Pontifícia Universidade Católica
do Rio de Janeiro

Olouyèmi Ilahko Anne Bénédicte Agbachi

Identifying Design Problems with a Visualization Approach of Smell Agglomerations

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-graduação em Informática, of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
April 2018

Pontifícia Universidade Católica
do Rio de Janeiro

Olouyèmi Ilahko Anne Bénédicte Agbachi

Identifying Design Problems with a Visualization Approach of Smell Agglomerations

Dissertation presented to the Programa de Pós-graduação em Informática, of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the undersigned Examination Committee.

Prof. Alessandro Fabricio Garcia
Advisor
Departamento de Informática – PUC-Rio

Prof. Simone Diniz Junqueira Barbosa
Departamento de Informática – PUC-Rio

Prof. Marcos Kalinowski
Departamento de Informática – PUC-Rio

Prof. Márcio da Silveira Carvalho
Vice Dean of Graduate Studies
Centro Técnico Científico – PUC-Rio

Rio de Janeiro, April 13th, 2018

Olouyèmi Ilahko Anne Bénédicte Agbachi

Anne Bénédicte is a bachelor in Computer Engineering at Matanzas University "Camilo Cienfuegos" of Cuba (2015). She has worked on research projects in software engineering and information systems. She has received the second best paper award from the 11th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS'17). Anne Bénédicte is a research scholar in software engineering for Opus Research Group at PUC-Rio.

Bibliographic data

Agbachi, Anne Bénédicte

Identifying Design Problems with a Visualization Approach of Smell Agglomerations / Olouyèmi Ilahko Anne Bénédicte Agbachi; advisor: Alessandro Fabricio Garcia. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2018.

v., 100 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Abordagem de visualização. 3. Problema de design. 4. Anomalia de código-fonte. 5. Experimento. I. Garcia, Alessandro Fabricio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my beloved parents and brothers.

Acknowledgments

Firstly, I would like to thank my Lord. I do not know who I would have been without His love even before my birth.

I thank all my family, especially my dear parents and my beloved brothers, for their support and advice to follow my dream.

I would like to thank my advisor. Prof. Alessandro Garcia, who gave me the opportunity to work with him. I welcome his unconditional support, as the extraordinary and pertinent way as he accompanied the realization of this work. His constructive criticisms, discussions, and reflections were fundamental throughout my Master's. I thank him for his availability, even in holiday period, patience, and advice. I am always grateful for his great contribution to my growth as a researcher.

I thank Eduardo Fernandes. I do not have words to describe all your support in the realization of this work. I have not seen a person as requested as you but who is always willing to help everybody. Thanks for your availability and fellowship. Having worked with you has made me more confident; you are a person with good energy and optimistic.

I also thank all the members of the Opus Research Group for their support. In particular, I would like to thank Alexander Chávez, Ana Carla Bibiano, Anderson Oliveira, Anderson Uchôa, Diego Cedrim, Isabella Ferreira, Leonardo Sousa, Rafael de Mello, Roberto Oliveira, and Willian Oizumi, who have given me their unconditional help during my research since the beginning of my Master's.

I thank all my friends in Brazil, as my Cuban friends; all those even afar have always given me the support I needed to finish this Master's degree.

I thank the members of this Master's dissertation defense, especially Prof.ª Simone Barbosa Junqueira Diniz for providing me feedback since I started my Master.

I also thank the Informatics Department, and the group of professors, who always supported me throughout the course.

Finally, I thank the National Council for Scientific and Technological Development (CNPq) and the Coordination for the Improvement of Higher Education Personnel (CAPES). I appreciate every opportunity that I had, including the financial support. Thank you very much.

Abstract

Agbachi, Anne Bénédicte; Garcia, Alessandro Fabricio (Advisor). **Identifying Design Problems with a Visualization Approach of Smell Agglomerations**. Rio de Janeiro, 2018. 100p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Design problems are characterized by violations of design principles affecting a software system. Because they often hinder the software maintenance, developers should identify and eliminate design problems whenever possible. Nevertheless, identifying design problems is far from trivial. Due to outdated and scarce design documentation, developers not rarely have to analyze the source code for identifying these problems. Past studies suggest that code smells are useful hints of design problems. However, recent studies show that a single code smell might not suffice to reveal a design problem. That is, around 80% of design problems are realized by multiple code smells, which interrelate in the so-called smell agglomerations. Thus, developers can explore each smell agglomeration to identify a design problem in the source code. However, certain smell agglomerations are formed by several code smells, which makes it hard reasoning about the existence of a design problem. Visualization approaches have been proposed to represent smell agglomerations and guide developers in identifying design problems. However, those approaches provide a very limited support to the identification of specific design problems, especially the ones affecting multiple design elements. This dissertation aims to address this limitation by proposing a novel approach for the visualization of smell agglomerations. We rely on evidence collected from multiple empirical studies to design our approach. We evaluate our approach with developers from both academy and industry. Our results suggest that various developers could use our visualization approach to accurately identify design problems, in particular those affecting multiple program elements. Our results also point out to different ways for improving our visualization approach based on the developers' perceptions.

Keywords

Visualization approach; Design problem; Code smell; Experiment.

Resumo

Agbachi, Anne Bénédicte; Garcia, Alessandro Fabricio. **Identificando Problemas de Design através de uma Abordagem de Visualização para Aglomerações de Anomalias de Código**. Rio de Janeiro, 2018. 100p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Problemas de *design* decorrem de violações de princípios de *design* em um sistema de *software*. Tais problemas podem prejudicar a manutenção de sistemas e, logo, devem ser identificados e eliminados sempre que possível. Porém, identificar problemas de *design* não é trivial. Isso pois a documentação de *design* desses sistemas é em geral obsoleta ou inexistente. Assim, o desenvolvedor de um sistema tende a analisar o código-fonte em busca de problemas de *design*. Estudos sugerem anomalias de código-fonte como indicadores úteis desses problemas. Porém, outros estudos recentes mostram que uma única anomalia não é indicador suficiente. De fato, em torno de 80% dos problemas de *design* estão associadas com múltiplas anomalias. Estas inter-relacionam-se na forma de aglomerações de anomalias. Embora as aglomerações de anomalias possam ajudar o desenvolvedor a identificar problemas de *design*, certas aglomerações contêm muitas anomalias. Isso então dificulta o raciocínio sobre a existência de um problema de *design*. Além disso, mesmo as propostas mais recentes de abordagens para a visualização de aglomerações de anomalias provêm suporte bastante limitado à identificação de problemas de *design*. Essa limitação é evidente quando um problema de *design* afeta múltiplos elementos na implementação de um sistema. Esta dissertação objetiva tratar essa limitação ao propor uma abordagem inovadora para a visualização de aglomerações de anomalias. Tal abordagem baseia-se em evidências coletadas a partir de vários experimentos propostos e conduzidos por nós. Contamos com a participação de desenvolvedores da academia e da indústria em cada experimento. Nossos resultados de estudo sugerem que vários desenvolvedores podem utilizar nossa abordagem de visualização para identificar de forma precisa problemas de *design*, especialmente aqueles que afetam múltiplos elementos de programa. Nossos resultados também apontam melhorias necessárias à abordagem com base na percepção dos desenvolvedores.

Palavras-chave

Abordagem de visualização; Problema de design; Anomalia de código-fonte; Experimento.

Table of contents

List of figures

List of tables

You never fail until you stop trying.

Albert Einstein, *(1879-1955).*

1
Introduction

Design problems are characterized by violations of one or more key design principles affecting a software system (1). These design principles encompass the best practices recommended by a specific programming paradigm. We did split into many, one per responsibility, so it would not violate such principle.

Each type of design problem is characterized by a different set of violated design principles (1). We describe two of the main principles of the object-oriented programming as follows. The Single Responsibility principle (2) states that each design element should realize a single responsibility of the software system. Thus, a design element that realizes two or more non-cohesive responsibilities, e.g., data persistence and user access, violate that principle. The Interface Segregation principle (2) states that each interface should address a cohesive set of functionalities. Thus, one interface that provides multiple non-cohesive functionalities should types affecting software systems are Concern Overload (3) and Scattered Concern (2, 4), which we explain as follows. Concern Overload occurs when a design element realizes multiple non-cohesive responsibilities of the software system. In other words, a design element is affected by Concern Overload whenever it violates the Single Responsibility principle. As a consequence, this design problem tends to hinder the maintenance tasks applied to the affected design element. That is because the variety of responsibilities realized by the same element makes it difficult for developers to understanding and change the affected design element.

Design problems often hinder the software maintenance (1). Thus, developers should identify and correct design problems whenever possible. Otherwise, it could lead to either the discontinuation or the re-engineering of the affected system (5). However, identifying a design problem is far from trivial (6). That is because developers eventually have to analyze the source code due to outdated and scarce design documentation (7). Such analysis has often been driven by the identification of code smells (8), i.e., anomalous code structures that might indicate design problems. There is empirical evidence that developers actually perceive code smells as useful indicators of design problems (9). In addition, studies (3, 10, 11) reveal that design problems are likely to affect those design elements also affected by code smells.

Recent studies (10, 12) show that a single code smell might not suffice to identify a design problem. They suggest that design problems are often realized by multiple code smells that interrelate. These interrelations form the so-called smell agglomerations, which could boost the identification of design problems. A recent study (10) observes that smell agglomerations and design problems are closely related in a system. This study observes that 80% of the smell agglomerations are located in elements affected by design problems. Several research challenges have emerged from this observation. A particular challenge regards the fact that a single smell agglomeration might contain several code smells (and their interrelations). Thus, it might be very hard for developers to reason about each smell agglomeration while identifying design problems.

Various factors contribute to the complexity of reasoning about a smell agglomeration towards the identification of code smells. For instance, each single smell agglomeration can crosscut the structures of several program elements, such as methods, classes, and packages (12, 10). By crosscutting multiple program elements, it becomes hard for developers to understand to what extent the code smell interrelations contribute to the realization of a design problem. As a consequence, developers might find difficult to: obtain a general view of the smell agglomeration; navigate through each affected design element in order to understand the code smell relations, which might be many; and decide whether there is a design problem affecting the program or not.

Software visualization approaches (13, 14, 15, 16, 17) have been proposed for summarizing source code information with the aim of assisting developers in identifying potential issues in the source code. For instance, CodeCity (13) aims to support large-scale code analysis via software metrics. It visually represents all code elements as city buildings; each property of the building represents certain metric values of a program. CodeCity could help identify code elements with anomalous metric values that require some restructuring. Class Blueprint (14) is an approach for visualizing the internal structure of classes. It visually represents the interrelations between attributes and methods of a single class. This approach aims at guiding the cohesion assessment of a class.

Both CodeCity and Class Blueprint are visualization approaches that somehow provide either a too abstract source code representation (e.g., CodeCity that represents all program elements together in a general abstraction level) or a fine-grained representation (e.g., Class Blueprint shows internal elements and interrelations of a class). Thus, they seem not to be suitable for the identification of design problems. That is because identifying design problems often requires: reasoning about the design elements of a program; and

reasoning about the implementation elements of the program. As a generalization, we observe that conventional approaches for software visualization are not tailored to explicitly represent smell agglomerations. Nevertheless, software visualization emerges as a promising mechanism for summarizing all information about the smell agglomeration in order to help developers identify design problems.

As aforementioned, a single smell agglomeration might have several code smells, whose interrelations might be difficult to assess without an information summary. In addition to the information that characterizes code smell agglomeration, the identification of design problems requires additional information about the design of a system. This is because a software design has two main elements: structural elements (components, classes, etc.), which represent how the system is structured or decomposed; and concerns that represent the main considerations taken until the design stage. Each concern might be embodied by one or more structural elements. Thus, unifying all information in a single visualization is a differential with respect to previous studies, which provide very limited information about the smell agglomerations.

1.1 Motivation

The literature (3, 10) states that most design problems are somehow associated with various code smell types. Thus, each code smell might be useful a partial hint of the design principle violations affecting the source code. For example, a class affected by God Class (8, 18) tends to be very large and implement too many non-cohesive functionalities. On the other hand, a method of this class that is affected by Feature Envy (8, 18) has at least one call for methods from other classes. By observing the occurrence of both God Class and Feature Envy in a single class, developers might obtain a wider view of the violations affecting the class than it could be obtained by assessing each code smell in isolation (10, 12). Thus, agglomerations are powerful mechanisms to reveal design problem that hinder the maintainability of a software system.

In the following, we discuss how a visualization approach could help the identification of design problems through smell agglomerations. Figure 1.1 presents a partial design view of the Workflow Manager module from Apache OODT. OODT is a framework for managing large-scale data. Workflow Manager describes, executes, and monitors data processing workflows. The figure presents two components (Engine and Instrepo) with their respective classes. We represent components, classes, and interfaces similarly to the Unified Modeling Language (UML) (19). We represent the dependencies

between the program elements by arrows, code smells affecting each class or interface through circles, and the concerns implemented by a class or an interface through lozenges. A concern is a conceptual unit that represents a relevant consideration made when producing the software, which can impact on the software design (20). According to this definition, concerns may vary from software functionalities to design patterns and non-functional properties.

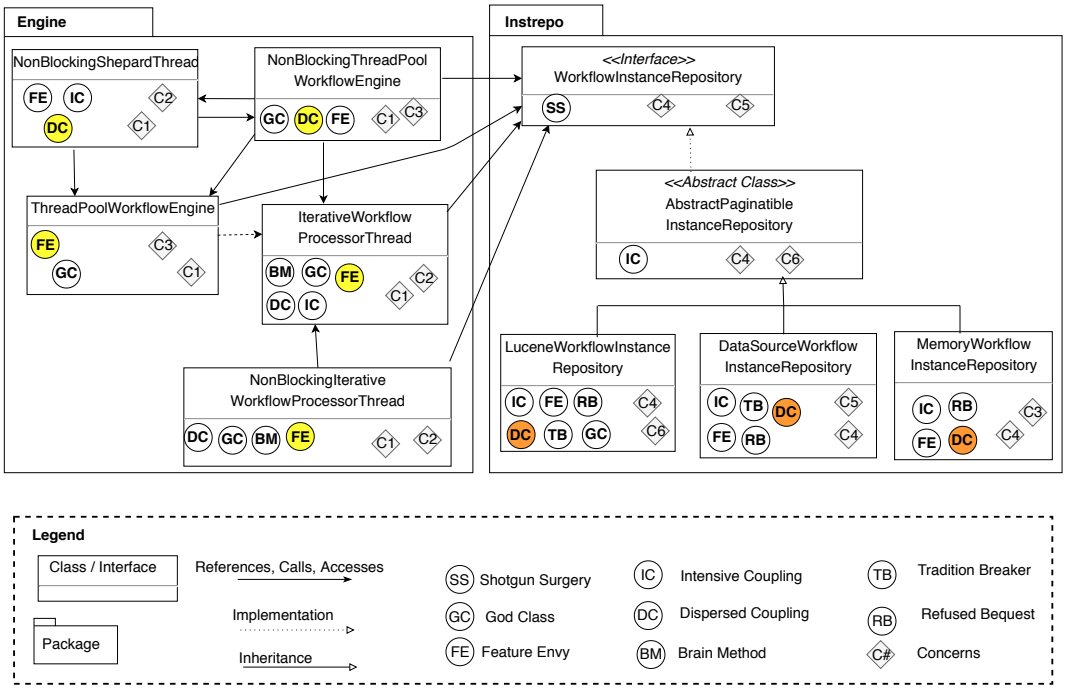

Figure 1.1: Partial Design View of the Workflow Manager Module

From Figure 1.1, we observe that at least one code smell affects each class or interface located in both components. Several code smells are spread over the classes, either within a single component or in different components. For instance, Shotgun Surgery (8) affects the WorkflowInstanceRepository interface. It implies that every change applied to this interface might propagate to several other classes (8). Moreover, multiple classes implement a common concern. For instance, all classes from the Engine component implement the C1 concern. In addition, the implementation of concerns is very difficult to capture through manual code analysis (21). It is very difficult because the developer needs to look line by line the code snippets that implement each concern. After that, the developer identifies what concerns are implemented in each method, in each class. The visualization of concerns by design element facilitates the analysis of code to identify design problem. All the aforementioned observations about Workflow Manager could be hard for developers to obtain from source code analysis. The difficulty stems from both the complex nature of code smells (9, 22) and the diffused nature of concerns (23, 24), which

we glimpse from the figure. In this scenario, a developer, who is in charge of identifying design problems, needs to analyze most of these code smells to identify the Scattered Concern design problem. This design problem happens whenever multiple code elements implement a concern that should have been implemented by only a few elements. We rely on the example of Figure 1.1 to illustrate how two different smell agglomerations help reveal design problems affecting Workflow Manager.

Agglomeration 1 revealing Scattered Concern and Concern Overload. In the Engine component, circles colored in yellow are part of a concern-driven agglomeration. In other words, these smells are occurring due to the fact that the Engine component is implementing various concerns (C1, C2, and C3). The component elements contain code smells Feature Envy (8, 25) and Dispersed Coupling (8, 25) that are related to the implementation of concerns. These are indicators of a Concern Overload design problem. In addition to the design problem Concern Overload, the component also suffers from the Scattered Concern design problem because concerns C2 and C3 are spread throughout several classes of the component, and also C3 is in the Instrepo component.

Agglomeration 2 revealing Scattered Concern and Concern Overload. The Instrepo component implements tree concerns: the main concern C4 and the additional concerns C5 and C6. Instrepo has several code smells affecting its class and interface. By analyzing the abstract class AbstractPaginatibleInstanceRepository we can see that it could be a problematic element because all classes that inherit have several code smells associated with the implementation of more than one concern. Circles in orange represent the code smells that form an agglomeration in the inheritance. This is because the classes that inherit from the abstract class have in common the code smell Dispersed Coupling (8, 25). All this information that the visualization provided are indicators of Concern Overload. Moreover, the agglomeration also reveals Scattered Concern since the additional concerns (C5, C6) were spread in 4 classes of the component and the Dispersed Coupling code smell affecting the classes is related to the implementation of more than one concern.

In addition to the excessive number of code smells affecting the system, developers still have to decide which code smells can help them to identify a design problem. At this point, developers can use the code smell agglomeration to focus on those code smells that may be related to a design problem. Although the agglomeration has the potential to help developers to focus on the code smells potentially related to a design problem, they still have to analyze several

code smells. They also need to understand how these code smells are related to each other in order to understand how they embody together the design problem. In other words, these multiple tasks would quickly turn into a very complex, cumbersome process. Developers would feel discouraged to perform all these tasks without a proper visualization of the agglomeration.

In summary, we hypothesize that visualization for code smell agglomeration could help developers to (i) grasp how the design problem is spread in the system, and also (ii) understand better how the code smells are related to each other in the source code. Such knowledge could alleviate the process of analyzing code smells to identify design problems.

1.2 Visualizing Smell Agglomerations

Previous studies (15, 26, 27, 28) propose approaches for visualizing code smells. Due to the recent introduction of smell agglomerations in the literature (10), these studies support the visualization of single code smells only. More recently, other studies (29, 30) provide the first attempts to support the visualization of smell agglomerations. Vidal et al. (30) proposed a preliminary visualization for code smell agglomerations (Figures 2.7 and 2.8). The authors had proposed a tool for detection of code smells, which also included a feature to represent code smell agglomerations. However, the tool presents very limited and abstract representation of the agglomerations. For instance, the visualization does not show the relationships between the code smells. Also, it does not provide any mechanisms that allow the user to navigate from the design level representation of the agglomerations to the coding level representation of each smell. Moreover, they share similar limitations as the visualization proposed by Oizumi and colleagues (29).

Oizumi et al. (29) investigated whether developers could accurately identify design problems through smell agglomerations. They compared the use of smell agglomerations against the use of a flat list of code smells. During the experiment, they provided a mechanism for supporting the visualization of smell agglomerations. Their study results indicated that smell agglomerations potentially help developers to identify more design problems when compared with the flat list; agglomerations provided 64% of accuracy versus 38.24% for the flat smell list, which showed the potential of agglomeration to help reveal a design problem. However, the support provided by smell agglomerations was far from sufficient due to low recall rates. Thus, the authors also conducted a qualitative study to investigate the reasons behind the limitations of smell agglomerations. They have found that the lack of a proper visual representation

of smell agglomerations was the main reason for these limitations.

In fact, the visualization proposed by Oizumi et al. (29) had some drawbacks, as found by the authors later. First, their approach does not unify the information about agglomerations and information needed for design problem identification. Second, the textual information provided for each smell agglomeration uses an excessively complex terminology. Many terms employed by the visualization were hard to be understood by developers, such as the names of the different types of agglomeration. All these disadvantages were not only related to the underlying terminology but were mainly caused by the poor visual representation. For instance, important details about the agglomeration, as the relationships between smells, were presented in separated views, without showing the full extension of the agglomeration in a single view. This led participants to analyze each code smell individually rather than together.

Overall, the study performed by Oizumi et al. (29) suggests that developers often start reasoning about a smell agglomeration from a general to a specific viewpoint. It has helped us understand how a visualization approach of smell agglomerations should support developers in identifying design problems.

1.3 Proposing a Graph-based Visualization Approach

As stated in Section 1.2, most previous studies (15, 26, 27, 28) propose visualization approaches for code smells rather than for smell agglomerations. That is because smell agglomerations have been introduced in the literature very recently. Moreover, the existing visualization approaches for smell agglomerations are very limited (10, 30). Aimed at addressing such limitation, this dissertation introduces a novel graph-based visualization approach (Section 3). We chose to represent smell agglomeration through a graph-based abstraction for the following reason. Our approach is based on the finding of a recent study of ours (29), which observed that it is good to start with a general view of agglomeration and then see a specific view. The graph-based structure gives the whole view of the agglomeration and the developer can analyze later for each smell.

The study (29) suggests that the structure of smell agglomerations might fit a graph representation since each smell agglomeration has multiple code smells; and these code smells interrelate. Thus, representing each design element (i.e., a class or an interface) d as a node, and the interrelation between two design elements d_1 and d_2 as an edge $e(d_1, d_2)$, seems promising. In our visualization approach, an interrelation might be either a method call, an inheritance relationship, or an interface implementation.

After proposing the graph-based visualization approach for smell agglomerations, we implement it as a Web application using the D3 API[1]. Finally, we evaluate to what extent our approach supports developers in identifying design problems through an empirical study. By recruiting ten developers from the industry, we asked the developers to identify design problems affecting the software systems that they have helped implement via the graph-based visualization approach. Our results suggest that developers reached an up to 100% of precision and 42% of recall in the identification of design problems. The most useful features of our approach were the graph-based abstraction and the representation of dependencies among design elements affected by code smells. However, the low rate of recall reached by the participants suggests that our approach was not sufficient to properly support developers in their daily basis.

1.4 Re-thinking our Visualization Approach

Evaluating our graph-based visualization approach has allowed us to refine it based on precision and recall results, but also on the developers'

[1] https://d3js.org/

feedback about the missing information (Section 3.4). However, we have observed that our approach required even more complex refinements. Because of that, this dissertation proposes another visualization approach for smell agglomerations called VISADEP. VISADEP approach evolves the previous one by combining multiple views of a single smell agglomeration at three design abstraction levels: component (or package), class, and method. The component view is the view of the highest level of abstraction. The purpose of the component view is to provide a more general view of the organization of design elements by showing code smells that affect a component and concerns implemented in a component. The class view is a zoom-in to the component view; it represents the classes of a component that are affected by an agglomeration by showing concerns implemented in each class, the code smells that affect each class and the relationships between the classes. The last view, which is the method view, serves to offer another possibility of zoom-in, but now applied to a single class. The method view is the one that is as fine-grained as possible because it represents how the agglomeration is affecting the methods of the classes. Our approach supports developers to navigate through these views so they can better reason about the smell agglomeration towards the identification of design problems. Our approach relies on the Unified Modeling Language (UML) (19) notation for representing classes, text for code smells, arrows for interrelations, and colors for concerns. We proposed an Eclipse (31) plugin, also called VISADEP, to support our approach to visualization.

