2015

The Interplay Among Software Volatility, Complexity and Development Outcomes: Evidence From Open Source Software

Jorge A. Colazo
Trinity University, jcolazo@trinity.edu

Repository Citation

Colazo, J. A. (2015). The interplay among software volatility, complexity and development outcomes: Evidence from open source software. *International Journal of Information Technology and Management, 14*(2-3), 160-171. https://doi.org/10.1504/IJITM.2015.068462

The interplay among software volatility, complexity and development outcomes: evidence from open source software

Jorge A. Colazo

Business School – Department of Finance and Decision Sciences,
Trinity University,
Office CGC 305, One Trinity Place,
San Antonio, TX 78238, USA
E-mail: jcolazo@trinity.edu

Abstract: The study posits a mediating role of software complexity in the association between software volatility and different software development outcomes. Empirical tests using data from 326 open source software projects support such a partial mediating role of software complexity in the association between software volatility and development outcomes. Archival data is tested using an ordinary least squares mediated model. The paper uses productivity, defect count and development speed as dependent variables.

Keywords: software complexity; software volatility; open source software; OSS; software development; development speed; software quality.

Reference to this paper should be made as follows: Colazo, J.A. (xxxx) 'The interplay among software volatility, complexity and development outcomes: evidence from open source software', *Int. J. Information Technology and Management*, Vol. X, No. Y, pp.000–000.

Biographical notes: Jorge A. Colazo is an Assistant Professor at the Finance and Decision Sciences Department of Trinity University's School of Business. An engineer by training, he holds a PhD in Business Administration from The University of Western Ontario, Canada. Prior to his academic work he had extensive managerial experience in firms such as Unilever and Toyota, working in R&D and production. His research interests are related to open innovation, technology management and lean manufacturing. His work has been published in *Journal of the Association for Information Systems*, *International Journal of Innovation Management* and other outlets.

1 Introduction

Software complexity impacts on many important software project outcomes, such as the number of expected defects, the understandability and maintainability of the code, and the effort required to grow the source base and interface it with other systems. It also limits the degree of code reuse that is possible for a given software solution, making scalability more difficult. 'Filtering through' software complexity is a programmer – intensive task that is very difficult to automate (Glass, 2002). In this context,

understanding complexity and exploring practical ways to manage it remains critically important.

Although complexity is determined by multiple factors, notoriously the scope and difficulty of implementing initial requirements, it also evolves along the life cycle of the product, and changes every time new features are added, bugs are fixed, and in general, changes are made to the source code base. The frequency and magnitude of ongoing modifications to the software's source code are captured by the concept of software volatility (Banker and Slaughter, 2000). Volatility and complexity are different constructs; while volatility captures the amount of changes made on the source base, those changes may or may not affect the code's complexity, i.e., the intricacy of the logic flow within the software's control structure.

Both volatility and complexity are known to impact diverse performance metrics. For instance, volatility has been shown to correlate with the cost of enhancing software (Banker and Slaughter, 2000), whereas complexity is related to the reliability and maintainability of code (Lew et al., 1988). However, the two constructs have to date been treated separately in empirical studies exploring their association with development outcomes.

Some anecdotal evidence supports that there is an interplay between complexity and volatility; when modifications are made to accommodate new requirements software complexity grows explosively: "for every 25% increase in problem complexity there is a 100% increase in the complexity of the software" (Woodfield, 1979). The nature of a potential interplay between volatility and complexity has not yet been hypothesised and empirically tested. This paper posits that volatility affects development outcomes both directly and indirectly through the mediating effect of complexity.

The interaction between volatility and complexity is not only academically interesting but also important from a very practical point of view, because volatility can be managed a priori or at least moderated by explicit design policy, for instance by structuring software more rigorously (Banker and Slaughter, 2000). In contrast, the baseline complexity of a software package is largely a result of initial requirements implementation and the kind of processing solution chosen to address those requirements. Once complexity is initially designed into the software it is very difficult to radically change it, and for the rest of the lifecycle of the product it will define the ability of programmers to understand the code, impact the developers' learning rates, and their ability to find issues and maintain the product.

What motivates this paper is the need to understand the mechanisms by which volatility affects development outcomes. Are volatility and complexity inextricably related, or are they independent? If the former were true, the focus to quickly and productively develop reliable software would need to be the clear definition of initial requirements and the use of proven algorithms to achieve functionality at the lowest complexity point. If the latter were true, the focus would have to be placed on minimising disruptions to the source code base once baseline requirements are achieved.

