Synthesis of Code Anomalies: Revealing Design Problems in the Source Code

Willian N. Oizumi[1], **Alessandro F. Garcia**[1] **(Advisor)**

[1]Informatics Department – Pontifical Catholic University of Rio de Janeiro (PUC-Rio)

{woizumi,afgarcia}@inf.puc-rio.br

Abstract. *Design problems affect most software projects and make their maintenance expensive and impeditive. Thus, the identification of potential design problems in the source code – which is very often the only available and up-to-date artifact in a project – becomes essential in long-living software systems. This identification task is challenging as the reification of design problems in the source code tend to be scattered through several code elements. However, state-of-the-art techniques do not provide enough information to effectively help developers in this task. In this work, we address this challenge by proposing a new technique to support developers in revealing design problems. This technique synthesizes information about potential design problems, which are materialized in the implementation under the form of syntactic and semantic anomaly agglomerations. Our evaluation shows that the proposed synthesis technique helps to reveal more than 1200 design problems across 7 industry-strength systems, with a median precision of 71% and a median recall of 78%. The relevance of our work has been widely recognized by the software engineering community through 2 awards and 7 publications in international and national venues.*

1. Introduction

Design problems are caused by the violation of key design principles or rules [Oizumi et al. 2016]. Software systems suffer from design problems, introduced either during original development or during evolution. *Fat Interface* and *Unwanted Dependency* [Garcia et al. 2009] are examples of design problems. Software systems – like Linux [Schach et al. 2002] and Mozilla Firefox [Godfrey and Lee 2000] – have had to be fundamentally reengineered or have been discontinued when design problems were allowed to persist in the code and to be compounded by other design problems introduced later [Oizumi et al. 2016]. Therefore, even presenting different degrees of severity, design problems should be identified and removed in early versions of a program.

Design problems are introduced and allowed to remain in a system because their localization in the source code is difficult. As design documentation is often informal or nonexistent, code anomalies – popularly known as code smells [Fowler 1999] – are the mechanisms used to locate possible design problems in the source code. However, each code anomaly represents only a partial indicator of a design problem [Oizumi et al. 2015]. Examples of typical code anomalies are *Long Method* and *God Class*. Even though each code anomaly can provide some hint to developers, it alone might not suffice to indicate the presence of a design problem. In fact, isolated anomalies are often irrelevant to software design [Oizumi et al. 2014a]. Each design problem is rarely localized in a single anomalous code element; instead, it is scattered into different anomalous code elements

of the implementation [Oizumi et al. 2016, Oizumi et al. 2015, Oizumi et al. 2014a]. Therefore, developers often have to analyze multiple code anomalies in order to locate and understand a single design problem.

Unfortunately, there is very limited knowledge about how design problems manifest in the source code. This occur because the relation between design problems and their counterpart code anomalies is often complex. There is little to no understanding of which relationships between code anomalies are frequent indicators of design problems in complex software systems. There is a recent growing interest in conceptually characterizing interactions between code anomalies. However, the relation of code anomalies and design problems is rarely investigated. Empirical studies only address how individual occurrences of code anomalies emerge during software evolution or affect quality attributes. They do not analyze how individual anomalies and their relationships in the code might help developers to spot design problems. As a result, conventional techniques for anomaly detection are unable to effectively reveal design problems in the source code [Oizumi et al. 2015].

In this context, this work addresses the aforementioned gap in the literature, proposing and evaluating a technique for the **S**ynthesis of **C**ode **A**nomalies (SCA, for short) [Oizumi et al. 2014b]. In order to reveal design problems, SCA searches for coherent groups of inter-related code anomalies – the so called anomaly agglomerations. With the aim of assessing the effectiveness of SCA, we conducted two evaluations: (1) a multi-case study involving 7 systems with different sizes [Oizumi et al. 2016, Oizumi et al. 2015], and (2) a controlled experiment and interviews with several professional developers [Oizumi 2015]. Both evaluations confirmed that SCA is at least twice better than conventional techniques to reveal critical design problems in a software system. Moreover, we found that our algorithm for synthesizing semantic agglomerations helped to locate design problems with a precision higher than 80%.

