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. Sjöberg 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. Maia 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 paper 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 experiment1: (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)

---

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

<table>
<thead>
<tr>
<th>Software</th>
<th>Packages</th>
<th>Classes</th>
<th>NLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess</td>
<td>5</td>
<td>15</td>
<td>1426</td>
</tr>
<tr>
<td>Jackut</td>
<td>8</td>
<td>19</td>
<td>978</td>
</tr>
<tr>
<td>Tic</td>
<td>2</td>
<td>5</td>
<td>616</td>
</tr>
<tr>
<td>Monopoly</td>
<td>3</td>
<td>10</td>
<td>2682</td>
</tr>
<tr>
<td>Solitaire</td>
<td>6</td>
<td>23</td>
<td>1758</td>
</tr>
<tr>
<td>Tetris</td>
<td>4</td>
<td>16</td>
<td>993</td>
</tr>
</tbody>
</table>

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 plugins. 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

<table>
<thead>
<tr>
<th>Group</th>
<th>Programs</th>
<th>Participants of FinG 1 (ID)</th>
<th>Participants of FinG 2 (ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Chess, Jackut and Solitaire</td>
<td>14, 21, 32, 35</td>
<td>1, 3, 4, 7, 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>42 and 44</td>
<td>13, 15, 17, 19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>21, 23, 25 and 27</td>
</tr>
<tr>
<td>2</td>
<td>Monopoly, Tetris and Tic Tac Toe</td>
<td>13, 15, 25, 31 and 41</td>
<td>5, 6, 8, 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12, 14, 16, 18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20, 22 and 26</td>
</tr>
</tbody>
</table>

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
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
reading is used to identify context aspects about the classes. For example, in some cases, to identify if a method is in accordance 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 on 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 12 and FinG 23. 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


2 wiki.dcc.ufba.br/LES/FindingGdoClassExperiment2012
3 wiki.dcc.ufba.br/LES/FindingTwo
4 www.ines.org.br