How Good Are Code Smells for Evaluating Software Maintainability? Results from a Comparative Case Study

Conference Paper · September 2013

DOI: 10.1109/ICSM.2013.97

CITATIONS
11

READS
550

1 author:

Aiko Yamashita
Oslo Metropolitan University
46 PUBLICATIONS **1,275** CITATIONS

SEE PROFILE

How Good are Code Smells for Evaluating Software Maintainability? - Results from a Comparative Case Study -

Aiko Yamashita
Mesan AS &
Simula Research Laboratory, Norway
Email: aiko@simula.no

***Abstract*—An advantage of code smells over traditional software measures is that the former are associated with an explicit set of refactorings to improve the existing design. Past research on code smells has emphasized the formalization and automated detection of code smells, but much less has been done to empirically investigate how good are code smells for evaluating software maintainability. This paper presents a summary of the findings in the thesis by Yamashita [1], which aimed at investigating the strengths and limitations of code smells for evaluating software maintainability. The study conducted comprised an outsourced maintenance project involving four Java web systems with equivalent functionality *but* dissimilar implementation, six software professionals, and two software companies. A main result from the study is that the usefulness of code smells differs according to the granularity level (e.g., whether the assessment is done at file or system level) and the particular operationalization of maintainability (e.g., maintainability can be measured via maintenance effort, or problems encountered during maintenance, etc). This paper summarises the most relevant findings from the thesis, discusses a series of lessons learned from conducting this study, and discusses avenues for new research in the area of code smells.**

***Keywords*- Code smells; Bad smells; Empirical study; Comparative Case Study, Software maintenance; Software quality.**

I. INTRODUCTION

Code smells are indicators of software design shortcomings that can potentially decrease software maintainability. An advantage of code smells over traditional software measures (such as maintainability index or cyclomatic complexity) is that code smells are associated with an explicit set of refactoring strategies. Thus, code smells can potentially be used to support both *assessment* and *improvement* of software maintainability.

Nevertheless, it is not clear how and to which extent code smells can reflect or describe how maintainable a system is. This makes the interpretation and use of code smells somewhat difficult and hinders the possibility of conducting cost-effective refactoring. Given that refactoring represents a certain level of risk (e.g., introduction of defects) and cost (e.g., time spent by developers modifying the code and cost of regression testing), it is essential to weigh the effort and risks of eliminating versus ignoring the presence of code smells. Furthermore, it is important to understand which maintenance aspects can be addressed by code smells and which should be addressed by other means.

Insufficient information on maintenance aspects, such as severity levels and the range of effects of code smells, makes refactoring prioritization a nontrivial task. To support cost-effective refactoring, we need to increase our understanding of how code smells affect maintenance, what kinds of difficulties they cause, and how they can affect productivity in a project. If we are to use code smells to assess (and improve) maintainability, we also need to understand better *when* they are useful (e.g., the contexts in which their "predictive power" is acceptable) in predicting maintainability, and *which* aspects of maintainability they can measure best. The thesis summarized in this paper investigates how good code smells are in: (1) Reflecting system-level maintainability of software (and how they compare to other system-level assessment approaches), (2) Identifying source code files that are likely to require more maintenance effort (time) than others, (3) Discriminating between source code files that are likely to be problematic and those that are not likely to be so during maintenance, and (4) Reflecting maintainability aspects that are as deemed critical by software developers. In order to answer these questions, a Comparative Case Study was conducted. The study consisted of outsourcing and observing a maintenance project involving 4 Java web systems, 6 developers and 2 software companies during seven weeks. The reminder of this paper describes how this study was conducted, what were the findings, and discusses some lessons learned from this experience. More specifically: Section 2 describes the methodology followed in the thesis, Section 3 presents the main findings from this research, and discusses some lessons learned from this experience, and Section 4 discusses potential avenues for research in the area of code smells.

II. DESCRIPTION OF THE RESEARCH STUDY

To investigate the usefulness of code smells, a maintenance project was carried out and observed as part of a comparative case study. This project involved four Java web systems, six software professionals, and two software companies. The remainder of this section will provide details for each of the elements involved in the study, and the design of the study.