After hypothesising the mediating role of product complexity on the relationship between software volatility and a series of software development outcomes, regression-based techniques to empirically test the hypotheses are used. We select as outcomes coding productivity, expected pre-test defects, and development speed.

Due to the wealth of data that can be obtained about volatility, complexity and development outcomes, the research framework for the empirical validation of hypotheses is a sample of open source software (OSS). OSS is software produced under

licensing terms that make the source code of the product available to users. Notable examples of OSS are Linux, Apache and MySQL.

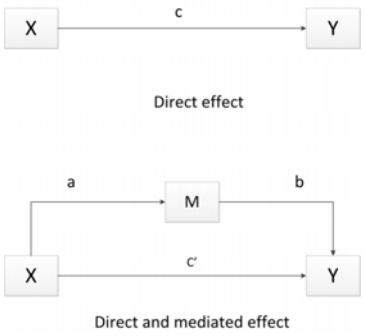
Results show that software complexity partially mediates the influence of volatility in all three outcomes, i.e., there is a significant mediation effect but complexity does not act as a full mediator.

The remainder of the paper is structured as follows: the second section reviews relevant research and lays out the hypotheses to be tested. Next, the analytical methods used in this empirical study are explained. The following section explains the results of the ordinary least squares (OLS) regression model used to test the hypotheses. The last section contains the study's conclusions, limitations and implications for future research.

2 Background and hypotheses

It is postulated that the effect of volatility on diverse development outcomes is mediated by software complexity. The theoretical model represents a simple mediation design, shown in Figure 1.

Figure 1 Research model: mediation



This mediated model requires supporting the association between the independent variable (IV) X (volatility) and the dependent variable(s) Y (DVs) (the different development outcomes) as well as the association between the IV and the mediator M (complexity), and between the mediator and the DVs. Based on complexity theory and previous research hypotheses can now be formulated.

2.1 Software volatility and software complexity

Software volatility is the frequency and extent to which the source code is changed along the product's lifetime (Banker and Slaughter, 2000). Volatility is in principle neither beneficial nor detrimental to software development outcomes, only reflecting changes to the source code base due to different events, such as new features, bug fixing, modularisation, restructuring and refactoring, among others.

The concept of software volatility refines previous ways of looking at changes in source code, from the broad software engineering concept of 'code churn' (Khoshgoftaar, 1994) or the more simple 'code delta' [absolute difference in lines of code (LOC)] and 'rate of change' (the relative difference in LOC) (Ajila and Dumitrescu, 2007).

Diverse outcomes are affected by volatility. Code churn has been associated empirically to defect density (Nagappan, 2005), whereas volatility was associated to software size and age in a study of software used by the oil industry (Zhang and Windsor, 2003).

Code churn has even been shown to be a good indicator of portions of the code where security vulnerabilities are likely to occur (Shin, 2011), and similar results were found in the case of the OSS development environment eclipse (Zimmermann et al., 2007). In the case of object oriented software, 'code smells' are sections of the source base that present maintainability issues and they are also related to volatility in the case of two open source projects that are part of the Apache web server (Olbrich et al., 2009).

Software volatility has been measured in extant research by the fraction of software modules that were changed per new release (Belady and Lehman, 1976), or by the number of enhancements per function point (Banker and Slaughter, 2000).

The definition of software complexity is more complicated than that of volatility, since there are different types of complexity such as structural and logic complexity, and for each type several metrics have been proposed (Weyuker, 1988). It is accepted that complexity is multidimensional, although there is still discussion on which dimensions are appropriate (Kearney et al., 1986; Glass, 2002).

For this paper, complexity is defined as the intricacy of the logical flow of data from input to output within the functions in the source code, and it is related to the degree of understandability of the code (Glass, 2002). This definition of complexity is captured by the time-tested concept of cyclomatic complexity or McCabe's complexity (McCabe, 1976).

2.2 The indirect effect: complexity vs. performance

Based on complexity theory it can be posited that these observed relationships can be condensed and explained by a simple model where software volatility has both direct effect on outcomes and a mediated indirect effect on them, with software complexity as mediator.

