# Measuring Refactoring Benefits: A Survey of the Evidence

Mel Ó Cinnéide
School of Computer Science
University College Dublin
Ireland
mel.ocinneide@ucd.ie

Aiko Yamashita
Dept. of Computer Science
Høgskolen i Oslo og Akershus
Norway
Aiko.Yamashita@hioa.no

Steve Counsell
School of Information Systems
Brunel University
United Kindom
Steve.Counsell@brunel.ac.uk

## ABSTRACT

Refactoring has become a standard technique for software developers to use when trying to improve or evolve the design of a program. It is a key component of Agile methods, the most popular family of software development methodologies in industrial practice. Refactoring has also been the subject of much attention from researchers and many practitioner textbooks have been written on the topic. It would be natural to assume then that the benefits of refactoring would be easy agree upon, and easy to measure. In this position paper we review a selection of the empirical studies that have attempted to measure the benefits of refactoring and find the situation to be quite unclear. The evidence suggests that what motivates developers to refactor, and what benefits accrue from refactoring, are open issues that require further research.

## CCS Concepts

•**Software and its engineering** → *Software design engineering;*

## Keywords

Refactoring; Empirical studies

## 1. INTRODUCTION

The practice of software refactoring is probably as old as programming itself, although the term 'refactoring' first appeared only in 1990 in a paper by Opdyke and Johnson [23]. Much of the early research work on refactoring took place at the University of Illinois [22] and it became recognised as a key software practice with the advent of Agile methods [3]. Fowler wrote the seminal refactoring handbook [10], and since then many textbooks have been written on the topic.

Refactoring has been claimed to remove code smells and to improve software quality including code extensibility, modularity, reusability, complexity, maintainability, efficiency etc. [18], and a number of research studies have investigated these claims. Given the obvious popularity of refactoring, one might expect that its benefits could easily be measured. In this paper we examine a number of the most-commonly made claims about refactoring:

- Refactoring is employed to eradicate code smells from the code base (Section 2).
- Refactoring is employed to improve the values of software quality metrics (Section 3).
- Refactoring is employed to improve flexibility, maintainability, extensibility etc. (Section 4).

The aim of this workshop paper is to summarise some of the key work in the area of refactoring assessment, point out the anomalies in this work and provide a basis for further discussion. It is not a formal meta-analysis of the existing results; neither is it a systematic literature review. The papers cited were chosen on the basis of the experience of the authors, and were not selected to support any particular side of the argument.

## 2. DOES REFACTORING REDUCE CODE SMELLS?

The notion that refactoring is used to eradicate code smells is very prevalent, no doubt partly due to the fact that this idea is strongly espoused in Fowler's handbook [10]. It is also a very appealing idea: we establish a set of code smells (the 'baddies') and then define refactoring sequences (the 'goodies') that convert these malignant structures into harmless ones. It is precisely because this idea is deeply appealing that we should be more alert to it being, in the terminology of Bossavit [4], a *software leprechaun*, i.e. a widely-believed 'ground truth' that is in fact false.

Counsell et al. studied five open source systems and one proprietary system to explore to what extent refactoring is used to eradicate code smells [7]. In the open source systems studied, they identified a total of 891 refactorings and 22 well-known code smells. Only two of the identified code smells were clearly removed by the refactorings, and these were smells that required only a small number of refactorings to remedy. This echoes the earlier result reported by Chatzigeorgiou and Manakos [6] who tracked the evolution of three code smells in two open source systems and found that of the 648 smells identified in the systems, only five could be determined to have been unambiguously removed by refactoring. These results provide evidence that the refactorings applied by developers are not strongly motivated by a desire to remove smells.



*IWoR'16*, September 4, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-4509-5/16/09...$15.00
http://dx.doi.org/10.1145/2975945.2975948

Empirical studies that support the refactoring/smells model also exist. Tsantalis et al. [29] studied the refactoring history of three well-known projects, namely JUnit, HTTPCore, and HTTPClient. One of the research questions they investigated was what was the purpose of the refactorings applied? In total, 280 refactoring instances were manually inspected of which 40% were determined by the authors to have been related to code smell removal. This seemingly contradicts the earlier studies; smells are driving refactorings to a significant degree. How can these results be resolved?

Firstly, there is little agreement on what precisely is, and is not, a code smell [9]. Taking one of the simplest code smells as an example, Long Method, Robert Martin sets 20 lines of code as a maximum for a method and admires Kent Beck's style where each method contains only two to four lines of code [16], while Steve McConnell states that methods of up to 200 lines are fine [17]. Of course the truth is that while a method may indeed be 'too long,' and this is one of smells developers best agreed upon in a study by Palomba et al. [24], the existence of a true Long Method code smell cannot be determined simply by counting the number of lines in the method.

