Refactoring – Does it improve software quality?

Konstantinos Stroggylos, Diomidis Spinellis
Athens University of Economics and Business
Department of Management Science and Technology
Patision 76, 10434 Athens, Greece
circular,dds@aueb.gr

Abstract

Software systems undergo modifications, improvements and enhancements to cope with evolving requirements. This maintenance can cause their quality to decrease. Various metrics can be used to evaluate the way the quality is affected. Refactoring is one of the most important and commonly used techniques of transforming a piece of software in order to improve its quality. However, although it would be expected that the increase in quality achieved via refactoring is reflected in the various metrics, measurements on real life systems indicate the opposite. We analyzed source code version control system logs of popular open source software systems to detect changes marked as refactorings and examine how the software metrics are affected by this process, in order to evaluate whether refactoring is effectively used as a means to improve software quality within the open source community.

1. Introduction

Software metrics have been proved to reflect software quality, and thus have been widely used in software quality evaluation methods [6]. The results of these evaluation methods can be used to indicate which parts of a software system need to be reengineered. The reengineering of these parts is usually performed using refactoring. Refactoring is defined as "the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure" [30].

The abundance of available tools for collecting software metrics and performing automated refactoring on source code simplifies maintenance. This means that more developers and software engineers use refactoring consciously as a means to improve the quality of their software. However, as this study suggests, the expected and actual results often differ. Although people use refactoring (or claim to do so) in order to improve the quality of their systems, the metrics indicate that this process often has the opposite results.[1]

2. Metrics and software quality

Software metrics provide a means to extract useful and measurable information about the structure of a software system. This explains why the first metrics like LOC (Lines Of Code) appeared very early. Today the software engineer has a very large list of metrics at his disposition in order to gain the insight required for understanding and evaluating the structure and quality of a system. This has created the need to determine which set of metrics is most appropriate for each environment [39]. Popular metrics suites are Order of Growth (Big O notation) [13], Halstead's Complexity Measures [19], McCabe's Cyclomatic Complexity [26, 27, 28] and Maintainability Index [38]. For object-oriented systems the metrics used more commonly include the suites proposed by Henry and Kafura [20, 21], Chidamber & Kemerer [11, 12], Li & Henry [24], Lorenz & Kidd [25] and Briand et al. [7, 8].

Several studies attempt to correlate software metrics with quality [35, 23, 34] or validate the significance of the various metrics proposed in the literature[9]. Others use metrics to predict the fault-proneness of classes from the early development phases, or evaluate their effect on the maintenance effort required to keep a system up to date with changing requirements, both of which are indicators of a system's quality[4, 24]. These methods have been used to reliably detect the parts of a system that are most likely to suffer from errors or exhibit maintenance difficulty and thus need to be reengineered.

Probably the most influential factor for software quality is its design. A good design will allow a software system to evolve with little effort and less money. Object oriented systems expose this behavior more than ones written in procedural languages, since many powerful mechanisms such as

[1]This research has been funded within the frame of the Operational Programme "COMPETITIVENESS", measure 8.3.1 (PENED), and is co-financed by European Social Funds (75%) and by national sources (25%)

inheritance, polymorphism and encapsulation are available. Therefore by evaluating the quality of the design of a system one can estimate its overall quality. Various studies attempt to correlate attibutes of the design of a system (often represented by design complexity metrics) to its quality. They are targeting mostly towards defect density and maintenance effort or time, and provide predictive models based on the values of the metrics for these attributes [16, 2, 3]. Others define formal models for object oriented design and use them to formally define established object-oriented metrics, so as to ease the automated design evaluation [10, 31]. It is therefore clear that metrics can be used to detect whether a system's quality suffers and needs to be reengineered, as well as identify which of its parts need to be refactored.

2.1. Relation of refactoring to metrics and software quality

Refactoring is considered as one of the most important means of transforming a piece of software in order to improve its quality. Its aim is to decrease the complexity of a system at design and source code level, allowing it to evolve further in a low-cost manner by ensuring the developers' productivity and leaving less room for design errors[29]. The problem faced by software engineers is discovering where to apply refactorings. Fowler [17] states that the detection of such problematic areas is based on human intuition and descibes the notion of 'bad smells' in code.