2. Synthesis of Code Anomalies: An Overview of the Proposed Technique

Existing techniques (e.g. [Lanza and Marinescu 2006]) for detecting design problems in the source code are based on the assumption that single measures or smells help to locate design problems. Instead, SCA reveals design problems through the systematic search and summarization of information about code-anomaly agglomerations [Oizumi et al. 2014b]. We present below a summarized description of the main steps of SCA. Refer to Chapter 3 of [Oizumi 2015] for a detailed description.

Detection of Individual Code Anomalies. Using conventional algorithms [Lanza and Marinescu 2006], SCA analyzes the source code of the program aiming at detecting instances of code anomalies [Fowler 1999][Lanza and Marinescu 2006]. SCA covers the types of code anomalies documented in [Fowler 1999].

Search for Agglomerations. After the detection of code anomalies is completed, SCA explores information about code anomalies and relationships between code elements [Oizumi et al. 2015] to search for anomaly agglomerations. The search performed by SCA explores syntactic and semantic relationships to identify anomaly agglomerations in a program [Oizumi et al. 2014b]. Method calls and design concerns involving anomalous elements in the program are examples of syntactic and semantic relationships, respectively. A concrete example of agglomeration is provided in Section 3.

In this work, we proposed an evaluated 6 categories of agglomeration based on semantic and syntactic relationships: (1) concern-based, (2) cross-boundary, (3) intra-boundary, (4) hierarchical, (5) intra-method and (6) intra-class. The first category encompasses agglomerations formed through design concerns. Agglomerations composed by relationships that crosses the boundaries between design components fall in the second category. The third category contains agglomerations that occur in the internal structure of design components. Agglomerations based on hierarchical relationships (inheritance and interface implementation) fall in the fourth category. Finally, the fifth and sixth categories consider agglomerations occurring in common methods and common classes, respectively. Each category of agglomeration provides information that other categories may not provide. In other words, each of them presents a distinct perspective to analyze anomaly agglomerations [Oizumi et al. 2014a, Oizumi et al. 2016]. The identification of each category of relationship is realized through a search strategy. To better understand SCA search strategies, consider the concern-based category. The search strategy for this category consists of grouping into an agglomeration anomalous elements of different components, which are related to the same design concern (e.g. Concurrency).

Summarization of Relevant Information. To provide valuable information about the agglomerations found, SCA synthesizes relevant information about each agglomeration. This includes a textual description of the anomalies in each agglomeration, a list of code elements surrounding each agglomeration, and historical information about each agglomeration. The aforementioned information is intended to support developers in the process of analyzing, understanding and removing design problems revealed by agglomerations.

3. Evaluation of SCA

Our initial studies indicated that several isolated anomalies are often not related to design problems [Oizumi et al. 2014a, Oizumi et al. 2014b]. This finding motivated us to perform the evaluation of SCA based on two research questions:

RQ1. Is SCA an accurate technique to support the location of design problems?
RQ2. What is the most useful category of relationships to locate design problems?

The objective of RQ1 is to investigate whether SCA is better than a conventional technique to identify design problems. The conventional technique consist of state-of-the-art algorithms for code anomaly detection [Lanza and Marinescu 2006]. To answer this question, we performed two evaluations regarding the proportion of false positives and false negatives of each technique. RQ2 is aimed at investigating which of the proposed categories of relationships is the best to reveal design problems. These questions were answered based on the use of quantitative and qualitative methods. Procedures and results of our studies are described below.

Methodology. In order to answer research questions RQ1 and RQ2, we conducted two assessments: a multi-case study in the context of seven industry-strength systems, and a controlled experiment and interviews with eight professional developers. In the first assessment, we investigated which technique presented the highest precision and recall in the identification of design problems. As agglomerations involves more code elements than single anomalies, we also analyzed the proportion of elements in agglomerations that are related to design problems [Oizumi et al. 2015]. Finally, we compared the accuracy

of anomaly agglomeration categories to understand which is the best one to reveal design problems. To perform our analysis, original developers helped us to identify instances of 8 design problems in the 7 target systems. Details about each design problem are provided in Section 2.2 of [Oizumi 2015].

In the second assessment, we compared the use of SCA and of the conventional technique by professional developers through a controlled experiment. In this experiment, professionals were asked to identify design problems in two different systems using both techniques. This experiment helped us to analyze, based on the developers' perception, the benefits of SCA in actual tasks on the identification of design problems. A thorough description of both evaluations is presented in Chapters 4 and 5 of [Oizumi 2015].

