Universidade Federal da Bahia
Universidade Salvador
Universidade Estadual de Feira de Santana

TESE DE DOUTORADO

Empirically investigating the human role on the code smell effect

José Amancio Macedo Santos

Programa Multiinstitucional de
Pós-Graduação em Ciência da Computação – PMCC

Salvador
18 de junho de 2015

PMCC-Dsc-0019

JOSÉ AMANCIO MACEDO SANTOS

EMPIRICALLY INVESTIGATING THE HUMAN ROLE ON THE CODE SMELL EFFECT

Esta Tese de Doutorado foi apresentada ao Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Estadual de Feira de Santana e Universidade Salvador como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: Manoel Gomes de Mendonça Neto

Salvador
18 de junho de 2015

Ficha catalográfica.

Santos, José Amancio Macedo

Empirically investigating the human role on the code smell effect/ José Amancio Macedo Santos– Salvador, 18 de junho de 2015.

159p.: il.

Orientador: Manoel Gomes de Mendonça Neto.
Tese (doutorado)– UNIVERSIDADE FEDERAL DA BAHIA, INSTITUTO DE MATEMÁTICA, 18 de junho de 2015.

1. Software design. 2. Code smell. 3. Engenharia de Software.
I. Mendonça, Manoel Gomes. II. UNIVERSIDADE FEDERAL DA BAHIA. INSTITUTO DE MATEMÁTICA. III Título.

CDD - 005.1
CDU - 004.41

TERMO DE APROVAÇÃO

JOSÉ AMANCIO MACEDO SANTOS

EMPIRICALLY INVESTIGATING THE HUMAN ROLE ON THE CODE SMELL EFFECT

Esta Tese de Doutorado foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Estadual de Feira de Santana e Universidade Salvador.

Salvador, 18 de junho de 2015

ACKNOWLEDGEMENTS

(in Portuguese)

Agradeço à minha esposa, Lidiane. Sem seu apoio e companheirismo certamente este trabalho não seria possível. Aos meus filhos, Artur e Joana, pela felicidade e amor que trazem às nossas vidas, nos motivando a enfrentar todas as dificuldades.

Agradeço à minha família, pelo apoio e presença. Em especial, aos meus pais, Roosevelt e Araiza. Aos meus irmãos e meus cunhados: Ivone, Germano, Milena, Jamile e Glauber. E um agradecimento especial aos meus sobrinhos Gabi e Bia, que enchem de alegria nossa família.

Ao meu orientador, Manoel Mendonça, pela orientação e ensinamentos precisos. Foi uma experiência enriquecedora ter sido guiado por uma mente tão inteligente.

A todos os colegas do grupo SOFTVIS e do LES. Em especial àqueles com quem trabalhei mais de perto e que me ensinaram e ajudaram de diferentes maneiras: Renato, Carol, Mário, Thiago, Alcemir, Mydiã, Rogeres, Luciana, Cleber e Álvaro.

E, finalmente, aos colegas da UEFS, que tornaram possível eu me concentrar no desenvolvimento deste trabalho.

RESUMO

Contexto: *Code smell* é um termo comumente utilizado para descrever potenciais problemas em projetos de software orientados a objetos. A teoria relacionada a *code smells*, inicialmente apresentada nos anos noventa, tem foco na caracterização de diferentes tipos de *smells* e nas estratégias para sua detecção e remoção. Desde então, estudos empíricos têm avaliado o impacto da adoção do conceito de *code smell* no desenvolvimento de software, ou seja, o "efeito *code smell*". Os resultados destes estudos têm apresentado inconsistências com relação aos efeitos propostos pela teoria. As causas deste fenômeno não são bem compreendidas. **Objetivo**: Melhorar a compreensão sobre o *efeito code smell*, identificando problemas que têm dificultado uso do conceito no desenvolvimento de software. Mais especificamente, nos concentramos no papel humano na detecção de *smells*. O fator humano é uma variável pouco considerada pela teoria, mas afeta significativamente os resultados dos estudos empíricos sobre o tema. **Método**: Nós exploramos o problema de forma empírica, a partir de duas perspectivas. Primeiro, nós sintetizamos o conhecimento relacionado ao *efeito code smell* a partir de um conjunto de estudos primários existente na literatura. Para isso, nós realizamos uma revisão sistemática, com base no método de síntese temática. Segundo, nós propusemos a investigação de fatores que afetam a percepção humana sobre *smells*. Neste caso, nós realizamos uma família de experimentos controlados, investigando um extensivo número de fatores que afetam a percepção humana sobre *smells*. Os fatores são: o uso de ferramentas de suporte à compreensão do projeto de software; a experiência; o conhecimento teórico sobre *code smells* e o treinamento dos desenvolvedores; e o tamanho do software. A partir da síntese temática e da família de experimentos, nós identificamos alguns desafios para a área, apresentando-os como uma agenda de pesquisa. **Resultados**: Nós percebemos que a avaliação humana de *smells* não é confiável. Os estudos indicam que o grau de concordância relacionado à avaliação humana é baixo. Além disso, os estudos mostram que atributos demográficos, como experiência do desenvolvedor, afetam o grau de concordância na detecção de *smells*. Também concluímos que não há evidências que indiquem o uso do conceito de *code smells* como ferramenta para a avaliação da qualidade dos projetos de software. Estas evidências são as inconsistências nos resultados correlacionando os *smells* e atributos de software, como mudanças no código. Em alguns casos, os estudos concordam que os *smells* não tem correlação com certos atributos. Isso ocorre, por exemplo, com estudos correlacionando *smells* e esforço na manutenção de software. **Conclusão**: Esta tese evidencia uma lacuna existente entre a teoria e os estudos empíricos relacionados ao *efeito code smell*. Nossas evidências apontam o baixo número de estudos abordando o papel humano na detecção de *smells* como principal causa desta lacuna. Pesquisas na área devem se concentrar em entender este aspecto. Este não parece ser o caminho seguido pela área, que tem se concentrado principalmente no desenvolvimento de ferramentas de suporte, como ferramentas para detecção automática e remoção de *smells*. Para auxiliar neste sentido, nós propomos uma agenda de pesquisa indicando pontos importantes a serem considerados: i) investigação da natureza dos *smells*, agrupando-os de acordo com suas características; ii) exploração de fatores humanos afetando a detecção de *smells* e seus inter-relacionamentos; iii) exploração de aspectos cognitivos afetando a percepção humana de *smells*; e iv) identificação de atributos relacionados aos estudos empíricos de forma a delinear o contexto onde resultados podem ser agrupados. Estes desafios indicam alguns caminhos que a área pode seguir para sistematizar o uso do conceito de *code smell* no desenvolvimento de software. Apesar das dificuldades, nós acreditamos que isso é possível e nossa tese visa a contribuir neste sentido.

Palavras-chave: Projeto de software, falhas de projeto, desarmonias em projetos de software

ABSTRACT

Context: *Code smell* is a term commonly used to describe potential problems in the design of object-oriented systems. The theory about code smells, initially presented in the nineties, is focused on characterization of different types of smells and heuristics for their detection and removal. Since then, empirical studies have evaluated the impact of the practical adoption of the code smell concept on software development. We call this issue *the code smell effect*. The findings of empirical studies on the code smell effect have presented inconsistencies with respect to what is expected by the theory. This phenomenon is not well understood. **Objective**: To enhance the understanding of the code smell effect, by characterization of problems hindering the practical adoption of the smell concept and by investigation of their origin. More specifically, we focus on the major confounding factor affecting observations on the code smell effect, which is the human role on smell detection. The human role is mostly disregarded by the theory and it strongly affects the experiments on the subject. **Method**: We explore the problem empirically, from two perspectives. First, we synthesize the current knowledge on the code smell effect from a set of primary studies published in the literature. To do this, we performed a systematic review based on thematic synthesis. Second, we propose the investigation of factors affecting the human evaluation of smells. To do this, we performed a family of controlled experiments. In it, we investigate an extensive number of factors affecting the human perception of smells. The factors are the use of design comprehension tool support; developer's experience, knowledge and training; and software size. From the thematic synthesis and the family of controlled experiments, we identify challenges for the area and present them as a research agenda. **Results**: We find out that human evaluation of smell should not be trusted. The studies indicate that the agreement on smell detection is low. They also show that demographic attributes, such as developers' experience, impact the agreement on smell detection. We have also concluded that, nowadays, there are no evidence supporting the use of code smells for practical evaluation of design quality. Our evidences are divergent findings correlating code smells and software attributes, such as changes on source code. In some cases, the studies converge showing that code smells do not impact some attributes. This happens, for example, with studies correlating smells and maintenance effort. **Conclusion**: This thesis points out to the gap between the theory and the experimental findings about the code smell effect. Our evidences indicate that the main source for this gap is the low number of studies focusing on the human role on the smell effect. The researches on the subject should focus on better understanding this issue. This does not seem the path followed by the area, which has directed its effort on researches focused on tool assessments, such as automatic detection or smell removal. In order to face the problem, we propose a research agenda, indicating that the area needs to: i) investigate the nature of smells, grouping them according to their characteristics; ii) explore human factors affecting smell detection, and their relationships; iii) explore cognitive aspects affecting human perception of smells; and iv) outline attributes of the experiments in order to classify the context where findings can be grouped. These challenges indicate some paths to be followed by the area in order to systematize the use of code smell in the practice of the software development. Despite difficulties, we believe that this is possible and our thesis contributes for this journey.

Keywords: Software design, code smell, design flaw

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

SE	Software Engineering
OO	Object-oriented
ESE	Experimental Software Engineering
SR	Systematic Review
MS	Systematic Mapping Study
LOC	Lines of code
GQM	Goal, question, metrics template

Chapter

1

This chapter presents the motivation of this work, including key aspects from literature, the research goals and approach, highlighting the main results. The chapter also discusses limitations of the research. Lastly, it outlines the thesis.

INTRODUCTION

The software engineering (SE) community has extensively discussed strategies for the systematic evaluation of design problems in object-oriented (OO) systems. This issue was raised as early as 1996 by Riel (1996). He presented insights into design improvement and introduced the term "design flaw" in his book. Fowler (1999), who coined the term "code smell", focused on refactoring and presented a catalog of smells, characterizing and proposing specific actions to remove them. Following suit, Lanza and Marinescu (2005) focused on metrics and heuristics to detect what they called "disharmonies".

Although the terms "design flaw", "code smell" and "disharmony" have been used to define potential design problems, this work adopts the term *code smell*, or simply *smell*, to refer to such problems. A common (and expected) characteristic of the works discussing design problems is that they are built on principles of the OO paradigm, such as inheritance, information hiding and polymorphism (Meyer, 1988). The authors consider that problems in design occur when these principles are broken. Then, they propose different strategies for code smell detection, which is fundamental as it affects common activities of software development, such as diagnostics in code inspection or refactoring and maintenance decisions.

The works of Riel, Fowler, and Lanza and Marinescu mainly outline a theory about code smells. The concept is well accepted as indicative of design problems in SE. The theory is formed by a wide discussion about heuristics presenting directives "pointing developers in the right direction on smell detection"(Fowler, 1999). The authors build their heuristics identifying aspects that affect the quality of software design. Lanza and Marinescu (2005), for example, consider that three distinct aspects contribute to the code smell characterization: its size, its interface and its implementation. The works present different types of code smells characterizing design problems, which potentially affect software evolution attributes, particularly software maintainability.

Despite the code smell concept being defined by an accepted theory, it is imperative to observe the effects related to its practical adoption on software development. We call this issue the *code smell effect*. The Experimental Software Engineering (ESE) discipline offers a suitable framework for this type of assessment (Wohlin et al., 2012). ESE discusses the application of methods of experimentation in the software engineering context. Since the early 90's, many empirical results have been presented in different areas of SE. Empirical studies have addressed the code smell concept from different perspectives, since the early 2000's.

We classify these studies into three categories. The first type is characterized as correlation studies. These studies try to establish a correlation between code smells and some attribute of the software, such as bugs or number of modifications on classes. They commonly evaluate the impact of smells, extracting data from software repositories (Fontana et al., 2013; Peters; Zaidman, 2012; Macia et al., 2012; Zazworka et al., 2011; Olbrich; Cruzes; Sjøberg, 2010; Olbrich et al., 2009; Li; Shatnawi, 2007). The second type of study investigates the role of humans on code smell detection (Palomba et al., 2014; Santos et al., 2014; Moonen; Yamashita, 2012; Schumacher et al., 2010; Mäntylä; Lassenius, 2006; Mäntylä, 2005). They involve humans on tasks related to the code smell evaluation, analyzing agreement, decision drivers and (in)consistencies in their answers. Lastly, the third type is related to tool assessment, such as automatic detection (Schumacher et al., 2010; Mäntylä; Lassenius, 2006) or software visualization supporting smell detection(Carneiro; Mendonça, 2013; Murphy-Hill; Black, 2010; Parnin; Görg; Nnadi, 2008; Simon; Steinbruckner; Lewerentz, 2001).

In this work, we consider that the first two types of studies strengths the understanding of the *code smell effect*. These types are the correlation studies and studies investigating the human role on smell detection. These studies have presented findings in the opposite direction of the well accepted idea that considers code smell as an indicative of potential problems arising from bad design. For example, Sjøberg et al. (2013) investigated the relationship between smells and maintenance effort. They noted that none of the investigated code smells was significantly associated with increased maintenance effort. Macia et al. (2012) investigated the relationship between code smells and problems that occur with an evolving system's architecture. In their study, they noted that many of the detected smells were not related to architectural problems. Yamashita (2013) summarized a wide analysis based on the same experimental setup used by Sjøberg et al. (2013). One of her findings is that "aggregated code smells are not so good indicators of system-level maintainability".

In fact, there is no strong evidence linking code smells with problems arising from bad design choices. Or at least, the question is not well understood, yet. We highlight some statements reinforcing this idea. Zhang, Hall and Baddoo (2011) declared that *"...we do not know whether using code bad smells to target code improvement is effective"*. Sjøberg et al. (2013) declared that *"the present focus on bad design as operationalized by code smells may be misdirected"*. From our literature review, we concluded that the experimental findings focused on code smells do not consistently support the theory related to the concept. The reasons for this phenomenon are not well explored.

1.1 PROBLEM STATEMENT

The problem target by this thesis is

the lack of understanding about the code smell effect.

In other words, the area lacks understanding how the adoption of the code smell concept reflects in practice and how aligned is this with the current theory on the subject. Figure 1.1 describes this scenario. At the top, we show theory snippets as presented in the books on the subject. The theory contains discussions about the causes (at the top left in the figure) and effects (at the top right in the figure) related to the existence of code smells in software systems. For example, Lanza and Marinescu (2005) declare that the causes for the existence of the *feature envy* code smell are methods accessing "directly or via accessor methods a lot of data of other classes". An expected consequence on source code infected by the feature envy code smell is: "a change in a method triggers changes in other methods and so on".

At the bottom of Figure 1.1, we represent the experimentations that the area has performed investigating the code smell effect. As previously discussed, these studies aim to correlate code smells with some software attributes, or they evaluate the human role on smell detection. Due to this, different treatments have been considered (at the bottom left in the figure). Some studies adopt tool support and different heuristic strategies in order to correlate code smells and software attributes. Other studies adopt humans as subjects performing code inspection or software maintenance activities. The empirical studies on code smell effect produce the output representing the actual observations on the subject (at the bottom right in the figure).

The arrow in the middle of Figure 1.1 represents the confounding factors affecting the experimentations. Different confounding factors might be considered. For example, studies focusing on tool support or heuristics for automatic detection might consider the heuristics, thresholds and the object software as potential confounding factors for the experiments. Studies involving humans might to consider an extensive set of aspects as confounding factors, such as demographic data, the task, the object software and any other aspect that potentially affect the human perception of code smells.

1.2 APPROACHING THE PROBLEM

Our premise is that the output of the empirical studies does not consistently support the theory related to the code smell effect. The investigation of this phenomenon involves explorations from different nature. We have to consider the theory and the experimentations about the code smell effect.

1.2.1 The theory

From the scientific perspective, we have to consider if the theory is consistent. Some questions springs up from this realization. One of these questions is: does the nature of code smells really reflect problems in software design? This requires evaluation of

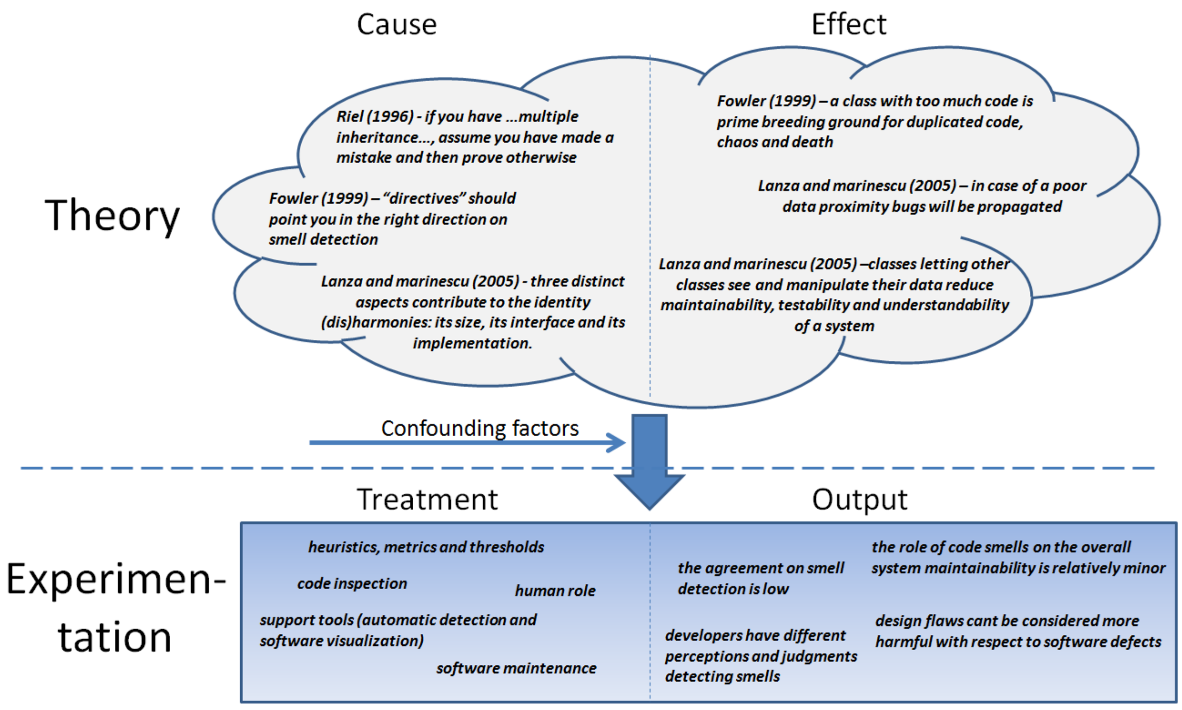

Cause
Effect
Theory
Riel (1996) - if you have ...multiple inheritance..., assume you have made a mistake and then prove otherwise
Fowler (1999) – "directives" should point you in the right direction on smell detection
Lanza and marinescu (2005) - three distinct aspects contribute to the identity (dis)harmonies: its size, its interface and its implementation.
Fowler (1999) – a class with too much code is prime breeding ground for duplicated code, chaos and death
Lanza and marinescu (2005) – in case of a poor data proximity bugs will be propagated
Lanza and marinescu (2005) –classes letting other classes see and manipulate their data reduce maintainability, testability and understandability of a system
Confounding factors
Treatment
Output
Experimen-tation
heuristics, metrics and thresholds
code inspection
human role
support tools (automatic detection and software visualization)
software maintenance
the agreement on smell detection is low
the role of code smells on the overall system maintainability is relatively minor
developers have different perceptions and judgments detecting smells
design flaws cant be considered more harmful with respect to software defects

Figure 1.1 The code smell based on the empirical paradigm

how the attributes of the smells, such as number of lines of code, or coupling, really represent a design problem. Note that this question is related to the principles of the OO paradigm. If the attributes representing smells do not reflect problems in software design, then breaking principles of the OO paradigm also do not.

Another question related to the theory is: do the presented heuristics for smell detection capture bad design? One should consider if the heuristics are well defined. This question involves an important aspect: the heuristics for smell detection disregard that, in practice, smell detection is a human-based task. This can be noted from the analysis of the main works on the subject. For example, Fowler (1999) does not provide objective criteria to identify code smells. He says that one needs to develop its own sense of observation about attributes that could characterize pieces of code as a smell. One has, for example, to develop its own sense of how many lines of code define a long method. Lanza and Marinescu (2005) used an objective definition for code smells based on metrics and thresholds. However, the thresholds are said to be context or domain dependent, again introducing subjectivity in the code smell detection process. Rapu et al. (2004) reinforced this observation declaring that metrics thresholds are mainly chosen based on the experience of the stakeholders. One may then speculate that code smell detection is a subjective task by nature, as its adoption is dependent on human evaluation. The theory discussing the code smell effect disregards this aspect.

1.2.2 The experimentations

We have to consider the biases of the empirical studies investigating the smell effect. Many factors affect the experimental setup of these studies: software domain, software complexity, types of smells, empirical methods, subjects and others. They represent threats intrinsic to the nature of the experimentation. Due to this, we have to consider if the setup of empirical studies on the code smell effect are well defined and limited. Considering each experiment by itself, it is possible to be confident about its findings, if the experiment is well defined. However, in order to evaluate how the empirical studies support the theory about code smells, it is necessary to consider a large set of studies.

Besides difficulties outlining the setup of the experiments on the code smell effect, the human role represents a bias in the evaluation of the experiments. Once smell detection is a human-based activity, it might be affected by the subjectivity. This implies in a complex problem by itself. We consider that two main paths should be followed to face it: i) to deepen the evaluation of the human aspects and how they affect smell detection in the experiments; and ii) to evaluate cognitive aspects on smell detection, which is related to program comprehension, and requires knowledge both in computer science and cognitive psychology (Maletic, 2008). We called this issue *"the code smell conceptualization problem"*. The *code smell conceptualization problem* involves discussions about ensuring the same comprehension of the smell concept and similar personal criteria or thresholds on smell detection.

1.2.3 The human role

We highlight the human role as the major confounding factor affecting the understanding of the code smell effect. It is disregarded by the theory and impacts the practice significantly. Despite its relevance, the human role has been little studied. In a systematic mapping study, Zhang, Hall and Baddoo (2011) noted that most studies in code smells focus on tools and methods for automatic detection. In a complementary remark, Mäntylä and Lassenius (2006) declared that the role of humans as decision makers to evaluate the code design has been little considered.

In some of the few studies focusing on the role of humans, the findings point to problems in the human perception of code smells. Mäntylä and Lassenius (2006) highlighted the conflicting perceptions of different evaluators using code smells to evaluate software quality. Schumacher et al. (2010) found low agreement among developers on code smell detection. In our first empirical studies on the subject, we also had similar findings (Santos et al., 2014; Santos; Mendonça; Silva, 2013).

Many authors in recent studies reinforce the idea that more empirical studies are necessary to enhance understanding of the code smell effect (Sjøberg et al., 2013; Zhang; Hall; Baddoo, 2011; Schumacher et al., 2010; Mäntylä; Vanhanen; Lassenius, 2004). However, the human role also remains little addressed. We argue that the human role is one of the main causes of the inconsistencies between experiments' output and the expected effect on practical adoption of code smells as proposed by the theory on the subject.

1.3 OUTLINING THE THESIS AIM AND METHODOLOGY

The main objective of this thesis is

> *to enhance the understanding of the human role on the code smell effect, characterizing problems hindering the practical adoption of the smell concept and investigating their origin.*

In order to achieve our aim, we adopted the empirical paradigm. Basili (1993), presents two paradigms to be adopted by researchers in SE: the analytic and the experimental. The analytic paradigm uses mathematical methods, proposing a formal theory or set of axioms, developing a theory, deriving results and if possible, comparing with empirical observations. The experimental paradigm is based on scientific methods, which might be engineering or empirical methods. The engineering method is based on observations of existent solutions; and the empirical method is based on the proposition of models from experimentations as proposed by the ESE discipline (Wohlin et al., 2012).

Note that our aim is the exploration of a problem related to the absence of knowledge about the code smell effect. The problem is affected by a potentially uncontrollable number of variables. Due to the novelty of the subject and difficulties on exhausting the discussions on the topic, we have approached this thesis as an exploratory research. As such, in order to reach our objectives we present evidences of the relevance of the human role on the code smell effect and a discussion about this topic as the main contribution of this thesis.

We explore two fronts that we consider as potential sources of evidence. One of these fronts is related to the experimentations being presented about the code smell effect by the SE community. We aim to outline the current knowledge on the topic through a synthesis based on primary studies' results. The other front explores the role of humans on code smell detection. In this case, we adopt controlled experiments as support method in our investigation.

1.3.1 Synthesizing results of the experiments on the code smell effect

As previously discussed, the results of each empirical study by itself can be trusted if the experiment is well defined. However, an overall comprehension about how the empirical results support the code smell theory involves an extensive set of empirical studies. We consider that the area lacks this overall comprehension.

Due to this, in order to explore the experimentations about the theory related to the code smell effect, we

> *present a perspective of the current knowledge on smell effect through a synthesis of the empirical studies addressing the practical adoption of code smells.*

Figure 1.2 describes our first strategy exploring the object problem of this thesis. At the left side, we represent the empirical studies addressing the code smell effect.

At the middle, we represent our process of synthesis. At the right side, we represent the output of this process. We consider our output as one perspective of the current knowledge because other conclusions can be presented. Once we adopted a qualitative research method (which we will discuss later on), the interpretation of the phenomenons is strongly dependent of the researchers (Reichardt; Cook, 1979).

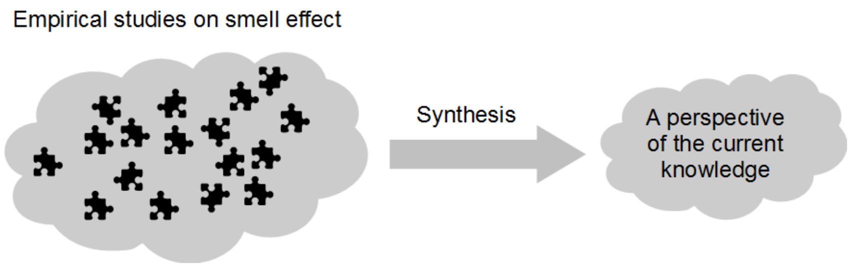

Empirical studies on smell effect
Synthesis
A perspective of the current knowledge

Figure 1.2 A perspective of the current knowledge on smell effect

Our synthesis explores the gap between the theory and the empirical results. We aim empirically and systematically strength the evidences of the human role as the major factor hindering practical adoption of the code smell concept on software development. In order to reach our aim, we focus on empirical studies related to the code smell effect. Note that these studies do not involve only experiments on human role. We also consider the correlation studies because they present findings about the practical adoption of code smells. We focus on three main aspects: i) identification of the topics addressed by the empirical studies on the smell effect; ii) outlining the experimental setup of the empirical studies; and iii) investigating convergence/divergence of the studies' findings.

The method that we adopted was the systematic review (SR) as proposed by Kitchenham and Charters (2007). As opposed to systematic mapping studies (MS) (Petersen et al., 2008), SR involves synthesizing empirical results. One of the main difficulties of this type of SR is that the findings of the empirical studies are frequently presented as textual data. In order to achieve our aims, we had to code the findings and to group them according to the setup of the primary studies. From the synthesis methods available (Dixon-Woods et al., 2004), such as thematic analysis, grounded theory or meta-ethnography, we adopted the thematic analysis, following recommendations proposed by Cruzes and Dybå (2011a). According Cruzes and Dybå, "thematic analysis is often used for identifying, analyzing and reporting patterns (themes) within data in primary qualitative research".

We highlight two main findings of our SR. The first is that human evaluation of smells should not be trusted. The evidences are the studies showing that the agreement on human evaluation of smells is low. They also highlight the relevance of demographic data on smell detection, such as developer experience. Other important finding is that the smell concept does not support the evaluation of design quality, in practice. The evidences are the divergent results from the primary studies of our SR. Moreover, the studies did not find correlation between smells and effort on maintenance activities, and between smells and architectural quality.

1.3.2 Experimenting on the human role on the code smell effect

As previously discussed, we identified the human role as one of the main factors hindering practical adoption of the smell concept on software development. More than that, to the best of our knowledge, this factor has not been systematically explored. In order to face this aspect, we propose

a family of experiments, controlling and investigating an extensive number of variables affecting the human perception of code smells.

The Figure 1.3 describes our second strategy of exploration of the object problem of this thesis. At the left, we represent the empirical studies. At the right, we highlighted some of these studies, representing our family of controlled experiments. Our experiments focus on factors affecting human evaluation of code smells.

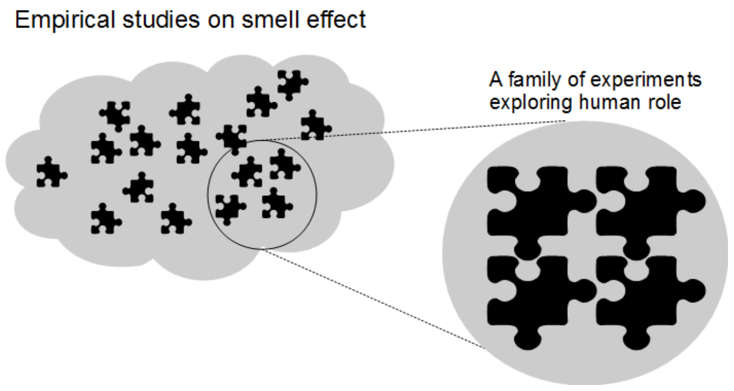

Empirical studies on smell effect
A family of experiments exploring human role

Figure 1.3 A family of experiments investigating the human role on smell detection

We adopted controlled experiment as the classical empirical method for identifying cause-effect relationships (Sjøberg et al., 2005). We run a family of four controlled experiments. According to Basili, Shull and Lanubile (1999), a family of experiments involves replications and variations among the experiments. In our case, we have replicated experiments to check results, as proposed by Shull et al. (2004), and also performed experimental variations to expand and evolve the knowledge on the topic, as proposed by Juristo and Vegas (2009).

The family has four experiments, up to now. The aim is to investigate an extensive number of factors affecting the human perception of code smells. We evaluated the impact of five variables in the agreement on smell detection, trying to identify cause-effect relationships between them. The variables were design comprehension tool support, developer experience, developer knowledge, developer training, and software size. From the experiments, we also explored other human factors not establishing cause-effect relationships, but identifying and grouping them as part of the developers' process for detecting smells. These factors were the strategies and decision drivers adopted by the developers. We call them as moderator variables [1].

[1]A moderator variable affects the causal relationship between the independent and the dependent variable (Dybå; Sjøberg; Cruzes, 2012)

We found that all factors, except design comprehension tool support, affects the agreement. This evidences that developers have their conceptualization about code smells independent of a better comprehension of the code design. As a consequence, we concluded that code smell detection is more related to human traits than to the use of tool support for a better comprehension of the software.

Another important finding is that each factor impacted the agreement, but not definitely. For example, we noted that the developers' experience impacts agreement only when the developers are sure about the smell. When some doubts were considered, the experience did not strongly impact the agreement. We also noted that the developers' knowledge factor is affected by the software size. We concluded that the factors we explored are related to each other, making the evaluation of the human role on code smell effect even more complex. We also consider that this issue evidences the human role as the major cause of the inconsistencies between the theory and the results of the experiments on the code smell effect. As a consequence, researches disregarding the human role on code smell effect might be misdirected.

We also highlight our observations about the developer training factor. We found that the training strongly impacts the agreement on code smell detection. It is one of the main factors affecting the smell detection that we can control. Due to this, we propose the training as the most important variable towards ensuring the practical adoption of code smells. We discuss other findings about our experiments in Chapter 4.

1.4 LIMITATIONS OF THE METHODOLOGY

We understand that our strategies facing the object problem of this thesis have limitations, which are intrinsic to empirical researches. We discuss these limitations considering both observation of the empirical studies through synthesis and investigation of the human role through our family of controlled experiments.

1.4.1 Limitations of the strategy exploring primary studies on the code smell effect

Once there are many empirical studies offering knowledge on code smell effect, we consider that the adoption of a secondary study as a suitable strategy to enhance the knowledge on the topic. A secondary study reviews primary studies relating to a specific research question with the aim of integrating/synthesizing evidence related to the research questions (Kitchenham; Charters, 2007). The types of secondary studies being adopted in SE are systematic literature reviews (SR) and systematic mapping studies (MS). The main difference among them is in the scope. While a SR provides a synthesis in an area, requiring a more limited scope, a MS maps out the research by different classifications of the primary studies, allowing adoption of a broader scope (Wohlin et al., 2013).

One may then speculate that we had to perform a MS, first evaluating the scope from a broad range of primary studies in the area, and only then synthesizing the knowledge. We adopted a SR because we found a MS related to the code smells (Zhang; Hall; Baddoo, 2011) reinforcing findings of our ad-hoc literature review. Moreover, to the best of our knowledge, there is not a SR on the code smell effect.

In our SR, we mapped out context information and synthesized findings of the primary studies, which are mostly textual information. This involved quantitative and qualitative analysis. The most advanced form of data synthesis for quantitative data is meta-analysis, but this approach is impractical when the analysis is based on qualitative methods. For qualitative analysis, different approaches are presented (Cruzes; Dybå, 2011b): thematic analysis, narrative synthesis, comparative analysis, case survey, meta-ethnography, meta-analysis (based on statistical methods to integrate quantitative data from several cases) and scoping analysis. We adopted the thematic analysis, following recommendations proposed by Cruzes and Dybå (2011a). However, we have to consider that other data synthesis strategies would offer a different perspective of the knowledge on the code smell effect.

Other limitations of our SR are inherent to the characteristics of this type of study. We detail them in Chapter 3.

1.4.2 Limitations of the strategy exploring the human role on the code smell effect

We characterize problems related to the investigation of human role on the code smell effect as the *code smell conceptualization problem*. As previously discussed, we consider that two main paths should be followed to face it: i) investigating human aspects affecting smell detection; and ii) evaluating cognitive aspects on smell detection. The first limitation of our strategy facing the problem is that we are not doing any evaluation about the developer's cognitive process. Understanding context variables that affect the cognition process would be useful to tackle the code smell conceptualization problem. This strategy was discarded after a preliminary analysis of works related to psychology and cognition. The effort would be very high due to our low knowledge in psychology and difficulties to perform experiments in the area. We focused on the investigation of human aspects affecting code smell detection.

Other risk involves the definition of the problem, which is very broad. It is possible that the set of variables and human factors affecting the conceptualization of code smells are simply too many to be studied. An alternative would be to limit the studies to a very specific context. However, due to the lack of studies on the topic, we believe it is necessary to produce a more extensive body of knowledge, before one tries limit the scope of studies in the area. For this reason, we end up doing a family of controlled experiments that explored five causal and two moderator variables.

Lastly, there are other research methods that could have been adopted. Four of the most important are:

- **Surveys**. "A survey is a strategy or design for an empirical study... through the data collection process of asking questions of people" (Punter et al., 2003). We did not apply surveys for two main reasons. At first, our aim is to understand which aspects affect the human perception of code smells. To do this we have to perform (or analyze) empirical studies controlling independent and dependent variables. This is in opposition to the central idea of surveys. Surveys are used when control of the independent and dependent variables is not possible or not desirable (Sjøberg; Dyba; Jørgensen, 2007).

- **Case studies**. Case study is a methodology for empirical research in natural (in-vivo) settings (Runeson; Höst, 2009). It would be interesting to observe developers detecting code smells *in loco*. However, we did not have access to an organization dealing with code smell detection that would be willing to support our study. Besides, case studies would affect the number of variables addressed.

- **Ethnographies**. Ethnography involves field observation, focusing on the sociology of meaning (Easterbrook et al., 2008). In this case, a high level of immersion in organizations is necessary. We did not have access to this type of environment.

- **Action research**. In action research, the researchers attempt to solve a real-world problem while simultaneously studying the experience of solving the problem (Easterbrook et al., 2008). As in the case of ethnographies, it was not possible to reach the necessary level of immersion in organizations.

1.5 CONTRIBUTIONS OF THE THESIS

The first contribution of this thesis is that it enhances the understanding of the human role on the code smell effect. We evidenced that there is a gap between the theory and the observations about the practical effect of using the smell concept on software development. For us, this is a relevant contribution because the area lacks discussions about this issue, which might misdirect researchers and practitioners. The characterization of the code smell effect as a poorly understood phenomenon highlights the need for redirecting the research in the area. Currently, most work have focus on tools and methods for automatic detection (Zhang; Hall; Baddoo, 2011), and the human role has been little studied (Mäntylä; Lassenius, 2006).

The main contribution of this thesis are the empirical studies that we carried out, by themselves. To the best of our knowledge, there is not a synthesis on the smell effect and there is not a family of experiments focused on the extensive number of variables as we did. These empirical studies present complementary findings.

For example, from our synthesis, we found that the primary studies focus mainly on the code smell *god class*. From our family of controlled experiments we found that developers focus more on coupling attributes than size of classes on *god class* detection. In this way, we strength the body of knowledge about god class from two different perspectives: we identified *god class* as the main smell addressed, and we present findings about the strategies adopted on *god class* detection.

Another example is related to the relevance of the subjectivity on smell detection. We found in both our synthesis and family of controlled experiments that human evaluation of code smells should not be trusted, because the agreement among the developers detecting smells is low. We discuss the findings of both the synthesis and the family of controlled experiments in Chapters 3 and 4, respectively.

Lastly, we present a research agenda as another contribution of this thesis. From the analysis of our empirical studies, we identified some challenges that we consider relevant, in order to direct researches about the smell effect. Exploring topics listed on the research agenda would enhance the understanding of the code smell effect. For example, we consider that the area lacks studies focusing on the nature of the smells and grouping them according to relevant characteristics. We concluded this because we noted that the extensive variety of smells makes impractical to synthesize findings about a specific type of code smell. We present the research agenda in Chapter 5.

1.6 THE STRUCTURE OF THE THESIS

The rest of this thesis is structured as follows.

Chapter 2 - Background and related work. It presents the theoretical background describing the main concepts and methods we adopted. In this chapter, we present the systematic review and thematic synthesis methods. We highlighted these two methods because there are few systematic reviews based on thematic synthesis in SE. We also present the code smell concept as the central concept of this thesis, and the god class concept as the central concept of the family of controlled experiments that we carried out. Lastly, we present studies related to our systematic review and the family of controlled experiments. We did not find works addressing the gap between theory and practical experimentation on the code smell effect, as we are proposing. Thus, the empirical studies presented in this chapter outline the subject of this thesis.

Chapter 3 - A thematic synthesis on the code smell effect. It presents our SR, following recommendations by Kitchenham and Charters (2007). We describe the review protocol defining the research questions, data sources, search terms, inclusion/exclusion criteria and the quality evaluation checklist adopted in the selection of the primary studies. Then, we present the filtering and data extraction process. Lastly, we show the results, a discussion about them and the threats to validity of the review.

Chapter 4 - A family of controlled experiments. It presents the family of controlled experiments on the human role on the code smell effect. Then, we present the planning and execution of the experiments following the guidelines proposed by Jedlitschka, Ciolkowski and Pfahl (2008). Besides detailed description of the experiments, we also discuss the variations in the experimental setup among the experiments, which made possible to control the studied factors. After detailing the setup of the experiments, we present the results and a discussion about them. Finally, we discuss the threats to the validity of the experiments.

Chapter 5 - A research agenda. It discusses the main challenges that we identified from our empirical studies. Then, we present and discuss an agenda in order to better steer researchers studying the subject.

Chapter 6 - Conclusion and future works. It presents an overall discussion of the research, summarizing its main ideas, exploring the main findings, and outlining possible future work in the area.

Chapter

2

This chapter overviews background and related work for this thesis. As background topics, the chapter elaborates on the methods adopted in the secondary study that we carried out: systematic review and thematic synthesis. Despite the systematic review method be well internalized by most researchers, there are few thematic synthesis in the area. After that, the chapter presents code smell as the central concept of this thesis. Then, it details the three types of code smells adopted in our family of controlled experiments: god class, large class and brain class. As related work, the chapter presents the central empirical studies outlining the problem defined in this thesis. The studies are classified in Chapter 1 as: i) correlation studies, ii) studies focusing on the human role on smell detection, and iii) tool assessment. Lastly, the chapter details the only secondary study that we found on the subject. This study mapped out the empirical studies on code smells.

BACKGROUND AND RELATED WORK

2.1 SYSTEMATIC REVIEWS AND THEMATIC SYNTHESIS

Systematic reviews and thematic synthesis are the central empirical methods that we adopted to synthesize the current knowledge on the code smell effect. We detail them as follows.

2.1.1 Systematic review

A SR is "a means of identifying, evaluating and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest" (Kitchenham; Charters, 2007). Kitchenham et al. (2010a) showed that SR has increasingly being used by the software engineering community. The method was originally proposed to support evidence-based medicine and it was adapted to software engineering researches (Kitchenham; Charters, 2007). Brereton et al. (2007) discuss the process of adaptation of SR to software engineering.

Wohlin et al. (2012) and Kitchenham and Charters (2007) present the steps for using SR in software engineering. We summarize these steps, as follows:

1. Planning

- *Identification of the need for a review.* A SR originates from an aim to understand the state-of-art in an area.
- *Specifying the research questions.* The research questions steer the researchers in the identification of the primary studies, the extraction of data from the studies, and the analysis.
- *Developing a review protocol.* The review protocol defines the procedures for the SR, involving research questions, search strategy for primary studies, data extraction strategies and others issues. The protocol is continuously adapted and evolved during the SR process.

2. Conducting the review

    - *Identification of search.* The main activity in this step involves specifying search terms and applying them to databases. Other aspects that might be considered are manual searches in journals and proceedings and the use of "snowballing" (Skoglund; Runeson, 2009).
    - *Selection of primary studies.* It is the definition of inclusion/exclusion criteria and selection strategies of selection, which might consider reading of title and abstracts or more thorough analysis of other sections, such as methodology or conclusions.
    - *Study quality assessment.* It is the definition of strategies for quality assessments about the primary studies. Checklists are the most common strategy used nowadays.
    - *Data extraction and monitoring.* It involves the data extraction from the primary studies. If quality assessment data is used for selection of primary studies, then the data extraction form is separated into two parts: one for quality data evaluation, and other for the data of the primary studies.
    - *Data synthesis.* It is the choice and application of the quantitative and/or qualitative analysis methods to summarize and gain insights on the information at hand. There are many methods available in the literature (Cruzes; Dybå, 2011b): thematic analysis, narrative synthesis, comparative analysis, case survey, meta-ethnography, meta-analysis and scoping analysis. Its choice depends on the type of data and the needs of the research.

3. Reporting the review

    - *Reporting and disseminating the review.* It involves writing up the results and circulating them to potential publishing sources. The results must be accordingly reported considering the aim, synthesis methods adopted, and outcomes of the review.
    - *Observing the audience.* An important aspect on reporting reviews is the observation of the audience. For practitioners, different publishing sources

might be used, such as practitioner-oriented journals and magazines, posters or web-pages. For academic audience, the detailed reporting of procedures is required.

2.1.2 Thematic synthesis

As previously discussed, we adopted the thematic synthesis as presented by Cruzes and Dybå (2011a). They define thematic synthesis from the thematic analysis method as follows:

> "Thematic analysis is an approach that is often used for identifying, analyzing, and reporting patterns (themes) within data in primary qualitative research. 'Thematic synthesis' draws on the principles of thematic analysis and identifies the recurring themes or issues from multiple studies, interprets and explains these themes and draws conclusions in systematic reviews."

The thematic synthesis method was originally presented by Thomas and Harden (2008), as one of the methods for thematic analysis. They presented the method in the context of the medical area, and divided it in three stages: "the coding of the text line-by-line; the development of descriptive themes; and the generation of analytical themes". Cruzes and Dybå (2011a) contextualize thematic syntheses for the SE area. As proposed by Thomas and Harden (2008), the main idea is to reduce the level of abstraction (from text to code, and from codes to themes) allowing the analysis of the subject being studied. Cruzes and Dybå (2011a) summarizes the thematic synthesis process as we show in Figure 2.1.

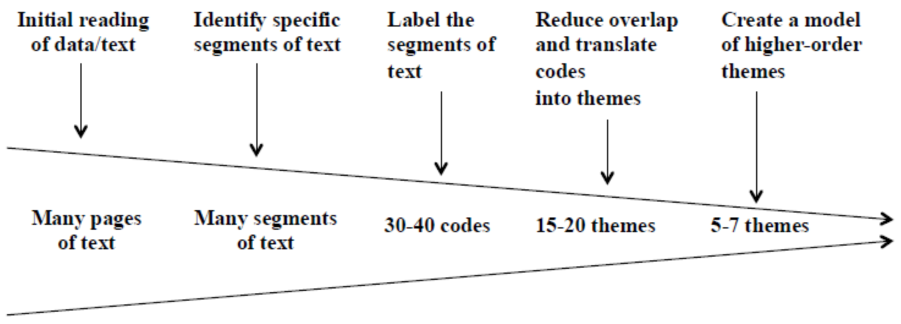

Initial reading of data/text
Identify specific segments of text
Label the segments of text
Reduce overlap and translate codes into themes
Create a model of higher-order themes
Many pages of text
Many segments of text
30-40 codes
15-20 themes
5-7 themes

Figure 2.1 Thematic synthesis process (Cruzes; Dybå, 2011a apud Creswell, 2005)

Then, they present the main steps and checklist items for thematic synthesis in SE. It is expected that the researchers will move among the steps iteratively. We summarize them as follows:

1. *Data extraction*

   It is the definition of a strategy to obtain essential text and data from primary studies. The definition of the strategies must to consider the extraction of different types of data:

- *Publication detail.* Some examples of this type of data are the authors, year, title and abstract They are commonly extracted straightforwardly.
- *Context description.* Some examples of this type of data are aims[1], subjects and technology. The extraction of the context description might not be straightforward. It depends on the paper's presentation.
- *Findings.* The findings are most likely be found in the sections describing the results, the analysis of the results, discussion and conclusions. Cruzes and Dybå (2011a) also consider figures and tables as sources of findings. They present a set of questions helping the identification of parts of the text as a finding. An example of these questions is "does it summarize raw data?".

In the beginning of data extraction phase, it is recommended a reading of the selected papers in order "to be familiar with depth and breadth of the evidence". Another recommendation by Cruzes and Dybå (2011a) is that the data should be extracted independently by two researchers, improving the trustworthiness of the synthesis.

2. *Data coding*

It involves the coding process as widely discussed in qualitative research. Coding is the process of labeling and grouping textual data, according to similar characteristics. The coding enables researchers to organize and classify similar data into categories. The literature presents works discussing the coding process (Saldana, 2012; Boyatzis, 1998).

In thematic syntheses, data coding involves systematic identification and coding of interesting concepts, categories, findings and results. Cruzes and Dybå (2011a) present three strategies to code the textual data extracted in the previous steps:

- *Deductive.* This strategy proposes the definition of a provisional 'start list' of codes. The list is based on the theory, research questions, hypothesis, and other sources. These codes help researchers integrate concepts from the literature.
- *Inductive.* This strategy is in opposition to the deductive strategy. The codes are defined from the textual data. This approach mitigates errors from preconceived conceptualizations.
- *Integrated.* This strategy combines the deductive and inductive strategies. In this case, high order terms are predefined. Then, specific codes are identified inductively. Cruzes and Dybå (2011a) point the integrated strategy as the most used one in SR.

  As in the data extraction phase, it is recommended that the data coding be realized by, at least, two researchers.

[1]Cruzes and Dybå (2011a) classified originally the aim as a publication detail. We understand the aim as part of the context of the primary study.

3. *Translation of codes into themes.*

   It is the definition of themes from the codes. The idea is to group codes to form themes. The themes are proposed to reduce the large amount of codes, that are usually produced during the previous phase. This helps researchers elaborating a thematic map to understand the interactions among the themes. The maps may involve themes, sub-themes and high-order themes, according to the needs.

   Cruzes and Dybå (2011a) recommend the use of visual representations to help the definition or relationships between the themes. Figure 2.2 shows an example of a visual representation for thematic maps. Other types of visualization might be adopted, such as tree-maps or mind-maps (Mazza, 2009; Johnson; Shneiderman, 1991).

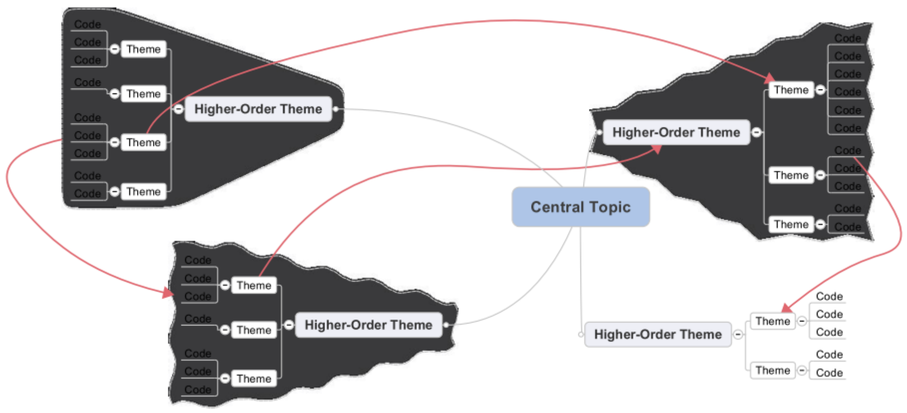


Figure 2.2 Example of thematic map (Cruzes; Dybå, 2011a)

4. *Creation of a model of higher-order themes.*

   It is the exploration of relationships between themes and/or sub-themes. The aim of this step is to return to the research questions, addressing them with arguments grounded on the thematic map identified in the previous phase. The output of this phase can be a high-order description of the themes, a taxonomy, a model or a theory. This is dependent on the type of data and the aims of the research.

5. *Assessing the trustworthiness of the synthesis.*

   It involves a discussion about the trustworthiness of the synthesis. The trustworthiness of a thematic synthesis depends on the quality and quantity of the evidence found. This is related to the primary studies selection and quality, and is also related to the rigor of the methodology adopted for each of the previous steps.

   In qualitative research, some concepts describe different aspects of trustworthiness:

   - *Credibility.* It is related to how well data and process of analysis address the focus of the research. This involves the decisions about the focus of the study and selection of the context, participants and approach to gathering primary studies. Credibility is also related to extraction of suitable snippets of the text, and to how well codes and themes cover the data, avoiding relevant data being excluded, or irrelevant data being included.

- *Confirmability.* It is related to the evaluation of extracted and coded data by more than one researcher. The agreement among the researchers about each type of data is a relevant aspect reinforcing the confirmability attribute of the synthesis.
- *Dependability.* It is related to the stability of the data, considering data change over time, and alterations in the decisions made by the researchers during the synthesis process.
- *Transferability.* It is related to the evaluation of findings being transferred to other related settings. A detailed description of all data and process adopted in the synthesis helps on reinforcing the transferability attribute of a synthesis.

To sum up, the trustworthiness of a thematic synthesis involves establishing arguments for the most probable interpretations. There is not a correct interpretation of the findings, but a most probably interpretation from a particular perspective. Thus, trustworthiness is reinforced offering alternative interpretation for each finding.

2.2 THE CODE SMELL CONCEPT

The main concept we address in this work is code smell. As discussed in Chapter 1, we are using the terms code smell, design flaw and disharmony indistinctly. All these terms were presented to refer to software design problems. Basically, the authors propose strategies for identification of aspects in the code that broke the principles of the object-oriented paradigm. In one of the first books on the topic, Riel (1996) used his experience to discuss common problems observed in OO design. He addressed the misuse of relationships between classes, inheritance, the containment relationships from classes and attributes, and others. After each discussion, he presented heuristics to avoid the problems. For example, related to the misuse of multiple inheritance, one of the Riel suggestions is *"if you have an example of multiple inheritance in your design, assume you have made a mistake and then prove otherwise"*.

