Influencing Factors on Code Smells and Software Maintainability: A Cross-Case Study

Tassio Vale[1], **Iuri Santos Souza**[2], **Cláudio Sant'Anna**[2]

[1]Centro de Ciências Exatas e Tecnológicas - CETEC
Universidade Federal do Recôncavo da Bahia (UFRB) - Cruz das Almas, BA - Brazil

[2]Departamento de Ciência da Computação - DCC
Universidade Federal da Bahia (UFBA) - Salvador, BA - Brazil

`tassio.vale@ufrb.edu.br, iuri.souza@ufba.br, santanna@dcc.ufba.br`

Abstract. *Code smell is a concept referring to code that needs refactoring and can degrade aspects such as understandability and changeability. To the best of our knowledge, there is a lack of qualitative studies investigating influencing factors and mitigation strategies for code smells, that play an important role to improve software evolution, quality and productivity. As consequence, this paper investigates, in real-world scenarios, influencing factors for code smells based on a software developer point-of-view. We performed three case studies by interviewing software developers from three different real-world systems, by collecting both qualitative and quantitative data to generate reliable research evidence.*

1. Introduction

Code smell is a concept referring to code that needs refactoring and can degrade aspects such as understandability and changeability [Fowler et al. 1999]. In addition, it can be related to software faults. Code smell detection approaches [Yamashita and Counsell 2013] and tools [Fontana et al. 2011] are suitable indicators of the need for maintenance in a way that other purely metric-based approaches lack.

In the literature, there are proposals investigating the relationship among code smells and maintainability. Evidence point software with bad smells are more change-prone than others [Khomh et al. 2009], and components infected by code smells exhibit a different change behavior [Olbrich et al. 2009]. In particular, architecture is influenced by code smells, since they often entail modularity problems in the evaluated systems [Macia et al. 2011].

[Zhang et al. 2011] performed a literature review, analyzing the most relevant papers in this area. The evidence indicate there is a higher research attention for specific smells such as Duplicated Code, and little attention on other ones (e.g. Message Chains). In addition, their findings show very few studies report on the impact of code smells in software development.

There is a lack of qualitative studies investigating influencing factors and mitigation strategies for code smells, that play an important role to improve software evolution, quality and productivity. In order to mitigate code smells in a given software, it is important to identify and manage factors that influences the emergence of them. Such influencing factors might be since time pressure for releasing a software until lack of information about code smells and their associated risks. Consequently, a causality analysis in real-world scenario from both qualitative and quantitative perspectives is essential.

This paper investigates factors that influence the emergence of code smells and possible mitigation strategies based on a software developer point-of-view. We applied a cross-case analysis to investigate multiple case studies seeking for empirical evidence on a specific context. Three cases provided qualitative and quantitative data from different real software development projects, with different developers. These results can indicate which factors should be managed, in order to decrease the emergence of smells in source code and consequently decreasing maintainability issues.

The remainder of this paper is structured as follows: first, we present related work. Section 3 describes the activities of the proposed the research design. Section 4 presents findings from our analysis, and Section 5 synthesizes the gathered evidence. Section 6 argues on threats to validity of this research. Finally, Section 7 summarizes findings and presents future work.

2. Related Work

[Katzmarski and Koschke 2012] investigate whether metrics agree with complexity as perceived by programmers. Their findings point programmers' opinions are quite similar and only few metrics and in only few cases reproduce complexity rankings similar to human raters. Our work focus on the influencing factors instead of analyzing the metrics associated to code smells.

[Yamashita and Moonen 2012] reports on an empirical study that investigates the extent to which code smells reflect factors affecting maintainability that have been identified as important by programmers. Furthermore, [Sjoberg et al. 2013] reported an empirical study that investigated the relationship between code smells and maintenance effort using the same study object from [Katzmarski and Koschke 2012].

According to [Zhang et al. 2011], the research attention is turned to code smells and related maintainability issues. We investigate influencing factors, what makes a software development environment proper for these code anomalies. Our assumption is discovering those factors, we can mitigate code smells and improve maintainability.

3. Research Design

This work takes a constructivist or interpretive philosophical perspective, assuming there are multiple interpretations of a single event [Merriam 2009]. We performed three case studies by interviewing software developers as well as gathering quantitative data from three different real-world systems. The case studies are instrumental, since we intend to understand the construct and build theories. After, a cross-case analysis [Eisenhardt 1989] synthesize evidence and present findings.