Secondly, how bad are code smells anyway? Sjøberg et al. [26] investigated the effects of 12 code smells on maintenance effort at file level, and found that, after adjusting for file size and the number of changes (revisions) as quality predictors, none of the code smells remained a significant driver of effort. Rather bizarrely, the Refused Bequest code smell [10] actually contributed significantly to *less* effort. Yamashita and Moonen [31] analysed the data from the same projects and also found the impact of code smells on overall maintainability to be relatively minor. Hall et al. [11] investigated the relationship between bugs and five little-studied smells in three open-source systems finding that some smells indicate fault-prone code in some circumstances, but that the effect size is small. Some of their finding were again counter-intuitive to say the least: Message Chains that occurred in larger files *reduced* faults and Data Clumps correlated with *reduced* faults in Apache and Eclipse. Their overall conclusion was that arbitrary refactoring is unlikely to significantly reduce fault-proneness, and in some cases may increase fault-proneness. An earlier literature review by similar authors [32] concluded that there is little evidence currently available to justify using code smells.

Given the uncertainty in defining and identifying code smells, all empirical studies in this area suffer a significant threat to construct validity. Even where developers agree that a code smell exists, it may not actually cause a problem, e.g. where the smelly code is non-volatile. Furthermore, in two of the works cited above it seems that code smells can be beneficial in some cases! Finally, even when a code smell does cause a problem, a cost-benefit analysis may indicate that suffering its existence is less costly than fixing it.

# 3. DOES REFACTORING IMPROVE SOFTWARE QUALITY METRICS?

Researchers have also examined if refactoring leads to a measurable improvement in software quality metrics, such as coupling and cohesion. Stroggylos and Spinellis analyzed the version control system logs of several popular open source software systems to determine the impact refactorings have on software metrics, and found that, contrary to popular belief, refactoring did *not* cause the metrics to improve [28]. However this work relied upon the use of commit messages to detect refactorings, an approach that was later discredited [20]. In his study of open source software, Alshayeb observed that refactoring has a positive effect on several cohesion metrics [2], but his later work showed that this effect was not necessarily positive in terms of other external software quality attributes such as reusability, understandability, maintainability, testability and adaptability [1]. Soetens and Demeyer analysed the evolution of an open source Java project and found that periods of refactoring activity did not affect the cyclomatic complexity of the software [27].

Evidence to show that refactoring can improve software quality metrics is also to be found. A study by Moser et al. [19] explored if refactoring improved developer productivity or software quality metrics. The study involved an Agile software project that had two episodes of explicit refactoring, where user stories where created solely based on refactoring activity. The authors found that developer productivity improved after both these refactoring episodes, and that for the metrics examined, CBO, WMC, RFC, and LCOM, refactoring was found to improve them to some degree. A similar result, that refactoring leads to some increase in software quality metrics, was reported by Kim et al. [14] as described in Section 4.

Simons et al. [25] conducted a survey with software professionals to investigate the relationship between popular SBSE refactoring metrics and the subjective opinions of software engineers. The empirical study results suggest that (i) there is little or no correlation between the two, and (ii) a simple static view of software is insufficient to assess software quality, and that software quality is dependent on factors that are not amenable to measurement via metrics.

Recent work by Ó Cinnéide et al. also casts grave doubt on the ability of structural software metrics to measure the software properties that they purport to measure [21]. In a study involving over 78,000 refactorings, they show that there is little agreement between a collection of popular cohesion metrics on whether a refactoring improves or disimproves cohesion. This means that whether a particular refactoring instance is deemed to improve or disimprove cohesion depends heavily on which cohesion metric is employed to make the decision. In the light of this result, discussing whether refactoring improves software quality metrics seems futile: it will and it won't depending on the metric employed.

# 4. DOES REFACTORING IMPROVE NON-FUNCTIONAL REQUIREMENTS?

Non-functional requirements such as maintainability, flexibility etc. (referred to informally as the 'ilities') are arguably better ways of assessing if refactoring has truly improved software quality. After all, if refactoring can make software more maintainable, who cares if code smells are removed or structural cohesion improves?

Wilking et al. [30] performed a study with 12 students where each participant was asked to develop a simple game program in C. Half of the participants were encouraged to refactor and the other half encouraged to document their code. After the development was completed, the participants were asked to correct a number of injected bugs and were given a number of new requirements to implement. The time taken to complete these tasks was measured and com-

pared. The refactored code displayed no significant improvement in either maintainability or modifiability.

