Transactions on Software Engineering*, *2*(4), December 1976, 308-320