A. The systems under study

In 2003, Simula Research Laboratory's Software Engineering department sent out a tender for the development of a web-based information system to keep track of their

empirical studies and resulting scientific publications. Based on the submitted bids, four Norwegian consulting companies were hired to independently develop a version of the system using the same requirements and specifications. The four development projects led to the creation of four systems with the same functionality. We will refer to them as System A, System B, System C, and System D. The systems were primarily developed in Java, and they all have similar three-layered architectures. Although the systems exhibit nearly identical functionality, there were substantial differences in how they were designed and coded. The systems were deployed in 2003 over Simula Research Laboratories' Content Management System (CMS), but in 2007, due to changes in the CMS, it was not longer possible for the systems to remain operational. This provided a realistic setting for a maintenance project based on a real need for adapting and enhancing the systems.

B. The tasks and the developers

The maintenance project involved three tasks: 1) To modify the systems so they could operate in the new CMS environment of Simula Research Laboratory (i.e., an adaptive maintenance task where a Data layer based on a relational DB was to be replaced with a series of calls to external web services), 2) To modify the authentication system by means of consuming a web service, and 3) To extend the systems with a new functionality, which would allow the users to build customisable reports. Two Eastern European software companies were hired to carry out the maintenance tasks between September and December 2008 at a total cost of 50,000 Euros. Thus, the case study was conducted in a way that resembled, as much as possible, a real-life consultancy project. The developers were recruited from a pool of 65 participants of a previously completed study on programming skill [2], and were evaluated to have sufficient English skills for the purpose of our study.

C. Study design

Having four functionally equivalent systems with different code enabled the design of a comparative case study, with software maintenance tasks embedded within almost identical *maintenance contexts* and differing in the variable of interest: *code smells*. The design of the study therefore, enables better control over the moderator variables, such as system functionality, tasks, programming skills, and development technology to better observe the relations between *code smells* (our variable of interest) and several dependent variables (i.e., different *maintenance aspects*), in a very similar way to experimental studies. In this study, we conducted both *theoretical* and *literal* replications[1], by asking each of the six developers to first conduct all tasks in one system and then to repeat the same maintenance tasks on a second system, resulting in 12 observations (six developers $\times$ two systems). Figure 1 describes the order in which the systems were assigned to each developer. This assignment was done randomly.

		Developer					
		1	2	3	4	5	6
Round	1	A	B	C	D	C	A
	2	D	A	D	C	B	B

Figure 1. Assignment of systems to developers in the case study.

D. Study protocol

First, the developers were given an overview of the tasks (e.g., the motivation for the maintenance tasks and the expected activities). Then they were provided with the specification of the maintenance tasks. When needed, they could discuss the maintenance tasks with a researcher; one was always present at the site during the entire duration of the project. We had daily meetings with the developers where we tracked their progress and the problems they encountered.

Think-aloud sessions were conducted every other day at a random point during the day, and they lasted for 30 minutes. Acceptance tests and individual open interviews, which had a duration of 20-30 minutes, were conducted once all three tasks were completed. In the open-ended interviews, the developers were asked about their opinions of the system, e.g., about their experiences when maintaining it.

Eclipse was used as a development tool along with MySQL[2] and Apache Tomcat[3]. Defects were registered in Trac[4], and Subversion[5] was used as the versioning system. A plug-in for Eclipse called Mimec [4] was installed on each developer's computer to log all the user actions performed at the graphical user interface (GUI) level with millisecond precision.

E. Variables observed

Figure 2 describes the moderator variables (those we control in the analysis), the variables of interest (those whose relationships we analyze), and the data sources for the variables. The variables of interest within this study are as follows: *(1) Code smells*: Number of code smells and code smell density (code smells/kLOC). The definition of the smells analyzed is provided in [1]. Two commercial tools were used (Borland Together® [5] and InCode [6]) to detect code smells. *(2) Developers' perception of the maintainability of the systems*: This includes subjective and qualitative aspects of maintainability reported by each developer

[1] In *literal replication*, cases that are similar in relation to certain variable(s) are expected to support the analysis of each and give similar results. When *theoretical replication* is used, the cases that vary on the key variable(s) are expected to have different results [3].

[2] http://www.genuitec.com
[3] http://tomcat.apache.org
[4] http://trac.edgewall.org
[5] http://subversion.apache.org