Complexity theory (Byrne, 2002) posits that a certain order is emergent when organisms or agents interact. The system will adopt a stability status ranging from stable to chaotic depending on the behaviour of feedback loops present in the agents' interactions (Benbya and McKelvey, 2006). For software, agents can be construed as the software files, modules or the algorithms that render the required functionality.

Complexity theory predicts that complexity will increase as the number of interfaces among agents grows. In practice, several attributes of the software's code related to the increase in the number of inter-agent interfaces have been found to impact on complexity: the absolute code size in LOC, the number of files, the number of modules and the degree of structure of the source code (Banker et al., 1989). As the system becomes more unstable, its required performance diminishes and spurious feedback loops produce glitches (bugs).

Empirically, complexity has in fact been shown to strongly correlate with maintainability (Banker et al., 1989) productivity (Gill and Kemerer, 1991) reliability and

defect count (Lew et al., 1988). More complex programmes are more difficult to understand (Stamelos et al., 2002), and require more cognitive effort from the developer (Rilling and Klemola, 2003). Complexity density (cyclomatic complexity divided by the number of code statements) is negatively related to maintenance productivity (Gill and Kemerer, 1991).

2.3 The mediating effect: volatility vs. complexity

Complexity theory helps understand the association between volatility and complexity. One of the leading reasons for software volatility is adding new features. Obviously, adding features require modifying the source base, and hence implies some degree of volatility. But it also introduces new interfaces between modules and amplifies mutual causal loops (Maruyama, 1963), some of which are unexpected and add logic paths in a non-linear fashion increasing complexity likewise.

This agrees with the practitioner's perspective, recognising that changes in source code required when adding features have a multiplicative effect on the design of algorithms and interfaces, and it has been suggested that "explicit requirements explode by a factor of 50 or more into implicit design requirements" (Glass, 2002). Adding or modifying algorithms changes the control structure within the logic of the code, thus changing the logic complexity. Bug fixing is also positively related to volatility, but when defects are fixed, the algorithmic solution tends to be refined and rid of unnecessary feedback loops that caused unexpected behaviour.

Other causes of volatility associated with complexity are restructuring and refactoring. Restructuring software involves rearranging the components that define the architecture of the system, such as modules or files (Arnold, 1989) and applies to procedural programming languages. Refactoring is the equivalent of restructuring but for object-oriented languages. In both restructuring and refactoring the main focus is to simplify the structure while preserving the expected behaviour of the software (Mens and Tourwé, 2004). Refactoring and restructuring often standardises algorithms, classes and objects, and often times they involve the breaking down of components, altering the number of control interfaces. All of these modifications potentially alter the complexity of the code base.

2.4 The direct effect: volatility vs. performance

Having argued for the association between volatility and complexity, it is now time to review evidence supporting a direct effect of volatility on productivity, defect count and development speed, the DVs.

First, volatility has been shown to positively associate with software enhancement costs (Banker and Slaughter, 2000). Software enhancement is part of the maintenance stage of software evolution, by far the most costly stage in software's lifecycle (Lehman, 1998). Maintenance costs are directly related to the productivity in generating the modifications necessary to enhance the functionality of the programme. It can be expected that volatility is going to be associated with development productivity, especially in mature programmes that are in the maintenance stage of their life cycles.

Second, volatility is also positively related to error count. Software volatility may alter the ability of software maintainers to identify and fix bugs, and hence impact on the expected defect count (Kemerer et al., 2012). In fact, less volatile software shows fewer

errors when technology-based approaches are used to maintain it while more volatile software does better when maintained based on the interaction of experienced developers (Kemerer et al., 2012). Volatility can be expected to positively relate to defect count.

Third, volatility can be expected to be related to development speed. Software is generally released when development and reliability milestones have been met and the behaviour of the system is stable. Improving reliability depends on finding and correcting issues. Volatility is related to the effort to understand code, critical step in finding new issues (Nagappan, 2005) and it can be expected that more volatile software will be less understandable and hence extend inter-release times. Also, restructuring activities aimed at simplifying the architecture while maintaining external behaviour require increased volatility, and then volatility can be expected to influence inter-release times (development speed).

Given the above support for both direct effects of volatility on development outcomes and for a potential indirect effect through software complexity, it can be posited:

H1 Complexity mediates the effect of volatility on productivity.

H2 Complexity mediates the effect of volatility on defect count.

H3 Complexity mediates the effect of volatility on development speed.