Another book in the area was presented by Fowler (1999). He focused on discussions about refactoring. He defined refactoring as *"the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure"*. The main idea is improving the internal structure (design), avoiding future problems, specially on maintenance. He named and presented refactoring techniques, such as "extract method", "move method", "replace data value with object", and many others. Due to the presentation based on step-by-step format, including pieces of code in his examples, in some cases it is possible to apply the techniques automatically.

The most interesting discussion proposed by Fowler is related to *when to apply refactoring*, not on *how to apply refactoring.* Fowler himself classify *how to apply refactoring* as a simple problem and *when apply refactoring* as "not so cut-and-dried" problem. He suggested that until then, a "vague notion of programming aesthetics" had been commonly proposed and he wanted "something a bit more solid". Then, he coined the term code smell to describe potential design problems. He presented 22 bad smells, from an informal discussion. He also recommended the refactoring techniques to eliminate the code smells.

In both books (Riel, 1996) and (Fowler, 1999), the characterization of the code smells is subjective. This was one of the motivation for the work of Lanza and Marinescu (2005). Their strategy for code smell detection is based on metrics and thresholds. To define their rules, they used informal definitions, such as the ones presented by Riel and Fowler. From them, Lanza and Marinescu identified metrics that could be affected by the code smell. Then, they defined the thresholds and finally, they composed metrics and thresholds in code smell detection rules.

For example, the code smell god class refers to classes that tend to centralize the intelligence of the systems (Riel, 1996). Lanza and Marinescu elected the metrics ATFD (Access To Foreign Data), WMC (Weighted Method Count) and TCC (Tight Class Cohesion)[2] to represent a god class. The composition of these metrics is presented in the Figure 2.3. The figure shows that a class is a god class if it has ATFD $>$ *FEW*, WMC $\geq$ *VERY HIGH* and TCC $<$ *ONE THIRD*. The thresholds values *FEW* and *VERY HIGH* are previously defined according to other attributes of the system, such as size. If the metrics and thresholds are well defined, then the code smell can be automatically detected.

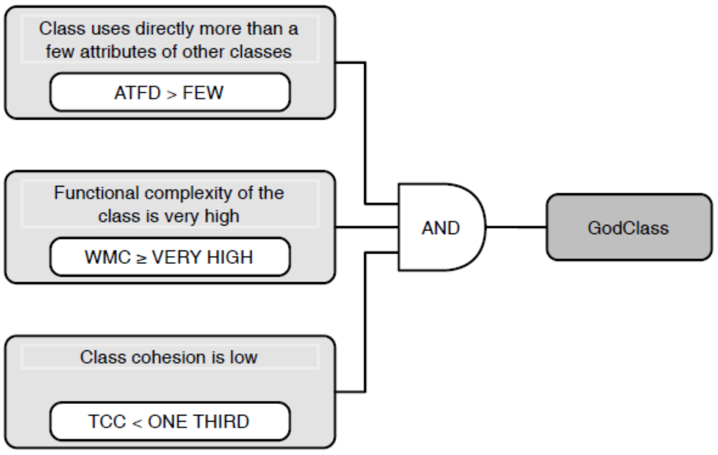

Class uses directly more than a few attributes of other classes
ATFD > FEW
Functional complexity of the class is very high
WMC ≥ VERY HIGH
Class cohesion is low
TCC < ONE THIRD
AND
GodClass

Figure 2.3 Detection strategy for god class (Lanza; Marinescu, 2005)

The Lanza and Marinescu work is the basis for many researches and tools focused on automatic detection of smells.

2.2.1 The god class, brain class and large class code smells

The term god class was coined by Riel (1996) to refer classes that tend to centralize the intelligence of the system. Since then, god class has been addressed in different

[2]We do not detail the metrics because it out of scope of our work.

empirical studies (Padilha et al., 2013; Abbes et al., 2011; Olbrich; Cruzes; Sjøberg, 2010; Li; Shatnawi, 2007). Riel presented god class as a problem of system intelligence poorly distributed. The problem manifests itself in the behavioral form when developers "attempt to capture the central control mechanism so prevalent in the action-oriented paradigm within their object-oriented design". The heuristics proposed by Reil to avoid the god class smell are:

- "Top-level classes in a design should share work uniformly. [...]"
- "Beware of classes with much non-communicative behavior. [...]"
- "Beware of classes that access directly data from other classes."

Fowler (1999) does not use the term god class to describe code smells. However, he presented a code smell with similar characteristics. He defined the large class code smell as a class that tries to do too much. For him, a "class with too much code is prime breeding ground for duplicated code, chaos, and death". He proposes the use of the refactoring techniques *extract class*, *extract subclass* and *extract interface* to solve the problem.

Lanza and Marinescu (2005) proposed a heuristic for god class detection, see Figure 2.3. They based it on the definition of Riel (1996) and explicitly declared that the concept is comparable to Fowler's large class smell. The authors also presented the code smell brain class in a similar way: complex classes that tend to accumulate an excessive amount of intelligence. The main difference is that a god class accesses directly many attributes from other classes.

God class, large class and brain class have a similar concept (Mäntylä; Lassenius, 2006; Lanza; Marinescu, 2005). Schumacher et al. (2010) summarizes the general idea of these three concepts presenting a set of support questions:

- Does the class have more than one responsibility?
- Does the class have functionality that would fit better into other classes?
    - By looking at the methods, could one ask: "Is this the class' job?"
- Do you have problems summarizing the class' responsibility in one sentence?
- Would splitting up the class improve the overall design?

2.3 EMPIRICAL STUDIES ON CODE SMELL

We did not find a specific work discussing the code smell effect considering both the empirical observations and the human role perspectives. As previously discussed, we will present a set of empirical studies outlining the subject. We consider that correlation studies and studies addressing the human role on smell detection outline well the code smell effect. However, we will also present some studies on tool assessments, because we adopted one of these tools in our family of controlled experiments. Lastly, we will detail the only secondary study that we found mapping out the experiments on code smell.

2.3.1 Correlation studies

Sjøberg et al. (2013) presented a controlled study where six professionals were hired to maintain four systems during 14 days. All systems were developed based on the same specification, as a result of a tender sent by the Simula Labs of Norway. It was a medium-size web-based information system to keep track of their empirical studies. Their aim was to quantify the relationship between code smells and maintenance effort. They focused on 12 code smells and how they affected maintenance. The heuristics for smell detection adopted were the ones presented by Lanza and Marinescu (2005) and by Marinescu (2002). One of the main findings of the study was that *"the 12 smells appear to be superfluous for explaining maintenance effort"*. In this context, if we consider that one of the main effects of bad design is the impact on maintenance activities, the presence of code smells might not represent bad design. The authors declared that "*the present focus in research and industry on 'bad design' as operationalized by code smells may be misdirected*".

Macia et al. (2012) assessed the usefulness of automated code smell detection for uncovering architecture modularity problems. To do this, they carried out an exploratory study to analyze the influences of code anomaly on architectural designs in 38 versions of five applications from heterogeneous domains. They used support tools to identify architecture modularity problems and to identify code smells. The heuristics used for code smell detection were the ones presented by Lanza and Marinescu (2005) and by Marinescu (2004). As a result, they suggested that many of the code smells detected were not related to architectural problems. "Even worse, over 50% of the anomalies not observed by the employed techniques (false negatives) were found to be correlated with architectural problems". This can be an indicative that developers may have neglected code smells that are critical to architectural design.

Olbrich et al. (2009) investigated the evolution of two smells (*god class* and *shotgun surgery*) from two open-source systems historical data, adopting strategies presented by Lanza and Marinescu (2005). They identified different phases in the evolution of code smells during the system evolution. Their strategy of analysis considered the increase or decrease of components infected by code smells, in different points in time. They found that components infected by code smells exhibit a different change behavior with respect to the change frequency. They propose the use of this information in the identification of risk areas within a software system that need refactoring.

In another study, Olbrich, Cruzes and Sjøberg (2010) investigated the influence of two smells (*god class* and *brain class*) on the frequency of defects. They analyzed historical data, adopting the Lanza and Marinescu (2005) strategies, from three open-source software systems. They found that, without taking the class size into account, there is a higher change frequency, change size, and defects rate in *god* and *brain classes* than in other classes. However, when these smells were normalized with respect to size, they had smaller values for change frequency, change size, and weighted defect rate than had other classes. Based on these findings, they conclude that, in specific cases, the presence of these smells might be beneficial to a software system.

Li and Shatnawi (2007) studied the relationship between six smells and error probability. The smells addressed were *data class*, *god class*, *god method*, *refused bequest*, *shotgun surgery* and *feature envy.* The smell detection heuristics adopted was proposed by Marinescu (2002) and Gronback (2003). Li and Shatnawi (2007) investigated three error-severity levels, in an industrial-strength open source system, the Eclipse Project. They noted that less than 10% of the classes had code smells. They also found that the *shotgun surgery*, *god class* and *god methods* are positively associated with the probability of errors.

2.3.2 Human role on smell detection

Palomba et al. (2014) investigated if what developers believe to be a problem is actually a problem. They manually identified instances of 12 smells in three open source systems. Then, they sent a questionnaire to potential participants, asking about code snippets affected and not affected by the smells. In the case of a positive answer, the participant was asked to explain the problem and assess its level of severity. Three types of participants were considered in the study: i) graduate students; ii) industrial developers and; iii) developers of the systems themselves. They received answers from 15 graduate students, nine industrial developers, and ten of the original developers. Their findings show that some smells are generally not perceived as a design problem: *class data should be private*, *middle man*, *long parameter list*, *lazy class*, and *inappropriate intimacy.* They also noted "instances of a bad smell may or may not represent a problem based on the 'intensity' of the problem". Another finding was that developers consider large/complex source code as an important threat. Finally, they noted that developer experience and system's knowledge play an important role on smell detection.

Moonen and Yamashita (2012) performed a case study from the same in vivo setting presented by Sjøberg et al. (2013) (previously discussed) and by Yamashita and Counsell (2013). In this case study, the authors performed a qualitative analysis based on interviews conducted daily. They analyzed the perception of participants on the maintainability of the systems and related it back to code smells. The findings fed another empirical study performed with experts and using the same set of systems, which was presented by Anda (2007). They identified thirteen factors affecting maintainability according to participants, factors such as *appropriate technical platform* and *simplicity.* The authors noted the difficulty of using code smell definitions for analyzing certain maintainability factors. For example, the smells *coherent naming* and *design suited to the problem domain* would need additional techniques to be assessed. In summary, they found that eight of the factors affecting maintainability are addressable by current code smell definitions. However, in most cases these code smells would need to be complemented with alternative approaches, such as semantic analysis and manual inspection, in order to help on identification of maintainability factors.

Mäntylä (2005) presented results of two experiments addressing agreement in smell detection and factors to explain it. A small application in Java, with nine classes and 1000 LOC was created and used in both experiments. In the first experiment, there were three questions about the *long method*, *long parameter list* and *feature envy* smells, and one question asked if the method should be refactored to remove the detected smells. In the second experiment, participants were asked only if some specific methods should be refactored. He found high agreement for simple code smells - *long method* and *long parameter list* - and weaker agreement concerning the *feature envy* code smell and refactoring decisions. Then, Mäntylä tried to identify factors impacting agreement on smell detection. He investigated the influence of software metrics and demographic data, such as years of experience of evaluators. He found good correlation between the metric *lines of code* and the smell *long method*; and the metric *number of parameters* and the smell *long parameter list*. However, metrics were not useful to explain evaluation of *feature envy* and refactoring decisions. Finally, he found low correlation between refactoring decisions and demographic variables, such as developer's years of experience.

Mäntylä and Lassenius (2006) investigated why and when people think a code needs refactoring. They analyzed one of the experiments presented by Mäntylä (2005) to investigate what issues in code define the refactoring decisions. They refer to these issues as drivers. They applied a questionnaire to understand refactoring decisions. The most important driver was the size of a method. One of their important findings was that there was a conflict of opinions among the participants with respect both to the assessed internal quality of the methods and the need to refactor them. They also found that some drivers are difficult or impossible to be detected automatically, and some code smells are better detected by experienced participants than automatically.

Schumacher et al. (2010) built on and extended Mäntylä and Lassenius (2006) work. They investigated the way professional software developers detect the *god class* code smell. Then, they compared these results to automatic classification. The study was done in a professional environment, using the results of two real projects and two participants from each project. The research questions focused on "Evaluation of Human Performance" and "Evaluation of Automatic Classifiers". Participants were introduced to the concept of *god class* in a short presentation and were asked to detect them in specific code pieces. During this task, they received a list of questions to help with the identification of god classes. We would eventually adopt the same support questions in our experiments (see Section 2.2.1). A process was designed to ensure all participants performing the inspection of classes in a similar fashion. To evaluate the participant performance on the task, they used a think-aloud protocol (recorded as audio) and data collection forms. Coding was carried out to identify drivers. Answers from the data collection forms were used to evaluate time and agreement. Their main findings were: (1) there was low agreement among participants detecting god class and (2) *misplaced method* was the strongest driver for *god class* detection. Related to the evaluation of automatic detection, their main findings were: (1) an automated metric-based pre-selection decreases the effort needed for manual code inspections and (2) automatic detection followed by manual review increases the overall confidence.

Mäntylä and Lassenius (2006) and Mäntylä, Vanhanen and Lassenius (2004) investigated agreement and the impact of demographic data, such as experience of developers, on smell detection by humans. Moreover, they compared results of human evaluation with metric-based heuristics. They defined the heuristics by themselves because researches on heuristics for smell detection were in their infancy at the time. Through a survey, they asked participants about 23 smells and used a scale from one (lack) to seven (large presence) to evaluate the presence of smells in a piece of code. They received 12 completed questionnaires out of 18 sent to developers of a small software company. They found that, some demographic variables, such as developer experience, partly explain the variation. Related to the correlation of human evaluation and the metric-based heuristics, they analyzed only four smells: *large class*, *long method*, *long parameter list*, and *duplicate code*. For *long method* and *long parameter list*, human evaluation correlated well with the metrics. For *large class* and *duplicate code*, they did not perceive the correlation as well. We emphasize that the last two findings highlight the impact of human perception on the practical adoption of the code smell concept on software development. In one of the findings the authors declare: "*the use of smells for code evaluation purposes is hard due to conflicting perceptions of different evaluators*". The authors also consider that "*the use of the smells for internal software quality assessment should be questioned*".

2.3.3 Tool assessment

Most of the code smell studies use some sort of support tool. Some of them use automatic detection tools (Sjøberg et al., 2013; Macia et al., 2012; Olbrich; Cruzes; Sjøberg, 2010; Schumacher et al., 2010; Olbrich et al., 2009; Li; Shatnawi, 2007; Mäntylä; Lassenius, 2006). Others use visualization tools to support smell detection (Murphy-Hill; Black, 2010; Carneiro et al., 2010; Parnin; Görg; Nnadi, 2008; Simon; Steinbruckner; Lewerentz, 2001). Our experiments used the later approach, so we will briefly discuss two works that influenced us. Murphy-Hill and Black (2010) presented a visualization implemented as an Eclipse plug-in. The tool is composed of sectors in a semicircle on the right-hand side of the editor pane, called petals: each petal corresponds to a smell. They performed a controlled experiment with 12 participants (six programmers and six students) to evaluate the tool. Their main findings were: i) programmers identify more smells using the tool than not using the tool ii) smells are subjective and iii) the tool helps in deciding what is and what is not a code smell. Carneiro et al. (2010) presented the SourceMiner tool, a multi-perspective visualization environment implemented as an Eclipse plug-in. SourceMiner has visualizations addressing inheritance and coupling characteristics of the software. The visualizations also portray previously mapped concerns of the analyzed software. The authors performed an exploratory study with five developers using the concern mapping multi-perspective approach to identify code smells. Two main findings are presented. First, the concern visualizations provided useful support to identify *god class* and *divergent change* smells. Second, the authors were able to identify strategies for smell detection, supported by the multiple concern views.

2.3.4 The Zhang, Hall and Baddoo (2011) secondary study on code smell

The only secondary study that we found on the subject is a mapping study written by Zhang, Hall and Baddoo (2011). One of their motivations was the absence of knowledge about the effectiveness of the code smell concept to target code improvement, which is similar to our main motivation to develop this thesis. They reviewed 319 papers from 2000 to 2009 and analyzed 39 of those in detail and explored four research questions:

1. *Which Code Bad Smells have attracted the most research attention?* For this ques-

tion, the answer was *duplicated code*. They found this code smell in 21 papers. They also noted that some smells have received little attention from researchers.

2. *What are the aims of studies on Code Bad Smells?* They identified that most studies have focus on identifying code smells as opposite as to investigate their impact. As the study addressed broad aspects of code smell, they selected papers addressing topics as wide as tools, refactoring, and heuristics for code smell detection. Due to this, they do not explore any of those topics in depth.

3. *What methods have been used to study Code Bad Smells?* They suggested that the focus has been on objective studies and there are few subjective studies. They consider this a gap in code smell research.

4. *What evidence is there that Code Bad Smells indicate problems in code?* Their findings indicate that the impact of code smell is not well understood. They noted that researchers concentrate on investigating how to identify Code Bad Smells, rather than examining their impact in practice.

Despite similar aims, the Zhang, Hall and Baddoo (2011) and our SR have different characteristics. While Zhang, Hall and Baddoo (2011) carried out a systematic mapping study, performing a broad, but not so deeply analysis, we did the opposite in our SR: a more focused and deep analysis on code smell effects. We highlight differences on the data sources, search terms and analysis strategy. Zhang, Hall and Baddoo (2011) adopted specific journals as sources. The only search engine that they used was IEEE Xplorer. We adopted two well-known publishers (IEEE Xplorer is one of them) and two well-known indexers. Related to search terms, Zhang, Hall and Baddoo (2011) limited to the concept of code smell. They did not adopt similar terms, such as "anomaly", "design flaw" or "disharmony" as we did. However, they adopted names of some specific smells, which we did not adopt. Another difference on search terms is that we considered only empirical studies focusing on smell effect. The analysis strategy was also different. Once Zhang, Hall and Baddoo (2011) aimed to present a broad mapping of the area, they performed a mapping study based on basic quantitative data that they needed for the mapping. In our case, as we focused on a deeper discussion of the smell effect, we performed a thematic synthesis based on qualitative data. We detail our study protocol in Chapter 3.

Chapter
3

This chapter presents a thematic synthesis on the smell effect. The synthesis is a result of a broad secondary study of the papers published over the last 12 years by the SE community. The chapter presents the review protocol, data collection and filtering processes, results, discussions and threats to the validity of the thematic synthesis.

A THEMATIC SYNTHESIS ON THE CODE SMELL EFFECT

Thematic synthesis is the method we adopted for our systematic review on the code smell effect. Due to this, we first present all steps as proposed by Kitchenham and Charters (2007) for performing systematic reviews and mapping studies. Then we present the thematic map and a discussion based on the findings of primary studies as the central results of the thematic synthesis.

3.1 REVIEW PROTOCOL

In this section, we present the definitions of the research questions, data sources, search terms, inclusion/exclusion criteria and quality assessments, for the identification of relevant primary studies of our systematic review (SR).

3.1.1 Research questions

As previously discussed, our SR focuses on synthesizing the current knowledge on how smell concept has been empirically investigated. We defined the following research questions in order to steer our research:

1. What aspects (themes) are studied on smell effect investigations?

2. How limited is the context of studies investigating smell effect?

3. Are the findings converging?

3.1.2 Data sources, search terms and range of years

We adopted electronic data sources (EDS) in our SR. The EDS and search terms were defined by an iterative process, refining terms related to our aims. We also considered the range of years and the volume of papers retrieved in different EDS. Both choice of EDS and definition of search terms were based on a previous ad-hoc literature review, and on our experience in the topics. The process consisted of three meetings with four researchers in the area.

In the first meeting, we discussed results of a preliminary search, considering two ranges: from 2006 to 2013, and from 2002 to 2013. As a result of the discussion, terms and EDS were added/removed. We discussed the new results in a second meeting. In it, we decided, for example, by the use of *"code anomaly"* and *"design anomaly"* terms, instead *anomaly*, in order to minimize the number of empirical studies from other engineering, such as chemistry and electronic. We also decided by the use of the term *controlled*, instead *"controlled experiment"*, because it was broader, more inclusive, but did not significantly expanded the volume of correlated material retrieved by the search engines. In the third meeting, we evaluated the new searches, refining some minor points and defining the final set of EDS, the search terms and range of years for our SR.

The final range of years we adopted was from 2002 to 2013, considering twelve years as a reasonable range of years to cover the discussions about smell effect. We detail the choices of EDS and search terms bellow.

3.1.2.1 Data sources Secondary studies in software engineering have adopted an extensive set of EDS, such as IEEE Xplore, ACM Digital library (ACM DL), Springer Link, Google Scholar, ISI Web of Science, Science Direct, Scopus, Kluwer Online, among others. In order to guarantee a broad range of potential primary studies, we selected EDS well accepted by the software engineering community, considering coverage, advanced features and exportability. From the meetings described previously, we selected two well-known publishers[1] (IEEE Explore and ACM DL); and two important indexers[2] (Scopus and Science Direct).

Using the advanced resources from each EDS, the search was initially done on the paper's meta data (title, abstract and keywords). This decision was based on our experience, volume of papers retrieved and recommendations presented by Dieste and Padua (2007). The IEEE Xplorer has resources allowing filtering by type of source. We filtered by journal and conferences. In the ACM DL, we filtered the search by journal, proceedings and transactions. Finally, in Socpus and Science Direct, we filtered the results by knowledge sub-area. In Science Direct, we chose Computer Science and Engineering. In Scopus, we adopted same sub-areas, adding the Multidisciplinary sub-area.

3.1.2.2 Search terms We considered terms from two main topics. At first, we searched by studies describing potential problems in the design of software. As pre-

[1]http://ieeexplore.ieee.org/Xplore/home.jsp and http://dl.acm.org/
[2]http://www.scopus.com/ and http://www.sciencedirect.com/

viously discussed, important books coined the well-known terms *smell*, *design flaw* and *disharmony*. We also considered the use of the *anomaly* term, which is used in some works in the area (Macia et al., 2012, 2012, 2011; Macia; Garcia; Staa, 2011; Carneiro et al., 2010; Dhambri; Sahraoui; Poulin, 2008). The other main topic that we considered was related to the definition of empirical methods adopted by the experimental software engineering community. We considered the methods listed by Easterbrook et al. (2008) as the main empirical methods adopted in software engineering. From the Easterbrook et al. work, we added methods such as *action research* and *ethnography* in our search.

We considered specific syntax of each EDS to compose terms (for some EDS we had to specify the plural of the terms, in others it was not necessary). A summary of the final terms using quotes and connectives *AND* and *OR* is:

(smell OR "design flaw" OR disharmony OR "code anomaly" OR "design anomaly")

AND

(experiment OR empirical OR survey OR ethnography OR "action research" OR "exploratory analysis" OR study OR controlled)

Table 3.1 shows the number studies retrieved by each EDS.

Table 3.1 Number of studies retrieved by the EDS

Years	**IEEE Xplore**	**ACM DL**	**ScienceDirect**	**Scopus**
2002- 2013	1079	71	70	910

3.1.3 Inclusion/exclusion criteria

To reduce bias all selection and quality evaluation was done by two researchers. In case of doubts, at least, one different researcher evaluated the study and after discussion, a consensus about the inclusion of the paper in the SR was reached. We searched for papers presenting empirical studies focused on smell effect, which are correlation studies or studies evaluating human aspects on smell detection. As the inclusion criteria, we only included papers:

- *written in English*,
- *from journal or conferences.* We did not consider gray literature, such as technical reports or PhD thesis,
- *from software engineering.* Once we adopted some generic terms, we noticed that many papers were related to other fields of engineering on science. For example, it was common to find papers addressing smells, in the sense of odor.
- *addressing smell effect.* An example of a papers removed because it was not in this classification was a paper investigating component modularity (Yu; Ramaswamy, 2009), where the authors only declared to be related to smell, but they did not investigate the smell effect.

The exclusion criteria were refined during all research. First, we analyzed a sample of the retrieved papers in order to align the perception of all researchers involved in the SR. Then, we iteratively refined the exclusion criteria. The main difficulty was related to papers addressing smells, but not the smell effect. We removed a paper if it was:

- *focused on evaluation of automatic detection heuristics.* Their findings were usually related to the quality of their heuristics in comparison with other strategy of smell detection.
- *focused on refactoring issues.* Some papers evaluated refactoring tools or strategies. We also found papers investigating types of refactoring being used.
- *focused on visualization tools.* In these cases, the findings point to effectiveness of the tools. Due to this, we considered them out of scope of our SR.
- *focused on metrics for smells.* Also in these cases, the findings are not related to smell effect, but to the quality of metrics.
- *a short paper.* This exclusion criterion was proposed after in-depth readings. We considered that the constraint of space makes difficult a consistent presentation of empirical results. We defined short paper as a paper with less than five pages.
- *not focused on smells for code design.* We found papers using the term smell, but not related to code design. In two cases, we found papers (Macia et al., 2011; Macia; Garcia; Staa, 2011) addressing aspect smell; and in one case, we found a paper addressing lexicon smell (Abebe et al., 2011). Once the nature of these smells is different, we disregarded these studies from our SR.
- *a summary of other studies.* We defined this exclusion criterion because we found one paper summarizing findings of other five studies based on the same experimental setup (Yamashita, 2013). Once we accepted the other five works as primary studies of our SR, we removed the one that summarizes the others. We discuss this issue later on.

3.1.4 Quality evaluation

Following principles of good practices for conducting SR (Kitchenham; Charters, 2007), we defined quality assessments of the primary studies according to our aims. As we were interested in the current empirical knowledge on smell effect, we had to consider a wide variety of context attributes, evaluating different empirical methods and setups. Due to this, we considered the quality on presentation of the aim, experimental setup, results, discussions and limitation of the potential primary studies of our SR. We adapted our quality criteria from Sjøberg et al. (2005) and Jedlitschka, Ciolkowski and Pfahl (2008) works. Both works mapped the main elements on reporting controlled experiments.

To align the researchers' evaluation, we defined a checklist based on Dybå and Dingsøyr (2008), Kitchenham et al. (2010b) and Nguyen-Duc, Cruzes and Conradi (2015) works, as set of questions that we considered relevant to describe the study. Table 3.2 shows these questions. Each researcher considered all these questions in the quality evaluation of the studies.

Table 3.2 Quality evaluation (adapted from Kitchenham et al. (2010b), Nguyen-Duc, Cruzes and Conradi (2015))

Problem statement
Q1. Is the research objective sufficiently explained and well-motivated?
Research design
Q2. Are the sample and experimental units described?
Q3. Is the design of the experiment described?
Data collection
Q4. Are the data collection procedures and measures described?
Data analysis
Q5. Are the data analysis procedures defined?
Q6. Are the results of the study adequately described?
Conclusion
Q7. Are the findings of study clearly stated?
Q8. Does the paper discuss limitations or validity?

3.2 FILTERING AND DATA EXTRACTION PROCESSES

The reliability of the filtering and data extraction processes was acquired by pair-review and group discussion, in case of disagreement. For all cases, the consensus was reached by the group discussion, which occurred in all phases of the filtering process.

3.2.1 The filtering process

The filtering process was supported by *StArt*[3], a support tool for SR (Hernandes et al., 2012; Fabbri et al., 2012). The tool helped us in all phases of the filtering process. We saved the retrieved results for each EDS in *.bibtex* format. Then, we imported the files in the *StArt* tool. The total number of retrieved material was 2130 papers. During all the filtering process, each study was reviewed by two researchers, independently. Four reviewers were involved in this phase. If some disagreement happened, the paper remained until the in-depth reading phase. Figure 3.1 shows the different phases of the filtering process and the final number of studies for each phase. We discuss each of them bellow.

3.2.1.1 Removal of unrelated material From the 2130 retrieved material, we found a set of unrelated material, which we removed. In the most cases, the unrelated material was made up of abstracts and conferences' proceedings, but we also removed some Master and PhD thesis, and some editorial documents. This was the only activity performed by one researcher, because the observation of these characteristics was very simple. After removing all unrelated material, there were 1727 papers remaining.

[3]http://lapes.dc.ufscar.br/tools/start_tool

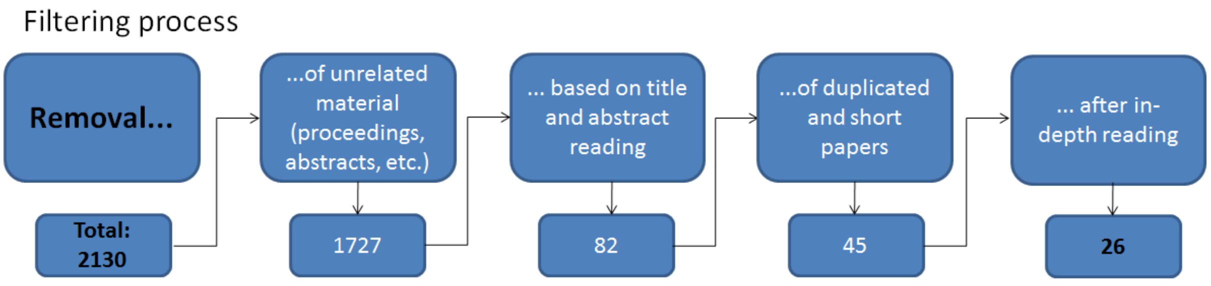

Filtering process
Removal...
Total: 2130
...of unrelated material (proceedings, abstracts, etc.)
1727
... based on title and abstract reading
82
...of duplicated and short papers
45
... after in-depth reading
26

Figure 3.1 Filtering process

3.2.1.2 Removal based on title and abstract reading Before starting the removal based on title and abstract reading, we randomly sampled 50 papers, in order to harmonize the exclusions criteria among the reviewers. All researchers performed the activity for these papers. Then, we met to discuss all 50 papers. Despite the small number of disagreements (only two out of 50), we were able to refine the exclusions criteria.

Then, the other papers were independently evaluated by two researchers. During this phase, shallow reading and group discussion were used to mitigate doubts, which were mainly related to paper addressing smell, but not its effect on software development. For example, we found papers addressing automatic detection of smells from different perspectives: i) the papers adopted automatic detection heuristics in order to establish some type of correlation between smells and software attributes; or ii) the papers propose and evaluate an automatic detection heuristics, presenting findings related to the heuristics. For us, the former perspective is related to the smell effect, while the latter is not. Even mitigating doubts during this phase, there were 98 disagreements, representing 5.8%, from the total of 1677 (1727 original - 50 training) papers. After this phase, there were 82 papers remaining.

3.2.1.3 Removal of duplicated and short papers In this phase, we downloaded all 82 papers. Once the EDS use different special characters, some papers were duplicated. They had the same title, but they were written in a different way, specially for cases where colon, or quotation marks were used. In this phase, we also decided to remove short papers, because we noted that the empirical description was insufficient for our aims: we considered unfair to consider findings of complete and short papers in the same way. At the end of this phase, there were 45 papers remaining.

3.2.1.4 Removal after in-depth reading During the in-depth reading, there were a lot of group discussions, specially related to doubts about findings on the smell effect, as previously discussed. The group discussions were very useful to align the view of the type of papers we were interested in, and on the final adjust in the exclusion criteria. After in-depth reading, we chose the 26 primary studies to be considered in our thematic analysis.

3.2.2 Data extraction

We followed two main paths in the data extraction process. The first was the extraction of context attributes, which are mainly quantitative data. The second was the extraction of the primary studies' findings, which were the basis of the thematic synthesis. In this last case, it is textual information, i.e., qualitative data. Figure 3.2 shows the type of data that we extracted. We discuss them bellow.

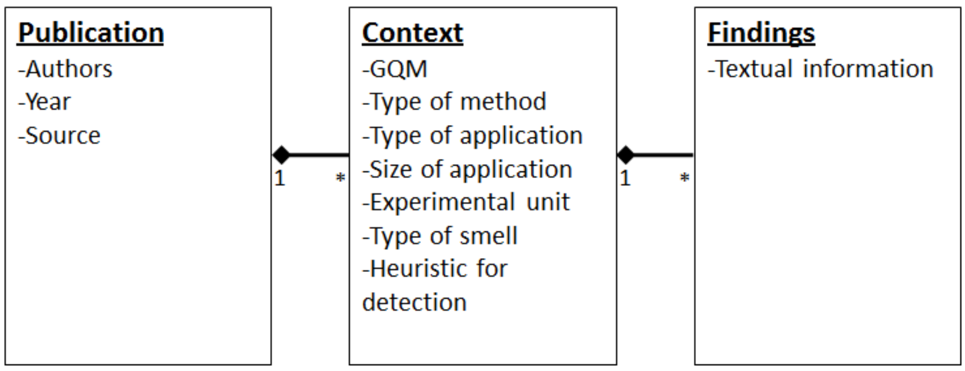


Figure 3.2 Data extraction (adapted from Cruzes and Dybå (2011a))

3.2.2.1 Context attributes The data extraction of context attributes was done by one researcher and checked by another researcher. To make the checking process simpler, the researcher extracting the data marked the document, inserting comments in parts of the text where he/she found the information. In order to maintain the data, we developed a simple software based on MVC (model-view-controller) architecture. The software is composed by input-views and a DBMS (database management system), supporting SQL querying.

We consider the two left boxes in Figure 3.2 as the relevant context attributes for our thematic analysis. The first box describes authors, year and source as the demographic data.

The other elements are detailed in the middle box in Figure 3.2. The first data type is GQM. We adapted the GQM (Goal/Question/Metrics) template (Basili V.R., 1994) to capture the description of the paper aim. The GQM template includes elements to be filled in as shown below.

Analyze < ... >
for the purpose of < ... >
with respect to their < ... >
from the point of view of the < ... >
in the context of < ... >

We only extracted the three first elements (*analyze* < ... >*, for the purpose of* < ... > *and with respect to their* < ... >). The other context attributes that we used to characterize the studies are detailed in Table 3.3. For sake of clarity, we present in this table the values that we found for these data.

Table 3.3 Data extracted from the context attributes

Data type	**Values**
Treatments	correlation over software repositories, code inspection activity, maintenance activity, developer knowledge
Type of method	correlation studies, controlled experiment (in-vivo), controlled experiment (in-vitro) and survey
Type of application	open source systems, commercial, constructed and unspecified
Size	small, medium, large and unknown
Experimental unit	undergraduate, graduate, professional and unknown
Type of smell	from known catalogs, such as Fowler (1999) and Lanza and Marinescu (2005)
Heuristic for detection	free text for description of the heuristic or the tool

3.2.2.2 From primary studies' findings to a thematic map In this section, we describe the process adopted for production of the thematic map from the primary studies' findings. Two researchers were involved in this phase. A third researcher participated when consensus was not reached, which occurred in few cases. During all extraction and production of the thematic map, we used double-checking. One of the researchers extracted and coded the data. A second researcher checked the results, returning comments and suggestions to the first researcher, who re-checked the results using the comments and suggestions. If they disagreed, they met to discuss. If some doubt persisted, a third researcher was consulted.

The process ran iteratively, avoiding extensive volume of data and aligning the researchers' ideas about extraction of useful information for the synthesis. In fact, the learning acquired in the iterations strengthened the data extraction process, coding and making of the thematic map. There were five iterations. In the first iteration, we chose eight primary studies we had read in depth and which were published in relevant sources. It was the most time-consuming iteration because it involved learning and harmonization of the procedures related to the thematic synthesis. For the other iterations, we sampled studies randomly.

For each iteration, we used a spreadsheet to maintain the data. At first, we copied textual parts of the studies where we found some finding. We focused on the sections "abstract", "introduction", "result", "discussion" and "conclusion", but we read the whole paper. We copied textual information containing sufficient information of context to make clear the findings of the studies. In some cases, it was necessary to copy whole paragraphs. Below, we show a simple example for the study with ID S13 (we will present the studies and their IDs in Table 3.5). Bellow, we show part of the sections "Abstract" and "Study results" of the S13 study, highlighting the most relevant information:

> **Abstract**: "The outcome of our evaluation suggests that ***many of the code anomalies detected by the employed strategies were not related to architectural problems***. Even worse, over 50% of the anomalies not observed by the employed techniques (false negatives) were found to be correlated with architectural problems."

> **Study results**: "In general, our analysis reveals that ***detection strategies are inaccurate in identifying architecturally-relevant code anomalies***. Specifically, ***most of the automatically-detected code anomalies were not associated with architectural modularity problems, leading to many false***

> ***positives***. In general, the average of the automatically-detected code anomalies represented about 45% (or less) of the total number of code anomalies related to architectural modularity problems.
> ....
> Even worse, many of the code anomalies harmful to architectural modularity problems were not automatically detected by strategies, leading to a high rate of false negatives. About 55% or more of the non automatically-detected code anomalies were related to architectural modularity problems. These results indicate that ***detection strategies seem to have a tendency to send developers in wrong directions when addressing code anomalies related to architectural modularity problem***."

From the textual information (on the right of Figure 3.2), we extracted what we considered relevant as the findings of the studies. Thus, we had the extraction of findings of the studies. After that, we coded the findings, summarizing them in a sentence. Table 3.4 shows an example of extraction and codes for the S13 study. The left column shows the findings that we extracted from the text above. The right column shows the code and the number of the related-finding, inside parentheses. For the sake of simplicity, we used an example where the coding was very simple, because we adopted the same sentence written in the study. In some cases, the coding required making a new sentence summarizing the finding.

Table 3.4 Example of extraction and coding of findings for the S13 study

Finding's extraction	Code (extraction number)
1 - many of the code anomalies detected by the employed strategies were not related to architectural problems 2 - detection strategies are inaccurate in identifying architecturally-relevant code anomalies	1 - detection strategies are inaccurate in identifying architecturally-relevant code anomalies (1,2)
3 - most of the automatically-detected code anomalies were not associated with architectural modularity problems, leading to many false positives 4 - detection strategies seem to have a tendency to send developers in wrong directions when addressing code anomalies related to architectural modularity problem	2 - the detection strategies seem to have a tendency to send developers in wrong directions when addressing code anomalies related to architectural modularity problem (3,4)

We defined the thematic map for our SR grouping codes according to themes and sub-themes that we identified. For the example in Table 3.4, we classified the codes 1 and 2 into the theme "correlation with issues of development", sub-theme "architectural quality", and into the theme "detection", sub-theme "smell as a predictor". We present the complete thematic map in Appendix A and the main results of our thematic synthesis in Section 3.3.3.

3.3 RESULTS

In this section, we present the results of our SR. First, we present the primary studies. Then, we present the demographic data, the contextual characterization of the studies and the thematic map.

Table 3.5 presents the primary studies that we selected in our SR. The table shows the ID, bibliographic reference and title. It is important to note that some different studies present results as part of the same experimental setup. The most evident case happened for the S4, S6, S7, S8 and S9 studies. The same happened for the S22 and S23 studies. For these two cases, the studies have the same experimental setup and, at least, one common author. Other studies have similar, but not identical, set of object software. They are the S1 and S15 studies, and the S19 and S20 studies. We considered all these cases as independent studies because they present findings from different perspectives on smell effect, despite same (or similar) experimental setup.

Table 3.5 The SLR's primary studies

ID	Reference	Title
S1	(Olbrich et al., 2009)	The evolution and impact of code smells: A case study of two open source systems
S2	(Santos; Mendonça; Silva, 2013)	An exploratory study to investigate the impact of conceptualization in god class detection
S3	(Schumacher et al., 2010)	Building empirical support for automated code smell detection
S4	(Yamashita; Moonen, 2013b)	Exploring the Impact of Inter-smell Relations on Software Maintainability: An Empirical Study
S5	(Zazworka et al., 2011)	Investigating the Impact of Design Debt on Software Quality
S6	(Yamashita; Counsell, 2013)	Code smells as system-level indicators of maintainability: An empirical study
S7	(Yamashita; Moonen, 2013c)	To what extent can maintenance problems be predicted by code smell detection? An empirical study
S8	(Sjøberg et al., 2013)	Quantifying the Effect of Code Smells on Maintenance Effort
S9	(Moonen; Yamashita, 2012)	Do Code Smells Reflect Important Maintainability Aspects?
S10	(Rahman; Bird; Devanbu, 2012)	Clones: what is that smell?
S11	(Macia et al., 2012)	On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms
S12	(Peters; Zaidman, 2012)	Evaluating the Lifespan of Code Smells Using Software Repository Mining
S13	(Macia et al., 2012)	Are Automatically-detected Code Anomalies Relevant to Architectural Modularity?: An Exploratory Analysis of Evolving Systems
S14	(Marinescu; Marinescu, 2011)	Are the clients of flawed classes (also) defect prone?
S15	(Olbrich; Cruzes; Sjøberg, 2010)	Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems
S16	(Carneiro et al., 2010)	Identifying code smells with multiple concern views
S17	(Chatzigeorgiou; Manakos, 2010)	Investigating the Evolution of Bad Smells in Object-Oriented Code
S18	(D'Ambros; Bacchelli; Lanza, 2010)	On the Impact of Design Flaws on Software Defects
S19	(Khomh; Penta; Guéhéneuc, 2009)	An Exploratory Study of the Impact of Code Smells on Software Change-proneness
S20	(Vauche et al., 2009)	Tracking Design Smells: Lessons from a Study of God Classes
S21	(Li; Shatnawi, 2007)	An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution
S22	(Mäntylä; Lassenius, 2006)	Subjective evaluation of software evolvability using code smells: An empirical study
S23	(Mäntylä; Vanhanen; Lassenius, 2004)	Bad Smells Humans as Code Critics
S24	(Mäntylä, 2005)	An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement
S25	(Yamashita; Moonen, 2013a)	Do developers care about code smells? An exploratory survey
S26	(Fontana et al., 2013)	Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains

3.3.1 Demographic

We considered source, year and authors as the demographic data.

3.3.1.1 Source's distribution Table 3.6 shows the distribution of the studies by sources. The first observation is that most of them were published in conferences. Only six studies were published in journals. ICSM and WCRE were the conferences with higher number of studies (four and three out of 26, respectively). Other conferences with two studies were ESEM and CSMR. Considering the journals, there were two studies in the JSS and EMSE. The other sources had only one study published.

Table 3.6 Distribution of empirical studies focusing on smell effect: 2002-2013

Journal/conference proceeding	Type	Number	Percent
International Conference on Software Maintenance (ICSM)	Conference	4	15.3
Working Conference on Reverse Engineering (WCRE)	Conference	3	11.5
Journal of Systems and Software (JSS)	Journal	2	7.7
International Symposium on Empirical Software Engineering and Measurement (ESEM)	Conference	2	7.7
Empirical Software Engineering (EMSE)	Journal	2	7.7
European Conference on Software Maintenance and Reengineering (CSMR)	Conference	2	7.7
International Symposium on Empirical Software Engineering (ISESE)	Conference	1	3.8
Aspect-Oriented Software Development conference (AOSD)	Conference	1	3.8
International Working Conference on Source Code Analysis Manipulation (SCAM)	Conference	1	3.8
Information and Software Technology (IST)	Journal	1	3.8
Brazilian Symposium on Software Engineering (SBES)	Conference	1	3.8
IEEE Transactions on Software Engineering	Journal	1	3.8
International Conference on the Quality of Information and Communications Technology (QUATIC)	Conference	1	3.8
International Conference on Evaluation and Assessment in Software Engineering (EASE)	Conference	1	3.8
International Conference on Quality Software (QSIC)	Conference	1	3.8
International Conference on Software Engineering (ICSE)	Conference	1	3.8
International Workshop on Managing Technical Debt (MTD)	Conference	1	3.8
Total		26	100

3.3.1.2 Distribution of studies by year Figure 3.3 shows the distribution of the studies by year. The number of studies addressing smell effect increased since 2009. This evidences that researchers have dedicated more attention to the topic. On the other hand, we consider that the number of studies still remains small. The years with higher number of studies are 2010 (six) and 2013 (seven).

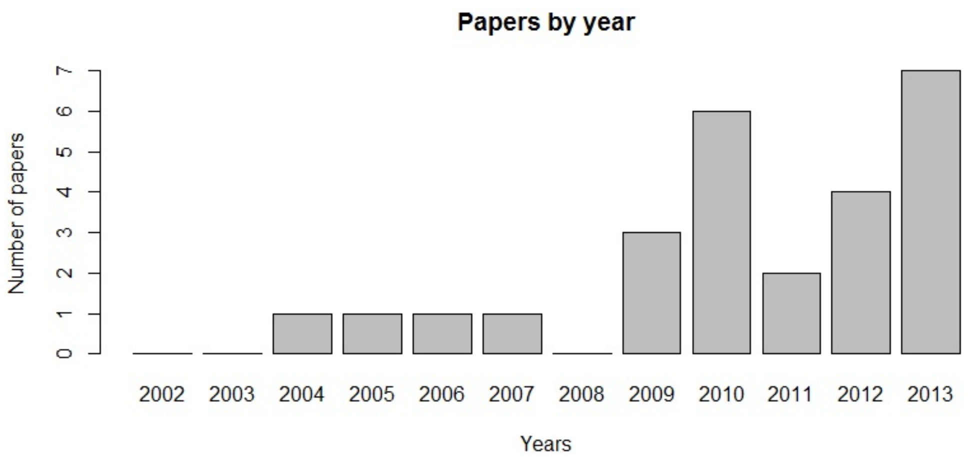


Figure 3.3 Distribution of papers by year

3.3.1.3 Distribution of studies by authors Table 3.7 shows distribution of studies by authors. Some authors appear in different studies based on the same experimental setup. We previously discussed these cases.

Table 3.7 Distribution of papers by authors

Author	Affiliation	Number of papers
Aiko Yamashita	Simula Research Laboratory, Lysaker, Norway	6
Leon Moonen	Simula Research Laboratory, Lysaker, Norway	4
Alessandro Garcia	Informatics Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil	3
Mika V. Mäntylä	Helsinki University of Technology Software Business and Engineering Institute, HUT, Finland	3
Nico Zazworka	Fraunhofer Center College Park, Maryland, United States	3
Arndt von Staa	Informatics Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil	2
Carolyn Seaman	Fraunhofer Center College Park, Maryland, United States	2
Casper Lassenius	Helsinki University of Technology Software Business and Engineering Institute, HUT, Finland	2
Dag Sjøberg	Department of Informatics, University of Oslo, Oslo, Norway	2
Daniela S. Cruzes	IDI, Norwegian University of Science and Technology (NTNU), Trondheim, Norway	2
Forrest Shull	Fraunhofer Center College Park, Maryland, United States	2
Foutse Khomh	Dépt. de Génie Informatique et Logiciel École Polytechnique de Montréal, Montréal, Canada	2
Isela Macia Bertran	Informatics Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil	2
Manoel G. de Mendoncça	Fraunhofer Project Center for Software Systems Eng., Federal University of Bahia, Salvador, Brazil	2
Michele Shaw	Fraunhofer Center College Park, Maryland, United States	2
Steffen Olbrich	Dept. of Computer Science, University of Applied Sciences, Mannheim, Germany	2
Yann-Gaël Guéhéneuc	Dépt. de Génie Informatique et Logiciel École Polytechnique de Montréal, Montréal, Canada	2
Other 40 authors		1

3.3.2 Characterization of the studies

In this section, we mapped the context attributes of the studies. Table 3.8 shows the studies' treatments. For most cases (16 out of 26), smell effect was mainly investigated from software repositories. The total number of studies is 27 because the S3 study investigated software repositories and code inspection activity. The maintenance activity was considered in five studies, but all of them based on the same experimental setup. Therefore, there was only one experimental setup as basis for five studies addressing smell effect from observations of developers performing software maintenance activities. The last observation related to the treatments is that the S25 study is a survey, where questions explored the knowledge of developers. The authors did not ask the participants to analyze any artifact or to perform some activity. Due to this, we classified this as the only study based on the "Developer knowledge".

Table 3.8 Treatments

Treatment	Number of studies
Correlation over software repositories	16
Code inspection activity	5
Maintenance activity	5
Developers knowledge	1
Total	27

Table 3.9 shows the experimental methods adopted, and the studies by method. Most cases (14 out of 26) were correlation studies, which is in accordance with the analysis of software repositories highlighted above. Another observation is related to the controlled experiments

using an in-vivo setting. Five of the six studies were based on the same experimental setup, where authors observed developers performing maintenance activities. It can be argued that the empirical method adopted was *case study*. We classified the method as controlled experiments because the authors controlled relevant factors, such as the definition of the maintenance tasks. This was highlighted in the S7 study.

Table 3.9 Experimental methods

Method	**Number of studies**	**Studies**
Controlled experiment (in-vivo)	6	S3, S4, S6, S7, S8, S9
Controlled experiment (in-vitro)	3	S2, S16, S24
Correlation study	14	S1, S5, S10, S11, S12, S13, S14, S15, S17, S18, S19, S20, S21, S26
Survey	3	S22, S23, S25
Total	26	

Figure 3.4 shows the number of smells by study. The controlled experiments carried out in an in-vitro setting, investigated a small number of smells (≤ 3). They are the S1, S16 and S24 studies. On the other hand, five controlled experiments carried out in an in-vivo setting, investigated a higher number of smells (≥ 12). They are S4, S6, S7, S8 and S9. Note that this does not indicate that the participants had knowledge or were trained about a higher number of smells. Instead, this happened because in the experimental setup, the authors had direct contact with the participants, interviewing them or using think aloud protocol as method of data collection. In the S4 study, for example, the authors automatically identified the smells and asked the participants if they agreed with the automatic results. In the S9 study, they extracted relevant sentences from the interviews and compared with definitions of smells.

The surveys had small (S23 addressed 3 smells) and high (S22 study addressed 22 smells) number of smells being evaluated. The same happened for the 14 correlation studies: the number of smells addressed significantly varied.

The S25 study did not address any specific smell. Due to this, it does not appear in Figure 3.4.

We also mapped the type of smells addressed by the studies. Table 3.10 shows them. As we can see, *god class* and *feature envy* were the most investigated smells: 18 and 16 studies considered them, respectively. Another observation is that there were 63 smells in total, which we consider a high number. We also have to comment that the S13 study considered some aspect smells. We disregarded them.

3.3.3 Thematic map

In this section, we present the thematic map produced in our SR. We grouped the studies' findings in five themes. For each theme, we identified sub-themes. Table 3.11 shows them, presenting an example of the findings, in order to make clear what the themes and sub-themes mean.

Table 3.12 shows the distribution of studies by themes and sub-themes. It is possible to note that the number of studies by theme (column *#studies by theme*) can be lower than the total number of studies by sub-theme (column *#studies by sub-theme*). For example, for the first theme (*correlation with issues of development*), the total number of studies by theme is 17, and the total number of studies by sub-theme is 22 (11+4+3+2+2). This happened because some

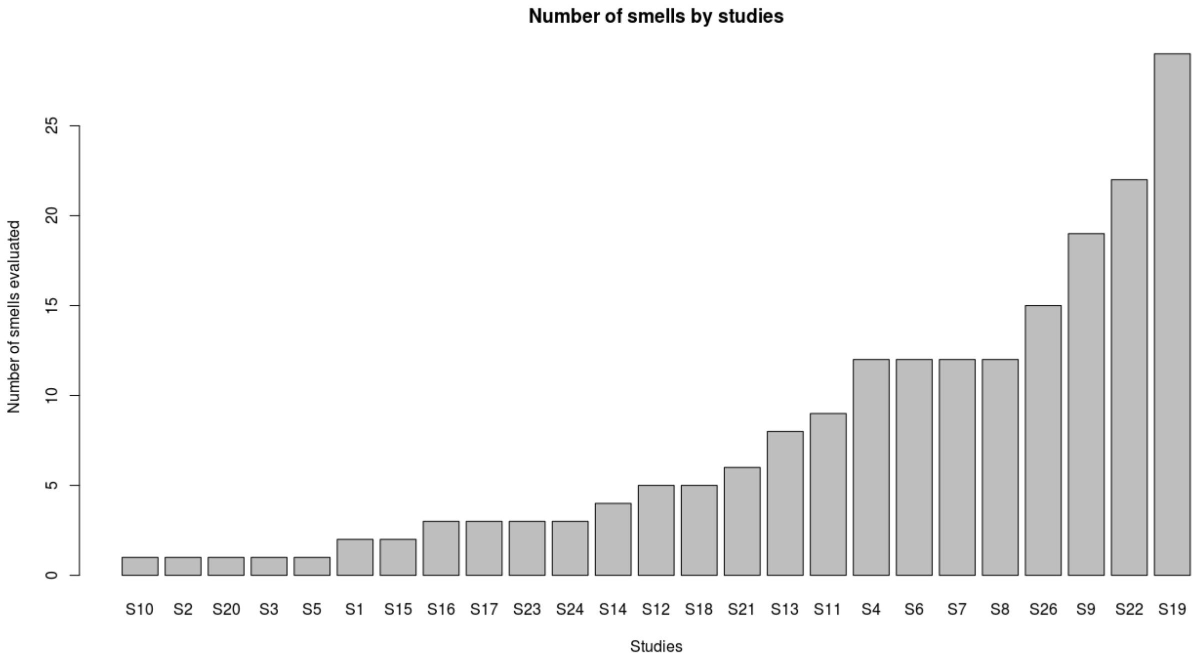


Figure 3.4 Number of types of smells by studies

studies present findings about different sub-themes. We detail the themes and main sub-themes as follows.