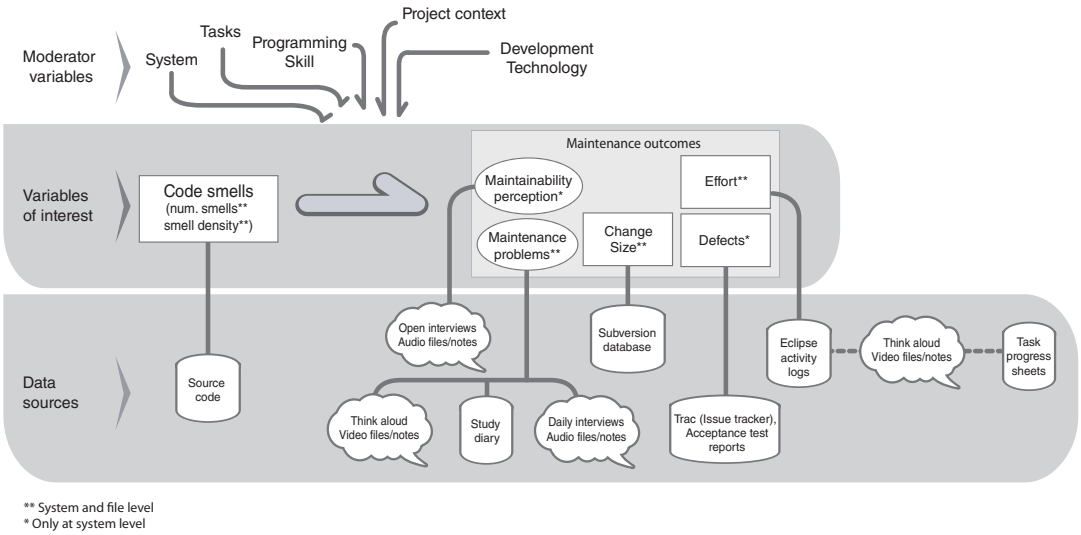

Figure 2. Illustration of variables involved in the study and the corresponding data sources.

once the three maintenance tasks of one system had been completed. *(3) Maintenance problems encountered by the developers during maintenance*: These include a qualitative aspect of the maintenance process based on problems reported through interviews or think-aloud sessions or observed by the researcher during the maintenance work. *(4) Change size*: This variable constitutes an outcome variable from the maintenance project, reflecting the sum of LOC added, changed, and deleted (i.e., file churn). *(5) Effort*: Another maintenance outcome, measured by time spent on the tasks and each of the files in the systems, *(6) Defects introduced during maintenance*: This variable is an aspect of the quality of the system after the tasks were completed.

Figure 2 also discriminates between outcomes/aspects that were observed at the system level (one asterisk) and at both system and file levels (two asterisks). The figure also distinguishes maintenance problems and maintainability perception–which are categorized as *qualitative aspects* (circles)–from change size, effort, and defects–which are categorized as *quantitative outcomes* (squares). This Figure also depicts the data sources from which each of the maintenance outcomes/aspects was derived.

F. Analysis Conducted

To investigate "How good are code smells in reflecting system-level maintainability of software (and how they compare to other system-level assessment approaches)", code smells were aggregated at the system level, and each system was ranked according to the *amount of code smells* they contained and their *code smell density* (i.e., less code smells and lower smell densities mean better maintainability ranking of a system). After the systems had undergone maintenance work, the maintenance outcomes: *total effort*, and the *introduced defects* at the system level were collected per system to rank them accordingly. To avoid the learning effect problems, we used only the data from the first round per developer. Cohen's kappa coefficient[6] was used to statistically measure the degree of agreement between the code-smell-based and maintenance-outcome-based rankings. Previous maintainability assessments of the systems based on a subset of C&K metrics and expert judgments, as reported in Ref. [7], were also compared with the maintenance-outcome-based rankings to analyze the differences in accuracy between the code-smell-based, expert-judgment-based, and metrics-based approaches for maintainability assessments.

To investigate "How good code smells are on identifying source code files that are likely to require more maintenance effort (time) than others", we focused on Java files as the unit of analysis and used *multiple regression analysis*. Effort at *file level* (effort used to view or update a file) was the variable to be explained. Variables representing the different code smells, the file size (measured in LOC), the number of revisions on a file, the system, the developer, and the round were included as independent variables. Several regression models, with different subsets of variables, were built to compare their fit and to discern the predictive capability of each of the variables considered.

To investigate "How good code smells are on discriminating between source code files that are likely to be problematic and those that are not likely to be so during maintenance", we focused on Java files as the unit of analysis and by used binary logistic regression analysis. The variable to explain was the variable "problematic," which was *true* (1) if a file was deemed problematic during maintenance by at least one developer who worked with the file, but *false* otherwise (0). The different types of code smells, files size

[6] Cohen's kappa coefficient is a statistical measure to represent inter-rater agreement for categorical items.

(measured in LOC), and change size (churn) were used as independent variables. A follow-up qualitative analysis based on the data from the interviews and the think-aloud sessions was performed (1) to support/challenge the findings from the binary logistic regression and (2) to understand better how the presence of a code smell contributed to the problems experienced by the developers during maintenance.