The main research question reflects the goal of this work, and it is derived in three specific questions, described as following:

- **Main question: Which factors influence the emergence of code smells according to the software developer point-of-view?**

- **RQ1:** ***What do software developers know about code smells?***
  Rationale: our hypothesis is knowledge about code smells might influence developers opinions, since they cannot see the impact of code smells on software maintainability. Additionally, the absence of knowledge can be an influencing factor.

- **RQ2:** ***Which factors influence which code smells?***
  Rationale: in this question, we investigate the relationship of each code smell with the respective influencing factor(s). The developers should also explain why a given factor is related to a specific code smell.

- **RQ3:** ***How to manage these influencing factors?***
  Rationale: considering developers' experience, besides indicating factors that influence the emergence of code smells, they are also able to propose mitigation strategies for this context. These evidence can be evaluated in future research.

3.1. Data Collection

The first source of evidence for data collection is the set of metrics associated to code smells. Code smell detection tools provide results for such metrics. In this study, we adopted the inCode tool[1]. A brief definition of each code smell [Lanza et al. 2005, Fowler et al. 1999] analyzed by inCode is described as follows:

Code smells related to class analysis:

Data class is the manifestation of a lacking data encapsulation, and of a poor data-functionality proximity. By allowing other modules or classes to access their internal data, data classes contribute to a brittle, and harder to maintain design.

Tradition breaker when a derived class breaks the inherited "tradition" and provide a large set of services which are unrelated to those provided by its base class

Schizophrenic class is a class that captures two or more key abstractions. It negatively affects the ability to understand and change in isolation the individual abstractions that it captures.

God class is a design flaw refers to classes that tend to centralize the intelligence of the system. A God Class performs too much work on its own, delegating only minor details to a set of trivial classes and using the data from other classes.

Code smells related to method analysis:

Feature envy refers to methods that seem more interested in the data of other classes than that of their own class. These methods access directly or via accessor methods a lot of data of other classes.

Data clumps They represent groups of data that appear together over and over again, as parameters that are passed to operations throughout the system. They represent cases of bad/lacking encapsulation and have a negative contribution on the ease of maintaining those parts of the system that use the data clumps (by Fowler).

[1]inCode tool - Available at http://www.intooitus.com/products/incode/

Sibling duplication means duplication between siblings in an inheritance hierarchy. Two or more siblings that define a similar functionality make it much harder to locate errors because the assumption ‘only class X implements this, therefore the error can be found there”does not hold anymore.

Internal duplication means duplication between portions of the same class or module. For example, an operation that offers a certain functionality should be solely responsible for that functionality.

External duplication means duplication between unrelated capsules of the system.

Message chains corresponds to the situation in which a method calls many data exposer methods that belong to other classes (i.e. including also accessor methods, but not limited to these, as data exposers can be also static methods, that return an object that is part of that class).

The second source of evidence is the opinion of software developers concerning the research questions. As data collection instrument, we applied interviews with software developers, our unit of analysis. Interviews are effective to elicit information about things that cannot be observed, such as feelings, thoughts, and intentions [Merriam 2009].

The interviews are semi-structured, considering open questions to guide the interviewer. It gives freedom on the sequence of the questions and exact wording, allowing the extraction of useful and rich information. Two researchers conduct the interviews: the interviewer, asking questions and talking directly with interviewees, and the another researcher taking notes of expressions, behavior and gestures of the interviewees.

The researchers conduct interviews by initially running the inCode tool and collecting the results. After, they start the interview asking a set of five questions to obtain the developer background and a description of the analyzed software. Then, the interviewer shows the code smell quantitative results and, for each detected code smell, asks the next five questions concerning whether he consider it as a code smell, influencing factors and mitigation strategies. Finally, the interviewer try to get an overall opinion from the interviewee through the last two questions.

3.2. Data Analysis

The qualitative data analysis aims to interpret data obtained from different sources. This activity follows incremental and iterative steps. This work performed several iterations for data collection and analysis. Furthermore, the techniques used for data analysis come from basic qualitative research. In this sense, coding, categorization and synthesis are based on [Corbin and Strauss 2008].

The analysis activities aim to build categories, that can be themes, patterns or findings. We adopted the coding technique for categories construction. Coding consists of making notations in parts of the data that leads to potentially relevant information for answering the research questions [Merriam 2009]. The codes identified from each interview were compared to codes in other interviews. The constant comparison is a way to group codes into specific categories that represent the influencing factors of code smells as well as mitigation strategies for them.