Fortunately, among other methods, software metrics can be used to identify areas that would benefit from refactoring. Tool support is necessary to assist the human intuition in this decision-making process in an efficient manner. Simon et al. in [33] employ distance-based metrics to detect targets for four common refactorings. Joshi and Joshi in [22] introduce two new microscopic metrics that they claim are best suited for fine-grained decisions about needed refactoring actions.

Software quality can be described as the conformance to functional requirements (related to correctness) and non functional requirements (NFRs), which are related to characteristics described in the ISO-9126 standard (reliability, usability, efficiency, maintainability and portability) [1]. Refactoring is by definition supposed to improve the maintainability of a software product, but it its effect on other quality aspects is unclear.

In order to determine which refactorings have a positive effect on quality it is necessary to analyze the dependecies between NFRs, as - more often than not - a reengineering decision affects more than one of its aspects. Such an attempt is described in [37]. This study, which also presents metrics for evaluating the effect of refactorings on a system, establishes an explicit relation between design decisions and quality requirements, by representing NFRs as soft-goals and examining the dependencies between them. The work in [36] also employs a modeling of soft goals and their interdependencies as graphs. However it takes a step further by associating them with specific transformations which are not applied unless an evaluation procedure suggests that the desired qualities will be met in the reengineered system. Yu et al. [40] also use such graphs combined with metrics in order to select the order in which refactorings should be performed, taking into account various tradeoffs between code complexity and performance.

2.2. Refactoring and software evolution

An extensive research has been performed under the name of *refactoring detection* in detecting refactorings between different revisions of a software system. This technique is useful for examining the evolution of a system but it does not always convey a clear image about the reason that led the developers to perform such changes.

A primary target of refactoring is the minimization of code duplication which usually occurs from copy-and-modify operations. Refactorings such as Extract Method help in minimizing the number of locations requiring modification when a change is performed. To facilitate the maintenance process, a clone detection-based method to automatically detect refactorings that occured between two consecutive versions of a component is proposed in [15]. Another method described in [18] is the detailed analysis of function call relations and origin analysis. Others employ data mining techniques to detect high-level similarities in code [5]. An older study [14] proposes heuristics for detecting refactorings by calculating metrics over successive versions of a system.

3. Experiments

Most of the studies presented in the previous section do not correlate the evolution of a system with the change in metrics. In this study we attempt to show how refactoring has affected metrics in open source software. Instead of using one of the proposed techniques to detect the refactorings performed between consequent revisions, the commit logs were used as a source of information. This means that we trust the developers in performing refactorings and documenting them in the system's change history. In essence we take into account only reengineerings that the developers mark as refactorings.

The whole process, which is straightforward and can be easily automated, is depicted in Fig.1 and can be described as a sequence of simple steps, as follows:

1. Obtain the commit log from the source code repository of the software system being examined

2. Search through the log entries for mentions of words stemming from the verb 'refactor' (e.g. refactoring, refactored, etc.)

3. Create a list *Lrev* containing pairs (Rev_{start}, Rev_{end}) of consecutive revision numbers of the system between which a refactoring was performed.

4. Enumerate through the items contained in *Lrev*. For each pair (Rev_{start}, Rev_{end}) of revisions in *Lrev* use the 'diff' command of the source code revision control system used for the system being examined in order to obtain a list of source files $Lfiles_{start-end}$ that were modified (added, removed or changed) between Rev_{start} and Rev_{end}

5. Enumerate again through the items contained in *Lrev*. For each revision Rev_{start} and Rev_{end} in each pair, download (check out) the corresponding revision of the source code of the system being examined from its repository and compile it.

6. Once the compilation of a revision R_{cur} is complete, enumerate through the list of files $Lfiles_{start-end}$ where R_{cur} is one of the revisions R_{start} and R_{end} of the pair currently examined. For each file contained in the list $Lfiles_{start-end}$ use a metrics tool to extract metrics from either the source code file itself or the compiled class file and store the results in a text file or other persistent storage for further processing

7. Enumerate through the list of files $Lfiles_{start-end}$ analyzed for the current pair (Rev_{start}, Rev_{end}) and examine how the various metrics were affected between the revisions Rev_{start} and Rev_{end}

Note that the compilation in step 5 does not necessarily have to complete successfully. As long as the classes or files in $Lfiles_{start-end}$, their dependencies and the ones depending on them compile the analysis can still be performed. This has allowed us to get away with commits that break the build for reasons unrelated to refactorings.