To investigate "How good code smells reflect maintainability aspects that are as deemed critical by software developers", we conducted qualitative analysis, which compared the developers' perceptions on the maintainability of the systems with the goal of identifying a set of factors relevant to maintainability. These factors were related to current definitions of code smells to observe their conceptual relatedness. The transcripts of the open-ended interviews were analyzed through *open* and *axial coding* [8]. The identified factors were summarized and compared across cases using a technique called *cross-case synthesis* [3]. The factors derived from this analysis were compared with the factors reported in a previous study [7], which were extracted via expert judgment.

III. Findings and Learnings

This section discusses the main findings from this thesis and some lessons learned along the research process.

A. Finding 1: Aggregated code smells are not so good indicators of system-level maintainability

System-level indicators of maintainability based on aggregated number of code smells were investigated in the four systems where a system's maintainability was ranked according to code smell measures and compared with respect to the maintenance outcome measures – change effort, and number of defects. Given that many code smells definitions are based on size-related parameters, aggregating them at system level would not provide more information than the more traditional *system size* measured in LOC. It was found that expert-judgment-based assessment was the most flexible of all the three approaches (i.e., code smell, C&K metrics, and expert-judgment) because it considered both the *effect* of the system size and the potential *maintenance scenarios* (e.g., small versus large extensions). However, we found that if smell density (code smells/LOC) is used to compare systems of similar size, it is likely to provide a more accurate assessment than the expert-judgment based one. We conclude that an advantage of the use of code smells is that when comparing similarly sized systems, they can spot critical areas that experts may overlook. This finding is reported in detail in [9].

B. Finding 2: Code smells are not good indicators of effort at file level

We conducted multiple regression, including the number of different code smells per file, the file size (LOC), and number of changes in the file, as the explanatory variables for file effort. We found that although with an R^2 if 0.58, the only code smell that constituted a significant independent variable of effort ($\alpha < 0.01$) in the regression model is *Refused Bequest*, which interestingly, displayed a *decreasing* effect on effort. If we exclude the code smells from this model (i.e., leaving only file size and number of changes as variables), the R^2 remained at 0.58. This implies that code smells may not provide additional explanatory power than file size and the number of revisions in the context of explaining effort usage per file. This finding is reported in detail in [10]

C. Finding 3: Code smells may be promising indicators of problematic files during maintenance

To investigate the capability of code smells to uncover problematic codes, binary logistic regression was conducted. We built a model in which the variables related to the different code smells, the file size, and the file churn were entered in a single step. The R^2 values (Hosmer & Lemeshow = 0.864, Cox & Snell = 0.233, and Nagelkerke = 0.367) confirm that our model provided a reasonably good fit of the data. In the model, the odds ratio for the code smell *ISP Violation* was the largest [Exp(B) = 7.610, $p = 0.032$], which suggests that this code smell was able to explain much of the maintenance problems at the file level. The model also finds Data Clump[7] as a significant contributor [Exp(B) = 0.053, $p = 0.029$], but contrary to ISP Violation[8], this code smell indicates less maintenance problems. Some of the problems caused by ISP Violation were: (1) Error propagation, (2) Change propagation, (3) Difficulties identifying the task context, and (4) Confusion due to inconsistent design. This finding, is reported in detail in [11].

D. Finding 4: Some code smells may deserve more attention from a practical maintenance perspective

Several factors were identified as critical by the developers: appropriate technical platform, coherent naming, design suited to the problem domain, encapsulation, inheritance, (proprietary) libraries, simplicity, architecture, design consistency, duplicated code, initial defects, logic spread, and use of components, where *design consistency* was considered as one of the most important factors. We found that there are code smells capable of supporting the analysis of the several of the maintainability factors–*encapsulation*, *design consistency*, *logic spread*, *simplicity*, and u*se of components*.

For example, *simplicity* is a factor traditionally addressed by static analysis means, but it is also closely related to God Class, God Method, Lazy Class, Message Chains, and Long Parameter List. Similarly, *logic spread* is related to

[7] Clumps of variables appearing repetitively across the code. [8] The violation of the *Interface Segregation Principle*, which dictates that there should not be 'all-purpose' interfaces with wide-spread incoming dependencies.