Then, we conducted a controlled experiment to evaluate our novel approach with developers from both academy and industry comparing the new approach with an improved graph-based visualization approach. We refined the initial proposition adding the feature like concern representation and textual information about concerns. During the experiment, participants identified design problems in 2 steps; at each step, they used different tools analyzing 2 different subsystems of Apache OODT. The results of the study reveal that for Concern Overload and Ambiguous Interface (4) design problems, our novel approach supported by the tool VISADEP was better than the enhanced graph-based visualization. Although there is not much difference between the two tools, for Scattered Concern, VISADEP was not better than the graph representation, but it did not underperformed with respect to the graph-based approach. During the experiment, we also asked the participants to report the elements of the visualization that helped them to find the design problems, in order to improve our approach based on what the developers mentioned more. Based on the results, we plan to improve our visualization approach to

better support the identification of Fat Interface (2), where VISADEP did not achieve a satisfactory result. For instance, VISADEP could represent other information such as the method parameters.

1.5 Dissertation Outline

The remainder of this dissertation is organized as follows.

Chapter 2 provides background information aimed at supporting the understanding of this dissertation. This chapter also discusses related work in order to contextualize this dissertation with respect to the literature.

Chapter 3 describes our preliminary visualization approach for smell agglomerations based on a graph visual structure. This chapter also discusses the results of an empirical study aimed at evaluating our approach. We conducted this study with software developers from the industry.

Chapter 4 introduces VISADEP, our novel approach for visualizing smell agglomerations. Based on the results of our evaluation described in Chapter 3, we re-think the visualization of smell agglomeration. We also introduce the VISADEP tool that implements our novel visualization approach.

Chapter 5 presents and discusses the results of an empirical study aimed at evaluating VISADEP. This study was conducted with software developers from both industry and academy.

Finally, **Chapter 6** summarizes the dissertation. This chapter discusses our main contributions and suggests future work.

2
Background and Related Work

Design problems (1) indicate violations of key design principles in a software system. Due to the non-local nature of most design problems, they might harm the maintainability of multiple design elements together (25). Previous studies (8, 32, 33) suggest that code smells provide useful hints about a design problem. However, identifying design problems is difficult for developers (29). Recently, studies (10, 12) provide evidence that a single code smell might not suffice to reveal a design problem. These studies suggest that code smells often interrelate to realize a design problem through the so-called smell agglomerations (10). However, it remains difficult for developers to identify design problems, because a single smell agglomeration might contain several code smells. In fact, it is difficult for a developer to reason about multiple code smells and their interrelations to properly identify a design problem.

Software visualization approaches have been largely used to summarize information about the source code, so that developers can easily reason about maintainability problems (13, 34). Thus, these approaches emerge as potentially useful means for representing smell agglomerations, so developers could reason about the interrelated code smells and identify design problems. Previous studies (29, 30) present the first visualization approaches for smell agglomerations. However, they represent very limited information about code smells and their interrelations. Consequently, developers still find it difficult to identify design problems via smell agglomerations. This dissertation addresses such limitation by introducing and evaluating a novel approach for visualizing smell agglomerations. Our goal is to support developers in reasoning about a smell agglomeration towards the identification of design problems.

This chapter presents background information and discusses related work. The chapter is organized as follows. Section 2.1 discusses about design problems. Section 2.2 overviews code smells and smell agglomerations. Section 2.3 discusses the state of the art about the visualization of either code smells or smell agglomerations. Section 2.4 presents a desiderata of visualization approaches for smell agglomerations. Finally, Section 2.5 concludes the chapter.

2.1 Design Problems

The violation of one or more design principles in a software system characterizes a design problem (1). Design principles represent guidelines for decomposing the functionalities of a software system into design elements. Design principles aim at supporting the modularization towards high maintainability of the software system (1). As a consequence, by violating design principles, developers might decrease the system maintainability, which often leads to an increase in maintenance cost and effort (25). Many design problems affect critical program locations, i.e., design elements that centralize the main system functionalities (35), such as interfaces and class hierarchies. We illustrate to what extent design problems violate design principles and might affect multiple design elements as follows.

Scattered Concern (2, 4) is a recurring design problem characterized by the violation of a basic principle known as Separation of Concerns (2). This principle states that each design element should realize exactly one concern. Thus, whenever multiple design elements realize the same concern, we have a violation of the Separation of Concerns principle. Additionally, some of those components are responsible for an orthogonal concern. In this context, a concern is anything a stakeholder may want to consider as a conceptual unit, including features, nonfunctional requirements, and design idioms (20). Scattered Concern also affects the program reusability (4) because developers cannot reuse the implementation of the spread concern without using other components that implement the same concern.

On the other hand, Fat Interface (1, 2) occurs when a design component offers only a general, ambiguous entry-point that provides non-cohesive functionalities, thereby complicating the clients' logic. This design problem affects properties like cohesion, abstraction, and separation of concerns. Moreover, Fat Interface is known to be a major source of major maintenance effort in large-scale software systems (3, 10). Whenever developers have to modify or evolve the interface, they have to introduce changes across many classes that either implement or use the interface, including classes that have no conceptual relation to the change or evolution being incorporated into the system.

Such examples show that, when neglected, a design problem like this one may cause harmful consequences to the system, such as redesign or even discontinuation of a system (36). Thus, the identification and removal of design problems are required for long-living systems. However, identifying design problems is not trivial. This difficulty stems from the fact that design problems are often spread over several program elements; thus developers have to locate

and to inspect multiple code elements that are part of the design problem. In this context, this dissertation focuses on helping developers during the analysis of code elements to identify design problems.

Dissertation Scopes. Table 2.1 presents the description of each design problem that we investigate in this dissertation. The first column of the table describes the type of design problem. The second column specifies if the design problem is related to concerns or structural issues. The third column is the definition of each design problem type. The last column shows the papers that investigated each design problem. The design problems that we investigate are: Ambiguous Interface, Concern Overload, Fat Interface, Scattered Concern, and Unwanted Dependency (37). The scope of design problems is varied. Some of the design problems studied are related to the concerns that each program element implements. These problems are Concern Overload and Scattered Concern. Other design problems are related to structural issues involving key design elements (e.g., interfaces or dependencies) in the source code. These problems are Ambiguous Interface, Fat Interface, and Unwanted Dependency. In other words, the problems studied cover, in general, the most elementary levels of the design and implementation of a system (e.g., concerns, interface implementation, and dependencies between classes).

Table 2.1: List of Design Problem Types

Design Problem	**Concern/ Structure**	**Description**	**Papers**
Ambiguous Interface	Structure	Interface that offers only a single, general entry-point, but provides two or more functionalities (4)	(4, 32)
Concern Overload	Concern	Design components that are responsible for realizing two or more unrelated system's concerns (3)	(3, 10, 29, 32)
Fat Interface	Structure	Interface of a design component that offers only a general, ambiguous entry-point that provides non-cohesive functionalities, thereby complicating the clients' logic (2)	(10, 29)
Scattered Concern	Concern	Multiple components that are responsible for realizing a crosscutting concern (2)	(4, 10, 29)
Unwanted Dependency	Structure	Dependency that violates an intended design rule (37)	(10, 29)

We highlight that the investigated design problems have distinct characteristics, which require different information about the program elements, such as classes, interfaces, and methods. We also selected these design problems because they are frequently investigated in the literature (3, 29, 32). In our first study (Chapter 3) was aimed to evaluate the graph-based visualization, we

explored three of these design problems; 2 design problems related to the mis-modularization of concerns (Scattered Concern and Concern Overload) and 1 related to dependency (Unwanted Dependency). Then, in the second study (Chapter 5), we used 4 of those design problems; 2 design problems related to the mis-modularization of concerns (Scattered Concern and Concern Overload) and 2 related to interfaces (Ambiguous Interface, Fat Interface).

2.2 Code Smells and Smell Agglomerations

Code Smells. Developers can use different symptoms to identify a design problem in the source code. One of these symptoms is called code smell. Code smell is a microstructure in the source code that may indicate the manifestation of a design problem (8). Code smells vary from those detected at the method-level (e.g., Long Parameter List, Long Method, and Feature Envy) to those detected at the class level (e.g., Complex Class, God Class, Data Class, Shotgun Surgery, and Divergent Change) (8, 25). Table 2.2 describes the types of code smells used in this research. We selected these types of code smell because: (i) they are well-grounded in the literature (8, 38), (ii) there is tool support for detecting them with source code metrics, (iii) they were extensively used in case studies and controlled experiments (3, 10, 29, 32, 39), and (iv) they represent smells often related to the design problems addressed in this dissertation (10). In this dissertation, we use the terms **smelly element** to refer to a program element that contains code smells.

Table 2.2: Types of Code Smell

Type	Description
Brain Method	Long and complex method that centralizes the intelligence of a class
Data Class	Class that contains data but not behavior related to the data
Disperse Coupling	The case of an operation which is excessively tied to many other operations in the system, and additionally these provider methods that are dispersed among many classes
Feature Envy	Method that calls more methods of a single external class than the internal methods of its own inner class
God Class	Long and complex class that centralizes the intelligence of the system
Intensive Coupling	When a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes
Refused Parent Bequest	Subclass that does not use the protected methods of its superclass
Shotgun Surgery	This anomaly is evident when you must change lots of pieces of code in different places simply to add a new or extended piece of behavior
Tradition Breaker	Subclass that provides a large set of services that are unrelated to services provided by the superclass

As each design problem is often spread over two or more code elements, the analysis of individual code smells is often not sufficient to identify a design problem (39). In addition, the analysis of code smells to reveal design problems tends to be difficult and time-consuming. This happens because developers often need to analyze and discard several code smells unrelated to any design problem (3). Even for small software systems, there are hundreds of smells (3). Thus, in order to effectively identify design problems, developers need to determine and prioritize which code smells should be analyzed.

Agglomerations of Code Smells. Recent studies indicate that design problems are likely to be located in program elements that contain multiple code smells (3, 10, 11). Oizumi et al. (10) proposed the notion of code smell agglomeration. A code smell agglomeration is a group of interrelated code smells in the program. The agglomeration is determined in the program by the co-occurrence of two or more code smells syntactically or semantically related. The agglomeration is located in the same method, class, hierarchy or component (10). According to the correlation analyses in their study, agglomerations are often the locus of design problems with an accuracy higher than 80%. Some categories of code smell agglomeration are: intra-method agglomeration, intra-class agglomeration, hierarchic agglomeration, intra-component agglomeration, and concern-based agglomeration (39). In this dissertation, we are interested in agglomerations that occur in hierarchies and components such as intra-component agglomeration, hierarchic agglomeration, and concern-based agglomeration (39). We selected these agglomerations because these agglomerations can involve several elements in the source code. Then a visualization could assist the developer in the analysis. Table 2.3 presents the description (39) of the categories of agglomeration considered in this dissertation and Figures 2.1, 2.2, and 2.3 are examples of each category of agglomeration.

Table 2.3: Categories of Agglomeration

Category	Description
Hierarchical	Agglomerations composed by program elements that are affected by the same type of code smell, and these elements implement the same interface or inherit from the same upper element (e.g., a superclass).
Intra-component	Agglomerations of code smells that occur inside of a single design component. This agglomeration comprises program elements that are located within a single component, and the elements are affected by the same type of code smell. The elements also must be connected by method calls or type references.
Concern-based	Agglomerations composed by smelly program elements that are located in the same component and these inner elements of the component implement diverse concerns.

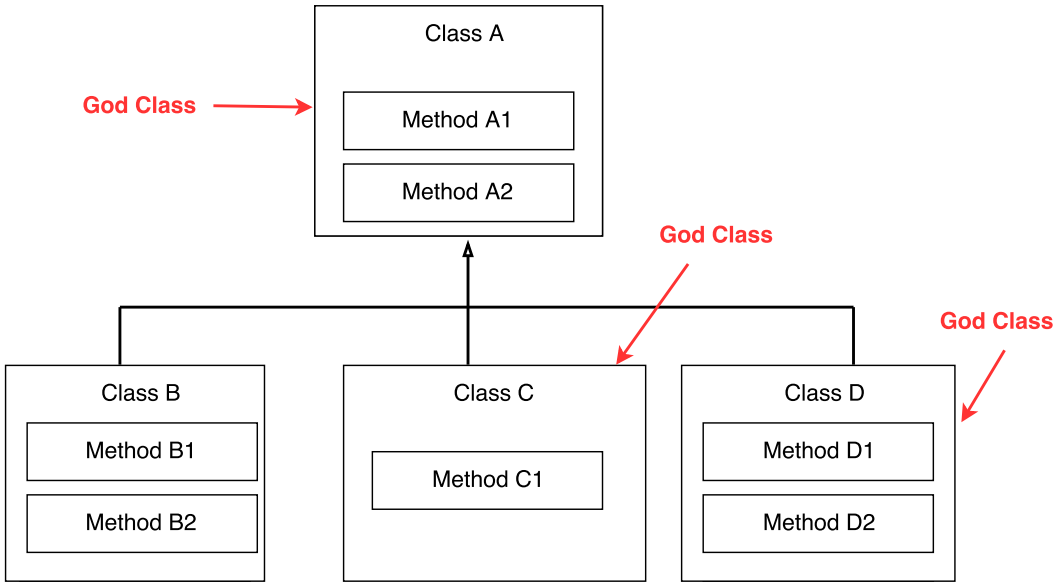

Figure 2.1: Hierarchic Agglomeration Example

An example of hierarchic agglomeration is illustrated in Figure In figure 2.1, a hierarchic agglomeration is shown. Classes B, C, and D inherit from A. A, C, and D are affected by the same type of code smell, God Class. These God Class instances affecting A, C, and D form a hierarchic agglomeration of God Classes. When we have a class hierarchy in object-oriented programming, it means that: the parent class implements the basic features that the daughter classes will inherit; each of the child classes implements its own specialty but, in the end, it also inherits and makes use of parent class implementations. So, if the code smells affect the parent class, and these code smells are problematic in terms of code maintenance, then the child classes will also suffer from maintenance problems. For example in Figure 2.1, the parent Class A that has a God Class is certainly very large, complex, and non-cohesive. Therefore, reading, understanding, and maintaining daughter classes can also be difficult because these daughter classes tend to be as large, complex, and non-cohesive as the parent class. In this case, a hierarchy agglomeration represents this problem that crosses the multiple classes that are in a hierarchical relation. This kind of agglomeration can help to identify the design problems Fat Interface and Ambiguous Interface (40, 41).

Figure 2.2 illustrates an intra-component agglomeration. As it can be observed, classes A and B belong to the same component. Besides class A and class B contain the same type of code smell, Feature Envy. These Feature Envy instances affecting A2 and B1 can form an intra-component agglomeration of Feature Envy. Intra-component agglomerations can mainly help to identify the design problems Unwanted Dependency and Scattered Concern (40, 41)

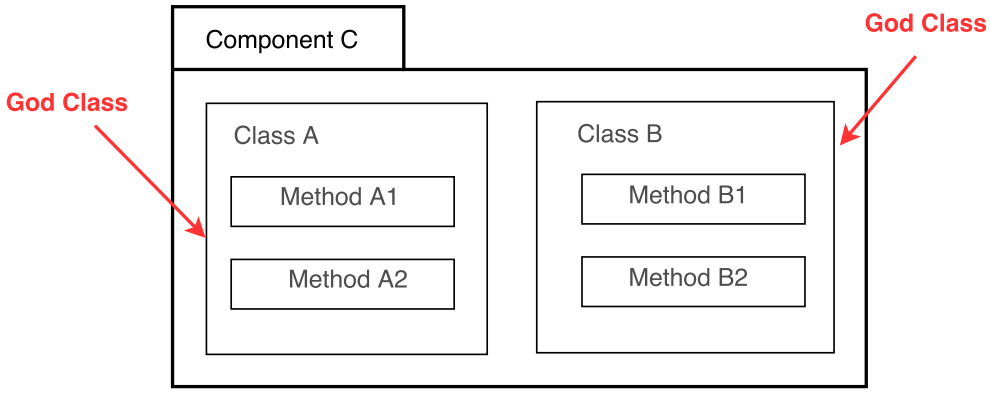

Figure 2.2: Intra-component Agglomeration Example

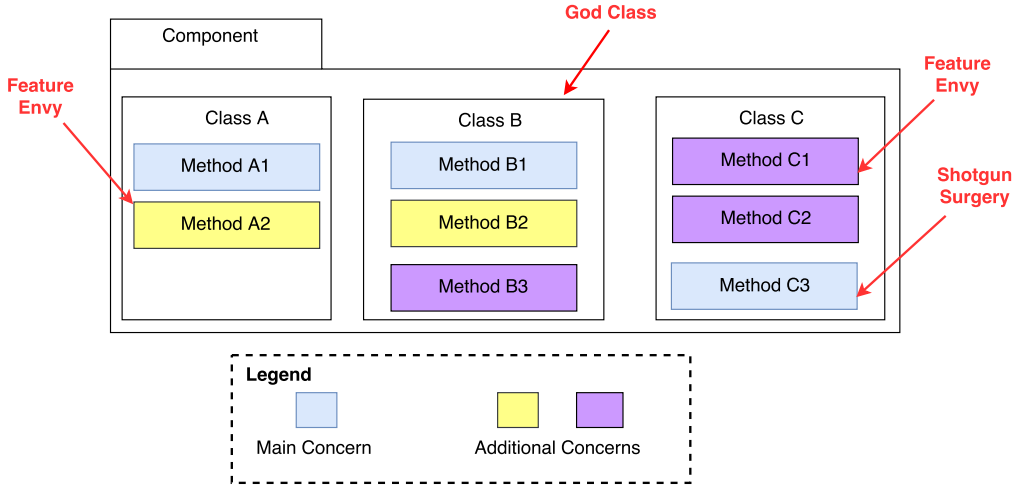

Figure 2.3: Concern-based Agglomeration Example

In figure 2.3, we illustrate a concern-based agglomeration. In the figure, classes A, B and C belong to the same component. Each class in the component implements a number of concerns. All classes implement the main concern (blue color). However, each of them implements at least one additional concern. For example, Class B implements two additional concerns. Besides, all classes in the component contain at least a code smell (e.g., Feature Envy, God Class, Shotgun Surgery). Then, the concern-based agglomeration is composed by the 4 code smells that affect the three classes. Concern-based agglomerations can mainly help to identify the design problem Concern Overload (40, 41)

2.3 Existing Visualization of Smells and Agglomerations

Software visualization transfers information about software artifacts (e.g., source code) into visual forms and aims at enhancing information understanding in software development (42). The visualization of one or more software artifacts is called software visualization. Software visualization en-

compasses the development and evaluation of methods for graphically representing different aspects of software (42). The purpose of the visualization is to help developers understand their software system.

According to Diehl (42), we can categorize three types of software visualization: Structure, Behavior, and Evolution. Structure visualization refers to the visualization of static parts (43) of the software, including the source code, architectural design, or specific static metrics. Static program visualization or structure visualization covers the visualization of textual and diagrammatic representations, visualization of results of program analyses and the visualization of software architecture (42). Behavior visualization refers to the visualization of the software execution, i.e., the dynamic part of a software. Evolution visualization refers to the visualization of how software evolves over time; for instance, it allows to represent code changes incorporated by developers along the version history of a system.

There are several studies that investigated visualization for code smells (15, 44, 45, 46). We found these literature works using an ad hoc review. There is a risk of not catching important papers, but most of the studies were captured from a literature review (47). However, we found few studies that proposed approaches for visualizing smell agglomerations (29, 30). Thus, we studied and compared existing approaches for visually representing code smells and smell agglomerations. This comparison helped us to identify their weaknesses as well as requirements of a visualization approach for agglomerations. We present below some previous work that proposed visualization for code smells (Section 2.3.1) and work that proposed visualization for smells and agglomerations (Section 2.3.2).

2.3.1 Visualization of Smells

We reviewed previous work about visualization of code smells. This review was also intended to define a baseline of expected requirements four our own approach. In this context, Murphy-Hill et al. (15) present Stench Blossom (Figure 2.4), an Eclipse plugin that provides code smell detection and visualization with a focus on improving interactivity by integrating the visualization into the source code editor. Using Stench Blossom, programmers are capable of switching between a quick and high-level overview of the smells while they are programming. This plugin represents the smells with petals; the size of a petal is directly proportional to the "strength" of the smell in the code element it refers. Stench Blossom creators claimed that embedding the visualization in the actual context of developer's programming makes the tool

useful for both smell identification and refactoring. However, the visualization of Stench Blossom limits the programmer in having an overview of the entire system. It shows only the state of actual code being written by the programmer without the possibility of analyzing other classes. It is often the case that even a single code smell can affect many other classes related to the current code being written or analyzed. This narrow view of code smells may misguide developers in prioritizing code smells that are more critical to the system. Moreover, the visualization of Stench Blossom makes it hard to group smells, located in different classes, that may indicate together a single design problem.

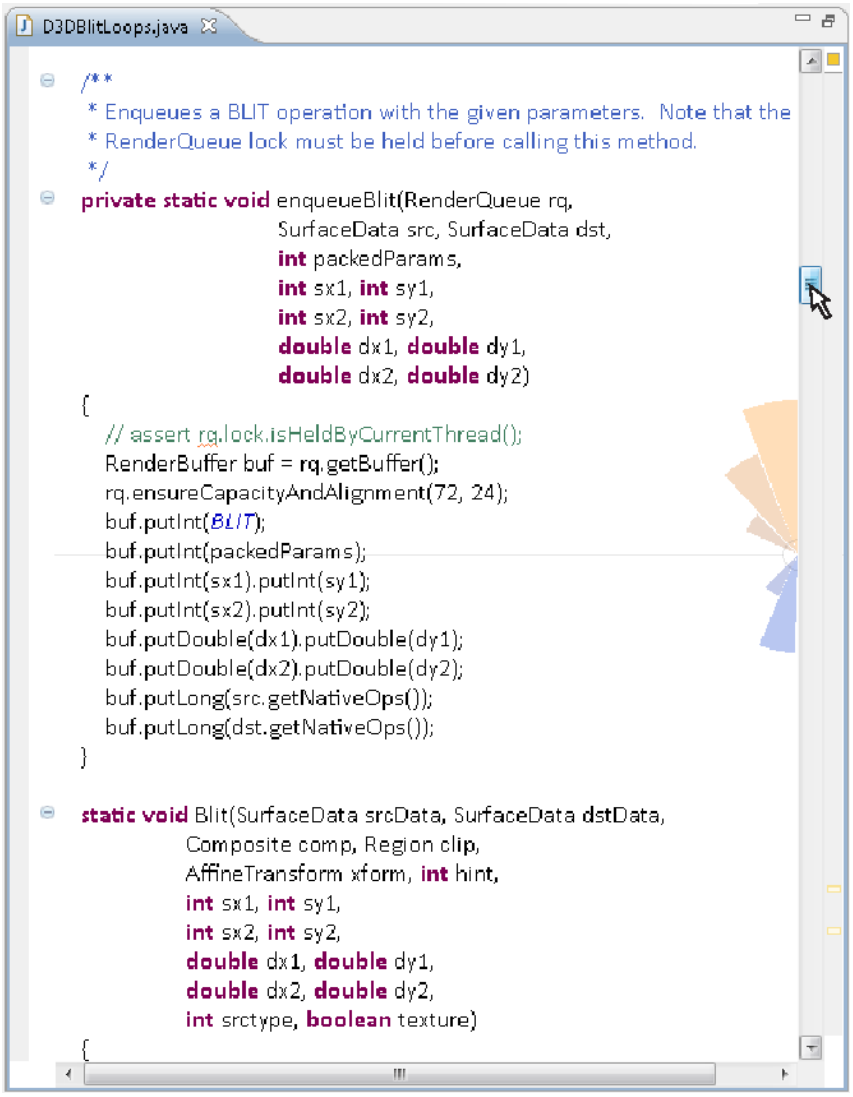

Figure 2.4: Stench Blossom Ambient View

Marinescu and colleagues (44) present InCODE, an Eclipse plugin that supports detection and visualization of code smells that can indicate design problems. When the Eclipse workbench starts, InCODE starts to analyze (in the background) the source file currently active in the editor. When a candidate design problem is detected, a red marker is placed next to the affected class

or method. It also implements a visualization detailing the components (i.e., packages) of the system and its level of affectation by code smells, as shown in Figure 2.5. Unfortunately, the visualization excludes dependencies between packages, which are useful to understand the structure of the system In addition, the visualization of this tool does not explicitly represent smell agglomerations.

