Revealing Developers' Arguments on Validating the Incidence of Code Smells: A Focus Group Experience

Luis Felipi Junionello
luiz.junionello@aluno.cefet-rj.br
CEFET/RJ, Brazil

Rafael de Mello
rafael.mello@cefet-rj.br
CEFET/RJ, Brazil

Roberto Oliveira
roberto.oliveira@ueg.br
UEG, Brazil

Leonardo Sousa
leo.sousa@west.cmu.edu
Carnegie Mellon University

Alexander López
axl@certi.org.br
Fundação CERTI, Brazil

Alessandro Garcia
afgarcia@inf.puc-rio.br
PUC-Rio, Brazil

ABSTRACT

Identifying code smells is considered a subjective task. Unfortunately, current automated detection tools cannot deal with such subjectivity, requiring human validation. Developers tend to follow different, albeit complementary, strategies when validating the identified smells. Intending to find out developers' arguments when validating the incidence of code smells, we conducted a focus group session with developers familiar with identifying code smells. We distributed them among two groups, in which they had to argue about the incidence of a code smell: either accepting or rejecting its presence. Based on their arguments, we compiled a set of general heuristics that developers follow when validating smells. We then used these heuristics for composing validation items. We understand that the set of validation items proposed may support developers in reflecting on the incidence of code smells. However, further studies are needed for reaching a more comprehensive and optimized set. The experience of this study reveals that conducting focus group sessions is helpful to emerge the tacit knowledge of developers when validating code smells.

1 INTRODUCTION

The variety of maintenance requests over different source code elements frequently challenge software developers [1]. This challenge typically results from the structural complexity of the source code, requiring considerable reading and comprehension efforts from these professionals for performing even simple development tasks. For mitigating these efforts, one key practice address continuously identifying and combating the incidence of code smells [11]. Code smells are known as indicators of deeper problems within a source code, commonly introduced due to the negligence of good programming practices [5]. The incidence of code smells harms maintenance activities [11] [8] once it hampers the source code readability and comprehension. Besides, different works associate the incidence of code smells with the acceleration of the software degradation in the long term [13].

For supporting smell identification, several detection tools have been proposed [4][12]. However, even though these tools may save developers' effort on identification, they cannot be considered the final word [10]. Developers should manually validate candidates to code smells reported by detection tools, avoiding wasting effort on modifying several code elements due to false positives. Besides, this waste of effort may lead developers to accidentally introduce new and even worse issues in the source code [2].

Validating the incidence of code smells is a considerably subjective task. The developer's decision about their incidence may be highly influenced by contextual factors, including technological, organizational, and human ones [3] [2]. Consequently, the subjectivity of the task frequently leads developers to disagree on their interpretations about the incidence of code smells [6]. On the other hand, the diversity of perspectives followed by two or more professionals analysing the source code together—when possible—contributes to increasing the performance of smell identification tasks [3] [9]. However, allocating two or more developers for conducting group reviews may be unfeasible due to several reasons, including schedule and budget restrictions.

In this way, we argue that providing validation items, i.e., items for supporting validation activities, is a promising approach for empowering the developers' capacity to reflect on the incidence of code smells from different perspectives, especially when they need to work individually. These validation items can be grounded on heuristics, i.e., a set of particular attention points for enabling the developers' analysis and, consequently, supporting decision.

To reach a comprehensive and hands-on set of validation items, we plan to conduct a set of empirical studies to extract heuristics from the tacit arguments used by developers on validating the incidence of code smells. In this paper, we report a study using the focus group methodology [7], in which developers performed smell validation over code snippets from open-source projects. These tasks address the possible incidence of five different types of code smells. We intentionally allocated the study participants in groups that should play different roles for promoting in-depth discussions. While a group of developers should provide arguments for accepting the indicated smells, the other group should argue for rejection.

The results of the focus group session allowed us to identify several heuristics that go beyond formal detection rules. Based on these heuristics, we compiled a set of validation items for supporting developers in reflecting on the incidence of the different types of code smells investigated. To the best of our knowledge, our study is the first one using the focus group methodology [7] to investigate the manual validation of code smells. The positive experience of this study indicates that running focus group sessions may be a useful approach for exploring and revealing developers' arguments when performing this task.