Kim et al. [13] investigated the role of API-level refactorings during the evolution of several open source applications, namely Eclipse, jEdit and Columba. One aspect they studied is the impact refactoring has on bug rate and on the time taken to fix bugs. They found that refactoring reduced the time taken to fix bugs, in keeping with the popular opinion that refactored code is more maintainable. However they found that the bug rate actually *increased* after refactoring, and a significant number of these bugs (50%) were as a result of the refactoring activities themselves. So refactoring improved maintainability, but at a price of a higher bug rate – probably not a trade-off acceptable to most practitioners.

Kannangara and Wijayanake report on a controlled study investigating the impact of refactoring on internal and external software quality [12]. A small C# application was refactored to remove code smells and the refactored and unrefactored versions were presented to two developer cohorts who were asked to answer a questionnaire about the code and to perform a number of fixes of injected bugs. The refactored code was *not* found to be superior to the unrefactored code in any of the areas examined. Interestingly, this paper was posted to the Linkedin Refactoring Group [15], a group of over 3,000 software professionals with an interest in refactoring, where it was largely vilified by practitioners who clearly see the practical value of refactoring and disbelieve an empirical study that suggests otherwise.

Again, studies exist that support the notion that refactoring can improve the 'ilities.' Canfora et al. [5] carried out an exploratory study on the evolution of four open source systems, namely ArgoUML, Eclipse-JDT, Mozilla, and Samba, with the aim of analyzing the relationship between source code 'change entropy' and refactoring activities (amongst others). They found that refactoring activity led to a reduction in change entropy, which can be interpreted as an improvement in maintainability.

In one of the most comprehensive refactoring field studies to date, Kim et al. [14] surveyed 328 professional software engineers at Microsoft, interviewed six members of a team tasked with refactoring Windows 7, and conducted a quantitative analysis of the Windows 7 version history. When surveyed, the developers cited the main benefits of refactoring to be: improved readability (43%), improved maintainability (30%), improved extensibility (27%) and fewer bugs (27%). When asked what provokes them to refactor, the main reason provided was poor readability (22%). Only one 'official' code smell was mentioned, that being code duplication (13%). In the subsequent interviews, the main reason given for the radical refactoring of Windows 7 was dependency reduction. In their quantitative study of Windows 7 their findings relevant to this paper were (1) refactoring is correlated with the decrease of inter-module dependencies, (2) some complexity measures were improved by refactoring (e.g., fan-in), but this did not apply to all complexity measures (3) larger modules (in terms of LOCs) were *not* more likely to be refactored, and refactoring tended to make modules *larger* (4) modules exhibiting greater code churn were *not* more likely to be refactored, and the decrease in code churn for refactored code was *less* than that of unrefactored code (5) modules exhibiting a greater degree of crosscutting changes (effectively shotgun surgery) were *not* more likely to be refactored, and refactored modules tended to exhibit an *increased* degree of crosscutting changes. Finding 1 fits the traditional view that refactoring is used to decrease inter-module dependency and Finding 2 partially fits the traditional view that refactoring reduces complexity measures. However, Findings 3, 4 and 5 all contradict the received wisdom that refactoring should decrease module size, reduce churn and reduce the degree of crosscutting changes. It is possible to explain these results, e.g. Kim et al. suggest that the modules that most require refactoring exhibit little churn as developers are afraid to touch them, and, by extension the relative post-refactoring increase in churn is due to the ease of extending these modules. While such a nuanced argument may transpire to be valid, to accept it would require a radical rethinking of what is currently regarded as indicators of software quality.

# 5. DISCUSSION AND CONCLUSIONS

In summary, a significant number of works attempt to evaluate the impact of refactoring on various aspects of code quality, but in general the results fail to show a clear link between refactoring and either a reduction in code smells or an improvement in overall quality. Nevertheless refactoring appears to be a very popular practice in the software industry and the likelihood is that the studies to date have not captured the full picture of refactoring praxis.

As already observed, there is no agreement on what is and is not a code smell, and software metrics cannot be accepted as reliable measures of what they purport to measure. Worse again, there is no consensus on what is and is not a refactoring, both in terms of what practitioners regard as refactoring [14] and the fact that researchers have no agreed approach to refactoring detection [8, 20]. Given that the treatments that are being applied cannot be determined with 100% precision and recall and that effects that are being measured are similarly vague, it is not surprising that empirical studies in the refactoring area have yielded a 'mixed bag' of sometimes conflicting results.

