Graph-based visualization for code smell agglomerations: A controlled experiment

Anne Agbachi[1], Leonardo Sousa[1], Willian Oizumi[1], Roberto Oliveira[1], Alessandro Garcia[1], Anderson Oliveira[1], Baldoino Fonseca[2], Carlos Lucena[1]

[1]PUC-Rio, Rio de Janeiro, Brazil, [2]UFAL, Maceió , Brazil,
{oagbachi, lsousa, woizumi, rfelicio, afgarcia, aoliveira, lucena}@inf.puc-rio.br,
baldoino@ic.ufal.br

Abstract. *Recent studies indicate that each design problem is often related to inter-connected code smells in a program. This group of inter-connected smells is called a smell agglomeration. As developers need to analyze agglomerations to identify design problems, a visualization may help them in the analysis. In this context, we conducted a study to evaluate whether a graph-based agglomeration visualization can help developers to identify design problems. The results show that the visualization can indeed support design problem identification. Indeed, developers often used the visualization to have the first sign of the presence of a design problem. Also, they used the visualization to reason about the concentration of smells and, therefore, find a design problem.*

1. Introduction

Design problems are structures that indicate violations of key design principles or rules [Suryanarayana et al. 2014]. An example of design problem is *Fat Interface* [Martin 2006]. This problem occurs when an interface violates the *Interface Segregation Principle* [Martin 2006] as the interface offers a general entry point for several non-cohesive services, complicating the logic of its clients and maintenance tasks. Design problems might be so harmful that software systems have been discontinued or reengineered due to their prevalence [Schach et al. 2002]. Unfortunately, the identification of a design problem is not a trivial task [Trifu and Marinescu 2005], specially when the design documentation is unavailable or outdated [Kaminski 2007]. Given this scenario, developers may use code smells in their quest to identifying design problems.

A code smell is a microstructure in the program that represents a surface indication of a design problem [Fowler 1999]. Developers often perceive code smells as relevant to identify design problems [Palomba et al. 2014]. In fact, recent studies revealed that design problems are likely to be located in code elements that contain multiple code smells [Macia et al. 2012, Oizumi et al. 2016, Yamashita et al. 2015]. In particular, Oizumi et al. [Oizumi et al. 2016] proposed a strategy to group multiple code smells. In their strategy, they group code smells that are syntactically related in the source code structure, in which each group is called *code smell agglomeration*. According to their correlation analyses, agglomerations are often the locus of design problems [Oizumi et al. 2016].

As an agglomeration is composed of multiple smells, developers may benefit from visualization mechanisms during the analysis of each agglomeration. For instance, a visualization mechanism may help developers to identify the elements affected by the agglomerated smells, i.e., code smells that compose an agglomeration. Thus, developers can

focus on these elements to identify a design problem. Oizumi et al. [Oizumi et al. 2017] conducted another study in which they compared the use of agglomeration against the use of a flat list of code smells. In this study, they presented a first version of a visualization mechanism for the agglomerations. Unfortunately, their proposed visualization mechanism fell short in fully representing each agglomeration. For instance, the agglomerations were presented without taking into consideration the relations between the smelly code elements – i.e., code elements that are affected by code smells. In fact, important details about the agglomeration, as the relations between smells, were presented in separated tabs, without showing the full extension of the agglomeration in a single view. The result of that study showed that, even when developers used agglomerations, they failed in accurately identifying design problems. According to several developers, the lack of proper visualization of agglomerations was a core reason for their limited success.

As previous studies have fallen short in exploring visualization mechanisms to support the analysis of the agglomerations [Vidal et al. 2016, Oizumi et al. 2017], we conducted a study to investigate if a graph-based visualization of smell agglomerations can help developers to identify design problems. In our study, we asked to 10 developers (organized in pairs) to use a set of agglomerations to identify design problems in their systems. We also asked them to evaluate to what extent the visualization helped them in the identification task. Our results indicate that a graph-based visualization can provide benefits for developers during the identification of design problems. The developers mentioned that the graph-based visualization helped them to understand the dependencies among the elements affected by agglomerated smells. In fact, we noticed that developers often used the visualization of smell agglomerations to have first sign of the presence of a design problem. For instance, they used the density of interrelated smelly elements presented in the visualization to make assumptions about the existence (or not) of the design problem. Upon qualitative data analysis, we also noticed some features that a visualization of agglomerations should provide; for instance, it should promote a smooth navigation between the source code of agglomeration' smells and their visualization.