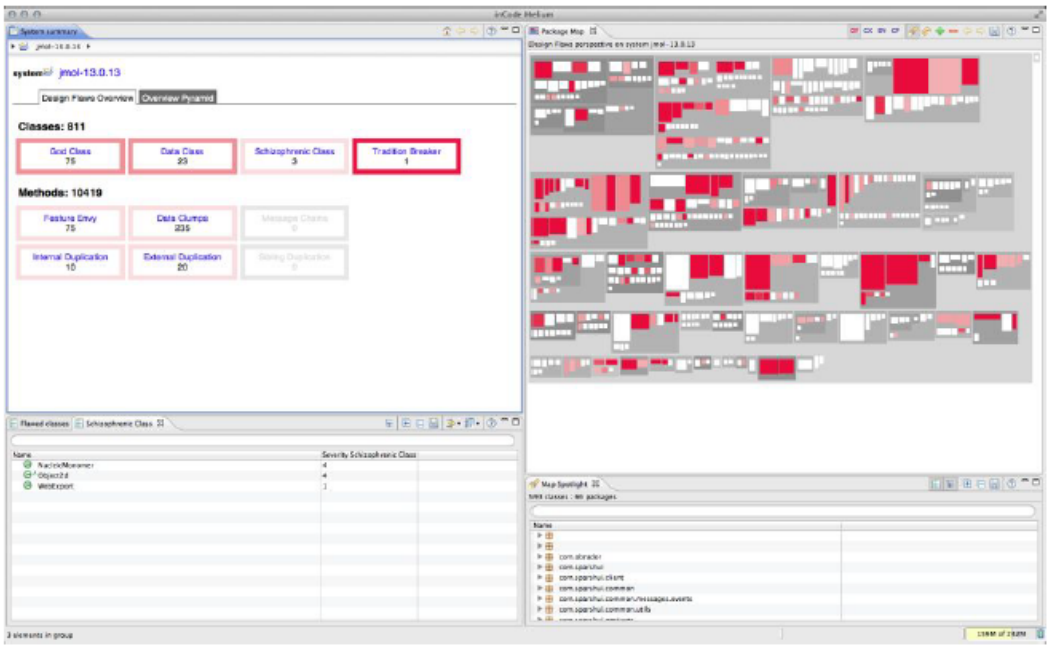

Figure 2.5: Incode Packages Overview

There are other studies (20, 49) that investigated the detection and visualization of code smells through the explicit representation of concerns. According to Robillard et al. (20), many code smells are caused by the particular ways one or more stakeholders' concerns are structured in the source code. In this context, Carneiro et al. present SourceMiner (49), an Eclipse plug-in which provides four categories of code views (Figure 2.6) with concern properties presented as follows.

The first visualization of Carneiro and colleagues (49), called concern's package-class-method structure view, is a 2D visualization that maps a tree structure into rectangles where each rectangle represents a program element. The visualization represents how modules are organized in packages, classes, and methods. The second visualization, called concern's inheritance-wise structure view, is a two-dimensional display that uses rectangles to represent classes and interfaces and edges to represent inheritance relationships between them. The rectangles colored in dark blue correspond to classes or interfaces that are affected by a specific concern selected by the programmer. The third one, called concern dependency view, is based on a graph view that represents the

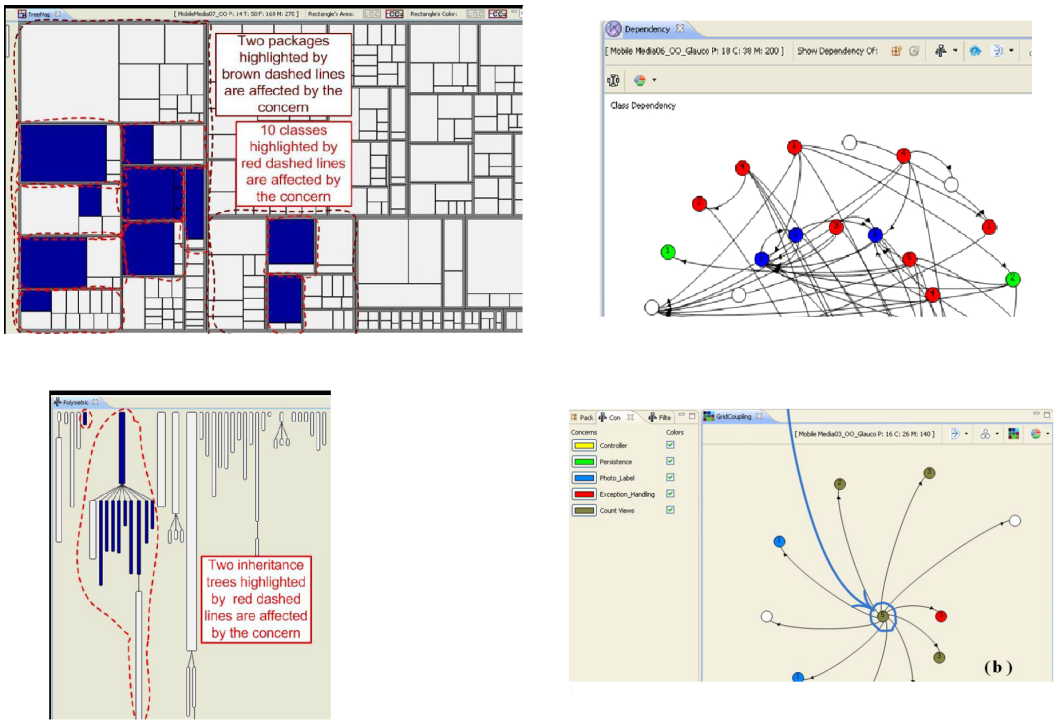

Figure 2.6: SourceMiner Views

dependencies among software packages and classes. Finally, the fourth one, called concern dependency weight view, is a complement to the concern dependency view. It displays the weight of each dependency, i.e., the number of syntactic references to a module in the dependent module. SourceMiner requires the developer manually filters the metrics used for detecting smells. Unfortunately, SourceMiner does not reveal relationships between code smells. Moreover, it does not provide developers with the possibility of navigating from the system's visualizations and the source code (and vice-versa).

2.3.2 Visualization of Smell Agglomeration

Regarding detection and visualization of code smell agglomeration, we found in the literature two tools: JSpIRIT (30) and Organic (29). JSpIRIT is an Eclipse plugin for detecting code smells and agglomerations of a (Java-based) system and ranking them according to different criteria (30). The main benefit of using JSpIRIT is that developers can configure and extend the tool by providing different strategies to identify and rank the smells and groups of smells (i.e., agglomerations). JSpIRIT proposes visualizations for intra-component agglomeration, hierarchic agglomeration, and intra-class agglomeration (10). Figure 2.7 presents two snapshots of the visualization of intra-component agglomeration and hierarchic agglomeration. The snapshot A

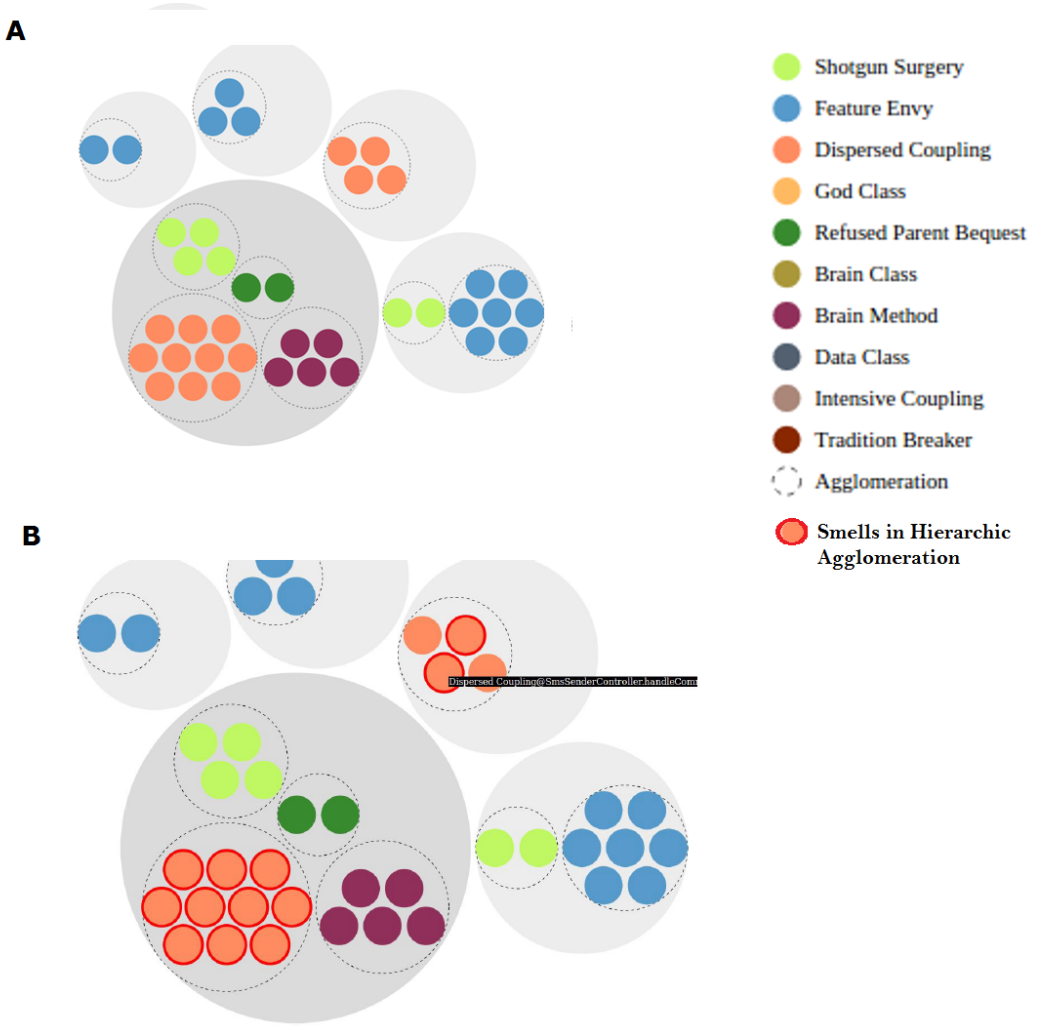

Figure 2.7: JSpIRIT:Intra-component and Hierarchic Agglomerations

presents the visualization of the agglomerations while snapshot B highlights the smells that are of a hierarchic agglomeration. In the figure, the larger grayish circles represent packages. Inside each package, the dotted outline circles are agglomerations and the smaller colored circles represent different smell types. When the mouse is over a smell, if this is a member of a hierarchical agglomeration, this and the rest of the members of the agglomeration are highlighted with a red outline as shown in the snapshot B (Figure 2.7). Figure 2.8 shows the intra-class agglomeration visualization. An intra-class agglomeration is a class (including its methods) affected by two or more code smells; i.e., the code smells (taking part of the agglomeration) are located within members of a single class. The model chosen for representation of intra-class resembles a flower; from the center outward, system packages are represented and connected to classes; at the same time, arising from each class, are smells that form the intra-class agglomeration. A disadvantage of this tool is that it represents the code smells of agglomeration (Figure 2.7 and

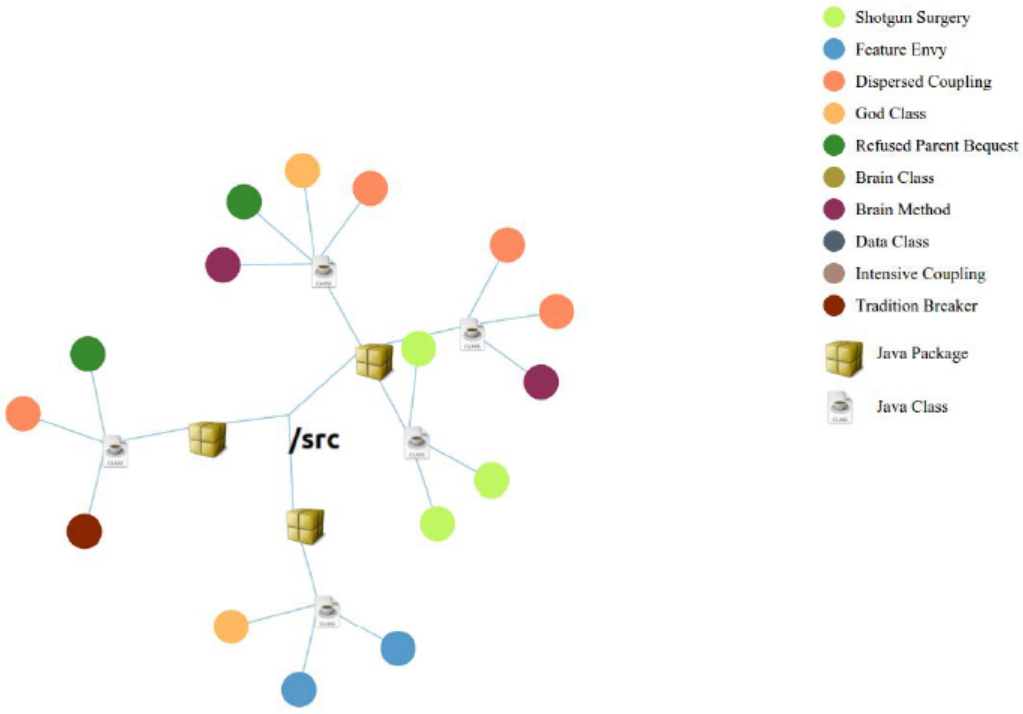

Figure 2.8: JSpIRIT:Intra-class

Figure 2.8) without showing the smelly elements and the relations that could exist between the elements.

Organic (29) provides detection and visualization of smell agglomerations. It is a plugin for Eclipse designed for Java programs and for helping developers in identifying design problems in the source code. Organic captures and represents the history of code smells that progressively compose an agglomeration along the versions of a system's evolution. It also provides the references of each program element affected by the selected agglomeration. Another feature is a graphical representation of agglomeration, which shows the elements that compose an agglomeration. However, it does not represent the dependencies between those elements. Organic also captures and leverages semantic relationships between smells that form an agglomeration; semantically-related smells are those smells that realize a single developer's concern, such as persistence, error handling and other high-priority requirements of a system (Figure 2.9). Oizumi et al. (29) conducted a study with some goals similar to ours. Their study compared the use of agglomerations with the use of individual code smells and also performed a qualitative analysis to evaluate the Organic tool based on semiotic engineering. They noticed during the data analysis that the proposed visualization mechanism was a core reason on why some developers had difficulties to identify design problems using smell agglomerations. Due to this result, we are investigating whether

our proposed visualization approach is suitable to support the identification of design problems.

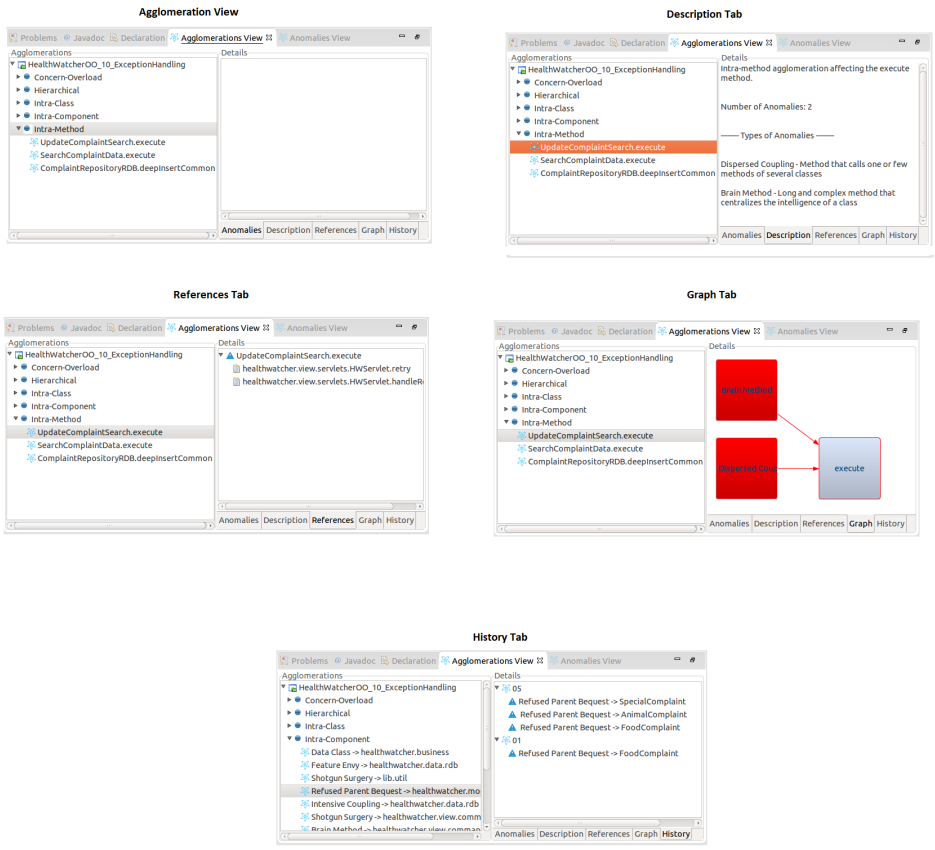

Figure 2.9: Organic Visualization

Table 2.4 presents a comparative description of the visualization approaches presented above. The second column of the table describes the purpose of the visualization and the approaches used to represent smells, agglomerations or concerns. The third and fourth columns describe their positive and negative aspects respectively.

2.4 Desiderata of Code Smell Agglomeration Visualization

After a comparative study of the approaches in terms of their characteristics, this section presents a brief overview of the derived requirements that may be useful for designing by approaches for visualizing smell agglomerations. We present below the overview of these requirements:

A visualization approach should be simple and easy to understand. The visualization must be presented in a way that the first time the developer sees the agglomeration, she understands it. In particular, the de-

Table 2.4: Comparative Study of Visualization Approaches

Tool	Visualization Characteristics	Advantages	Drawbacks
Stench Blossom	Visualization of code smell occurrences in a petal-like metaphor, located in the right side of the affected code element.	Easy navigation from visualization to source code (and vice-versa).	Does not provide a general view of code smells affecting multiple classes.
InCode	Visualization of packages with its level of affectation by code smells	Visualization of code smells at the package level	No visualization of the relationship among packages
SourceMiner	Four concern-based visualizations: package-class-method view, inheritance-wise view, dependency view, and dependency weight view	Support to package dependency visualization via graphs	Non-intuitive and overloaded visualization. No mapping between code smells and concerns
JSpIRIT (code smells)	Heat map-like visualization of code smells through the system packages	Support to package dependency visualization	Overloaded visualization with low comprehensibility to large-sized systems. No mapping between code smells and packages
JSpIRIT (agglomerations)	Two agglomeration visualizations: intra-component view based on circle and color metaphors, and intra-class view based on a flower-like metaphor	Support to agglomeration sorting at the package level, zooming, and navigation through affected code elements	No visualization of dependencies between code smells and agglomerations
Organic	Multiple views of code smell agglomerations	Support to navigate to the affected source code per agglomeration. History data per system version	No visualization of dependencies among code elements affected by code smells

veloper should be able to easy understand how the agglomeration is affecting the multiple elements of a program. This will allow easy access to the agglomeration's information and encourage him to identify a design problem. The existing visualization approaches are either too general or too detailed to represent the smell agglomerations. For example, the approaches behind CodeCity, Stench Blossom are not appropriate. Thus, represent agglomerations via these approaches can make hard to reason about a smell agglomeration to identify design problem. Besides, in large systems where several elements are involved in the problem of design Scattered Concern, if the visualization is not simple it would be difficult to identify this problem because there would be too much information, too many classes in the visualization.

A visualization approach could rely on a graph-based structure. We have evidence from the study Oizumi et al. (29) (see Section 1.2 for details) that a graph structure can be an indicator to represent agglomerations. Analyzing the property of an agglomeration, there is a natural mapping between the smell agglomeration structure and a graph structure. Let consider vertexes as the smelly elements and the edges as the smell interrelations. This is possible because there are several smells interrelated in an agglomeration. Thus, we can abstract smell agglomeration in a graph structure. So, it is a natural mapping that makes sense. In addition, we could rely on graph

because some of the design problems (Fat Interface, Ambiguous Interface) are associated with hierarchical structures of code elements. For example, for Unwanted Dependency it is necessary to show that there are dependencies between elements that should not and the graph can be used to show this.

A visualization approach should represent all code smell interrelations that characterize a smell agglomeration. It is important to represent the relationships between code elements affected by code smells. This information will serve to characterize how code smells are grouped while forming an agglomeration. If these relationships are not properly detected and presented to the developers, they are unlikely to make proper meaning out of a flat list of agglomerated smells. As a consequence, developers may fail to understand how two or more smells are contributing together to the realization of a single design problem. For instance, in a hierarchic agglomeration, it is necessary to show the relationships that compose the inheritance chains. In the case of both intra-component agglomeration and concern-based agglomeration, it is necessary to show all relationships between code smells, i.e., the inheritance, association, and interface implementation relationships.

The visualization approaches should differently represent each type of code smells interrelations. The nature of each interrelation might differ (inheritance, interface implementation, and association). Each type of interrelation is associated with a specific design problem. For example association is fundamental to identify problem design Unwanted Dependency. Since Unwanted Dependency by definition means looking at the dependencies between classes and the association represents calls from one class to another. The implementation of interface is fundamental to identify Fat Interface because if the developer does not know if there is an interface, the developer can not know if there is a Fat Interface or not, and if it implements many non-cohesive concerns. Finally, inheritance can help eliminate false positives in identifying design problems Scattered Concern and Concern Overload. This is because the existence of an inheritance can help to decide whether the existence of a code structure is a problem or not. Each of these relationships must have a specific representation to facilitate the identification of specific design problems.

The approach should avoid information overload whenever possible. The approach should represent relevant information about agglomerations the smelly elements that compose an agglomeration. An agglomeration, by definition, has too much information to be presented given the huge amount of smells and their relationships that usually compose a single agglomeration. When designing a new form of visualization, it is important to prioritize ag-

glomeration's data so that the developer will not be overwhelmed with the amount of information offered. For instance, the approach should offer a high-level view of the agglomeration and the multiple levels of visualization, from the most abstract to the most concrete. The notion of relevance may be captured, for instance, as the developer interacts with the visualization approach, as discussed above. Note that we might need to represent all code smell interrelations that characterize a smell agglomeration. Thus, certain information could be omitted to the developers and revealed only when it becomes necessary, for instance.

2.5 Summary

This chapter presented background information aimed at supporting the proper understanding of this dissertation. Since our study focuses on the proposal of visualization mechanisms for code smell agglomerations, which are expected to be useful to the identification of design problems, we discussed concepts such as design problems and their negative effects on the software maintainability, the identification of design problems in the source code, as well as software visualization. This chapter also presented existing visualization approaches to support the understanding of state of the art about visualization of code smells and agglomerations. In addition, we also made a comparative analysis of the visualizations to determine what positive aspects we can use or adapt in our proposed visualizations. We also discussed their negative aspects that should be avoided by our visualization proposal.

After introducing the main concepts and the literature review, the next chapter presents our first visualization alternative for agglomeration and its evaluation through an exploratory study.

3
A Graph-Based Visualization of Agglomerations: First Study

The identification of a design problem is not a trivial task. For instance, when a developer is analyzing an agglomeration to reveal a design problem, the developer also 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 technique is likely to help the developer in the analysis of an agglomeration. In this context, Oizumi et al. (29) conducted a study that aimed to compare the precision in identifying design problems when developers use agglomerations versus the use of single smells. Their results revealed that agglomerations help developers in revealing more design problems than single smells.