SCA Outperformed the Conventional Technique. Regarding RQ1, SCA was at least twice better than the conventional technique to indicate the presence of design problems, presenting a median precision of 71% and a median recall of 78%. In four systems, SCA was five times more accurate than the conventional technique. Using SCA, developers could discard near by 4000 isolated code anomalies, which would be irrelevant to reveal design problems. Finally, we observed that developers make several mistakes as they are exposed to various false positives when using a conventional technique.

Example of Critical Design Problem Detected with SCA. Our findings in both studies also confirmed that SCA was accurate to reveal the most critical design problems in the analyzed systems. Consider the hierarchy of classes presented in Figure 1. This figure shows a snapshot of the *Versioner* class hierarchy in th OODT system [Mattmann et al. 2006]. The *Versioner* hierarchy is responsible for managing and storing versions of different *Product* types using alternative storage strategies. All classes in the *Versioner* hierarchy have to implement the *createDataStoreReferences* method. This method has two parameters: a *Product* instance and a *Metadata* instance. As there are no sub-classes for each type of *Product*, each *createDataStoreReferences* implementation has to decide if it is handling the correct *Product* type (e.g., the *MetadataBasedFileVersioner* must only process "flat" products). Consequently, the *Product* type handled by each *Versioner* implementation cannot be discovered from the *createDataStoreReferences* interface. Instead, this can only be discovered by analyzing details of each *createDataStoreReferences* implementation. Hence, the developer can conclude the *Versioner* implementations are affected by the Fat Interface design problem. This design problem occurs in interfaces that expose multiple functionalities through a general interface. This problem can only be identified through a careful analysis of the OODT source code.

Only through the use of SCA, developers were able to identify this critical design problem. All implementations of *Versioner* (*SingleFileBasicVersioner*, *BasicVersioner*, *DateTimeVersioner* and *MetadataBasedFileVersioner*) are affected by instances of the Feature Envy anomaly – a method that calls more methods of a single external class than of its own class. All these anomaly instances occur in classes that implement the same interface – which is the *Versioner* interface. This means the anomalous classes are inter-related through hierarchical relationships. SCA explores such relationships to detect this category of anomaly agglomeration. This agglomeration provides fundamental information to reveal the Fat Interface: anomalous counter-part elements of the design problem in the source code, and the relationships between the anomalous elements. We observed in the experiment that, analyzing this agglomeration, a developer did not waste

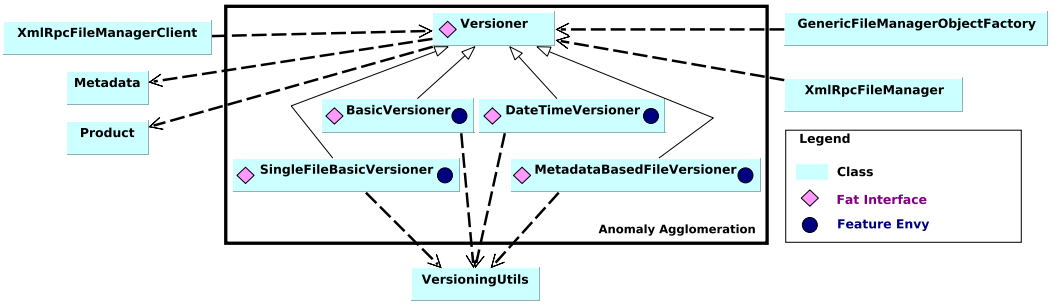

Figure 1. Example of agglomeration in the OODT system.

time in analyzing irrelevant non-agglomerated anomalies.

Other features of SCA were also responsible for achieving high accuracy. SCA provides information about the relationships between the agglomeration and its surrounding code elements. In the example of Figure 1, the code elements *Metadata*, *Product*, *VersioningUtils*, *XmlRpcFileManager*, *XmlRpcFileManagerClient* and *GenericFileManagerObjectFactory* are not anomalous. However, they are surrounding the agglomeration, as they are related to anomalous classes in the agglomeration. This information is important for a developer to recognize the extent and relevance of the design problem. Moreover, if a developer eventually removes the design problem, some surrounding elements would surely be affected.