2. Our Proposed Visualization and Related Work

The identification of a design problem is not a trivial task. For instance, if a developer is analyzing an agglomeration to reveal a design problem, she has to keep in mind the elements that contain the agglomerated smells and how these smells together can indicate a design problem. In this case, a visualization mechanism is likely to help her in the analyses of an agglomeration. Given this assumption, we are evaluating in this paper a graph-based visualization [Herman et al. 2000] for smell agglomerations.

Background. Figure 1 presents an example of the graph-based visualization of smell agglomerations. This visualization uses nodes (rectangles) to represent code elements like classes and interfaces, and edges (arrows) to represent the dependency relation between smelly code elements. The visualization uses different arrows to represent the three types of relation among the code elements. A dotted arrow represents an interface implementation, while a straight arrow indicates an inheritance. The direction of the arrow indicates the direction of the relation. For instance, in Figure 1, *OcorrenciaService* inherits from *AbstractService*. A straight line represents a simple association between two classes. All the classes involved in the agglomeration are presented in the blue color. The red color indicates the class that the developer is analyzing at the moment (e.g. *OcorrenciaService*).

When the developer moves the mouse over a class, the visualization shows the smells affecting the class (e.g. *Long Parameter List* in *AbstractService* class).

This graph-based visualization can help developers during the analysis of an agglomeration. For instance, the visualization can help developers to identify all the elements affected by the agglomerated smells and help them to see the relation among these smelly elements. As an example, let us consider the agglomeration presented in Figure 1. The visualization for this agglomeration shows that most of the classes depend on the *AbstractService* class (node in the center), which is the source of the *Concern Overload* design problem [Garcia et al. 2009]. The identification of this type of design problem requires the identification of problematic structures that may be indicated by different types of code smells and by their relations. In the example of Figure 1, the agglomeration contain multiple code smells like *God Class, Dispersed Coupling, Intensive Coupling, Divergent Change and Feature Envy* [Fowler 1999]. These smells indicate symptoms like high coupling, high complexity and overload of responsibilities, which helps to diagnose the *Concern Overload* problem. In addition, the confirmation of this design problem depends on identifying the impact of *AbstractService* in other classes. By using a visualization to observe the dependencies among the smelly elements, the developer could observe that *AbstractService* provides services to multiple smelly classes. Therefore, the analysis of the visualization and the smells can help the developer to find the design problem.

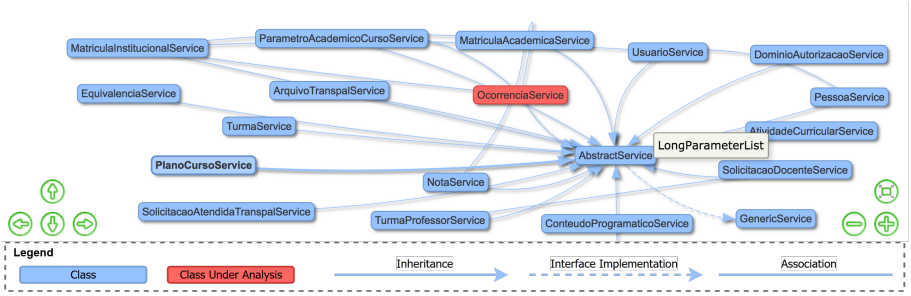

Figure 1. Graph-based visualization for agglomeration

Related Work. Previous studies have not focused on investigating if the visual representation of agglomerations helps to find design problems. In fact, Oizumi et al. [Oizumi et al. 2017] conducted a study close to ours. However, their focus was not evaluating how a visualization mechanism can support the detection of design problems. Even though, they noticed during the data analysis that the proposed visualization mechanism was a core reason why some developers had difficult to identify design problems. Due to this result, we are investigating if a graph-based visualization is suitable to support the identification of design problems. Vidal et al. [Vidal et al. 2016] is other study that focused on analyzing agglomerations, but not in the merit of the underlying visualization mechanism. They presented JSpIRIT, a tool for detecting and ranking smells and agglomerations in Java programs [Vidal et al. 2016]. The JSpIRIT visualization also did not represent the relations between the code smells of an agglomeration. Marinescu et al. [Marinescu et al. 2010] presented InCODE, a tool that provides detection and visualization of code smells that can indicate design problems. Unfortunately, the visualization of this tool does not explicitly represent smell agglomerations. In addition, it excludes dependencies between packages, which are useful to understand the underlying structure

of the system. Both information can bring out problems in the design produced by a violation in the system architecture. These limitations of existing studies motivated us to propose a visualization for agglomerations based on the notion of graphs. A graph-based visualization may be useful for the developers because it can represent the dependencies among classes affected by the agglomerated smells. As illustrated above, this information can help developers to analyze the agglomerations to reveal design problems.