The process described above was used for the evaluation of the effect of refactorings on three object-oriented and one procedural system. The tool used for the object oriented systems is the latest version of ckjm [2], an open source tool written by this paper's second author, which calculates the metrics proposed by Chidamber&Kemerer as well as Ca (Afferent Coupling, the number of other packages depending upon a class) and NPM (Number of Public Methods of a class), by processing the bytecode of compiled Java classes.

As the experiments were being conducted it became evident that in many cases the commits performed by the developers would break the system's build. In order to cope with this situation another tool was used to calculate the metrics for the examined classes, the (misnamed) C and C++ Code Counter (CCCC)[3]. CCCC can process C, C++ and Java source files directly, and it calculates 4 out of the 6 metrics of the Chidamber&Kemerer suite, as well as McCabe's Cyclomatic Complexity (the number of linearly independent paths), so it was used both whenever the compilation of a revision failed and for the procedural system examined.

The metrics examined also include WMC(Weighted Methods per Class), DIT (Depth of Inheritence Tree), NOC (Number Of immediate Children subclasses), CBO (Coupling Between Objects, a count of the non-inheritance relations with other classes), RFC (Response for a class, the sum of the number of methods of the class itself and all other methods it calls) and LCOM (Lack of Cohesion of Methods, expresses the similarity of methods). As shown in [32] big values in these metrics indicate possible problems.

3.1. A procedural example - Apache httpd

A big percentage of software systems in use today is not written in object oriented languages. A prime example of this is operating systems, such as the kernels of the GNU/Linux and BSD families. In the context of this study, the popular Apache Software Foundation HTTP server[4] was chosen as a sample for numerous reasons besides its popularity[5]. It is one of the most actively developed open source projects with many contributors from all over the world. Additionaly, the description page of the latest branch of the system (version 2.2) states that one of the major core enhancements was the refactoring of certain modules (namely the bundled authentication and authorization modules). This meant that it wouldn't be necessary to examine specific consecutive revisions, but one could do a more macroscopic examination of these modules instead.

The versions of the system that were studied were 2.0.59 from the stable branch and 2.2.3 from the experimental branch. Besides the renaming of the examined modules, which is not of interest for this study as it does not affect metrics, one module (mod_auth) was split into two separate ones (mod_auth_basic and mod_authn_file), which were the ones examined in detail.

As one can see in Figure 2, the McCabe Complexity has increased about 30% for mod_auth due to the fact that it was broken down into two modules, although the overall module architecture has been simplified. On the other hand, the mod_auth_digest module displays slightly improved met-

[2]Available at `http://www.spinellis.gr/sw/ckjm/`

[3]Available at http://cccc.sourceforge.net/

[4]Available at `http://httpd.apache.org/`

[5]Netcraft Ltd. Web Server Survey. Available at `http://news.netcraft.com/archives/web_server_survey.html`

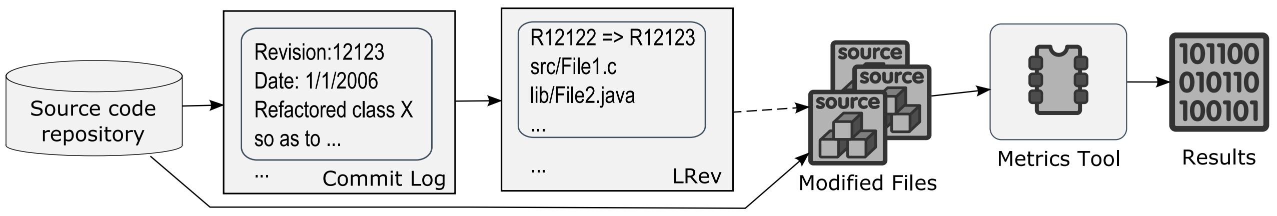
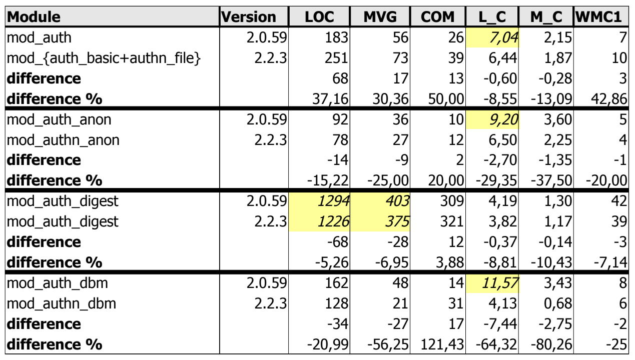