However, the precision of smell agglomeration in identifying design problems could have even been better if the difficulties faced by the developers are addressed. For example, the authors observed that the information provided by an agglomeration would be best used if such information was synthesized in a single, concise visual representation. Also, another result of the study was that developers reported that agglomerations with code smells spread in hierarchies or class packages are often difficult, time-consuming and requires adequate visualization support. Thus, developers showed an interest in a general view of the code elements involved in the agglomeration. Another reason for the low precision, according to the authors, was the lack of adequate visualization that sums up all the necessary information about an agglomeration to identify design problems. Moreover, their study revealed form the feedback of the experiment participants that visualizing agglomeration through a graph structure can help to provide an overview of an agglomeration.

Based on the aforementioned requirements, we proposed a graph-based visualization approach for code smell agglomerations. The purpose of this visualization is to give an overview of the program elements that are affected by the agglomeration and the relationships between these elements. Besides, we defined a graph-based visualization since it is a technique commonly used to show the relationships between elements (49, 50, 51). Additionally, graphs are frequently used for information visualization (52). The graph-based visualization can help developers during the analysis of the agglomeration.

For example, it can help developers to identify all the elements affected by the agglomerated smells and help them to see the relationships among these smelly elements, *i.e.*, an element with code smell. In the context of design problem identification, developers need to reflect upon these relationships in order to better understand if their co-existence may imply a more severe design problem.

In this chapter, we present characteristics of the graph-based visualization in Section 3.1. Section 3.2 describes an exploratory study whose purpose was to investigate whether the graph could actually support the identification of design problems. Section 3.3 and 3.4 present and discuss the experiment results. Finally, Section 3.5 describes the threats that could limit the validity of the study.

3.1 Graph-Based Visualization for Code Smell Agglomeration

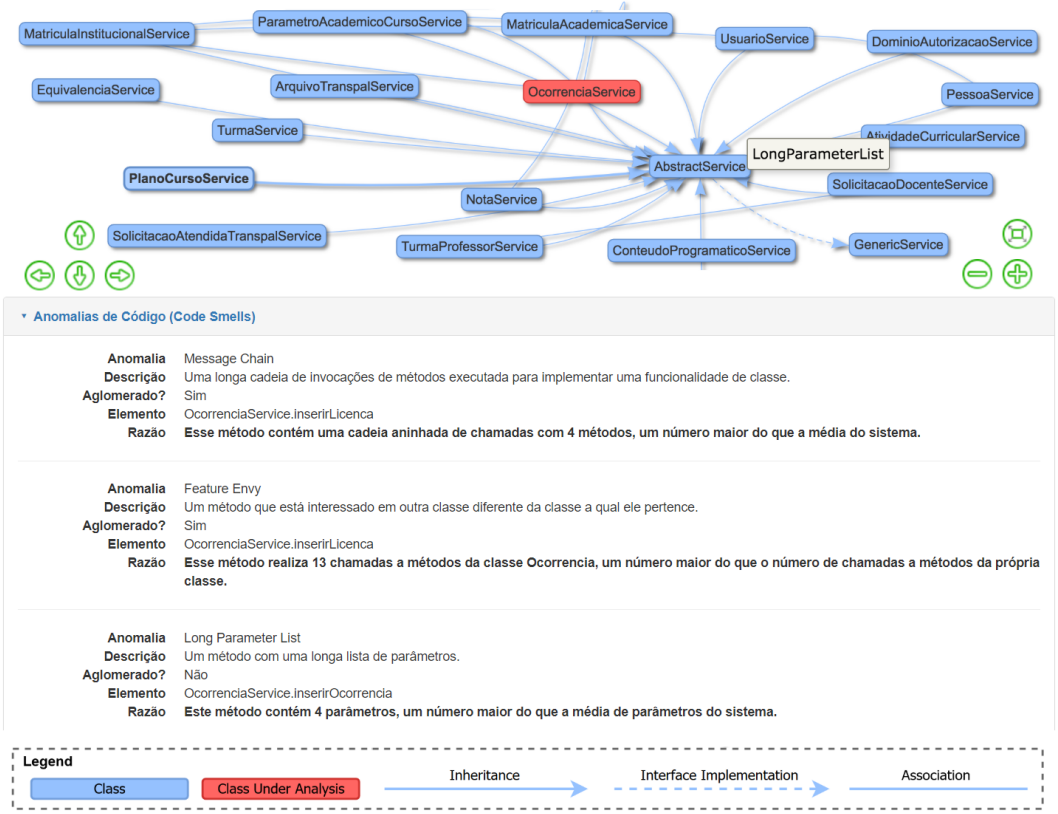

Figure 3.1: Graph-based visualization for agglomeration

Figure 3.1 shows an example of the graph-based visualization for smell agglomerations. This visualization uses nodes (rectangles) to represent smelly elements, such as classes and interfaces with one or more smells, and edges (lines and arrows) to represent the dependency relation between smelly ele-

ments. The visualization uses different lines and arrows to represent the three types of relationships 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 Section 3.1, OcorrenciaService inherits from AbstractService. A straight line represents a simple association between two classes. All the classes involved in the agglomeration are presented in 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).

In the graph, we present only the program elements involved in the agglomeration and their relationships in the source code with the other program elements in the agglomeration. To show the hierarchic agglomeration, in the graph we show the superclass, the subclasses and the relationship of inheritance or interface implementation and the relation of calls between the elements of the program. On the other hand, intra-component agglomerations and concern-based agglomerations are represented by showing the classes of the component involved in an agglomeration and the relationships between these elements.

Below the graph, we have a section that shows the description of the code smells that affect the classes or interfaces of the graph. This description contains the name of the code smell, the definition of the code smell in literature, the name of the affected element and the reason for the existence of the smell. For example, the code smell Message Chain affects the InserirLicencia method of the OccorenciaService class. The reason for the existence is the call chains with 4 methods. The description also shows further information on whether the code smell is part of the agglomeration or not.

How does the graph-based approach meet the requirements of Section 2.4?

We designed the graph-based visualization not only based on the requirements but also on the empirical evidence (29) as discussed at the beginning of this chapter. In this case, in order to avoid the undesired effect of information overload in the graph, we decided to incorporate in the graph only what was the basic information to identify design problems. Besides, our proposal is intended to meet the requirement of being simple and easy to use because we used a graph structure, which is well known among developers, in order to facilitate their learning. 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 relationships among these smelly elements. We intend to assess the graph-based visualization approach for the purpose of investigating the precision of the identification of design problems. We also investigate to what extent the visualization supports developers on the identification of design problems from the viewpoint of researchers and software developers.

3.2 Study Protocol

This section describes our study protocol. Section 3.2.1 introduces our study goal and research questions. Section 3.2.2 contains information about the procedures we followed to select participants for our experiment. Section 3.2.3 presents details about the experiment procedures. Section 3.2.4 describes the artifacts used during this study. Finally, Section 3.2.5 describes how the data was analyzed in this study.

3.2.1 Goal and Specific Research Questions

We intend to assess the graph-based visualization approach for the purpose of investigating the precision of the identification of design problems. We also investigate to what extent the visualization supports developers on the identification of design problems from the viewpoint of researchers and software developers. The context comprises professional and novice developers reviewing the source code of software projects developed by themselves. We conducted an exploratory study aimed to answer the following research questions:

RQ1: How accurate is the identification of design problems using the graph-based visualization?

RQ2: To what extent does the graph-based visualization of smell agglomerations support the identification of design problems?

RQ3: How to improve the proposed visualization of smell agglomerations?

To address these research questions, we performed a quantitative and qualitative analysis. First, we analyzed the number of design problems correctly identified to calculate the precision and recall. For this purpose, we have selected the reference list of design problems of the systems that will be analyzed. Then we compared the identification of design problems made by the developers to calculate the precision and recall. Later, we analyzed the developers' opinion about the relevance of the visualization in supporting the identification of design problems to understand the reason behind each

classification. Then, we observed how developers used the graph-based visualization, which allowed us to identify features that can be used to improve the visualization of smell agglomerations.

3.2.2 Study Participants

To answer both research questions, we selected two Brazilian software companies (CO1 and CO2) to collect software systems for analysis and select developers to act as participants in the study. The company selection was based on their different features including the adoption of code reviews, the number of developers, their application domains, and development process. As each selected company has to provide software systems, we selected Java (53) projects given the popularity of the Java programming language (54, 55). Java is a representative language of object orientation. Besides, object-oriented programming was chosen because most design problems that have been documented in the literature address this programming paradigm. The selected companies and programs are described as follows.

CO1: This company is incubated at a northeastern university, and they provide systems for the university. One of their systems is responsible for all undergraduate academic control and registration. It was developed by the Nucleus of Information Technology (NTI). The software was built in a monolithic way using the Spring Framework, JBoss Seam, and Hibernate frameworks. It went into production in the second half of 2010 and is still in use today.

CO2: This company deals with renting and selling print devices. Also, they have a department specialized in software development. Their software system deals with tracking the process of arrival and departure of print devices; contract deals with their clients and replacement and reusability of compatible machinery pieces.

Table 3.1 shows the details about the two software systems. The first column is the company software systems (S1 for CO1 system, S2 for CO2 system). The second column shows the programming languages that are used in each system. The third column is the number of files in each language. The fourth column is the number of blank lines given a file. The fifth column is the number of comments. Finally, the last column is the number of line of code.

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

Table 3.1: Software System Details

System	Language	Files	Blank	Comment	LOC (18)
S1 of CO1	Java	994	20211	2596	71327
	JavaServer Faces	247	3904	26	21657
	Javascript	12	416	873	3793
	HTML	9	12	0	3731
	XML	237	875	242	3207
	Maven	5	195	66	2949
	CSS	6	95	200	771
	DTD	1	48	50	129
	Visualforce component	1	0	0	24
	SUM	1516	25785	4060	1083381
S2 of CO2	Java	141	3467	1231	11729
	Javascript	70	1067	378	660
	CSS	27	749	110	5166
	JSON	16	7	0	3360
	Maven	1	16	3	317
	YAML	2	12	0	93
	XML	2	2	0	26
	SQL	1	3	2	14
	Bourne Shell	1	0	0	6
	SUM	323	5703	1940	33661

regarding their knowledge of software development concepts, Java programming language, code smell, and design problems. Some characteristics of the developers are presented in Table 3.2. The first column indicates the identification number of the name of the company of the developer, and the second column indicates the identification number of the developer. The third informs the level of graduation of each participant. The fourth indicates the number of years the participants work in the industry. The fifth column shows the number of years of experience with Java language. The sixth column informs whether the participants have performed review a least one before. Practice that might impact the ability of developers to reason about the source code (56). The seventh and the last columns show the familiarity of the participants with the term code smells and design problem respectively. When a developer knows code smell, he could better understand the smell agglomerations to find design problems. Similarly, the knowledge about design problem could help the developer on how to find design problems.

As we can see in the table, the developers of the two companies have at least two years of experience in the industry. Except for the developer D8 of company 2, all participants have at least two years of experience with Java. Half of the developers in the two companies has performed at least once the code review. Also, the table shows that at least half the developers know the concept of code smell and design problems.

Table 3.2: Characterization of the Participants

Company	ID	Education	Experience in Industry (Years)	Experience with Java (Years)	Perform Code Review?	Familiar with the term Code Smells?	Familiar with the term Design Problem?
CO1	D1	UnderB.Sc.	3	3	No	No	No
	D2	B.Sc.	5	10	No	No	Yes
	D3	Specialization	13	13	Yes	No	Yes
	D4	M.Sc.	14	12	Yes	Yes	Yes
	D5	B.Sc.	14	12	Yes	No	Yes
	D6	M.Sc.	6	6	No	Yes	Yes
CO2	D7	B.Sc.	7	6	Yes	Yes	Yes
	D8	UnderB.Sc.	2	0	Yes	Yes	Yes
	D9	UnderB.Sc.	4	5	No	Yes	No
	D10	UnderB.Sc.	4	4	No	Yes	Yes

3.2.3 Study Procedure

To answer our research questions, the study was structured into three main activities: a training session and a session comprising the tasks of design problem identification.

Activity 1: The training was organized in two parts: the first took 25 minutes and addressed the explanation of the concepts (e.g., code smell, design problem, and graph-based visualization). We presented various examples of design problems pertaining to different categories (Section 2.1) We selected the design problems together with the project managers, who suspected that these represented common cases of design problems in their projects. However, we let clear to the participants (developers) that they were allowed to identify other types of design problems. The second one (approx. 15 minutes long) was devoted to discussion and questions, if necessary. The second part took approximately 10 minutes and was dedicated to discussion and questions about the concepts.

Activity 2: In this activity, the developers were provided with the list of smell agglomerations detected in their software system, the graph-based visualization for each agglomeration, and the description of the code smells. We highlighted that participants were free to use or not those code smells. We distributed the participants of each company in pairs, and we asked them to identify the design problems in pairs to encourage developers to discuss the design problems. The discussions of this activity gave us more input for our qualitative data analysis. Each pair analyzed the selected agglomerations in their company software system. Developers took on average 1 hour and 45 minutes to complete this activity. They reviewed software systems they developed. 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. Each pair evaluated five instances of agglomeration visualization, except a team who evaluated four. The developers

evaluated the same instances of agglomeration.

Activity 3: In this activity, the developers answered a questionnaire (post-experiment form) about the usefulness of the visualization of agglomerations. The answers were also used to complement the qualitative analysis.

3.2.4 Instrumentation

The main goal of instrumentation is to provide means for performing the experiment and its monitoring. The instruments of an experiment can be classified into objects, guidance and measuring instruments (57). Thus, the experiment was conducted in environments provided by the companies. The participants used a computer (object), in which we installed the applications needed to perform the experiment. As a guide to the experiment the participants, we used an online form with the graph-based visualization. As measuring instrument, we used a form in which participants transcribed the design problems identified and the relevance of the graph-based, and we also used a reference list of design problems. In order to support the study, we required the following artifacts (for more details, see our study companion website (58)):

- *Companies characterization form* (Table 3.1). The overall goal of this form was to characterize the companies. The answers obtained through this form allowed us to identify some key characteristics about the company such as the number of developers, the products, and services of the companies, if the companies perform code review in general, code smell code review or design problem code review, etc. Firstly, we sent a week before the experiment an invitation to companies. Then, we did presentations for the companies explaining the objective of our study. Then we asked them to fill out the characterization form, to indicate the systems. The company characterization form was completed by the owner of the company who is also a developer in the company.
- *Consent form.* The consent form was used to get authorization from the subjects to use their data in this study.
- *Subjects characterization form* (Table 3.2). The characterization form aimed at collecting the subject background and working experience.
- *Online form.* This form helps to collect data about the identified design problems from the analysis of smell agglomerations and the classifications of the relevance of each graph-based visualization instance. The online form page is available but requires authentication. It was fulfilled online

by the participants, and the answers were saved on our server. To analyze the data (participants' answers) we need to see it in the Django database.

– *Post-experiment form*. The post-experiment form aimed at capturing feedback from the subjects after the experiment.

– *List of smell agglomerations*. We used well-known metrics-based strategies to identify code smells from Fowler's Catalog (8). The agglomerations of smells were detected with the Organic tool (39).

– *Reference list of design problems*. We spent a week before the day of the experiment, analyzing the systems of companies to create the graph. When we analyzed the systems and found design problems or thought it was design problems, we sent the companies to validate whether it was design problems or not. Our reference list of design problems has been validated by experienced developers which are involved in the company's systems. These developers are not the same people who participated in our study. Our reference list contains three design problems; 2 design problems related to the mis-modularization of concerns (Scattered Concern and Concern Overload) and 1 related to dependency (Unwanted Dependency).

– *Graph-based*. The graph-based tool is the computational tool that supports the graph-based approach we are assessing in this study. Implementation of the graph: the graph was implemented using two technologies: Django[1] and D3.js[2]. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. D3.js is a JavaScript library for manipulating documents based on data. Django was used to create database access, create summary pages (pages with the information about the agglomeration and the visualization). The web page was created using Django (it creates the web page and the access to the database). We saved it in Django's database, and we used D3.js to display the graph. Then, we get all the information and save it to the Django database. We created the Django script that reads the database and displays the information on the web page. In the same way, we created the graph. We saved in the database, the annotations (elements, smells, and relationships) for generating the graph. Then the script also reads the information from the graph in the database and calls the D3 that displayed the graph.

[1]Django. Available at https://www.djangoproject.com/
[2]D3.js. Available at https://d3js.org/

– Software: Eclipse IDE (31), Camtasia[3]. Developers used the Eclipse IDE (Integrated Development Environment) to import and analyze the source code. We used Camtasia to record a video of the computer screen as well as the audio. In addition, we installed a video camera in the room in order to record the video and audio around the participants, creating an audio and record environment.

3.2.5 Data Analysis

We analyzed the collected data using the coding process from the Grounded Theory (GT) (60). 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). For instance, we related codes about dependencies, analysis of agglomeration, negative aspect of the visualization. The analysis of the codes allowed us to answer our research questions.

3.3 Results

RQ1. *How accurate is the identification of design problems using graph-based visualization?*

Based on the design problems reported by the developers and the list of design problems, we compute the precision and recall achieved by the pairs in the use of graph-based visualization to answer this research question. The quantitative analysis shows that the graph-based approach enabled the identification of design problems. Considering the results of all pairs, we compute 13 True Positives, 0 False Positive, and 18 False Negatives. The concern of this study was to see if the visual elements and information we provide in the graph are useful to identify problem designs. As our concern was to assess visual information we selected agglomerations with at least one design problem; and there the chance of the developer making a mistake is less. That explains the fact of not having false positives. Then, the precision rate is 100% considering the result of all pairs. A 100% of precision means that when participants use the graph-based approach to identify design problems, they can correctly identify problems; the approach does not lead to misidentification

[3]Camtasia Studio. Available at https://www.techsmith.com/camtasia.html

(i.e., it does not lead to false positives). However, another important measure is the recall because it captures the proportion of correctly identified design problems among the full list of existing design problems. However, the recall is not high (42%) if we compare it with the high rate of precision. The reason for the low recall is because to identify some problems that are quite hidden in the code the developer needs extensive and time-consuming reasoning.

To better understand if the graphical visualization had any contribution to the previous results, we analyzed the number of times participants evaluated visualizations in the four scales. We also analyzed in they found or not the design problems. Both analyses are represents in Table 3.3. By considering the responses of visualization classified as Relevant or Highly Relevant, we can see that in 36.4% of responses participants found at least one design problem against 27.2% in which participants did not find any design problems. The results in Table 3.3 suggests that most of the participants who found the design problems said that the visualization was relevant or highly relevant.

Table 3.3: Visualization Relevance and Design Problem Responses

Perception	**Did the pair identify design problems?**			
	Yes		**No**	
	#	%	#	%
Irrelevant	1	4.5	1	4.5
Slightly relevant	2	9.1	4	18.2
Relevant	2	9.1	3	13.6
Highly relevant	6	27.3	3	13.6

RQ2. To what extent does the graph-based visualization of smell agglomerations support the identification of design problems?

Relevance of a Graph-based Visualization. The graph-based visualization is relevant for identifying design problems. Regarding research question RQ2, Figure 3.2 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 scores 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%). Pair 1 and Pair 5 had negative result with respect to the visualization's relevance. Pair 1 tried to justify their response about visual-

izations classified an irrelevant: this pair argued that the dependency graph was only relevant to a class, not to identify the design problem. On the other hand, the Pair 5 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 (Section 3.2.5) to better understand these results.

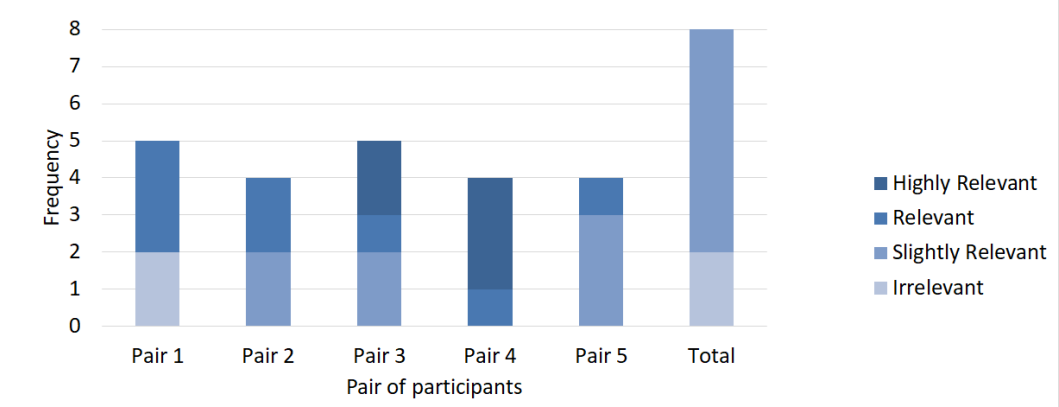

Figure 3.2: Relevance Classification for the Graph-based Visualization

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 smell 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 Pair 3 was analyzing the agglomeration of Figure 3.1, they mentioned that they remembered nothing about the smelly elements. Thus, they were not only 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 in which Pair 1 classified the visualization as irrelevant. In a few other 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 did not already know.

3.4 Limitations

RQ3. *How to improve the proposed visualization of smell agglomerations?*

Bottom-up and top-down analyzes of agglomerations. A single visualization may not represent both how the smells are scattered over several elements and different levels in the system. The projection of agglomerations in different modular software structures (component-level, class-level, etc.), are not enabled characteristics in the graph approach, and some developers said, in fact, that the graph visualization had few details.

Lack of representation of concerns. We observed that developers found many barriers on the identification of certain design problems, such as Concern Overload and Scattered Concern because the graph-based visualization misses several relevant details about concern implementation in the program elements. The lack of concerns affected for example the recall. When the design problem was related to the concern implementations, developers could not identify as a result, recall went down quite a bit. By designing the graph, we thought that the representation of the concerns could overload the visualization, but we noticed in the evaluation of the graph approach that the concerns are basic information for the identification of the selected design problems. For example, to identify a Scattered Concern design problem, developers need to identify the program elements that implement the spreading concern, as well as relationships between the smelly elements. By representing the concerns, developers might deeply understand the origin of the code smells and the severity of the design problem.

Developers need explicit and easy access to information. 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, which might vary from one design problem to another. There are a few hints of the most relevant information in the literature (3, 4, 10), but further studies should be conducted to systematically elicit this information per design problem. In our implementation, we have two perspectives: one that shows the graph-based visualization and another 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 Section 3.1. 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 at 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 them, at some point, give up on using the visualization to guide their analysis. We noticed that, because they did not use 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 observed 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 such navigation was not supported, the developers often spent considerable time to analyze the agglomeration's symptoms in the source code. This kind of analysis should be facilitated because it is critical to confirm the symptoms provided by the visualization. Thus, by having two-way navigation between visualization and source code, developers may be able to effectively perform the analysis of agglomerations to identify design problems.

3.5 Threats to Validity

This section discusses some threats to the validity (57) of our study results. For each threat, we present the actions taken to mitigate their impact

on the research results.

Construct Validity. The major risk here is related to possible errors in the validation of identification design problems. To mitigate this threat, we recruited developers, who had extensive knowledge of their system's design. They also had previous experience with design reviews and source code inspections. Developer familiarity with the analyzed systems is a threat to our study. This may have influenced the high precision (100%) achieved by the developers using the graph-based visualization. To mitigate this, we present the same agglomerations for different developers. Regarding the computation of smell agglomeration instances for conducting the experiment, we have applied a technique reported by the literature (10). The precision of this technique is equal to 80% in the identification of design problems. Besides, a threat is the sampling of computed agglomerations to form the set of agglomerations that participants have to reason about to identify design problems. This is a threat because we can not represent all agglomerations computed from a system, especially when these agglomerations are large. For instance, participants would not be able to reason about each agglomeration given the experiment time constraint of 1 hour and 30 minutes. To mitigate this threat, we try to choose agglomerations that are not too large. So when choosing the design problem, we needed to choose agglomerations that help identify that design problem in a short time.