3.3.3.1 Correlation with issues of development This was the theme with higher number of studies. There were 17 studies correlating smells and some aspect related to the software development. We identified five sub-themes: *changes and defects*, *effort*, *design quality*, *architectural quality* and *harmfulness*. The most interesting analyses were related to the sub-themes *changes and defects*, *effort* and *architectural quality*.

Changes and defects. By changes, we considered maintenance or refactoring activities. *Changes and defects* was the main sub-theme, with 11 studies. Eight of them are correlation studies. Three of them are controlled experiments carried out in an in-vivo setting, based on the same experimental setup. The studies evaluated medium and large, commercial or open source systems. Due to their focus on evolution, they adopted automatic detection. Only the controlled experiments adopted human evaluation, beyond automatic detection. In total, the studies correlating smells and changes/defects investigated 53 smells. Once the studies are not focused on specific types of smells, it is not possible to generalize their findings.

Despite difficulties on generalizations, it is possible to propose some interesting insights. We found some divergent findings, i.e., reading some studies we tend to consider the adoption of smell concept as an useful practice on software development, but reading other studies we tend to consider the opposite. Table 3.13 shows, side-by-side, some of these divergent findings. On the left column, we show studies and findings indicating which we called as a *positive correlation.* We consider that a study present a positive correlation if its findings tend to point smell as an useful concept to be adopted on software development. For the opposite cases, we called *negative correlations* (the right column in the table).

Table 3.10 Smells addressed

Smell	**Number of papers**	**Percent (%)**
God class	18	28.6
Feature envy	16	25.4
Shotgun surgery	12	19.0
Data Class	10	15.9
Duplicated code	9	14.3
Long Parameter List	9	14.3
Refused Bequest	8	12.7
Long Method	7	11.1
Misplaced Class	7	11.1
Data Clump	6	9.5
God Method	6	9.5
Divergent Change	5	7.9
Interface Segregation Principle Violation	5	7.9
Large class	4	6.3
Message Chains	4	6.3
Temporary variable used for several purposes	4	6.3
Use interface instead of implementation	4	6.3
Brain class	3	4.8
Speculative Generality	3	4.8
Alternative Classes with Different Interfaces	2	3.2
Brain Method	2	3.2
Inappropriate Intimacy	2	3.2
Intensive Coupling	2	3.2
Lazy Class	2	3.2
Switch statements	2	3.2
Temporary Field	2	3.2
Other 37 smells	1	1.6
Total of smells = 63		

Effort. It is the second sub-theme in number of studies addressing correlations between smells and software development attributes. There were four studies addressing this sub-theme. Two of them point to negative correlation between smells and effort on maintenance activities. The S8 study found that *after adjustments for file size and the number of changes, code smells are not significantly associated with increased effort.* The authors consider that file size and the number of changes (code churn) are a better predictor of variations on effort than code smells. In the same direction, the S3 study declares that the smell *god class do not require a higher maintainability effort than non-code smells.* Although we had found a tendency on negative correlations between smells and effort on maintenance activities, we have to consider that the number of studies agreeing on this aspect is low (only two).

The other two studies in this sub-theme addressed other aspects related to effort. The S2 study addressed the impact of software characteristics on god class detection effort. They consider that *the software characteristic does not impact effort on god class detection.* And the S17 study proposes a strategy on applying effort. For their authors, *maintenance effort should prioritize Long Method smells over other smells.*

Architectural quality. This sub-theme also presented some interesting results, despite only two studies had addressed it. We noted that they were consistent. Both S11 and S13 studies identified problems in the detection strategies of smells from the architectural perspective.

The S11 study found that *there is a correlation between code anomaly and architectural degradation.* However, despite they had noted some correlation between smells and architectural degradation, the authors noted that *there is no specific code anomaly more relevant as indicator of architecture degradation.* In the same way, the S13 study also presented a negative correlation between smells and architectural quality. It found that *the detection strategies seem to have*

Table 3.11 Themes and sub-themes

Theme	Sub-theme	Example of findings
Correlation with issues of development	Changes and defects	*Changes of infected components with ISP Violation is significant larger*
	Effort	*God class don't require a higher maintainability effort than non-code smells*
	Design quality	*Code smells often related to immature design and implementation*
	Architectural quality	*There is a correlation between code anomaly and problems in architectural design*
	Harm	*Presence of God class and Brain class may be beneficial to a system*
Human aspects	Factors affecting detection	*Demographics aspects of the developers affect the code smell evaluations*
	Agreement on detection	*The agreement on god class detection is low*
	Human evaluation versus software measures	*The developers' evaluations on code smells do not correlate with related source code metrics*
	Decision drivers	*Misplaced methods is the strongest driver for letting the subjects classify a class as a god class*
	Knowledge about smells	*There is not strong understanding about code smells and practical application of these concepts*
	Strategy on detection	*Participant with optimistic style investigate code smell candidates using more combinations of views when compared to participants with more conservative styles*
	Detection difficulty	*The subjects perceive detecting god classes as an easy task*
Programming	Smell removal	*The developers resolve code smells for opportunistic reasons*
	Smell introduction	*The developers who revised most often also introduced the most smells*
Detection	Supporting smell detection	*Developers need support to enhance detection of code anomaly*
	Smell as a predictor	*Expert judgment is most accurate method for assessing system maintainability*
	Novel smell	*A new smell was propose: Multiple Inheritance simulation*
Other correlations	Metrics versus smells	*Code smells are strongly influenced by size (LOC)*
	Smell density	*Code smell density is likely to be inaccurate when comparing size differing systems*
	Frequency of smells	*Some design flaws are more frequent than others*
	Inter-smell relation	*There is interaction between code smells*

a tendency to send developers in wrong directions when addressing code anomalies related to architectural modularity problem.

3.3.3.2 Human aspects There were seven studies investigating this theme. We identified seven sub-themes being addressed by these studies: *factors affecting detection*, *agreement on detection*, *human evaluation versus software measures*, *decision drivers*, *knowledge about smells*, *strategy of detection* and *detection difficulty.* As previously, we discuss only the main sub-themes.

Factors affecting detection. All studies addressing human aspects presented findings on this sub-theme, and many different aspects of the context were considered. Three studies carried out controlled experiments, three carried out surveys, and one performed a correlation study. The studies used small, medium and large systems; and they used open source, commercial and constructed systems. Four of these studies used professionals as participants subjects, two used students, and the correlation study did not adopted human participants. Excepted by the correlation study, the studies based their analysis on code inspection activity.

Analyzing findings of studies compounding this sub-theme, we also found a tendency on results. It is related to the impact of demographic data on human evaluation of smells. The S23 study found that *demographic data (knowledge, role, work experience) seemed to explain some of the variances in smell evaluations.* This is similar to the finding presented by the S22 study: *demographics aspects of the developers affect the code smell evaluations.* The S2 and

Table 3.12 Distribution of studies by themes and sub-themes

Theme	**Sub-theme**	**#studies by sub-theme**	**#studies by theme**
	Changes and defects	11	
Correlation with issues of development	Effort	4	
	Design quality	3	17
	Architectural quality	2	
	Harm	2	
	Factors affecting detection	7	
	Agreement on detection	6	
Human aspects	Human evaluation versus software measures	2	7
	Decision drivers	1	
	Knowledge about smells	1	
	Strategy of detection	1	
	Detection difficulty	1	
Programming	Smell removal	7	7
	Smell introduction	4	
	Supporting smell detection	6	12
Detection	Smell as a predictor	5	
	Novel smell	1	
	Metrics versus smells	6	
Other correlations	Smell density	3	10
	Frequency of smells	3	
	Inter-smell relation	2	

Table 3.13 Positive versus negative correlations between smells and changes/defects

Positive correlations		**Negative correlations**	
Study	**Finding**	**Study**	**Finding**
S1	*the change proneness of components with smells is higher than the ones without*	S4	*there are smells that are not associated with maintenance problems*
S5	*god classes impacts maintainability*	S7	*the role of code smells on the overall system maintainability is relatively minor*
S9	*there is a correlation between maintainability factors and code smell*	S10	*there is no correlation between clones and bugs*
S19	*classes with code smells are more change-prone than others*	S14	*taken in isolation, classes exhibiting the identity disharmonies design flaws do not have an increased likelihood to exhibit defects that classes which do not reveal design flaws*
S21	*some bad smells are associated significantly with class error proneness*	S18	*design flaws cant be considered more harmful with respect to software defects*

S16 studies found that human evaluation of smells is affected by *human characteristics* and *knowledge acquired in one version.* The S25 study considered the knowledge about smells as a factor affecting human evaluation. For their authors, *the more details people actually know about code smells, the more concerned they are by the presence of smells.* Only one study (S24) did not find strong evidences of the impact of demographic data on human evaluation of smells. It found that *the demographics were not useful predictors neither for evaluating code smells nor the refactoring decision.*

Another factor addressed was software domain. The S26 study considered the software domain as *an important factor that should be included in the detection strategies for various code smells.* It was the only study addressing factors affecting smell detection that was not related to demographic data.

Agreement on detection. Six studies (out of 26) addressed agreement among participants. All of them are contained in the set of studies addressing factors affecting smell detection. Due to this, the characteristics of the studies are very similar: i) four carried out controlled experiments, and two carried out surveys; ii) the studies used small, medium and large systems; iii) they used open source, commercial and constructed systems; iv) four studies had professionals as participants, while the other two studies adopted students as participants; and v) all studies adopted code inspection as the task during the experiments themselves. Excepted for the S22 study, which was a survey asking participants for an extensive variety of smells, all other studies investigated agreement on a small set of smells (three, at the most, for each study).

We found a tendency on findings. Only the S24 study found that *there is partial concordance among the evaluators in all evaluations.* All others found low agreement. This was declared by the S2 and S3 studies. The S16 study found the same idea, but it presented the idea in a different way. The authors declared that *participants had different perceptions and judgments during the code smells identification.* The same happened with the S22 study. Their authors found that *developers have not a uniform opinion on the 'smelliness' of the source code.*

Human evaluation versus software measures. This sub-theme was addressed by S22 and S23 studies, which adopted survey as experimental method. Both studies are based on the same commercial system, with professional participants, including developers of the object software. They found that *developers' evaluations of the smells do not correlate with the used source code metrics.*

3.3.3.3 Programming Seven studies addressed aspects related to smell programing. We identified two sub-themes being addressed by them: *smell removal* and *smell introduction.*

Smell removal. All seven studies addressed this sub-theme. Four of them are correlation studies and three of them are controlled experiments (two of them based on the same experimental setting). They observed maintenance activities, software repositories and code inspection activity. Nineteen smells were addressed by these studies. As for the discussion about the *changes and defects* sub-theme, once the studies are not focused on specific types of smells, it is not possible to generalize their findings.

Despite difficulties generalizing findings, we noted that smells removal is not systematized. Evidences are the findings presented by some of these studies. The S6 study, for example, found that *there is no clarity on the intent of developers prioritize or even consider the eradication of code smells*; the S12 study found that *the developers resolve code smells for opportunistic reasons*; and the S17 study found that *designers perform refactorings routinely based on their subjective perception of problematic code areas rather than applying them as solutions to identified design problems.*

Smell introduction. About introduction of smells we noted that, in some cases, smell removal attempts might actually introduce new smells. This was observed by the S20 study. The authors consider that *the correction of a god class may also move the problem to a different class.* The S8 study found that *the developers who revised most often also introduced the most smells.* The S11 and S17 studies presented discussions about when smells are introduced. The S11 study found that *early code anomalies induces architectural anomalies*, and the S17 study found that *smells are introduced during the initial design/implementation.*

3.3.3.4 Detection Ten studies addressed aspects related to smell detection. We identified three sub-themes on this topic: *smell as a predictor*, *supporting smell detection* and *novel smell.*

Smell as a predictor. Some studies compared strategies of smell detection with other predictors of potential problems in software development. Five studies addressed this sub-theme. Two controlled experiments, two correlation studies and one survey. The two controlled experiments were based on the same experimental setup. Four studies used commercial systems, and one study used open source systems. The size of the systems varied from medium to large. The participants were professionals, excepted for one of the correlation studies, where human evaluation was not considered.

Again, the main problem in order to generalize the knowledge related to the findings of these studies is the high number of smells addressed (48 in total). Moreover, many topics were considered. The S13 study focused on architectural problems (*detection strategies are inaccurate in identifying architecturally relevant code anomalies*); the S6 study addressed the relationship between smell detection strategies and expert judgment, presenting findings from different perspectives, such as *expert judgment is most accurate method for assessing system maintainability* and *code smells approach can detect design flaws that may be overlooked by experts.* For the authors of the S6 study, the combination of software measures and expert judgment can lead to more complete evaluations of maintainability than the smell concept.

Another perspective was presented by the S9 study. The authors consider that code smell *can cover a more heterogeneous spectrum of factors than software metrics and expert judgment individually.* However, they understand that the smell detection needs to be *complemented with alternative approaches, such as semantic analysis and manual inspection, for address maintainability factors.* In the same way, the S19 study considers smells as as an interesting predictor, but not as the most interesting strategy. They declared that *smells can provide to developers recommendations easier to understand than what metric profiles can do*, but *smells are not replacement to metrics in the ability of building change-proneness or fault-proneness prediction models.*

Supporting smell detection. Four studies addressed the adoption of some type of support for smell detection. The context was well defined, considering medium to large commercial and open source systems; and using professional as participants of the studies. All of them investigated the code inspection activity. Two of them carried out controlled experiments, one of them performed a correlation study, and one adopted a survey as empirical method. All them consider relevant, or at least, interesting, the use of some type of complementary support on smell detection. They highlighted definition of a process (S3), visual resources (S16), tools for automatic detection (S11) or source code metrics (S22) as complementary support for smell detection.

3.3.3.5 Other correlations We identified some topics not directly related to smell effect. We classified them in the following sub-themes. *Metrics and smells* (studies correlating smells and software measures); *smell density* (studies discussing the relation smell/line of code); *inter-smell relation*; and *frequency of smells.* For the sake of simplicity, once we consider that these sub-themes do not impact significantly the smell effect, we do not detail them.

3.4 DISCUSSION

In this section, we discuss our main findings based on the results previously presented. It is important to note that, the findings are related to the current knowledge about the smell effect. We expect that, these findings will be affected by new studies on the topics. We present the findings considering each theme of the thematic map as follows.

Correlation with issues of development. Our finding is: *code smell concept does not support evaluation of design quality, in practice.* We believe that there is not strong evidences motivating the practical adoption of the code smells to evaluate the design quality. Our first evidence in order to conclude this are the divergent findings correlating smells and changes and defects. Other evidences are the studies showing no correlation between code smells and effort on maintenance activities. Lastly, we highlight the studies agreeing that there is not a correlation between smell and architectural quality.

Human aspects. Our finding is: *human evaluation of smells should not be trusted.* We found that the human evaluation is diverse and should be better understood. All studies addressing the topic found low agreement on human evaluation of smells. One needs to understand if the cause of the low agreement is an effect of the adoption of different criteria and personal thresholds by the developers. Or if the question is deeper, because it might be related to differences on developers cognitive aspects. We refer to this discussion as the *code smell conceptualization problem.*

We also highlight other evidences showing the human evaluation of smells should not be trusted: i) the studies tend to agree that human evaluation of smells does not correlate with metrics; and ii) they tend to agree that demographic data impacts code smell detection.

Programming. Our finding is: *there is not a systematized process in order to avoid/remove smells.* The evidence is the convergence on findings of studies addressing smell removal and introduction.

Detection. Our finding is: *the current smell detection strategies are not consistent with practical software development problems.* We consider two main evidences for this finding. The first is that the studies on the topic consider that some complement is necessary on smell detection. Besides, the most of studies have shown that the adoption of smells as a predictor can not be trusted, not only considering human evaluation, but also using automatic detection heuristics.

3.5 THREATS TO VALIDITY

In this section, we discuss some factors that might have biased our SR. We highlight the search and selection of the primary studies, and the data extraction and synthesis processes.

3.5.1 Search of primary studies

A common threat of SR is not finding all the relevant studies (Nguyen-Duc; Cruzes; Conradi, 2015; Kitchenham et al., 2010a). We consider two threats in this category. First, the search of relevant studies might be biased by the choice of the EDS. We adopted four EDS, but we understand that this number could be higher. Despite this, we adopted well known EDS, including two relevant indexers, which were Scopus and Science Direct. We based our choices on our experience, discussions among researchers of our research group and other relevant secondary studies. ACM and IEEE, for example, were adopted by all secondary studies that we found in Software Engneering (Dybå; Dingsøyr, 2008; Nguyen-Duc; Cruzes; Conradi, 2015; Kitchenham et al., 2010a; Neto et al., 2011). Scopus and Science Direct were also adopted by, at least, one of these studies. We also selected some relevant studies that we found in our ad-hoc review and checked if they were returned by one of the EDS that we chose. For all cases, the studies were found in, at least, one of the EDS. Due to these aspects, we are confident that our choices were well comprehensive, despite the absence of some EDS.

Another threat related to the search of primary studies is the definition of the search terms. We considered terms related to empirical methods and to the code smell concept. One of the problems is that we did not use names of specific smells. During the group discussions, we understood that, even using specific names of smells in the title and abstract, some contextual discussion linking the type smell with the theory would be necessary. We considered that papers disregarding this link were not focused on the smell effect. This can be noted considering *duplicated code* (or *code clone*), which is the most studied code smell, according to the Zhang, Hall and Baddoo (2011)' mapping study. Some papers (Lin et al., 2014; Choi et al., 2011; Higo et al., 2007) address *duplicated code* as problem independent of the discussions about code smells and they did not use the term *smell* or their synonyms in the title or abstract. Moreover, many papers that we found addressing *duplicated code* in our search also focused on detection and refactoring of the smell (Bian et al., 2013; Choi; Yoshida; Inoue, 2012; Zibran; Roy, 2012; Li; Thompson, 2009). Due to these aspects, we consider this a weak threat of our search. We also checked the search terms looking for them in some studies that we had found in our previous ad-hoc review in order to validate the quality of our search strings.

3.5.2 Selection of primary studies

We have to consider the threat of removing some relevant primary study in the selection phase. To mitigate this bias all studies generating doubts remained until the in-depth reading phase, where, at least, two researchers read and discussed the study. In the in-depth reading phase 19 studies were removed, representing about 42% of the 45 input studies of this phase. One may speculate that many studies were removed, increasing the probability of removal some relevant study. On the other hand, we argue that we maintained a high number of studies generating doubts about its relevance for analysis in the in-depth reading phase.

3.5.3 Data extraction and thematic synthesis

We used double-checking, in order to minimize bias in the data extraction, including in the extraction of qualitative data used in the thematic synthesis. We understand that bias related to the subjectivity is part of qualitative syntheses (Cassell; Symon, 1994), however we mitigated it by checking and re-checking the extraction. All data extraction were done by one researcher, checked by the other and re-checked by the first researcher. When doubts remained, a third researcher was consulted.

3.6 CONCLUSION FROM THE THEMATIC SYNTHESIS

In this chapter, we presented a SR aiming to synthesize the current knowledge about how the smell concept impacts the software development practices, which we called the smell effect. The SR had focus in two types of empirical studies: i) studies correlating smells with some aspect of software development, such as effort on maintenance activities, and ii) studies investigating the human role on smell detection, which are studies investigating agreement, factors and decision drivers affecting humans evaluation of smells. We selected 26 papers as primary studies of our SR.

The SR is based on a thematic synthesis (Cruzes; Dybå, 2011a), which is a technique aiming identifying, analyzing and reporting patterns from qualitative research. In our case, we coded and grouped textual information, which is how findings are mainly presented in empirical studies.

One of the findings of our SR is that the smell concept does not support the evaluation of design quality, in practice. We considered the set of divergent findings correlating smells with issues of software development as an evidence that the area lacks understanding of the smell effect. Another finding is that human evaluation of smells should not be trusted. We state this because we found a tendency on primary studies showing that the agreement on smell detection is low. Moreover, the studies found that demographic data, such as developers experience, significantly impacts smell evaluation. We also highlight that we did not find evidences that the current strategies of smell detection are consistent with practical problems of software development.

Chapter

4

This chapter presents the family of controlled experiments that we carried out in order to face the human role on code smell effect. It overviews the family, describing the variables that we addressed and the variations in the experimental setup among the experiments. Then, it details each experimental setup and presents the results, discussions, and threats to validity of the experiments.

A FAMILY OF CONTROLLED EXPERIMENTS

As showed in the previous chapter, we identified the human role on smell detection as the central confounding factor on empirical studies investigating the smell effect. The family of controlled experiments presented in this chapter investigates an extensive set of factors affecting the human perception of the god class code smell.

4.1 OVERVIEW OF THE FAMILY OF CONTROLLED EXPERIMENTS

We call our family of experiments of *Finding God Classes* (*FinG*). Its definition occurred as an iterative process. Currently, FinG has four controlled experiments. We named FinG 1, the first experiment of FinG; FinG 2, the second experiment, which is a replication of FinG 1; and so on. FinG was iteratively defined as proposed by Mendonça et al. (2008). Each experiment gave us insights to the next experimental setup.

FinG 1, which originates the FinG family, was performed as an exploratory study to acquire experience and insights. It extended and built on the empirical study presented by Schumacher et al. (2010). Schumacher et al. wanted to understand how developers detect the code smell god class. They explored decision drivers and effort during the detection of god classes. They evaluated the agreement among the participants and compared their answers with automatic detection.

The FinG family has focused on the idea of three smells: god class, brain class and large class. As previously discussed, these three smells have a similar concept (Mäntylä; Lassenius, 2006; Lanza; Marinescu, 2005). We will adopt the term god class to refer to this general idea, indistinctly. FinG maintains the focus on god class because of the original study of Schumacher et al. (2010) and because we found other works addressing the same code smell (Padilha et al., 2013; Abbes et al., 2011; Olbrich; Cruzes; Sjøberg, 2010; Li; Shatnawi, 2007) as our SR shows. In this way, we contribute to build on a body of knowledge about this code smell. Moreover, the code smell god class seems suitable to be adopted in controlled experiments: it involves information hiding and cohesion principles of the OO paradigm, focusing on only one class, which minimizes the participant's effort analyzing the object software during the experiments.

Each experiment asks participants to detect god classes in different pieces of source code. We evaluated how a set of variables impacts agreement on smell detection. Figure 4.1 shows the variables evaluated, up to now. We defined agreement on god class detection as the dependent variable (the arrow at the right side in the figure). We will discuss our definitions of agreement later on. We evaluated the impact of the following independent variables on the agreement (the arrows at the left side in the figure): developer experience, knowledge and training, design comprehension tool support, and software size.

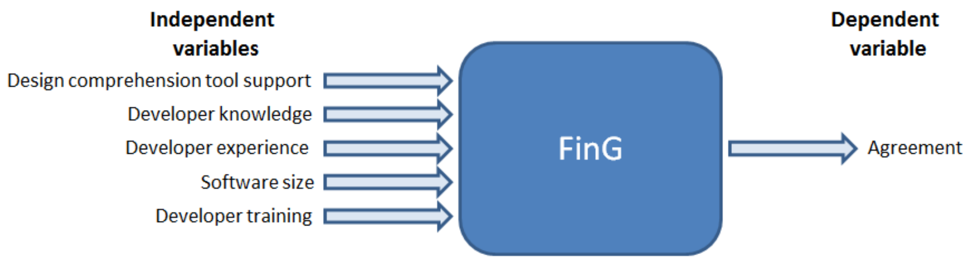


Figure 4.1 Factors affecting god class detection in FinG

The choice of input variables was based on insights gained after the analysis of other empirical works on the subject, or the analysis of the previous experiment in the FinG family. To the best of our knowledge, this is the first study analyzing such extensive set of independent variables with respect to code smell detection agreement. Below, we explain the variables in more detail:

- **Design comprehension tool support**. One of our first ideas to explore factors affecting smell detection was based on the nature of the concept. As the code smell concept is intrinsically related to the quality of design, we decided to investigate the impact of the overall comprehension of the design on human evaluation of smell. To do this, we considered the use of software visualization. We found some studies addressing software visualization tools and code smell detection (Murphy-Hill; Black, 2010; Parnin; Görg; Nnadi, 2008; Emden; Moonen, 2002; Simon; Steinbruckner; Lewerentz, 2001), but their focus was on the comparison of their tools with other tools or methods. None of them focused on discussing how a better comprehension of the design (acquired by the use of visual resources) affects the human perception of a code smell.

- **Developer knowledge**. Dybå, Sjøberg and Cruzes (2012) highlighted individual skills when discussed the importance of context in experimentation. They consider individual skill as a variable that directly might influence behavior or moderate relationships between other variables. We investigated if extensive reading about god classes and code smell would impact on god class detection agreement.

- **Developer experience**. The experience is an important contextual aspect in the software engineering discipline (Höst; Wohlin; Thelin, 2005; Carver et al., 2003; Höst; Regnell; Wohlin, 2000). An evidence of the importance of experience in smell detection is noted in Kreimer (2005)'s statement: "*depending on the perception and experience of the searching engineer, design flaws are interpreted in a different way*". Despite the importance, the topic is lightly addressed in code smell evaluations. Even the Kreimer's work did not explore the impact of the experience on code smell detection. He proposed a method to find code smells based on metrics.

- **Software size**. In FinG 1 and FinG 2, we looked at simple and familiar applications to minimize the effort of participants during the inspection of code. Our argument was that the human evaluation is affected by subjectivity for any type of software. We felt the need, though, to assess if software size and complexity, such important factors, play a role in smell detection agreement. FinG 3 was used to investigate this subject.

- **Developer training**. This variable emerged from the analysis of FinG1, FinG 2 and FinG 3. We found out that inconsistencies on code smell detection were not mitigated by the use of tool support, such as using metrics (without established threshold) or visualization resources. We then hypothesized that, in order to mitigate subjectivity, it is important to teach people about the conceptualization of smells. We decided to use training, based on golden examples of smells and group discussions, as one of the independent variables in our work.

The FinG family also investigated decision drivers and strategies adopted by the participants during the detection of god classes. Once we did not evaluated a cause-effect relationship, we have considered the analysis of these factors simpler than the analysis of the independent variables that we discussed above. Due to this, for sake of simplicity, we do not present the analysis of the decision drivers and participants' strategies on god class detection in this chapter. Instead, we present published papers analyzing these factors in Appendix C (Santos; Mendonça, 2015) and D (Santos; Mendonça, 2014).

4.2 VARIATIONS IN THE EXPERIMENTAL SETUP

Figure 4.2 shows how we controlled factors that we addressed in FinG. Figure 4.2(A) and 4.2(B) shows the variables that were investigated for each experiment independently. In these cases, we used two experiments to strength our analysis of each variable. Figure 4.2(C)-(E) shows the cases where our analysis was based on the comparison among the experiments' results.

As shown in Figure 4.2, factors were investigated as follows:

- **Design comprehension tool support**. Participants detected god classes with and without the use of software visualization tool support, in FinG 1 and FinG 2 (Figure 4.2(A)). Due to this, we evaluated agreement results for both with and without visualization cases, independently, for each experiment.

- **Developer knowledge**. During training, participants received a questionnaire, where they answered conceptual questions about code smell and god class, in FinG 2 and FinG 3 (Figure 4.2(B)). We evaluated the agreement among participants with "good" and "bad" knowledge rating, independently, for each experiment.

- **Developer experience**. The main difference on the experimental setup of FinG 1 and FinG 2 was the experience of the participants (Figure 4.2(C)). While FinG 1 was carried out with undergraduate students, FinG 2 only had graduate students and professionals. We then compared results of FinG 1 against results of FinG 2 to investigate the impact of experience on smell detection agreement.

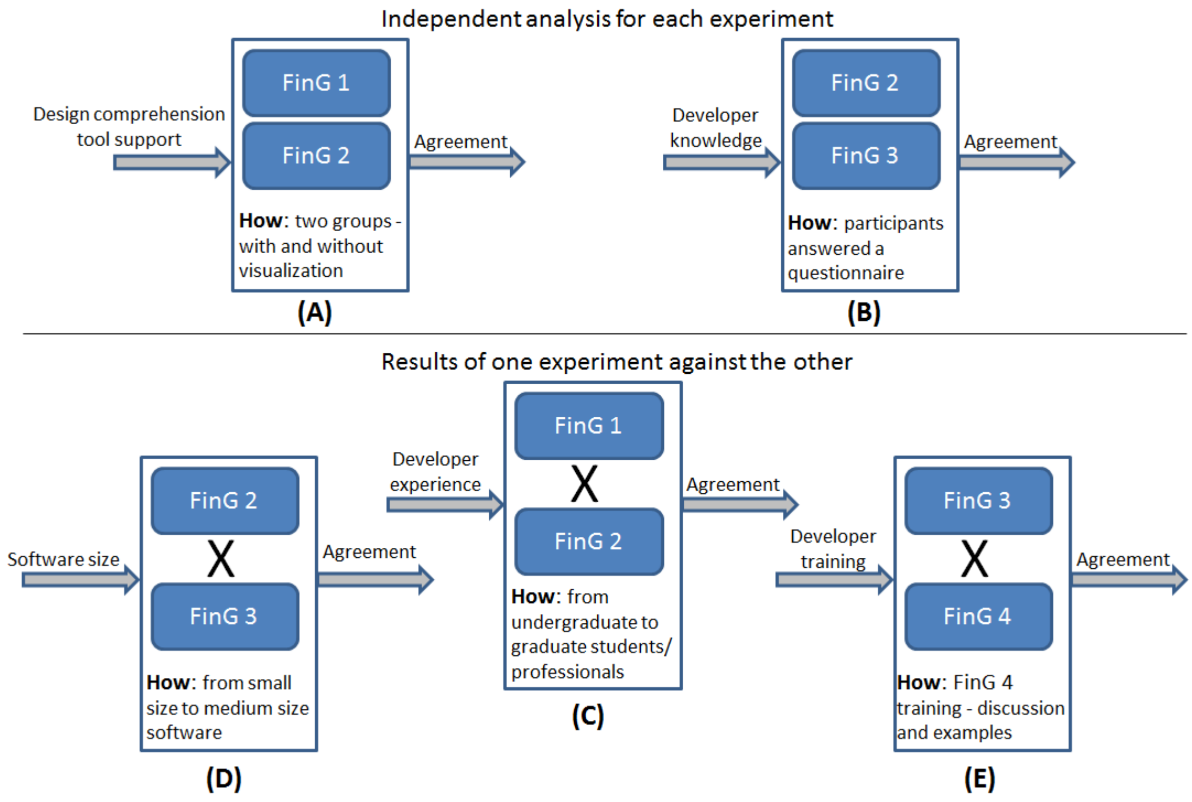

Independent analysis for each experiment
Design comprehension tool support
FinG 1
FinG 2
How: two groups - with and without visualization
Agreement
(A)
Developer knowledge
FinG 2
FinG 3
How: participants answered a questionnaire
Agreement
(B)
Results of one experiment against the other
Software size
FinG 2
X
FinG 3
How: from small size to medium size software
Agreement
(D)
Developer experience
FinG 1
X
FinG 2
How: from undergraduate to graduate students/ professionals
Agreement
(C)
Developer training
FinG 3
X
FinG 4
How: FinG 4 training - discussion and examples
Agreement
(E)

Figure 4.2 Control strategies for establishing cause-effect relationships

- **Software size**. The main difference on the experimental setup of FinG 2 and FinG 3 was the size of the analyzed software (Figure 4.2(D)). We considered line of code (LOC) as the measure of software size. While FinG 2 was carried out with six small programs, FinG 3 used four medium size programs. This relation was also observed in the LOC value for the classes. Considering the classes observed by the participants of FinG 2, the LOC value mean is 93. Considering the classes observed by the participants of FinG 3, the LOC value mean is 684. Thus, we compared results of FinG 2 against results of FinG 3 to investigate the impact of software size on smell detection agreement.

- **Developer training**. The main difference on the experimental setup of FinG 3 and FinG 4 was the training of the participants (Figure 4.2(E)). Participants of FinG 3 just read a set of training material and answered an evaluation questionnaire, while participants of FinG 4 read the training material, had a training lecture with examples and group discussion on smells. We then compared results of FinG 3 against results of FinG 4 to investigate the impact of training on smell detection agreement.

4.2.1 Summarizing experimental variations

Table 4.1 summarizes variations between the experiments. We show changes in the main experimental attributes affecting our analysis: the participants' subjects, the object software adopted, the design of the experiments and the training performed in the experiments.

We have to comment some decisions about our analysis strategy. First, we discuss why we

Table 4.1 Summary of experimental setup of FinG's experiments

Experiment	**Participants**	**Software**	**Design**	**Training**
FinG 1	Undergraduate students	Six small software artifacts	Two groups: with and without visualization	Presentation
FinG 2	Graduate students/ professionals	Six small software artifacts	Two groups: with and without visualization	Reading books and questionnaire
FinG 3	Graduate students/ professionals	Four medium software	Two groups: with and without visualization	Reading books and questionnaire
FinG 4	Graduate students/ professionals	Four medium software	One group: without visualization	Reading and training based on examples

disregarded some experiments during the analysis of the design comprehension tool support, training and software size factors:

- **Design comprehension tool support**. We did not consider the results of FinG 3 in the analysis of the use of design comprehension tool support, despite the tool had been also adopted in FinG 3. We did this because the task procedure would represent a confounding factor in the analysis. While in FinG 1 and FinG 2, the participants had to search god classes looking all software, the participants of FinG 3 had to look at specific classes, because of the size of the software adopted in this experiment.

- **Developer experience**. We did not consider the results of FinG 4 in the analysis of the developer experience factor because the training of FinG 4 aimed to affect the participants comprehension about god class. In FinG 2 and FinG 3, the participants individually read part of books on the topic and decided by themselves how to identify a god class. Due to this, the training of FinG 4 would represent an important confounding factor for this analysis.

- **Software size**. We did not consider the results of FinG 1 and FinG 4 in the analysis of the software size factor because of the participants' profile and the training. In FinG 1, the participants' profile would represent an important confounding factor in the analysis, because they were undergraduate students. And in FinG 4, the training would represent an important confounding factor because it affected the participants comprehension of a god class (we detail the training latter on).

Now, we justify the experiments used in the analysis of design comprehension tool support and training factors, despite differences on the experimental setup.

- **Developer experience**. We compared results of FinG 1 and FinG 2, despite differences on the training. We did this because the presentation performed in FinG 1 was based on the same parts of the books used on the training of FinG 2. Due to this, we consider that the training is not a relevant confounding factor in this analysis. We discuss the bias related to the influence of the presenter in FinG 1 as a threat in Section 4.7.

- **Developer training**. We compared results of FinG 3 and FinG 4, despite differences on design (we did not use a visual support tool in FinG 4). We did this because the participants of both experiments had to analyze the same specific classes, and the advantages gained with the use of the tool were less relevant. We discuss the use of the visual support in Section 4.3.2. Moreover, when we planned FinG 3 and FinG 4, we had already observed that the use of the design comprehension support tool does not impact the agreement among participants detecting god classes (see Section 4.6).

4.3 EXPERIMENTAL PLANNING OF FING FAMILY

We followed the guidelines proposed by Jedlitschka, Ciolkowski and Pfahl (2008) to detail the experiments.

4.3.1 Experimental unit, motivation and reward

All experiments involved undergraduate or graduate students (including professionals) from the Federal University of Bahia (UFBA), in Brazil.

FinG 1. FinG 1 involved 11 junior (3rd year) undergraduate Computer Science students. All students were enrolled in the Software Quality course offered in the first semester of 2012. This is an optional subject of the Computer Science undergrad program, in which design quality and smells are addressed. The course was considered appropriate for the experiment, both because it was focused and was not mandatory, which means that most students enrolled on it were interested in the subject. Furthermore, participation in the experiment was voluntary.

FinG 2, FinG 3 and FinG 4. The other three experiments involved graduate Computer Science students. All students were enrolled in the Experimental Software Engineering course offered in the second semester in 2012 (FinG 2), first semester in 2013 (FinG 3) and second semester in 2013 (FinG 4). There were 25 participants in FinG 2, 25 in FinG 3 and 17 in FinG 4. The participation in the experiments was mandatory. The participants received grades for their participation, but not for their performance. From the analysis of the characterization form (we will discuss the forms later on), we noted that, in FinG 2, all participants were, or had been, software development professionals. In FinG 3, only four participants had no professional software development experience. And, in FinG 4, only two participants had no professional experience.

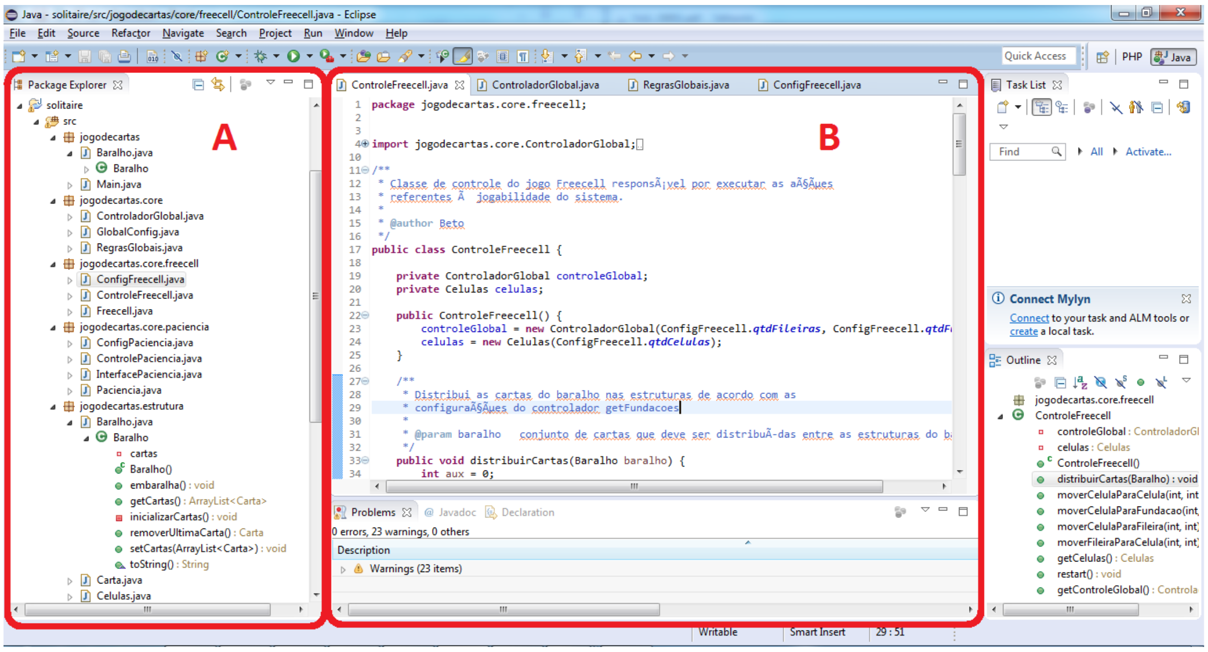
4.3.2 Tools

FinG 1, FinG 2 and FinG 3. We adopted two main software tools[1]: Eclipse, a well-known software IDE; and SourceMiner, an Eclipse plug-in that provides visual resources to enhance software comprehension activities (Carneiro; Mendonça, 2014, 2013; Carneiro et al., 2010).

We adopted SourceMiner because it enhances the comprehension of relevant software attributes on god class detection, in comparison with the traditional Eclipse IDE setup. We detach the observation of size and coupling relations among software classes. In order to observe these relations, developers using Eclipse typically analyze the package-class-method structure of programs, by collapsing and expanding the nodes of the vertical tree into the Package Explorer view (Figure 4.3-A). Then, they open and read classes in the Source Code Editor view (Figure 4.3-B) to identify size and coupling attributes. The two views present problems. Kersten and Murphy (2006) discuss limitations related to the use of Package Explorer for developers navigating between software artifacts in different modules; and Alwis and Murphy (2006) discuss difficulties on software comprehension by reading lines of code.

We argue that these difficulties also happen for small software, which we used in FinG 1

[1]Eclipse IDE -http://www.eclipse.org/downloads/ and SourceMiner - http://www.sourceminer.org/



```
solitaire
  src
    jogodecartas
      Baralho.java
        Baralho
      Main.java
    jogodecartas.core
      ControladorGlobal.java
      GlobalConfig.java
      RegrasGlobais.java
    jogodecartas.core.freecell
      ConfigFreecell.java
      ControleFreecell.java
      Freecell.java
    jogodecartas.core.paciencia
      ConfigPaciencia.java
      ControlePaciencia.java
      InterfacePaciencia.java
      Paciencia.java
    jogodecartas.estrutura
      Baralho.java
        Baralho
          cartas
          Baralho()
          embaralha() : void
          getCartas() : ArrayList<Carta>
          inicializarCartas() : void
          removerUltimaCarta() : Carta
          setCartas(ArrayList<Carta>) : void
          toString() : String
      Carta.java
      Celulas.java
```

```
package jogodecartas.core.freecell;


import jogodecartas.core.ControladorGlobal;

/**
 * Classe de controle do jogo Freecell responsÃ¡vel por executar as aÃ§Ãµes
 * referentes Ã  jogabilidade do sistema.
 *
 * @author Beto
 */
public class ControleFreecell {

    private ControladorGlobal controleGlobal;
    private Celulas celulas;

    public ControleFreecell() {
        controleGlobal = new ControladorGlobal(ConfigFreecell.qtdFileiras, ConfigFreecell.qtdF
        celulas = new Celulas(ConfigFreecell.qtdCelulas);
    }

    /**
     * Distribui as cartas do baralho nas estruturas de acordo com as
     * configuraÃ§Ãµes do controlador getFundacoes
     *
     * @param baralho   conjunto de cartas que deve ser distribuÃ-das entre as estruturas do b
     */
    public void distribuirCartas(Baralho baralho) {
        int aux = 0;
```

```
0 errors, 23 warnings, 0 others
Description
Warnings (23 items)
```

Connect Mylyn
Connect to your task and ALM tools or create a local task.

```
jogodecartas.core.freecell
ControleFreecell
  controleGlobal : ControladorGl
  celulas : Celulas
  ControleFreecell()
  distribuirCartas(Baralho) : void
  moverCelulaParaCelula(int, int
  moverCelulaParaFundacao(int,
  moverCelulaParaFileira(int, int)
  moverFileiraParaCelula(int, int)
  getCelulas() : Celulas
  restart() : void
  getControleGlobal() : Controla
```

Figure 4.3 Traditional Eclipse IDE showing Package Explorer and Source Code Editor views

and FinG 2. In Figure 4.3, we show the Solitaire program. It has 1758 lines of code and 23 classes. Even for this small number of classes, the Package Explorer does not show all packages and classes (Figure 4.3-A) of the program. Another observation is about the number of classes in the Source Code Editor view (Figure 4.3-B). Developers have to navigate between one view for each class in the Source Code Editor to read the classes.

The SourceMiner has five visualizations, divided into two groups. The first group is composed of three software coupling visualizations. They show different types of dependency among entities, like direct access to attributes or method calling, for instance. They also show the direction of the coupling. The coupling views are based on radial graphs (Figure 4.4-A), relationship matrix (Figure 4.4-B), and tabular view (Figure 4.4-C). As an example for comparison with the use of the traditional Eclipse IDE, lets look the view based on radial graphs (Figure 4.4-A). Circles represent classes, and arrows represent dependencies among classes. A tool-tip informing data about the class appears when the cursor is positioned over the class. Moreover, double click opens the code of the class on the Source Code Editor. It is possible to observe coupling attribute for all classes looking the radial graph view (and the other coupling views of SourceMiner). The tool also has interaction resources enhancing comprehension of the code design (Carneiro; Mendonça, 2014; Carneiro et al., 2010). Using the traditional Eclipse IDE setup, it is necessary to read the source code of each class in order to identify coupling.

The second group is composed of two hierarchical visualizations. The Treemap view (Figure 4.4-D) shows the hierarchy of package-class-method of the software. A Treemap is a hierarchical 2D visualization that maps a tree structure into a set of nested rectangles (Johnson; Shneiderman, 1991). In SourceMiner, rectangles representing methods of the same class are drawn together inside the rectangle of the class. Likewise, the rectangles of the classes that belong to the same package are drawn together inside the rectangle of the package. Lines of code (LOC) or cyclomatic complexity are associated to the area and colors of the rectangles. Thus,

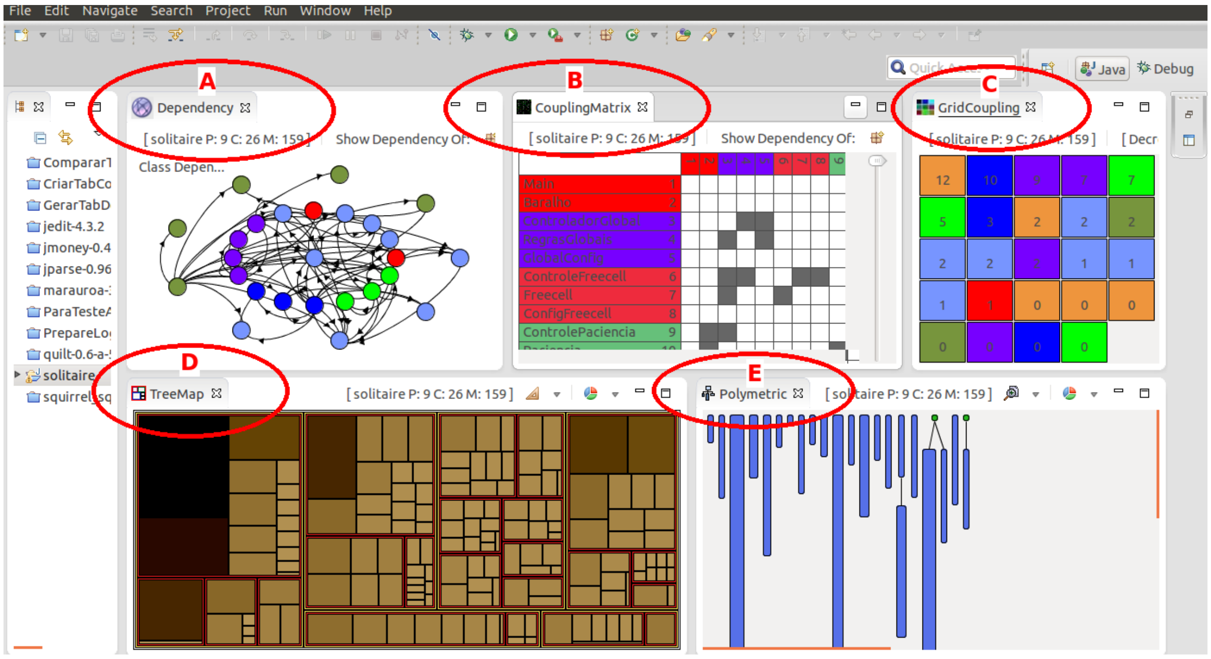


Figure 4.4 Eclipse IDE showing SourceMiner views

it is possible to observe classes with higher LOC value easier than opening and reading the classes, such as using the traditional Eclipse IDE. The Polymetric view (Figure 4.4-E) shows the hierarchy between classes and interfaces. A polymetric view uses a forest of rectangles to represent the inheritance trees formed by classes and interfaces in a software system (Lanza; Ducasse, 2003). In SourceMiner, rectangles are linked by edges representing the inheritance relationship between them. The length and width of the rectangles are used to represent the size and number of methods of a class.

FinG 4. In FinG 4 we adopted only the Eclipse IDE tool. We did not adopt the software visualization infrastructure.

4.3.3 Forms and docs

Five forms were used in total. During the training, participants filled a Consent and a Characterization form. Except for FinG 4, where we did not adopt software visualization, the participants also received a SourceMiner exercise guide. During the experiment itself, they filled in an Answer form where participants had to fill in: i) initial and end time of the each smell detection task, and ii) each candidate god class with a "yes" or "maybe" level of certainty.

At the end of the experiments, the participants filled in a Feedback form. On it, we asked the participants to classify the training and the level of difficulty performing the detection tasks. It was also possible to write down suggestions and observations about the experiment.

Besides these forms, we used two other documents, during experiments themselves. One of them presented an overview of the software artifacts adopted in each experiment. The other was a Support Question guide used to steer the participants in the search for god classes. The questions were the same ones used by Schumacher et al. (2010). We presented them in the very end of Section 2.2.1.

4.3.4 Software artifacts

FinG 1 and FinG 2. Six programs were used. All of them implement simple applications or games in Java. Chess, Tic Tac Toe, Monopoly and Tetris implement well known games. Solitaire-Freecell (Solitaire) is a framework for card games with Solitaire and Freecell. Jackut is a very simple social network application. Table 4.2 characterizes the used programs in terms of number of packages, classes and LOC.

Table 4.2 Software objects for FinG 1 and FinG 2 experiments

Software	**Chess**	**Jackut**	**Tic Tac Toe**	**Monopoly**	**Solitaire**	**Tetris**
Packages	5	8	2	3	6	4
Classes	15	19	5	10	23	16
LOC	1426	978	616	2682	1758	993

FinG 3 and FinG 4. Four programs were used. We adopted more complex software than in FinG 1 and FinG 2. All software also implement familiar Java applications: i) Quilt is a software development tool that measures test coverage; ii) JMoney is a personal finance (accounting) manager; iii) jParse is an Eclipse plugin in which you can parse XML returned from an Ajax request; and iv) SQuirrel SQL Client is a graphical Java program that allows one to view the structure of a JDBC compliant database, browse the data in the tables, and issue SQL commands, among other functions. Table 4.3 characterizes the programs in terms of number of packages, classes and LOC.

Table 4.3 Software objects for FinG 3 and FinG 4 experiments

Software	**Quilt**	**JMoney**	**Squirrel**	**jParse**
Packages	20	4	3	4
Classes	104	79	73	69
LOC	13030	11299	9081	32270

4.3.5 Task

FinG 1 and FinG 2. We asked participants to detect god classes in the software. We provided only a set of Support Questions as a guide. Each participant was free to use her/his own strategy to do the task.

FinG 3 and FinG 4. As in FinG 1/FinG 2, we gave the software and the Support Questions as a guide. However, we did not ask participants to detect god classes on all classes of the software. Instead, we specified 12 candidate classes in each software. We did this because the analysis of the whole software during the experiments was impractical, so we chose the 12 classes with the highest LOC in each software. This number is similar on average to the total number of classes in the small programs used in FinG 1/FinG 2. The participants were free to define their process for detecting god classes.

4.3.6 Design

All experiments were run in a laboratory at UFBA. Participants had about three hours to carry out the task. Each participant worked at an independent workstation.

FinG 1 and FinG 2. The workstations were divided into two groups. In FinG 1, there were six participants in group 1 and five participants in group 2. In FinG 2, there were 13 participants in the group 1 and 11 participants in the group 2. At each workstation, we set up two Eclipse IDEs. We fitted the SourceMiner only for one of the Eclipse installations in the workstation. Each installation had three of the six programs in their workspace. The programs in the workspaces were rotated between the groups. Table 4.4 presents this design.