3 Methods

The research framework for this study consists of OSS projects. The unit of analysis is the project. OSS project team information is hosted in web-based repositories which have been used repeatedly as sources of archival data for empirical studies (c.f., Mockus et al., 2002). In line with the majority of these studies, this paper uses projects hosted in Source Forge (SF) (http://www.sourceforge.net).

When measuring software metrics, the use of a samples with a single language is preferred (Jones, 1986). For this paper, the programming language of the projects was restricted to 'C' because besides being one of the most popular programming languages in SF, it has a rich tradition of static software metrics that have been validated empirically and can be obtained with off-the-shelf software analyzers. In accordance with previous empirical research on OSS, projects with six or more team members were selected, because most OSS projects with fewer developers do not show in general high levels of activity (Crowston and Howison, 2003; Colazo and Fang, 2010) and projects in a relatively mature stage are desirable.

In the SF repository, there were 326 software projects being developed by six or more core team members and using 'C'. These 326 projects constituted the sample. In spite of the non-probabilistic nature of the sample (see limitations), it contained projects with varied types of applications, sizes, and degrees of maintainability (Oman and Hagemeister, 1994) (Table 1). Data were cross-sectional, and collected between starting January 2008 with updates until July 2011. The average tenure of the projects was 30 months since their inception.

The source code for OSS is accessed by developers through some form of version control software that allows developers to keep track of modifications to the source code. The versioning system currently used at SF is called SubVersion. A Subversion client was used to download the source code files for each project. The client programme

provided a history of the modifications committed to files, yielding the name of the committer, date of commit, name of file modified, total LOC added and total LOC deleted for each commit.

The projects' source code files were downloaded and parsed with software analyzer programmes to obtain both product size in LOC as well as other static metrics such as code complexity and Halstead's (1977) metrics that were later used to calculate number expected bugs and maintainability.

From the source code files for each project, quality of coding was measured by the number of total expected pre-test software defects (called 'B' after the initial letter of 'Bugs') (Ottenstein, 1981), standardised per thousand lines of source code (B/KSLOC). The parameter 'B' estimates the expected number of defects latent in the source code, and it has been validated (Gremillion, 1984) against actually found quality defects in the software testing stage. Note that a higher defect count corresponds to lower software quality. The *number of delivered bug*s 'B' (Ottenstein, 1981) was derived from *Halstead's software volume* 'V' (Halstead, 1977) by the formula:

$$B = V^{-2/3} / 3{,}000$$

Halstead's volume is calculated as:

$$V = N_H \log_2 n$$

N_H is called programme length:

$$N_H = N_1 + N_2$$

And n is called programme vocabulary:

$$n = n_1 + n_2$$

Being n_1 the *number of distinct logic operators* (e.g., +, /, *, mod, abs, IF-THEN, SWITCH and so on) and n_2 the *number of distinct operands* (those that take part in the operations), N_1 the *total number of operators* and N_2 the *total number of operands*. Obtaining the number of operators and operands depends on language-specific accepted conventions and then requires a language-specific parser (Szulewski et al., 1984). Scitools (2012) metrics suite was used.

Development speed was measured by the average number of days between any two consecutive versions (inter-release time). The version control software log files provided the version numbers of the software as it was being developed. For each project, all dates when a change in the revision number occurred were recorded for all different types of software versions: major versions (e.g., 1.0, 2.0, etc.), minor versions (1.1 to 1.2 for example), and 'build' versions (e.g., 2.1.1 to 2.1.2). Times between minor releases were used and show the highest statistical power, but conclusions do not change if major or build releases are used instead.

Productivity was measured by the number of LOC written per developer per month, a popular and established measure for development productivity (c.f., Jones, 1986).

Software complexity was measured using the project's average McCabe's (1976) cyclomatic complexity factor. Two different off-the-shelf code analyzers were used (Scitools, 2012; Testwell Oy, 2012). The range of complexity in the sample was inspected using the guidelines given by Marciniak (1994). This analysis showed that the programmes ranged from simple to highly complex (Table 1).

Table 1 Sample programme complexity

Cyclomatic complexity range	*Category*	*%*
1–10	Simple	59
11–20	Moderately complex	24
21–50	Complex	14
> 50	Highly complex to intractable	3

To measure volatility the number of feature requests, the number of bug patches submitted and the number of changes to the source code (commits) per month for each project, standardising them by the number of LOC were obtained, to which principal components analysis was applied to extract the common variance.