Also, we are not sure that the selected agglomeration is the ideal one to find the design problem. Eventually, more than one agglomeration of smells may be required to identify a design problem. In fact, some agglomerations only include smells of the same type (for example, only Feature Envy instances). Thus the experiment participant may have failed to identify the design problem, not because the agglomeration is not good, not even because the visualization is not adequate, but because the agglomeration we provided was not enough. We mitigate this by doing a manual analysis of each agglomeration to see if it provides information that is minimal enough to help identify the design problem that co-occurs with it in the source code. This mitigation was done by a single author based on the opinions of experts and experienced developers who worked on the systems evaluated.

Another threat to the validity of this study is the selected design problems. It may be that there are other design problems in the analyzed systems that the employees of the companies that validated the design problems forgot to mention. It is also possible that the authors did not find all existing design problems in the initial analyzes performed on the systems. It could also be that design problems validated by the employees of the companies are not

design problems. To mitigate the threats in the design problem set, all the design problems we encountered analyzing the systems were validated one week before the experiment by experienced developers who worked on the systems that were not the same developers that participated in our study.

Internal Validity. We conducted a careful data collection to minimize problem concerning missing data, the possibility that the author may have introduced bias in the data collection process. To mitigate this threat, the process of data collection was performed alongside other two researchers. There is a threat concerning the inherent difficulty in the experimental tasks. To reduce this threat, we trained all participants to resolve any gaps in knowledge or conflicts about the experiment. In addition, we performed pilot experiments with volunteers to improve both experiment design and artifacts. There is a threat regarding the time of the experiment. To mitigate this threat we adjusted the time required to perform the identification tasks in the pilot phase.

Conclusion Validity. This threat concerns the relation between treatment and outcome. We tried to mitigate it by combining data from different resources: quantitative and qualitative data obtained with videos, and questionnaires. We believe data collection and analysis were properly built to answer our questions. To mitigate this threat, the process of data analysis was performed alongside other two researchers.

External Validity Some factor may prevent the generalization of our results (Section 3.3). A threat to the study validity is the limited diversity of contexts involved in the study. However, we argue that the selected companies represent typical software development organizations as shown in Table 3.1. Another threat to the study is the variety of design problems selected. In this study the number of types of design problems considered is limited. The problem it causes is the difficulty of generalizing our results to any design problem. We mitigate this threat by selecting for the study 2 categories of design problems related to concern and dependency. Finally, the set of subjects considered in this study is small. In fact, our study involved a sample of 10 developers, divided into 5 pairs, which may not be enough to generalize our results (Section 3.3). Thus, we cannot generalize our conclusions. This threat was mitigated by selecting developers with varied backgrounds and at least two years of experience with industry software development.

3.6 Summary

In this chapter, we reported on our proposal of graph-based visualization. We also performed a first study to assess whether the graph-based visualization can help developers to identify design problems. Our 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. The size of the graph and the visualization of dependencies between smelly elements allow developers to have a comprehensive overview of the design problems. The study also allowed to identify some missing features in our graph-based visualization. We noticed that the developer needs important details about agglomerations that our approach did not provide, such as concern representation, the representation of agglomerations across different modular software structures (component level, class level, and method level), as well as the navigation from the visualization to the source code.

After presenting our initial proposal of visualization for smell agglomeration and applied the evaluation of the visualization, the next chapter proposes a new visualization approach based on barriers faced by developers during the graph-based visualization study.

4
Re-thinking the Visualization of Smell Agglomerations

Based on the findings of the initial proposal (Section 3.3 and 3.4), we found that the graph-based visualization can be useful to identify design problems. In fact, representing the design elements affected by code smells as graph nodes, together with their interrelations represented via graph edges, has a potential to guide developers in identifying violations of design principles. That is because a design principle mostly concerns the organization of design elements in a system. Thus, certain violations of these design principles might be perceived by developers through the visual analysis of interrelations and smelly elements. For instance, a Concern Overload (3) problem is often indicated by too many interrelations between a design element and others, which suggest that this design element realizes too many concerns that should be realized by a single design element each.

However, there were various limitations in our previous proposal (Section 3.4) that should be properly addressed. For instance, when reasoning about a Concern Overload, the developers have to understand what concerns are realized by each design element. However, our first approach was unable to reveal such information and help developers in this sense. As a consequence, it has hindered developers in the identification of this design problem and several others. Therefore, we decided to rethink the visualization of agglomerations based on the lessons learned in the previous study. We defined a novel visualization approach called **VISADEP**. VISADEP stands for VISualization Approach for identifying DEsign Problems. VISADEP is intended to: (i) leverage strengths as well as addressing weaknesses of our previous proposal, and (ii) support new capabilities missed by developers in our previous study.

This novel approach provides mechanisms to users to navigate from the high-level representation to the low-level representation of an agglomeration. Our approach supports multiple views for smell agglomerations; it provides detailed views on different levels of the agglomeration, namely component-level, class-level, and method level decomposition. The component view is the view of the highest level of abstraction. The purpose of the component view is to provide a more general view of the organization of design elements. The class view has a somewhat more specific goal; it represents the classes of a

component that are affected by an agglomeration. The class view is a zoom-in to the component view. The last view, which is the method view, is the one that is as fine-grained as possible because it represents how the agglomeration is affecting the methods of the classes. Then it serves to offer another possibility of zoom-in but now applied to a single class. For each class, the zoom-in is applied, and we reveal that the list of the methods affected by agglomeration. In addition, VISADEP combines multiple data sources that developers have to analyze to reason about an agglomeration. VISADEP also provides the concern representation and easy navigation from the visualization to the source code, features that the graph approach did not have (Section 3.4). We used the ConcernMapper[1] tool to compute the concerns of the evaluated systems and then we got an output file that we passed as input to our tool. Then, our approach has processed this file. Our approach associated the concerns that were documented in the files with the design elements that we show in the visualization.

In this chapter, we present in Section 4.1 the information presented in the VISADEP approach. Section 4.2 describes the mockup of the multi-view representation of the agglomeration at the component, class and method levels. Finally, Section 4.3 presents the VISADEP tool which is a support tool for the VISADEP approach.

4.1 Represented Information of VISADEP

The graph-based assessment (Section 3.4) revealed that the representation of dependencies is important to help developers understand the agglomeration. However, although the accuracy was high, the recall was low due to the limitations of our graph-based approach. We have identified several limitations to take into account to improve our approach. We noticed that developers face many barriers to identify design problems with the graph-based visualization. The reason is a graph-based is often too abstract to provide all the detailed information needed to find a design problem. Developers tried to search for different information for each type of design problem. Following, we present all the information that VISADEP provides.

Firstly, VISADEP propagates the positive aspects of graph-based approach and proposes a solution for the limitations observed in the graph-based assessment. As the dependencies in the graph were used to identify some design problems, we do not need to rethink it, but we improved the representation to make it even more interesting for the developer. We changed the representation

[1]ConcernMapper. Available at https://www.cs.mcgill.ca/ martin/cm/

of the nodes and edges. We left the abstract representation nodes to give way to UML notations to represent the classes and edges by an arrow with a label (which defines the name of the type of relationship, e.g., Uses). Although the nodes and edges remained, we observed the following limitations and each one of them; we are representing or providing in the new approach in the following ways.

One of the limitations of the graph-based approach study was the lack of details about the agglomeration to help in the bottom-up and top-down analysis of the agglomeration. Our initial study showed that developers need a higher-level and low-level abstraction to understand agglomeration. So, to solve this limitation, our new approach proposes three complementary views of agglomerations. Our approach allows seeing the code smells at different levels of abstraction: component, class, and method. Each agglomeration can be visualized at the level of the component; thus, we present to the developers both the code smells affecting the component and the concerns that are implemented by the component. While at the class-level, we show developers a zoom on the component by showing classes of the component that contains code smells in the agglomeration. The method-level provides the names of the affected method in each class of the component.

We also discussed in our previous study that concern representation was a limitation to the low recall we achieved. So we decided to change the structure of the graph so that it can show the concerns. Rethinking our approach, we decided to represent the concerns using colors. The concerns are represented in any of the views provided by VISADEP, for each program element (either a component, a class, or an interface). Concern representation is relevant to identify design problems because the program elements and concern representation allows knowing when a component or a class is implementing functionality that it should not and, in fact, some design problems, such as Scattered Concern, Fat Interface, and Concern Overload, are related to the mis-modularized implementation of concerns.

We observed in our previous study (Chapter 3) that different information on source code is needed to identify design problems. We noticed that, for each design problem, the developers tried to search for different information. This led to the need to elicit what information was necessary for each design problem being considered in our study. Based on this, we identify which information developers require to identify each design problem in the scope of this dissertation. In this context, the types of information supported by the VISADEP approach are: (i) the program elements affected by code smells, (ii) the code smells, (iii) the dependencies between the program elements,

and (iv) the concern representation. The visualization does not only show the syntactic relations between smelly elements, but also the semantic ones. The semantic relation is the relation between program elements that address the same concern in the program. With this information, it is possible to analyze the relations between program elements that take part of different forms of agglomeration (Section 2.4).

4.2 VISADEP Mockups: Visualizing Agglomerations across Components, Classes and Methods

We propose global and detailed views of agglomerations, which represent their manifestation in different software decompositions, namely component-level, class-level, and method-level decomposition. Our approach includes three complementary views of an agglomeration. We call these views according to the level they represent: component-level, class-level, and method-level. The component-level view comprises the highest level of abstraction for an agglomeration (component), while the class-level and method-level views comprise the detailed levels of abstraction for each agglomeration.

We proposed three views based on what we observed in our preliminary experiments, where we noticed that developers navigate through the classes to components, either in a bottom-up or top-down fashion. Some developers like to see first a macro-view of the program structure and, then, inspect structural details of the inner elements of each component; others prefer to start from the micro-view and, then, go upwards to understand the implications of smell structures to higher-level elements. The visualization used the Unified Modeling Language (UML) notation to represent components, classes, and methods. Rectangles are used to represent classes, while rectangles combined with a component icon are used to represent the components. Methods of a class are represented by text, which describes the method signature of the program. The boxes are similar to those used for UML classes.

Our first approach used a graph notation (nodes and edges), but we think it is better to use another notation more appropriate to the representation of design elements, that is, the UML. In any case, we use UML classes to represent each class being affected by a code smell. These classes correspond to the nodes of our graph-based approach; thus, we embedded to the class representation all their relations, which correspond to the edges of our graph-based approach. The views of our approach show only the code elements (class, methods) that are part of the agglomeration to avoid information overload. In the following, we describe the views.

The **Component view** shows the architectural components affected by the agglomeration. In this work, we assume that each component is realized as a package in a Java program. Figure 4.1 illustrates the mockup of the component view. The colors represent concerns implemented by inner elements of a component. We use colors to represent concerns because there might be several concerns being realized by a single component. Thus, we need to have a different representation for each concern.

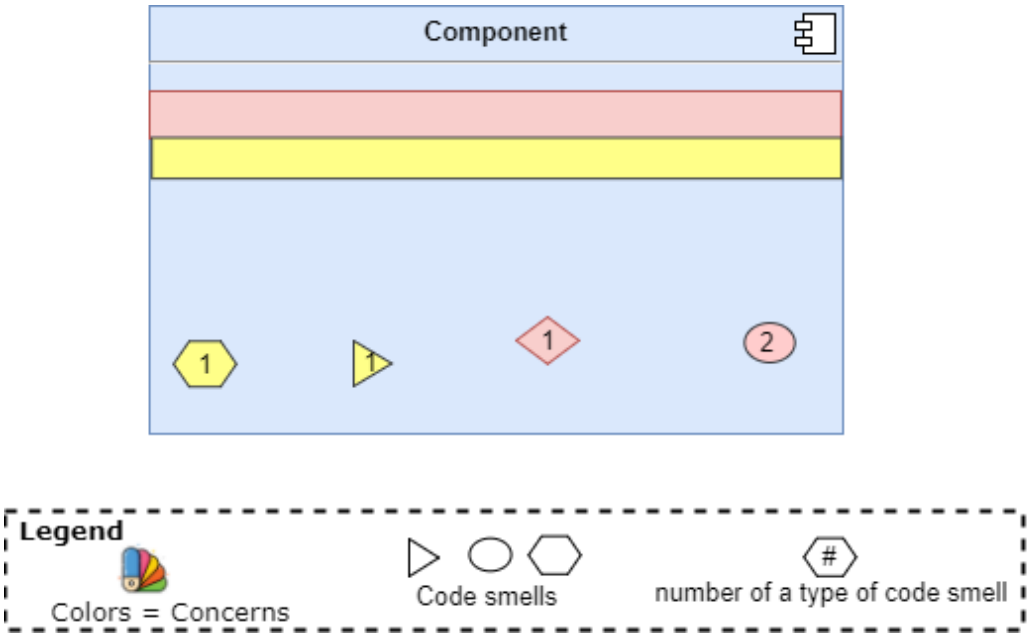

Figure 4.1: Component View

As we can observe in Figure 4.1, the component implements three different concerns (sky blue, red and yellow), where the dominant concern is the color of the component. The dominant concern is the concern that is realized by the majority of the inner elements of the component. In the figure, the dominant concern is the sky blue; this the reason why the component color is sky blue. In addition, the height of the rectangle representing each concern means how much of the component implements the concern. This will help to know the proportion of component elements, if a few or many, address a concern. Each shape in the component represents a different type of code smell (e.g., God Class, Feature Envy). The number within each shape represents the number of smell instances (of a particular smell type) found in the agglomeration. The color of the shape means that a code smell is located in the implementation of the concern with the same color. The length of the shapes (code smells) is not yet mapping a particular characteristic of the smell; by now, it is merely illustrative, but we will determine which characteristic the length should represent (if needed) along our research experiments.

The **Class view** (Figure 4.2) shows the classes of a program affected by an agglomeration and the dependencies between classes. The arrows represent the dependencies. For each class of the agglomeration, we represent the concerns that the class implements and the code smells affecting the class, using a similar representation of the component view. In this view, it is possible to observe how many classes are contributing to the implementation of a concern as well as how many concerns are affecting a single class. Through the view, we observe that there is an explicit relationship between the classes because Classname1 and Classname3 use Classname2 resources. Also, we can see that the three classes of the component, besides implementing the main concern (blue color), are implementing additional concerns (pink and yellow). On the other hand, some of these implementations are associated with code smells. For instance, both Classename1 and Classname3 implement the pink concern, whose implementation is affected by different code smell types (each smell type is represented by a different shape). This may indicate that this concern has been spread and should be moved from these classes to another one whose main concern is the pink one. Therefore, the class view can be used to help identify the Scattered Concern design problem.

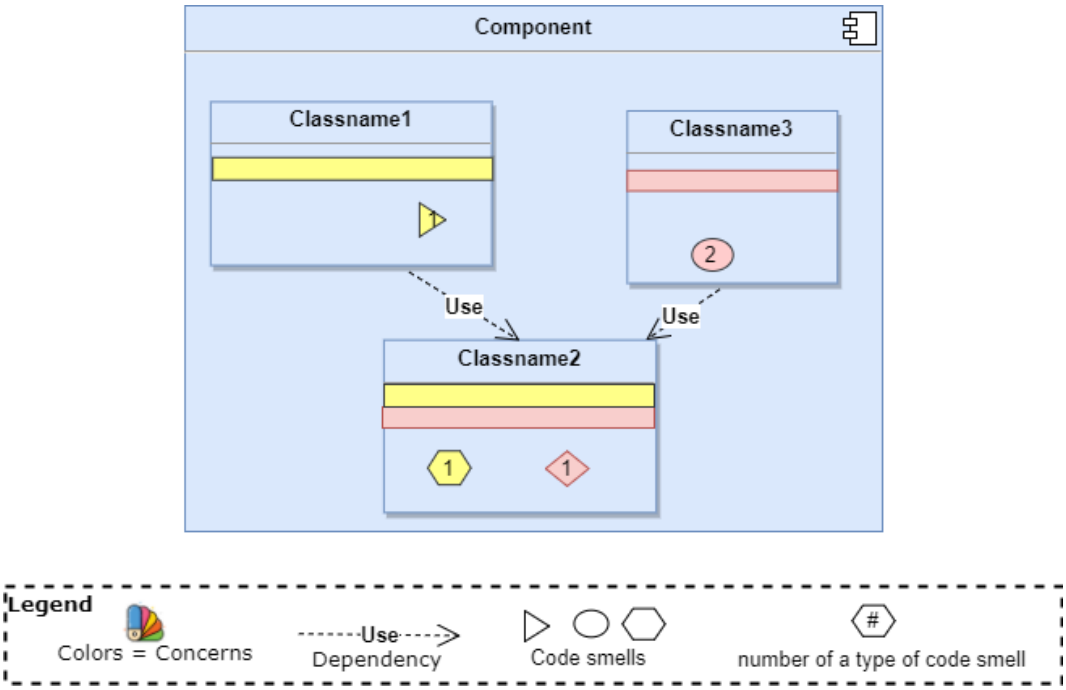

Figure 4.2: Class View

The **Method view**, as shown in Figure 4.3, provides more detailed information about the agglomeration by presenting the code smells associated with each method. The representation of the methods helps the developer to know the fragment of code where the code smell is located. Differently from the

component view, we only represent the dominant concern of each method; i.e., a method can, in principle, contribute to the implementation of more than one concern. However, only the dominant concern of the method is represented. This is a limitation imposed by the existing techniques for concern mining (59).

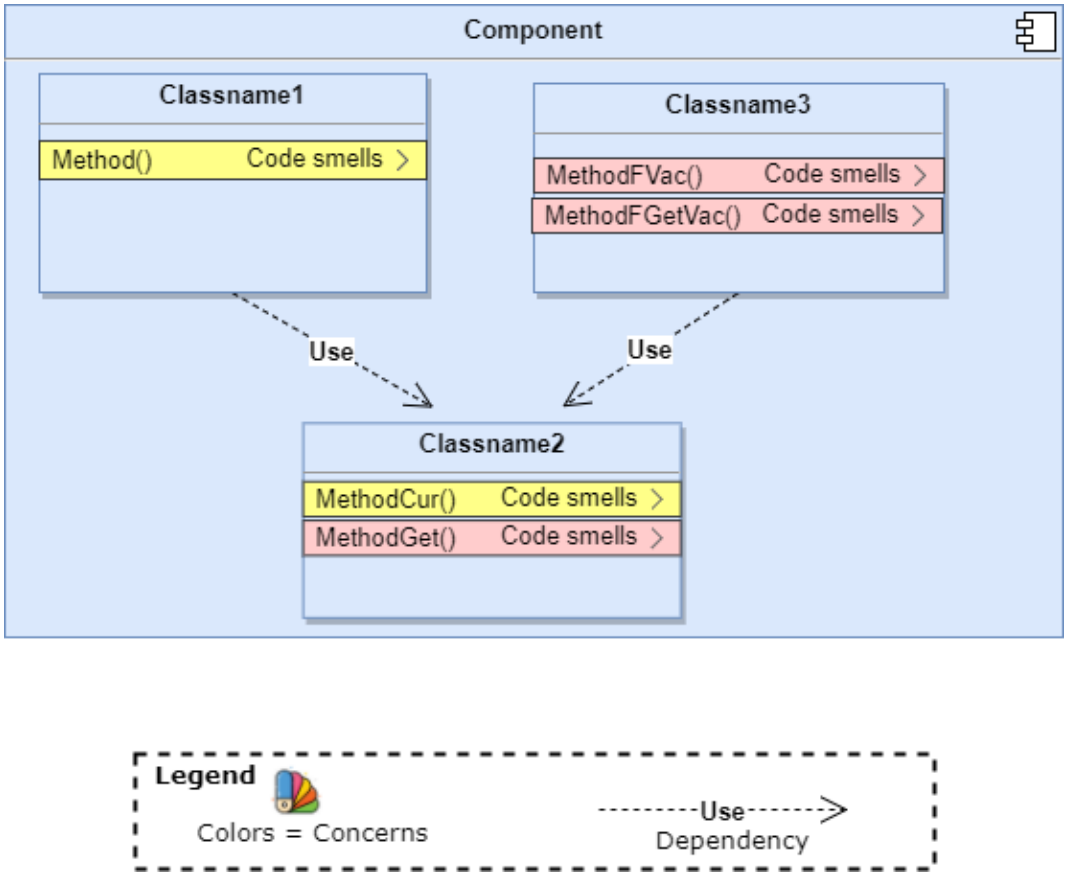

Figure 4.3: Method View

4.3 Support Tool

This section provides an overview of VISADEP tool: a 2D prototype tool intended to graphically represent code smell agglomerations. VISADEP was designed to help programmers in identifying design problems in the source code of Java (53) systems. The tool is implemented as a plug-in for the Eclipse platform (31) with a multi-view visualization (Section 4.2).

Our tool receives as input a list of agglomerations detected in the program. To fulfill its role, the tool first uses the Organic (29) tool to collect the code smell agglomerations in a program. From the list of agglomerations, VISADEP main's goal is to provide a graphic visualization of agglomerations in three different views (component, class and method views). In this version of the tool, we implemented the visualization for the three types of agglom-

eration investigated in this dissertation: concern-based agglomeration, intra-component agglomeration, and hierarchic agglomeration. Next, we present more details about the tool:

Agglomerations View. Figure 4.4 shows a snapshot of the Agglomerations View. As it can be observed, this view is separated into 3 parts: the first part (snapshot A) is called "List of Code Smell Agglomerations by Types for each Project" on the left panel; the second part (snapshot B) contains the tabs; and the third part (snapshot C) displays information about an agglomeration according to the tab selected. In snapshot A, we have a tree-based visualization,

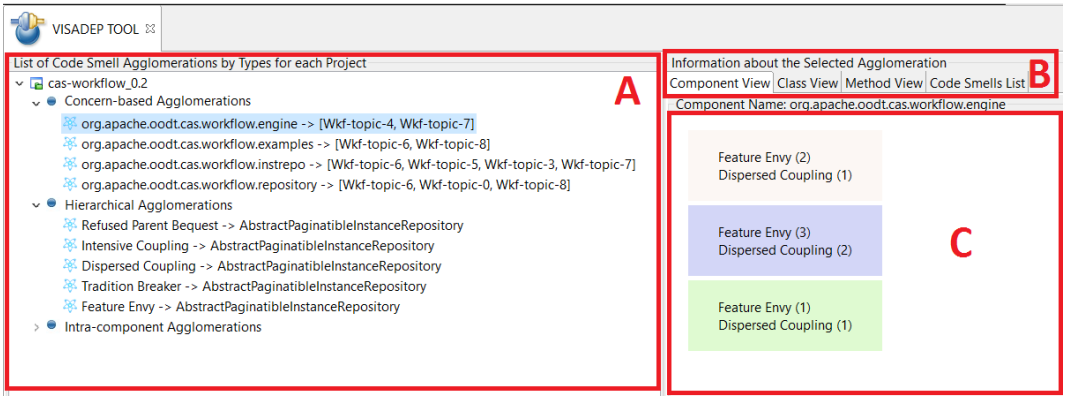

Figure 4.4: VISADEP Agglomeration View in Eclipse IDE

which shows all the agglomerations detected in a program, grouping them according to their category. For instance, in Figure 4.4, the view shows all the agglomerations found in the *cas-workflow-0.2* system. When clicking on one of the agglomeration categories, all the detected agglomerations of that category are displayed. At the right panel, the representation of agglomerations is shown in the tabs Component View, Class View, and Method View.

Component View Tab. Figure 4.5 shows a snapshot of the representation of a selected agglomeration, a concern-based agglomeration in the org.apache.oodt.cas.workflow.engine package at the component level. As shown in this figure, we have a component or a package system with its name. Inside the component, we have rectangles with colors which represent the concerns that the component implements. Within each rectangle, we represent those forms in the mockups by the name of each type of code smell that is related to the concern. By moving the mouse over the representation of a concern, a *Tooltip* appears with the name of the concern represented. The number in parenthesis side to side with the kind of code smell represents the number of instances found for this kind of smell.