As the process of data analysis progressed, relationships among categories were built, leading to explanatory propositions. Finally, core categories were chosen according to their general explanatory power, propositions emerged and a narrative was created to describe the central story of the case. We then interrogated this narrative to build a proposal of strategy to increase motivation in the organization.

3.3. Cross-Case Analysis

The cross-case method analyzes multiples cases studies seeking for empirical evidence on a specific fact [Yin 2008], synthesizing data, drawing inferences, and providing recommendations [Eisenhardt 1989]. It enables researchers to take a look beyond the initial insights, developing concrete findings based on them. [Yin 2008] emphasize that if similar results are found in both analyzed studies, the findings can be considered robust.

The benefits of adopting a cross-case approach [Glaser and Strauss 1967] are: ensuring evidence accuracy, establishing the generality of a fact, clarifying the relevant particulars of a case, testing some theory, and generating a theory (or establishing some definitions that should be followed).

The procedure adopted in this study is to select pairs of cases to the analysis, looking for similarities and differences among each pair [Eisenhardt 1989]. As result, we could identify new categories and concepts about the fields under study, which the investigators had not foreseen so far in the individual settings of each case study.

4. Results and Findings

Following the previously described research protocol, we present as follows the results of three investigated cases.

4.1. Case #1: National Transplant System

The National Transplant System (SNT) is a federal system responsible for managing the distribution of grafts for organ donation over the country. The software development team comprises five developers, and the system has implemented around nine thousand methods. The inCode quantitative results pointed the following code smels for SNT: *Data Class*, *Schizophrenic Class*, *God Class*, *Feature Envy*, *Data Clumps*, *Sibling Duplication*, *Internal Duplication*, *External Duplication* and *Message Chain*. Such results were used to interview two developers (with eight and six years of experience) of this team.

Considering *Data Class*, the developers consider this is not a code smell. According to them, classes present this characteristic due to architectural decisions, and every system has classes representing business domain data. Additionally, they state *Data Clumps* is not a code smell, since separate groups of data in specific classes does not make sense. For *Sibling Duplication*, the developers pointed the tool results as an error because it does not analyze classes semantically, and the presented duplications are semantically appropriate.

However, there are different opinions between the developers for *Feature Envy*. One developer states *Feature Envy* needs to be solved, and it was resulting from a bad design decision. The other developer understand this smell as a good design decision.

For the next five code smells, the developers agree they are relevant problems in the source code. *Schizophrenic Class*, *God Class* and *Message Chain* emerge due to architectural and design decisions, and specific refactoring can mitigate it. The influencing factors for *Internal Duplication* are certain technologies to support the system development and the lack of experience of some developers in the team. Training can help to mitigate this issue. *External Duplication* present different factors: lack of understanding on business processes and rules as well as low priority to perform refactoring and mitigate it.

4.2. Case #2: Dental Information System

RadioWeb is a software to support dentists during cephalometry and radiograph by applying specific annotations in such exams. This software has around seven hundred methods and built by a freelancer (the interviewee, with eight years of experience). The inCode quantitative results pointed the following code smells for RadioWeb: *Data Class*, *Schizophrenic Class*, *God Class*, *Feature Envy*, *Internal Duplication*, *External Duplication* and *Message Chain*.

Data Class and *Feature Envy* are not a code smells, according to the interviewee, since they comprise correct design decisions.

According to the developer, *Schizophrenic Class*, *God Class* and *Message Chain* emerge due to design decisions, and specific refactoring can mitigate it. *Internal Duplication* is caused by lack of understanding on business processes and rules, and influencing factor for *External Duplication* is the adopted technology to implement the system.

4.3. Case #3: UML Sequence Diagram Generator

Lirio is an academic project that consists of a compiler to generate UML sequence diagrams using information from the source code of a given software. This project was developed by two undergraduate students in one year. The inCode quantitative results pointed the following code smells for Lirio: *Data Class*, *God Class*, *Feature Envy*, *Data Clumps*, *Internal Duplication* and *External Duplication*. One of the developers, with ten years of experience, was interviewed.

The developer stated *Data Class* is not a code smell, since it is a good design decision concentrating data representation in specific classes. Furthermore, he does not agree with tool results for *Internal Duplication*, since it does not perform semantic analysis in the methods.

On the other hand, *God Class* is a code smell emerging due to short deadlines and lack of business understanding. Lack of communication among team members influences the emergence of *Feature Envy*. In addition, *Data Clumps* and *External Duplication* are result of bad design decisions. According to the developer, specific refactoring and longer deadlines are essential to mitigate these code smells.