The measures for volatility and complexity were subjected to a content validity assessment (Moore and Benbasat, 1991; Hinkin, 1998) using a panel of software development experts associated to a university. Eight experts were provided with construct definitions and asked to match constructs with measures. They came to 87.5% agreement, which is higher than the 75% normally indicative of content adequacy (Schriesheim et al., 1993).

Data passed robustness check of the factor analysis by examining and addressing the key assumptions of number of cases, linearity of factor items, multivariate normality, and absence of outliers. In principle the items discussed would load on only one significant factor that would reflect software volatility. The criterion of eigenvalues higher than 1.0 (Kaiser, 1974) yielded as expected only one significant factor, with 43% of the total variance captured by the factor.

For each dependent variable, the model tested can be seen in Figure 1. The total effect of X on Y is represented by path *c*. Path a represents the effect of X on the mediator M, whereas path *b* is the effect of the mediator M on the dependent variable Y partialling out the effect of X. A valid identity is that:

$$ab = c - c'$$

where *ab* is the indirect effect of X on Y through M, or the difference between the total effect *c* and the direct effect c', and as such a measure of the mediating effect of M. In order to support mediation *a*, *b* and *c* should be significant, and the difference between *c* and c' should be non-trivial. The latter condition can be difficult to interpret, and bootstrapping methods are used to get a confidence interval of the product *ab*, where if the confidence interval does not contain zero, the mediating effect is supported (Preacher and Hayes, 2008).

Following those premises, a regression analysis with bootstrapping was carried out to ascertain the significance of the mediating effect of complexity on the impact of volatility on the different DVs.

4 Results

Results of the regression and bootstrap analyses are presented in Table 2.

Table 2 Results

	Unstandardised coefficients					
	Dependent variables (DVs)					
	Productivity		*Defects*		*Speed (time b/releases)*	
Volatility → complexity (*a*)	–0.0185	+	–0.0184	*	–0.0282	**
Complexity → DV (*b*)	0.3228	**	0.1141	***	–58.5712	*
Volatility → DV, total effect (*c*)	0.1653	***	0.016701	**	–26.7977	***
Volatility → DV, direct effect (*c*′)	0.1713	***	0.0188	**	–28.4494	***
R^2	0.07	*	0.16	**	0.29	**
95% bootstrap confidence intervals for indirect effects ab						
Mediation effect (ab)	*–0.0060*		*–0.0021*		*1.6517*	
	Lower	*Upper*	*Lower*	*Upper*	*Lower*	*Upper*
	–0.0103	–0.0021	–0.0036	–0.0010	0.5657	2.8719

Notes: $N = 326$.
Number of bootstrap samples: 5,000.
$+p < 0.1$; $*p < 0.05$; $**p < 0.01$; $***p < 0.001$.

H1 was supported. The model with productivity as a dependent variable shows all paths were significant, as well as the mediating effect of complexity. Volatility is negatively correlated with complexity ($p < 0.1$), and complexity is positively correlated with productivity ($p < 0.01$). Volatility's direct effect on productivity is positive ($p < 0.001$), i.e., the more volatile the software, the higher the productivity. The mediating effect of complexity is partial, since the introduction of the mediator does not render the direct effect insignificant.

H2 was supported. The model for number of defects shows that all paths were significant as well as the mediating effect of complexity. More volatile software is associated with a higher defect count ($p < 0.01$), and the direct effect is partially mediated by software complexity.

Finally, H3 was supported as well. The results for development speed (time between releases) shows again a partial mediating effect for complexity, with more volatile software associated with faster development ($p < 0.01$). The mediating effect of complexity is again partial.

The sign for path coefficient '*b*' seems to indicate that projects with higher productivity become more complex, more error prone and faster to reach release milestones. The direct effect of volatility on defect count supports previous studies linking volatility with reliability issues (Nagappan, 2005). Also, results show that projects that release often are more volatile.

5 Conclusions, limitations and future research

Results support that there is a mediating effect of software complexity in the association between software volatility and diverse development outcomes. Results have several contributions for both academia and practice.

From the theoretical perspective, first, the mediating effect of complexity helps explain the observed similar impacts of both volatility and complexity on performance.

Second, the mediating effect is partial in all cases. The effect of volatility on productivity, quality and speed is not totally explained by a mediating effect. This can be explained by changes to the source base that do not modify complexity, such as adding comments, and by some restructuring activities that do not change the underlying logic structure.