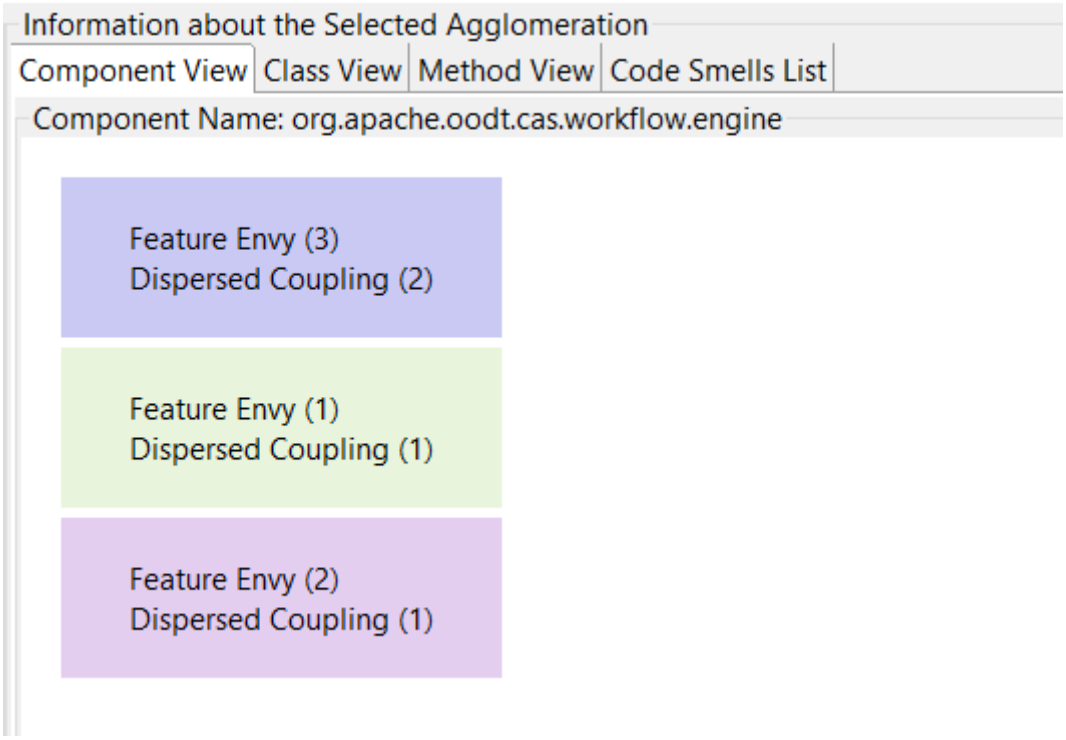

Figure 4.5: Component View in Eclipse IDE

Class View Tab. Figure 4.6 shows a snapshot of the representation of a selected agglomeration at the class level. The tool represents with UML boxes the classes within the component that are affected by the selected agglomeration. As explained in the mockups, the class view represents the concerns that each class implements, the dependencies between the classes of the agglomeration and the smells of the agglomeration that affect each class. The number in parenthesis represents the number of times this kind of smell has been found. One limitation of ConcernMapper is that the information it provides is not enough to implement the mockups at class and method levels. Because in the mockups, the objective was to show the concerns associated with each code smell, but in the implementation of the VISADEP tool it was not possible as ConcernMapper does not give this information with this granularity, only at the class level. Thus, we can not know at the class level the code snippets that implement a concern to associate it with the code smells. As future work, we plan to assess the tool to know if developer requires other dependencies that contain smells and are not part of the agglomeration.

Method View Tab. Figure 4.7 shows a snapshot of the representation of a selected agglomeration at the method level. This view provides more details about the methods affected by smells in each agglomeration. For each code smell detected we show the name of the method affected if it is the case. When the code smell is at the class level (i.e., God Class), we did not show any

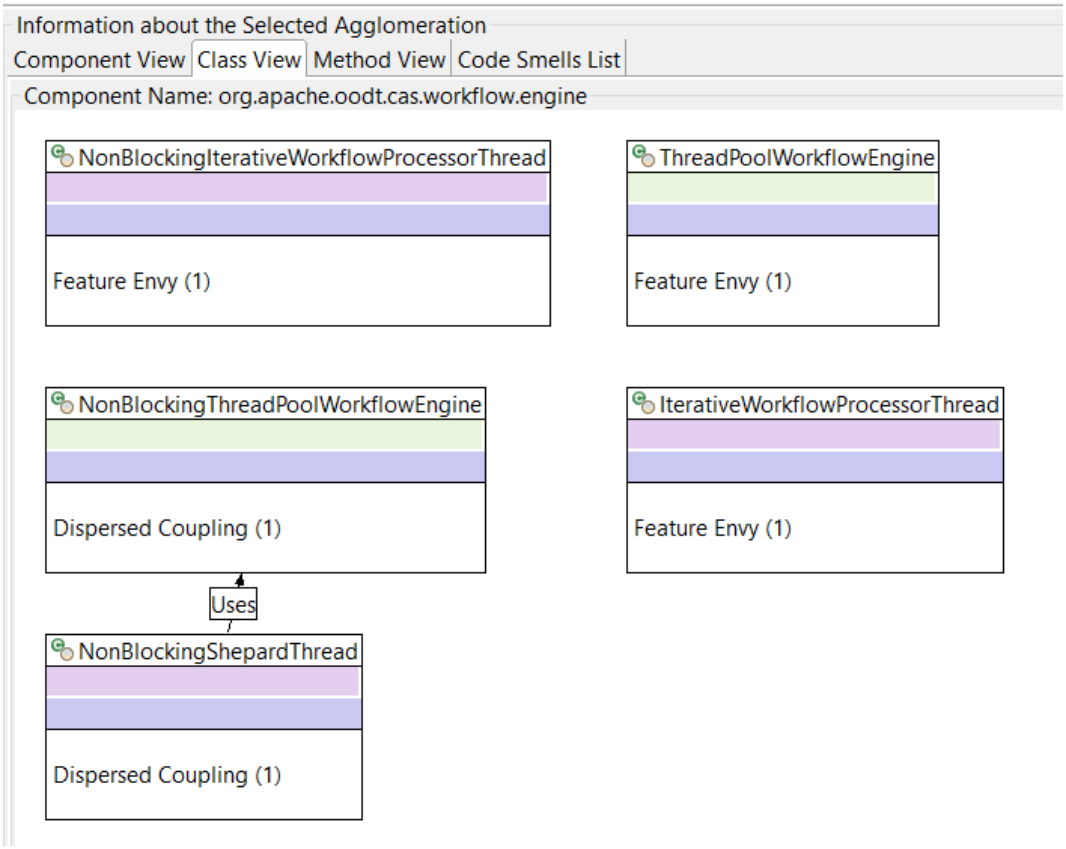

Figure 4.6: Class View in Eclipse IDE

method name.

Technical Details. The table 4.1 below shows the technologies that were used to implement the VISADEP tool. The tool is available for download in our study companion website (58).

4.4 Summary

In this chapter, we presented the mockups and the implementation of our visualization approach which is the visualization of agglomerations across components, classes, and methods. We also described the VISADEP, a prototype tool, to support our approach to visualization. Our proposal visualization goal is to provide developers with a visualization of agglomerations in order to support their understanding of how code smells composing an agglomeration are interrelated in the software decomposition, thereby facilitating the identification of a design problem. However, a study has to be conducted to evaluate whether indeed the visualization supports the identification of design problems. Thus, after presenting our proposal visualization, the next chapter describes a study that aims to assess our novel visualization approach by comparing it with an improved graph-based visualization. Our goal is to inves-

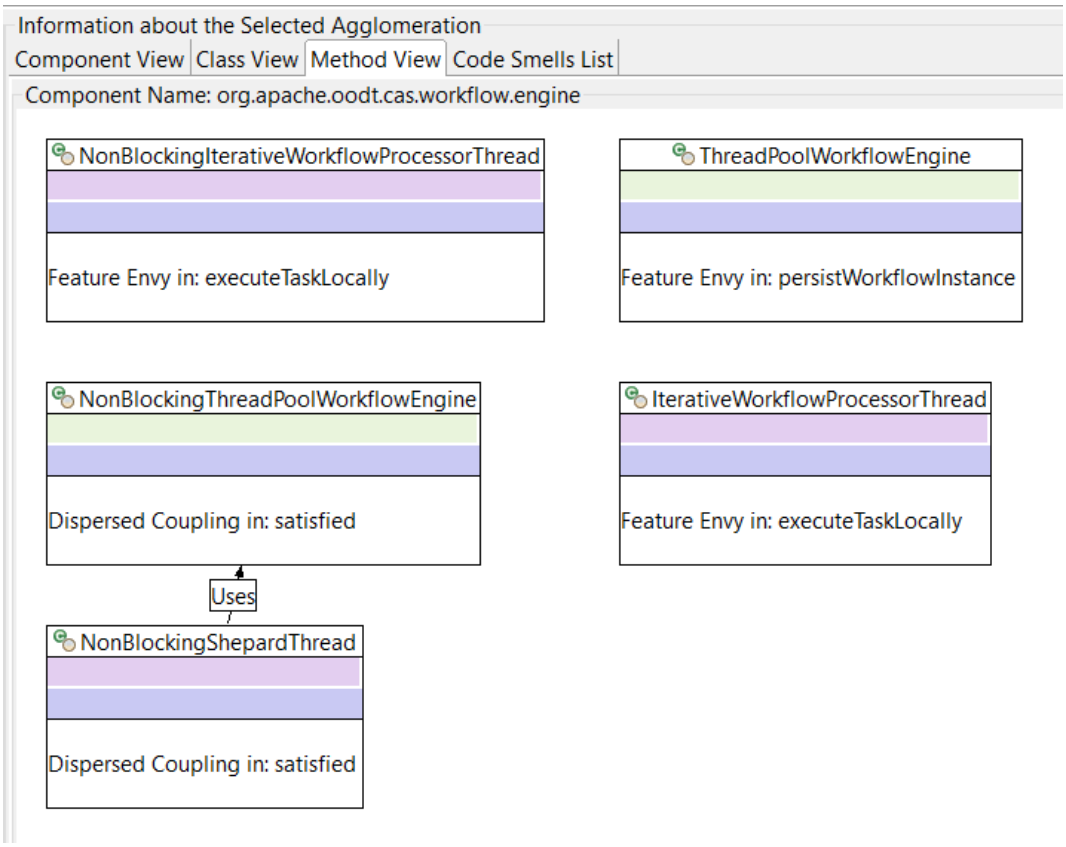

Figure 4.7: Method View in Eclipse IDE

tigate whether both visualizations can support developers in the identification of design problems.

Table 4.1: Technical Details of VISADEP

Tool Name	**Purpose**
ConcernMapper	The user of VISADEP needs an input of the concerns to be able to see concern representation in VISADEP. Thus, ConcernMapper was used to extract concerns implemented by a system.
Eclipse Luna	We used the Eclipse Luna as environment to create our tool. The actual version of VISADEP works only with Eclipse Luna
Java 7	The code of VISADEP were implemented using Java 7
SWT	The Standard Widget Toolkit (SWT) is a standard UI library used by Eclipse. The widgets provide for example, buttons and text fields, as well as layout managers.
Draw2D	Draw2d is a lightweight toolkit set for displaying graphics components on an SWT screen. Lightweight means that all graphic components, which are called figures in Draw2d. We used this technology to represent graphic such as the figure of component, classes and dependencies, and concerns.
JFace	JFace provides several standard viewer implementations. We used Jface in our tool for the interface of the Agglomerations view.

5
Assessing the VISADEP Approach: Second Study

In order to support the identification of design problems, previous studies show that code smell agglomerations, i.e., code smells that somehow interrelate in the system, can be used as indicators of design problems (29) (12). However, a single agglomeration may contain several code smells in multiple code elements (10, 12). Consequently, developers might find difficult to identify design problems with agglomerations. Thus, a need emerges for proposing mechanisms which summarize data about code smell agglomerations for displaying it to the developer sufficiently and effectively. In this context, visualization of agglomerations potentially could help developers in identifying design problems. To the best of our knowledge of the literature, none of the previous work focuses on the visualization of smell agglomerations for the purpose of supporting developers to identify design problems.

In the previous chapter, we presented the VISADEP approach and in this chapter, we describe a study that aims to assess our visualization approach. Section 5.1 describes a controlled experiment whose purpose is to investigate and compare the accuracy of the VISADEP approach and the graph-based approach for identifying design problems. Section 5.2 presents and discusses the experiment results. Finally, Section 5.3 describes the threats that could limit the validity of the study.

5.1
Study Protocol

This section describes our study protocol. Section 5.1.1 introduces our study goal and research questions. Section 5.1.2 provides the description of all artifacts used in this study. Section 5.1.3 contains information about the procedures we followed to select subjects for our experiment. Finally, Section 5.1.4 presents details about the experiment procedures.

5.1.1
Goal and Research Questions

We conducted a controlled experiment aimed at investigating the accuracy of the two visualization approaches (Graph-based and VISADEP) for

identifying design problems. The first approach is a new visualization called VISADEP. The second approach is a conventional visualization that uses graphs for representing agglomerations (Chapter 3). We did not compare with existing agglomeration approaches (Organic and JSpIRIT) because the visual elements presented in these approaches are different from those shown in our visualization approach. The reason is that our approach shows smell dependencies, concerns implementation and program elements (classes, interfaces, and methods). The existing approaches are very different from our proposal, so the comparison would not be fair. That is the reason why we have refined the graph to have at least the minimum of information to compare with VISADEP.

As previously mentioned, VISADEP relies on different views for representing agglomerations at three abstraction levels of a software system: component, class, and method. As a matter of fact, previous research shows that the analysis of multiple code smells is not a trivial task and, then, may require the use of multiple views (29). In the graph approach study, we also noted the need of different views for the agglomeration visualization to assist developers in understanding design problems. Hence, by providing multiple views, we expect that VISADEP will help developers to perform more precise analysis.

The graph-based visualization was chosen to be the baseline of our study. We performed this comparison to see whether (and to what extent) VISADEP led to improvements with respect to our first approach (graph-based visualization), since the second one also had the objective to fill deficiencies observed in the first approach.

In order to conduct the comparison between VISADEP and the graph-based visualization, we have defined a research goal, which is presented according to the Goal-Question-Metric (GQM) template (57) as follows: *analyze* two visualization approaches of code smell agglomerations, namely VISADEP and a graph-based visualization; *for the purpose of* investigating the accuracy of the approaches; *with respect to* accuracy of identification of design problems; *from the viewpoint of* researchers. *The context* comprises professional software developers and computer science students from different Brazilian education institutions (Section 5.1.3), two subsystems (Section 5.1.2) of the Apache OODT system (Workflow and PushPull), and a reference list of design problems (Section 5.1.2) identified in both subsystems.

We designed two research questions (RQs) aimed at reaching our research goal. Next, we present and describe each RQ.

RQ1. *How accurate is the identification of design problems using each visualization approach?*

To answer the first research question, we assessed the accuracy of subjects when using each approach to identify design problems. For this assessment, we relied on two metrics: precision and recall. Similarly to the accuracy measurement performed in the first study (Chapter 3), precision refers to the percentage of correctly identified design problems. Recall is the percentage of identified design problems with respect to all existing design problems in a given system. In this study, we are interested in the accuracy of the VISADEP approach compared to the graph approach. Additionally, we conducted a qualitative study intended to: (i) point out opportunities for improvement in the graph approach, and (ii) derive insights that led to the proposition of the VISADEP approach.

RQ2. *How accurate are the visualization approaches in identifying each type of design problem?*

To answer the second research question, we assessed the accuracy of subjects to identify each type of design problem. This means that, for RQ2, we conducted an individual analysis for each type of design problem investigated in this study: Scattered Concern, Concern Overload, Ambiguous Interface, and Fat Interface. The assessment was also based on precision and recall metrics. This research question is important because the identification and analysis of each type of design problem may benefit from different visualization approaches. Therefore, the result of RQ2 may help developers to select the best approach for a given context.

Besides answering the aforementioned research questions, we also conducted a complementary qualitative analysis. For this purpose, we asked subjects to report the usefulness of each visual element based on a five-level Likert scale as follows: -2 (strongly disagree), -1 (disagree), 0 (undecided), 1 (agree), and 2 (strongly agree). To understand what should be improved in both approaches, we also analyzed the explicit suggestions made by subjects via their textual answers given in the experiment forms. We identified the most recurring suggestions, and also suggestions that were not frequent but were considered as being sound suggestions. The conduction of a qualitative analysis is important to help us in understanding and possibly explaining the quantitative results of our research questions.

5.1.2 Instrumentation

To conduct this study, we used the following artifacts (for more details, see our study companion website (58)): (a) characterization form, (b) consent form, (c) design problem identification form, (d) post-experiment form, (e)

Apache OODT System, (f) reference list of design problems, (g) Eclipse IDE (Integrated Development Environment), (h) VISADEP (Figure 4.4) tool, (i) the graph-based approach (Figure 5.1), and (j) Camtasia Studio[1]. We have decided to compare VISADEP with the graph-based approach only due to the following reason. None of the existing visualizations for smell agglomerations attempts to visualize smell agglomerations (29, 30) represent a sufficient number of visual elements that are similar to our approach. Thus, comparing them with out tool would difficult drawing conclusions about what information represented by each approach actually affects the identification of code smells either for the better or for the worse.

The *characterization form* aimed at collecting the subject background and working experience. The *consent form* was used to get authorization from the subjects to use their data in this study. The *design problem identification form* aimed at collecting data about the identified design problems from the analysis of smell agglomerations. The *post-experiment form* aimed at capturing feedback from the subjects after the experiment.

Apache OODT is the system we provided to participants for the task of identifying design problems. Since OODT is a large heterogeneous system (61), we selected only two subsystems for the experiment: Push Pull and Workflow Manager. A brief description of the subsystems is presented as follow. Push Pull is the OODT component responsible for downloading remote content (pull) or accepting the delivery of remote content (push) to a local staging area. Workflow Manager is a component that is part of the OODT client-server system. It is responsible for describing, executing, and monitoring workflows. Table 5.1 shows an overview of the number of classes and size (in lines of code) of each subsystem.

Table 5.1: Characteristics of Subsystems

Subsystem	Number of Classes	Size (LOC)
Push Pull	90	6526
Workflow Manager	76	8819

Reference List of Design Problems. The reference list of design problems is composed by Ambiguous Interface, Fat Interface, Scattered Concern, and Concern Overload. This reference list was different at the beginning. It became this way after analyzing the results of the experiment. Before the execution of the experiment, there was an OODT primary reference list that was extracted from previous works (3, 10, 29, 32, 39) composed by Scattered Concern and Ambiguous Interface. After the experiment, we included two

[1]Camtasia Studio. Available at https://www.techsmith.com/camtasia.html

design problems. The following is the criterion of inclusion of design problems: (i) at least 3 participants said they encountered the design problem in the same agglomeration independently of the approach used; and (ii) after manual review of the source code by two consultants, we conclude that the design problem reported by the participants is missing from the primary reference list. Therefore, during data analysis, when at least three participants determined that there was a design problem in a particular agglomeration and this problem was not in the reference list, we manually analyzed the source code and used the VISADEP tool in order to identify this problem and validate it. When we got to validate, that problem was included in the reference list.

Eclipse IDE. The Eclipse IDE was provided for the subjects to navigate in the source code of Apache OODT. VISADEP tool and Graph-based tool are the computational tools that support the approaches we are assessing in this study. Finally, Camtasia Studio was used during the experiment to record the computer screen and audio of each subject for posterior analysis.

Enhanced Graph Visualization. Based on the results of evaluating the graph-based visualization (Section 3.3), we observed that the proposed graph representation did not provide the necessary information to identify design problems related to the implementation of concerns. Thus, in order to compare our new approach with the graph-based visualization, we decided in order to make some adjustments to the initial graph representation to make a fair comparison. The purpose was to perform a more direct comparison between the VISADEP visualization and a graph-based visualization. In other words, both approaches would provide the same types of information, while differing in the way the information was visually presented. The graph was enhanced from different aspects, such as an improved way to graphically represent concerns; and the addition of textual information about the concerns implemented by each class. Figure 5.1 presents the enhanced graph representation.

The visualization uses different lines and arrows to represent the four types of relationships among the program elements as shown in Figure 5.2. A straight line in blue represents association, i.e., the classes (connected by straight line) depends among themselves. A dotted line in orange represents concern implementation, i.e., the two classes (connected by dotted line) share the implementation of (at least) one concern. For the sake of simplicity, differently from the VISADEP approach, we decided to not distinguish the concerns with different colors. A dotted arrow in blue represents interface implementation while a straight arrow indicates an inheritance. The direction of the arrow indicates the direction of the relation.

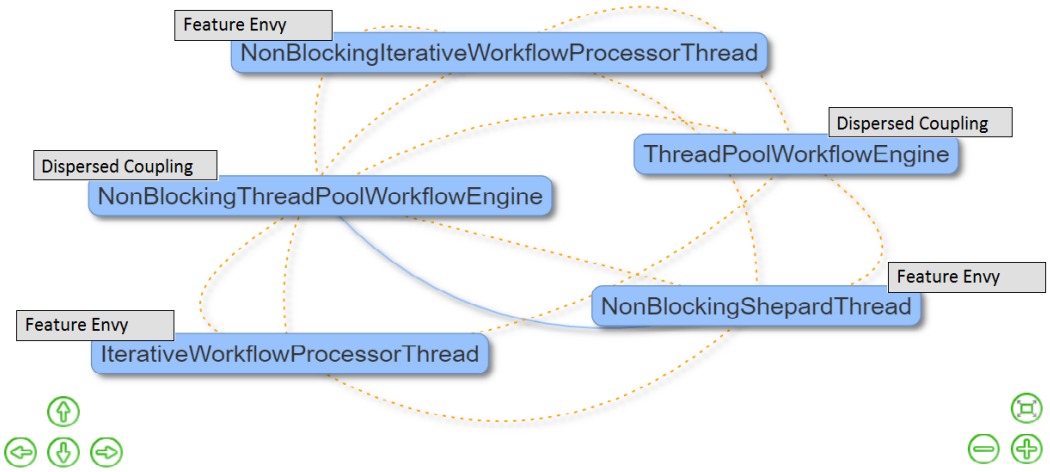

Figure 5.1: Enhanced Graph-based Visualization

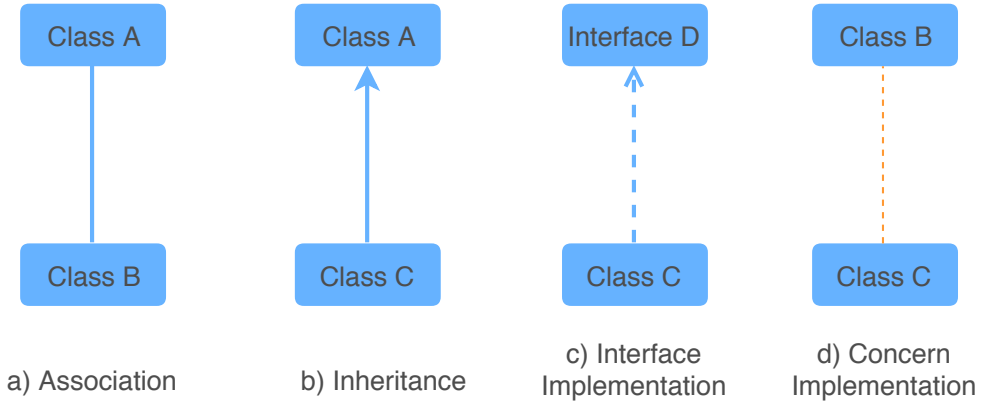

Figure 5.2: Graph Dependencies

In order to show how the two tools are comparable, we present in the Table 5.2, the information of the two visualizations and how they are represented in each visualization.

5.1.3 Subject Selection and Cross over Design

In order to conduct this experiment, we recruited professional software developers and computer science students from our professional network. For a subject to be chosen as a participant, they would need to meet at least the following requirements:

- Intermediary knowledge about the Eclipse IDE.
- Basic knowledge about software architecture.
- Basic knowledge of the Java programming language.
- Basic knowledge of object-oriented software design principles, such as cohesion and coupling.