Table 4.4 Design of FinG 1 and FinG 2

Group	With SourceMiner	Without SourceMiner	Participants' ID (FinG 1)	Participants' ID (FinG 2)
1	Chess, Jackut and Solitaire	Monopoly, Tetris and Tic Tac Toe	14, 21, 32, 35, 42 and 44	1, 3, 4, 7, 9, 11, 13, 15, 17, 19 21, 23, 25 and 27
2	Monopoly, Tetris and Tic Tac Toe	Chess, Jackut and Solitaire	13, 15, 25, 31 and 41	5, 6, 8, 10 12, 14, 16, 18 20, 22 and 26

FinG 3. The only difference in the design of FinG 1/FinG 2 and FinG 3 was the number of programs in the workspace of each Eclipse IDE. In FinG 3, there were two medium sized, instead of three small sized, software in each Eclipse workspace. There were 13 participants in the group 1 and 11 participants in the group 2. Table 4.5 presents this design.

Table 4.5 Design of FinG 3

Group	With SourceMiner	Without SourceMiner	Participants' ID (FinG 3)
1	jParse and Quilt	JMoney and Squirrel	1, 2, 3, 6, 7, 10, 11, 12, 13, 21, 23, 24 and 25
2	JMoney and Squirrel	jParse and Quilt	9, 14, 15, 16, 17, 18, 19, 20, 27, 28, 29

FinG 4. FinG 4 used the same medium sized software as FinG 3, but it had only the basic setup of one Eclipse at each workspace. In this case, each participant always detected god classes without SourceMiner. In all, 17 participants performed the task for the four programs. Table 4.6 shows this design.

Table 4.6 Design of FinG 3

Group	Without SourceMiner	Participants' ID (FinG 4)
1	jParse, Quilt, JMoney and Squirrel	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 and 17

4.3.7 Hypotheses

Our hypotheses were based on the discussion about the independent variables that we addressed.

Design comprehension tool support. Our hypothesis is: *developers detecting god class using visual resources to enhance their overall comprehension of the design agree and succeed (agreeing with oracle) more than developers not using any visual resource.* This hypothesis is based on the fact that code smells represent "bad" design decisions and the visualization tool we adopted enhances the comprehension of the design.

Developer knowledge. The hypothesis is: *developers who have better knowledge on code smell agree and succeed more than the others.* We used a questionnaire, where participants answered simple questions about concepts related to god class to organize the participants in two groups: those with "bad" and those with "good" knowledge on code smell concepts.

Developer experience. The hypothesis is: *more experienced developers agree and succeed more than less experienced developers.* We used the participants of FinG 1 and FinG 2, who had different experience profile, to test this hypothesis.

Software size. Our hypothesis is: *developers agree and succeed more if they are looking at simpler software.* We compared the results of FinG 2 and FinG 3, which had similar design and participant profile, but different software size and complexity, to test this hypothesis.

Developer Training. The hypothesis is: *better trained developers agree and succeed more than the others.* We compared the results of FinG 3 and FinG 4, which had similar participant profile and software size, but different training procedures, to test this hypothesis. Here it is important to observe that FinG 4 did not use the visualization tool in any treatment, while FinG 3 did that for half of its trials. This raises the concern that *Design Comprehension Tool Support*, an important confounding factor, was not considered in the analysis of the *Developer Training* factor. However, as will be presented in Section 4.5.1, FinG 1 and FinG 2 experiments showed that *Design Comprehension Tool Support* does not affect the agreement results.

4.3.8 Procedure

In our analysis, we considered two cases. First, the cases where participants were sure about the god class candidates, i.e., only classes marked with "yes" option. Second, we included the "maybe" cases in the analysis. For sake of simplicity, we only detail the results for the cases where participants were sure about a god class candidate, the "yes" cases. We summarize the statistical tests for the cases where doubts were considered, the "maybe" cases, later in this chapter.

An important aspect of our analysis was the definition of the oracle. The oracle of FinG 1 and FinG 2 was based on two experienced researchers from academia and industry. As the object software had a small number of LOC (varying between 616 and 1758) and well known domains, the researchers were free to use their own approaches to identify the god classes. Each of the researchers did the god class detection task independently, based on their skills and experience. The two researchers then met to discuss their disagreements and to define the final oracle for the experiments. During the definition of the oracle answers, the researchers did not have any access to the answers of the experiments' participants.

Table 4.7 shows the oracle for FinG 1 and FinG 2. According to the oracle, Tic Tac Toe, Jackut and Solitaire do not have any god class. Monopoly has two god class candidates: *Jogo* and *Tabuleiro*. The oracle is certain that *Jogo* is a god class. *Tabuleiro* is marked as a "maybe" god class. For the Chess program, *Chess* is marked as a sure god class, and for the Tetris program, *Tetris* is marked as a "maybe" god class.

Table 4.7 Final oracle of FinG 1 and FinG 2

Program	**Total number of classes**	**God class candidate**	**Level of certainty**
Tic Tac Toe	5	-	-
Monopoly	10	Jogo Tabuleiro	Yes Maybe
Chess	15	Chess	Yes
Tetris	16	Tetris	Maybe
Jackut	19	-	-
Solitaire	23	-	-

The oracle of FinG 3/FinG 4 was defined by three researchers, two of them with years of experience in academia and industry. The other researcher was a graduate student. Due to the size of the programs (number of LOC varying between 9081 and 32270), we defined a strategy based on group discussions. After identifying the twelve largest classes for each of the four programs, the following strategy was adopted to classify them:

1. to evaluate the name of the class, understanding its role on the software;
2. to evaluate the name of each method, understanding their role on the class;
3. for large or suspect methods, reading code;
4. if the method is considered out of scope, then to mark as such;
5. if two methods are considered out of scope, then the class should be classified at least as a "maybe" god class;
6. if more than two methods are considered out of scope, the class is a "yes" god class;
7. if doubts remain, considers size and readability to determine if the class is a god class.

The experimenter (the author of this thesis), which was one of the oracle researchers, defined the god class detection strategy. Alone, he classified the 12 classes selected for each program as a sure or may be god class. The other two researchers were trained on the proposed strategy using an independent set of classes. Together, these two researchers investigated pairwise the 12 selected classes. After that, the all three researchers met to compare and discuss their answers, defining the final oracle for FinG 3 and FinG 4.

As shown in Table 4.8, JMoney program has two "maybe" god class (*AcountEntriesPanel* and *QIF*) and one sure god class (*MainFrame*). JParse has one "maybe" god class (*JavaParser*), and one sure god class (*Type*). Squirrel has only two "maybe" god class (*ConstraintView* and *TableFrameController*). Lastly, the Quilt program did not have any god class candidate.

Table 4.8 Final oracle of FinG 3 and FinG 4

Program	Number of classes investigated	God class candidate	Oracle answer
JMoney	12	AccountEntriesPanel	Maybe
		MainFrame	Yes
		QIF	Maybe
JParse	12	JavaParser	Maybe
		Type	Yes
Squirrel_SQL	12	ConstraintView	Maybe
		TableFrameController	Maybe
Quilt	12	-	-

It is important to note we did not use any automatic detection strategy because all heuristics are based on metrics and thresholds. Once we were interested in investigation of factors affecting the human perception of god classes, we did not provide any metrics values, avoiding to influence the participant answers. Moreover, we adopted the general concept of god class, which involves concepts of god class, brain class and large class, as discussed in Section 2.2, Chapter 2.

4.3.9 Analysis procedure

Our analysis considered two perspectives: agreement among the participants and agreement between each participant and the oracle.

Agreement among the participants. We adopted the Finn coefficient (Finn, 1970), as opposed to the Kappa coefficient (Fleiss et al., 1971), adopted in other works addressing code smells (Yamashita; Counsell, 2013; Fontana; Braione; Zanoni, 2012; Zhang; Hall; Baddoo, 2011; Schumacher et al., 2010). We did this because of problems identified in Kappa coefficient, by data analysis researchers (Powers, 2012; Gwet, 2002; Feinstein; Cicchetti, 1990; Whitehurst, 1984). The Kappa test is done in two phases. First, an agreement rate is calculated and, then, this value is used to calculate the coefficient. Feinstein and Cicchetti (1990) shows that one can have high agreement rate and low values of the Kappa coefficient when the variance on values of raters is low. We noted this situation in the work of Schumacher et al. (2010). The Finn coefficient is recommended when variance between raters is low (Finn, 1970). Whitehurst (1984) suggests Finn as an alternative to problems with Kappa, and affirms that it is the most reasonable index for agreement.

Our analyses were based on the evolution of the agreement level between the different groups analyzed (with and without visualization support, more and less experienced participants, small and medium software size, knowledge level, and training). We adopted classification levels to make it easier the comparison between the agreement values. Like Schumacher et al. (2010) and Zhang, Hall and Baddoo (2011), we used the agreement classification levels defined by Landis and Koch (1977). The classification is as follows: slight, for values between 0.00 and 0.20; fair (between 0.21 and 0.40); moderate (between 0.41 and 0.60); substantial (between 0.61 and 0.80); and almost perfect (between 0.81 and 1.00) agreement level.

A problem occurs with the use of agreement coefficients when there is a high number of items that have a sure negative classification. In these cases, the agreement coefficient tends to be very high, hiding disagreements on the items with possible positive classifications. This occurred in FinG 1 and FinG 2, where most of the classes are small and clearly are not god classes. To mitigate this problem, we considered that the total number of classes to be used in the agreement test was twice the total number of god class candidates. We defined a candidate god class as class chosen by, at least, one participant. If this number was higher than the total number of classes of the software under analysis, we considered the total number of classes of the software. This problem did not occur in FinG 3 and FinG 4, because we selected the biggest 12 classes of medium sized software, most of them good god class candidates.

Agreement between participants and oracle. To evaluate participant success with respect to the oracle, we adopted an accuracy measure. The confusion matrix shown in Table 4.9 captures the idea of our analysis. The top line values represent candidate god classes selected by the participants. The left column values represent candidate god classes indicated by oracle. True positives represent the case where the participant and oracle agreed on a candidate god class. False positives occurred when participants marked a god class in disagreement with the oracle. False negatives occur when the participant did not mark a god class in disagreement with the oracle. Lastly, true negatives are the cases where the participant and the oracle agreed on a negative.

The accuracy measure captures all these cases. The formula is given below. We calculated the accuracy measure for each participant, considering both “yes” or “maybe” and only “yes” cases.

Table 4.9 Confusion matrix

		Predicted (participant)	
		god class	**no god class**
Actual	**god class**	True positive (TP)	False positive (FP)
(oracle)	**no god class**	False negative (FN)	True negative (TN)

$$accuracy = \frac{number\,of : TP + TN}{number\,of : TP + FP + FN + TN} \tag{4.1}$$

After calculate accuracy values, we generated box plots grouping participants' values according to the independent variable addressed. Next, we used the Shapiro-Wilk normality test. For all cases, there was, at least, a group of the samples in which the distribution was not normal. Due to this, we adopted the Mann-Whitney, a non-parametric alternative to t-test, with a 0.05 p-value, to statistically test our hypotheses.

We also evaluated our results in terms of magnitude, testing the effect size measure. We adopted the non-parametric Cliff's Delta test (Cliff, 1996) to evaluate the effect size. In order to assess the magnitude on effect size, we used the classification presented by Romano et al. (2006). The classification is as follows: negligible, for $|d| < 0.147$; small ($|d| < 0.33$); medium ($|d| < 0.474$); and large, otherwise. We accepted Cliff's Delta values on small (or higher) classification as evidence that the effect size also impacted our results. We did this because our samples had a small number of data points, which is common for controlled experiments in software engineering involving humans as participant subjects.

4.4 EXECUTION

4.4.1 Preparation

The standard experimental setup took four days, in a span of three weeks. Two days for training, one day for a pilot and one day for running the experiment itself. In the experiment day, all participants performed the task for all programs, using and not using the visualization tool, when it was permitted. The training was performed in two days due to the quantity of activities. We performed a motivational presentation in the first day of the training. In this presentation, we discussed the experimental software engineering scene and tied it with discussions about code smells. At the end of the first day, we asked the participants to fill out the Consent and Characterization forms. On the second day of training, we focused in the visualization tool. We performed the activity in a lab, introducing SourceMiner and running a practical exercise. The practical exercise focused on the use of the tool and how it could help on perception of differences on size, complexity and coupling among classes. The last activities of the exercise asked the participants to search for god classes using SourceMiner. For these activities, we did not influence the participants on their evaluation.

As discussed earlier, there were variations on the standard setup of the experiments. Table 4.10 shows the schedule and variations for each experiment. In the motivational presentation of FinG 1, we discussed shortly the concepts of code smell, large class, god class and brain class. We did this because the participants were undergraduate and had limited software development experience. For FinG 2, FinG 3 and FinG 4, we did not discuss the god class or code smell concepts. We simply asked the participants to read parts of Fowler (1999) and Lanza and Marinescu (2005)'s books, instead. They received a document explaining which pages of the books they had to read. From Fowler's, they read about refactoring, smells and large classes.

From Lanza and Marinescu's, they read about disharmony, god class, brain methods and brain classes.

Table 4.10 The schedule and variations of the experiments

Day	**Activity**	**FinG**				**Local**	**Time (Hour)**
		1	**2**	**3**	**4**		
1	Motivational presentation	●	●	●	●	Classroom	1.0
	Code smell concept presentation	●				Classroom	0.25
2	Questionnaire after reading		●	●		Lab	0.25
	SourceMiner presentation	●	●	●		Lab	0.5
	SourceMiner exercise	●	●	●		Lab	2.0
	Training based on examples				●	Lab	2.5
3	Pilot	●	●	●	●	Lab	2.5
4	Execution	●	●	●	●	Lab	2.5

In the following week, before starting their training on SourceMiner, participants of Fing 2 and FinG 3 were given a 15 minutes questionnaire. The questions (three in total) addressed concepts of refactoring, large class, god class and brain class. This requirement motivated the participants to read the training material more carefully and it was later used to measure *developer knowledge* (one of our independent variables). FinG 4 training was different. There was no SourceMiner training. Instead, we performed a training based on examples. First, we showed examples of classes with different size, complexity and coupling, discussing why we consider them as candidate or not candidate to be a god class. We based on the process defined in Section 4.3.8. Then, we asked the participants to perform the same activity looking for other classes and identifying if they were god class candidates or not. They answered in a WEB form that we created. In the final part of the training day, we used the WEB form to perform a group discussion about the differences on the answers, aligning the personal thresholds and rigor on the evaluation of god classes.

4.4.2 Deviations and non-conformities

We discarded some data points in each of the first three experiments. FinG 1 started with 17 participants, but we used only 11 data points; FinG 2 started with 28 participants, but we used only 25; and FinG 3 started with 29, but we used only 25. For all cases, the results were discarded because, or the participants performed the pilot, or they missed, at least, one of the experiment's activities. FinG 4 started with 17 participants, all completed the experiment.

Except for FinG4, we performed a pilot with two participants. In FinG 1, the pilot helped us to evaluate the use of the Answer form, in paper or electronic format, and to evaluate the time needed to complete the experiment. In FinG 2, the pilot did not indicate any problem, probably because its setup is very similar to FinG 1. The pilot was useful in FinG 3 because it helped us to adjust the Answer form, which was different from FinG 1 and FinG 2's Answer form, and to evaluate the time, because we asked the participants to analyze 12 classes of four medium size programs (more complex that those used in FinG 1/FinG 2). Finally, the pilot in FinG 4 used two independent researchers. It helped us to evaluate and to perform some adjustments in the training on god class concept, which was based on examples.

4.5 RESULTS

In this section, we present the results of our analyses for all independent variables we addressed.

4.5.1 Design comprehension tool support

To analyze how design comprehension impacts agreement on god class detection, we considered FinG 1 and FinG 2 experiments, independently. We grouped the participants using and not using visualization, because we consider that the visual resources of SourceMiner enhance overall comprehension of the program design (Carneiro; Mendonça, 2013; Carneiro et al., 2010).

4.5.1.1 Agreement among the participants In Table 4.11, we show agreement coefficient values for each program, considering both with and without visualization cases. First column shows the software. Next, there is a group of columns showing total of classes considered in the agreement test, number of participants, the Finn coefficient value and the p-value, for the cases where visualization was not used. The last group of columns present the same fields, but considering the cases where visualization was used.

Table 4.11 Finn agreement test for FinG 1, grouped by the use of visualization tool

	Without visualization				**With visualization**			
Program	**#class**	**#part**	**Finn**	**p-value**	**#class**	**#part**	**Finn**	**p-value**
Monopoly	4	6	0.567	0.0135	2	5	0.4	0.221
Tetris	4	6	0.567	0.0135	6	5	0.467	0.0307
Tic Tac Toe	4	6	0.567	0.0135	2	5	0.6	0.0788
Chess	6	5	0.467	0.0307	6	6	0.6	0.0014
Jackut	6	4	0.5	0.0403	6	6	0.667	0.000226
Solitaire	4	5	0.4	0.113	8	6	0.617	0.000147

Table 4.12 presents the same structure of Table 4.11, but considering results of FinG 2 experiment.

Table 4.12 Finn agreement test for FinG 2, grouped by the use of visualization tool

	Without visualization				**With visualization**			
Program	**#class**	**#part**	**Finn**	**p-value**	**#class**	**#part**	**Finn**	**p-value**
Monopoly	8	14	0.717	3.68e-14	6	11	0.636	1.44e-06
Tetris	6	14	0.758	5.19e-13	8	11	0.636	2.94e-08
Tic Tac Toe	4	13	0.667	3.73e-06	5	11	0.68	1.17e-06
Chess	8	11	0.755	4.2e-13	8	14	0.739	1.67e-15
Jackut	6	11	0.612	4.98e-06	4	14	0.516	0.000604
Solitaire	6	11	0.745	7.38e-10	6	14	0.579	1.54e-06

In order to analyze evolution of agreement coefficient from the case without visualization to the case with visualization, we used level of agreement defined by Landis and Koch, explained earlier in Section 4.3.9. Table 4.13 shows the evolution of agreement level in FinG 1 and in FinG 2. First column shows the programs. Next, the group of columns shows the evolution of agreement level for FinG 1. Last group of columns shows evolution of agreement level for FinG 2. As we can see in the second line, for FinG 1, the level of agreement related to Tetris program was "moderate" in both without and with visualization cases. Column evolution was filled with the term "SAME", indicating there was no evolution in the agreement level. For the Jackut program, the column evolution indicates the agreement level increased when participants used visual support. The opposite occurred in FinG 2. For the Jackut program, the column evolution indicates the agreement level decreased when the participants used visual support. Some values are not filled because the p-value is out of the significance range (> 0.05). This indicates that we could not be confident about these coefficient values, in comparison with the other coefficient values. We disregarded these cases from the analysis.

Overall, the agreement among the participants using visualization was better for only one case, in FinG 1. Three cases were disregarded. And, the agreement level was the same for the other two cases. In FinG 2, the agreement was worse in two cases, and it was the same for the other four cases.

Table 4.13 Evolution of agreement level, from without visualization to with visualization cases, in FinG 1

	From without visualization to with visualization					
	FinG 1			FinG 2		
Program	**From**	**To**	**Evolution**	**From**	**To**	**Evolution**
Monopoly	Moderate	-	-	Substantial	Substantial	SAME
Tetris	Moderate	Moderate	SAME	Substantial	Substantial	SAME
Tic Tac Toe	Moderate	-	-	Substantial	Substantial	SAME
Chess	Moderate	Moderate	SAME	Substantial	Substantial	SAME
Jackut	Moderate	Substantial	**INCR**	Substantial	Moderate	**DECR**
Solitaire	-	Substantial	-	Substantial	Moderate	**DECR**

4.5.1.2 Agreement between the participants and the oracle As discussed in Section 4.3.9, we calculated the accuracy of the participants' answers, considering our oracle. Figure 4.5 shows their distribution in FinG 1 (left graphic) and in FinG 2 (right graphic). As we can see, the distribution and means are similar for both cases. However, the mean of accuracy values for FinG 1 is higher, when visual resources were adopted. In opposition, in FinG 2, the mean of accuracy values is higher when visual resources were not used.

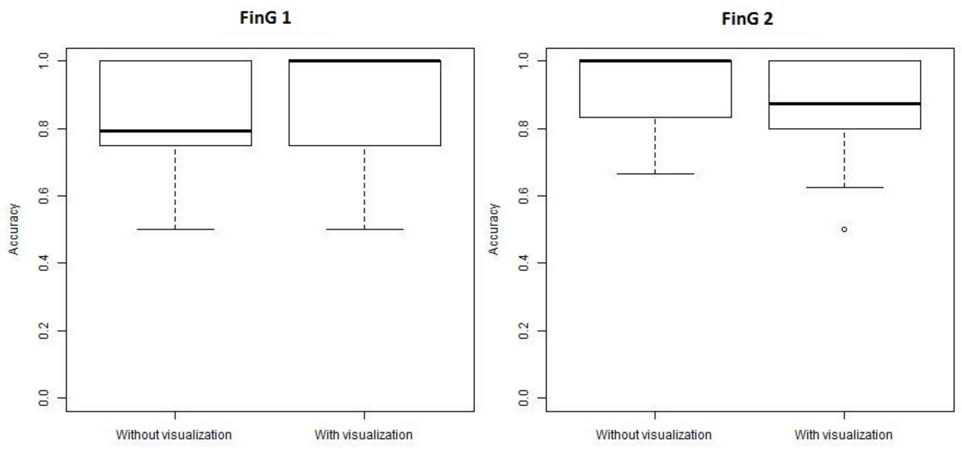


Figure 4.5 Accuracy distribution for FinG 1 and FinG 2, grouped by the use of visualization

Hypothesis test. We expected higher accuracy values for the cases where participants had a better comprehension of the design. Based on discussion about our hypothesis in Section 4.3.7, we defined the null hypothesis as:

- *H0: there is no difference of accuracy values between the cases where participants were using and not using visualization tool*

Table 4.14 shows the statistical tests. In the first line, we considered results for FinG 1. In the second line, we considered results for FinG 2. In both cases, we could not reject the null hypothesis using the Mann-Whitney non-parametric test (notice that the distribution is not normal according to the Shapiro-Wilk test). The final columns of the table show the Cliff's Delta effect size. The magnitude on the effect size, for both experiments, was negligible.

Table 4.14 Hypothesis test for analysis of design comprehension tool support, considering FinG 1 and FinG 2

FinG	Shapiro-Wilk				Mann-Whitney		Cliff's Delta effect size	
	Without vis		With vis					
	W	p-value	W	p-value	W	p-value	delta	magnitude
1	0.8045	4.953e-05	0.7688	8.641e-06	478	0.4884	0.09	negligible
2	0.7723	2.34e-09	0.7867	4.619e-09	3070	0.2283	0.11	negligible

4.5.2 Developer knowledge

In this section, we present the results about how developer knowledge impacts agreement on god class detection. We based our analysis on a questionnaire applied during training of FinG 2 and FinG 3. We discussed the questionnaire in Section 4.4.1. To evaluate the knowledge, we analyzed answers of the questionnaire. We defined a template, before reading the answers. Then, we classified the answers as follows:

1. Wrong answer (distant of the central idea)
2. Some correct propositions, but without capturing the main idea
3. Capturing main idea, but without a good explanation
4. Capturing main idea, with a good explanation (perfect answer)

From our template, we scored the answers of each participant from 3 (three wrong answers) to 12 (three perfect answers). We grouped the participants according to three different thresholds scores: i) 70%, ii) 80% and, iii) the median. All results impacted our analysis in the same way. In this thesis, we show results considering the threshold score as 70%.

We considered participants with scores equal or above 70% as with "good" knowledge. In opposition, we considered participants with scores under 70% as "bad" knowledge. In FinG 2, there were 15 participants with "good" knowledge, and nine participants with "bad" knowledge. Note that, the total of participants considered was 24: one of them did not send us his/her questionnaire. Due to this, we removed the participant from this analysis. In FinG 3, there were nine participants with "good" knowledge, and 16 participants with "bad" knowledge.

4.5.2.1 Agreement among the participants Table 4.15 shows agreement coefficient values for FinG 2. We show in the first part of the table, agreement among participants with "bad" knowledge. In the second part of the table, we show agreement among participants with "good" knowledge.

Table 4.16 has the same structure of the previous table, but it shows the agreement values among participants of FinG 3. The main difference on the setup of Fing 2 and FinG 3 is software size. FinG 3 used four medium size programs

Table 4.17 shows the evolution of agreement level from "bad" to "good" knowledge, in FinG 2. There were two cases where the agreement level decreased, when participants had "good"

Table 4.15 Finn agreement test for FinG 2, grouped by participants' knowledge

	Bad knowledge				Good knowledge			
Program	**#class**	**#part**	**Finn**	**p-value**	**#class**	**#part**	**Finn**	**p-value**
Monopoly	8	9	0.681	4.02e-08	8	15	0.757	6.88e-18
Tetris	10	9	0.756	3.71e-13	10	15	0.775	9.43e-24
Tic Tac Toe	5	9	1	0	5	15	0.611	9.02e-07
Chess	10	9	0.856	1.3e-20	10	15	0.779	3.7e-24
Jackut	6	9	0.63	2.02e-05	6	15	0.683	2.94e-10
Solitaire	6	9	0.63	2.02e-05	6	15	0.657	2.66e-09

Table 4.16 Finn agreement test for FinG 3, grouped by participants' knowledge

	Bad knowledge				Good knowledge			
Program	**#class**	**#part**	**Finn**	**p-value**	**#class**	**#part**	**Finn**	**p-value**
JMoney	12	16	0.506	1.39e-09	12	9	0.556	4.78e-07
JParse	12	16	0.381	1.51e-05	12	9	0.481	2.54e-05
Quilt	12	16	0.593	7.65e-14	12	9	0.611	1.04e-08
Squirrel	12	16	0.501	2.04e-09	12	9	0.574	1.46e-07

knowledge: Tic Tac Toe and Chess programs. For the other four cases, the agreement level was the same.

In Table 4.18, we show the evolution of agreement from "bad" to "good" knowledge, in FinG 3. In two cases, the agreement level among participants with "good" knowledge increased. For the other two cases, the agreement level was the same.

4.5.2.2 Agreement between the participants and the oracle We show accuracy distribution in Figure 4.6. On the left graphic, we show results for FinG 2; on the right graphic, we show results for FinG 3. A significant difference occurred in the FinG 2 graphic: the mean is higher when participants had "bad" knowledge.

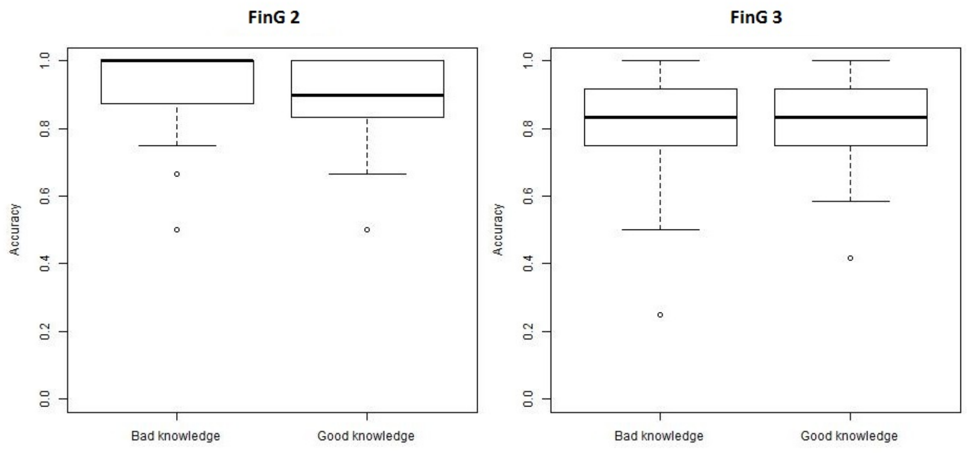


Figure 4.6 Accuracy distribution for FinG 2 and FinG 3, grouped by developers' knowledge

Hypothesis test. We expected that participants with "good" knowledge succeeded more than participants with "bad" knowledge. Then, we defined the null hypothesis as:

- *H0: there is no difference of accuracy values between participants with "good" knowledge and "bad" knowledge*

Table 4.17 Evolution of agreement level in FinG 2, from "bad" to "good" knowledge

Program	**From bad knowledge**	**To good knowledge**	**Evolution**
Monopoly	Substantial	Substantial	SAME
Tetris	Substantial	Substantial	SAME
Tic Tac Toe	Almost Perfect	Substantial	**DECR**
Chess	Almost Perfect	Substantial	**DECR**
Jackut	Substantial	Substantial	SAME
Solitaire	Substantial	Substantial	SAME

Table 4.18 Evolution of agreement level in FinG 3, from "bad" to "good" knowledge

Program	**From bad knowledge**	**To good knowledge**	**Evolution**
JMoney	Moderate	Moderate	SAME
JParse	Fair	Moderate	**INCR**
Quilt	Moderate	Substantial	**INCR**
Squirrel	Moderate	Moderate	SAME

In Table 4.19, we show statistical tests for both FinG 2 and FinG 3 experiments. In both cases, we could not reject the null hypothesis. The magnitude on the effect size, for both experiments, was negligible.

Table 4.19 Hypothesis test for analysis of developer knowledge, considering FinG 2 and FinG 3

	Shapiro-Wilk				**Mann-Whitney**		**Cliff's Delta effect size**	
FinG	**Bad knowledge**		**Good knowledge**					
	W	**p-value**	**W**	**p-value**	**W**	**p-value**	**delta**	**magnitude**
2	0.7147	7.807e-09	0.7995	1.002e-09	2673.5	0.1921	0.12	negligible
3	0.892	4.049e-05	0.896	0.002658	1080.5	0.6025	0.06	negligible

4.5.3 Developer experience

To evaluate how experience impacts agreement on god class detection, we compared results of FinG 1 against FinG 2. The experiments had similar setup and the main difference between them was the participants' profile. We captured experience based on two questions of the Characterization form. The first question asked how long the participant worked professionally (or as a researcher) on software development. The second question asked how long they had programmed using the object oriented paradigm, considering course work in this case. We present distribution of the answers in Figure 4.7. The participants of FinG 2 had two years or more of experience than the participants of FinG 1.

Due to this difference, we grouped participants of FinG 1 and FinG 2 in different categories. We adopted the classification presented by (Höst; Wohlin; Thelin, 2005). They addressed experience as context variable in empirical studies and proposed a classification, which we show in Table 4.20. The scale in the table is ordinal, a higher value corresponds to more experience than a lower value. It is important to note the authors did not present the classes as a rigid classification, but as a starting point to analysis of experience. As all participants of FinG 1 were undergraduate students, they are all in the E1 class. Participants of FinG 2 were classified in-between the E2 and E5 classes. In spite being graduate students, their profiles indicated most of them were or had been professionals.

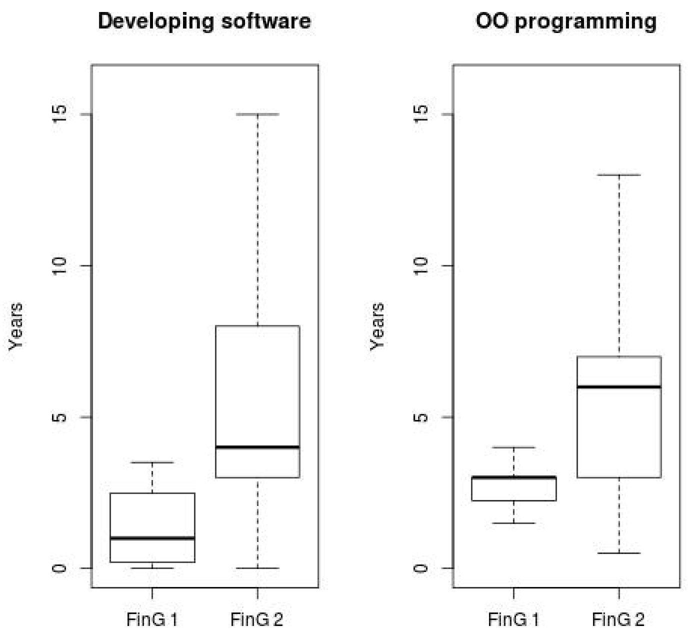


Figure 4.7 Distribution of practical years of experience and software programming, for FinG 1 and FinG 2's participants

Table 4.20 Class of experiences of participants (Höst; Wohlin; Thelin, 2005)

Category	Description
E1	Undergraduate student with less than 3 months recent industrial experience
E2	Graduate student with less than 3 months recent industrial experience
E3	Academic with less than 3 months recent industrial experience
E4	Any person with recent industrial experience, between 3 months and 2 years
E5	Any person with industrial experience for more than 2 years

4.5.3.1 Agreement among the participants Table 4.21 shows agreement coefficient values for FinG 1 and FinG 2. We show in the first part of the table, values for FinG 1, and in the second part, values for FinG 2. Note that, for Jackut program in FinG 1, and Tic Tac Toe program in FinG 2, the number of participants is different of the other cases. This happened because the participants marked in the form a class as a candidate, but they did not fill the name of the classes. We removed these data to avoid affecting the test.

Table 4.22 shows the evolution of agreement level from FinG 1 to FinG 2. It is possible to note that, when some difference in the level of agreement occurred, the agreement was higher in FinG 2. It happened for Chess program.

4.5.3.2 Agreement between the participants and the oracle We show the accuracy distribution in Figure 4.8. The main observation from the figure is that the accuracy mean is significantly higher in FinG 2 than in FinG 1.

Hypothesis test. To perform statistical tests comparing accuracy values of FinG 1 and FinG 2, we defined the null hypothesis. We expected more experienced participants succeed

Table 4.21 Finn agreement test for FinG 1 and FinG 2 experiments

	FinG 1				FinG 2			
Program	**#class**	**#part**	**Finn**	**p-value**	**#class**	**#part**	**Finn**	**p-value**
Monopoly	4	11	0.636	7.4e-05	8	25	0.732	2.51e-26
Tetris	8	11	0.682	8.12e-10	10	25	0.78	2.56e-40
Tic Tac Toe	4	11	0.691	7.94e-06	5	24	0.713	3.12e-15
Chess	10	11	0.716	1.22e-13	10	25	0.803	8.14e-45
Jackut	8	10	0.722	1.67e-10	6	25	0.66	5.65e-15
Solitaire	10	11	0.702	7.3e-13	6	25	0.649	2.61e-14

Table 4.22 Evolution of agreement level, from FinG 1 to FinG 2

Program	From FinG 1	To FinG 2	Evolution
Monopoly	Substantial	Substantial	SAME
Tetris	Substantial	Substantial	SAME
Tic Tac Toe	Substantial	Substantial	SAME
Chess	Substantial	Almost perfect	**INCR**
Jackut	Substantial	Substantial	SAME
Solitaire	Substantial	Substantial	SAME

more than less experienced participants. Then, our null hypothesis is:

- *H0: there is no difference of accuracy values between participants of FinG 1 and FinG 2*

Table 4.23 shows statistical tests. We highlighted Mann-Whitney p-value because we rejected the null hypothesis. We also highlighted the magnitude of the Cliff's Delta value because it showed some magnitude (despite small) on the effect size.

Table 4.23 Hypothesis test for analysis of developer experience, considering FinG 1 and FinG 2

Shapiro-Wilk				Mann-Whitney		Cliff's Delta effect size	
FinG 1		FinG 2					
W	**p-value**	**W**	**p-value**	**W**	**p-value**	**delta**	**magnitude**
0.8419	8.066e-07	0.7807	1.163e-13	3974.5	**0.02735**	0.18	**small**

4.5.4 Software size

In this section, we present results related to our analysis of how software size impacts agreement on god class detection. We compared agreement of FinG 2 against FinG 3. We did this because the main difference on experimental setup was the type of software adopted. In FinG 2, we used simple software; in FinG 3, we used medium sized – more complex – software.

4.5.4.1 Agreement among the participants Table 4.24 shows results of agreement among the participants of FinG 2. In the same way, we show results for FinG 3, in Table 4.25. Once software adopted in the experiments were different, we did not analyze the evolution. Instead, we compared the general tendency of agreement level in both FinG 2 and FinG 3 experiments. In FinG 2, there was one almost perfect level of agreement for Chess software. For all other values, agreement level was substantial. In FinG 3, there was one substantial agreement level, for Quilt software. For all other values, agreement level was moderate.

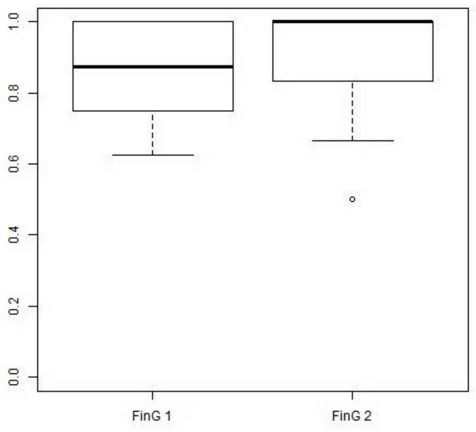


Figure 4.8 Accuracy distribution, grouped by FinG 1 and FinG 2 experiments

Table 4.24 Finn agreement test and level for FinG 2

Program	#class	#part	Finn	p-value	Agreement level
Monopoly	8	25	0.732	2.51e-26	Substantial
Tetris	10	25	0.78	2.56e-40	Substantial
Tic Tac Toe	5	24	0.713	3.12e-15	Substantial
Chess	10	25	0.803	8.14e-45	Almost perfect
Jackut	6	25	0.66	5.65e-15	Substantial
Solitaire	6	25	0.649	2.61e-14	Substantial

4.5.4.2 Agreement between the participants and the oracle Figure 4.9 shows accuracy distributions. As we can see, accuracy values for FinG 2 were higher than accuracy values for FinG 3.

Hypothesis test. We consider that participants detecting god class on simpler software succeed more than participants detecting god class on more complex software. The null hypothesis is:

- *H0: There is no difference of accuracy values between participants of FinG 2 and FinG 3*

Table 4.26 shows statistical test. In this case, we rejected the null hypothesis. We highlighted the Cliff's Delta value because it showed some magnitude (medium) on the effect size.

4.5.5 Developer training

The main difference between the experimental setup of FinG 3 and FinG 4 was the training. After the analysis of the other factors, we concluded that the conceptualization of god class is more dependent of human traits than of technical aspects. Due to this, we conjectured it would be possible to affect the agreement aligning the personal thresholds on god class detection. In order to test this idea, the training of FinG 4 went beyond reading, including training with golden examples and live discussions.

4.5.5.1 Agreement among the participants Table 4.27 shows the agreement coefficient among participants of FinG 3 and FinG 4.

The evolution of agreement from FinG 3 to FinG 4 is shown on Table 4.28. There were three, in four, cases where the agreement level in FinG 4 was higher than in FinG 3.

Table 4.25 Finn agreement test and level for FinG 3

Program	#class	#part	Finn	p-value	Agreement level
JMoney	12	25	0.511	1.04e-14	Moderate
JParse	12	25	0.43	4.14e-10	Moderate
Quilt	12	25	0.608	1.63e-22	Substantial
Squirrel	12	25	0.529	6.29e-16	Moderate

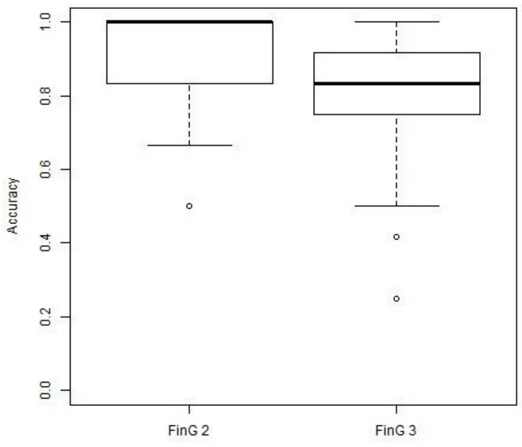


Figure 4.9 Accuracy distribution for FinG 2 and FinG 3

4.5.5.2 Agreement between the participants and the oracle Figure 4.10 shows accuracy distribution of FinG 3 and FinG 4. From the graphic, it is possible to note that accuracy values are higher in FinG 4 than in FinG 3.

Hypothesis test. According previous discussion, we expect participants who are trained with "golden" examples, discussing personal thresholds, succeed more than participants trained only reading books. Due this, the null hypothesis is:

- *H0: There is no difference of accuracy values between participants of FinG 3 and FinG 4*

Table 4.29 shows statistical test. As in our previous analysis, we rejected the null hypothesis and the Cliff's Delta value showed some magnitude (despite small) on the effect size.

4.5.6 All factors, considering "yes" or "maybe" marks

This section presents the results for all factors we analyzed considering, as candidates god class, classes marked with "yes" or "maybe" options. We focus only on the statistical tests, avoiding more tables and figures to make the thesis shorter, without omission of the most significant results.

We show the statistical tests in Table 4.30. The first column shows the factors. The second column shows how we considered the samples from the experiments. Lastly, we show the statistical tests considering classes marked with "yes or maybe" and "only yes" options, as candidate god class. As previously, we adopted the Mann-Whitney test because the distribution of all samples was not normal. We highlight values where significance value was lower than 0.05.

The developer knowledge factor in FinG 2 was exactly the opposite of what would be expected. According to it, participants with "bad" knowledge succeed more than participants with

Table 4.26 Hypothesis tests for software size, considering FinG 2 and FinG 3 accuracy values

Shapiro-Wilk				Mann-Whitney		Cliff's Delta effect size	
FinG 2		FinG 3					
W	p-value	W	p-value	W	p-value	delta	magnitude
0.7807	1.163e-13	0.8934	6.978e-07	10183	**4.424e-07**	0.37	**medium**

Table 4.27 Finn agreement test for FinG 3 and FinG 4

	FinG 3				FinG 4			
Program	**#class**	**#part**	**Finn**	**p-value**	**#class**	**#part**	**Finn**	**p-value**
JMoney	12	25	0.511	1.04e-14	12	17	0.64	8.12e-18
JParse	12	25	0.43	4.14e-10	12	17	0.515	1.61e-10
Quilt	12	25	0.608	1.63e-22	12	17	0.841	1.41e-43
Squirrel	12	25	0.529	6.29e-16	12	17	0.718	7.86e-25

"good" knowledge (with a p-value of 0.00728). We checked the means in order to understand this finding. The accuracy mean for "bad" knowledge is 0.88, and for "good" knowledge is 0.82. We discuss this distortion in the next section. In opposition, the developer knowledge factor in FinG 3 indicated that participants with "good" knowledge succeed more than participants with "bad" knowledge (with a p-value of 0.03572). We also checked the means in order to confirm the outcome. The accuracy mean for "good" knowledge is 0.80, and for "bad" knowledge is 0.73. Disregarding these values, the only relevant result considering "yes" or "maybe" marks is for software size factor (with a p-value of 2.627e-06).

4.6 DISCUSSION

In this section, we present our findings. We discuss each factor addressed, independently. Then, we provide a general discussion about the subject.

4.6.1 Design comprehension tool support

We found *a design comprehension tool support does not improve agreement on god class detection*. An interesting aspect related to this analysis is that there were contradictory results. In some cases, participants of FinG 1 had better results using visualization, and participants of FinG 2 had better results not using visualization. We have to consider the experience of the participants as a potential confounding factor in this analysis. However, we consider that the evidences indicate that the design comprehension tool support did not impact the agreement among the participants detecting god class. Our evidences were: i) "SAME" as the value for the most cases in the evaluation of the level of agreement among the participants; ii) the similar accuracy distribution (agreement with the oracle) for both using and not using visualization cases; and iii) the statistical tests, which did not confirm the hypothesis that participants using a design comprehension tool succeed more than participants who do not use it. We conjecture that god class detection is more related to personal conceptualization than to design comprehension, and that technical support does not affect the conceptualization of the smell. We believe this is transversal to other kinds of technical support. Metrics-based tools, for example, are dependent on heuristics and value thresholds, which are dependent on human definition anyway. Inconsistencies on tools using metrics-based heuristics are, by the way, discussed by Fontana, Braione and Zanoni (2012).

Table 4.28 Agreement evolution, from FinG 3 to FinG 4

Program	From FinG 3	To FinG 4	Evolution
JMoney	Moderate	Substantial	**INCR**
JParse	Moderate	Moderate	SAME
Quilt	Substantial	Almost perfect	**INCR**
Squirrel	Moderate	Substantial	**INCR**

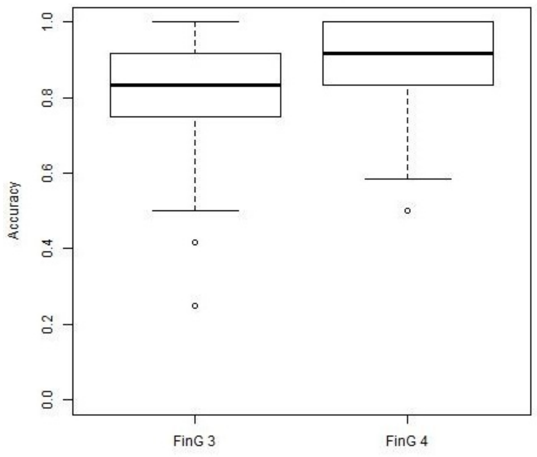


Figure 4.10 Accuracy distribution, grouped by FinG 3 and FinG 4 experiments

4.6.2 Developer knowledge

As discussed previously, we found conflicting results in FinG 2. There were two cases where the agreement level was lower for participants with "good" knowledge in FinG 2. However, we had opposite findings in FinG 3, where there were two cases in which the agreement was higher for participants with "good" knowledge. Accuracy distribution (agreement with the oracle), the hypothesis test and effect size were not conclusive. It is important to note that the dependent variable, in this case, might be affected by the different types of software used in FinG 2 and FinG 3. Remember that the software used in FinG 3 is more complex than the software used in FinG 2. Clearly, software size is an important confounding factor, but the findings also indicate that the developer knowledge is more relevant on evaluation of more complex software. Due to this, although we do not consider the evidences are very strong, we believe knowledge also impacts agreement to a certain level, and the subject should be further analyzed.

4.6.3 Developer experience

The experience impacts agreement on god class detection, but not definitively. Despite most cases of agreement among the participants had been the same (only in one case the agreement level was higher for more experienced developers), accuracy distribution (agreement with the oracle) and the hypothesis test confirmed this idea. The magnitude of the effect size also confirmed that there are a difference (despite small) on the accuracy values with respect to the oracle for both more and less experienced participants cases. As discussed in Section 4.3.9, we are considering small magnitude on the effect size as evidence reinforcing our hypothesis because of the small number of data points in our samples. When considering the doubtful cases, i.e., classes marked with "yes" or "maybe" options, we could not confirm the hypothesis statistically.

Table 4.29 Hypothesis tests for analysis of developer training, considering FinG 3 and FinG 4

Shapiro-Wilk				Mann-Whitney		Cliff's Delta effect size	
FinG 3		FinG 4					
W	p-value	W	p-value	W	p-value	delta	magnitude
0.8934	6.978e-07	0.8367	3.531e-07	2392	**0.0008652**	0.3	**small**

Table 4.30 Hypothesis tests for all factors addressed, considering as candidates god class, classes marked with "yes" or "maybe" options

Factor	Experiment	Mann-Whitney			
		"Yes" or "maybe"		Only "yes"	
		W	p-value	W	p-value
Design comprehension	FinG 1 (two groups)	504	0.7552	478	0.4884
	FinG 2 (two groups)	2726	0.8532	3070	0.2283
Developer knowledge	FinG 2 (two groups)	3024.5	**0.00728**	2673.5	0.1921
Developer experience	FinG 1 X FinG 2	5144	0.468	3974.5	**0.02735**
	FinG 3 (two groups)	863.5	**0.03572**	1017.5	0.5217
Software size	FinG 2 X FinG 3	10059.5	**2.627e-06**	10183	**4.424e-07**
Developer training	FinG 3 X FinG 4	3236.5	0.5936	2392	**0.0008652**

These findings hint that there are other human aspects affecting god class detection.

4.6.4 Software size

All analysis procedure we adopted indicates that *software size impacts agreement on god class detection.* In our experiments, participants detecting god class in simpler software (FinG 2) agreed and succeeded more than in more complex software (FinG 3). We found strong evidences of this, evaluating the agreement among the participants, accuracy distribution (agreement with the oracle), and observing the hypothesis tests (including classes marked with the "maybe" option as candidates god classes) and the magnitude of the effect size (which was classified as medium). This reinforces our previous conjecture about of software size as a confounding factor in the analysis of the developer knowledge.

4.6.5 Developer training

The last factor we analyzed was the training. We found that *participants discussing the conceptualization and personal thresholds based on examples agree and succeed more than participants in a reading-based training.* As in the previous analysis, we based our finding in the evaluation of agreement among the participants, accuracy distribution, and in the hypothesis test and magnitude of the effect size. We conjecture that this finding reinforces our beliefs that conceptualization of code smells is the main cause of inconsistencies of empirical studies on the topic. We believe that the training based on examples and discussions is the best approach to control human traits, which significantly affects the agreement on god class detection.

4.6.6 Overall discussion

Developer's training strongly impacted the agreement, but the use of software visualization did not. These evidences show that god class detection is more related to human traits than to an overall comprehension of the code design. We believe this finding is transversal to other types of code smell, because the human aspects are related to subjective evaluation of the design quality, which occurs with other types of smell, in different level of intensity. Based on this, we consider studies focused on smell detection that disregard human aspects might be misdirected. The misdirection of studies focused solely on smell effect was also conjectured by Sjøberg et al. (2013).

Software size also did affect the agreement. Agreement was lower in medium size than in small size software. This is evidence that context factors, other than human traits, affect the conceptualization of code smells. This dimension must also be further explored in future experiments.

Also, the use of heuristics for automatic detection of smells, which are created by humans, should be adjusted for specific contexts. The proposition of new heuristics should be presented after the evaluation of the relationship between the current heuristics and the context in which the smells are used (human traits there included). This does not seem to be the current trend on the subject. In their systematic mapping study, Zhang, Hall and Baddoo (2011) noted that the most of studies in code smells focus on tools and methods for automatic detection.

To get a better comprehension of the smell effect, it is necessary to get a better comprehension of how human evaluate design quality. We call this issue the *"code smell conceptualization problem"*. The *code smell conceptualization problem* involves the discussion about ensuring same comprehension of the smell concepts and similar personal criteria or thresholds. We consider that two main paths should be followed to solve this problem: i) to deepen the evaluation of the human aspects and how they affect smell detection; and ii) to evaluate cognitive aspects on smell detection, which is related to program comprehension, and requires knowledge both in computer science and cognitive psychology (Maletic, 2008).

Table 4.31 summarizes the main topics discussed in this section. At the top of the table, we present the main outcome related to the addressed factors by FinG. At the bottom, we summarize the overall discussion that we propose.

Table 4.31 Summary of the topics discussed

Factor	**Outcome**
Design comprehension	A design comprehension tool support does not improve agreement on god class detection
Developer knowledge	It should be further analyzed
Developer experience	The experience impacts agreement on god class detection, but not definitively
Software size	Software size impacts agreement on god class detection
Developer training	Participants discussing the conceptualization and personal thresholds based on examples agree and succeed more than participants in a reading-based training
Overall discussion	God class detection is more related to human traits than to an overall comprehension of the code design
	Other context factors not related to the human traits, such as software size, also affect human conceptualization of smells. This issue should be further investigated
	Heuristics for smell detection should be presented after the evaluation of the relationship between the current heuristics and the context in which the smells are used (human traits there included)
	The area should addresses the *code smell conceptualization problem*, which involves the discussion about ensuring same comprehension of the smell concepts and similar personal criteria or thresholds

4.7 THREATS TO VALIDITY

Our analysis of threats to validity of the study was based on the classification adopted by Wohlin et al. (2012).