3. Study Planning

3.1. Goal and Research Questions

We intend to assess a graph-based visualization for the purpose of evaluating its effectiveness in supporting developers on the identification of design problems. The context comprises professional developers reviewing source code of software projects developed by themselves. Thus, we performed a controlled experiment aimed at answering the following research questions:

RQ1: Does graph-based visualization of smell agglomerations support identification of design problems?

RQ2: How to improve visualization of smell agglomerations?

These questions are intended to reveal: (i) whether and how graph-based smell visualization helps to locate a design problem (RQ1), and (ii) what visualization features should be added or removed to better represent smell agglomerations (RQ2). To address these research questions, we performed a qualitative analysis. First, we analyzed the developers' opinion about the relevance of the visualization in supporting the identification of design problems. Then, we observed how developers used the graph-based visualization, which allowed us to identify features that can be used to improve the visualization.

To answer both research questions, we selected two Brazilian software companies (CO1 and CO2). The company selection was based on their different features as the adoption of code reviews, the number of developers, their application domains, and development process. After selecting the companies, we selected 10 developers, 6 from CO1 and 4 from CO2. To obtain a profile of each developer, we applied a questionnaire regarding their knowledge of software development concepts, Java programming language, code smell, and design problems. More information about these profiles and the companies are available on our website [Agbachi 2017].

3.2. Experimental Procedure and Data Collection

The experiment encompassed two main activities: a training session and a session comprising the tasks of design problem identification. The training session consisted of explaining and discussing basic concepts (e.g. code smell, design problem, and graph-based visualization). In the second activity, the developers were provided with the list of smell agglomerations, the graph-based visualization for each agglomeration, and the description of the code smells. To encourage developers to discuss the design problems, we asked them to identify the design problems in pairs. The discussions of this activity gave us more input for our qualitative data analysis. During this activity, they also had to classify the visualization for each agglomeration into four categories of relevance: *irrelevant*, *slightly relevant*, *relevant* and *highly relevant*. After the task, the developers answered a questionnaire about the usefulness of the visualization of agglomerations.

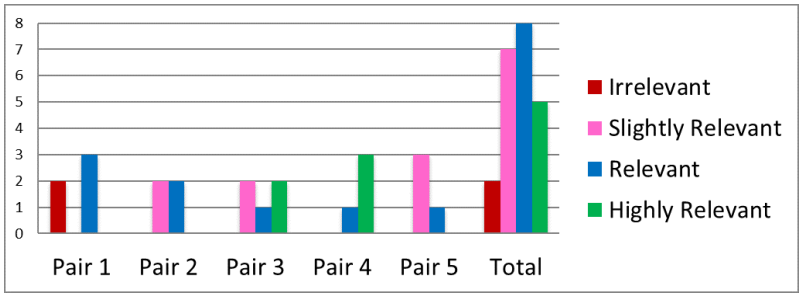

Figure 2. Relevance classification for the graph-based visualization

We collected the classifications of relevance and the questionnaire answers through an online form. In addition, we used Camtasia to record audio and video of the computer screen while developers used the Eclipse IDE to import and analyze the source code. We analyzed the collected data using the coding process from the Grounded Theory (GT) [Strauss and Corbin 1998]. We focused on the open coding (1st phase) and the axial coding (2nd phase) from GT to answer our research questions. We transcribed the data of logs, audios, and video. When analyzing the qualitative data, we created codes related to the speeches of the developers (1st phase of GT). To eliminate discrepancies, three researchers validated the transcription. After coding all the transcriptions, we related the codes to each other (2nd phase of GT). The analysis of the codes allowed us to answer our research questions. Full details about the experimental procedures, data analysis, and data collection are available in our complementary material [Agbachi 2017].