5. Cross-Case Synthesis

The first question (RQ1) investigates the understanding of code smells by the participants. The results point different levels of understanding considering the list of smells addressed in this paper. However, such a difference did not influence their opinion about the importance of code smells and the respective quantitative results for software maintainability. According to them, the results guide stakeholders to seek solutions for common issues (code smells) in a given system, and it consolidated the existence of problems they already knew.

The second question, RQ2, aimed to identify the influencing factors on code smells. Considering the factors previously described in the respective cases, the participants also identified general aspects that can influence the emergence of any code smell. A consolidated list of factors is presented next:

- Short deadlines and time pressure to deliver software;
- Bad architectural decisions, that represent implementation decisions in a higher level of granularity (e.g. at a component level, adopted technology);
- Bad design decisions, that represent implementation decisions in a lower level of granularity (e.g. concentration of business rules into specific methods or classes);
- Lack of understanding of the business;
- Lack of experienced developers;
- Lack of refactoring effort after achieving a specific requirement;
- Low priority to software quality;

The last question (RQ3) focused on identifying how to mitigate the code smells and the respective influence factors. Most of the mitigation strategies concerned specific refactoring that we could not generalize. The participants also proposed the following strategies: additional time for refactoring in the development schedules, higher priority for software quality, better understanding of the adopted technologies and training for inexperienced developers.

6. Threats to Validity

We discuss four threats to validity in this study [Yin 2008]: construct validity, internal validity, external validity, and reliability. For construct validity, our cross-case design considers multiples sources of evidence in the data collection (quantitative data, interview and field notes) as a way of encouraging convergent lines of inquiry. Considering internal validity, our strategy consisted of using data analysis results to make possible inferences by using strategies such as coding-causal loop diagram on the evidence, textual explanation-building and discussion-validation. Despite the generalization of the findings is not possible, we applied the case study protocol in three different real-world scenarios to address external validity. Reliability was addressed by developing a detailed research protocol, in order to clarify all activities of this cross-case analysis.

7. Conclusions

This paper presented a qualitative and quantitative approach to investigate influencing factors of code smells based on a software developer point-of-view. Participants opinions point code smells and the quantitative information around them are important to improve software maintainability. Furthermore, we identified a set of factors that influence the emergence of code smells and possible strategies to mitigate them.

As future work, we intend to have a better understanding on mitigation strategies for code smells. In addition, despite of applying the research protocol in different scenarios, it needs to be applied in other different contexts to improve external validity, since we considered only systems implemented in the Java language.

References

Corbin, J. M. and Strauss, A. L. (2008). *Basics of qualitative research*. Sage Publ., 3. ed. edition.

Eisenhardt, K. M. (1989). Building Theories from Case Study Research. *The Academy of Management Review*, 14(4):532–550.

Fontana, F., Mariani, E., Morniroli, A., Sormani, R., and Tonello, A. (2011). An experience report on using code smells detection tools. pages 450–457.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Glaser, B. G. and Strauss, A. L. (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Transaction, 8th printing edition.

Katzmarski, B. and Koschke, R. (2012). Program complexity metrics and programmer opinions. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 17–26.

Khomh, F., Di Penta, M., and Gueheneuc, Y.-G. (2009). An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 75–84. IEEE Computer Society.

Lanza, M., Marinescu, R., and Ducasse, S. (2005). *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Macia, I., Garcia, A., Von Staa, A., Garcia, J., and Medvidovic, N. (2011). On the impact of aspect-oriented code smells on architecture modularity: An exploratory study. In *Software Components, Architectures and Reuse (SBCARS), 2011 Fifth Brazilian Symposium on*, pages 41–50.

Merriam, S. B. (2009). *Qualitative research: a guide to design and implementation*. Jossey-Bass higher and adult education series. Jossey-Bass.

Olbrich, S., Cruzes, D., Basili, V., and Zazworka, N. (2009). The evolution and impact of code smells: A case study of two open source systems. pages 390–400.

Sjoberg, D., Yamashita, A., Anda, B., Mockus, A., and Dyba, T. (2013). Quantifying the effect of code smells on maintenance effort. *Software Engineering, IEEE Transactions on*, 39(8):1144–1156.

Yamashita, A. and Counsell, S. (2013). Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software*, 86(10):2639–2653.

Yamashita, A. and Moonen, L. (2012). Do code smells reflect important maintainability aspects? In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 306–315. IEEE Computer Society.

Yin, R. K. (2008). *Case Study Research: Design and Methods (Applied Social Research Methods)*. Sage Publications, fourth edition. edition.

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