External validity. Our first threat fits in the "interaction of selection and treatment" subcategory and is related to the fact that the experiments were carried out in an in-vitro setting. Participants of FinG 2, FinG 3 and FinG 4 were graduate students; and participants of FinG 1 were undergraduates. One aspect mitigates the threat: most participants of FinG 2, FinG 3 and FinG 4 had some professional software development experience. Moreover, our focus was the investigation of the impact of human traits on smell detection, as opposed to the quality of the evaluation of the smell detection, which would arguably be more affected by the profile of the participants. Other threat to external validity fits in the "interaction of setting and treatment" subcategory. In this case, the threat is the type of software used in FinG 1 and FinG 2. We adopted six small programs. We mitigated this threat by using medium programs in FinG 3 and FinG 4.

Internal validity. There is a threat of "ambiguity about direction of causal influence". In FinG 1, the training on god class was influenced by the view of the experimenter (also the trainer). To minimize this effect, we limited the time of training in the god class concept and adopted the support questions used in the Schumacher's experiment (Schumacher et al., 2010) to guide participants on the god class detection during the experiment. We also mitigated this threat in FinG 2, FinG 3 and FinG 4. Instead of a presentation by the experimenter, participants read reference books to acquire or improve their knowledge on the topic. Another threat in same subcategory was the training in the visualization tool in FinG 1 and FinG 2. In our feedback form, participants indicated that the quality of training was good, in general. However, we cannot confirm that it was sufficient to prepare participants in the use of visualization. We consider this a weak empirical evidence of the validity of the training. However, we are confident about the answers because the software visualization tool is very simple and some participants gave positive comments after the experiments.

Another subcategory of the internal validity is "maturation". Participants could be affected because they did the same task over six programs, in FinG 1 and FinG 2, and four – more complex – programs, in FinG 3 and FinG 4. They may have learned and worked faster as they progressed on the experiments. On the other hand, they could be negatively affected by boredom. We consider maturation a weak threat because the experiments were performed in 2 hours, on average. We consider this a reasonable period of time to do a task in a balanced way.

Conclusion validity. In the "reliability of treatment implementation" threat, we have to consider that a participant who could have used visualization may have completely disregarded the resources, which would impact the analysis of FinG 1 and FinG 2. However, we logged and checked actions performed in the Eclipse IDE. We observed that only one participant in FinG 1, and two participants in FinG 2 did not use the visualization tool resources when they were available. This was done by the participants own choice and did not affect the results significantly. This issue is discussed in detail in Santos et al. (2014). We also have to consider if the visualization tool was appropriated for the identification of useful attributes for god class detection. We are confident about this, because of former experiments with the tool (Carneiro; Mendonça, 2014, 2013; Carneiro et al., 2010) and the discussions that occurred during the training of FinG 1 and FinG 2. Another threat in the same subcategory was related to the evaluation of the questionnaire answers, which was part of the training in FinG 2 and FinG 3. The evaluation of the participants' answers was subjective. We mitigated this threat comparing the answers with a template defined previously. Moreover, the questions were very simple.

Another subcategory of the conclusion validity is "random heterogeneity of subjects". We analyzed how software size impacts agreement on god class detection comparing results of FinG 2 and FinG 3. We have to consider that, despite differences on software size, the population of the experiments was formed by different participants subjects. We consider this a weak threat because the profile of the participants is similar. For both FinG 2 and FinG 3 experiments, most of the participants were graduate students with some years working with software development in software industry or research projects. There were few cases (six considering both experiments) where the participants had no professional experience on software development. Even in that cases, the participants declared to have knowledge about OO programming.

Construct validity. In the "inadequate preoperational explication of constructs", we have to consider the god class concept we adopted involves conceptual definitions of god class, brain class and large class. In order to mitigate this threat, we considered the adoption of the support questions and the effort on the training for all experiments. This was added to the presentation, reading of books, and questionnaire answers. Moreover, on FinG 4 training, it was possible to discuss with participants their conceptualization about god class.

Another threat that we found in the same subcategory is related to the material support considered in the analysis of some factors. For example, the analysis of the level of knowledge was based on the answers from a questionnaire defined by us. Due to the simplicity of the questions, we defined the questionnaire based on our own perceptions about what is important in order to capture the developer knowledge on the subject. In the same way, we defined the questions of the characterization form used in the analysis of the developer experience; and in the definition of the SourceMiner exercise, which trained the participants on the visualization tool. We mitigated this threat by adapting some of the material from works previously published in the literature Novais et al. (2012), Schumacher et al. (2010) and discussing about them with other researchers from our research community.

4.8 CONCLUSION FOR FING FAMILY

FinG aims to improve the understanding of factors affecting human evaluation of code smells. The FinG's experiments were defined in an iterative process. Insights of one experiment were used in definition of the next ones, in order to batter control factors and explore insights gained in the previous experimental cycle (Mendonça et al., 2008). Specifically, we addressed cause-effect relation between five independent variables and the agreement on god class detection. The variables were: design comprehension tool support, developer experience, developer knowledge, size software and training.

Due to the difficulties of controlling factors affecting smell detection, the software engineering community has more and more focused on the use of software metrics for smell detection. The discussion we provided here, which we called the *code smell conceptualization problem*, shows that the problem is complex and the community must ensure similar personal criteria and thresholds on smell detection.

Tool support for design comprehension (which was obtained by the use of software visualization) was the only factor did not impact agreement on god class detection. In opposition, the training strongly impacted the agreement. This evidences god class detection is more affected by subjective factors related to human traits than to an overall comprehension of the software design. We believe this finding is transversal to other code smells.

As smell detection is strongly affected by human aspects, and other context attributes, the heuristics based on metrics, that are currently being used by the software engineering community, should be always checked and adjusted for specific situations. This also raises out the need for more studies focusing on the identification and characterization of factors that affects the perception and impact of code smells, specially human aspects.

We showed that training impacts god class detection. As this is controllable factor, we propose training based on "golden" examples and discussions between developers to align their conceptualization about code smells in software organizations. This way they may seize some control over the smell conceptualization problem.

We also believe that the discussion on the effectiveness of adoption of code smell concepts in software development needs to evolve. We have already published other partial results of our work (Santos; Mendonça, 2015; Santos et al., 2014; Santos; Mendonça, 2014; Santos; Mendonça; Silva, 2013), exploring detection effort, code smell decision drivers and detection strategies. To support replications of our experiments, we make our experimental packages available to the software engineering research community[2].

[2]http://wiki.dcc.ufba.br/LES/JoseAmancio

Chapter

5

This chapter presents a research agenda based on our observations from both the thematic synthesis and the family of controlled experiments studies that we carried out. It present the main challenges that we identified from our studies. The agenda aims to steer researchers toward a better understanding of the code smell effect.

A RESEARCH AGENDA

We identified some challenges for the area based on our thematic synthesis and controlled experiments. Our first observation is that the area needs more empirical studies addressing code smells. This statement is in accordance with other authors (Sjøberg et al., 2013; Zhang; Hall; Baddoo, 2011; Schumacher et al., 2010; Mäntylä; Lassenius, 2006; Mäntylä; Vanhanen; Lassenius, 2004). However, we believe that the focus of the studies can be better directed. We consider that the empirical studies in the area should focus on understanding the code smell effect, which involves correlation studies and investigations of the human role on smell detection. For us, this type of study explores more relevant questions about practical adoption of code smells than studies focused on tool assessments.

This does not seem to be the path followed by the area. In their mapping study, Zhang, Hall and Baddoo (2011) found that most studies focus on how to identify smells, instead of investigating their impact. Our thematic synthesis also showed that many studies focus on tool assessments. We removed them from the thematic synthesis because they present findings related to their method or process for smell detection, and nothing about the impact of the smell on software development. We conjecture about one of the reasons for this phenomenon: we believe that studies focusing on tool assessments produce empirical results more quickly than studies addressing the smell effect, specially involving humans, where qualitative research plays a relevant role.

5.1 UNDERSTANDING THE NATURE OF THE CODE SMELLS

The first observation we did in our thematic synthesis was related to the large number of smells addressed by the primary studies. For the most cases, it is not possible to synthesize findings about the same theme because of the variety of smells addressed. In order to enhance syntheses on the topic, we have to group smells according their characteristics. We did this informally in our family of controlled experiments. During the experiments' training, we considered concepts of three smells, which were *god class*, *brain class* and *large class*. We showed, in Chapter 2, that these smells are based on similar idea, which was also observed by Mäntylä and Lassenius

(2006) and Lanza and Marinescu (2005). This was the case for the three smells adopted in our controlled experiments. In our thematic synthesis, we found 63 code smells being addressed by the empirical studies. This number might be even higher if our secondary study had broader scope such as in the case of a mapping study, for example (see discussion in Chapter 2).

The area lacks discussions about grouping smells according to their characteristics. For example, Fowler (1999) presents the code smell *large class* as a class "trying to do too much" and also as "a class with too much code". Lanza and Marinescu (2005) consider the metric *lines of code* (LOC) calculating values of a metric named *size of exact clone*, which is used in the identification of the *significant duplication* code smell. LOC is also used by the heuristic for detection of *brain class* code smell. We understand that other aspects affect the definition of these code smells. However, one may speculate that these smells impact software evolution in the same way. Due to this, one also may speculate: is it possible to group findings related to these three smells together?

These discussions are related to the nature of the code smells. They will enhance the understanding about how specific aspects of the smells impact software evolution, making possible to group smells according to impact, features and interests of analysis. We found only one work grouping smells (Mäntylä; Lassenius, 2006). However, it was not based on a theory and grounded on empirical studies. We considered the Mäntylä and Lassenius (2006) work a start point, but much remains to be done.

5.2 UNDERSTANDING THE RELEVANCE OF THE SUBJECTIVITY ON SMELL EVALUATION

From our thematic synthesis, we noted that most of studies addressing human role on smell detection showed that human evaluation should not be trusted. Moreover, these studies focused on specific smells, which was mainly *god class*. Note that the evaluation of *god class* is mostly arguably easier than the evaluation of other smells, such as *feature envy* or *shotgun surgery*, because it involves the observation of a smaller number of classes.

This also evidences difficulties of studies in the area. As previously discussed, we consider that two paths of research may be followed: i) to deep the evaluation of the human factors affecting smell detection; and ii) to evaluate cognitive aspects on smell detection, which is related to program comprehension, and requires knowledge both in computer science and cognitive psychology, as observed by Maletic (2008).

In both paths, we have to face the issue which we called the *code smell conceptualization problem* (Santos et al., 2014; Santos; Mendonça, 2014; Santos; Mendonça; Silva, 2013). Consider only the evaluation about the impact of the experience on agreement among participants detecting *god class* as an example. We carried out two controlled experiments to evaluate this variable. In our case, we noted that the developers' experience impacts *god class* detection, but not definitely (see discussion in Chapter 4). In order to improve our understanding of how experience impacts human evaluation of smells, it is necessary better outlining the variable. This involves to replicate and/or extend the experiments considering other variables, such as types of code smell, developers with different level of experience, types of software, types of analysis, and so on.

Code smell remains as a useful concept on learning how to produce good OO design. The concept summarizes important topics addressed in undergraduate and graduate Software Engineering courses. However, in order to systematize the practical adoption of the smell concept, the area needs to better comprehend the human role on smell evaluation.

5.2.1 Deepening evaluation of training on smell effect

We highlight the relevance of studies focusing on the training as a controllable variable towards systematizing the practical adoption of code smells on software development. In our family of controlled experiments we showed evidences that the training impacts smell evaluation. Studies refining the training focused on golden examples and group discussions might offer insights about factors affecting human evaluation of smells. Once we proposed only one strategy for *god class*, a wide range of other aspects might be investigated.

5.3 OUTLINING CONTEXT ASPECTS OF THE EMPIRICAL STUDIES

One of the difficulties of our thematic synthesis was related to the broad context aspects that should be considered. This occurred, for example, in the analysis of studies addressing human aspects. While some studies adopted small software, others studies adopted large software. The same for participants subjects. While some studies adopted students, other studies were carried out with professionals. When these context aspects were well outlined, the problems were related to the high number of smells addressed, such as in the case of the sub-theme *changes and defects*, or the small number of studies, such as in the case of the sub-theme *effort*.

Table 3.3, in Chapter 3, provides a good start point to establish the context of empirical studies in the subject. We based our definition on the Sjøberg et al. (2005) and Jedlitschka, Ciolkowski and Pfahl (2008) works. However, once they were focused on controlled experiments, we understand that the area needs to better outline the context aspects of the empirical studies. This will help on synthesizing the knowledge for specific contexts and on the definition of experimental setups for research in the area. For example, it would make possible to investigate if a method affects the results, or what are the main relevant context factors affecting results.

Chapter

6

This is the final chapter of this thesis. It presents the conclusion, along with its major contributions. This chapter also addresses our publishing planning as well as the recommendations for further research

CONCLUSION AND FUTURE WORK

This thesis is an exploratory research aiming at empirically investigating the practical adoption of the code smell concept on software development. We call this issue the *code smell effect*. The motivation for this study was the observation of divergent findings on empirical studies addressing the topic. We noted that, despite a consistent theory discussing heuristics for smell detection, the empirical studies do not support the well accepted idea that code smells represent potential design problems. The phenomenon is not well understood.

We empirically explored the problem from two perspectives. First, we synthesized the results of the experimentations on the code smell effect. To do this, we performed a systematic review, based on a thematic synthesis, in order to investigate how the empirical studies have addressed the code smell effect, synthesizing the findings presented. Second, we explored the human role on smell detection from a family of controlled experiments. We investigated an extensive set of independent variables establishing a cause-effect relationship between these variables and the agreement on god class detection. The independent variables we investigated were: design comprehension support tool, developer's experience, knowledge and training, and software size. In our experiments, we addressed *god class*, one of the most known code smells.

6.1 REVISITING THE MAIN CONTRIBUTIONS

The first contribution of this thesis is the discussion evidencing problems related to the code smell effect. Empirical studies on the subject have been presented, but independently they do not evidence how the empirical results support the expected effect of code smells infecting software. We identified the human role as an important confounding factor affecting the experiments on the subject. We argue that, despites affecting experiments, the human role is disregarded by the theory presented. Our thesis mitigates the gap on the topic through our qualitative synthesis and our family of controlled experiments. In this way, we strengthened the discussion about the human role on smell detection, presenting it as one of the sources of problems hindering practical adoption of code smells on software development.

We consider our empirical studies as another contribution of this thesis. Both the thematic synthesis and the family of controlled experiments present specific findings.

From the thematic synthesis, we find out that the experiments' output do not support the theory on the code smell effect. In other words, there are not strong evidences supporting the practical adoption of code smells for evaluation of design quality. We concluded this because of the set of divergent findings on correlation studies evaluating the smell effect. An additional finding is that the human evaluation of smells should not be trusted. The main evidence for this finding is the convergence of the results presented by the primary studies on the topic. All of them affirm that the agreement among developers detecting code smells is low or partial.

Our family of controlled experiments reinforces these findings. Our family of controlled experiments reinforces these findings. The human factors, such as developers experience and training, impacted the agreement on god class detection. We also noted that the overall comprehension of the code design (by the use of a visualization tool support) does not affect the human evaluation of smells. We believe that all these aspects evidence the relevance of the subjectivity on smell evaluation. We have called discussions about human aspects affecting smell detection as the *code smell conceptualization problem*.

Another contribution from the family of controlled experiments is the observation about the training on code smells. We highlight the training as a controllable factor towards the systematization of the practical adoption of code smells. The use of the training as one of the independent variables in our experiments showed that it strongly affects the agreement on smell detection. The approach we used controlling this variable might be an initial start point to discuss this issue. The training was based on golden examples and group discussion.

Another relevant contribution from our thesis is a research agenda, extracted from the analysis of the empirical studies that we performed. We believe that three main topics should be addressed prior other approaches evaluating the use of code smells, minimizing effort on misdirected research.

The first topic is understanding the nature of the code smells, grouping them according to impact, features and interest of analysis. We found only one work grouping smells (Mäntylä; Lassenius, 2006). However, it was not strongly validated by other studies, yet. The second topic of the research agenda is to better understanding the subjectivity on smell evaluation. This is related to the code smell conceptualization problem, and we consider that two paths might be followed: i) evaluation of the human factors affecting smell detection; and ii) evaluation cognitive aspects on smell detection. The third topic is the outlining of contextual aspects. We believe that the area lacks approaches to classify the empirical studies, making it difficult to synthesize findings and make generalizations. In our thematic synthesis, we identified some attributes for the empirical studies on the subject, but we did not discuss how to consider them in a research agenda.

6.2 PUBLICATIONS

We have presented results of our thesis for the scientific community. During the evolution of the research, we submitted papers to journals and conferences in order to get feedback and to strength the ideas that we have discussed. Some papers were not published, yet. Due to this, in this section, we present the published and not published papers as a publishing agenda. For the sake of simplicity, some analyses and discussions that we performed were not presented in this thesis. For these cases, we present three papers addressing these issues as appendix of this thesis. At the begin of each topic, we show the title of the paper as we have defined it:

1. *An exploratory study to investigate the impact of conceptualization in god class detection* (Santos; Mendonça; Silva, 2013). This paper was presented in the 17th International

Conference on Evaluation and Assessment in Software Engineering (EASE 2013). It discusses results of our first controlled experiment, addressing effort, agreement and decision drivers on god class detection. In this paper, we did not discuss the use of visual support in the experiment.

2. *The problem of conceptualization in god class detection: agreement, strategies and decision drivers* (Santos et al., 2014). After publishing partial results of the first experiment in the EASE 2013 conference, we decided to submit a paper discussing all analysis we performed for the first experiment, considering agreement, strategies and decision drivers adopted by the participants detecting god class. In this case, we discussed the use of visual support. The paper was published in the Journal of Software Engineering Research and Development (JSERD), in September, 2014. We present this paper in the Appendix B of this thesis.

3. *Identifying strategies on god class detection in two controlled experiments* (Santos; Mendonça, 2014). After submission of the paper discussing all results for the first controlled experiment, we decided to present the results considering the factors over several experiments. We did this because we had more consistent analysis considering more than one experiment. In this case, we presented a paper addressing the strategies adopted by the participants detecting god class in the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE 2014). We present this paper in the Appendix D of this thesis.

4. *Exploring decision drivers on god class detection in three controlled experiments* (Santos; Mendonça, 2015). We also presented a paper exploring the decision drivers adopted by the participants of three of the controlled experiments that we carried out. The paper was published in the 30th ACM/SIGAPP Symposium On Applied Computing (ACM-SAC 2015). We present this paper in the Appendix C of this thesis.

5. *Identifying factors that affect god class detection: a family of four controlled experiments.* This paper presents our analysis establishing cause-effect relationship between all independent variables addressed in the controlled experiments and the agreement on god class detection. We submitted this analysis to the Empirical Software Engineering journal (EMSE) in November, 2014. It is now on major review after the first feedback from the editorial team. The analysis presented in this paper is the same as we presented in the Chapter 4 of this thesis.

6. *Code smell effect: a thematic synthesis.* This paper presents our thematic synthesis. We will submit to a major journal in the area. The paper is on final design phase. We will submit it soon.

7. Lastly, we will submit a position paper, summarizing our main findings and presenting the research agenda to a major magazine in the area.

Table 6.1 summarizes the publishing planning for this thesis. It presents the title, source and the status for each paper.

Table 6.1 Publishing agenda for this thesis

Title of the paper	**Source**	**Type**	**Status**
An exploratory study to investigate the impact of conceptualization in god class detection	EASE 2103	Conference	Presented
The problem of conceptualization in god class detection: agreement, strategies and decision drivers	JSERD (Sep 2014)	Journal	Published
Identifying strategies on god class detection in two controlled experiments	SEKE 2104	Conference	Presented
Exploring decision drivers on god class detection in three controlled experiments	ACM-SAC 2015	Conference	Accepted
Identifying factors that affect god class detection: a family of four controlled experiments	EMSE (Nov 2014)	Journal	Submitted
Code smell effect: a thematic synthesis	Not defined	Journal	Outlining
Not defined (the position paper)	Not defined	Transaction	Outlining

6.3 FUTURE WORK

The complexity of the problem that we have addressed in this thesis opens different possibilities of study. We present some future works following the approaches we adopted in order to explore the lack of comprehension of problems related to the code smell effect, specially focusing on the research agenda that we presented.

These are some of the further works that we propose:

1. To discuss the nature of smells, investigating the existent catalogs, identifying the relevant characteristics of each smell, linking these characteristics with the practical problems arising from bad design, and grouping the smells according to these issues. Note that this involves a theoretical discussion and empirical observation of the results.

2. To explore context aspects affecting results of the empirical studies on smell effect: one of the possibilities is to explore these aspects from the primary studies of our SR, mapping all context characterizations of the studies and searching for correlations between then. For example, the studies focused on god class, using controlled experiments converge? If yes, what context aspects are different and what are common? How different is the training of the controlled experiments presenting divergent findings? As a result, we expect a mapping of the correlations between some of the contextual aspects.

3. To face the code smell conceptualization problem from the cognitive perspective. We propose a multidisciplinary research, involving specialists in theories of psychology related to cognition or in other areas of human sciences. This will make easier propose experiments investigating how developers conceptualize quality of design.

4. To face the code smell conceptualization problem exploring other factors affecting human evaluation of smells and their relationships. For example, it is possible to explore the relation between experience and knowledge on smell detection, or the relation between the training and experience, or other demographic factor.

5. To enrich our family of controlled experiments. It is possible to explore other aspects making some changes on the experimental setup of the experiments that we performed. It is also possible to explore other smells, other tool supporting design comprehension, other approach to evaluate the knowledge on the topic. The main topic that we intent to explore is the training, searching for an approach to better control the human evaluation of smells.

Appendix

A

THEMATIC MAP

In this appendix, we present the complete thematic map of our SR. We grouped the themes in different tables because the extensive volume of information. Table A.1 shows the thematic map for the theme “correlation with issues of development”, sub-theme “changes and defects”. The columns ID in the table are formed by the ID of the study and the number of the finding, that we identified sequentially during the extraction process. The other columns describe the sub-theme showing their codes and forming the thematic map. All other following tables have the same structure.

Table A.2 shows the thematic map for the theme “correlation with issues of development”, sub-themes “architectural quality” and “harm”. Table A.3 shows the thematic map for the theme “correlation with issues of development”, sub-themes “effort” and “design quality”. Table A.4 shows the thematic map for the theme “human aspects”, sub-themes “factors affecting detection”, “decision drivers” and “human evaluation versus software measures”. Table A.5 shows the thematic map for the theme “human aspects”, sub-themes “agreement on detection”, “strategy of detection”, “detection difficulty” and “knowledge about smells”. Table A.6 shows the thematic map for the theme “programming”, sub-themes “smell introduction” and “smell removal”. Table A.7 shows the thematic map for the theme “detection”, sub-themes “supporting smell detection”, “smell as a predictor” and “novel smell”. Table A.8 shows the thematic map for the theme “other correlations”, sub-themes “inter-smell relation”, “metrics versus smells”, “smell density” and “frequency of smells”.

Table A.1 Thematic map for the theme “correlation with issues of development”, sub-theme “changes and deffects”

ID	Changes and defects	ID	Changes and defects (cont. I)	ID	Changes and defects (cont. II)
S4.3	There are smells that are not associated with maintenance problems.	S21.3	Shotgun Surgery (and also God class and God method, in less intensity level) was the most bad smell that were associated with all severity levels of errors in all releases.	S17.1	The number of design problems (smells) increases as the system evolves.
S4.4	Feature Envy and God Method together were associated with time-consuming changes.	S1.2	The change proneness of components with smells is higher than the ones without..	S18.2	Design flaws cant be considered more harmful with respect to software defects.
S4.5	Changes of infected components with ISP Violation is significant larger	S1.3	The code churn of infected classes with god class is significant larger.	S18.3	In some systems there is no design flaw which strongly correlates with defects.
S7.1	The role of code smells on the overall system maintainability is relatively minor.	S1.4	Code smells evolve during the system development.	S18.4	Some systems are characterized by one flaw is correlated with defects much more frequently than all the others.
S7.12	The overlap between problems related to code smells and problems related to artifacts associated to code smells can explain causes of inconsistencies in empirical studies about maintainability.	S1.5	Classes infected seem to need more maintenance than non-infected classes.	S5.1	The change likelihood of god classes is higher than the change likelihood of non-god classes.
S7.13	Some difficulties on maintenance activities are associated to a combination of smells and other characteristics, whereas others were not associated code smells at all.	S9.1	There is a correlation between maintainability factors and code smell.	S5.2	The defect likelihood of god classes is higher than the defect likelihood of non-god classes.
S7.2	Code smells can help to explain and potentially identify some specific difficulties occurred during maintenance beforehand.	S10.1	There is no correlation between clones and bugs.	S5.4	God classes impacts maintainability.
S7.7	Feature Envy (which indicates efferent coupling) or ISP Violation (which indicates afferent coupling) can help to identify cross-cutting concerns situations.	S10.2	There is no evidence that clones are bad smells.	S19.1	Classes with code smells are more change-prone than others.
S21.1	Some bad smells are associated significantly with class error proneness.	S14.1	Taken in isolation, classes exhibiting the identity disharmonies design flaws do not have an increased likelihood to exhibit defects that classes which do not reveal design flaws.	S19.2	Specific smells are more correlated than others to change-proneness.
S21.2	The bad smells will not reveal all classes that contain errors.	S14.2	Classes exhibiting the identity disharmonies design flaws used by their clients shows the increase of likelihood for the clients to exhibit defects, especially for the post-release defects.		

Table A.2 Thematic map for the theme "correlation with issues of development", sub-themes "architectural quality" and "harm"

ID	Architectural	ID	Harm
S13.1	The detection strategies are inaccurate in identifying architecturally-relevant code anomalies.	S15.1	The presence of God and Brain Classes is not necessarily harmful.
S13.2	The detection strategies are more effective in systems where architecture conformance is more strictly enforced in the code.	S15.4	Presence of God class and Brain class may be benefical to a system.
S13.3	Architecturally-relevant code anomalies often occurred in code elements responsible for implementing different architectural elements.	S5.3	God classes have been confirmed to have the technical debt property.
S13.5	The detection strategies seem to have a tendency to send developers in wrong directions when addressing code anomalies related to architectural modularity problem.		
S13.6	The imperfection of the detection strategies (related to impact on architectural problems) is not simply related to specific thresholds or combinations of particular measures.		
S11.1	Refactoring strategies do not contribute to remove architectural-relevant code anomaly.		
S11.2	There is a correlation between code anomaly and problems in architectural design.		
S11.3	There is a correlation between code anomaly and architectural degradation.		
S11.5	There is no specific code anomaly more relevant as indicator os architecture degradation.		
S11.7	Architecturally-relevant code anomalies are left in the code when the system evolves.		

Table A.3 Thematic map for the theme “correlation with issues of development”, sub-themes “effort” and “design quality”

Correlation with issues of development			
ID	**Effort**	**ID**	**Design quality**
S8.1	Without any adjustments for file size and the number of changes smells were associated with more effort than files without these smells. (S8.2)	S4.7	The presence of a code smell may intensify or spread the effects of bad/limited design choices throughout the system.
S8.2	After adjustments for file size and the number of changes code smells are not significantly associated with increased effort. (S8.1)(S8.3)(S8.4)	S19.3	Code smells often related to immature design and implementation.
S8.3	File size and the number of changes explained almost all of the modeled variation in effort. (S8.2)	S7.6	Inconsistent design can be identified by detecting a combination of code smells.
S8.4	There is a correlation between effort and code churn. (S8.2)		
S8.6	Effort may decrease by reducing file size and improving work processes to reduce the number of revisions, instead refactoring.		
S3.4	God class don’t require a higher maintainability effort than non-code smells.		
S2.3	Software characteristics don’t impact effort on god class detection.		
S17.4	Maintenance effort should prioritize Long Method smells over other smells.		

Table A.4 Thematic map for the theme “human aspects”, sub-themes “factors affecting detection”, “decision drivers” and “human evaluation versus software measures”

ID	Factors affecting detection	ID	Decision drivers	ID	Human evaluation versus software measures
S2.1	Human characteristic impact agreement, effort and drivers’ choice on god class detection.	S3.6	“Moderate agreement” among the subjects detecting GC , for the misplaced method issue (also called driver).	S23.2	Developers’ evaluations of the smells do not correlate with the used source code metrics.
S23.3	Demographic data (knowledge, role, work experience) seemed to explain some of the variances in smell evaluations.	S3.7	Misplaced methods is the strongest driver for letting the subjects classify a class as a god class.	S23.5	Developers’ evaluations on the Large Class smell seem to be conflicting, when compared to large class measures.
S16.4	The knowledge acquired in one version effectively supported the identification of related code smells in other versions.	S3.8	Lack of cohesion and complexity are identified as issues that let the human subjects identify a class as a god class.	S22.3	The developers’ evaluations on code smells dont correlate with related source code metrics.
S22.2	Demographics aspects of the developers affect the code smell evaluations.			S22.6	The developers’ evaluations of the smells correlated better with the metrics for smells that are simple and easy to spot.
S22.5	Developers with better knowledge of the module evaluated that there is more of the Lazy Class smell that is difficult to spot.				
S24.2	The demographics were not useful predictors neither for evaluating code smells nor the refactoring decision. (and Erradication)				
S25.2	The more details people actually know about code smells, the more concerned they are by the presence of smells in their code.				
S26.2	The domain of an analyzed system is an important factor that should be included in the detection strategies for various code smells				
S26.6	Smells exhibit similar behavior in different software domains.				

Table A.5 Thematic map for the theme “human aspects”, sub-themes “agreement on detection”, “strategy of detection”, “detection difficulty” and “knowledge about smells”

ID	Agreement on detection	ID	Strategy of detection	ID	Detection difficulty	ID	Knowledge about smells
S3.2	The agreement on GC classification and detection is low.	S16.3	Participant with optimistic style investigate code smell candidates using more combinations of views when compared to participants with more conservative styles.	S3.1	The subjects perceive detecting god classes as an easy task.	S25.1	There is not strong understanding about code smells and practical application of these concepts.
S2.2	The agreement on god class detection is low.					S25.3	Duplicated code was by far the most mentioned smell.
S23.1	Software developers show low agreement about “smelliness” of the source code.					S25.6	Respondents who were extremely or moderately concerned about smell benefits gave as rationale reasons like product evolvability, end-product quality, and developer productivity
S16.2	The participants had different perceptions and judgments during the code smells identification.						
S22.1	Developers have not a uniform opinion on the ”smelliness” of the source code.						
S24.1	There is partial concordance among the evaluators in all evaluations.						

Table A.6 Thematic map for the theme “programming”, sub-themes “smell introduction” and “smell introduction”

ID	Smell removal	ID	Smell introduction
S8.5	The developers who revised most often also introduced the most smells.	S8.5	The developers who revised most often also introduced the most smells.
S6.7	There is no clarity on the intent of developers prioritize or even consider the eradication of code smells.	S11.6	Early code anomalies induces architectural anomalies.
S11.7	Architecturally-relevant code anomalies are left in the code when the system evolves.	S17.2	Smells are introduced during the initial design/implementation.
S12.1	The developers resolve code smells for opportunistic reasons.	S20.1	A large proportion of GCs seems to be introduced as a conscious design decision by developers.
S12.2	Code smells of early revisions are resolved within a few revisions.	S20.4	Changes can result in the degradation of GCs.
S17.3	Designers perform refactorings routinely based on their subjective perception of problematic code areas rather than applying them as solutions to identified design problems.	S20.3	The correction of a GC may also move the problem to a different class
S20.2	Specific maintenance activities can eliminate GCs when they are accidents.		
S20.3	The correction of a GC may also move the problem to a different class		
S24.2	The demographics were not useful predictors neither for evaluating code smells nor the refactoring decision.		

Table A.7 Thematic map for the theme “detection”, sub-themes “supporting smell detection”, “smell as a predictor” and “novel smell”

ID	**Supporting smell detection**	**ID**	**Smell as a predictor**	**ID**	**Novel smell**
S3.3	Automatic detection accompanied by a manual review increases the overall confidence in the results of metric-based classifiers.	S13.1	The detection strategies are inaccurate in identifying architecturally-relevant code anomalies.	S7.8	A new smell was propose: Multiple Inheritance simulation.
S3.5	If provided with a suitable process, humans can detect code smells in an effective fashion.	S13.3	The detection strategies are more effective in systems where architecture conformance is more strictly enforced in the code.		
S11.4	Developers need support to enhance detection of code anomaly.	S13.5	The detection strategies seem to have a tendency to send developers in wrong directions when addressing code anomalies related to architectural modularity problem.		
S16.1	The use of the visual representation of concerns in the multiple views approach provided useful means to visually spot some kinds of code smells more than others.	S6.4	Expert judgment is most accurate method for assessing system maintainability.		
S22.4	Using source code metrics in conjunction with human evaluations is likely to be the best alternative.	S6.5	Code smells approach can detect design flaws that may be overlooked by experts.		
S25.4	Software professionals who are interested on code smells and anti-patterns expressed a need for better support, including tools, during the software evolution cycle.	S6.6	Software measures and expert judgment addressed different aspects of maintainability in a system, and combining them can lead to more complete evaluations of maintainability.		
S25.5	Respondents indicated that they use technical blogs, programmer forums, colleagues and industry seminars as their main sources of information.	S9.2	Code smells would need to be complemented with alternative approaches such as semantic analysis and manual inspection for address maintainability factors		
S26.3	Accuracy of code smells detection rules used by the tools can be improved by exploiting knowledge about both the domain and the design of a system.	S9.3	Code smells can cover a more heterogeneous spectrum of factors than software metrics and expert judgment individually.		
		S23.4	The use of the smells for internal software quality assessment should be questioned		
		S19.4	Smells can provide to developers recommendations easier to understand than what metric profiles can do.		
		S19.5	Smells are not replacement to metrics in the ability of building change-proneness or fault-proneness prediction models		

Table A.8 Thematic map for the theme "other correlations", sub-themes "inter-smell relation", "metrics versus smells", "smell density" and "frequency of smells"

Other correlations							
ID	**Inter-smell relation**	**ID**	**Metrics versus smells**	**ID**	**Smell density**	**ID**	**Frequency of smells**
S4.1	There is interaction between code smells.	S3.9	Automated metric-based preselection of smells decreases the effort spent on manual code inspection.	S6.2	Code smell density is likely to be inaccurate when comparing size differing systems.	S15.2	The larger systems have a larger proportion of God and Brain Classes.
S4.2	There is a correlation between inter-smells smells and maintenance problems.	S6.1	Code smells are strongly influenced by size (LOC).	S6.3	Code smell density distinguishes maintainability across similar sized systems.	S18.1	Some design flaws are more frequent than others.
S4.6	Studies into the effects of inter-smell relations are a topic that deserves more attention.	S7.11	Wide coupling displayed Feature Envy or ISP Violation and to some extent Shotgun Surgery.	S15.3	Size normalizing impacts evaluation of change frequency of God class and Brain class.	S18.5	Feature envy is the most frequent design flaw, but it is not the most correlated with software defects.
S7.10	Observing combinations of code smells could be useful to discriminate instances of God Classes that are potentially problematic, against instances of no problematic god class.	S7.9	Large size exhibited either the God Class smell or the God Method smell and in most cases a combination of both smells.	S5.5	There isn't a linear relationship between class size and change likelihood that justifies LOC-normalization.	S26.1	Some smells are more prevalent than others
S7.13	Some difficulties on maintenance activities are associated to a combination of smells and other characteristics, whereas others were not associated code smells at all.	S1.1	There is a correlation between size and number of: god class and shotgun surgery.				
S7.5	Interaction effects amongst collocated smells and coupled smells should be taken into account during analysis.	S24.3	Simple code smells and source code metrics have a relationship.				
S7.6	Inconsistent design can be identified by detecting a combination of code smells.	S26.4	There is correlation between several code smells and selected metrics.				
		S26.5	The strength of correlation between code smells and metrics varies with respect to the domain				

Appendix

B

PAPER PRESENTING A DETAILED ANALYSIS OF FING 1

This appendix shows the paper "The problem of conceptualization in god class detection: agreement, strategies and decision drivers". The paper was published in the Journal of Software Engineering Research and Development (JSERD), in September, 2014.

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH Open Access

The problem of conceptualization in god class detection: agreement, strategies and decision drivers

José Amancio M Santos[1*†], Manoel Gomes de Mendonça[2,3†], Cleber Pereira dos Santos[4†] and Renato Lima Novais[4†]

*Correspondence:
zeamancio@ecomp.uefs.br
†Equal contributors
[1] Department of Technology, State University of Feira de Santana, Transnordestina avenue S/N - Feira de Santana - Bahia, Feira de Santana, Brazil
Full list of author information is available at the end of the article

Abstract

Background: The concept of code smells is widespread in Software Engineering. Despite the empirical studies addressing the topic, the set of context-dependent issues that impacts the human perception of what is a code smell has not been studied in depth. We call this the code smell *conceptualization problem*. To discuss the problem, empirical studies are necessary. In this work, we focused on conceptualization of god class. God class is a code smell characterized by classes that tend to centralize the intelligence of the system. It is one of the most studied smells in software engineering literature.

Method: A controlled experiment that extends and builds upon a previous empirical study about how humans detect god classes, their decision drivers, and agreement rate. Our study delves into research questions of the previous study, adding visualization to the smell detection process, and analyzing strategies of detection.

Result: Our findings show that agreement among participants is low, which corroborates previous studies. We show that this is mainly related to agreeing on what a god class is and which thresholds should be adopted, and not related to comprehension of the programs. The use of visualization did not improve the agreement among the participants. However, it did affect the choice of detection drivers.

Conclusion: This study contributes to expand empirical evidences on the impact of human perception on detecting code smells. It shows that studies about the human role in smell detection are relevant and they should consider the conceptualization problem of code smells.

Keywords: Code smell; God class; Controlled experiment; Code visualization

Background

Challenges in object-oriented (OO) software design have been historically addressed from different perspectives. Riel (1996) wrote one of the first books on the subject in 1996. This book presents insights into OO design improvements and introduces the now well-known term "design flaws". In 1999, Fowler (1999) came up with the concept of refactoring and coined the term "smell" to represent bad characteristics observable in the code. In 2005, Lanza and Marinescu (2005) focused on OO metrics to characterize what they called “disharmonies”. All these terms are used to define potential design problems. In this paper, we adopt the term *code smell*, or simply *smell*, to refer to such design problems.

The works of (Riel 1996; Fowler 1999 and Lanza and Marinescu 2005) discuss code smells from the principles of the OO paradigm, such as information hiding or polymorphism (Meyer 1988). However, there is a set of context-dependent issues that impacts how one considers the concept of smell. These include: developers’ experience, the software process, software domain, and others. The extensive number of context-dependent issues make it difficult to express even simple tasks rigorously, such as smell detection. Fontana et al. (2011) claim that smell detection “can provide uncertain and unsafe results”. This is because most smells are subjectively defined and their identification is human-dependent.

Fowler (1999) does not define smell formally. He says that one needs to develop one’s own sense of observation of attributes that could characterize pieces of code as a smell. For example, one has to develop one’s own sense of how many lines of code define a long method. This is because smell detection is a subjective task by nature. In contrast, Lanza and Marinescu (2005) use a formal definition for smells based on metrics and thresholds. However, Rapu et al. (2004) state that the thresholds are mainly chosen based on the experience of the analysts. These indicate that smell detection remains an ill-defined task. In addition, as cited by Parnin et al. (2008), “metrics produce voluminous and imprecise results”. Given these pitfalls, alternatives have emerged to address smell detection. One of them is the use of software visualization (Murphy-Hill and Black 2010; Parnin et al. 2008; Simon et al. 2001; Van Emden and Moonen 2002). Software visualization tools combined with metrics may help humans to identify design problems.

Understanding which, and how, subjective aspects affect smell detection demands empirical evaluation. According to Mäntylä, “we need more empirical research aiming at critically evaluating, validating and improving our understanding of subjective indicators of design quality” (Mäntylä et al. 2004). Recent empirical studies carried out to better understand this scenario, can be classified into three categories. The first type is correlation studies. They evaluate the impact of smells based on data extracted from software repositories (Li and Shatnawi 2007; Olbrich et al. 2009, 2010), establishing a correlation between a smell and some attribute of the software, such as bugs or the number of modifications on classes. The second type is related to tool assessment, such as automatic detection (Moha et al. 2010; Mäntylä and Lassenius 2006a; Schumacher et al. 2010) or software visualization (Carneiro et al. 2010; Murphy-Hill and Black 2010; Parnin et al. 2008; Simon et al. 2001). Finally, the third type investigates the role of humans in smell detection (Mäntylä 2005; Mäntylä and Lassenius 2006b; Santos et al. 2013; Schumacher et al. 2010). Although the number of studies on this topic is increasing, they are considered insufficient (Schumacher et al. 2010; Sjøberg et al. 2013; Zhang et al. 2011). In particular, the role of humans has not been studied in depth (Mäntylä and Lassenius 2006a).

The role of humans is one of the most important and one of the most open and broad topics in this area. Several uncontrollable variables affect the smell conceptualization. Examples include experience, personal differences in cognition, level of knowledge on the subject, and the environment. In this context, this work aims to understand how some aspects of the human conceptualization impact smells detection. In particular, this exploratory study investigates how personal comprehension of the smell concept affects the detection of god classes. God class is a term proposed by Riel (1996) to refer to classes that tend to centralize the intelligence of the system. Fowler (1999) defined it as a class that tries to do too much, and adopted the term large class. Lanza and Marinescu (2005) proposed a heuristic based on metrics to identify god class. The definition of god class has been addressed in several empirical studies (Abbes et al. 2011; Li and Shatnawi 2007; Olbrich et al. 2010; Padilha et al. 2013).

Our exploratory study was based on a controlled experiment carried out in an in-vitro setting. The experiment extended an empirical study presented by Schumacher et al. in (Schumacher et al. 2010). While Schumacher et al. presented a wide discussion about both human and automatic detection of god classes, our work focused on questions related to the human perception. We considered two factors, the use or non use of software visualization to achieve the study goal. The use of visualization made it possible to evaluate the impact of the overall comprehension of the design on human aspects, increasing the evidences of the relevance of conceptualization in the detection of god class. It is important to note that we did not adopt a visualization tool to support smell detection, instead, we adopted a tool focused on enhancing the comprehension of the code design. The tool helps developers to perceive of coupling, size, complexity, and hierarchical relations among classes from the use of visual resources. We evaluated the effect of these facilities on human aspects, such as decision drivers, agreement and strategies adopted by the participants detecting god classes. To the best of our knowledge, this is the first study that analyses smell conceptualization using all of these variables. We have already featured this experiment partially (Santos et al. 2013), addressing effort, decision drivers, and agreement on smell detection, but disregarding the use of visualization.

The structure of this paper is as follows. Section 'Method' presents the planning and execution of the experiment. Section 'Results' and 'Discussion' present the results and a discussion about them. Section 'Threats to validity' discusses the threats to the validity of the study. Section 'Context and related works' summarizes prior empirical studies that address aspects which are context-related to smells. Lastly, Section 'Conclusions' presents our conclusions and proposes future works.

Method

In this section, we present the experimental planning and execution of the experiment.

In order to attend ethical issues on Empirical Software Engineering, we followed principles proposed by (Vinson 2008).

Research question

Our work aims to investigate the impact of conceptualization on god class detection. The research questions (RQ) are:

1. *How well do humans agree on identifying god classes?*
2. *How well do humans and an oracle agree on identifying god classes?*
3. *Which strategies are used to identify god classes?*
4. *What issues in code lead humans to identify a class as a god class?*

All questions help us to observe the differences in perception among the participants during the detection of god classes. They were used to observe how conceptualization affects the identification of god classes in our experiment. Research questions one and four were first proposed by Schumacher et al. (2010). In this paper, we analyzed them using a different approach: by the use of both code review and visualization. We have introduced question two and three. Question two addresses agreement between participants and an oracle, defined in a controlled process. Question three addresses the strategies adopted by participants when detecting god classes.

Experimental units

The experiment involved 11 undergraduate Computer Science students from the Federal University of Bahia (UFBA), in Brazil. All students were enrolled in the Software Quality course offered in the first semester of 2012. This is an optional subject of the Computer Science program, in which design quality and smells are addressed. The course was considered appropriate for the experiment, both because it was focused and was not mandatory, which means that most students enrolled on it were interested in the subject. Furthermore, participation in the experiment was a voluntary activity.

Experimental material

Tools

We adopted four software tools in the experiment[a]: (i) The Eclipse Indigo IDE; (ii) Usage Data Collector (UDC), an Eclipse plug-in for collecting IDE usage data information (interactions between participants and Eclipse can be accessed by the log of UDC). This tool is embedded in the Eclipse Indigo IDE; (iii) Task Register plug-in, a tool we developed to enable participants to indicate what task was being done at any given moment. This information was also registered in the UDC log. All the participants had to do was to click on a "Task Register" view on Eclipse (Figure 1-F) to indicate when they were starting or finishing a task; and (iv) SourceMiner, an Eclipse plug-in that provides visual resources to enhance software comprehension activities (Carneiro and Mendonça 2013; Carneiro et al. 2010).

SourceMiner has five views, divided into two groups. The first group is made up of three coupling views. These views show different types of dependencies among entities, like direct access to attributes or method calling, for instance. Moreover, they show the

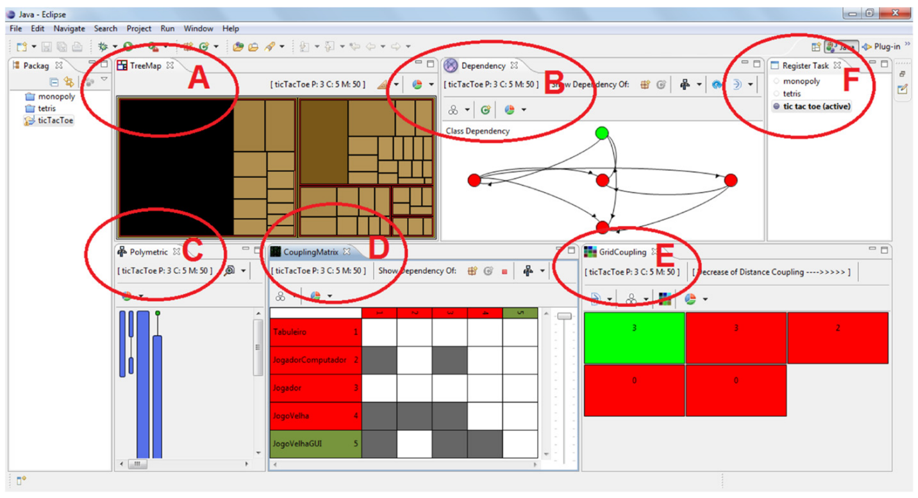


Figure 1 Eclipse IDE showing the Task Register (F) plug-in and SourceMiner views: hierarchical views (A and C) and coupling views (B, D and E).

direction of the coupling. The coupling views are based on radial graphs (Figure 1-B), matrix of relationships (Figure 1-D), and tabular view (Figure 1-E). The second group is made up of two hierarchical views. These views associate the number of lines of code, complexity and number of methods of classes to the area and colors of the rectangles. The Treemap view (Figure 1-A) shows the hierarchy of package-class-method of the software. A Treemap is a hierarchical 2D visualization that maps a tree structure into a set of nested rectangles (Johnson and Shneiderman 1991). In SourceMiner, the nested rectangles represent software entities, like packages, classes and methods. Rectangles representing methods of the same class are drawn together inside the rectangle of the class. Likewise, the rectangles of the classes that belong to the same package are drawn together inside the rectangle of the package. The Polimetric view (Figure 1-C) shows the hierarchy between classes and interfaces. A polymetric view uses a forest of rectangles to represent the inheritance trees formed by classes and interfaces in a software system (Lanza and Ducasse 2003). Rectangles are linked by edges representing the inheritance relationship between them. The length and width of the rectangles can be used to represent software attributes such as the size and number of methods of a class.

Forms

We used five forms and two guides during the experiment. During the training, which we will present thereafter, the participants filled in a Consent and a Participant Characterization form, and received a SourceMiner exercise guide. During the experiment itself, participants received a Support Question guide to steer them in search for god classes. The questions are the same ones used by Schumacher et al. (2010):

- Does the class have more than one responsibility?
- Does the class have functionality that would fit better into other classes?
    - By looking at the methods, could one ask: "Is this the class' job?"
- Do you have problems summarizing the class' responsibility in one sentence?
- Would splitting up the class improve the overall design?

Another document used during the experiment was the Step-by-Step guide used to assure consistency during the data collection process. This document prompts the participants to open and close the projects on Eclipse and to select the task under execution in the Task Register view. It is important to note that this guide did not define how participants should do the task of identifying god classes. It defined some activities that participants had to do before and after the identification of god classes. Each participant defined their own strategy to identify god classes.

The Answer form was one of the most important forms used during the experiment. On this form, participants had to fill in: i) one or more candidate(s) god class(es), ii) their level of certainty, i.e., fill in the "yes" or "maybe" option for each candidate class, iii) the decision driver(s) which helped her/him to select the class, and iv) the start and end time of the task, which we used to evaluate effort in (Santos et al. 2013) (this topic is out of the scope of this paper).

We considered the items i) the candidate god classes and ii) level of certainty, which is self-explanatory. Here, we will explain item iii) decision drivers: the form listed nine drivers as predefined options for the participants, but it was also possible to write down a new one. The drivers listed in the form are the ones identified by Schumacher et al. (2010) during their think-aloud data collection. Some examples are “method is misplaced” and “class is highly complex”.

The last form of FinG was the Feedback form. At the end of the experiment the participants filled in a Feedback form. On it, we asked the participants to classify the training and the level of difficulty performing the task. It was also possible to write down suggestions and observations about the experiment.

Software artifacts

Six programs were used in the experiment. All of them implement familiar applications or games in Java e.g. Chess, Tic Tac Toe, Monopoly and Tetris. Solitaire-Freecell (Solitaire) is a framework for card games with Solitaire and Freecell. Jackut implements a simple social network application, such as Facebook and Orkut.

During our selection, we looked for familiar applications to minimize the effort of participants during the task of identifying god classes. However, we also looked for programs with different characteristics: some without god classes, others which perhaps had god classes and others with, at least, one god class. An oracle was used to identify god classes in each of the selected programs. This oracle is presented later. Table 1 characterizes the programs we used in terms of the number of packages, number of classes and number of lines of code (LOC).

Task

While Schumacher et al. (2010) designed a mini-process for participants to detect god classes, we gave only support questions as a guide. Each participant was free to use her/his own strategy to do the task. Furthermore, they could choose the order of the tasks. The Step-by-Step guide only indicated the activities to be done before and after the god class detection for each program, such as filling in the start and finish time on the answer form.

Design

The experiment was carried out in a laboratory at UFBA. Participants had 2 hours to carry out the task. Each participant worked at a separate workstation. At each workstation, we set up two Eclipses IDEs. One contained the SourceMiner plug-in, and the other did not. Both Eclipses were fitted with a Task Register and UDC plug-ins. Each Eclipse had three of the six programs in their workspace. The workstations were divided into two groups. After analysis of the characterization form, participants were randomly allocated to groups, because they had similar profiles: years of programming and knowledge on the topic were used. There were six participants in group 1 and five participants in group 2. We present the distribution of participants by group in Table 2. The ID of the participants was formed by the position of the workstation in the lab where the task was performed.

Table 1 Software objects

Software	Chess	Jackut	Tic Tac Toe	Monopoly	Solitaire	Tetris
Packages	5	8	2	3	6	4
Classes	15	19	5	10	23	16
LOC	1426	978	616	2682	1758	993

Table 2 Set of Eclipses installations at the workstations and allocation of participants

Group	With SourceMiner	Without SourceMiner	Participants' ID
1	Chess, Jackut and Solitaire	Monopoly, Tetris and Tic Tac Toe	F14, F21, F32, F35, F42 and F44
2	Monopoly, Tetris and Tic Tac Toe	Chess, Jackut and Solitaire	F13, F15, F25, F31 and F41

Execution

The experiment took four days. Two days were allocated to training, one day for a pilot and one day to perform the experiment. There were three small presentations on the first day of training. As the experiment was a voluntary activity and students had little experience with experimental software engineering, we decided to do a motivational presentation. In this presentation, we discussed the experimental software engineering scene, tying it with discussions about smell effects. The second presentation focused on smells and god class concepts. The third presentation showed the design of the experiment: we just talked about the lab, the individual use of a workstation with two different set ups of the IDE, and the time.