4. Results and Analysis

In this section, we present the results used to answer our research questions.

4.1. Relevance of a Graph-based Visualization

The graph-based visualization is relevant for identifying design problems. Regarding research question RQ1, Figure 2 presents the number of times each pair classified the agglomeration instances according to their relevance. It also shows the total number of classifications for each category. In general, our results suggest our results indicate that developers consider the graph-based visualization relevant (36,4%) or highly relevant (22,7%) in most of the cases (59,1%) – the visualization was classified as irrelevant in only 2 cases. On the other hand, in 40,9% of the cases, developers classified the graph-based visualization as slightly relevant (31,8%) or irrelevant (9,1%). Only one pair (Pair 5) had a negative result with respect to to the visualization's relevance. This pair was the only one where the visualization was considered by their majority to be Slightly Relevant. Surprisingly, they only found a design problem when they classified the visualization as relevant. We conducted a qualitative analysis to better understand these results.

Understanding smell dependencies to find a design problem. We noticed that the most useful characteristic in a graph-based visualization is the intrinsic capability to show the dependencies between smelly classes of an agglomeration. Developers mentioned that visualizing all the dependencies as a graph allowed them to have a first sign of the presence of a design problem. That is, if an agglomeration had several smelly classes and they are strongly connected, the graph size in the visualization would be large. Thus,

they used the size and connectivity of the graph as an initial indication of a design problem. As an example, when the Pair 3 was analyzing the agglomeration of Figure 1 (Section 2), they mentioned that they remember nothing about the smelly elements. Thus, not only they were unaware of the relation between the classes, but more importantly they did not know about the presence of a design problem. This was until they opened the graph-based visualization, where they were alarmed by the size of the agglomeration and the density of its edges. Based on these findings, they were able to determine which classes were dependent upon one another. With the newly found knowledge of the dependencies, the developers were able to determine which classes would be more helpful in identifying the design problem indicated by the agglomeration.

The visualization can only be as good as the agglomeration. The internal precision of an agglomeration was another factor that influenced the claimed relevance for the graph-based visualization. In this context, the internal precision of an agglomeration is measured based on the number of agglomerated code smells that are actual symptoms of a design problem. In a few cases, the developers did not agree that most code smells within an agglomeration were, in fact, symptoms of design problem. Thus, they classified the graph-based visualization as irrelevant and justified the classification by claiming that the agglomeration was irrelevant (or slightly relevant). This happened in the two times that the Pair 1 classified the visualization as irrelevant. In other few cases, the developers classified the graph-based visualization as slight relevant or only relevant because they already knew about the code smells and the dependencies displayed by the visualization. Thus, they justified the low relevance of the visualization by the fact that the visualization did not present any new information about the agglomeration that they already knew.

4.2. Improving the visualization for code smell agglomerations

Developers need explicit and easy access to information. Regarding research question RQ2, we noticed some features that a visualization mechanism should provide. The most important one is that relevant information should be explicit and easy to locate. In our implementation, we have two perspectives: one that shows the graph-based visualization and other that shows the details of the agglomeration. The name of each agglomerated smell is displayed when the developer hovers over a node, as illustrated in Figure 1. However, developers have to leave the visualization and go to the other perspective to find detailed information about each code smell. Unfortunately, this behavior did not allow the developers to use the visualization and to explore the smells at the same time. In addition, we noticed that in the beginning of the task, some developers missed useful information. For example, they only realized the importance of the dependencies when they analyzed the second agglomeration. As the description of each code smell is not alongside with the visualization, they missed some code smells due to the constant changing between perspectives. Developers reported that this is a cumbersome procedure, which made of them, at some point, give up of using the visualization to guide their analysis. We noticed that for not using the visualization, developers missed some code smells that could be useful to identify a design problem.

Developers need navigating through the visualization and the source code. Besides the difficulty in accessing information, the navigation from the visualization to the source code was another issue. As the graph-based visualization was not integrated with IDE editors, the developers had to manually open each class. Sometimes, we ob-

served that developers were expecting to click over a node in the graph, and then to be automatically sent to the location in the source code that contains the agglomeration's smells. In addition, we noticed that they also had difficulty returning from the source code to the visualization. As the navigation was not supported, the developers often spent considerable time to analyze the agglomeration's symptoms in the source. This kind of analysis should be facilitated because it is critical to confirm the symptoms provided by the visualization. Thus, by having a two-way navigation between visualization and source code, developers may be able to effectively perform the analysis of agglomerations to identify design problems.