Feature Envy, Shotgun Surgery, and ISP Violation, and *design consistency* is related to several code smells: Alternative Classes with Different Interfaces, ISP Violation, Divergent Change, and Temporary Field. We concluded that in some cases, code smells would need to be complemented with alternative approaches, such as expert judgment (see Refs. [7, 12]) and semantic analysis techniques (for example, see Maletic et al. [13]) to achieve a comprehensive assessment of maintainability. This finding is reported in detail in [14].

E. Finding 5: Code smells can 'interact' with each other, or with other types of design shortcomings

In this study, we discovered that interaction effects occur between code smells and also between other kinds of design shortcomings, often causing more maintenance problems than when interactions do not occur. Code smells that appear together in the same artifact (file) can interact with each other, but also interaction effects can occur between code smells that are distributed across *coupled* artifacts (e.g., artifacts that display data/functional dependencies). Consequently, in practice, there is no difference between the interaction effects of *coupled smells* and the interaction effects of *collocated smells*. This finding has considerable implications for further studies on code smells, since it means that, to get a more complete understanding of the role of code smells in software maintenance, dependency analysis should be included in the code smell analysis process. This finding is reported in detail in [15].

F. Lessons learned

Despite the intricacies and challenges involved, this study design allowed us to conduct both *theoretical* and *literal* replication in a case study, which is often very difficult to attain. This enabled the cross-validation of the observations across cases, and strengthened the internal validity of findings derived from qualitative sources. By combining qualitative and quantitative data collection, we were also able to used a mixed method approach, which not only can identify trends or connections between variables, but also help to derive theories or explanations for those relationships.

Although the element of 'control' introduced a certain degree of *artificiality* in a case study, the degree of "intrusiveness" was not found to differ greatly from "normal" case studies. Considering the fact that designs as the one presented here can provide richness in details (from an in-vivo context) that cannot be achieved by experimental settings, we suggest that this approach may deserve further exploration and usage.

One particular challenge was that although it is important to adhere to protocols in normal case studies, within case studies that need a certain element of control, this becomes of paramount importance for the validity of certain results. Since the environment for studies of this nature often constitute an industrial setting, we often face situations where certain factors cannot be controlled for. Consequently, it is important to prepare adequately beforehand, building contingency plans on potential issues from *practical* and *research* perspective. This study also showed the importance of a study protocol that can be "tested" through a pilot study. Pilot studies can allow the identification of potential threats to validity, and practical issues not contemplated in the protocol. A pilot study helps also to adjust the time that is allocated to each of the different activities (project-related and research related), which can support better management of the study, and the prioritization of data collection activities. Consequently, whenever possible, we strongly recommend to run a pilot study, even if it constitutes a down-scaled version of the final study.

Another great challenge of this study after the data was collected, was the summarisation, integration and analysis of the data. In particular, processing and indexing large amounts of log-data (e.g., the MimEc logs amounted for 1400 hours in approx. 87MB of .csv files) and qualitative data (e.g., 3000 minutes for daily interviews and 480 minutes of recorded audio from the open interviews) was found extremely time consuming. Also, the integration of the different data sources for the purposes of data triangulation had to be automated via a Java program written for that purpose. Although we counted with a structured and functional repository for storing the data from the study, more "on the fly" analysis and summarization would have been preferred, as this would have eased the navigation of the data in latter stages. In our study, the lack of human resources during the study execution limited the amount of "on the fly" analysis that could have been conducted otherwise.

A very useful approach for facilitating the indexing of data in this study was a *logbook*, where the researcher logged all her observations during the study. This was an invaluable resource to pinpoint and identify observations of interest that could be further examined in through other data sources, and to synchronise time-wise the different events during the project across the different data sources. For example, as the logbook contained the dates of the observations, it was an intuitive way to navigate the progress of the project and validate the events via multiple sources sharing the same date, such as the SVN or the progress report written by the developers.

In conclusion, an adequate balance should be seek for; for example, "how much data" should be collected, versus how much resources should be used in terms of time, staff and funds. Our experiences in this study warns researchers against focusing on a too wide research scope, in detriment of an adequate number of human resources/staff that can carry out, summarise and analyse the data in a large study.

IV. Future work for the community

The areas for future work identified through this research include the following:

1) Interaction effects among code smells: More focus is needed on the implications of combinations of code smells (and other types of design flaws) on maintainability instead of investigating only the effects of individual code smells (this corresponds with the ideas of Walter and Pietrzak [16]). This entails building more comprehensive symptomatic characterizations of different types of potential maintenance problems (e.g., in the form of *inter-smell relations*) and uncovering the causal mechanisms that lead to them.