On the second day of training, we did an activity in the lab focusing on the SourceMiner tool (explanation and exercise).

On the third day, we ran a pilot experiment with two students who were also enrolled in the same course of the participants. These two students were out of the 11 we presented in the Table 2. The pilot helped us to evaluate the use of the answer form in paper or electronic format. In the pilot, we presented the Step-by-Step guide in paper format. We did this because we thought that it would not be useful to use electronic format forms because the experiment ran with two opened Eclipses installations. However, after the pilot, we noted that paper forms were less convenient. Therefore, in the final experiment, we kept only the answer sheet form and Support Questions in paper format. The pilot also helped us to validate the inspection time. We confirmed that 1.5 - 2.0 hours was enough time to analyze the six programs.

On the final day we ran the experiment. Table 3 shows the complete schedule. The column Day gives an idea of the time between the activities. For example, the second training (Day 8) was seven days after the first training (Day 1).

Table 3 Experiment schedule

Day	Activity	Presentation	Local	Time (Hour)
1	Training	Motivational + Concepts + Experiment design	Classroom	2,0
8	Training	SourceMiner + Exercise	Lab Lab	2,0
18	Pilot	-	Lab	1,5
20	Experiment	-	Lab	2,0

Deviations

We ran the experiment with 17 students, but only 11 completed the experiment. Four students missed at least one presentation and were excluded. Two students participated in the pilot experiment. We also had an unexpected problem with the schedule. The original schedule was changed and there was a holiday between the SourceMiner presentation and experiment. Due to this, there were only 2 days between the pilot and the experiment. Despite this, the pilot still helped us, as previously discussed.

Data

We collected and analyzed two types of data. The first was the answers on the answer form, as explained in the Section 'Forms': i) the selected god class candidates; ii) the level of certainty, i.e. "yes" or "maybe" option for each candidate class; and iii) decision drivers that helped participants to indicate the candidate god classes.

The other type of data was the UDC Log. We used the UDC plug-in to log participants' actions while the experiment was running. UDC is a framework for collecting usage data on various aspects of the Eclipse workbench. It gathers information about the kinds of activities that the user does in the IDE (i.e. activating views, editors, etc.). The Task Register (Figure 1-F) was used to enrich the UDC log with the name of the program on which the participant was performing the task. Figure 2 shows a clipping of the UDC log annotated by the Task Register plug-in. The first column ("task") does not exist in the original UDC log. It was added by the Task Register. We highlighted columns that we were interested in. The first column ("task") indicates the program for which the participant was doing the god class detection task. Columns "what", "kind" and "description" describe the actions. For example, the first line represents: user activated the Package Explorer view.

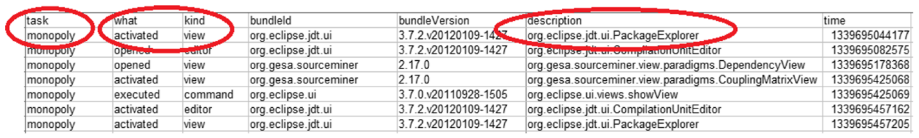
Each line represents one user action. As a result, we have sequences of actions for each participant and for each program.

Results

This section presents the results of the experiment. We created one subsection to present the results for each research question.

RQ1: How well do the participants agree on identifying god classes?

To evaluate this research question we considered the agreement on the candidate god classes for both cases, with and without visualization. For the case without visualization, there were six participants in group 1 (Monopoly, Tetris and Tic Tac Toe) and five in group 2 (Chess, Jackut and Solitaire). In the case of with visualization, there were six participants in group 1 (Chess, Jackut and Solitaire) and five participants in group 2 (Monopoly, Tetris and Tic Tac Toe). We tabulated god class candidates into the "yes" and "maybe" category



task	what	kind	bundleId	bundleVersion	description	time
monopoly	activated	view	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.PackageExplorer	1339695044177
monopoly	opened	editor	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.CompilationUnitEditor	1339695082575
monopoly	opened	view	org.gesa.sourceminer	2.17.0	org.gesa.sourceminer.view.paradigms.DependencyView	1339695178368
monopoly	activated	view	org.gesa.sourceminer	2.17.0	org.gesa.sourceminer.view.paradigms.CouplingMatrixView	1339695425068
monopoly	executed	command	org.eclipse.ui	3.7.0.v20110928-1505	org.eclipse.ui.views.showView	1339695425069
monopoly	activated	editor	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.CompilationUnitEditor	1339695457162
monopoly	activated	view	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.PackageExplorer	1339695457205

Figure 2 Clipping of the user UDC log.

for each participant. Table 4 summarizes the results for the Solitaire program. The F13 participant, for example, marked one class as a "yes" and one class as a "maybe" god class. To analyze the results, we consider both to be god class candidates.

Out of all of the data sets, there were ten cases in which participants filled in the class name and the drivers, but did not mark the option "yes" or "maybe". In these cases, we considered the weakest option, i.e., "maybe". There were also two cases in which the participants did not fill in the name of the class. In this case, we excluded the data entry from the analysis.

We used two approaches to address the research question. The first considered the percentage of candidate classes. The second was an agreement test.

The percentage of god class candidates

To analyze and compare cases with and without visualization, we generated bar chart diagrams (Figure 3). The diagrams show the percentage of candidate god classes with respect to the number of classes for each program. We show the number of classes for each program under the name of the programs. Considering "yes" or "maybe" options, the percentage of god class candidates tends to be higher for programs with a fewer number of classes. In the case of without visualization, the blue bars, there is a small difference for the Tetris program. However the tendency is the same: the percentage of god class candidates tends to be higher for programs with a fewer number of classes. In the case of with visualization, the gray bars, the difference is for the Solitaire program.

Considering only the "yes" option, the difference in the percentage of god class candidates is very small among the programs. Despite this, in the case without visualization, the percentage of god class candidates tends to be higher for programs with a fewer number of classes, once again. For the case with visualization, we noted that the values are very similar, excepted for the Monopoly program, which had the lowest percentage of god class candidates.

Finn agreement test

To evaluate the level of agreement among participants, we adopted the Finn coefficient (Finn 1970) as opposed to the Kappa coefficient (Fleiss 1971), adopted by Schumacher et al. (2010). We did this because of the problems identified in the Kappa coefficient by other authors (Feinstein and Cicchetti 1990; Gwet 2002; Powers 2012; Whitehurst 1984). The Kappa test is done in two phases. First an agreement rate is calculated, and, then this value is used to calculate the coefficient. Feinstein (1990) shows that one can have high

Table 4 God classes for participants working on the Solitaire-Freecell program (task carried out without visualization)

	Solitaire-Freecell (23 classes)				
God Class	**Participant**				
	F13	**F15**	**F25**	**F31**	**F41**
ControladorGlobal	Yes		Yes	Maybe	Yes
FrameFreecell	Maybe	Yes	Yes		Maybe
InterfacePaciencia		Maybe	Maybe		Maybe
Estoque		Maybe			

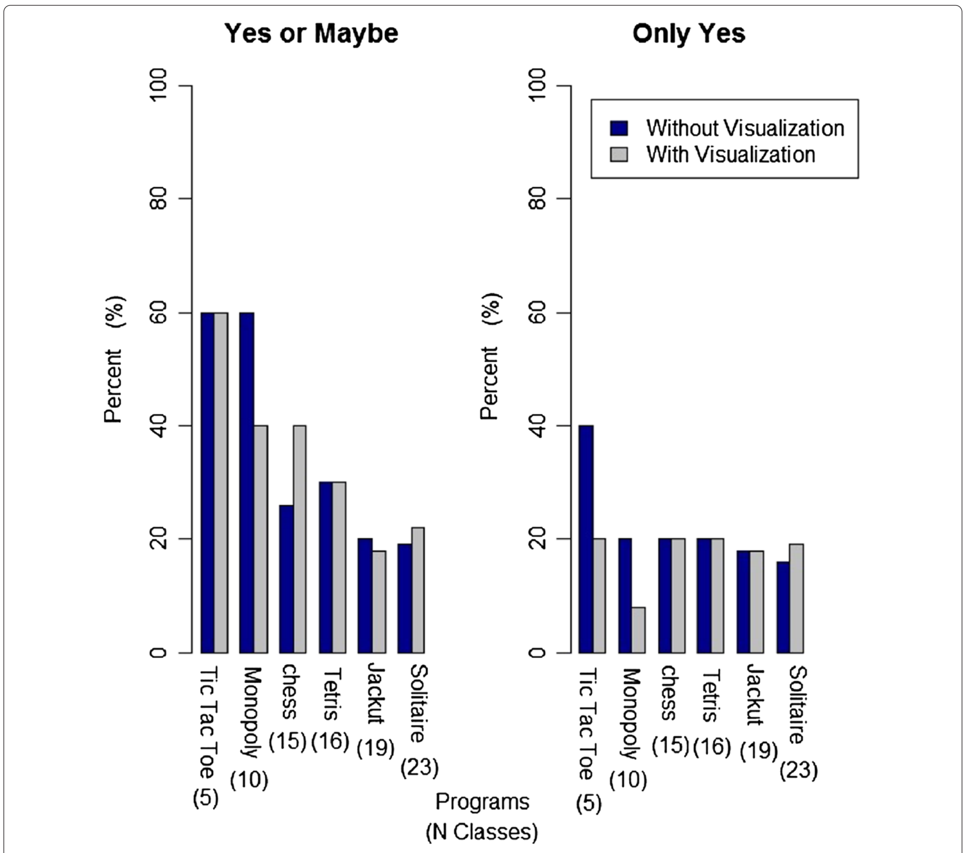


Figure 3 Percentage of god class candidates.

agreement rate and low values of the Kappa coefficient, when the variance on values of raters is low. We noted this situation in the work of Schumacher et al. The Finn coefficient is recommended when variance between raters is low (Finn 1970). Whitehurst (1984) proposes Finn as an alternative to problems with Kappa, and affirms that it is the most reasonable index for agreement.

To make the comparison of agreement values for the cases with and without visualization easier, we adopted classification levels. We used the same defined by Landis and Koch (1977), such as Schumacher et al. (2010) had done. Landis and Koch proposed the following classification: slight, for values between 0.00 and 0.20; fair (between 0.21 and 0.40); moderate (between 0.41 and 0.60); substantial (between 0.61 and 0.80); and almost perfect (between 0.81 and 1.00) agreement.

Table 5 presents Finn values for the programs considering "yes" or "maybe" options. On the left, we present values for the case of without visualization. On the right, we present values for the case with visualization. There were two cases where the agreement level was higher with visualization: Monopoly (from moderate to substantial) and Jackut (from substantial to almost perfect). There was a reduction in agreement for Tic Tac Toe (from moderate to slight), but it is the only change without significance (p-value = 0.283), therefore we did not consider it in the agreement analysis. Table 6 considers only cases where participants were sure about the god classes. There were two cases where the level of agreement was higher with visualization (Tic Tac Toe and Chess, changing from

Table 5 Agreement among participants, considering "yes" or "maybe" marked

Program	Without visualization					With visualization				
	Number of classes	**Raters (participants)**	**Finn coefficient**	**p-value**	**Agreement**	**Number of classes**	**Raters (participants)**	**Finn coefficient**	**p-value**	**Agreement**
Monopoly	10	6	0.507	0.000996	Moderate	10	4	0.7	7.37e-05	Substantial
Tetris	16	6	0.733	5.09e-12	Substantial	16	5	0.8	4.91e-13	Substantial
Tic Tac Toe	5	6	0.6	0.00335	Moderate	5	5	0.2	0.283	Slight
Chess	15	5	0.787	1.21e-11	Substantial	15	6	0.664	4.46e-08	Substantial
Jackut	19	4	0.772	1.73e-10	Substantial	19	6	0.881	1.33e-27	Almost perpect
Solitaire	23	5	0.843	4.38e-22	Almost perfect	23	5	0.843	4.38e-22	Almost perpect

Table 6 Agreement among participants, considering only "yes" marked

Program	Without visualization					With visualization				
	Number of classes	**Raters (participants)**	**Finn coefficient**	**p-value**	**Agreement**	**Number of classes**	**Raters (participants)**	**Finn coefficient**	**p-value**	**Agreement**
Monopoly	10	6	0.827	8.44e-12	Almost perpect	10	4	0.867	3.87e-09	Almost perpect
Tetris	16	6	0.892	5.34e-25	Almost perpect	16	5	0.675	5.89e-08	Substantial
Tic Tac Toe	5	6	0.653	0.00102	Substantial	5	5	0.84	7.14e-06	Almost perpect
Chess	15	5	0.787	1.21e-11	Substantial	15	6	0.84	5.26e-18	Almost perpect
Jackut	19	4	0.842	3.31e-14	Almost perfect	19	6	0.895	6.683-30	Almost perpect
Solitaire	23	5	0.896	3.62e-29	Almost perfect	23	5	0.843	4.38e-22	Almost perpect

substantial to almost perfect agreement). There was one case where the level of agreement was lower with visualization (Tetris changing from almost perfect to substantial).

RQ2: How well do humans and an oracle agree on identifying god classes?

To deepen the analysis of the impact of conceptualization on god class identification, we extended the human performance questions in Schumacher et al. (2010) using an oracle and comparing the answers of the oracle and participants. The oracle was made up of two experienced researchers in academia and industry. Each of the researchers did the task independently and without any contact with the participants' answers. We show their answers in Table 7.

We used the Finn coefficient (1970), the same as the previous RQ, to test the agreement. Table 8 shows the results. Tic Tac Toe, Tetris and Solitaire all had an agreement. The Chess and Jackut programs had one disagreement. The Monopoly program had two disagreements.

After these observations, the researchers met to discuss the differences and to define the oracle. An interesting observation is that the researchers noted that they were very strict in their analysis. Due to this, for some cases (two classes for Monopoly, one class for Chess, one class for Solitaire and one class for Jackut), classes were deleted from the list. We highlight the class FrameFreeFreecell in the Solitaire program. In this case, both researchers found that the class was a candidate to be god class. However, during the meeting, the researchers were more flexible about the size and the few methods out of scope, because the class represents the graphical user interface of the program. After the meeting, the oracles reached the agreement presented in Table 9.

Figure 4 shows the distribution of the Finn coefficient, comparing the agreement among the participants and the oracle. Let us initially focus on Figure 4(A) and (B), confirmed and possible god classes. It is possible to note that the average agreement with visualization

Table 7 Oracle answers

Program	God class	Oracle	
		Or1	Or2
Ti Tac Toe (5 classes)			
	-	-	-
Monopoly (10 classes)			
	Jogo	Yes	Yes
	Tabuleiro	Maybe	Yes
	UserStoriesFacade	-	Maybe
	Jogador	-	Maybe
Chess (15 classes)			
	Chess	Yes	Yes
	BoardGUI	-	Yes
Tetris (16 classes)			
	Tetris	Maybe	Maybe
Jackut (19 classes)			
	Usuario	-	Maybe
Solitaire (23 classes)			
	FrameFreeFreecell	Maybe	Yes

Table 8 Finn coefficient among the researches considering "yes" marked and "yes" or "maybe"

			Finn coefficiente	
Program	**Subjects (Nclasses)**	**Raters (oracle)**	**Yes or maybe**	**Only yes**
Ti Tac Toe (5 classes)	5	2	1	1
Monopoly	10	2	0.6	0.8
Chess	15	2	0.867	0.867
Tetris	16	2	1	1
Jakut	19	2	0.895	1
Solitaire	23	2	1	0.913

is higher only for the Monoploy program, is the same for Tic Tac Toe, and is lower for the other four cases. Figure 4(C) and (D) focus on confirmed god classes. First, one should notice that, as expected, the average is higher and the variances are smaller than in Figure 4(A) and (B). Comparing Figure 4(C) and (D), the values are higher for cases with visualization for three cases (Chess, Solitaire and Tic Tac Toe), and lower for the other three cases (Jackut, Monopoly and Tetris).

RQ3: Which strategies are used to identify god classes?

We used the logged actions to investigate the strategy adopted by the participants to detect god classes. We analyzed two aspects. The first one was the differences between relevant actions for the cases, with and without visualization. The second were the preferences of the participants. We grouped participants with similar preferences and evaluated if there was a "better" strategy.

Relevant actions

Table 10 shows the number of actions performed during the experiment by all participants. We grouped the columns "what", "kind" and "description" from the log and shortened the terms presented in Figure 2. The first action represents the activation of the Package Explorer view. The Package Explorer shows the Java element hierarchy of Java projects. It is a tree view that shows Eclipse projects on the first level, folders on the

Table 9 Final oracle answers

Program	God class	Oracle
Ti Tac Toe (5 classes)		
	-	-
Monopoly (10 classes)		
	Jogo	Yes
	Tabuleiro	Maybe
Chess (15 classes)		
	Chess	Yes
Tetris (16 classes)		
	Tetris	Maybe
Jackut (19 classes)		
	-	-
Solitaire (23 classes)		
	-	-

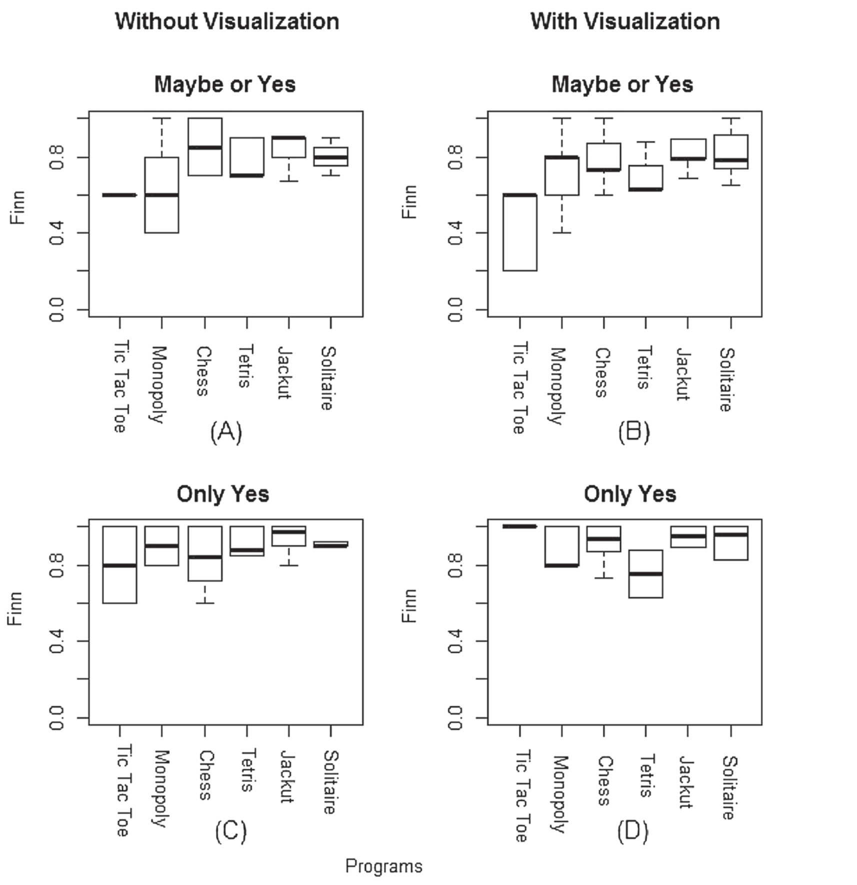


Figure 4 The distribution of Finn agreement values between participants and oracle: cases without visualization in (A) and (C); cases with visualization in (B) and (D).

second level, packages on the third level, classes on the fourth level, and methods and attributes on the fifth level. It is used to navigate in the original Eclipse's set up.

The second relevant action represents the activation of the Compilation Unit Editor, which is commonly used for code reading/writing. The actions numbered three to seven represent the activation of views of the visualization tool. For example, action three represents the activation of the Polimetric view, presented in Section 'Tools'. Actions four

Table 10 Actions performed with and without the visualization tool

No	Actions	Visualization	
		With	Without
1	activated_view_PackageExplorer	311	806
2	activated_editor_CompilationUniEditor	263	1240
3	activated_view_PolymetricView	135	
4	activated_view_DependencyView	117	
5	activated_view_TreeMapView	114	
6	activated_view_CouplingMatrixView	70	
7	activated_view_GridMatrixView	59	
8	Others: 10 with SourceMiner and 23 without SourceMiner	47	71
	Total	1116	2117

to seven represent the activation of the Dependency, Tree Map, Coupling Matrix and Grid Coupling views. The other actions were not shown because they did not occur very often.

It is possible to note the difference in the total. For the case with visualization, the total number of actions was 1116. Without visualization the total number was 2117. The views of the visualization tool were used frequently, when permitted: Polymetric (135), Dependency (117), Tree Map (114), Coupling Matrix (70) and Grid Coupling (59).

Strategies of god class detection

To evaluate the strategies we only observed cases with visualization because, in the case without visualization, the main action was related to activation of the Compilation Unit Editor, i.e. reading source code, and Eclipse does not log its use in detail.

As discussed in Section 'Data', we defined a sequence as all actions performed by the participant for a program.

In this experiment, there were three sequences of actions for each participant using the visualization tool, one for each program. There were 11 participants, but we deleted three of the sequences because we found problems in these sequences. The problems were caused by misuse of the Task Register plug-in. Therefore, 30 sequences were evaluated.

Our first analysis was related to individual sequences. We searched for common patterns. To do this, we used the LTL Checker of the ProM[b], a support tool for techniques of process mining (van der Aalst 2011). With the LTL Checker it is possible to check a property of the set of sequences expressed in terms of Linear Temporal Logic (LTL).

Consider the following three sequences, as a simple example. The sequences one and two have three actions, and the third sequence has two actions:

1. *activate_view_Polimetric, activate_view_TreeMap, open_CompilationUnitEditor*
2. *activate_view_Polimetric, open_CompilationUnitEditor, activate_view_TreeMap*
3. *activate_view_Polimetric, activate_view_TreeMap*

The LTL Checker allows checking, for example, that the action *activate_view_ TreeMap* always occurs some time after *activate_view_Polimetric*. It is also possible to check that the action *open_CompilationUnitEditor* occurs in two out three sequences. Or that the action *activate_view_TreeMap* occurs next to *activate_view_Polimetric* in two out three sequences. Table 11 shows the main results and interpretations for the sequences of the experiment.

We also investigated preferences grouped by participants. Some participants read more and used fewer views, whereas others did the opposite. To evaluate these aspects, we calculated the ratio between the number of classes investigated for each program and the use of views and readings. We counted the number of actions related to reading (activation of Compilation Unit Editor), activation of hierarchical views (Polymetric or Tree Map), and activation of coupling views (Dependency, GridCoupling or CouplingMatrix). For example, there were 12, two and four (18 in total) "activated_editor_CompilationUnitEditor" for the sequences of the F13 participant, for the programs Monopoly, Tetris and Tic Tac Toe, respectively. The total number of classes for these three programs is 31. We defined the ratio of using the CompilationUnitEditor, for F13 participant, as 18/31. He/she activated Coupling views 31 times for the three programs. In this case, the ratio is 31/31. Note that, if the participant activated the views more than the total number of classes of the

Table 11 Main results of the evaluation of sequences with LTL checker

Formula eventually_activity...	Action (activity) Activitation of ...	Number of cases (in 30)	Interpretation
A	Compilation Unit Editor	20	There were 20 cases where the participants read source code to identify god classes
A, B, C, D and E	Polimetric, TreeMap, Grid Coupling, Coupling Matrix, Dependency	11	Despite hierarchical views ad coupling views show the same attributes, there were 11 cases where the participants adopted all views of the visualization tool to identity god classes
A and B	Hierarchical views and Coupling views	27	There were 27 cases where the participants combine the use, at least, one of the hierarchical views and one of the coupling views to identify god classes
A, B and C	Hierarchical views and Coupling views and Compilation Unit Editor	17	There were 17 cases where the participants combined reading source code with one of the hierarchical (at least) and one of the coupling views (at least)

investigated programs, the ratio will be greater than 1. The complete results are presented in Figure 5.

From the figure, we identified six different profiles. We present them in Table 12. The first profile is composed of participants F14, F21 and F42. They used very little or no reading, and had a slight preference for coupling views. In profile 2, the two participants (F31 and F41) also did little reading in comparison to the usage of coupling views. In profile 3, participants F15 and F25 had preferences for coupling views, but they focused on reading unlike the previous group. For the other cases we found only one participant.

To investigate the "quality" of profiles we evaluated the agreement between each participant and the oracle. We plotted the Finn coefficient in Figure 6. From the graphs, it can be seen that no participants strongly agreed with the oracle. For example, participant F21 (Figure 6(A) and (C)) had the highest agreement for the program Chess and the worst agreement for Jackut. This was the general pattern.

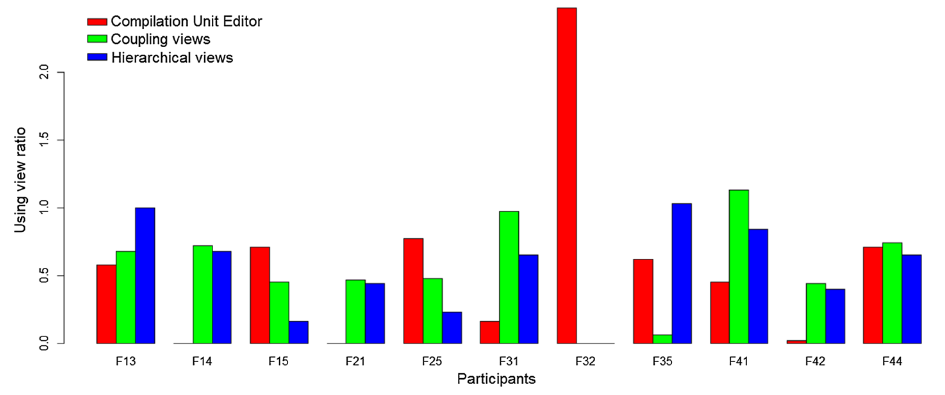


Figure 5 Ratio of using CompilationUnitEditor, hierarchical and coupling views.

Table 12 Profiles of participants related to the use of reading, hierarchical and coupling views

No	Profile	Participants
1	No reading and slight preference by coupling views	F14, F12 and F42
2	Few reading and strong preference by coupling views	F31 and F41
3	Focus on reading and preference by coupling views	F15 and F25
4	Preference by hierarchical views	F35
5	Slight preference by hierarchical views	F13
6	Similar using of all views and reading	F44
7	None using the visualization tool	F32

RQ4: What issues in code lead humans to identify a class as a god class?

To address this question we collected data related to drivers used by participants on the identification of god classes. The answer form provided a multiple choice list of nine drivers extracted from the work of Schumacher et al. (2010) (see Section ‘Forms’). It also allowed the participants to write down new drivers.

For cases where visualization was not used, the participants wrote down 19 short descriptions of new drivers. The coding process on these 19 new drivers’ descriptions was simple. For example, some participants wrote down “class has many lines of code”, others wrote down “class has big size”. In these cases, we defined “class has high LOC” as the driver. After this process, we narrowed the descriptions down to six actually new drivers. Table 13 shows all the drivers. The most common drivers were “class is highly complex” (48 times) and “method is highly complex” (31 times). An intermediate group included drivers like “class is special/framework” (12 times), “class represents a global function” (8 times) and “lack of comments” (7 times). The other drivers had low marks (≤ 5 times).

We also analyzed the distribution of drivers by participants. The drivers “class is highly complex” and “method is highly complex” were filled by all and almost all participants, respectively. The other 13 drivers were used by no more than four participants,

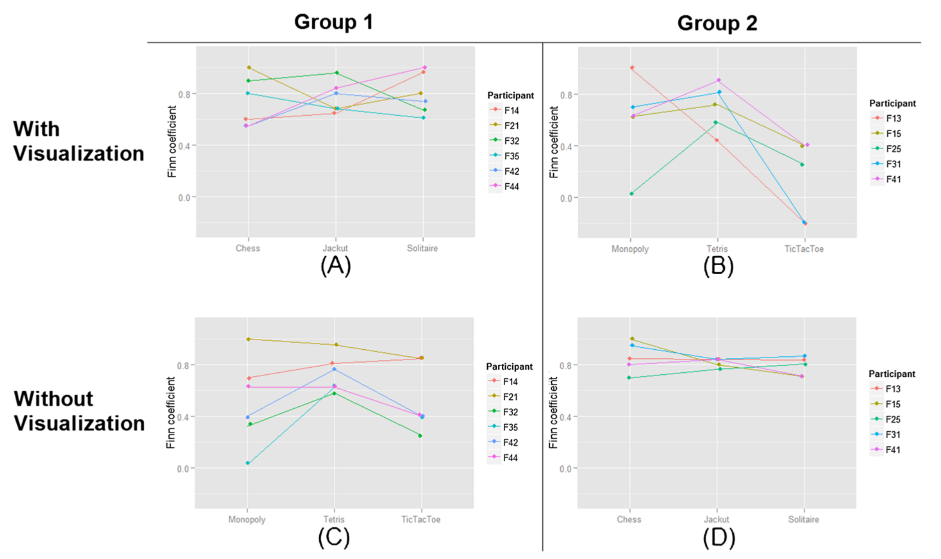


Figure 6 The Finn agreement coefficient among participants and the oracle: Group 1 in (A) and (C); and Group 2 in (B) and (D).

Table 13 Drivers by participants (Schumacher et al. and new drivers) without visualization

Participant	Based on Schumacher *et al.* coding									Coding on comments					
	Class				Method			Attribute	Method/class	Class				Method	
	Not used	Highly complex	Misplaced	Special/ framework	Wrong named	High complex	Misplaced	Not used	Lacks comments	High LOC	Many dependencies	Many methods	Global functions	Can be split	High LOC
F13	0	6	0	3	0	6	2	0	3	0	0	0	0	1	1
F14	0	3	0	0	1	1	1	0	0	1	0	0	0	0	0
F15	0	4	0	0	1	3	0	0	0	0	0	0	0	0	0
F21	0	4	0	4	0	0	0	1	0	0	0	0	0	0	0
F25	0	9	0	0	0	6	0	3	1	0	0	0	0	0	0
F31	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0
F32	0	4	0	0	0	0	0	0	0	0	0	0	7	0	0
F35	0	5	0	0	0	2	1	0	0	0	0	0	0	0	0
F41	0	5	0	2	0	7	0	1	0	3	0	2	0	0	2
F42	0	3	0	3	0	1	0	0	0	0	4	0	0	0	0
F44	0	4	1	0	0	4	0	0	3	0	0	0	0	0	0
Totais	0	48	1	12	2	31	4	5	7	4	4	2	8	1	3

even the drivers in the intermediate group. This indicates that drivers like "class is special/framework" were consistently adopted by some, but this was not a consensus among the participants.

For cases where visualization was used, the participants wrote down 27 short descriptions of "new drivers". Like before, these descriptions were reduced to a group of six actually new drivers. This group was almost the same as before. As shown in Table 14, it excludes/includes just one driver ("method can be split" is substituted by "problem with hierarchy"). The most common drivers were "class is highly complex" (46 times), "method is highly complex" (21 times) and "class has many dependencies" (20 times). The intermediate drivers were "class is special/framework" (11 times), "class/method lack comments" (8 times) and "class has high LOC" (7 times). The others had low marks (≤ 3 times).

Like before, "class is highly complex" and "method is highly complex" were chosen by all or almost all participants. "class has many dependencies" was also chosen by most (seven) participants. "class is special/framework" was chosen by five. The other 11 drivers were chosen by less than three participants, following the same pattern previously described.

Finally, we analyzed the differences by cases with and without visualization. The coding we performed produced practically the same new drivers and the distribution of the total marks was also quite similar. The most common drivers were the same: "class is highly complex" and "method is highly complex". The notable difference was the "class has many dependencies" driver. It was common in the visualization-based analysis and uncommon in the code-based analysis (without visualization).

Discussion

This section presents the discussion of the results of the experiment. Following the same logic of the previous section, we created one subsection to present the discussion for each question.

RQ1: How well do the participants agree on identifying god classes?

We addressed this question from two perspectives: i) number of candidate classes and ii) agreement test.

Number of candidate god classes

Comparison with Schumacher et al. (2010)'s work. Schumacher et al. (2010) found a very small number of god class candidates: only two in 52 inspected for one of the projects and three in 51 for the other project (3.8% and 5.8%, respectively). The numbers were much higher in our case: the average number of candidate classes were 34.8% and 35.8% for cases with and without visualization, respectively. One possibility is that the difference was related to the type of programs. Because the programs used in our experiment have a smaller number of classes, one god class candidate represents a high percentage.

On the other hand, it is possible to conjecture about the difference on the number and experience of the participants. In both studies, the number of participants is small: 11 in our case, and only four in the Schumacher et al. work. Our participants were undergraduates and Schumacher et al. ran the study with professionals. However, despite experience, three of their participants were unfamiliar with the concept of smell and god class in the experiment. The other participant had only heard about god classes before. Therefore, for both studies the participants had little knowledge about the concepts. We consider this

Table 14 Drivers by participants (Schumacher et al. and new drivers) with visualization

Subject	Based on Schumacher *et al.* coding									Coding on comments					
	Class				Method			Attribute	Method/class	Class					Method
	Not used	Highly complex	Misplaced	Special/ framework	Wrong named	High complex	Misplaced	Not used	Lacks comments	High LOC	Many dependencies	Many methods	Global functions	Hierarchy structure	High LOC
F13	0	4	0	1	0	4	0	0	2	3	4	1	0	0	1
F14	0	5	0	0	1	1	0	0	0	1	3	0	0	0	1
F15	0	3	0	0	0	1	0	0	0	0	0	0	0	0	0
F21	0	4	0	5	0	0	0	1	0	0	0	0	0	0	0
F25	0	8	0	0	0	3	0	1	2	0	0	0	0	0	0
F31	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0
F32	0	2	0	3	0	0	0	0	0	0	0	0	2	0	0
F35	0	6	0	0	0	3	2	0	0	0	0	0	0	0	0
F41	0	3	0	1	0	3	0	1	0	3	2	0	0	1	0
F42	0	6	0	1	0	3	0	0	0	0	5	0	0	0	0
F44	0	4	0	0	0	2	0	0	4	0	1	0	0	0	0
Totais	0	46	0	11	1	21	2	3	8	7	20	1	2	1	2

aspect empirical evidence that the knowledge related to reading or having heard about the concept of smell or god class is weaker than the experience related to work in a development environment. We usually believe in that, but we do not have much empirical evidence. To us, this makes the importance of expanding the discussion on the context of software engineering experiments evident. This problem has been rarely addressed in Experimental Software Engineering (Dybå et al. 2012; Höst et al. 2000, 2005).

Comparison between cases with and without visualization. There was no significant difference between the number of god class candidates with and without visualization (Figure 3). Considering the "yes" or "maybe" marks, values for cases with visualization are higher for the two programs, and lower for two other programs. Considering only "yes" marks, values for cases with visualization are higher only for one program and lower for two programs. The main finding is that the use of visualization does not impact the number of candidate god classes, i.e. people do not identify more or fewer god classes because of a better comprehension of the design of the program (obtained via the visualization tool). For us, this is evidence that people have their own conceptualization of what a god class is, and the visualization tool does not affect this.

This finding appears to run contrary to the one presented by Murphy et al. 2010. In this work, the first hypothesis is "Programmers identify more smells using the tool than not using the tool". However, their tool uses visual aids based on detection mechanisms. These mechanisms use metrics to visually highlight certain smells related to attributes. This contributes to harmonize (for better or worse) the conceptualization of what a smell is.

Agreement test

Comparison with Schumacher et al. (2010)'s work. We could not compare both works in terms of numbers because Schumacher et al. used Cohen's Kappa and we adopted the Finn coefficient. In Shcumacher's et al. work, the agreement for the one project was -2%. Based on Landis's and Koch's (1977) interpretation of Kappa, this indicates no agreement among the two participants. In the other project, the calculated Kappa was 48% (suggesting moderate agreement). Their conclusion was that there was not a high level of agreement. The work of Mäntylä (2005), in a similar study about agreement, found a similar result. For comparative purposes, we tested agreement with the Fleiss Kappa coefficient[c] and Landis's and Koch's interpretation. Our results showed only a slightly higher agreement than Schumacher's et al. work.

The values of the Finn coefficient were significantly higher as we can see in Tables 5 and 6. However, it is important to note that the Finn and Kappa tests measure agreement using both god and non-god classes. Consequently, we were cautious about finding high levels of agreement (almost perfect), because many classes are clearly not god classes. Participants were expected to agree on this. The same situation might have happened in the Scumacher's et al. work. We believe that their values were affected by the problem of the Kappa coefficient, which we discussed in the Subsection 'Finn agreement test'. Despite this, the results considering just the "yes" mark were convincing. Almost perfect agreement occurred for four of the six cases without visualization, and five of the six cases with visualization. The results were not as convincing when some level of doubt was allowed. For the "maybe" and "yes" marks, almost perfect agreement occurred in only one of the six cases without visualization, and two of the six with visualization. Based on these analyses, we consider our results aligned with previous studies: we did not find a high level of agreement.

The main finding here is that the participants have different conceptualizations of what makes a god class. This may happen in two dimensions. First, participants may have different classification thresholds in their evaluation of candidates. For example, participants may have different perceptions of how many roles a class can assume before it becomes a god class. Another possibility is that participants simply have different views of the god class concept. We consider this to be the weaker possibility because they had the same training and they had a similar level of knowledge before the experiment. Furthermore, the difference in the classification threshold was observed first hand when the oracle was being defined by more experienced professionals.

Comparison between cases with and without visualization. Another important dimension is the impact of visualization. We can not affirm that the use of visualization leads to "big" improvements. We can see in Tables 5 and 6 that there were four in 11 cases where the level of agreement was higher and there was one case where the level of agreement was lower with visualization. As a result, the agreement was slightly better for cases with visualization. We use the next research question to further investigate the impact of visualization.

RQ2: How well do humans and an oracle agree on identifying god classes?

We addressed this question from two perspectives: i) the oracle definition process; and ii) the agreement between the oracle and participants.

The oracle definition process

An important aspect in the agreement analysis between participants and the oracle is that we can not claim that the discrepancies between the participants and the oracle are errors, but we can claim that the oracle definition process was more rigorous. During the oracle meeting, both oracle researchers noted that the choice of a god class was associated with their personal perception about "how many roles the class needs to be considered god class" or "how much LOC the class needs to be considered a large class", or other subjective thresholds related to the characteristics of classes. This can be an explanation for the lack of agreement found among the participants of our study, as well as in (Mäntylä's 2006a and Schumacher's et al. 2010) studies.

The agreement between the oracle and participants

We analyzed this from two perspectives: level of certainty or doubts ("yes" or "maybe" marks) , and the use of visualization (Figure 4).

Comparison between cases with and without doubts. From Figure 4, we can see that when certainty – "yes" mark – is involved, the agreement between the participants and the oracle is generally high, while the variation in these agreements – see the box sizes – is usually small. These values are significantly reduced for the "yes or maybe" marks. The agreement average and variation between the oracle and the participants are much lower. If we consider the discussion on the high level values naturally produced by the Finn coefficient, this reinforces the thesis that agreement is low when doubt is involved. This also reinforces the evidence that participants have a personal conceptualization of what is a god class. As discussed previously, the likely causes for this is the different classification thresholds, or the different views of the god class concept. Like before, we consider the former more probable.

Comparison between cases with and without visualization. Considering the averages and the "yes or maybe" options in Figure 4(A) and (B), one can observe that there was only one case where the visualization improved agreement. Considering only the "yes" option (Figure 4(C) and (D)), there were three out of six cases where the visualization improved agreement. We conclude that the visualization does not affect the agreement values. This weakens the argument made in the discussion of RQ2, where we stated that visualization slightly improves the agreement among the participants. It might be the case that visualization slightly improves the precision among the participants, but it does not improve the accuracy of their detection against an oracle reference.

RQ3: Which strategies are used to identify god classes?

Relevant actions

First we addressed differences in the amount of reading for the cases with and without visualization. Comparing both cases, the reduction in the activation of the Compilation Unit Editor, or the reading of source code, was significant: from 1240 (without visualization) to 263 (with visualization). On the other hand, actions related to the activation of the views of the visualization tool emerged. We concluded that participants exchanged reading for the observation of views, i.e., they used visualization to search for attributes that indicate god classes. Based on the use of the views and tacit knowledge, we suggest that, to identify god classes, participants compared LOC, complexity or number of methods; observed coupling attributes and read code to understand the context of the class in the program.

Strategies of god class detection

Evaluating sequences. We analyzed each case presented in Section 'Strategies of god class detection'. The first case was related to reading. It was found in 20 of the 30 sequences. We believe that it was adopted by the participants to comprehend the role of the class in the program. Considering the concepts presented by (Fowler 1999 and Lanza and Marinescu 2005), some reading is necessary. In the definition of a Large Class, Fowler says that "*...it often shows up as too many instance variables*". In the God Class and Brain Class definitions, Lanza and Marinescu adopt metrics that involve a number of branches, deep nesting and a number of variables. In the experiment, it was not possible to see these characteristics with the visualization tool. However, despite the expected behavior, 10 sequences did not contain actions related to reading code. We analyze these aspects in the evaluation of participants hereafter.

In the other formula, we detected that in 11 sequences participants activated all five views of the visualization tool. As discussed in Section 'Tools', the two hierarchical views present similar attributes, like the other three coupling views. This indicates that in some cases participants worked more than necessary. Our conjecture is that they wanted to increase their level of certainty. It is important to note that, for most sequences (19), the participants did not use all the views, which we consider more efficient. Despite this, the need to confirm ideas is a behavior that must be considered in the analysis.

Another analysis that we performed was related to the combination of views and reading. The combination of hierarchical and coupling views occurred in 27 of the 30 sequences. This was the main strategy used to detect god classes.

Evaluating participants' strategies. We identified three participants who combined structural, coupling views, and used very little or no reading. It is profile 1 in Table 12. In profile 2, the two participants also used little reading in comparison with their preference for coupling views. From these two profiles, we suggest that the preference for coupling indicates that this attribute is the strongest, and that reading source code was not considered so relevant in god class detection. Another interesting profile is the number 3. In this case, the participants focused on reading and had a preference for coupling. In comparison with the other two profiles, this agrees with our previous discussion that participants exchanged reading for observation of specific attributes in the views of the visualization tool. In some cases, participants seem to be more confident reading than using the visualization tool. The other profiles indicate different preferences. For example, profile 4 shows a strong preference for structural views and profile 6 shows the similar use of all types of views and reading.

An interesting observation is related to the absence of a "better" strategy, or even a "better" participant, considering the agreement with the oracle (Figure 6). This reinforces our idea that god class detection is strongly affected by personal conceptualizations. If this was not true, a participant with a similar approach to the oracle would produce high levels of agreement with it for most programs. This was not what happened in our experiment.

RQ4: What issues in code lead humans to identify a class as a god class?

Some drivers were chosen by participants, independent of visualization. "Class is highly complex" (chosen by all participants) and "Method is highly complex" (chosen by nine out eleven participants), two of the strongest drivers, are examples of that. The only, and very interesting exception to the rule, is the "class has many dependencies" driver. Only one participant chose this driver for the case without visualization, and seven participants chose the driver for the case with visualization. While the visualization tool provides several views which showed signs of dependency, the current IDEs and the source code do not help in the identification of dependency. This case is evidence that the use of a visualization tool can indeed help, because some detection drivers are poorly supported by the current state of the practice. This, however, does not solve the conceptualization problem.

It is also worthwhile to observe that, although participants did not always use the same drivers, they were very consistent in using their drivers of choice. For "Method is highly complex", for example, the two participants who did not choose the driver are the same (F21 and F32) with and without visualization. Other cases where the same participants chose the same drivers were: "attribute is not used", "methods or class lack comments". For other cases, the difference between the choices of the participants is very small; at most one participant. That suggests that in some cases the choice of drivers is a personal issue. This idea reinforces our conjecture about the importance of the community discussing the problem of conceptualization on smells in depth.

Summary of findings and insights

In this work, we addressed the problem of conceptualization in the god class detection, from different perspectives. In this section, we gather the findings and present our conclusions. We also present peripheral, but not less important findings:

- Agreement
  - *Related to the conceptualization problem*: The low agreement rate showed us that the problem exists. However, we consider the question related to visualization more interesting, because it showed us where the problem is, or at least where it is not. The problem of conceptualization is that it is not related to the comprehension of design, as we had expected. We argue this because visualization did not increase agreement. The finding is that the problem is related to personal understanding of the smell concepts or of personal thresholds adopted.
  - *Peripheral findings*: We found evidence that experience affects the degree of agreement. Our evidence was: i) the comparison of the number of candidate god classes in our and Shumacher's et al. work; and ii) low agreement among the participants and our oracle (made up of more experienced researchers). Another interesting peripheral finding is related to the using of the Kappa coefficient for agreement analysis. We noted that, for the Kappa coefficient, in some cases, the agreement value is high and the coefficient is low. Based on our research, we suggest the adoption of the Finn coefficient (see discussion in the Section 'Finn agreement test').
- Strategies
  - *Related to conceptualization problem*: The absence of a "better" strategy reinforced our finding that each participant has his/her own idea about how and what he/she has to do to identify god classes.
  - *Peripheral findings*: We proposed an approach to identify strategies: the identification of used views from the logs. We grouped participants according to their strategies. The main profile identified focused on coupling attributes, and little reading.
- Decision drivers
  - *Related to the conceptualization problem*: As mentioned under "strategies of god class detection", some decision drivers are also personal choices. In a consistent way, participants had different preferences for characteristics they used to identify god classes. This also reinforced our previous findings.
  - *Peripheral findings*: The main decision driver was "class is highly complex".

The results discussed here strengthen the idea that the problem in god class detection is more related to conceptualization than to making the comprehension of the code design easier by facilitating the (visual) observation of certain attributes of the code. Note that this type of problem is not solved by using other observation approaches, such as proposing a new metric. The key problem is to identify concrete "good" examples of smells, providing standard definitions of them, teaching people about their conceptualization, and only then providing the tools and methods (using metrics, thresholds or visualizations) to aid their identification.

Threats to validity

Our analysis of threats was based on Wohlin et al. (2012).

External validity. Our first threat fits in the "interaction of selection and treatment" subcategory and is related to the fact that the participants in our experiment were undergraduates and had little experience in a real software development environment. Moreover, the experiment was run with 11 participants. Although the aspects related to the participants could be considered a problem for generalization, we have strong evidence that the existence of the problem discussed in this work is intrinsic to smell detection and also happens with experienced developers. The evidence is: i) smell concepts are presented subjectively Fowler (1999) or are dependent on thresholds Lanza et al. (2005); ii) findings show low agreement in other works with more experienced participants Schumacher et al. (2010); and iii) the initial lack of agreement among the experienced researchers who created the oracle in this work. We believe that experience affects our results only in terms of intensity, but the problem of conceptualization exists for all cases. We are planning to replicate this experiment with more experienced participants to evaluate the impact of experience.

Other threats to external validity fit in the "interaction of setting and treatment" subcategory. In this case, the threat is the type of program. We adopted simple programs. Another point is the domain: they are familiar software, and in most of the cases, games. Software of a different domain might present different characteristics. However, we argue that the same, previous reasoning is valid here, weakening the threat: the phenomenon can be studied in any type of software. In fact, because we used simple and familiar software, the problem should be minimized, which was not the case. Moreover, we did not address the difficulty of god class identification, but how conceptualization affects smell detection.

Internal validity. The study has threats in two subcategories related to internal validity. The first one is "ambiguity about direction of causal influence". We highlight the fact that the training about god classes reflects the view of the experimenter. The view of the experimenter could affect the participants' conceptualization and their ideas about what they had to look for to identify a god class. To minimize this effect, we limited the time of the training in the god class concept and adopted the support questions from Schumacher's experiment to guide participants in god class detection during our experiment. These actions also mitigate the same threat in the opposite way: participants could not have a general idea of what to search for. We consider that the training and the support questions steered the participants to search for classes that represent the god class concept adopted.

Another threat is the training in the visualization tool. In our feedback form, participants indicated that the quality of training was good, in general. However, we can not confirm that it was sufficient to prepare participants in the use of visualization. Another subcategory of the internal validity is "maturation". Participants could be affected because they do the same task over six programs, so they may learn as they go and work faster. On the other hand, they could be affected negatively because of boredom. We consider maturation a weak threat because the experiment was performed in 1.5 hours, on average. We consider this a reasonable period of time to do a task in a balanced way.

Conclusion validity. In the "reliability of measures" category, we should report that the logged information represents the actions of the participants only indirectly. They actions of the Eclipse IDE. For example, if a developer changes the perspective in the Eclipse, some views are activated by the tool and these actions are registered in the log. To mitigate this aspect, we investigated the logging to evaluate actions in detail and eliminated lines clearly related to Eclipse actions. Moreover, these registers occurred for all participants and did not affect the general conclusion. In the "reliability of treatment implementation", we have to consider that a participant who could have used visualization may have completely disregarded the views of the visualization tool. However, we checked the UDC logging and only one participant did not use the visualization resources. Because this occurred in one case, the results were not affected. Lastly, due to the number of data points, some of our findings were based on the analysis of graphs and tables, and inferential testing was done in a few cases.

Context and related works

As the use of the concept of smells has become widespread, empirical studies have been presented to help understand their effects. As discussed in Section 'Background', we identified three types of empirical work in the area. In this section we present them. We provide more details of papers related to human aspects, which are closer to our work. However, we present correlation studies because they provide evidence that the use of code smells as an indicator of a problem in the design has been inconsistent, which we believe might be caused by the problem of conceptualization. We also present some works related to the use of support tools, because our work is focused on the use of a visualization tool.

Correlation studies

This type of work usually focuses on analyzing software evolution and tries to link smells with some characteristic of code. Normally, they investigate data in software repositories. Olbrich et al. (2010), for example, investigated the influence of two smells ("God Class" and "Brain Class") on the frequency of defects. In order to do this, they analyzed historical data from three open-source software systems. They found that, in specific cases, the presence of these smells might be beneficial to a software system. In another study, Olbrich et al. (2009) investigated the evolution of two other smells ("God Class" and "Shotgun Surgery") for these same systems. Li and Shatnawi (2007) studied the relationship between smells and error probability. They investigated three error-severity levels in an industrial-strength open source system. Their findings indicate that some bad smells are positively associated with the probability of errors.

Sjoberg et al. (2013) presented a controlled study where six professionals were hired to maintain four systems for 14 days. Their aim was to quantify the relationship between code smells and maintenance effort. One of the main findings was that "the ... smells appear to be superfluous for explaining maintenance effort". Abbes et al. (2011) adopted the concept of anti-pattern, which, like the concept of code smell, presents "poor" solutions to recurring design problems. They performed an empirical study to investigate whether the occurrence of anti-patterns does indeed affect the understandability of systems by developers during comprehension and maintenance tasks. They found that the occurrence of one anti-pattern does not significantly decrease developers' performance while the combination of two anti-patterns significantly impedes developers.

Smells and support tools

Some works evaluated smell detection using automatic detection tools (Moha et al. 2010; Mäntylä and Lassenius 2006a; Schumacher et al. 2010). Other works addressed smell detection with visualization tools (Carneiro et al. 2010; Murphy-Hill and Black 2010; Parnin et al. 2008; Simon et al. 2001). From the former, we discuss the first tool, because it was developed to mitigate subjectiveness. The authors propose DECOR. It is a method/tool "that embodies and defines all the steps necessary for the specification and detection of code and design smells". Despite the tool mitigating some subjectiveness, the focus of the work was on the method, not on the aspect that affects (or does not affect) the conceptualization.