Accuracy of Agglomeration Categories. Regarding RQ2, intra-component, cross-component and hierarchical categories cannot be considered very strong indicators of design problems. Their statistical significance was not high. Nevertheless, considering all the analyzed systems, approximately 50% instances of these categories of agglomerations were related to design problems, with a median recall of 55%. This accuracy is much higher than the accuracy of individual code anomalies. Design problems were often much more precisely indicated by semantic anomaly agglomerations. In general, the accuracy was approximately 80% when considering all the design problems and systems analyzed in our study. This result was also confirmed by our controlled experiment with professionals. Moreover, design problems identified with the help of such semantic anomaly agglomerations were considered the most complex and hard to spot. Considering our sample of systems, semantic agglomerations presented a median recall of 13%. Our analysis revealed that this low recall occurred due to inaccurate information provided by original architects, which can be improved by using automatically detected design concerns. When provided with more precise information about design concerns, semantic agglomerations revealed 39.58% more design problems [Oizumi et al. 2016].

4. Concluding Remarks

We designed and proposed SCA, a technique for the synthesis of code anomalies and support the location of design problems in the source code. The proposed technique was evaluated in the context of two empirical studies. Both studies provided evidence that SCA, in fact, significantly outperforms existing state-of-the-art techniques. We also created *Organic* [Oizumi and Garcia 2015], an open-source project for the practical use and evaluation of SCA. **Relevance.** The relevance of our idea and concrete solution

have been recognized by the software engineering community. We published our results in three international conferences [Oizumi et al. 2016] (**Qualis A1**), [Vidal et al. 2015], and [Vidal et al. 2016]; a symposium [Oizumi et al. 2014a]; an international journal [Oizumi et al. 2015]; and a workshop [Oizumi et al. 2014b, Albuquerque et al. 2014]. **Awards.** Our work was awarded twice at the Congresso Brasileiro de Software (CBSoft 2014): **best paper** of WMod [Oizumi et al. 2014b] and **third best paper** of SBES [Oizumi et al. 2014a], which is the most traditional Brazilian symposium on Software Engineering. As a result, we were invited to submit an extended version of our work to the Journal of Software Engineering Research and Development (JSERD) [Oizumi et al. 2015]. **Acknowledgements.** This research was sponsored by PUC-Rio, FAPERJ and CAPES. We thank Nenad Medvidovic, Arndt von Staa and Leonardo da Silva Sousa for their technical contributions.

References

Albuquerque, D., Garcia, A., Oliveira, R., and Oizumi, W. (2014). Deteccao interativa de anomalias de codigo: Um estudo experimental. In *WMod'14 at Cbsoft*.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Boston.

Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009). Identifying architectural bad smells. In *13th European Conf. on Software Maintenance and Reengineering*.

Godfrey, M. and Lee, E. (2000). Secrets from the monster: Extracting Mozilla's software architecture. In *CoSET-00*, pages 15–23.

Lanza, M. and Marinescu, R. (2006). *Object-Oriented Metrics in Practice*. Springer.

Mattmann, C. et al. (2006). A software architecture-based framework for highly distributed and data intensive scientific applications. In *Proceedings of the 28th ICSE*.

Oizumi, W. (2015). Synthesis of code anomalies: Revealing design problems in the source code. MSc dissertation. In *Pontifical Catholic University, Informatics Department*.

Oizumi, W. and Garcia, A. (2015). Organic. http://wnoizumi.github.io/organic/.

Oizumi, W., Garcia, A., Colanzi, T., Staa, A., and Ferreira, M. (2015). On the relationship of code-anomaly agglomerations and architectural problems. *JSERD, Springer*.

Oizumi, W., Garcia, A., et al. (2014a). When code-anomaly agglomerations represent architectural problems? An exploratory study. In *SBES'14; Maceio, Brazil*.

Oizumi, W., Garcia, A., et al. (2016). Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *ICSE'16; Austin, USA*.

Oizumi, W., Garcia, A., Sousa, L., Albuquerque, D., and Cedrim, D. (2014b). Towards the synthesis of architecturally-relevant code anomalies. In *WMod'14 at CBSoft*.

Schach, S. et al. (2002). Maintainability of the linux kernel. *Software, IEE Proceedings*.

Vidal, S., Guimaraes, E., Oizumi, W., et al. (2016). On the criteria for prioritizing code anomalies to identify architectural problems. In *SAC'16; Pisa, Italy*.

Vidal, S., Vazquez, H., Diaz-Pace, J. A., Marcos, C., Garcia, A., and Oizumi, W. (2015). Jspirit: a flexible tool for the analysis of code smells. In *34th SCCC*.