Figure 1. The process used to evaluate the effect of refactorings



Module	Version	LOC	MVG	COM	L_C	M_C	WMC1
mod_auth	2.0.59	183	56	26	*7,04*	2,15	7
mod_{auth_basic+authn_file}	2.2.3	251	73	39	6,44	1,87	10
difference		68	17	13	-0,60	-0,28	3
difference %		37,16	30,36	50,00	-8,55	-13,09	42,86
mod_auth_anon	2.0.59	92	36	10	*9,20*	3,60	5
mod_authn_anon	2.2.3	78	27	12	6,50	2,25	4
difference		-14	-9	2	-2,70	-1,35	-1
difference %		-15,22	-25,00	20,00	-29,35	-37,50	-20,00
mod_auth_digest	2.0.59	*1294*	*403*	309	4,19	1,30	42
mod_auth_digest	2.2.3	*1226*	*375*	321	3,82	1,17	39
difference		-68	-28	12	-0,37	-0,14	-3
difference %		-5,26	-6,95	3,88	-8,81	-10,43	-7,14
mod_auth_dbm	2.0.59	162	48	14	*11,57*	3,43	8
mod_authn_dbm	2.2.3	128	21	31	4,13	0,68	6
difference		-34	-27	17	-7,44	-2,75	-2
difference %		-20,99	-56,25	121,43	-64,32	-80,26	-25

Figure 2. The effect of refactoring on Apache

rics, which can be explained by the fact that the digest mechanism has been redesigned in the 2.2 branch. Finally, the metrics for the other modules appear to have improved significantly. This is mainly caused by changes in other parts of the system and the module architecture in general, which allow the code for these modules to become simpler. The effect is less visible in mod_auth_digest because it is much more complicated than the others, so the metrics difference is relatively smaller.

3.2. Object Oriented Libraries

The method described was used to evaluate the effect of refactoring on the metrics of three popular open source object oriented libraries. The libraries selected were Apache Software Foundation's logging library for Java (Log4J), MySQL's JDBC Driver (MySQL Connector/J) and JBoss Hibernate Core. These projects were chosen because they have been around and studied for years, they are actively being developed, they have strong communities of users supporting them and are very popular among developers.

3.3. Apache Log4J

Log4J is an open source logging library for Java developed by Apache Software Foundation. It is a medium-sized component, as the latest revision weighs in at about 89KLOC[6]. For the purpose of this study, a set of 30 revisions in 16 pairs was selected (some revisions participated in two consequent pairs). The metrics for 4 of these pairs that compiled cleanly were calculated using ckjm, whereas CCCC was used for the rest. The experimental results are presented in Figure 3.

These results imply that the refactorings overall had little impact on the calculated metrics - especially the NOC and DIT were not affected at all, which makes sense considering that the refactorings examined did not include transformations that affect the class hierarchy except for two cases.

The experimental results also suggest that there was overall a non-trivial increase in the value of the MVG and RFC metrics. While the CBO displays a small increase, we noticed a significant increase in the values of the RFC and Ca, which approaches 10% and 11% respectively. These values may suggest that each reengineered class ended up with more (diverse) responsibilities. Given that the increase in Ca was accompanied by an increase in LCOM it may be safe to suggest that the methods contained in the classes refactored became less related to each other - and perhaps with their initial purpose.

3.4. MySQL Connector/J

MySQL Connector/J is an open-source JDBC driver for the MySQL server [7]. It is a medium-sized component, as the latest stable revision weighs in at about 89KLOC. For the purpose of this study, a set of 10 revisions in 8 pairs was selected. The metrics for the files affected between the various revisions were calculated using CCCC. The results calculated are presented in Figure 3.

As one can see, these results do not seem to agree with the ones of Log4J. This could be due to various reasons. For example it could have occured because diferrent types of refactorings were applied to the source code, which may affect the calculated metrics in a different way. However since not all the Chidamber & Kemerer metrics are calculated by CCCC and the set of revisions examined is small the calculated results may not be representative.