Here, we also discuss here the two most recent tools related smell detection and software visualization. In (Murphy-Hill and Black 2010), Murphy-Hill and Black presented a visualization implemented as an Eclipse plug-in. The tool is composed of sectors in a semicircle on the right-hand side of the editor pane, called petals: each petal corresponds to a smell. They performed a controlled experiment with 12 participants (6 programmers and 6 students) to evaluate the tool. Their main findings were: i) programmers identify more smells using the tool than not using the tool ii) smells are subjective and iii) the tool helps in deciding. Carneiro et al. (2010) presented the SourceMiner tool; a multi-perspective environment Eclipse based plug-in. SourceMiner has visualizations that address inheritance and coupling characteristics of a program. The visualizations also portray the previously mapped concerns of the analyzed software. The authors performed an exploratory study with five developers, using a concern mapping multi-perspective approach to identify code smells. Two main findings were presented. First, the concern visualizations provided useful support to identify God Class and Divergent Change smells. Second, strategies for smell detection supported by the multiple concern views were revealed.

Smells and human aspects

Mäntylä (2005) presents results of two experiments addressing agreement in smell detection and factors to explain it. A small application in Java with nine classes and 1000 LOC was created and used in both experiments. In the first experiment, there were three questions about "Long Method", "Long parameter List" and "Feature Envy" smells, and one question asked if the method should be refactored to remove the detected smells. In the second experiment, participants were only asked if some specific methods should be refactored. He found high rates of agreement for simple smells: long method and long parameter list. He found weaker agreement levels, however, concerning the feature envy smell and the decisions regarding refactoring. Mäntylä then tried to identify factors that influenced agreement in smell detection. He investigated both the influence of software metrics and demographic data as factors for smell detection agreement. His findings point to the influence of metrics.

Mäntylä and Lassenius (2006a) investigated why and when people think a code needs refactoring. They analyzed one of the experiments presented in (Mäntylä 2005) to investigate what drivers define the refactoring decisions. They applied a questionnaire to understand refactoring decisions. A taxonomy was defined after a qualitative analysis (coding process) of the textual answers. The authors also compared the results with an automatic detection tool. The most important driver was the size of a method. One of their important findings was that there was a conflict of opinions between the participants. The conflict was related to the assessed internal quality of the methods and the need to refactor them. Regarding the automatic detection, they found that some drivers are difficult or impossible to detect automatically, and some smells are better detected by experienced participants than by automatic means.

Schumacher et al. (2010) build on and extend Mäntylä and Lassenius's (2006a) work. They investigated the way professional software developers detect god class smells, and then compared these results to automatic classification. The study was done in a professional environment, with two real projects and two participants in each project. The research questions focused on "Evaluation of Human Performance" and "Evaluation of Automatic Classifiers". Participants were introduced to the god class smell in a short presentation and were asked to detect them in specific code pieces. During this task, they received a list of questions to help with the identification of god classes (the support questions adopted by us in this work), questions such as: "Does the class have more than one responsibility?". A process was designed to ensure that all participants performed the inspection of classes in a similar fashion. To evaluate the participant performance in the task, they used a "think-aloud" protocol (recorded as audio) and data collection forms. Coding was carried out to identify drivers and answers from the data collection form were used to evaluate time and agreement. Their main findings were: (1) there was low agreement among participants and (2) "misplaced method" was the strongest driver for god class detection. Related to the evaluation of automatic detection, their main findings were: (1) an automated metric-based pre-selection decreases the effort needed for manual code inspections and (2) automatic detection followed by manual review increases the overall confidence.

A study with aims similar to Schumacher's was presented by (Mäntylä et al. in 2004 and Mäntylä and Lassenius 2006b). Through a survey, they asked participants about 23 smells and used a scale from 1 (lack) to 7 (large presence) to evaluate the presence of smells in a piece of code. They received 12 completed questionnaires from 18 sent, all being sent to developers in a small software company. In one of the findings the authors declare: "*the use of smells for code evaluation purposes is hard due to conflicting perceptions of different evaluators*".

It is important to note that these studies focused on specific human aspects. Basically, they investigated agreement and decision drivers adopted by participants. We built on the discussions considering two important aspects: the strategies adopted and the impact of visualization on each investigated aspect. Moreover, we proposed a discussion about how each of these aspects reflects the problem of conceptualization. We consider this our main contribution.

Conclusions

The purpose of this work was to find empirical evidence to evaluate the impact of personal conceptualization in god class detection. We were interested in understanding how differences in the perception of the concept affect identification. To do this, we performed a controlled experiment that extended another study focusing on investigating how developers detect god classes. Our experiment deepened and detailed some research questions previously presented and added new research questions. We addressed agreement among participants, and also among participants and an oracle, decision drivers, the impact of using a visualization tool, and strategies adopted by participants in god class detection. Our analysis considered how these elements are more related to personal choice than to the conceptual aspects in god class detection.

Our main finding is that the problem of god class detection is mainly related to conceptualization, i.e., agreeing on what a god class is, and which thresholds should be adopted. We believe that this type of problem is not solved by using other observation approaches, such as proposing a new metric or a new visual resource. We also believe this issue is transversal to other code smells. The smell detection problem would be better addressed if the community identifies concrete "good" examples of smells, providing standard definitions of them, teaching people about their conceptualization, and only then providing tools and methods (using metrics, thresholds or visualizations) to aid in their identification. Another important finding was that our work produced low agreement rates in code smell detection among the experiment participants, which is in accordance with other works.

To address the limitations of this study and to further develop it in this area, we are planning to replicate the experiment with more experienced participants to evaluate the impact of experience on the process. Other aspects that we may replicate as well is the evaluation of other software and other smells. To support replication we provide the experimental package[d]. The package contains forms, data and software.

Endnotes

[a]Eclipse IDE - http://www.eclipse.org/downloads/; Usage Data Collector (UDC) plug-in - http://www.eclipse.org/epp/usagedata/; Task Register - private; SourceMiner - visual support http://www.sourceminer.org/

[b]Web address of ProM tool: www.processmining.org

[c]Fleiss Kappa is a Cohen's Kappa variation that permits test with more than two raters

[d]Experimental package: http://wiki.dcc.ufba.br/LES/FindingGdoClassExperiment2012

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
JAMS and MGM planned, performed and analyzed the experiment and drafted the manuscript. CPS supported some specific analysis. RLN also supported some analysis and drafted the manuscript. All authors read and approved the final manuscript.

Acknowledgements
We would like to thanks: Claudio Sant'Anna for allowing execution of the experiment in the Software Quality course; Bruno Carneiro for valuable participation in definition of oracle; and to participants for their availability and effort.

Author details
[1]Department of Technology, State University of Feira de Santana, Transnordestina avenue S/N - Feira de Santana - Bahia, Feira de Santana, Brazil. [2]Mathematics Institute, Federal University of Bahia, Ademar de Barros Avenue, S/N, Salvador - Bahia, Salvador, Brazil. [3]Fraunhofer Project Center for Software & Systems Eng., Ademar de Barros Avenue, S/N, Salvador - Bahia, Salvador, Brazil. [4]Information Technology Department, Federal Institute of Bahia, Araujo Pinho Avenue, 39, Salvador - Bahia, Salvador, Brazil.

Received: 19 December 2013 Accepted: 31 August 2014
Published online: 18 September 2014

References

Abbes M, Khomh F, Guéhéneuc Y-G, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: Proc. of 15th European Conference on Software Maintenance and Reengineering (CSMR). IEEE, Oldenburg, Germany, pp 181–190

Carneiro GF, Mendonça MG (2013) Sourceminer: A multi-perspective software visualization environment. In: Proceedings of 15th International Conference on Interprise Information Systems. ICEIS. SciTePress, Angers, France

Carneiro G, Silva M, Maia L, Figueiredo E, Sant'Anna C, Garcia A, Mendonça M (2010) Identifying code smells with multiple concern views. In: Proc. of 1th Brazilian Conference on Software: Theory and Practice, CBSOFT. IEEE, Salvador, Bahia, Brazil

Dybå T, Sjøberg DIK, Cruzes DS (2012) What works for whom, where, when, and why?: On the role of context in empirical software engineering. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '12. ACM, New York, NY, USA, pp 19–28

Feinstein AR, Cicchetti DV (1990) High agreement but low kappa: I. the problems of two paradoxes. J Clin Epidemiol 43(6):543–549

Finn RH (1970) A note on estimating the reliability of categorical data. Educ Psychol Meas 30:71–76

Fleiss JL (1971) Measuring nominal scale agreement among many raters. Psychol Bull 76(5):378–382

Fowler M (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Fontana FA, Mariani E, Morniroli A, Sormani R, Tonello A (2011) An experience report on using code smells detection tools. In: Proc. of 4th Software Testing, Verification and Validation Workshops, ICSTW. IEEE, Berlin, Germany

Gwet K (2002) Kappa statistic is not satisfactory for assessing the extent of agreement between raters. Stat Methods Inter-rater Reliability Assess 1:1–5

Höst M, Regnell B, Wohlin C (2000) Using students as subjects— a comparative study of students and professionals in lead-time impact assessment. Empirical Softw Eng 5(3):201–214

Höst M, Wohlin C, Thelin T (2005) Experimental context classification: incentives and experience of subjects. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference On. IEEE, St Louis, Missouri, USA, pp 470–478

Johnson B, Shneiderman B (1991) Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In: Visualization, 1991. Visualization '91, Proceedings., IEEE Conference On. IEEE, San Diego, CA, USA, pp 284–291

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33(1):159–174

Lanza M, Ducasse S (2003) Polymetric views - a lightweight visual approach to reverse engineering. Softw Eng IEEE Trans 29(9):782–795

Lanza M, Marinescu R, Ducasse S (2005) Object-Oriented Metrics in Practice. Springer, Secaucus, NJ, USA

Li W, Shatnawi R (2007) An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J Syst Softw 80(7):1120–1128

Meyer B (1988) Object-Oriented Software Construction, 1st edn. Prentice-Hall, Inc., Upper Saddle River, NJ, USA

Moha N, Gueheneuc Y-G, Duchien L, Le Meur A-F (2010) Decor: A method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36

Murphy-Hill E, Black AP (2010) An interactive ambient visualization for code smells. In: Proc. of the 5th ACM Symposium on Software Visualization, SOFTVIS. ACM, Salt Lake City, Utah, USA

Mäntylä M (2005) An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In: Proc. of the 4th International Syimposium on Empirical Software Engineering, ISESE. IEEE, Noosa Heads, Australia

Mäntylä MV, Lassenius C (2006a) Drivers for software refactoring decisions. In: Proceedings of the International Symposium on Empirical Software Engineering, ISESE. ACM, Rio de Janeiro, Brazil

Mäntylä M, Lassenius C (2006b) Subjective evaluation of software evolvability using code smells: An empirical study. Empirical Softw Eng 11(3):395–431

Mäntylä M, Vanhanen J, Lassenius C (2004) Bad smells - humans as code critics. In: 20th IEEE International Conference on Software MaintenanceICSM 2004, ICSM. IEEE, Chicago Illinois, USA

Olbrich S, Cruzes DS, Basili V, Zazworka N (2009) The evolution and impact of code smells: a case study of two open source systems. In: Proc. of the 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM. IEEE, Lake Buena Vista, Florida, USA

Olbrich SM, Cruzes DS, Sjoberg DIK (2010) Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In: Proc. of the IEEE International Conference on Software Maintenance, ICSM. IEEE, Timisoara, Romania

Padilha J, Figueiredo E, Sant'Anna C, Garcia A (2013) Detecting god methods with concern metrics: An exploratory study. In: Proceedings of the 7th Latin-American Workshop on Aspect-Oriented Software Development(LA-WASP), Co-allocated with CBSoft. IEEE, Brasília, Brazil

Parnin C, Görg C, Nnadi O (2008) A catalogue of lightweight visualizations to support code smell inspection. In: Proc. of the 4th Software Visualization, SOFTVIS. ACM, Herrsching am Ammersee, Germany

Powers DMW (2012) The Problem with Kappa. In: Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics, EACL '12, Stroudsburg, PA, USA, pp 345–355

Rapu D, Ducasse S, Girba T, Marinescu R (2004) Using history information to improve design flaws detection. In: Proc. of 8th European Conference on Software Maintenance and Reengineering, CSMR. IEEE, Tampere, Finland

Riel AJ (1996) Object-Oriented Design Heuristics, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Santos JA, Mendonça M, Silva C (2013) An exploratory study to investigate the impact of conceptualization in god class detection. In: Proc of 17th International Conference on Evaluation and Assessment in Software Engineering, EASE. ACM, Porto de Galinhas, Brazil

Schumacher J, Zazworka N, Shull F, Seaman C, Shaw M (2010) Building empirical support for automated code smell detection. In: Proc. of the International Symposium on Empirical Software Engineering and Measurement, ESEM. ACM, Bolzano-Bozen, Italy
Simon F, Steinbruckner F, Lewerentz C (2001) Metrics based refactoring. In: Proc. of 5th European Conference on Software Maintenance and Reengineering, CSMR. IEEE, Lisbon, Portugal
Sjøberg DIK, Yamashita A, Anda BCD, Mockus A, Dyba T (2013) Quantifying the effect of code smells on maintenance effort. IEEE Trans Softw Eng 39(8):1144–1156
van der Aalst WMP (2011) Process mining: discovery, conformance and enhancement of business processes. 1st edn., p. 352. Springer, Berlin
Van Emden E, Moonen L (2002) Java quality assurance by detecting code smells. In: Proc. of the 9th Working Conference on Reverse Engineering, WCRE. IEEE, Washington, DC, USA
Vinson NG, Singer Ja (2008) A practical guide to ethical research involving humans. In: Shull F, Singer J, Søberg DIK (eds) Guide to Advanced Empirical Software Engineering. Springer, London, pp 229–256
Whitehurst GJ (1984) Interrater agreement for journal manuscript review. Am Psychol 39(1):22–28
Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in Software Engineering. p 250
Zhang M, Hall T, Baddoo N (2011) Code bad smells: A review of current knowledge. J Softw Maint Evol 23(3):179–202

doi:10.1186/s40411-014-0011-9
Cite this article as: Santos *et al.*: The problem of conceptualization in god class detection: agreement, strategies and decision drivers. *Journal of Software Engineering Research and Development* 2014 **2**:11.

Appendix

C

PAPER PRESENTING ANALYSIS OF DECISION DRIVERS

This appendix shows the paper “Exploring decision drivers on god class detection in three controlled experiments”. The paper was published in the 30th ACM/SIGAPP Symposium On Applied Computing (ACM-SAC 2015).

Exploring decision drivers on god class detection in three controlled experiments

José Amancio M. Santos
Technology Department
State University of Feira de Santana
Bahia, Brazil
zeamancio@ecomp.uefs.br

Manoel G. de Mendonça
Fraunhofer Project Center for Software &
Systems Eng. at Federal University of Bahia
Bahia, Brazil
mgmendonca@dcc.ufba.br

ABSTRACT

Context: Code smells define potential problems in design of software. However, some empirical studies on the topic have shown findings in opposite direction. The misunderstanding is mainly caused by lack of works focusing on human role on code smell detection. ***Objective***: Our aim is to build empirical support to exploration of the human role on code smell detection. Specifically, we investigated what issues in code make a human identify a class as a code smell. We called these issues decision drivers. ***Method***: We performed a controlled experiment and replicated it twice. We asked participants to detect god class (one of the most known smell) on different software, indicating what decision drivers they adopted. ***Results***: The stronger drivers were "class is high complex" and "method is misplaced". We also found the agreement on drivers' choice is low. Another finding is: some important drivers are dependent of alternative support. In our case, "dependency" was an important driver only when visual resources were permitted. ***Conclusion***: This study contributes with the comprehension of the human role on smell detection through the exploration of decision drivers. This perception contributes to characterize what we called the "code smell conceptualization problem".

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design method*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Empirical study; Human factors; Decision drivers

Keywords

Code smell, God class, Controlled experiment

1. INTRODUCTION

Technical aspects of quality in design of object-oriented systems have been addressed historically. Terms like "design flaws" [18], "code smell" [7] and "disharmony" [10], are widely used to define potential design problems. In this paper, we adopt the term *code smell*, or simply *smell*, to refer to such design problems.

Despite code smell being a concept widely accepted as indicative of bad design, some empirical studies have presented findings in opposite directions. Sjøberg et al. [23], for example, investigated the relationship between smells and maintenance effort. They noted that none of the investigated smells were significantly associated with increased maintenance effort. Macia et al. [12] investigated the relationship between code smells and problems that occur with an evolving system's architecture. They noted that many of the detected smells were not related to architectural problems. Zhang et al. [27] performed a systematic mapping on the subject and declared: "...we do not know whether using Code Bad Smells to target code improvement is effective". In general, the authors agree that more empirical studies are necessary for better understanding the smell effect [22, 23, 27].

The evaluation of the human aspects is a complex task because of the extensive number of context variables affecting the perception of code smell. In this work, our general aim is to build more empirical studies evaluating human aspects and code smells. Specifically, we have focus on decision drivers adopted on god class detection, one of the most known code smell. By decision drivers we mean what issues in code make a developer to identify a class as a god class. Empirical studies addressing drivers support the comprehension of inconsistencies related to practical adoption of smell on software development. This papers is proposed as part of the knowledge on the subject. Evidence of the relevance of the topic are other studies with similar aim [13, 14, 22].

To explore the topic, we performed three controlled experiments. Besides drivers identification, we evaluated the level of agreement on drivers' choice. We found the agreement is low. The main drivers were related to complexity, misplaced methods and dependencies among classes. We also noted the necessity of technical support in the evaluation of some important aspects of code: in our experiments, we adopted software visualization, which supported the observation of dependencies and size relation among classes. Some drivers were strongly used only when visualization was permitted.

The structure of this paper is as follows. Section 2 addresses some concepts and summarizes prior empirical studies on the subject. Section 3 presents the planning and ex-

SAC'15 April 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2695682

ecution of the experiments. Section 4 and 5 present the results and a discussion about them. Section 6 discusses threats to validity of the studies. Lastly, Section 7 presents our conclusions and proposes future works.

2. CONTEXT AND RELATED WORKS

God class is a central concept in our study. The term was coined by Riel [18] to refer classes tending to centralize the intelligence of the system. Since then, god class has been addressed in different empirical studies [1, 11, 15, 16]. Lanza and Marinescu [10] presented an heuristic for god class identification based on metrics. They also presented the code smell brain class, in a similar way: complex classes tending to accumulate an excessive amount of intelligence. The main difference is that a god class accesses directly many attributes from other classes. Another smell with similar characteristics, presented by Folwer [7] subjectively, is large class: a class trying to do too much. These three smells have similar idea.

In this paper, we did not define formally these concepts, because we explored the conceptualization about smell, not a metric based evaluation. To do this, we presented to participants a set of support questions to guide them. The questions were extracted from [22] and concentrate the general idea of god class, brain class and large class. Some examples are: "Does the class have more than one responsibility ?" and "Does the class have functionality that would fit better into other classes ?". We will discuss the experimental setup latter on. From now, we will adopt the term god class indistinctly.

2.1 Empirical studies on decision drivers

In this section, we briefly present some works addressing human decision drivers on smell detection.

Mäntylä and Lassenius[13] investigated why and when people think a code needs refactoring. They investigated what drivers define the refactoring decisions using a questionnaire. A taxonomy was defined after coding. The most important driver was size of a method. One of their findings was that there was a conflict of opinions between the participants, with respect both to the assessed internal quality of the methods and the need to refactor them.

Schumacher et al. [22] built on and extended Mäntylä and Lassenius's [13] work. They investigated the way developers detect god class, then compared these results to automatic classification. Schumacher et al. study was done in an in-vivo setting. During the experiment, participants received the support question guide (same we adopted) to help with the identification of god classes. Schumacher et al. used a "think-aloud" protocol (recorded as audio) and data collection forms. Their main findings were: (1) there was low agreement between the participants; and, (2) "misplaced method" was the strongest driver.

Yamashita and Consuell [26] investigated the potential of code smells to reflect system-level indicators of maintainability, in an in-vivo setting. Despite the focus on correlation between smell and indicators of maintainability, the authors conducted open interviews to compare result of both expert judgment and automatic detection approaches. From the interviews, it is possible to extract some potential drivers, such as "the business logic into one large class, considered very fault-prone"; or "the arbitrary use of variables and the duplicated code were deemed as the main reasons for the introduction of faults in this class". They found that some smells can be used for assessments of maintainability.

3. EXPERIMENTAL SETUP

In this section we present settings of a family of three controlled experiments: we named it *Finding God Class (FinG)* experiments. FinG aimed to address a set of context-aspects on god class detection, such as agreement, effort, strategies, and others. We had already partial results in [19, 20, 21]. In this paper, we focused only in the investigation of drivers adopted by the participants. We will call FinG 1 the original experiment, and FinG 2 the second experiment, which is a replication of FinG 1. In the same way, FinG 3 is a replication of FinG 2. There were two main differences in the experimental setup: i) the experimental unit (FinG 1 - undergraduate students; FinG 2/FinG 3 - graduate students and professionals); ii) application type (FinG 1/FinG 2 - six simple software; FinG 3 - four medium size software).

3.1 The Experimental Planning

3.1.1 Experimental Unit

FinG 1 involved 11 undergraduate Computer Science students from the Federal University of Bahia (UFBA), in Brazil. All students were enrolled in the Software Quality course offered in the first semester in 2012. The participation in the experiment was voluntary. FinG 2 and FinG 3 involved 27 and 24 graduate Computer Science students, respectively. All them from the UFBA. All students were enrolled in the Experimental Software Engineering course offered in the second semester in 2012 and first semester of 2013. Due to the special circumstances, almost all students of the course were (or had been) professionals. In FinG 2 and FinG 3, the participants received grades for their participation.

3.1.2 Tools

We adopted two main software tools in the experiment[1]: the Eclipse Indigo IDE and SourceMiner (SM), an Eclipse plug-in that provides visual resources to enhance software comprehension activities [2, 3].

We describe the SM because it affects the drivers on god class detection. The tool has visual resources making easier the design comprehension of the programs. It has five visual metaphors, divided in two groups. The first group is formed by three coupling views. These views show different types of coupling, as direct access to attributes or method calling, for instance. Moreover, they show the direction of the coupling. The coupling views are based on radial graphs, matrix of relationships, and tabular view (Figures 1 A, B and C).

The second group is formed by two hierarchical views. These views associate LOC, complexity and number of methods of classes to area, width and length of rectangles. The treemap view (Figure 1 D) shows the hierarchy of package-class-method. The more internal rectangles represent methods. The area and color are used to highlight the attributes of LOC. The red boards represent classes and the yellow boards represent packages. The Polimetric view (Figure 1 E) shows the hierarchy between classes and interfaces. The wider rectangles represent classes with more number

[1]Eclipse IDE - http://www.eclipse.org/downloads/; SourceMiner - http://www.sourceminer.org/

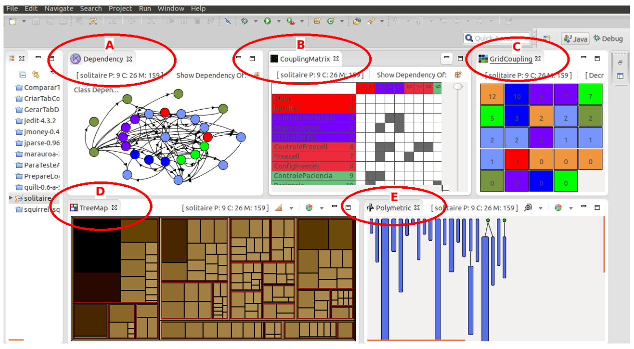


Figure 1: Views of the SourceMiner (SM)

of methods. Rectangles of greater length represent classes with larger LOC.

3.2 Forms and docs

Five forms were used. During training the participants filled in a Consent and a Characterization form. The participants also received a SourceMiner exercise guide. During the experiments themselves, they filled in an Answer form. The participants had to indicate god class(es) candidate(s). They also had to indicate the decision drivers adopted in their evaluation. The FinG 1/FinG 2 Answer form listed nine drivers as predefined options, but it was also possible to write down a new one. The drivers listed in the form were the ones identified in [22]. Some examples are "method is misplaced" and "class is highly complex". In the FinG 3 Answer form there was not any option of drivers available. The participants had to write down their own decision drivers.

Participants filled in a Feedback form at the end of all experiments.

Besides these forms, we used two other documents, during all experiments themselves. One of them presented an overview of the software used in the experiments. The other document was the Support Question guide. It was used to steer the participants in the search for god classes, by presenting some related questions, such as *"Does the class have more than one responsibility"*, or *"Does the class have functionality that would fit better into other classes?"*. The questions were extracted from [22].

3.2.1 *Software Artifacts*

FinG 1 and FinG 2. Six software were used. All of them implement familiar applications or games in Java. Chess, Tic Tac Toe, Monopoly and Tetris implement known games. Solitaire-Freecell (Solitaire) is a framework for card games. Jackut implements a simple social network application, in the line of Facebook or Orkut. Table 1 characterizes the used programs in terms of the number of packages, number of classes and LOC. As we can see, all them are simple software.

Table 1: Software used in FinG 1 and FinG 2

Software	Chess	Jackut	Tic	Monopoly	Solitaire	Tetris
Packages	5	8	2	3	6	4
Classes	15	19	5	10	23	16
LOC	1426	978	616	2682	1758	993

FinG 3. Four software were used. In this case, we used more complex software than in FinG 1/FinG 2. All of them also implement familiar Java applications: i) Quilt is a software development tool that measures test coverage; ii) JMoney is a personal finance (accounting) manager; iii) jParse is an Eclipse plugin in which you can parse XML that is returned from a jQuery Ajax request; and iv) SQuirreL SQL Client is a graphical Java program that allows one to view the structure of a JDBC compliant database, browse the data in the tables, issue SQL commands among other functions. Table 2 characterizes the software in terms of number of packages, classes and LOC.

Table 2: Software used in FinG 3

Software	Quilt	JMoney	Squirrel	jParse
Packages	20	4	3	4
Classes	104	79	73	69
LOC	13030	11299	9081	32270

3.3 Design

All experiments were run in a laboratory at UFBA. Participants had about three hours to carry out the task. Each participant worked at an independent workstation.

FinG 1 and FinG 2. At each workstation, we set up two Eclipse IDE. We fitted the SourceMiner only for one of the Eclipse in the workstation. Each Eclipse IDE had three of the six programs in their workspace. The workstations were divided into two groups. In FinG 1, there were six participants in the group 1 and five participants in the group 2. In FinG 2, there were 14 participants in the group 1 and 13 participants in the group 2. We present the distribution in Table 3. As an example, look at the participant 14, in FinG 1, group 1. In his/her workstation, the Eclipse fitted with SourceMiner had Chess, Jackut and Solitaire programs. The other Eclipse (without SourceMiner) had Monopoly, Tetris and Tic Tac Toe programs in its workspace.

Table 3: Setup of Eclipse and allocation of participants (FinG 1 and FinG 2)

Group	With SourceMiner	Without Source Miner	Participants' id (FinG 1)	Participants' id (FinG 2)
1	Chess, Jackut and Solitaire	Monopoly, Tetris and Tic Tac Toe	14, 21, 32, 35, 42, 44	1, 3, 4, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27
2	Monopoly, Tetris and Tic Tac Toe	Chess, Jackut and Solitaire	13, 15, 25, 31, 41	2, 5, 6, 8 10, 12, 14, 16, 18, 20, 22, 24, 26

FinG 3. The only difference in design among FinG 1/FinG 2 and FinG 3 was the number of programs at each of the two Eclipse-by-workstation. In FinG 3, each Eclipse had two programs, instead three, in its workspace. The workstations were also divided into two groups. There were 13 participants in the group 1, and 11 participants in the group 2. We present the distribution in Table 4.

3.4 Procedure

We collected drivers from the Answer form. In FinG 1 and FinG 2, we collected the drivers checked in the form. We performed a simple coding, for the cases where participants wrote drivers. In FinG 1, we identified six new drivers, and in FinG 2 we identified five new drivers. In FinG 3, there was no options of drivers in the Answer form. Participants wrote

Table 4: Setup of Eclipse and allocation of participants (FinG 3)

Group	With Source-Miner	Without Source-Miner	Participants' ID (FinG 3)
1	jParse and Quilt	JMoney and Squirrel	1, 2, 3, 6, 7, 10, 11, 12, 13, 21, 23, 24 and 25
2	JMoney and Squirrel	jParse and Quilt	9, 14, 15, 16, 17, 18, 19, 20, 27, 28, 29

down the drivers and we coded them. It was a simple process because the terms are normally similar. For example, we consider the term "dependency" as the driver for cases where participants wrote down sentences like "class has high coupling", or "class is dependent of many classes". In FinG 3, the coding was checked by two experienced researchers.

3.5 Execution

Participants of the experiments were trained on god class concept and on SourceMiner tool. We performed the two trainings in different weeks. In order to guarantee some expertise using SourceMiner to detect god class, we performed, as part of the training, a practical exercise in the lab. We asked participants to search god class using the SourceMiner. All participants were involved in all training activities.

4. RESULTS

Our first analysis was based on the "stronger" drivers. We also considered how visualization impacted the drivers' choice. To perform the analysis, we used the bubble plot metaphor, where we show the distribution of drivers, by each participant. We considered both with and without visualization cases independently.

The second analysis was based on drivers' choice agreement. In this case, we used two different approaches: we investigated common set of drivers chosen by participants, and we performed a statistical agreement test.

4.1 Analysis of stronger drivers

FinG 1. In Figure 2, we show results for the case with visualization. The vertical axis shows the drivers, and the horizontal axis shows the id of the participants. The bubbles show the number of times each participant chose the driver. We highlighted the stronger drivers: "class is highly complex" and "method is highly complex". They were chosen by all and almost all (nine in 11) participants, respectively. In another level, seven participants chose "class has many dependencies".

Figure 3 shows distribution of drivers for the cases where visualization was not used. The stronger drivers were same as in the case where visualization was permitted. The main difference between both with and without visualization cases was related to the driver "class has many dependencies". As we can see in Figure 3, only the participant 13 chose the driver, and only once.

FinG 2. Figure 4 shows drivers' distribution for the case where visualization was used. As in FinG 1, the stronger drivers were "class is highly complex" and "method is highly complex". They were chosen by 26 and 21 participants, respectively. The total number of participants was 27. The following drivers were "method is misplaced" and "class is special": 11 participants chose them. "class has many dependencies" was chosen by ten participants.

Figure 5 shows drivers' distribution for the case where visualization was not used. Again, the stronger drivers were "class is highly complex" and "method is highly complex". As in the case with visualization, the following driver was "method is misplaced", chosen by 12 participants.

FinG 3. There is a difference on drivers' names of FinG 3 and FinG 1/FinG 2, which happened because participants had to write down their decision drivers in the FinG 3 Answer form (there was no options as in the FinG 1/FinG 2's Answer forms). Due to this, we found the drivers from a coding process. As a result, the drivers' names were simpler in FinG 3 than in FinG 1/FinG 2, and we did not correlate them. In Figure 6, we show drivers for the case with visualization. The two main drivers were "dependencies" and "LOC value", chosen by 16 and 15 participants, respectively. The total number of participants was 24. In another level, "responsibilities" (which means a class has many roles in the system) and "number of methods" were chosen by 10 and 9 participants, respectively.

In Figure 7, we show drivers' distribution of FinG 3, for the case where visualization was not permitted. "responsibilities" was the strongest driver, chosen by 18 participants. "loc value" and "dependencies" were chose by 13 and 11 participants, respectively.

4.2 Agreement on drivers

Set of drivers. At first, we checked all set of drivers chosen by the participants. In FinG 1 there was not a common set of drivers, i.e., there was not two participants choosing the same set of drivers. In FinG 2, there were two identical sets: two participants chose "class is highly complex" and "method is highly complex"; and three participants chose the same previous set, adding "class is special". FinG 3 was the experiment with more identical set of drivers. It had 24 participants and four identical sets: two participants chose only "responsibilities"; three participants chose "responsibilities" and "LOC value"; three participants chose "responsibilities" and "misplaced methods"; and two participants chose "responsibilities", "LOC value" and "misplaced methods".

Agreement test. To evaluate the level of agreement on drivers' choice, we adopted the Finn coefficient [5], as opposed to Kappa coefficient [6], the most common coefficient we found. We adopted Finn because Feinstein [4] shows that one can have high agreement rate and low values of the Kappa coefficient, when the variance on values of raters is low. Other data analysis researchers also addressed the problems with Kappa values [8, 17, 24]. We discuss this issue in [19, 21].

In our analysis, we adopted the agreement classification level of Landis and Koch [9]: slight, for values between 0.00 and 0.20; fair (between 0.21 and 0.40); moderate (between 0.41 and 0.60); substantial (between 0.61 and 0.80); and almost perfect (between 0.81 and 1.00) agreement.

It is important to note that, in this analysis, we disregarded the number of times that the participants chose the driver. If a participant chose a driver many times and other chose the driver only once, we considered the two participants agreed on the driver. Table 5 shows the Finn coefficient value and the level of agreement, for each experiment. For FinG 1 and FinG 3, with visualization, the agreement was fair. For all other cases, the agreement was moderate.

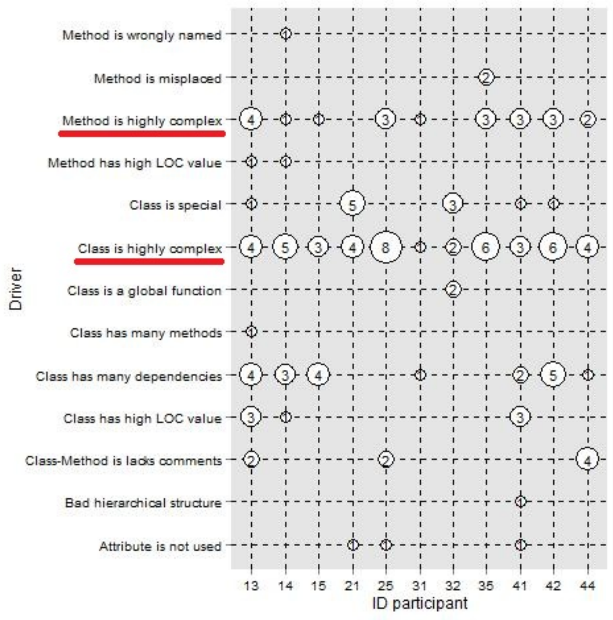


Figure 2: Drivers' distribution by participants of FinG 1, for the case with visualization

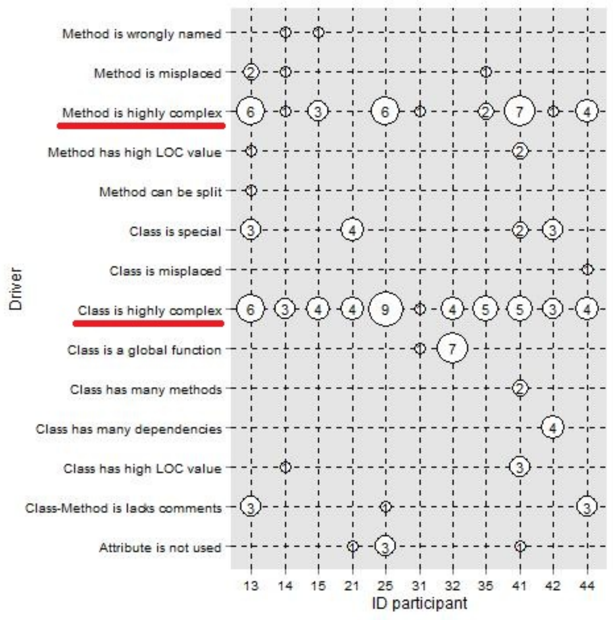


Figure 3: Drivers' distribution by participants of FinG 1, for the case without visualization

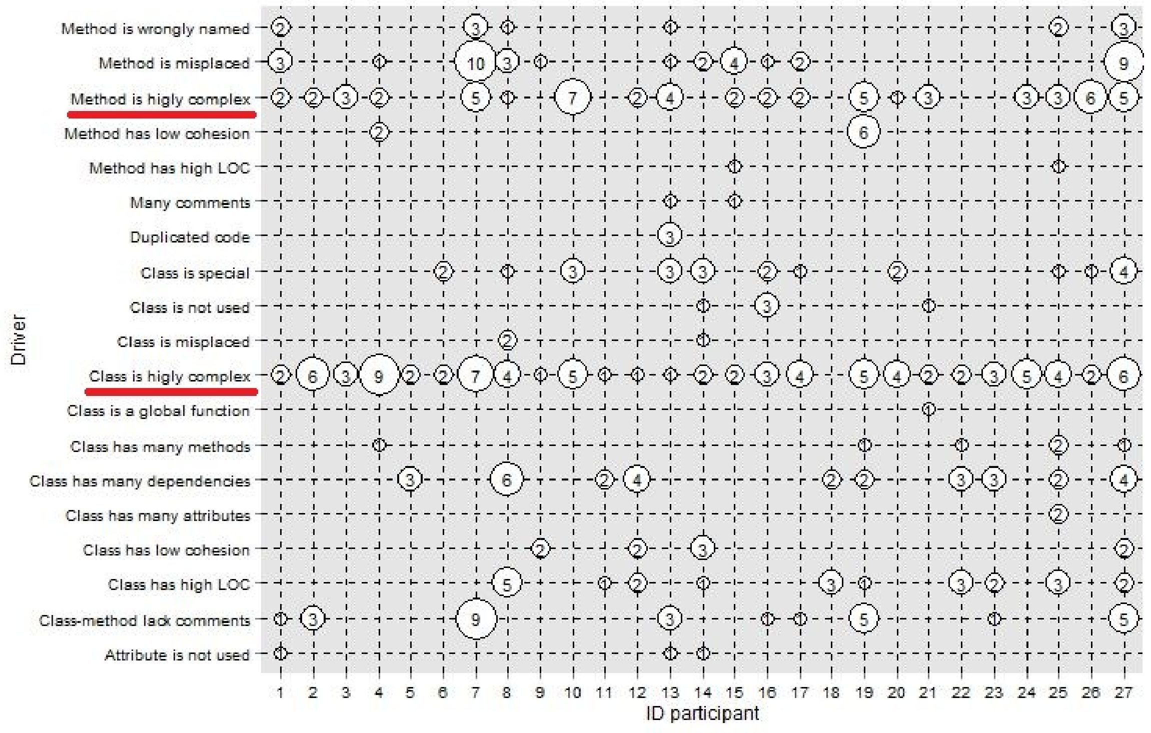


Figure 4: Drivers' distribution by participants of FinG 2, for the case with visualization

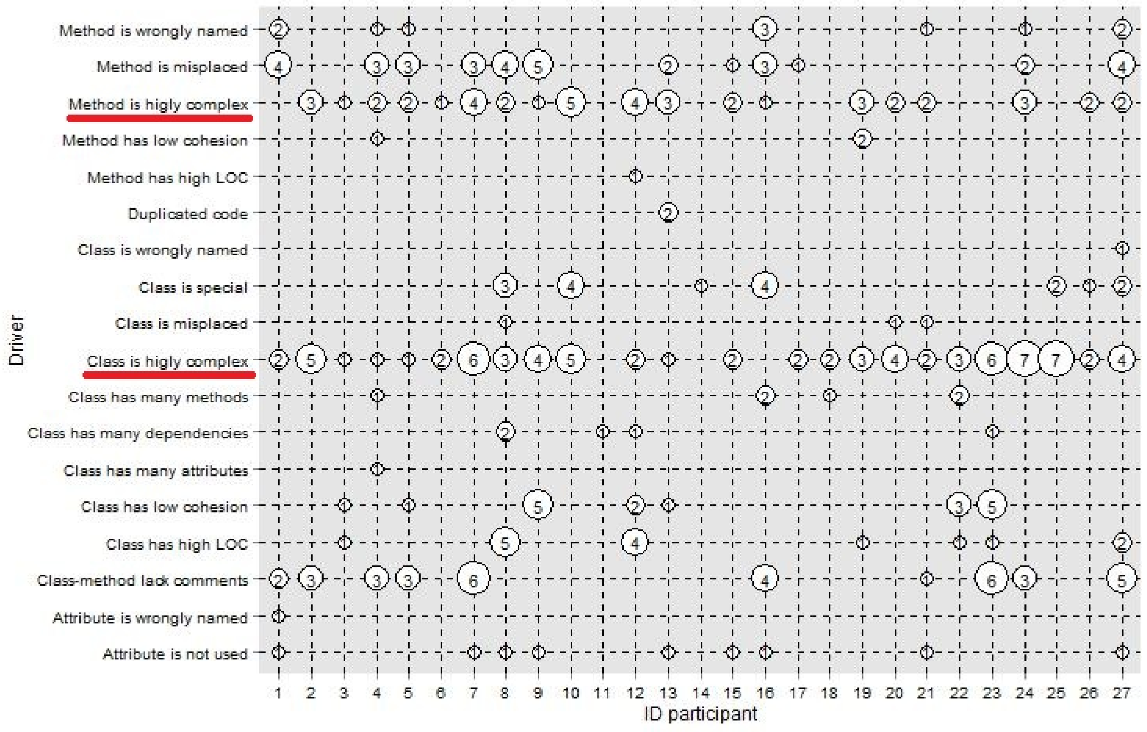


Figure 5: Drivers' distribution by participants of FinG 2, for the case without visualization

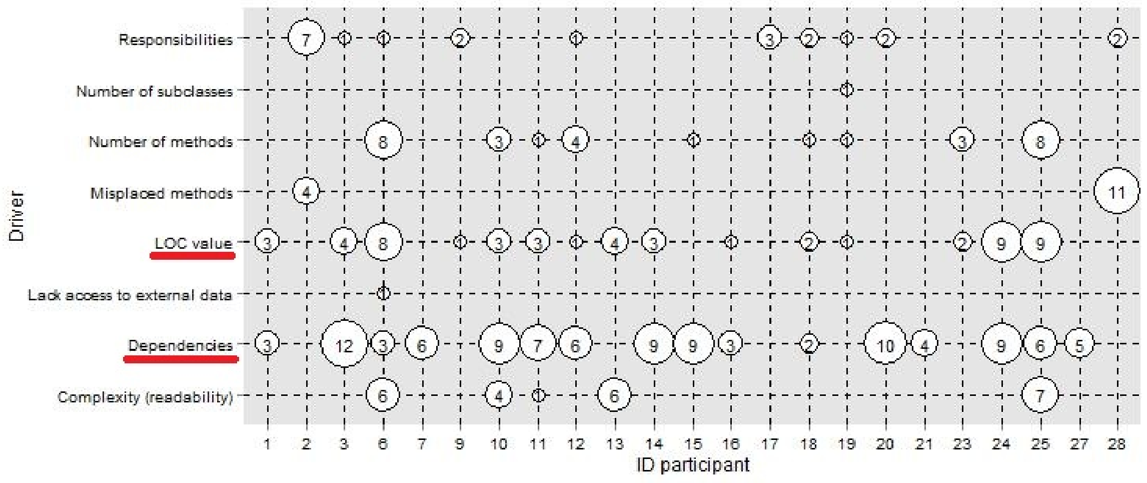


Figure 6: Drivers' distribution by participants of FinG 3, for the case with visualization

Table 5: Agreement test for FinG 1, FinG 2 and FinG 3 considering both with and without visualization cases

Experi-	With visualization					Without visualization				
ment	#Drivers	#Part	Finn	p-value	Level	#Drivers	#Part	Finn	p-value	Level
FinG 1	13	11	0.396	0.000106	Fair	14	11	0.403	4.28e-05	Moderate
FinG 2	19	27	0.466	2.9e-19	Moderate	18	27	0.452	3.88e-17	Moderate
FinG 3	8	23	0.32	0.000383	Fair	13	24	0.486	1.57e-13	Moderate

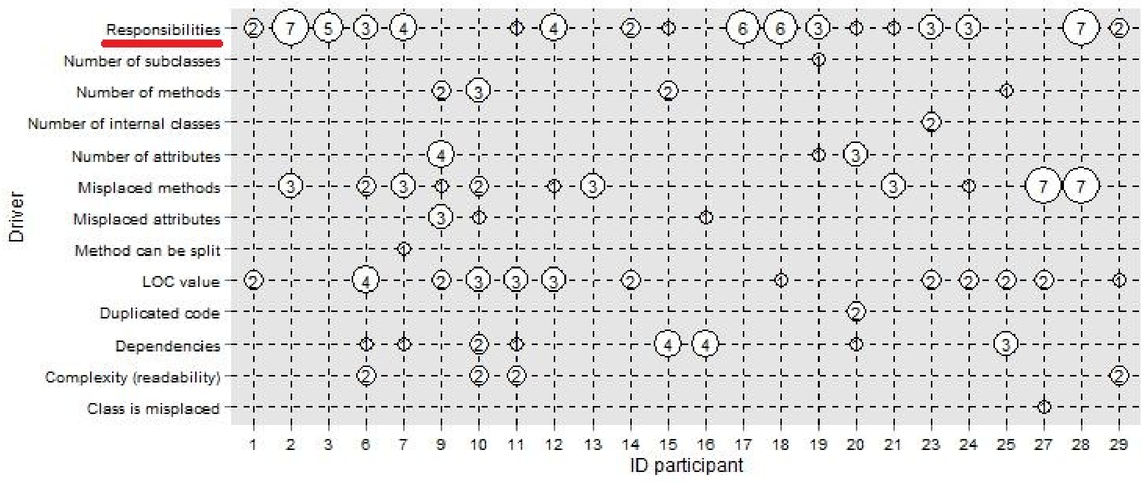


Figure 7: Drivers' distribution by participants of FinG 3, for the case without visualization

5. DISCUSSION

We compared results between FinG 1/FinG 2 and Schumacher et al. work [22] because we used the same drivers. Schumacher et al. highlighted the following drivers: "method is misplaced", "method is wrongly named", "method is highly complex", and "class is highly complex". "Method is misplaced" was the strongest driver in the Scumacher et al work. All these were also chosen in our experiments. However, the stronger drivers, in our case, were "class is highly complex" and "method is highly complex". After these, "method is misplaced" was the strongest driver. We conjecture that the "complexity" involves subjective aspects, which indicates the relevance of the subjectivity on drivers' choice. Considering technical aspects, our finding is based on the similarity of the results in both our and Schumacher et al. works: "method is misplaced" is the strongest driver on god class detection.

Another aspect we analyzed is related to the impact of visualization. "Dependency" was one important driver for the three experiments, when visualization was used: it was adopted very fewer times by a less number of participants when visualization was not used. For us, this is an evidence that some drivers are poorly supported by the current state of the practice. Another evidence can be observed in FinG 3 analysis, when no options of drivers were available in the Answer form. When visualization was used, the stronger drivers were "dependency" and "LOC value". While the visualization tool provides several views which show signs of dependency and size hierarchy, the current IDEs and the source code do not help in the identification of these drivers.

An interesting observations can be done comparing the distribution of drivers by participants. Excepted by the stronger drivers, all other were chosen by a small number of participants. The evidences are: the number of similar set of drivers among the participants and the fair/moderate level of agreement (Table 5). Due to this, we consider the agreement on drivers tends to be low, which has similarities with findings in [22]. For us, this reinforce the relevance of subjectivity on god class detection. The drivers' choice is more related to personal conceptualization of the smell than to technical aspects. This helps to characterize what we called the *code smell conceptualization problem* [19, 21].

6. THREATS TO VALIDITY

Our analysis of threats was based on Wohlin et al. [25].

External validity. Our first threat fits in the "interaction of selection and treatment" subcategory and is related to the participants' experience. Participants in FinG 2/FinG 3 were graduate students and participants in FinG 1 were undergraduate. One aspect mitigates the threat: almost all participants of FinG 2/FinG 3 had some professional experience developing software. Other threat fits in the "interaction of setting and treatment" subcategory. In this case, the threat is the type of software of FinG 1/FinG 2. We adopted six simple software. We mitigated this threat using medium software in FinG 3. We argue the impact of this threat was small because the agreement on drivers' choice was similar (low) for the three experiments.

Internal validity. The study has threats in two subcategories. The first one is "ambiguity about direction of causal influence". It is the training in the visualization tool. In our feedback form, participants indicated quality of training as good for all experiments. We consider this a weak empirical evidence of the validity of the training. However, we are confident about the answers because the software visualization tool is very simple and some participants gave comments after the experiment. Another subcategory of the internal validity is "maturation". Participants could be affected negatively because of boredom, because we used six software in FinG 1/FinG 2 and four software in FinG 3. We consider maturation a weak threat because the experiments were performed in 2 hour, on average. We consider this a reasonable period of time to do a task in a balanced way.

Conclusion validity. In the "reliability of treatment implementation" we have to consider if the visualization tool was appropriated for the identification of useful attributes on god class detection. We are confident this did not impact our results, because of the discussion occurred during the training. Another threat is related to the coding on textual information. We did not use interviews to explore the drivers. To mitigate this threat, the coding in FinG 3 was performed by two experienced researchers on the topic.

7. CONCLUSION AND FUTURE WORKS

In this work, we explored decision drivers adopted by developers detecting god class. To do this we performed three controlled experiments. During the experiments themselves, we asked to participants to indicate what issues in code they used to consider a class as a god class.

The main driver, when drivers' options were available in the answer form, was "class is high complex". When options were not available in the answer form, the main driver was "misplaced methods". We also found some drivers were not considered because lack of support in the current IDE's. In our case, "dependency" was an important driver only when software visualization was used. Another of our findings was: the agreement on drivers' choice is low, which is in accordance with other work in the subject [22].

To evolve our studies on the subject, we will explore other human factors affecting god class detection, such as experience or knowledge. We also intent to replicate the experiments focusing in other types of smells. To support replication, we provide the experimental packages[2], containing forms, data and software.

8. ACKNOWLEDGMENTS

This work was partially supported by SECTI-BA (Bureau of Science and Technology of Bahia) - Fraunhofer Project Center for Software & Systems Eng. agreement 2012/001.

9. REFERENCES

[1] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proc. of 15th CSMR*, 2011.

[2] G. Carneiro, M. Silva, L. Maia, E. Figueiredo, C. Sant'Anna, A. Garcia, and M. Mendonça. Identifying code smells with multiple concern views. In *Proc. of the 1th CBSOFT*, 2010.

[3] G. F. Carneiro and M. G. Mendonça. Sourceminer - a multi-perspective software visualization environment. In *Proc. of the 15th ICEIS*, 2013.

[4] A. R. Feinstein and D. V. Cicchetti. High agreement but low kappa: I. the problems of two paradoxes. *Journal of Clinical Epidemiology*, 43(6):543–549, 1990.

[5] R. H. Finn. A note on estimating the reliability of categorical data. *Educational and Psychological Measurement*, 30:71–76, 1970.

[6] J. Fleiss et al. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[7] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman, 1999.

[8] K. Gwet. Kappa statistic is not satisfactory for assessing the extent of agreement between raters. *Statistical Methods for Inter-rater Reliability Assessment*, (1):1–5, 2002.

[9] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):pp. 159–174, 1977.

[10] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice.* Springer-Verlag, 2005.

[11] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, July 2007.

[12] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa. Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems. In *Proc. of the 11th AOSD*, 2012.

[13] M. V. Mäntylä and C. Lassenius. Drivers for software refactoring decisions. In *Proc. of 5th ISESE*, 2006.

[14] L. Moonen and A. Yamashita. Do code smells reflect important maintainability aspects? In *Proc. of 28th ICSM*, 2012.

[15] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Proc. of the 26th ICSM*, 2010.

[16] J. Padilha, E. Figueiredo, C. Sant'Anna, and A. Garcia. Detecting god methods with concern metrics: An exploratory study. In *Proc. of the 7th LA-WASP, co-allocated with CBSoft*, 2013.

[17] D. Powers. The Problem with Kappa. *EACL 2012*, pages 345–355, 2012.

[18] A. J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1st edition, 1996.

[19] J. A. Santos, M. G. de Mendonça, C. P. dos Santos, and R. L. Novais. The problem of conceptualization in god class detection: agreement, strategies and decision drivers. *Journal of Software Engineering Research and Development (JSERD)*, 2(11):1–33, 2014.

[20] J. A. Santos and M. Mendonça. Identifying strategies on god class detection in two controlled experiments. In *Proc. of the 26th SEKE*, 2014.