If we are to really understand the benefits/risks of refactoring and hence be able to perform appropriate cost/benefit analysis, we propose that the following steps are essential:

1. We need to create more *direct* measurements of software quality, e.g. effort, number of defects, etc. instead of proxy measures like code smells, cohesion, coupling, maintainability index, etc.

2. We need to be more systematic when it comes to describing the context in which refactoring takes place, in order to know under which circumstances refactoring proves to be beneficial and when not.

3. We need to perform more long-term studies where refactorings, maintenance problems and the aforementioned direct measures are available in order to understand better where in the evolution process refactorings are triggered, for which reasons, and what their effects are in later stages of the system's evolution.

Overall, it is better to identify actual problems experienced by developers during maintenance and observe how they are addressed in their real context rather than to propose further artificial constructs based on our assumptions of what constitutes a code smell or a refactoring. By using a more inductive approach to research we could perhaps be more successful at operationalising more realistic constructs of code smells and refactorings, and hence build more effective tool support.

# 6. REFERENCES


[1] M. Alshayeb. Empirical investigation of refactoring effect on software quality. *Information & Software Technology*, 51(9):1319–1326, 2009.

[2] M. Alshayeb. Refactoring effect on cohesion metrics. In *International Conference on Computing, Engineering and Information*, Apr. 2009.

[3] K. Beck. Manifesto for Agile Software Development, 2001.

[4] L. Bossavit. *The Leprechauns of Software Engineering*. Leanpub, 2014.

[5] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta. How changes affect software entropy: an empirical study. *Empirical Software Engineering*, 19(1):1–38, 2014.

[6] A. Chatzigeorgiou and A. Manakos. Investigating the Evolution of Bad Smells in Object-Oriented Code. In *International Conference on the Quality of Information and Communications Technology (QUATIC'2010)*, pages 106–115. IEEE, Sept. 2010.

[7] S. Counsell, R. M. Hierons, H. Hamza, S. Black, and M. Durrand. Exploring the Eradication of Code Smells: An Empirical and Theoretical Perspective. *Advances in Software Engineering*, 2010:1–12, 2010.

[8] F. A. Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):1–38, 2012.

[9] F. A. Fontana, J. Dietrich, W. Bartosz, A. Yamashita, and M. Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2016.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] T. Hall, M. Zhang, D. Bowes, and Y. Sun. Some code smells have a significant but small effect on faults. *ACM Trans. Softw. Eng. Methodol.*, 23(4):33:1–33:39, Sept. 2014.

[12] S. H. Kannangara and W. Wijayanayake. An Empirical Evaluation of Impact of Refactoring on Internal and External Measures of Code Quality. *International Journal of Software Engineering & Applications*, 6(1), Jan. 2015.

[13] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of API-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 151–160, New York, 2011. ACM.

[14] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Trans. Softw. Eng.*, 40(7):633–649, July 2014.

[15] Linkedin Refactoring Group, 2016.

[16] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[17] S. McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004.

[18] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[19] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In B. Meyer, J. R. Nawrocki, and B. Walter, editors, *Balancing Agility and Formalism in Software Engineering*, pages 252–266. Springer-Verlag, 2008.

[20] E. Murphy-hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[21] M. Ó Cinnéide, I. Hemati Moghadam, M. Harman, S. Counsell, and L. Tratt. An experimental search-based approach to cohesion metric evaluation. *Empirical Software Engineering*, pages 1–38, 2016.

[22] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, Department of Computer Science, Champaign, IL, USA, 1992.

[23] W. F. Opdyke and R. E. Johnson. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications*. ACM, Sept. 1990.

[24] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia. Do they really smell bad? A study on developers' perception of bad code smells. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, Sept 2014.

[25] C. Simons, J. Singer, and D. R. White. Search-based refactoring: Metrics are not enough. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering*, volume 9275 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2015.

[26] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyba. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, Aug. 2013.

[27] Q. Soetens and S. Demeyer. Studying the effect of refactorings: A complexity metrics perspective. In *Seventh International Conference on the Quality of Information and Communications Technology (QUATIC'2010)*, pages 313–318, Sept 2010.

[28] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality*, WoSQ '07. IEEE Computer Society, 2007.

[29] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 132–146, Riverton, NJ, USA, 2013. IBM Corp.

[30] D. Wilking, U. Khan, and S. Kowalewski. An Empirical Evaluation of Refactoring. *e-Informatica Software Engineering Journal*, 1(1), 2007.

[31] A. Yamashita and L. Moonen. To what extent can maintenance problems be predicted by code smell detection? - an empirical study. *Information and Software Technology*, 55(12):2223–2242, 12 2013.

[32] M. Zhang, T. Hall, and N. Baddoo. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(3):179–202, 2011.