[6]Available at `http://logging.apache.org/log4j`
[7]Available at `http://dev.mysql.com/`

(a) Hibernate	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	NPM	MVG
Avg diff	0,51	-0,01	0,01	0,51	0,21	-3,70	0,13	0,34	-0,14
Avg diff %	10,3%	2,4%	4,0%	7,8%	7,3%	17,9%	15,6%	10,2%	4,0%
Avg Rstart	23,18	1,32	0,36	16,57	64,83	538,46	2,29	16,73	20,64
Avg Rend	23,69	1,32	0,37	17,09	65,03	534,76	2,43	17,06	20,49
Median Rstart	12	1	0	12	41	35	1	5	6
Median Rend	13	1	0	13	45	35	1	5	6

(b) Connector/J	WMC	DIT	NOC	CBO	MVG
Avg diff	1,90	-0,05	-0,05	-0,95	-2,45
Avg diff %	1,6%	-5,0%	-5,0%	-5,4%	-1,4%
Avg Rstart	60,65	0,3	0,15	12,15	121,4
Avg Rend	62,55	0,25	0,1	11,2	118,95
Median Rstart	32,5	0	0	9,5	91,5
Median Rend	40,5	0	0	10	92,5

(c) Log4J/ckjm	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	NPM
Avg diff	1	0	0	-3	19	113	4	0
Avg diff %	5,8%	0,0%	0,0%	1,7%	9,9%	9,0%	10,8%	7,8%
Avg Rstart	12,74	1,16	0,11	6,95	32,68	46,26	1,26	11,26
Avg Rend	12,79	1,16	0,11	6,79	33,68	52,21	1,47	11,26
Median Rstart	10	1	0	8	31	0	1	8
Median Rend	11	1	0	7	28	1	1	8

(d) Log4J/CCCC	WMC	DIT	NOC	CBO	MVG
Avg diff	50	2	1	26	0,28
Avg diff %	12,3%	2,6%	1,3%	2,7%	10,7%
Avg Rstart	18,00	1,05	0,13	12,36	13,46
Avg Rend	18,54	1,08	0,15	12,38	13,74
Median Rstart	13	1	0	10	6
Median Rend	13	1	0	9	9

Figure 3. The effect of refactorings on the object oriented libraries examined

3.5. JBoss Hibernate

Hibernate is a popular open source Object-Relational Mapping (ORM) library written in the Java programming language[8]. The component examined in this study is Hibernate Core, which is fairly large, as its source code weighs in at about 185KLOC. For the purpose of this experiment a total of 59 revisions in 30 pairs were examined. The results are presented in Figure 3.

The statistics calculated for Hibernate indicate a similarity in the way the various metrics were affected to the ones of Log4J. Both Log4J and Hibernate displayed a quite strong overall and average increase of the RFC and CBO metric. Moreover, the increase in the value of the LCOM and Ca metrics is even bigger, approaching 18% and 16% respectively. This means that as the classes were reengineered they became less coherent and more interdependent, and given the large average values of these metrics one could come to the conclusion that these refactorings were not beneficial to the project's overall quality.

4. Evaluation of results and future work

Software systems need to go under modifications, improvements and enhancements in order to cope with evolving requirements. This maintenance can adversely affect their quality. Refactoring is one of the most important and commonly used techniques for improving the quality of software, which can be measured by employing various metrics. However, although it would be expected that the improvement is reflected in the various metrics, this does not seem to be the case in real life systems.

This study examines how the metrics of popular open source projects were affected when the development team performed refactorings, regardless of the reasons that led to that decision. The results indicate a significant change of certain metrics to the worse. Specifically it seems that refactoring caused a non trivial increase in metrics such as LCOM, Ca and RFC, indicating that it caused classes to become less coherent as more responsibilities are assigned to them. The same principles seem to apply in procedural systems as well, in which case the effect is captured as an increase in complexity metrics. Since it is a common conjecture that the metrics used can actually indicate a system's quality, these results suggest that either the refactoring process does not always improve the quality of a system in a measurable way or that developers still have not managed to use refactoring effectively as a means to improve software quality.

To further validate these results, more systems and even more revisions must be examined, because the number examined so far is relatively small. Using a refactoring detection technique to identify the refactorings performed each time, one could also correlate each kind of refactoring to a specific trend in the change of various metrics and thus deduce which ones are more beneficial to the overall quality of a system.