[21] J. A. Santos, M. Mendonça, and C. Silva. An exploratory study to investigate the impact of conceptualization in god class detection. In *Proc. of the 17th EASE*, 2013.

[22] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw. Building empirical support for automated code smell detection. In *Proc. of the 4th ESEM*, 2010.

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

[24] G. J. Whitehurst. Interrater agreement for journal manuscript review. *American Psychologist*, 39(1):22–28, 1984.

[25] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering.* Springer Berlin, 2012.

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

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

[2]https://wiki.dcc.ufba.br/LES/JoseAmancio

Appendix D

PAPER PRESENTING ANALYSIS OF PARTICIPANTS' STRATEGIES ON GOD CLASS DETECTION

This appendix presents the paper "Identifying strategies on god class detection in two controlled experiments". The paper was published in the The Twenty-Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE 2014).

Identifying strategies on god class detection in two controlled experiments

José Amancio M. Santos
State University of Feira de Santana
Technology Department
Bahia, Brazil
Email: zeamancio@ecomp.uefs.br

Manoel G. de Mendonça
Fraunhofer Project Center for Software & Systems Eng.
Federal University of Bahia
Bahia, Brazil
Email: mgmendonca@dcc.ufba.br

***Abstract*—*Context*: "Code smell" is commonly presented as indicative of problems in design of object-oriented systems. However, some empirical studies have presented findings refuting this idea. One of the reasons of the misunderstanding is the low number of studies focused on the role of human on code smell detection. *Objective*: Our aim is to build empirical support to exploration of the human role on code smell detection. Specifically, we investigated strategies adopted by developers on god class detection. God class is one of the most known code smell. *Method*: We performed a controlled experiment and replicated it. We explored the strategies from the participant's actions logged during the detection of god classes. *Result*: One of our findings was that the observation of coupling is more relevant than the observation of attributes like LOC or complexity and the hierarchical relation among these. We also noted that reading source code is important, even with visual resources enhancing the general comprehension of the software. *Conclusion*: This study contributes to expand the comprehension of the human role on code smell detection through the use of automatic logging. We suggest that this approach brings a complementary perspective of analysis in discussions about the topic.**

Keywords: Code smell, god class, controlled experiment

I. Introduction

Object-Oriented (OO) design challenges are a key aspect in Software Engineering (SE). A poor design can lead to future problems when evolving the code. Important works address the problem from different perspectives. Fowler [1] presented several scenarios where the code may indicate a bad design. They refer to these cases as "code smells". A code smell could be a sign that one should refactor the code to improve it. Lanza and Marinescu [2] focused on OO metrics to characterize bad design. They refer to bad design as "disharmonies". These terms are used to define potential design problems. In this paper, we adopt the term code smell, or simply smell, to refer to those design problems.

Despite the conception of code smell be widely accepted as indicative of bad design, some empirical studies have presented findings in other directions. Sjøberg et al. [3], for example, investigated the relationship between code smell and maintenance effort. They noted that none of the investigated code smells were significantly associated with increased maintenance effort. Macia et al. [4] investigated the relationship between code smells and problems that occur with an evolving system's architecture. In their study, they noted that many of the detected code smells were not related to architectural problems. Zhang et al. [5] performed a systematic mapping study on the subject and declared that "...we do not know whether using Code Bad Smells to target code improvement is effective". In general, the authors agree that more empirical studies are necessary to better understanding the smell effect [3], [5], [6]. Specifically, the role of humans has been little studied [7]. The studies focused on human aspects commonly address agreement and decision drivers (ex. [6], [8], [9]).

The evaluation of the human aspects is not a simple task because of the extensive number of context variables that affect the human perception of code smell. In this work, our general aim is to build more empirical studies evaluating human aspects and code smells. Specifically, we have focus on the strategies adopted on *god class* detection, one of the most known code smell. Evidence of the relevance on the topic are other studies with similar aim [9], [10], [11], [6]. We explored the strategies of god class detection based on actions logged in a non-intrusive way in two controlled experiments (an original experiment and a replication).

The rest of the paper is structured as follows. Section II address some concepts and summarizes prior empirical studies that address the subject. Section III presents the planning and execution of the experiments. Section IV and V contains the results and a discussion about them. Section VI discusses the threats to the validity of the study. Lastly, Section VII presents our conclusions and proposed future works.

II. Context and related works

God class is a central concept in our study. The term was coined by Lanza and Marinescu [2] to refer classes that tend to centralize the intelligence of the system. Lanza and Marinescu presented an heuristic for god class identification based on metrics. The authors also presented the code smell brain class, in a similar way: complex classes that tend to accumulate an excessive amount of intelligence. The main difference is that a god class accesses directly many attributes from other classes. Another code smell with similar characteristics was presented by Folwer [1], but in this case subjectively. He defined the code smell large class, which is a class that try to do too much. These three code smells have a similar concept.

We presented to participants of our experiments a set of support questions to guide them. The questions were extracted from another empirical study [6] and concentrate the general idea of the three code smells. Some examples are: "Does the class have more than one responsibility ?" and "Does the class have functionality that would fit better into other classes ?" We did not define formally the god class concept in this paper because we focused on the strategies, not in the correctness on detection. From now we will adopt only the term God Class.

A. Studies with similar aim of ours

Schumacher et al. [6] investigated the way developers detect god class, then compared these results to automatic classification. They built on and extended Mäntylä and Lassenius's [9] work. Schumacher et al. study was done in a professional environment, with two real projects and two participants of each project. The participants were introduced to the god class smell in a short presentation and were them asked to detect them in specific code pieces. During this task, they received the list of questions (the same we adopted) to help with the identification of god classes.

Schumacher et al. used a "think-aloud" protocol (recorded as audio) and data collection forms. Coding was carried out to identify drivers and answers from the data collection form were used to evaluate time and agreement. Their main findings were: (1) there was a low agreement between the participants; and, (2) "misplaced method" was the stronger driver for god class detection. Related to the evaluation of automatic detection, their main findings were: (1) an automated metric-based pre-selection decreases the effort needed for manual code inspections; and, (2) automatic detection followed by manual review increases the overall confidence.

A study with aims similar to Schumacher's was presented by Mäntylä et al. in [11] and [10]. Through a survey, they asked participants about 23 smells and used a scale from 1 (lack) to 7 (large presence) to evaluate the presence of smells in a piece of code. They received 12 completed questionnaires from 18 sent to developers in a small software company. In one of the findings the authors declare: "*the use of smells for code evaluation purposes is hard due to conflicting perceptions of different evaluators*".

B. Studies with similar strategy of analysis of ours

We found only one work where the researchers used log of actions to try identify strategies of detection. Carneiro et al. [12] presented a software visualization tool with support to concerns. They investigated how the tool, named SourceMiner, helped developers on code smell detection. We will detail the SourceMiner thereafter because it is the tool that we adopt in this paper. Carneiro et al. also logged the participant's actions, but how they had focus on the presentation of the tool and the support to concern, their observations on this topic were restricted.

Another empirical study that adopted log to understand performing of a Software Engineering task was presented by Wang et al. [13]. In this case the authors investigated how developers perform feature location tasks. Their analysis was based on 76 hours of full-screen videos of 38 developers working on 12 feature-location tasks. They adopted log to analyze the videos: they developed a tool to create and maintain a log of each developer's work while they watched the videos. They also detached that there are few works evaluating strategies adopted by developers performing Software Engineering tasks.

III. The Finding God Class Experiment

In this section we present settings of two controlled experiments: we named *Finding God Class (FinG)*. FinG aimed to address a set of context-aspects on god class detection, such as agreement, effort, strategies, and others. We had already partial results in [14]. In this paper we focused only on the investigation of the strategies adopted by the participants. We will call FinG 1 the original experiment, and FinG 2 the second experiment, which is a replication of FinG 1. In the following sections, the only difference among the FinG 1 and FinG 2 settings is the experimental unit.

A. The Experimental Planning

1) Experimental Unit: FinG 1 involved 11 undergraduate Computer Science students from the Federal University of Bahia (UFBA), in Brazil. All students were enrolled on the Software Quality course offered in the first semester in 2012. The participation in the experiment was voluntary. FinG 2 involved 23 graduate Computer Science students also from the Federal University of Bahia. All students were enrolled in the Experimental Software Engineering course offered in the second semester in 2012. Due to the special circumstances, all students of the course were (or had been) professionals. In this case, the participants received grades for their participation.

2) Tools: We adopted four software tools in the experiment[1]: (i) The Eclipse Indigo IDE; (ii) Usage Data Collector (UDC), an Eclipse plug-in for collecting IDE usage data information (interactions between participants and Eclipse can be accessed by the log of UDC). This tool is embedded in the Eclipse Indigo IDE; (iii) Task Register plug-in, a tool we developed to enable participants to indicate what task was being done at a given moment. This information was also registered in the UDC log. What all the participants had to do was to click on an item of the "Task Register" view to indicate when they were starting or finishing a task. The Figure 1-F shows the "Task Register" view; and (iv) SourceMiner, an Eclipse plug-in that provides visual resources to enhance software comprehension activities [12], [15].

We detach the SourceMiner tool because our analysis of the strategies on god class detection was based on it. The tool has visual resources that make easier the design comprehension of the programs. The SourceMiner has five visual metaphors, divided in two groups. The first group is formed by three coupling views. These views show the different types of coupling, as direct access to attributes or method calling, for instance. Moreover, they show the direction of the coupling. The coupling views are based on radial graphs, matrix of relationships, and tabular view (Figures 1 B, D and E). The second group is formed by two hierarchical views. These views associate LOC, complexity and number of methods of classes to area, width and length of rectangles. The treemap view (Figure 1 A) shows the hierarchy of package-class-method. The more internal rectangles represent methods. The area and color are used to highlight the attributes of LOC or cyclomatic complexity. The red boards represent classes and the yellow boards represent packages. The Polimetric view (Figure 1 C)

[1]Eclipse IDE - http://www.eclipse.org/downloads/; UDC - http://www.eclipse.org/epp/usagedata/; Task Register - private; SourceMiner - http://www.sourceminer.org/

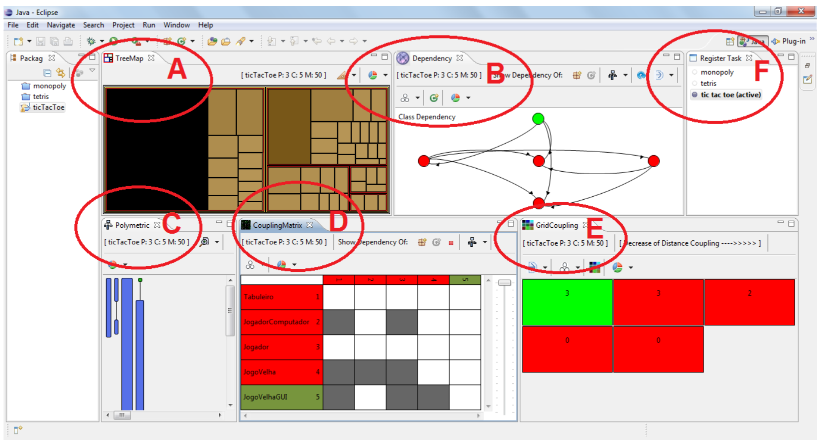


Fig. 1. Views of the SourceMiner and the Task Register plug-in

shows the hierarchy between classes and interfaces. The wider rectangles represent classes with more number of methods. Rectangles of greater length represent classes with larger LOC.

3) Software Artifacts: Six programs were used in the experiment. All of them implement familiar applications or games in Java. Chess, Tic Tac Toe, Monopoly and Tetris implement known games. Solitaire-Freecell (Solitaire) is a framework for card games with Solitaire and Freecell. Jackut implements a simple social network application, in the line of Facebook or Orkut. Table I characterizes the used programs in terms of the number of packages, number of classes and NLOC. It is possible to note that Monopoly is the software with higher NLOC: 2682. Due this, we consider all them simple programs. We chose simple programs to make easy their comprehension by participants because we were interested in the strategies, not in the evaluation of the difficulties on god class detection.

TABLE I. SOFTWARES USED AS STUDY OBJECTS

Software	Chess	Jackut	Tic	Monopoly	Solitaire	Tetris
Packages	5	8	2	3	6	4
Classes	15	19	5	10	23	16
NLOC	1426	978	616	2682	1758	993

4) Design: The both FinG 1 and FinG 2 experiments were run in a laboratory at UFBA. Participants had about 2 hours to carry out the task. Each participant worked at a separate workstation. At each workstation, we set up the Eclipse IDE fitted with the SourceMiner, the Task Register and UDC plug-ins. Each Eclipse with SourceMiner had three of the six programs in their workspace. The workstations were divided into two groups. In FinG 1 there were six participants in the group 1 and five participants in the group 2. In FinG 2 there were 13 participants in the group 1 and 11 participants in the group 2. We present the distribution in the Table II.

TABLE II. DISTRIBUTION OF THE PARTICIPANTS BY GROUP

Group	Programs	Participants of FinG 1 (ID)	Participants of FinG 2 (ID)
1	Chess, Jackut and Solitaire	14, 21, 32, 35 42 and 44	1, 3, 4, 7, 9, 13, 15, 17, 19 21, 23, 25 and 27
2	Monopoly, Tetris and Tic Tac Toe	13, 15, 25, 31 and 41	5, 6, 8, 10 12, 14, 16, 18 20, 22 and 26

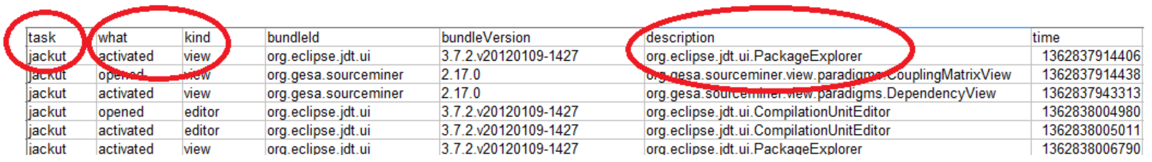
B. Execution

1) Preparation: Participants of the both FinG 1 and FinG 2 experiments were trained on the god class concept and on the SourceMiner tool. We performed the two training in different weeks. In order to guarantee some expertise using the SourceMiner to detect god class, we performed, as part of the training, a practical exercise in the lab asking them to search god class using the SourceMiner. All participants of both experiments performed the exercise.

2) Data: We used the UDC plugin to log participant's actions while the experiment was running. UDC is a framework for collecting usage data on various aspects of the Eclipse workbench. It gathers information about the kinds of activities that the user is doing in the IDE (i.e. activating views, editors, etc.). The Task Register (Figure 1-F) was used to enrich the UDC log with higher level information. Figure 2 shows a clipping of the UDC log annotated by the Task Register plug-in. The first column ("task") does not exist in the original UDC log. It was added by the Task Register. We highlighted columns that we were interested in. The first column ("task") indicates the program for which the participant was doing the god class detection task. Columns "what", "kind" and "description" describe the actions. For example, the first line represents: user activated the Package Explorer view. As a result, we have sequences of actions for each participant and for each program.

IV. RESULTS

To investigate the strategies adopted by the participants we explored their preferences for the views of SourceMiner during the detection of god classes. We used the UDC log of actions to do this. Some participants read more and used fewer views, others did the opposite. To evaluate these aspects we calculated the ratio between the number of classes investigated for each program and the use of views and reading. We counted the number of actions related to reading (activation of the Compilation Unit Editor), activation of hierarchical views (Polimetric or Tree Map), and activation of coupling views (Dependency, GridCoupling or CouplingMatrix). For example, the participant 14 of FinG 1 did not activate the Compilation Unit Editor (i.e. read the code), activated some of the coupling



task	what	kind	bundleId	bundleVersion	description	time
jackut	activated	view	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.PackageExplorer	1362837914406
jackut	opened	view	org.gesa.sourceminer	2.17.0	org.gesa.sourceminer.view.paradigms.CouplingMatrixView	1362837914438
jackut	activated	view	org.gesa.sourceminer	2.17.0	org.gesa.sourceminer.view.paradigms.DependencyView	1362837943313
jackut	opened	editor	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.CompilationUnitEditor	1362838004980
jackut	activated	editor	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.CompilationUnitEditor	1362838005011
jackut	activated	view	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.PackageExplorer	1362838006790

Fig. 2. Clipping of the user UDC log

views 41 times and activated some of the hierarchical views 39 times. He/she did this for the programs Chess, Jackut and Solitaire. The total number of classes for these three programs is 57. We defined the ratios for the participant 14 as 0/57, 41/57 and 39/57 for reading, use of coupling views and use of hierarchical views, respectively.

The Figure 3 (shown on next page) shows in a bar graph the ratio using views for all participants of FinG 1. The ratio for the use of reading is in red. In green we show the ratio for the use of coupling views. In blue the ratio for the use of hierarchical views. We grouped participants with similar strategy. The lines above the bars group name them. In the Figure 4 we show the results for FinG 2.

From the Figures 3 and 4 we identified a set of strategies. For example, in the Figure 3 we noted that the first three participants (id's 14, 21 and 42) had similar focus on the use of hierarchical and coupling views and they did not read the source code or used very few reading. The first participant of FinG 2, in the Figure 4, adopted the same strategy. We called it Strategy 1. In the Table III we show all strategies identified. All them were identified according to high focus or none/few use on the views or reading. For example, for the Strategy 3, the participants had focus on coupling views, and none or few use of reading. It is important to note that we did not define a threshold for high or none focus because we compared preferences between views considering participants individually. We disregarded the number of clicks on the views because we did not analyze the effort.

After identifying the strategies we counted the number of participants with focus on each type of views or reading. We also counted the number of participants with few or none use of each type of view or reading. We show the result in the Table IV. For the Strategy 1, there were three participants of the FinG 1 (id's 13, 21 and 42) and one participant of the FinG 2 (id 4). Then in the Table IV we registered that there were four participants with focus on coupling and hierarchical views, and four participants with few or none use of reading. The two last lines in the table show the total of participants with focus or with few or none use of views or reading and the percentage considering the total of participants of both FinG 1 and FinG 2 experiments (35).

V. Discussion

From the Table IV we noted that the god class detection was mainly focused on coupling views. Almost 63% of the participants adopted strategies based on coupling views. Once coupling was preferred instead hierarchical attributes, we suggest that on god class detection to observe coupling is more relevant. As the essence of the god class concept is related to classes with many roles, we linked the idea of participants on identifying many roles with the coupling among classes. Hierarchical attributes, as LOC or complexity, were less adopted.

Another interesting finding is that, even with visualization resources, 40% of the participants had focus on reading in their strategies. For us, this is an evidence that some reading is necessary on god class detection. Our conjecture is that the

TABLE III. Strategies adopted

N	Strategy	Participants of FinG 1 (ID)	Participants of FinG 2 (ID)
1	Similar focus on coupling and hierarchical/ few or no reading	14, 21, 42	4
2	Similar focus on coupling and hierarchical		12, 18, 19
3	Focus on coupling/ few or no reading	31, 41	5, 10, 20
4	Focus on reading/ few hierarchical	15, 25	7, 8, 16
5	Focus on coupling		3, 15, 17, 22
6	Similar focus on reading and coupling		13, 27
7	Focus on reading/ few coupling		14, 25
8	Focus on hierarchical/ few reading		26, 23
9	Similar focus on coupling, hierarchical and reading	44	9, 21
10	Focus on coupling/ few or no hierarchical		6
11	Focus on reading		1
12	Focus on hierarchical/ few coupling	35	
13	Only reading/no views	32	
14	Focus on hierarchical	13	

TABLE IV. Number of participants focusing on views

Strategy	Reading		Coupling		Hierarchical	
	High focus	Few/ none	High focus	Few/ none	High focus	Few/ none
1		4	4		4	
2			3		3	
3		5	5			
4	5					5
5			4			
6	2		2			
7	2			2		
8		2			2	
9	3		3		3	
10			1			1
11	1					
12				1	1	
13	1			1		1
14					1	
Total	14	11	22	4	14	7
% (out of 35)	40.0	31.4	62.9	11.4	40.0	20.0

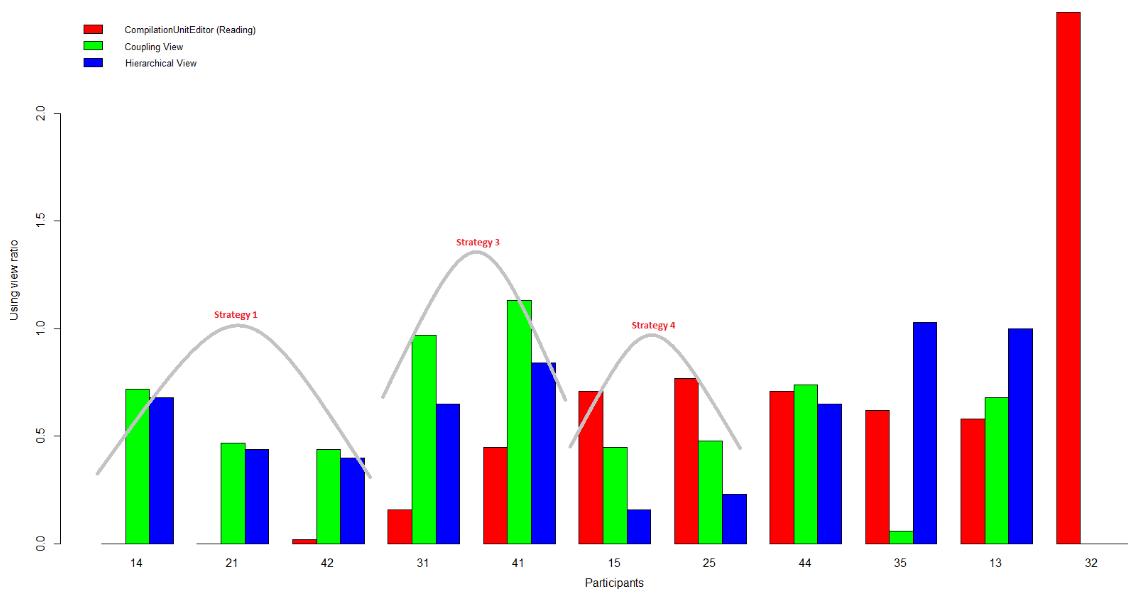


Fig. 3. Ratio of using reading, coupling and hierarchical views for FinG 1

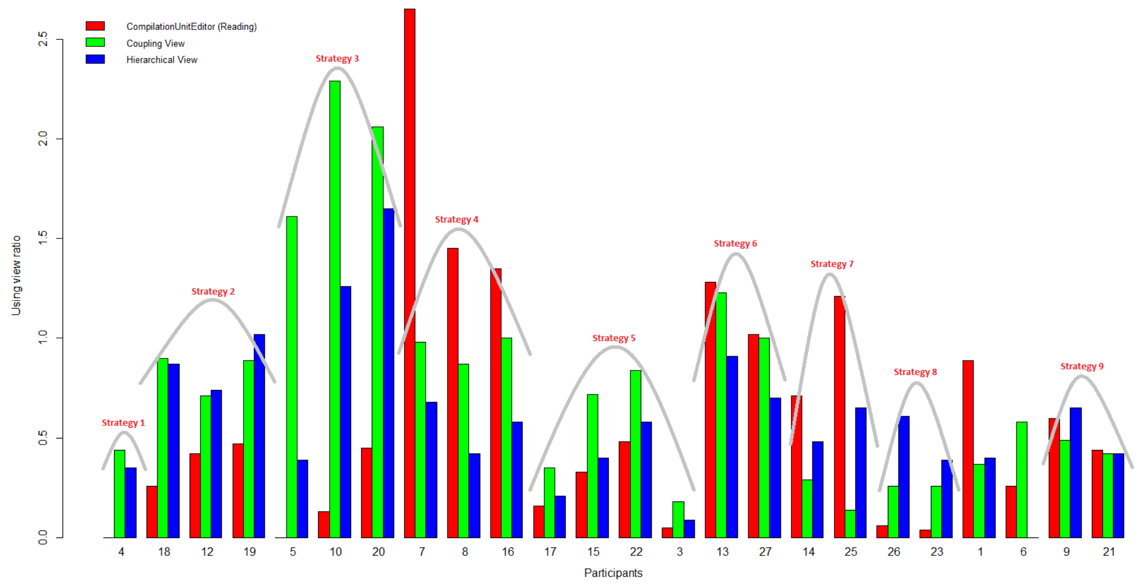


Fig. 4. Ratio of using reading, coupling and hierarchical views for FinG 2

reading is used to identify context aspects about the classes. For example, in some cases, to identify if a method is in according with the role of the class, reading source code is important. On the other hand, the few or none use of reading had highest percentage among the few use of views, 31.4%. We consider that the value is not high, but it can be an evidence that visual resources are important because make possible for developers filter specific classes to read.

An interesting observation is that our analysis presents a non usual perspective investigating code smell empirically. Papers addressing how developers detect code smell usually depends on the human answers, in interviews [6], questionnaires [9] or surveys [11], [10]. We adopt a non usual technique. The main benefits of the use of log are the high volume of data and the absence of disparity between the participants responses and the "reality" [16], [17]. We highlight the importance of the use of complementary strategies of data collection to analyze similar problems from different perspectives. Due to this focus and the space constraints, we did not present analysis of other variables (such as the impact of the experience) in this paper, which is possible by the setting of FinG.

VI. Threats to Validity

Our analysis of threats was based on Wohlin et al. [18].

External validity. Our first threat fits in the "interaction of selection and treatment" subcategory and is related to fact that the participants in FinG 1 were undergraduate students.

Although this aspect can be considered a problem for generalization, we found similar strategies in both FinG 1 and FinG 2 experiments, which make this threat weaker. Other threat to external validity fits in the "interaction of setting and treatment" subcategory. In this case, the threat is the type of programs. We adopted simple softwares. However, we argue the impact of this threat is small because we consider that the strategy is more dependent of the participant than of the complexity of the software.

Internal validity. A subcategory of the internal validity is "maturation". Participants could be affected because they do the same task over three programs so they may learn as they go and work faster. On the other hand they could be affected negatively because of boredom. We consider maturation a weak threat because the experiment was performed in 1 hour, on average. We consider this a reasonable period of time to do a task in a balanced way.

Conclusion validity. In the "reliability of measures" category we should report that the logged information represents the actions of the participants only indirectly. They represents actions of the Eclipse IDE. For example, if a developer changes the perspective in the Eclipse, some views are activated by the tool and these actions are registered in the log. To mitigate this aspect, we investigated the logging to evaluate actions in detail and eliminated lines clearly related to Eclipse actions. Moreover, these registers occurred for all participants and did not affect the general conclusion. In the "reliability of treatment implementation" we have to consider if that the visualization tool was appropriated for the identification of useful attributes on god class detection. We are confident that this threat did not have impact on our results because of the discussion occurred during the training based on exercise. Lastly, our findings were based on the analysis of he log, and we did not present inferential testing.

VII. Conclusion and future works

In this work was investigated how developers detect god class. More specifically, we explored strategies adopted by developers detecting god class. To do this we performed two controlled experiments. The setup of the experiments was based on the use of a software visualization tool fitted in the IDE Eclipse. We logged actions performed by participants using the visual resources grouped in three categories: use of Compilation Editor of the Eclipse (or reading code); use of hierarchical views (focused in attributes as LOC or complexity); or use of coupling views.

Our first finding was that coupling attributes are more relevant than LOC or complexity on god class detection. We also noted that, even with visual resources, that make possible to enhance the general comprehension of the design, reading of the source code remains important. We suggest that the reading is important for developers to evaluate if the methods are in accordance with the role of the classes. We also noted that the visual resources helped participants to filter candidate god classes to read. We consider our empirical results as an initial and complementary approach to investigate the real impact of code smell on software development.

To evolve our study in this topic, we will explore other context aspects of the experiments, as effort and decision drivers. We also intent replicate the experiments focusing in other types of code smell. To mitigate some limitations we will replicate the experiments with similar setting addressing more complex software. To support replication we provide the experimental package, available in the site of FinG 1[2] and FinG 2[3]. The packages contain forms, data and software.

Acknowledgment

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES)[4], funded by CNPq, grants 573964/2008-4.

References

[1] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

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

[3] D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, 2013.

[4] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems," in *Proc. of the 11th AOSD*, 2012, pp. 167–178.

[5] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," *J. Softw. Maint. Evol.*, vol. 23, no. 3, Apr. 2011.

[6] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *Proc. of the 4th ESEM*, 2010.

[7] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Softw. Engg.*, vol. 11, no. 3, pp. 395–431, 2006.

[8] L. Moonen and A. Yamashita, "Do code smells reflect important maintainability aspects?" in *Proc. of 28th ICSM*, 2012, pp. 306–315.

[9] M. V. Mäntylä and C. Lassenius, "Drivers for software refactoring decisions," in *Proc. of 5th ISESE*, 2006, pp. 297–306.

[10] M. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, no. 3, pp. 395–431, 2006.

[11] M. V. Mantyla, J. Vanhanen, and C. Lassenius, "Bad smells humans as code critics," in *Proc. of the 20th ICSM*, 2004, pp. 399–408.

[12] G. Carneiro, M. Silva, L. Maia, E. Figueiredo, C. Sant'Anna, A. Garcia, and M. Mendonça, "Identifying code smells with multiple concern views," in *Proc. of the 1th CBSOFT*, 2010.

[13] J. Wang, X. Peng, Z. Xing, and W. Zhao, "An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions," in *Proc. of 27th ICSM*, 2011.

[14] J. A. Santos, M. Mendonça, and C. Silva, "An exploratory study to investigate the impact of conceptualization in god class detection," in *Proc. of the 17th EASE*, 2013, pp. 48–59.

[15] G. F. Carneiro and M. G. Mendonça, "Sourceminer - a multi-perspective software visualization environment," in *Proc. of the 15th ICEIS*, 2013.

[16] J. Singer, S. E. Sim, and T. C. Lethbridge, "Software engineering data collection for field studies," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjoberg, Eds., 2008.

[17] H. K. Jonathan I. Maletic, "Expressiveness and effectiveness of program comprehension: Thoughts on future research directions," in *Frontiers of Software Maintenance,FoSM*, 2008, pp. 31–37.

[18] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012.

[2]wiki.dcc.ufba.br/LES/FindingGdoClassExperiment2012

[3]wiki.dcc.ufba.br/LES/FingTwo

[4]www.ines.org.br

BIBLIOGRAPHY

ABBES, M. et al. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: *Proc. of 15th European Conference on Software Maintenance and Reengineering (CSMR).* [S.l.: s.n.], 2011. p. 181–190. ISSN 1534-5351.

ABEBE, S. et al. The effect of lexicon bad smells on concept location in source code. In: *Proc. of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM).* [S.l.: s.n.], 2011. p. 125–134.

ALWIS, B. de; MURPHY, G. C. Using visual momentum to explain disorientation in the eclipse ide. In: *Proc. of the Visual Languages and Human-Centric Computing (VLHCC).* [S.l.: s.n.], 2006. p. 51–54. ISBN 0-7695-2586-5.

ANDA, B. Assessing software system maintainability using structural measures and expert assessments. In: *Proc. of the 23rd IEEE International Conference on Software Maintenance (ICSM).* [S.l.: s.n.], 2007. p. 204–213.

BASILI, V.; SHULL, F.; LANUBILE, F. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, v. 25, n. 4, p. 456–473, Jul 1999.

BASILI, V. R. The experimental paradigm in software engineering. In: *Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions.* [S.l.: s.n.], 1993. p. 3–12.

BASILI V.R., C. G. R. H. Goal question metric paradigm. In: *Encyclopedia of Software Engineering.* [S.l.: s.n.], 1994. p. 528–532.

BIAN, Y. et al. Spape: A semantic-preserving amorphous procedure extraction method for near-miss clones. *J. Syst. Softw.*, v. 86, n. 8, 2013.

BOYATZIS, R. E. *Transforming Qualitative Information: Thematic Analysis and Code Development.* [S.l.]: SAGE Publications, 1998.

BRERETON, P. et al. Lessons from applying the systematic literature review process within the software engineering domain. *J. Syst. Softw.*, Elsevier Science Inc., v. 80, n. 4, p. 571–583, 2007.

CARNEIRO, G.; MENDONçA, M. Sourceminer: Towards an extensible multi-perspective software visualization environment. In: HAMMOUDI, S. et al. (Ed.). *Enterprise Information Systems.* [S.l.]: Springer International Publishing, 2014, (Lecture Notes in Business Information Processing, v. 190). p. 242–263.

CARNEIRO, G. et al. Identifying code smells with multiple concern views. In: *Proc. of the 1th Brazilian Conference on Software: Theory and Practice (CBSOFT).* [S.l.: s.n.], 2010. p. 128–137.

CARNEIRO, G. F.; MENDONçA, M. G. Sourceminer - a multi-perspective software visualization environment. In: *Proc. of the 15th International Conference on Enterprise Information Systems (ICEIS)*. [S.l.: s.n.], 2013.

CARVER, J. et al. Issues in using students in empirical studies in software engineering education. In: *Proc. of the 9th International Software Metrics Symposium*. [S.l.: s.n.], 2003. p. 239–249.

CASSELL, C.; SYMON, G. *Qualitative methods in organizational research: a practical guide*. [S.l.]: Sage, 1994.

CHATZIGEORGIOU, A.; MANAKOS, A. Investigating the evolution of bad smells in object-oriented code. In: *Proc. of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC)*. [S.l.: s.n.], 2010. p. 106–115.

CHOI, E.; YOSHIDA, N.; INOUE, K. What kind of and how clones are refactored?: A case study of three oss projects. In: *Proceedings of the Fifth Workshop on Refactoring Tools (WRT 2012)*. [S.l.: s.n.], 2012. p. 1–7.

CHOI, E. et al. Extracting code clones for refactoring using combinations of clone metrics. In: *Proceedings of the 5th International Workshop on Software Clones*. [S.l.: s.n.], 2011. p. 7–13.

CLIFF, N. *Ordinal Methods for Behavioral Data Analysis*. [S.l.]: Erlbaum, 1996.

CRESWELL, J. *Educational Research: Planning, Conducting, and Evaluating Quantitative and Qualitative Research*. [S.l.]: Merrill, 2005.

CRUZES, D.; DYBÅ, T. Recommended steps for thematic synthesis in software engineering. In: *Proc. of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2011. p. 275–284.

CRUZES, D.; DYBÅ, T. Research synthesis in software engineering: A tertiary study. *Information and Software Technology*, v. 53, n. 5, p. 440 – 455, 2011.

D'AMBROS, M.; BACCHELLI, A.; LANZA, M. On the impact of design flaws on software defects. In: *Proc. of the 10th International Conference on Quality Software (QSIC)*. [S.l.: s.n.], 2010. p. 23–31.

DHAMBRI, K.; SAHRAOUI, H.; POULIN, P. Visual detection of design anomalies. In: *Proceeding of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*. [S.l.: s.n.], 2008. p. 279–283. ISSN 1534-5351.

DIESTE, O.; PADUA, A. G. Developing search strategies for detecting relevant experiments for systematic reviews. In: *Proc. of the 1th ESEM*. [S.l.: s.n.], 2007. p. 215–224.

DIXON-WOODS, M. et al. Integrative approaches to qualitative and quantitative evidence. *Health Development Agency*, p. 1–35, 2004.

DYBÅ, T.; DINGSØYR, T. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, v. 50, n. 9–10, p. 833 – 859, 2008.

DYBÅ, T.; SJØBERG, D. I.; CRUZES, D. S. What works for whom, where, when, and why?: On the role of context in empirical software engineering. In: *Proc. of the 6th International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2012. p. 19–28.

EASTERBROOK, S. et al. Selecting empirical methods for software engineering research. In: SHULL, F.; SINGER, J.; SJØBERG, D. (Ed.). *Guide to Advanced Empirical Software Engineering*. [S.l.]: Springer London, 2008. p. 285–311.

EMDEN, E. V.; MOONEN, L. Java quality assurance by detecting code smells. In: *Proc. of the 9th Working Conference on Reverse Engineering (WCRE)*. [S.l.: s.n.], 2002. p. 97–106.

FABBRI, S. et al. Managing literature reviews information through visualization. In: *Proc. of the 14th International Conference on Enterprise Information Systems (ICEIS*. [S.l.: s.n.], 2012. p. 36–45.

FEINSTEIN, A. R.; CICCHETTI, D. V. High agreement but low kappa: I. the problems of two paradoxes. *Journal of Clinical Epidemiology*, v. 43, n. 6, p. 543–549, 1990.

FINN, R. H. A note on estimating the reliability of categorical data. *Educational and Psychological Measurement*, v. 30, p. 71–76, 1970.

FLEISS, J. et al. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, v. 76, n. 5, p. 378–382, 1971.

FONTANA, F. et al. Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. In: *29th IEEE International Conference on Software Maintenance (ICSM)*. [S.l.: s.n.], 2013. p. 260–269.

FONTANA, F. A.; BRAIONE, P.; ZANONI, M. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, v. 11, n. 2, p. 5: 1–38, 2012.

FOWLER, M. *Refactoring: improving the design of existing code*. [S.l.: s.n.], 1999.

GRONBACK, C. Software remodeling: Improving design and implementation quality (using audits, metrics, and refactoring. In: *Borland Together ControlCenter, a Borland white paper*. [S.l.: s.n.], 2003.

GWET, K. Kappa statistic is not satisfactory for assessing the extent of agreement between raters. *Statistical Methods for Inter-rater Reliability Assessment*, n. 1, p. 1–5, 2002.

HERNANDES, E. M. et al. Using GQM and TAM to evaluate start - a tool that supports systematic review. *CLEI Electron. J.*, v. 15, n. 1, 2012.

HIGO, Y. et al. Method and implementation for investigating code clones in a software system. *Inf. Softw. Technol.*, v. 49, n. 9-10, 2007.

HöST, M.; REGNELL, B.; WOHLIN, C. Using students as subjects: A comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, Kluwer Academic Publishers, Hingham, MA, USA, v. 5, n. 3, p. 201–214, nov. 2000. ISSN 1382-3256.

HöST, M.; WOHLIN, C.; THELIN, T. Experimental context classification: incentives and experience of subjects. In: *Proc. of the 27th International Conference on Software Engineering (ICSE).* [S.l.: s.n.], 2005. p. 470–478.

JEDLITSCHKA, A.; CIOLKOWSKI, M.; PFAHL, D. Reporting experiments in software engineering. In: SHULL, F.; SINGER, J.; SJøBERG, D. I. K. (Ed.). *Guide to Advanced Empirical Software Engineering.* [S.l.]: Springer London, 2008. p. 201–228.

JOHNSON, B.; SHNEIDERMAN, B. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In: *Proc. of IEEE Conference on Visualization.* [S.l.: s.n.], 1991. p. 284–291.

JURISTO, N.; VEGAS, S. Using differences among replications of software engineering experiments to gain knowledge. In: *Proc. of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM).* [S.l.: s.n.], 2009.

KERSTEN, M.; MURPHY, G. C. Using task context to improve programmer productivity. In: *Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14).* [S.l.: s.n.], 2006. p. 1–11.

KHOMH, F.; PENTA, M. D.; GUÉHÉNEUC, Y.-G. An exploratory study of the impact of code smells on software change-proneness. In: *Proc. of the 16th Working Conference on Reverse Engineering (WCRE).* [S.l.: s.n.], 2009. p. 75–84.

KITCHENHAM, B.; CHARTERS, S. *Guidelines for performing systematic literature reviews in software engineering (version 2.3).* [S.l.], 2007.

KITCHENHAM, B. et al. Systematic literature reviews in software engineering - a tertiary study. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 52, n. 8, p. 792–805, 2010. ISSN 0950-5849.

KITCHENHAM, B. et al. Can we evaluate the quality of software engineering experiments? In: *Proc. of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* New York, NY, USA: ACM, 2010. p. 2:1–2:8.

KREIMER, J. Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science*, Elsevier Science Publishers B. V., v. 141, n. 4, p. 117–136, dez. 2005. ISSN 1571-0661.

LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. *Biometrics*, International Biometric Society, v. 33, n. 1, p. pp. 159–174, 1977. ISSN 0006341X.

LANZA, M.; DUCASSE, S. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, v. 29, n. 9, p. 782–795, 2003.

LANZA, M.; MARINESCU, R. *Object-Oriented Metrics in Practice.* [S.l.: s.n.], 2005.

LI, H.; THOMPSON, S. Clone detection and removal for erlang/otp within a refactoring environment. In: *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2009).* [S.l.: s.n.], 2009. p. 169–178.

LI, W.; SHATNAWI, R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, Elsevier Science Inc., New York, NY, USA, v. 80, n. 7, p. 1120–1128, jul. 2007.

LIN, Y. et al. Detecting differences across multiple instances of code clones. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014).* [S.l.: s.n.], 2014. p. 164–174.

MACIA, I. et al. On the relevance of code anomalies for identifying architecture degradation symptoms. In: *Proc. of the 16th European Conference on Software Maintenance and Reengineering (CSMR).* [S.l.: s.n.], 2012. p. 277–286.

MACIA, I.; GARCIA, A.; STAA, A. von. An exploratory study of code smells in evolving aspect-oriented systems. In: *Proc. of the 10th International Conference on Aspect-oriented Software Development (AOSD).* [S.l.: s.n.], 2011. p. 203–214.

MACIA, I. et al. On the impact of aspect-oriented code smells on architecture modularity: An exploratory study. In: *Proc. of the 5th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS).* [S.l.: s.n.], 2011. p. 41–50.

MACIA, I. et al. Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems. In: *Proc. of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD).* [S.l.: s.n.], 2012. p. 167–178.

MALETIC, H. K. J. I. Expressiveness and effectiveness of program comprehension: Thoughts on future research directions. In: *Frontiers of Software Maintenance (FoSM).* [S.l.: s.n.], 2008.

MÄNTYLÄ, M. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In: *Proc. of the 4th International Symposium on Empirical Software Engineering (ISESE).* [S.l.: s.n.], 2005.

MÄNTYLÄ, M.; LASSENIUS, C. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, v. 11, n. 3, p. 395–431, 2006.

MäNTYLä, M. V.; LASSENIUS, C. Drivers for software refactoring decisions. In: *Proc. of 5th International Symposium on Empirical Software Engineering (ISESE).* [S.l.: s.n.], 2006. p. 297–306.

MäNTYLä, M. V.; VANHANEN, J.; LASSENIUS, C. Bad smells humans as code critics. In: *Proc. of the 20th IEEE International Conference on Software Maintenance (ICSM).* [S.l.: s.n.], 2004. p. 399–408.

MARINESCU, R. *Measurement and Quality in ObjectOriented Design.* Tese (Doutorado) — Politehnica University of Timisoara, 2002.

MARINESCU, R. Detection strategies: Metrics-based rules for detecting design flaws. In: *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM).* [S.l.: s.n.], 2004. p. 350–359. ISBN 0-7695-2213-0.

MARINESCU, R.; MARINESCU, C. Are the clients of flawed classes (also) defect prone? In: *Proc. of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM).* [S.l.: s.n.], 2011. p. 65–74.

MAZZA, R. *Introduction to Information Visualization.* [S.l.]: Springer Publishing Company, Incorporated, 2009.

MENDONçA, M. et al. A framework for software engineering experimental replications. In: *Proc. of the 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*. [S.l.: s.n.], 2008. p. 203–212.

MEYER, B. *Object-Oriented Software Construction*. [S.l.: s.n.], 1988.

MOONEN, L.; YAMASHITA, A. Do code smells reflect important maintainability aspects? In: *Proc. of 28th the International Conference on Software Maintenance (ICSM)*. [S.l.: s.n.], 2012. p. 306–315.

MURPHY-HILL, E.; BLACK, A. P. An interactive ambient visualization for code smells. In: *Proc. of the 5th International Symposium on Software visualization (SOFTVIS)*. [S.l.: s.n.], 2010. p. 5–14.

NETO, P. A. da M. S. et al. A systematic mapping study of software product lines testing. *Information and Software Technology*, v. 53, n. 5, p. 407 – 423, 2011.

NGUYEN-DUC, A.; CRUZES, D. S.; CONRADI, R. The impact of global dispersion on coordination, team performance and software quality – a systematic literature review. *Information and Software Technology*, v. 57, n. 0, p. 277 – 294, 2015.

NOVAIS, R. et al. On the proactive and interactive visualization for feature evolution comprehension: An industrial investigation. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2012. p. 1044–1053.

OLBRICH, S. et al. The evolution and impact of code smells: A case study of two open source systems. In: *3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2009. p. 390–400.

OLBRICH, S. M.; CRUZES, D. S.; SJØBERG, D. I. K. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In: *Proc. of the 26th International Conference on Software Maintenance (ICSM)*. [S.l.: s.n.], 2010. p. 1–10.

PADILHA, J. et al. Detecting god methods with concern metrics: An exploratory study. In: *Proc. of the 7th Latin-American Workshop on Aspect-Oriented Software Development(LA-WASP), co-allocated with CBSoft*. [S.l.: s.n.], 2013.

PALOMBA, F. et al. Do they really smell bad? a study on developers' perception of bad code smells. In: *Proc. of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2014.

PARNIN, C.; GöRG, C.; NNADI, O. A catalogue of lightweight visualizations to support code smell inspection. In: *Proc. of the 4th international Smposium on Software visualization (SOFT-VIS)*. [S.l.: s.n.], 2008. p. 77–86.

PETERS, R.; ZAIDMAN, A. Evaluating the lifespan of code smells using software repository mining. In: *Proc.of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*. [S.l.: s.n.], 2012. p. 411–416.

PETERSEN, K. et al. Systematic mapping studies in software engineering. In: *Proc. of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. [S.l.: s.n.], 2008. p. 68–77.

POWERS, D. M. W. The problem with kappa. In: *Proc. of the 13th Conference of the European Chapter of the Association for Computational Linguistics (EACL).* [S.l.: s.n.], 2012. p. 345–355. ISBN 978-1-937284-19-0.

PUNTER, T. et al. Conducting on-line surveys in software engineering. In: *Proc. of the International Symposium on Empirical Software Engineering (ISESE).* [S.l.: s.n.], 2003.

RAHMAN, F.; BIRD, C.; DEVANBU, P. Clones: what is that smell? *Empirical Software Engineering*, Springer US, v. 17, n. 4-5, p. 503–530, 2012.

RAPU, D. et al. Using history information to improve design flaws detection. In: *Proc. of 8th European Conference on Software Maintenance and Reengineering (CSRM).* [S.l.: s.n.], 2004. p. 223–232.

REICHARDT, C. S.; COOK, T. D. *Beyond qualitative versus quantitative methods.* [S.l.]: Sage Publications, 1979.

RIEL, A. J. *Object-Oriented Design Heuristics.* 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

ROMANO, J. et al. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In: *In annual meeting of the Florida Association of Institutional Research.* [S.l.: s.n.], 2006. p. 1–3.

RUNESON, P.; HöST, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, Kluwer Academic Publishers, Hingham, MA, USA, v. 14, n. 2, p. 131–164, 2009. ISSN 1382-3256.

SALDANA, J. *The Coding Manual for Qualitative Researchers.* [S.l.]: SAGE Publications, 2012.

SANTOS, J. A.; MENDONçA, M. Identifying strategies on god class detection in two controlled experiments. In: *Proc. of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE).* [S.l.: s.n.], 2014. p. 244–249.

SANTOS, J. A.; MENDONçA, M. Exploring decision drivers on god class detection in three controlled experiments. In: *Accepted to be published in the Proc. of the 30th ACM/SIGAPP Symposium On Applied Computing (SAC).* [S.l.: s.n.], 2015. p. –.

SANTOS, J. A.; MENDONçA, M.; SILVA, C. An exploratory study to investigate the impact of conceptualization in god class detection. In: *Proc. of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE).* [S.l.: s.n.], 2013. p. 48–59.

SANTOS, J. A. et al. The problem of conceptualization in god class detection: agreement, strategies and decision drivers. *Journal of Software Engineering Research and Development (JSERD)*, v. 2, n. 11, p. 1–33, 2014.

SCHUMACHER, J. et al. Building empirical support for automated code smell detection. In: *Proc. of the 4th International Symposium on Empirical Software Engineering and Measurement (ESEM).* [S.l.: s.n.], 2010. p. 1–10.

SHULL, F. et al. Knowledge-sharing issues in experimental software engineering. *Empirical Software Engineering.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 9, n. 1-2, p. 111–137, mar. 2004.

SIMON, F.; STEINBRUCKNER, F.; LEWERENTZ, C. Metrics based refactoring. In: *Proc. of 5th European Conference on Software Maintenance and Reengineering (CSMR).* [S.l.: s.n.], 2001. p. 30–38.

SJØBERG, D. et al. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, v. 39, n. 8, p. 1144–1156, 2013.

SJØBERG, D. I. K.; DYBA, T.; JØRGENSEN, M. The future of empirical methods in software engineering research. In: *2007 Future of Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2007. (FOSE '07), p. 358–378. ISBN 0-7695-2829-5. Disponível em: ⟨http://dx.doi.org/10.1109/FOSE.2007.30⟩.

SJØBERG, D. I. K. et al. A survey of controlled experiments in software engineering. *IEEE Transaction on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 31, n. 9, p. 733–753, set. 2005. ISSN 0098-5589. Disponível em: ⟨http://dx.doi.org/10.1109/TSE.2005.97⟩.

SKOGLUND, M.; RUNESON, P. Reference-based search strategies in systematic reviews. In: *Proceedings of the 13th International Conference on Evaluation and Assessment in Software Engineering (EASE).* [S.l.: s.n.], 2009. p. 31–40.

THOMAS, J.; HARDEN, A. A systematic literature review to identify and classify software requirement errors. *BMC Medical Research Methodology*, v. 8, n. 45, 2008.

VAUCHE, S. et al. Tracking design smells: Lessons from a study of god classes. In: *Proc. of the 16th Working Conference on Reverse Engineering (WCRE).* [S.l.: s.n.], 2009. p. 145–154.

WHITEHURST, G. J. Interrater agreement for journal manuscript review. *American Psychologist*, v. 39, n. 1, p. 22–28, 1984.

WOHLIN, C. et al. *Experimentation in Software Engineering.* [S.l.]: Springer Berlin Heidelberg, 2012.

WOHLIN, C. et al. On the reliability of mapping studies in software engineering. *Journal of Systems and Software*, v. 86, n. 10, p. 2594 – 2610, 2013.

YAMASHITA, A. How good are code smells for evaluating software maintainability? results from a comparative case study. In: *29th IEEE International Conference on Software Maintenance (ICSM).* [S.l.: s.n.], 2013. p. 566–571. ISSN 1063-6773.

YAMASHITA, A.; COUNSELL, S. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software*, v. 86, n. 10, p. 2639 – 2653, 2013. ISSN 0164-1212.

YAMASHITA, A.; MOONEN, L. Do developers care about code smells? an exploratory survey. In: *20th Working Conference on Reverse Engineering (WCRE).* [S.l.: s.n.], 2013. p. 242–251.

YAMASHITA, A.; MOONEN, L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: *Proc. of the 35th International Conference on Software Engineering (ICSE).* [S.l.: s.n.], 2013. p. 682–691. ISBN 978-1-4673-3076-3.

YAMASHITA, A.; MOONEN, L. To what extent can maintenance problems be predicted by code smell detection? an empirical study. *Information and Software Technology*, v. 55, n. 12, p. 2223 – 2242, 2013.

YU, L.; RAMASWAMY, S. An empirical approach to evaluating dependency locality in hierarchically structured software systems. *Journal of Systems and Software*, v. 82, n. 3, 2009.

ZAZWORKA, N. et al. Investigating the impact of design debt on software quality. In: *Proc. of the 2Nd Workshop on Managing Technical Debt (MTD)*. [S.l.: s.n.], 2011. p. 17–23.

ZHANG, M.; HALL, T.; BADDOO, N. Code bad smells: A review of current knowledge. *Software Maintenance and Evolution: Research and Practice*, John Wiley & Sons, Inc., New York, NY, USA, v. 23, n. 3, p. 179–202, abr. 2011. ISSN 1532-060X.

ZIBRAN, M. F.; ROY, C. K. Ide-based real-time focused search for near-miss clones. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC 2012)*. [S.l.: s.n.], 2012. p. 1235–1242.