Table 5.2: Information of Approaches being Compared

Information	VISADEP visualization	Enhanced Graph
Program elements	- Rectangle with the name of the component - UML boxes representing classes or interfaces - Textual description of the name of the methods	Graph nodes with the name of the elements
Concern	Colored rectangles in elements contributing to the concern implementation	Dotted lines (in orange) connecting elements implementing the same concern
Dependency	- Arrow with the text Uses, connecting classes or classes and interfaces - Arrow with the text Inherits, connecting elements of a hierarchic agglomeration	- Straight lines (in blue) representing Calls, Uses, and references - Normal arrows (in blue) for inheritance dependency - Dashed arrows (in blue) for implementation of an interface
Code smell	Explicit textual description of the name of code smell in each program element representation	The name of the code smell appears on mouse over the node (element). A textual description is obtained after clicking on the smell name

– Basic knowledge about the definition of code smell.

We defined the knowledge in each topic based on a scale composed of five levels: *none*, *minimum*, *basic*, *intermediary*, *advanced* and *expert*. To measure each requirement, we asked participants to fill out an online form. We included in the characterization form a description of each knowledge level, allowing all subjects to have a similar interpretation of the answers. The characterization of all subjects selected for this experiment is presented in Table 5.3.

Table 5.3: Characterization of Subjects

Subject	Experience in Years	Education Level	Category
P1	6	B.Sc.	Student
P2	8	M.Sc.	Professional
P3	7	M.Sc.	Professional
P4	3	Other	Student
P5	2	B.Sc.	Student
P6	3	B.Sc.	Professional
P7	0	B.Sc.	Student
P8	6	M.Sc.	Student
P9	3	B.Sc.	Student
P10	4	B.Sc.	Student
P11	2	B.Sc.	Student
P12	7	M.Sc.	Professional
P13	4	B.Sc.	Professional
P14	7	M.Sc.	Professional
P15	9	M.Sc.	Professional

To reduce the bias of experience and knowledge, subjects were divided into four groups. The groups were formed based on the characterization of subjects. Each group includes subjects with different characteristics. Table 5.4 presents the cross design for the four groups. The first column of the table

Table 5.4: Experiment Cross Design

	Subject		First Step		Second Step	
Group	**ID**	**Location**	**Approach**	**Project**	**Approach**	**Project**
1	P1 P5 P6 P7 P10 P11	Industry Academy Industry Academy Academy Academy	Graph	Workflow Manager	VISADEP	Pushpull
2	P13 P2 P14	Academy Industry Academy	Graph	PushPull	VISADEP	Worflow Manager
3	P15 P3 P4	Academy Industry Academy	VISADEP	PushPull	Graph	Workflow Manager
4	P8 P9 P12	Industry Academy Industry	VISADEP	Workflow Manager	Graph	PushPull

shows the four groups. The second column informs the distribution of subjects in each group. The third column indicates subject location (industry for professional and academy for student). Each group used the visualization approach (i.e. VISADEP and Graph-based) in a different order as shows in fourth and sixth columns of Table 5.4. In addition, each group received a different combination of projects (fifth and seventh columns) and approaches. For instance, subjects of Group 1 use graph-based to identify design problem in Workflow Manager first. Next, they used the VISADEP approach to identify design problems in PushPull. In this way, we could mitigate the influence of variables, such as knowledge, fatigue, and complexity.

5.1.4 Experiment Procedures

The execution of this experiment occurred in two weeks. During the first week, we characterized and selected the subjects. The experiment occurred in two days of the second week, being each day dedicated to a specific phase of the experiment. In each phase, subjects performed a set of activities on a given component, using a visualization approach (VISADEP or Graph-based). This helped us to avoid fatigue and to have full dedication and feedback from the subjects. Below we present a detailed description of the activities that occurred during the experiment.

Selection of subjects. One week before the experiment, we sent a questionnaire of characterization to potential participants. Based on this

questionnaire we followed the procedures described in Section 3.2.2 to select subjects for our experiment.

Training about core concepts. In the first day of second week, we conducted training about concepts related to the identification of design problems with agglomerations. The training was organized in 2 parts: a presentation (20 minutes), and an open discussion with the subjects (10 minutes). The presentation addressed concepts associated with code smells, smell agglomerations, and design problems. During the presentation, we introduced examples of design problems that could be found by the subjects during the experiment. On the second day of the second week, we conducted quick training (5 minutes) to help subjects remembering the concepts. Even after this training, we provided to the subjects a document summarizing the core concepts. This document was available during the whole experiment, so they could use it whenever necessary.

Training about the visualization approach and the subsystem. In each day of second week, we conducted brief training about the visualization approach (VISADEP or graph-based) and the subsystem (PushPull or Workflow Manager) used by the subject, according to the cross design (Table 5.4) of our experiment. During this activity, we explained the main characteristics of the visualization approach, such as the metaphors used to represent an agglomeration. We also provided the subjects with the documentation and the source code of the subsystem that would be analyzed. This activity helped subjects to understand the subsystem and to identify the main source code elements.

Identification of design problems. In the main activity of this experiment, we asked subjects to use a visualization approach (VISADEP or graph-based) in order to identify design problems in a given OODT subsystem (PushPull or Workflow Manager). For this activity, we provided up to 5 instances of agglomeration to be analyzed; an equal number of agglomeration instances was provided for both visualization approaches. During this activity, we used Camtasia to record the computer screen and audio of each subject. We also used the Rabbit plugin to track all interactions in Eclipse IDE. We asked subjects to fill an experiment form providing the following information for each agglomeration: (1) detailed description of each design problem found in the agglomeration (if any), (2) code elements realizing the design problems in the source code, (3) the visualization elements (e.g. concerns, code smells and dependencies) that were useful to identify one or more design problems.

Post-experiment Questionnaire. After the identification task, each subject completed a questionnaire about using the visualization approach. In this questionnaire, we asked subjects to provide feedback that would be helpful

to improve the visualization approach. In addition, we asked subjects to report the usefulness of each visual element, according to their perception.

5.2 Results and Discussion

RQ1. *How accurate is the identification of design problems using each visualization approach?*

Table 5.5 presents the precision and recall rates computed for both approaches. The first column characterizes the table results per OODT subsystem (i.e., Workflow Manager, PushPull, or both). The second column lists the assessed approaches, namely graph-based and VISADEP. The third, fourth, and fifth columns present the absolute values of TP, FP, and FN. The two last columns present the precision and recall rates, respectively. We discuss the main finding and implications as follows.

Table 5.5: Overall Precision and Recall

Subsystem	Approach	TP	FP	FN	Precision	Recall
Workflow	Graph	15	11	62	57.69%	19.48%
	VISADEP	19	8	69	**70.37%**	**21.59%**
PushPull	Graph	18	13	70	58.06%	20.45%
	VISADEP	17	9	60	**65.38%**	**22.08%**
Both	Graph	33	24	132	57.89%	20.00%
	VISADEP	36	17	129	**67.92%**	**21.82%**

About precision and recall of the visualization approaches. Our study results suggest that VISADEP provides a better accuracy in the identification of design problems when compared to the graph-based approach. In fact, when considering both subsystems (see the two last rows of Table 5.5), our novel approach has obtained a precision rate of 67.92%, which is around 10% greater than precision of the enhanced graph-based approach. A similar result is observed for recall: VISADEP has reached a recall rate of 21.82, which is 1.82% greater than recall reached by graph-based but still low for the purpose of identifying design problems. Thus, we highlight that VISADEP had performed better than the graph-based approach regardless the assessed subsystem. We noticed that the precision and recall measures were, in general, higher (Section 3.3) in our first study (of the graph-based approach). The main reason was that the developers had already extensive knowledge of the system being analyzed, while they did not have this previous knowledge in this experiment

We already expected a low recall rate for our visualization approach. This is because our experiment design included the identification of several design problems per instance of smell agglomeration. For instance, in the case of the Instrepo component (collected from the Workflow Manager module), it is affected by a smell agglomeration also affected by three design problems (namely, Scattered Concern, Ambiguous Interface, and Concern Overload). In other words, our reference list of design problems contains a high number of design problems, which hinders for developers to reach a high recall rate given the time constraints of the experiment. On the other hand, we kept the analysis of recall in order to understand to what extent each visualization approach has allowed the identification of design problems in each Apache OODT module selected for the developers to analyze.

RQ2. *How accurate are the visualization approaches in identifying each type of design problem?*

Table 5.6 presents the precision and recall rates computed for both approaches. The first column characterizes the table results per OODT subsystem (i.e., Workflow, PushPull or both). The second column lists the assessed approaches, namely graph-based and VISADEP. The third, fourth, and fifth columns present the absolute values of TP, FP, and FN. The two last columns present the precision and recall rates, respectively. We discuss the main finding and implications as follows.

Table 5.6: Precision and Recall per Type of Design Problem

Design Problem	**Approach**	**TP**	**FP**	**FN**	**Precision**	**Recall**
Concern Overload	Graph	12	10	33	54.55%	26.67%
	VISADEP	22	10	23	**68.75%**	**48.89%**
Scattered Concern	Graph	18	17	42	**51.43%**	**30.00%**
	VISADEP	10	13	50	43.48%	16.67%
Ambiguous Interface	Graph	1	21	51	4.55%	**1**.92%
	VISADEP	3	12	50	**20%**	**5.66%**
Fat Interface	Graph	2	3	6	**40%**	**25%**
	VISADEP	1	5	7	16.67%	12.50%

Types of design problem that VISADEP helps identify at most. We analyzed what visualization approach (VISADEP or the graph-based approach) has provided the best precision and recall rates. We have obtained the following results as shown in Table 5.6:

- VISADEP has helped identify Concern Overload more than the graph-based approach. By the accuracy and recall results for Concern Overload, we can see that the VISADEP was in precision 14% better than the graph and much better in recall (22%).
- For Scattered Concern, the graph-based was slightly higher than VISADEP. However, the difference (8%) in precision was not significant to discourage the use of VISADEP to identify Scattered Concern.
- The graph-based approach has performed better in supporting the identification of Ambiguous Interface. VISADEP had 15% more precision and only 4% more recall. But the result was low in both approaches. Regardless of the difference, no approach is recommended for the Ambiguous Interface but VISADEP had a significant improvement (15%) in precision. We highlight that this result might be a consequence of the fact that Ambiguous Interface often requires to analyze the source code of a software system and, therefore, the visualization approach might not have hindered the identification of this particular design problem.
- The graph-based approach has helped identify Fat Interface more than VISADEP, but both approaches performed poorly. For Fat Interface in precision, the graph-based was more than 20% better than VISADEP in precision. A 40% precision obtained in the graph is already a reasonable value but the recall was low in both approaches, but the graph-based improved 13% than the VISADEP.

One possible explanation for the fact that Scattered Concern was better with the graph-based than with the VISADEP is that in the case of a Scattered Concern, in principle, it is enough to check if there is a Scattered Concern, that is, if several elements share the implementation of an interest, which can be checked in the improved representation of graphs quickly. However, we can say that the developer has difficulty saying what is the name of the concern that is spread, because the graph-based approach does not distinguish between different concerns. In the case of Concern Overload, it is even more important to have a differentiated representation for each concern (as in VISADEP and, in fact, VISADEP had better results) because it is not enough to know how many concerns occur in the component, but rather the cohesion / proximity "semantics" of the different concerns affecting the same component. There may be cases where the component implements three or more concerns, plus all of them are cohesive concerns and/or part of a more general concern; that is, they are not concerns whose implementation should be moved to another component.

Regarding the interface problems (Fat Interface, Ambiguous Interface), they were not as satisfactory, although the graph-based was superior in the Fat Interface and VISADEP in the Ambiguous Interface. However, we noticed that the approaches of visualization did not explicitly represent the information of the smells, the relationships, and concerns at the interface level. As it stands today, for the developer to identify the Fat Interface, he needs to analyze the concerns of the smells around the interface that is in the classes that implement the interface, and in the client classes. For the identification of Ambiguous Interface, the developer has to look at the classes that implement the interface, the classes that use the interface to infer whether the interface has design problem.

The results of the precision and recall per design problem indicate that VISADEP achieved a better result for Concern Overload than the graph-based. VISADEP can already be used in practice. So if the developers are interested in problems associated with large amounts of concern implemented by the modules, VISADEP is recommended. In the case of Scattered Concern, you can use VISADEP; however, the graph-based was better than VISADEP. In the case of Ambiguous Interface and Fat Interface, many studies still have to be done to investigate how to better represent the interfaces.

Visual representation that each design problem benefits at most. We have crossed our study results about the quality of the representation of each visual element provided by the approaches (Table 5.7) with the ones about the relevance of each element (provided by the Experiment Form). From that, we have drawn some interesting findings as follows. First, even though concerns were the improved representation, the representation quality of code smells provided by VISADEP have decreased in 17% when compared to the graph-based approach. The utilization of dependencies representation decreased 27% to identify design problems. Second, the elements Zoom-In and Zoom-Out were useful to only 25% of participants. However, the VISADEP different views obtained among 34% and 39% of utilization.

About the visual representation of the approaches. Table 5.7 presents the developers' perception about the quality of representation per program element in both approaches (Graph-based and VISADEP). While analyzing the behavior of the participants along the experiment, we tried to observe the general strengths and weaknesses of the VISADEP approach. On the representation quality of the visuals elements of each approach, in relation to relevant data for to identify design problems (e.g., concerns, code smells,

and dependencies), and we noticed that VISADEP:

(i) improved the representation of concerns in relation to graphs. In the fourth line of Table 5.7, we can see that VISADEP won over the graph in the representation of concerns. 60% of participants agreed that concerns are well represented versus 40% in the graph. In addition, 33% of answers were neutral for VISADEP versus 20% in the graph. Most importantly only 7% did not agree with the representation regarding the high number (40%) in the graph.

(ii) equipped the code smell representation in relation to graphs, and Through the table, the third line of the table shows that both the graph and the VISADEP have had a quality degree in the representation of code smells. For example, 73% agreed that code smells were well represented in both approaches.

(iii) likely impaired the representation of dependencies in relation to graphs. As shown in the second line of the table, 93% of the participants agreed that the dependencies are better represented in the graph than in the VISADEP (80%). No participants disagree with the quality of the representation in both approaches.

Table 5.7: Quality Degree of the Visualization Elements

Visualization Element	**Approach**	**Agree**	**Neutral**	**Disagree**
Dependencies	Graph	93%	7%	0%
	VISADEP	80%	20%	0%
Code smells	Graph	73%	13%	13%
	VISADEP	73%	13%	13%
Concerns	Graph	40%	20%	40%
	VISADEP	60%	33%	7%
Classes	Graph	73%	27%	0%
	VISADEP	87%	7%	7%
Component	VISADEP	60%	20%	20%
Method	VISADEP	87%	7%	7%
Code navigation	VISADEP	73%	20%	7%

Precision and Recall per Participant

After computing precision and recall for each type participant in both approaches, we might make the following observations (see Table 5.8 for details). We noticed that regardless the type of design problem, participants tend to have greater precision in VISADEP, in most cases and recall is the same between better and worse.

Table 5.8: Effect of VISADEP on Precision and Recall per Participant

Participant	Tool	Precision	Recall	Effect on Precision	Effect on Recall
P1	**Graph**	50%	9%	–	↑
	VISADEP	50%	27%		
P2	**Graph**	0%	0%	–	–
	VISADEP	0%	0%		
P3	**Graph**	50%	55%	↑	↓
	VISADEP	57%	36%		
P4	**Graph**	50%	36%	↑	↓
	VISADEP	60%	27%		
P5	**Graph**	50%	36%	↑	↑
	VISADEP	100%	45%		
P6	**Graph**	0%	0%	–	–
	VISADEP	0%	0%		
P7	**Graph**	0%	0%	↑	↑
	VISADEP	67%	18%		
P8	**Graph**	50%	18%	–	↓
	VISADEP	50%	9%		
P9	**Graph**	50%	36%	↑	↓
	VISADEP	75%	27%		
P10	**Graph**	50%	9%	↓	↓
	VISADEP	0%	0%		
P11	**Graph**	50%	18%	↑	↑
	VISADEP	100%	36%		
P12	**Graph**	50%	18%	↑	–
	VISADEP	100%	18%		
P13	**Graph**	50%	45%	↑	↓
	VISADEP	100%	27%		
P14	**Graph**	0%	0%	↑	↑
	VISADEP	50%	9%		
P15	**Graph**	50%	18%	↑	↑
	VISADEP	63%	45%		

5.3 Threats to Validity

We carefully designed and conducted our empirical study, as discussed in Section 5.1. For instance, we delimited our study scope before conducting the designed controlled experiment. Additionally, we relied on previous work (10, 12) to define our research questions and how to assess them after conducting the experiment. However, our study might have been affected by certain threats to validity, even through we have applied techniques to mitigate them whenever possible. We discuss these threats and their respective minimization according to a well-known guideline (57) as follows.

Construct Validity. We designed our controlled experiment through a cross-over design study technique in order to mitigate threats due to the limited number of participants available for conducting the experiment. For instance, we aimed at minimizing threats regarding the learning that participants might

have experienced by using one visualization approach before the other approach and *vice-versa*. Regarding the selected system for participants to identify design problems, namely the Apache OODT system, we have selected two specific components: Workflow Manager and PushPull. We aimed at minimizing any threats regarding the different difficulty levels for identifying design problems in each component as follows. We have selected two components that are quite similar in terms of size and complexity, as well as the number of code anomalies and design problems. Moreover, all subjects received basic training about the two components, and half of the participants identified design problems in a specific component through a particular approach (Section 5.1.3). By analyzing the experiment results, we did not find evidence that one component was easier to identify design problems than the other. Regarding the collection of smell agglomeration instances for conducting the experiment, we have applied a technique reported by the literature (10). The precision of this technique is equal to 80% in the identification of design problems. In addition, we performed pilot experiments with volunteers to improve both experiment design and artifacts. Also, we are not sure that the selected agglomeration is the ideal one to find the design problem. Eventually, more than one agglomeration of smells may be required to identify a design problem. In fact, some agglomerations only include smells of the same type (for example, only Feature Envy instances). Thus the experiment participant may have failed to identify the design problem, not because the agglomeration is not good, not even because the visualization is not adequate, but because the agglomeration we provided was not enough. We mitigate this by doing a manual analysis of each agglomeration to see if it provides information that is minimal enough to help identify the design problem that co-occurs with it in the source code. This mitigation was done by a single author (rather than peers as was done in other validation activities) during the selection of agglomerations based on the existing design problems oracle of the selected systems.

Internal Validity. Also due to the limited number of participants, we had to conduct our experiment in different dates and, sometimes, remotely via videoconference. Aimed at mitigating threats regarding the experiment conduction, we have provided equivalent training for all participants about the main concepts required to conduct the experiment. For this purpose, we have determined a strict duration of the participant training and use the same training artifacts, e.g., a common slide presentation (all artifacts are available in our study companion website (58)). During the experiment conduction, three participants encountered problems with the online experiment form we used

to collect their response and, therefore, they were not able to complete the form. To collect their response, we transcribed the video of screen recording of everything that the participant wrote on the form. This may be a threat to our study since it is based on the researcher's perception. To reduce this threat, we carefully transcribed the answers written by the participants and presented in their recorded screens. Taking into account that we have among the participants, a participant who have developed the graph-based visualization and two others that have wide knowledge of the analyzed systems this can be a threat to the validity of our study. In addition, as our analysis was based on the participants' opinions, a threat to the validity of our study is the poor completion of the experiment questionnaires. To mitigate this threat we observed the experiment, we have provided advice to subjects whenever necessary. This assistance was fundamental to ensure that all participants properly answered the questionnaires. Regarding this last item, it is important to mention that we never interfered in the tasks performed by subjects. In fact, we only helped them to understand all the questions and tasks. We tried to answer the participants' questions whenever possible without biasing their participation. In order to have the complete data of some participants of both the characterization form and the experiment questionnaires that were not fully filled, we contacted once again the participants to respond the missing questions.

Conclusion Validity. We conducted a careful data collection to minimize the problem with respect to missing data, but we encountered some problems during the experiment that could be a threat to our study. A threat to the validity of our study is the version of the Pushpull that two out of the 15 participants (13%) analyzed in graph-based visualization, which made them unable to do the *pushpull.protocol* agglomeration proposed for the experiment since the classes of the agglomeration were not in this version. To mitigate this threat, we ensured that the version of the classes of the other agglomeration analyzed by the participants had the same code as the version that we selected for the experiment. Regarding the computation of precision and recall, we relied on a reference list of design problems which was built by a previous work (29). However, due to the limited number of design problem instances, we have manually revised the reference list and added other instances (see Section 5.1.2 for details). Due to the subjective nature of these tasks, some threats might have affected the list revision and, consequently, precision and recall computations. We mitigate possible threats by conducting the revision tasks in a pair: two researchers have participated and discussed the inclusion of

a design problem in the reference list. They have discussed all instances based on source code analysis and counted on the opinion of a third researcher to reach a consensus.

External Validity. Some factors may prevent the generalization of our research findings. Our study relies on a limited set of participants and systems, which might have affected the generality of our findings. In fact, we used a sample of 15 subjects. This sample may not be enough to achieve conclusive results. However, we mitigate possible threats in several ways: we selected a balanced number of participants from academia and industry to make our findings representative of both contexts; we used a cross-design study to mitigate learning biases; and we selected two different subsystems of a large and complex software system (Apache OODT), which are affected by different types of design problems. The limited set of design problems selected for the study is another threat to our study. The problem it causes is the difficulty of generalizing our results to any design problem. A design problem can vary in size (number of design elements affected), difficulty in reasoning about it (some are in more "hidden" regions of the source code), etc. The decision for a limited set of design problems was due to the short time of the experiment and the limited amount of participants available to carry out the experiment. Our mitigation was to try to select varied design problems in size, difficulty, etc., but without this difficult the identification during the experiment.

5.4 Summary

This chapter has presented and discussed the results of an empirical evaluation of the VISADED approach. We discuss that, for certain types of design problems, VISADEP outperforms the graph-based approach. That is the case of design problems such as Concern Overload and Ambiguous Interface. However, for the other types design problems, VISADEP has an equivalent or lower accuracy (both precision and recall) in the identification of design problems when compared to the graph-based approach.

The next chapter concludes the dissertation and provides some suggestions for future work.

6
Conclusion

There is empirical evidence that design problems are useful hints of maintenance problems affecting a system (8, 32, 33). Each design problem is characterized by either a single or multiple violations of design principles (1), which might affect multiple code elements together. Due to their negative effect on software maintenance, developers should identify and eliminate design problems whenever possible. However, identifying design problems is far from trivial (6). In fact, several software systems suffer from obsolete or scarce design documentation (7), which often leads developers to analyze the source code in order to identify design problems. However, the scattered nature of most design problems often makes it difficult for developers to identify them (9, 22).

Past work suggests that code smells often indicate design problems (10, 11, 32). A code smell is an anomalous code structure that should be corrected whenever possible (1). However, it has been shown that a single code smell might not suffice to reveal a design problem (10, 12, 29). That is because around 80% of the code smells affecting a single system tends to interrelate while realizing a design problem (10). In this context, a recent study (10) has introduced the smell agglomerations, aimed at characterizing whenever code smells interrelate in the source code. However, a limitation of using smell agglomerations to identify design problems is that a single agglomeration might contain several code smells. Thus, approaches for summarizing data about a smell agglomeration for developers have become essential.

A few recent studies (29, 48) have introduced visualization approaches for smell agglomerations. They aimed at visually representing the code smells that compose an agglomeration, so that developers can reason about their interrelations and identify a design problem. However, they limitedly represent the smell agglomerations and, therefore, fall short in supporting developers in practice. This dissertation addresses this limitation by introducing VISADEP, a novel visualization approach for smell agglomerations. For this purpose, we first propose and evaluate a preliminary graph-based approach. Based on our evaluation results, we refine this preliminary approach and introduce a novel one. We present the VISADEP tool, which implements our novel approach and evaluate our approach with developers from both academy and industry.