2) Study of collocated smells and coupled smells: More focus is needed on dependency analysis alongside the analysis of interaction effects across code smells. We suggest this among others because interactions between code smells can occur across coupled files. This interaction is currently ignored due to the fact that code smells are mostly analyzed at the file level and "coupled code smells" are not identified. Also, the dependency analysis should focus on *types* of dependencies (e.g., data, functional, abstract definition, and inheritance) and their *quantifiable attributes* (e.g., intensity, spread, depth).

3) Nature and severity of maintenance problems: Future work should focus on quantifying the severity and the degree of the impact of different types of maintenance problems in different contexts to establish the *relative* importance and context dependency of code smells. This way, it may be possible to assess not only whether and how code smells cause maintenance problems, but also *how much* those problems matter on concrete outcomes of maintenance projects compared with other problems and in different contexts.

4) Cost-/benefit-based definition/detection of code smells: Further research should focus on defining and extending a catalog of design factors that have empirical evidence of their relevance on maintainability. This catalog may be used to guide further efforts in new definitions of code smells and corresponding detection methods/tools.

As a final remark, we cite Hannay et al. [17] who asserted that theory-driven research is not yet a major issue in empirical software engineering and referred to several articles that commented explicitly on the lack of relevant theory. Case study research could significantly contribute to the development of theories from observations in relevant fields and contexts (i.e., *inductive research* [3]). Runeson [18] equally argues for the adequacy of case studies in the Software Engineering field, given the commonalities of our discipline to fields as Social and Political Sciences, where the complexity of the context plays an intrinsic role on the different phenomena being investigated.

We hope that the example and experiences described here can help further on the design and conduction of "mixed approaches" not only combining 'quantitative' and 'qualitative' techniques, but also combining features from 'case studies' with 'experimental' approaches in the field of Software Engineering.

REFERENCES

[1] A. Yamashita, "Assessing the Capability of Code Smells to Support Software Maintainability Assessments: Empirical Inquiry and Methodological Approach," Doctoral Thesis, University of Oslo, 2012.

[2] G. R. Bergersen and J.-E. Gustafsson, "Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective," *Journal of Individual Differences*, vol. 32, no. 4, pp. 201–209, 2011.

[3] R. Yin, *Case Study Research : Design and Methods (Applied Social Research Methods)*. SAGE, 2002.

[4] L. M. Layman, L. A. Williams, and R. St. Amant, "MimEc," in *Int'l Ws. Cooperative and Human Aspects of Softw. Eng. (CHASE)*. New York, New York, USA: ACM Press, 2008, pp. 73–76.

[5] Borland, "Borland Together. http://www.borland.com/us/products/together. Accessed 10 May 2012," 2012.

[6] Intooitus, "InCode. http://www.intooitus.com/inCode.html. Accessed 10 May 2012," 2012.

[7] B. C. D. Anda, "Assessing Software System Maintainability using Structural Measures and Expert Assessments," in *IEEE Int'l Conf. Softw. Maintenance (ICSM)*, 2007, pp. 204–213.

[8] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE, 1998.

[9] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An Empirical Study," *Journal of Systems and Software*, 2013.

[10] D. Sjøberg, A. Yamashita, B. Anda, A. Mockus, and T. Dybå, "Quantifying the Effect of Code Smells on Maintenance Effort," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2013.

[11] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data," *Empirical Software Engineering*, pp. 1–33, 2013.

[12] M. Jorgensen, "Estimation of Software Development Work Effort:Evidence on Expert Judgment and Formal Models," *International Journal of Forecasting*, vol. 23, no. 3, pp. 449–462, 2007.

[13] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Int'l Conf. Softw. Eng. (ICSE)*, ser. ICSE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 103–112.

[14] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *IEEE Int'l Conf. Softw. Maintenance (ICSM)*, 2012, pp. 306–315.

[15] A. Yamashita and L. Moonen, "Exploring the Impact of Inter-Smell Relations on Software Maintainability: An Empirical Study," in *Int'l Conf. Softw. Eng. (ICSE)*, 2013, pp. 682–691.

[16] B. Walter and B. Pietrzak, "Multi-criteria Detection of Bad Smells in Code with UTA Method 2 Data Sources for Smell Detection," in *Extreme Programming and Agile Processes in Softw. Eng. (XP)*. Springer Berlin / Heidelberg, 2005, pp. 154–161.

[17] J. E. Hannay, D. I. K. Sjøberg, and T. Dybå, "A Systematic Review of Theory Use in Software Engineering Experiments," *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 87–107, 2007.

[18] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.