[8]Available at `http://www.hibernate.org/`

References

[1] International Organization for Standardization. Software Engineering - Product Quality - Part 1: Quality Model. ISO, Geneva, Switzerland, 2001. ISO/IEC 9126-1:2001(E), 2001.

[2] F. B. e. Abreu and W. L. Melo. Evaluating the impact of object-oriented design on software quality. In *Proceedings of the 3rd International Symposium on Software Metrics*, page 90, 1996.

[3] R. K. Bandi, V. K. Vaishnavi, and D. E. Turk. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Trans. Softw. Eng.*, 29(1):77–87, 2003.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[5] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *Proceedings of the 10th Eu-*

ropean Software Engineering Conference, pages 156–165, 2005.

[6] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software engineering*, pages 592–605, 1976.

[7] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for C++. In *Proceedings of the 19th International Conference on Software Engineering*, pages 412–421, 1997.

[8] L. C. Briand, S. Morasca, and V. R. Basili. Defining and validating measures for object-based high-level design. *IEEE Trans. Softw. Eng.*, 25(5):722–743, 1999.

[9] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.

[10] A. Chatzigeorgiou. Mathematical assessment of object-oriented design quality. *IEEE Trans. Softw. Eng.*, 29(11):1050–1053, 2003.

[11] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings of the 6th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 197–211, 1991.

[12] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design˙ *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001. Chapter 1: Foundations, pp. 3-122.

[14] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 166–177, 2000.

[15] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings for libraries and frameworks. In *Proceedings of Workshop on Object Oriented Reengineering (WOOR'05)*, July 2005.

[16] F. B. e. Abreu, M. Goulão, and R. Esteves. Toward the design quality evaluation of object oriented software systems. In *Proceedings of the 5th International Conference on Software Quality*, October 1995.

[17] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, August 1999.

[18] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, February 2005.

[19] M. H. Halstead. Elements of software science. *Operating and Programming Systems Series*, 7, 1977.

[20] S. M. Henry and D. G. Kafura. Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, 7(5):510–518, 1981.

[21] S. M. Henry and D. G. Kafura. The evaluation of software systems' structure using quantitative software metrics. *Software - Practice and Experience*, 14(6):561–573, 1984.

[22] P. Joshi and R. K. Joshi. Microscopic coupling metrics for refactoring. In *Proceedings of the 10th Conference on Software Maintenance and Reengineering*, pages 145–152, 2006.

[23] S. H. Kan. *Metrics and Models in Software Quality Engineering (2nd Edition)*. Addison-Wesley Professional, 2002.

[24] W. Li and S. M. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[25] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall Object-Oriented Series. Prentice Hall, 1994.

[26] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, December 1976.

[27] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, December 1989.

[28] T. J. McCabe and A. H. Watson. Software complexity. *Crosstalk, Journal of Defense Software Engineering*, 7(12):5–9, December 1994.

[29] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, February 2004.

[30] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Doctoral thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[31] R. Reißing. Towards a model for object-oriented design measurement. In *5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2001.

[32] L. H. Rosenberg, R. Stapko, and A. Gallo. Applying object-oriented metrics. In *Sixth International Symposium on Software Metrics - Measurement for Object-Oriented Software Projects Workshop*, 1999.

[33] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Proceedings of the 5th Conference on Software Maintenance and Reengineering*, pages 30–38, 2001.

[34] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[35] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, 2003.

[36] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software re-engineering. *Journal of Systems and Software*, 66(3):225–239, June 2003.

[37] R. Tiarks. Quality-driven refactoring. Technical report, University of Bremen, 2005.

[38] K. D. Welker and P. W. Oman. Software maintainability metrics models in practice. *Crosstalk - The Journal of Defense Software Engineering*, 8(11):19–23, 1995.

[39] M. Xenos, D. Stavrinoudis, K. Zikouli, and D. Christodoulakis. Object-oriented metrics - a survey. In *Proceedings of Federation of European Software Measurement Associations, Madrid, Spain*, 2000.

[40] Y. Yu, J. Mylopoulos, E. Yu, J. C. Leite, L. L. Liu, and E. D'Hollander. Software refactoring guided by multiple soft-goals. In *The 1st International Workshop on Refactoring: Achievements, Challenges, Effects*, November 2003.