From the practical point of view, the observed partial mediation provides an avenue to influence the evolution of complexity by policing actions that contribute to volatility. Examples are increasing accountability to minimise unnecessary changes, implementing performance inquiry, critical managerial judgement and sanctions, all of which have been shown to mitigate the effects of complexity (Midha and Slaughter, 2011).

Project managers can also increase reliability by carefully administering the frequency of restructuring/refactoring. When restructuring and considering alternative algorithms, care should be taken to choose the least complex solution whenever possible, and documenting the source code with comments that may help understandability.

The sample contained relatively mature projects, and a negative association was obtained between volatility and complexity, perhaps contrary to standard knowledge. This result is understandable because at a mature stage changes in the source code aim to decrease complexity. This result is worth replicating in samples with projects at an earlier stage in their life cycle.

As it is always the case with empirical research, this study has limitations. One of them is the non-probabilistic nature of the sample, calling for replication with software written in other languages and in proprietary applications rather than in the OSS realm. Due to limitations in the dataset more covariates have not been included as statistical controls, which should be explored in future research. Another limitation is the definition of complexity, which predominantly captures logic intricacy. Other definitions such as structural complexity or architectural complexity should be explored as well. Productivity and quality in software are complex constructs that were simplified in terms of considering only LOC and bugs per KSLOC; these results should be confirmed using different and broader measures.

References

Ajila, S.A. and Dumitrescu, R.T. (2007) 'Experimental use of code delta, code churn, and rate of change to understand software product line evolution', *Journal of Systems and Software*, Vol. 80, No. 1, pp.74–91.

Arnold, R.S. (1989) 'Software restructuring', *Proceedings of the IEEE*, Vol. 77, No. 4, pp.607–617.

Banker, R.D. and Datar, S.M. et al. (1989) 'Software complexity and maintainability', *ICIS 1989*, AIS, Boston, MA.

Banker, R.D. and Slaughter, S.A. (2000) 'The moderating effects of structure on volatility and complexity in software enhancement', *Information Systems Research*, Vol. 11, No. 3, pp.219–240.

Belady, L.A. and Lehman, J.A. (1976) 'A model of large program development', *IBM Systems Journal*, Vol. 15. No. 3, pp.225–252.

Benbya, H. and McKelvey, B. (2006) 'Toward a complexity theory of information systems development', *Information Technology & People*, Vol. 19, No. 1, pp.12–34.

Byrne, D. (2002) *Complexity Theory and the Social Sciences: An Introduction*, Routledge, Abingdon, Oxon, UK.

Colazo, J.A. and Fang, Y. (2010) 'Following the sun: temporal dispersion and performance in open source software project teams', *Journal of the Association for Information Systems*, Vol. 11, No. 11, pp.684–707.

Crowston, K. and Howison, J. (2003) 'The social structure of free and open source software development', *24th International Conference on Information Systems*, Seattle, WA.

Gill, G.K. and Kemerer, C.F. (1991) 'Cyclomatic complexity density and software maintenance productivity', *IEEE Transactions on Software Engineering*, Vol. 17, No. 12, pp.1284–1288.

Glass, R.L. (2002) 'Sorting out software complexity', *Communications of the ACM*, Vol. 45, No. 11, pp.19–21.

Gremillion, L.L. (1984) 'Determinants of program repair maintenance requirements', *Communications of the ACM*, Vol. 27, No. 8, pp.826–832.

Halstead, M.H. (1977) *Elements of Software Science*, Elsevier, New York, NY.

Hinkin, T.R. (1998) 'A brief tutorial on the development of measures for use in survey questionnaires', *Organizational Research Methods*, Vol. 1, No. 1, pp.104–121.

Jones, C. (1986) *Programming Productivity*, McGraw-Hill, New York.

Kaiser, H.F. (1974) 'An index of factorial simplicity', *Psychometrika*, Vol. 39, No. 1, pp.31–36.

Kearney, J.K. and Sedimeyer, R.L. et al. (1986) 'Software complexity measurement', *Communications of the ACM*, Vol. 29, No. 11, pp.1044–1050.

Kemerer, C.F. and Moody, G.D. et al. (2012) *Effective Corrective Maintenance Strategies for Managing Volatile Software Applications*, Working paper [online] http://www.pitt.edu/~ckemerer/Research.htm.