6.1 Main Findings and Contributions

The content of this Master's dissertation relies on several empirical studies, which we have conducted in either an *ad hoc* (study literature review) or a systematic way (experiments). Each study has contributed to the derivation of study findings and other contributions. Our goal is to support both developers in their daily basis, but also researchers in conducting future research towards improving the states of the art and the practice. Thus, we summarize and discuss our main findings and contributions as follows.

A literature review about visualization approaches. In a first moment, we have conducted an *ad hoc* literature review aimed at characterizing the state of the art about the visualization approaches for either code smells and smell agglomerations. The literature has been shown extensive about software visualization (52). However, we expected that only a few studies have been focusing on the visualization of code smells (15, 49, 44) Specifically, we were aware that the concept of smell agglomeration had been formally introduced very recently in 2016 (10). Thus, we assumed that only a few studies would be available online for consultation and exploring. All these factors have led us to conduct an *ad hoc* literature review instead of a systematic literature review (62). We present the results of our literature review in Section 2.3.

A desiderata for visualization approaches of smell agglomerations. Based on the results of our literature review, we have identified various positive aspects of the existing visualization approaches for code smells and smell agglomeration that should exist in our novel visualization approach for smell agglomerations. For instance, the two existing approaches for smell agglomerations (10, 30) provide a minimalist visualization of the code smells and their interrelations. All these positive aspects have inspired us to build a desiderata, i.e., a list of desired aspects that any visualization approach for smell agglomerations should implement. We present and discuss our desiderata in Section 2.4. Our goal is to support future research about the proposition of novel and accurate visualization approaches.

A mixed-method empirical study about the identification of design problems via smell agglomerations. Prior to the proposition of our visualization approach for smell agglomerations, we have conducted a study (29) aimed at understanding if developers can effectively identify design problems via smell agglomerations. We performed a quasi-experiment and interviews with 11 de-

velopers. Surprisingly, our results revealed that only 36.36% of the developers found more design problems when explicitly reasoning about the interrelated code smells of a smell agglomeration when compared to each single code smells. However, 63.63% of the developers have had less false positives when using the smell agglomerations to identify design problems. Additionally, our interviews have revealed that a graph-based structure could be appropriate to represent smell agglomerations. The latter has inspired us to propose our graph-based visualization approach presented and evaluated in Chapter 3.

A graph-based visualization approach for smell agglomerations. As aforementioned, our previous study (29) has revealed that, from the developers' perception, smell agglomeration might fit well to a graph-based visual representation. Because of that, we looked at the visualization approaches for smell agglomerations proposed by the literature (29, 30) and decided to propose a novel, graph-based visualization approach. We introduce this approach in Chapter 3). In order to empirically evaluate it, we have conducted a mixed-method empirical study (also in Chapter 3) composed of: (i) a quantitative study aimed at computing precision and recall of developers through the use of our approach; and (ii) a qualitative study to assess how well each visual element is represented by our approach. Our results for (i) suggested a precision of up to 100% but a recall of up to 42%. It suggests a need for improving the visualization to help developers identify even more design problems in an accurate way. For (ii), we have observed that the graph-based structure has helped reveal design problems through the smell agglomeration, but certain visual elements should be better represented (such as the code smells that affect each design element), and other should be added up (such as the concern representation).

An enhanced graph-based visualization approach. The study results presented in Chapter 3 have revealed some drawbacks of our visualization approach. As aforementioned, the lack of concerns being represented together with the code smell interrelations has possibly impacted on the low recall rates. On the other hand, we have that the graph-based representation of a smell agglomeration actually supported developers in identifying design problems. By considering this trade-off, we decided to enhance our graph-based visualization approach by adding visual information about the concerns that affect the design elements also affected by design problems. This enhancement aimed at addressing the lack of concern representation, which we considered the most immediate deficiency of our approach at that time. We further detailed our

enhanced graph-based visualization approach in Section 5.1.2.

A novel visualization approach for smell agglomerations. After empirically evaluating our graph-based visualization approach, we have drawn several conclusions as discussed in Chapter 3. However, even after enhancing our approach (as discussed in Section 5.1.2), we still fell a need for improving the way how we intend to support the identification of design problems through smell agglomerations. For instance, the visual representation of design elements as graph nodes (in our case, coloured boxes) differs a lot from a very usual representation with similar purpose and used by the Unified Modeling Language (UML) (19). Thus, we decided to fully re-think our visualization approach, as described in Chapter 4. As a result, we introduced the VISADEP approach, which implements several improvements that we did not include in the enhanced visualization approach. In fact, VISADEP remains graph-based: for instance, it still represents design elements as graph nodes and smell interrelations as graph edges. However, the novelty relies on exploring different and complementary views of each single smell agglomeration. In detail, we present the smell agglomeration at the levels of component (the highest abstraction level), class (an intermediate abstraction level), and method (the lowest abstraction level except for the source code level itself).

A mixed-method empirical study to evaluate VISADEP. In order to evaluate the practical differences of identifying design problems through our enhanced graph-based visualization approach and VISADEP, we have designed and conducted a mixed-method study. Similarly to the study presented in Chapter 3, we have combined qualitative and quantitative analyzes to understand the scenarios in which each visualization approach proposed by us might perform better than the other. All study results and their respective implications are further detailed in Chapter 5.

From our quantitative analysis, we have drawn two main observations. First, VISADEP provides higher precision and recall for the identification of Concern Overload and Ambiguous Interface. However, for the latter, both precision and recall rates are far from sufficient for practical use by developers in the industry. Second, the graph-based approach performs better than VISADEP for the identification of Scattered Concern. However, for the former, both the graph-based approach and VISADEP have quite close precision. From our qualitative analysis, we observed: slight to significant improvements in the representation of information such as concerns; and a slight worsening in the representation of smell interrelations, which might be justified by the fact

that the graph-based approach has a more intuitive notation to interrelations (simple arrows instead of UML-like dependency representations).

Expected publications. In this Master's dissertation, we summarize various study findings and contributions, as we discuss previously in this section. Because of that, a natural scientific movement would be publishing each set of cohesive findings and contributions as workshop, conference, and journal papers. Table 6.1 presents a list of publications that we expect to achieve from this dissertation in the near future. The first column presents the type of publication (i.e., workshop, conference, or journal paper). The second column describes the main purpose of the publication. We highlight that significant efforts have been made to write and submit these papers as soon as possible.

Table 6.1: Expected Publications from this Master's Dissertation

Publication Type	Purpose
Workshop paper	Introducing and empirically evaluating our enhanced graph-based visualization approach for smell agglomerations. This paper includes the contents of Chapter 3 and Section 5.1.2
Conference paper	Introducing and empirically evaluating the VISADEP approach for visualizing smell agglomerations. This paper includes the content of Chapters 4 and 5
Journal paper	Presenting the entire dissertation achievements, especially our desiderata for visualization approaches of smell agglomerations (Section 2.4). This paper includes all chapters and additional content, such as the first insights about how to improve VISADEP

6.2 Limitations

This Master's dissertation is composed of: (i) a literature review aimed at characterizing the existing visualization approaches for code smells and smell agglomerations; (ii) a desiderata of visualization approaches for smell agglomerations; (iii) the proposition of novel visualization approaches for smell agglomerations; and (iv) multiple empirical studies aimed at evaluating each proposed visualization approach. Each component of this dissertation has specific limitations, which mostly concern the applied methodologies, human aspects, and data sets, for instance. We present the main limitations of this dissertation and draw directions for addressing them in future work, as follows.

Theoretical limitations. These limitations include the threats to the validity affecting each empirical study, eventual deficiencies of our *ad hoc* literature review about visualization approaches, and related topics.

- VISADEP was assessed for a small set of design problems (Concern Overload, Scattered Concern, Fat Interface, and Ambiguous Interface). So we can not generalize the results. In addition, there are different types of problems, and each design problem needs specific information. Fat Interface and Ambiguous Interface require an improvement. Although the results are not good, they do not differ much from the results obtained in the previous study for interface design problems (29).
- Small set of participants and systems. Both studies (graph-based and VISADEP) relies on a limited set of participants and systems, which might have affected the generality of our findings. The sample of participants in both studies may not be enough to achieve conclusive results, as this sample size did not enable us to achieve statistically-significant results.

Limitations of the proposed visualization approaches. These limitations regard the design of each visualization approach for smell agglomerations proposed in this dissertation, namely, the graph-based approach and VISADEP.

- Based on the result (Section 5.2) of our study, we noticed the need to better represent the interfaces of a program. It could support the identification of design problems related to interface, such as Fat Interface, in which our approach did not achieve a satisfactory result.
- Both VISADEP and graph-based does not show the method calls in method view. This would help a lot to analyze the classes. Having methods helps to analyze the interfaces of the classes that are defined by the public methods.
- Both VISADEP and graph-based does not show in the code smell visualization, the information of the program elements involved in the existence of a code smell. For example, show the methods envied in other classes for a Feature Envy code smell. VISADEP does not provide a clear graph of dependencies for code smells, as supported by the graph-based approach. One possible solution is to recommend the use of both VISADEP and graph-based visualizations to support the detection of certain design problems.

Limitations of the support tools of each visualization approach. These limitations concern technical issues of the tools that implement each proposed visualization approach, i.e., the graph-based approach and VISADEP.

– The current version of VISADEP only runs on the Eclipse Luna. It depends on Eclipse Luna and because it is an extension of Organic that has been implemented with a series of dependencies with Eclipse Luna that are not compatible with the current Eclipse. The solution is to refactor the code so that the dependencies are directed towards the last versions of the code. Another solution would be to redesign the application to depend on the most current versions.

– The fact that VISADEP depends on an input of concern is a limitation. VISADEP does not provide representative names for the concerns implemented in the source code. The tool does not show the part of the code of a class is addressing a specific concern. This is due to the tool used to extract the concerns. These limitations only depend on the literature in the area. This can be solved when the data mining community provides a powerful tool for extraction of concerns.

– VISADEP does not provide, in the class and method views, the navigation to the source code from each program element (class, method). The user cannot click on the program element and navigate to the source code of this element. In the current version of VISADEP, developer can navigate to the source through the Smell List view.

6.3 Future Work

There are several suggestions for future work based on the achievements of this dissertation. We present some suggestions, which help address the limitations of the previous section, as follows.

– To implement another feature for the VISADEP tool to support visualization of the historic information of the agglomerations. The visual representation of the version history will serve to support the understanding of the evolution of a smell agglomeration. In this way, the developer can figure out what made the agglomeration to grow or shrink. In addition, this may help to know in which design version, a component started to contain an agglomeration and/or implementing other concerns.

– To evolve our visualization approach aimed at supporting the identification of additional design problem types. Our study results (Section 5.2) reveal that concern-based design problems, such as Concern Overload and Scattered Concern, are likely to be identified through our approach rather than others. Thus, an open challenge is improving our approach in

a way that enhances the visual representation of certain data to support better reasoning about non-concern-based design problems.

– To conduct large-scale empirical studies with software developers aimed at evaluating our visualization approach. In this dissertation, we discuss the results of a controlled experiment with developers from both academy and industry. However, as discussed in Section 5.3, the limited number of developers, agglomerations, and design problems represent threats to the study validity that additional studies could address this gap.

– To evaluate the use of our visualization approach in industry settings. This dissertation presents a controlled experiment aimed at evaluating our approach (Chapter 5). However, understanding to what extent the approach supports developers in identifying design problems in their daily basis might provide unique insight about how to improve the approach to address the developers' needs. Our study of Chapter 5 included only participants that were not previously familiar with the subsystems and source code being analyzed.

Bibliography

[1] SURYANARAYANA, G.; SAMARTHYAM, G. ; SHARMAR, T.. **Refactoring for Software Design Smells: Managing Technical Debt**. Morgan Kaufmann, 2014.

[2] MARTIN, R. C.. **Agile Principles, Patterns, and Practices in C#**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[3] MACIA, I.; GARCIA, J.; POPESCU, D.; GARCIA, A.; MEDVIDOVIC, N. ; VON STAA, A.. **Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems**. In: AOSD '12, p. 167–178, USA, 2012. ACM.

[4] GARCIA, J.; POPESCU, D.; EDWARDS, G. ; MEDVIDOVIC, N.. **Identifying architectural bad smells**. In: CSMR09; KAISERSLAUTERN, GERMANY. IEEE, 2009.

[5] SCHACH, S.; JIN, B.; WRIGHT, D.; HELLER, G. ; OFFUTT, A.. **Maintainability of the linux kernel**. Software, IEE Proceedings -, 149(1):18–23, 2002.

[6] TRIFU, A.; MARINESCU, R.. **Diagnosing design problems in object oriented systems**. In: WCRE'05, p. 10 pp., Nov 2005.

[7] KAMINSKI, P.. **Reforming software design documentation**. In: 14TH WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE 2007), p. 277–280, Oct 2007.

[8] FOWLER, M.. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley Professional, Boston, 1999.

[9] PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R. ; LUCIA, A. D.. **Do they really smell bad? a study on developers' perception of bad code smells**. In: 2014 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, p. 101–110, Sept 2014.

[10] OIZUMI, W.; GARCIA, A.; SOUSA, L. S.; CAFEO, B. ; ZHAO, Y.. **Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems**. In: PROCEEDINGS OF THE 38TH

INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (SUBMITTED), ICSE '16, 2016.

[11] YAMASHITA, A.; ZANONI, M.; FONTANA, F. A. ; WALTER, B.. **Inter-smell relations in industrial and open source systems: A replication and comparative analysis**. In: ICSME, Sept 2015.

[12] FERNANDES, E.; VALE, G.; SOUSA, L.; FIGUEIREDO, E.; GARCIA, A. ; LEE, J.. **No code anomaly is an island**. In: INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, p. 48–64. Springer, 2017.

[13] WETTEL, R.; LANZA, M.. **Codecity: 3d visualization of large-scale software**. In: COMPANION OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE Companion '08, p. 921–922, New York, NY, USA, 2008. ACM.

[14] LANZA, M.; DUCASSE, S.. **The class blueprint-a visualization of the internal structure of classes**. In: SOFTWARE VISUALIZATION WORKSHOP (OOPSLA, 2001.

[15] MURPHY-HILL, E.; BLACK, A. P.. **An interactive ambient visualization for code smells**. In: PROCEEDINGS OF THE 5TH INTERNATIONAL SYMPOSIUM ON SOFTWARE VISUALIZATION; SALT LAKE CITY, USA, p. 5–14. ACM, 2010.

[16] PANAS, T.; EPPERLY, T.; QUINLAN, D.; SAEBJORNSEN, A. ; VUDUC, R.. **Communicating software architecture using a unified single-view visualization**. In: ENGINEERING COMPLEX COMPUTER SYSTEMS, 2007. 12TH IEEE INTERNATIONAL CONFERENCE ON, p. 217–228. IEEE, 2007.

[17] LANZA, M.; DUCASSE, S.. **Understanding software evolution using a combination of software visualization and software metrics**. In: IN PROCEEDINGS OF LMO 2002 (LANGAGES ET MODÈLES À OBJETS. Citeseer, 2002.

[18] LANZA, M.; MARINESCU, R.. **Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems**. Springer Science & Business Media, 2006.

[19] RUMBAUGH, J.; JACOBSON, I. ; BOOCH, G.. **Unified modeling language reference manual, the**. Pearson Higher Education, 2004.

[20] ROBILLARD, M. P.; MURPHY, G. C.. **Representing concerns in source code**. ACM Transactions on Software Engineering and Methodology (TOSEM), 16(1):3, 2007.

[21] KÄSTNER, C.; APEL, S. ; KUHLEMANN, M.. **Granularity in software product lines**. In: PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 311–320. ACM, 2008.

[22] YAMASHITA, A.; MOONEN, L.. **Do developers care about code smells? an exploratory survey**. In: REVERSE ENGINEERING (WCRE), 2013 20TH WORKING CONFERENCE ON, p. 242–251. IEEE, 2013.

[23] FIGUEIREDO, E.; SILVA, B.; SANT'ANNA, C.; GARCIA, A.; WHITTLE, J. ; NUNES, D.. **Crosscutting patterns and design stability: An exploratory analysis**. In: PROGRAM COMPREHENSION, 2009. ICPC'09. IEEE 17TH INTERNATIONAL CONFERENCE ON, p. 138–147. IEEE, 2009.

[24] EADDY, M.; ZIMMERMANN, T.; SHERWOOD, K. D.; GARG, V.; MURPHY, G. C.; NAGAPPAN, N. ; AHO, A. V.. **Do crosscutting concerns cause defects?** IEEE transactions on Software Engineering, 34(4):497–515, 2008.

[25] MARINESCU, R.. **Measurement and quality in object-oriented design**. In: 21ST IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM'05), p. 701–704, Sept 2005.

[26] EMDEN, E.; MOONEN, L.. **Java quality assurance by detecting code smells**. In: PROCEEDINGS OF THE 9TH WORKING CONFERENCE ON REVERSE ENGINEERING; RICHMOND, USA, p. 97, 2002.

[27] RATZINGER, J.; FISCHER, M. ; GALL, H.. **Improving evolvability through refactoring**, volumen 30. ACM, 2005.

[28] WETTEL, R.; LANZA, M.. **Visually localizing design problems with disharmony maps**. In: PROCEEDINGS OF THE 4TH ACM SYMPOSIUM ON SOFTWARE VISUALIZATION, p. 155–164. ACM, 2008.

[29] OIZUMI, W.; SOUSA, L.; GARCIA, A.; OLIVEIRA, R.; OLIVEIRA, A.; AGBACHI, O. ; LUCENA, C.. **Revealing design problems in stinky code: A mixed-method study**. In: SBCARS17 (ACCEPTED), 2017.

[30] VIDAL, S. A.; MARCOS, C. ; DÍAZ-PACE, J. A.. **An approach to prioritize code smells for refactoring**. Automated Software Engg., 23(3), Sept. 2016.

[31] ECLIPSE. **Eclipse integrated development environment**, 2015.

[32] MACIA, I.; ARCOVERDE, R.; GARCIA, A.; CHAVEZ, C. ; VON STAA, A.. **On the relevance of code anomalies for identifying architecture degradation symptoms**. In: CSMR12, p. 277–286, March 2012.

[33] YAMASHITA, A.; MOONEN, L.. **Do code smells reflect important maintainability aspects?** In: SOFTWARE MAINTENANCE (ICSM), 2012 28TH IEEE INTERNATIONAL CONFERENCE ON, p. 306–315. IEEE, 2012.

[34] LIVIERI, S.; HIGO, Y.; MATUSHITA, M. ; INOUE, K.. **Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder**. In: SOFTWARE ENGINEERING, 2007. ICSE 2007. 29TH INTERNATIONAL CONFERENCE ON, p. 106–115. IEEE, 2007.

[35] BASS, L.; CLEMENTS, P. ; KAZMAN, R.. **Software Architecture in Practice**. Addison-Wesley Professional, 2003.

[36] HOCHSTEIN, L.; LINDVALL, M.. **Combating architectural degeneration: A survey**. Information and Software Technology, 47:643–656, 2005.

[37] PERRY, D. E.; WOLF, A. L.. **Foundations for the study of software architecture**. ACM SIGSOFT Software engineering notes, 17(4):40–52, 1992.

[38] MARINESCU. **Detection strategies: metrics-based rules for detecting design flaws**. In: PROCEEDINGS OF 20TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM); CHICAGO, USA, p. 350–359, 2004.

[39] OIZUMI, W.; GARCIA, A.; COLANZI, T.; STAA, A. ; FERREIRA, M.. **On the relationship of code-anomaly agglomerations and architectural problems**. Journal of Software Engineering Research and Development, 3(1):1–22, 2015.

[40] OIZUMI, W. N.; GARCIA, A. F.. **Synthesis of Code Anomalies: Revealing Design Problems in the Source Code**. PhD thesis, PUC-Rio, 2015.

[41] BERTRÁN, I. M.. **On the detection of architecturally-relevant code anomalies in software systems**. PhD thesis, PhD thesis, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil, 2013.

[42] DIEHL, S.. **Software visualization: visualizing the structure, behaviour, and evolution of software**. Springer Science & Business Media, 2007.

[43] CASERTA, P.; ZENDRA, O.. **Visualization of the static aspects of software: A survey**. Visualization and Computer Graphics, IEEE Transactions on, 17(7):913–933, July 2011.

[44] MARINESCU, R.; GANEA, G. ; VEREBI, I.. **Incode: Continuous quality assessment and improvement**. In: CSMR, p. 274–275, March 2010.

[45] VAN EMDEN, E.; MOONEN, L.. **Java quality assurance by detecting code smells**. In: NINTH WORKING CONFERENCE ON REVERSE ENGINEERING, 2002. PROCEEDINGS., p. 97–106, 2002.

[46] HERMANS, F.; PINZGER, M. ; VAN DEURSEN, A.. **Detecting and visualizing inter-worksheet smells in spreadsheets**. In: 34TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE 2012, JUNE 2-9, 2012, ZURICH, SWITZERLAND, p. 441–451, 2012.

[47] FERNANDES, E.; OLIVEIRA, J.; VALE, G.; PAIVA, T. ; FIGUEIREDO, E.. **A review-based comparative study of bad smell detection tools**. In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING, p. 18. ACM, 2016.

[48] VIDAL, S.; GUIMARAES, E.; OIZUMI, W.; GARCIA, A.; PACE, A. D. ; MARCOS, C.. **Identifying architectural problems through prioritization of code smells**. In: SBCARS16, p. 41–50, Sept 2016.

[49] CARNEIRO, G.; SILVA, M.; MARA, L.; FIGUEIREDO, E.; SANT'ANNA, C.; GARCIA, A. ; MENDONÇA, M.. **Identifying code smells with multiple concern views**. In: SOFTWARE ENGINEERING (SBES), 2010 BRAZILIAN SYMPOSIUM ON; SALVADOR, BRAZIL, p. 128–137. IEEE, 2010.

[50] BYELAS, H.; BONDAREV, E. ; TELEA, A.. **Visualization of areas of interest in component-based system architectures**. In: SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS, 2006. SEAA'06. 32ND EUROMICRO CONFERENCE ON, p. 160–169. IEEE, 2006.

[51] BYELAS, H.; TELEA, A.. **Visualization of areas of interest in software architecture diagrams**. In: PROCEEDINGS OF THE 2006 ACM SYMPOSIUM ON SOFTWARE VISUALIZATION, p. 105–114. ACM, 2006.

[52] HERMAN, I.; MELANCON, G. ; MARSHALL, M. S.. **Graph visualization and navigation in information visualization: A survey**. IEEE Trans. Visual Comput. Graphics, 6(1):24–43, Jan 2000.

[53] ORACLE. **Java 7 programming language**, 2015.

[54] SOFTWARE, T.. **The java programming language**, Apr. 2017.

[55] CASS, S.. **The 2016 top programming language**, July 2016.

[56] BACCHELLI, A.; BIRD, C.. **Expectations, outcomes, and challenges of modern code review**. In: PROCEEDINGS OF THE 2013 INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 712–721. IEEE Press, 2013.

[57] WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B. ; WESSLÉN, A.. **Experimentation in software engineering**. Springer Science & Business Media, 2012.

[58] AGBACHI, A.. **Online companion**. `https://benedicteagbachi.github.io/MasterDissertationCompanion/`, year=2018.

[59] ROBILLARD, M. P.; WEIGAND-WARR, F.. **Concernmapper: simple view-based separation of scattered concerns**. In: PROCEEDINGS OF THE 2005 OOPSLA WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE, p. 65–69. ACM, 2005.

[60] STRAUSS, A.; CORBIN, J.. **Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory**. SAGE Publications, 1998.

[61] MATTMANN, C.; CRICHTON, D.; MEDVIDOVIC, N. ; HUGHES, S.. **A software architecture-based framework for highly distributed and data intensive scientific applications**. In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING: SOFTWARE ENGINEERING ACHIEVEMENTS TRACK; SHANGHAI, CHINA, p. 721–730, 2006.

[62] KITCHENHAM, B.; CHARTERS, S. C.. **Guidelines for performing systematic literature reviews in software engineering**. Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.