Graph manipulation is helpful for the analysis of agglomerations. Our graph-based visualization allows the manipulation of nodes. We noticed that this feature helped some developers. They moved the nodes of the graph to explore the agglomeration, and also to separate the nodes (classes) that actually contribute to a design problem from those that do not contribute. This behavior indicates that the visualization should allow them to temporally remove certain nodes that do not contribute to the identification of design problems. This kind of manipulation could be used, for instance, as input to improve the algorithm for agglomeration detection. That is, we could analyze the characteristics of the excluded classes to improve the internal precision of agglomerations.

5. Threats to Validity

The first threat to the validity of this study is the generalization of our results (Section 4). Our study involved a sample of 10 developers, divided in 5 pairs, which may not be enough to achieve conclusive results (Section 4). Thus, we cannot generalize our conclusions. Another threat to the validity is the limited diversity of contexts involved in the study. However, we argue that the selected companies represent typical software development organizations as provided in our complementary material [Agbachi 2017]. The third threat to the validity of this study is the possibility that the first author might have introduced bias in the data collection process. To mitigate this threat, the analysis process of was performed alongside other two researchers. Finally, there is a threat concerning the inherent difficulty in the experimental tasks. To minimize this threat, we trained all participants to resolve any gaps in knowledge or conflicts about the experiment.

6. Concluding Remarks

In this paper, we conducted a controlled experiment with 10 professional developers. Our goal was to investigate whether a graph-based agglomeration visualization can support developers in the identification of design problems. Our results (Section 4) indicate that developers benefit from using the graph-based visualization to identify design problems. The size of the graph and the visualization of dependencies between smelly elements allow developers to have an comprehensive overview of the problem. This encourages the use of visualization to support design problem identification. By analyzing how developers used the visualization, we noticed that the developers often analyzed the relations among the agglomerated code smells as a first sign of a design problem. In some cases the developers used the agglomeration size to reason about the presence of a design problem. We also plan to implement some features that a visualization should provide, such as the capability to navigating through the visualization and the source code.

References

Agbachi, A. (2017). Online companion. `http://wnoizumi.github.io/VEM2017`.

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 *CSMR09; Kaiserslautern, Germany*. IEEE.

Herman, I., Melancon, G., and Marshall, M. S. (2000). Graph visualization and navigation in information visualization: A survey. *IEEE Trans. Visual Comput. Graphics*, 6(1):24–43.

Kaminski, P. (2007). Reforming software design documentation. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 277–280.

Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., and von Staa, A. (2012). Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems. In *AOSD '12*, pages 167–178, USA. ACM.

Marinescu, R., Ganea, G., and Verebi, I. (2010). Incode: Continuous quality assessment and improvement. In *CSMR*, pages 274–275.

Martin, R. C. (2006). *Agile Principles, Patterns, and Practices in C#*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Oizumi, W., Garcia, A., Sousa, L., Cafeo, B., and Zhao, Y. (2016). Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *The 38th International Conference on Software Engineering; USA*.

Oizumi, W., Sousa, L., Garcia, A., Oliveira, R., Agbachi, O., Oliveira, A., and Lucena, C. (2017). Revealing design problems in stinky code: A mixed-method study. In *SBCARS17 (Submitted)*.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., and Lucia, A. D. (2014). Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 101–110.

Schach, S., Jin, B., Wright, D., Heller, G., and Offutt, A. (2002). Maintainability of the linux kernel. *Software, IEE Proceedings* -, 149(1):18–23.

Strauss, A. and Corbin, J. (1998). *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications.

Suryanarayana, G., Samarthyam, G., and Sharmar, T. (2014). *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann.

Trifu, A. and Marinescu, R. (2005). Diagnosing design problems in object oriented systems. In *WCRE'05*, page 10 pp.

Vidal, S. A., Marcos, C., and Díaz-Pace, J. A. (2016). An approach to prioritize code smells for refactoring. *Automated Software Engg.*, 23(3).

Yamashita, A., Zanoni, M., Fontana, F. A., and Walter, B. (2015). Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *ICSME*.