Khoshgoftaar, T.M. (1994) 'Improving code churn predictions during the system test and maintenance phases', *International Conference on Software Maintenance*, IEEE, Victoria, BC, Canada.

Lehman, M.M. (1998) 'Software's future: managing evolution', *IEEE Software*, Vol. 15, No. 1, pp.40–44.

Lew, K.S. and Dillon, T.S. et al. (1988) 'Software complexity and its impact on software reliability', *IEEE Transactions on Software Engineering*, Vol. 14, No. 11, pp.1645–1655.

Marciniak, J.J. (Ed.) (1994) *Encyclopedia of Software Engineering*, John Wiley & Sons, New York, NY.

Maruyama, M. (1963) 'The second cybernetics: deviation-amplifying mutual causal processes', *American Scientist*, Vol. 51, No. 2, pp.164–179.

McCabe, T. (1976) 'A software complexity measure', *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, pp.308–320.

Mens, T. and Tourwé, T. (2004) 'A survey of software refactoring', *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, pp.126–139.

Midha, V. and Slaughter, S. (2011) 'Mitigating the effects of structural complexity on open source software maintenance through accountability', *ICSIS 2011*, AIS, Shanghai, China.

Mockus, A. and Fielding, R.T. et al. (2002) 'Two case studies of open source software development: Apache and Mozilla', *ACM Transactions on Software Engineering and Methodology*, Vol. 11, No. 3, pp.309–346.

Moore, G.C. and Benbasat, I. (1991) 'Development of an instrument to measure the perceptions of adopting an information technology innovation', *Information Systems Research*, Vol. 2, No. 3, pp.192–222.

Nagappan, N. (2005) 'Use of relative code churn measures to predict system defect density', *ICSE 2005*, AIS, St. Louis, MO.

Olbrich, S. and Cruzes, D.S. et al. (2009) 'The evolution and impact of code smells: a case study of two open source systems', *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, IEEE Computer Society.

Oman, P. and Hagemeister, J. (1994) 'Construction and testing of polynomials predicting software maintainability', *Journal of Systems and Software*, Vol. 24, No. 3, pp.251–266.

Ottenstein, L. (1981) 'Predicting numbers of errors using software science', *ACM SIGMETRICS Performance Evaluation Review*, Vol. 10, No. 1, pp.157–167.

Preacher, K.J. and Hayes, A.F. (2008) 'Assymptotic and resampling strategies for assessing and comparing indirect effects in multiple mediator models', *Behavior Research Methods*, Vol. 40, No. 3, pp.879–891.

Rilling, J. and Klemola, T. (2003) 'Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics', *11th IEEE International Workshop on Program Comprehension*, Portland, OR.

Schriesheim, C.A. and Powers, K.J. et al. (1993) 'Improving construct measurement in management research: comments and a quantitative approach for assessing the theoretical content adequacy of paper-and-pencil survey-type instruments', *Journal of Management*, Vol. 19, No. 2, pp.385–417.

Scitools (2012) *Understand*, Static Metrics Analyzer, Scientific Toolworks Inc.

Shin, Y. (2011) 'Evaluating complexity, code churn and developer activity metrics as indicators of software vulnerabilities', *IEEE Transactions on Software Engineering*, Vol. 37, No. 6, pp.772–787.

Stamelos, I. and Angelis, L. et al. (2002) 'Code quality analysis in open-source software development', *Information Systems Journal*, Vol. 12, No. 1, pp.43–60.

Szulewski, P.A. and Sodano, N.M. et al. (1984) *Automating Software Design Metrics*, Rome Air Development Center, Rome, NY.

Testwell Oy (2012) *CMT++. Hermia*, Testwell: Software Static Metrics Analyzer, Finland.

Weyuker, E.J. (1988) 'Evaluating software complexity measures', *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp.1357–1365.

Woodfield, S.N. (1979) 'An experiment on unit increase in software complexity', *IEEE Transactions on Software Engineering*, Vol. 5, No. 2, pp.76–79.

Zhang, X. and Windsor, J. (2003) 'An empirical analysis of software volatility and related factors', *Industrial Management and Data Systems*, Vol. 103, No. 4, pp.275–281.

Zimmermann, T. and Premraj, R. et al. (2007) 'Predicting defects for eclipse', *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007*, International Workshop on, IEEE.