2 THE FOCUS GROUP

Different developers may follow diverse but complimentary heuristics for concluding that some code element is poorly structured

or not [6]. However, they typically need to perform this analysis individually. Consequently, the arguments for accepting/rejecting a code smell tend to be limited to certain points of view. In this way, our research aims at *characterizing a relevant set of validation items for supporting developers on validating the incidence of code smells.*

2.1 Research Question

Based on our research goal, we defined the following research question: *Which arguments do developers use to validate the incidence of code smells?* By answering this research question, we want to characterize possible arguments used by these professionals when validating the incidence of code smells. From these arguments, we want to derive relevant heuristics for composing validation items to guide developers in future validation tasks. For this purpose, we designed a focus group session for promoting in-depth discussion about the incidence of code smells in five code snippets considered controversial about the incidence of code smells due to the low level of agreement observed among developers [6]. Focus group is a qualitative research method based on gathering data through the conduction of group interviews, called sessions [7]. Each focus group session is planned for addressing in-depth discussions about a particular topic during a controlled time slot. Among others, focus group studies have been conducted in software engineering for revealing arguments and feedback from practitioners [7] [14].

2.2 The Participants

We invited 12 Master and Doctorate students experienced with software development to participate in the study. Most of these professionals are also specialists in code smells. Before conducting the focus group session, we applied a characterization form. In this form, we asked about the subjects' experience from three distinct perspectives: *self-assessment*, *years of experience*, and *number of projects*. The self-assessment indicates that most participants have high or very high experience in software development (9/12) and Java programming (7/12). Besides, no participant declared having no development experience. Of the 12 participants, eight also declared having experience with code smells identification. From these, we identified that six also have experience with research in code smells. Table 1 summarizes the average experience of the participants.

Table 1: Average experience of the participants in the measured skills.

Metric	Software Dev.	Java	Smell Ident.
years	5.92	4.75	1.00
# of projects	9.42	7.17	3.08

The different perspectives used for characterizing the participants' experience led us to conclude that the sample investigated is experienced in building software for different Java projects. Besides, most of the participants are also skilled in the identification of code smells. In the characterization form, we also asked the participants to briefly summarize their experience in software development. Based on the answers provided, we observed that most of the developers have experience building systems in different programming languages for several domains.

Based on these characteristics, we composed two balanced groups with six developers each. According to the group assigned, the participant should play a particular role. With this approach, we intend to encourage the developers of each group to work together, reaching solid arguments to support their positions in an attempt to convince the other group. The first group defined was composed of *accepters*, i.e., developers that should favorably argue to the incidence of code smells in all code snippets analysed during the focus group. The second group is composed of *rejecters*, i.e., developers that should play the role of arguing for the rejection of all the code smells reported. The focus group session was conducted by four researchers. Besides the *moderator*, two researchers played the role of *scribes*, each one taking notes addressing the attitudes, behaviours, and reactions of a particular group. The fourth researcher played the role of *observer*, reporting attitudes, behaviours, and reactions expressed by all participants, including the other researchers.

2.3 The Code Snippets

The code snippets used in this study were selected among those ones identified in the study from [6] as having higher levels of disagreement among developers regarding the incidence of code smells from different types: Long Parameter List, Feature Envy, Middle Man, Primitive Obsession, and Refused Bequest[1]. Considering the nature of the projects involved in the study (large open-source projects), we checked that none of the study participants have previous knowledge of the analysed modules. It would lead developers to feel more comfortable on exploring different perspectives for providing their arguments, especially on those favorable to the incidence of poorly structured code. These code snippets were obtained from three popular open-source projects developed in the Java programming language: *GanttProject- version 2.0.10*, *Apache Xerces- version 2.11.0*, and *Eclipse- version 3.6.1*.

2.4 Instrumentation

Before starting the focus group session, the source code of the projects containing the snippets to be analyzed was shared among the participants. However, the code snippets/smells to be analysed were only presented during the focus group session. Each group should use one or more laptops to perform the tasks. The study was planned to be executed in two sequential phases. In the first phase, the moderator should distribute subjects in the room and present the dynamics of the activity. Then, the moderator should distribute pens and post-its for each group writing their arguments addressing each code snippet analysed.

The second phase is composed of tasks for validating the incidence of a particular code smell in each code snippet selected for the study, in the following order: long parameter list, feature envy, middle man, primitive obsession, and refused bequest. In each task, the following steps should be repeated:

(1) Moderator presents the code snippet, its location and the corresponding code smell.
(2) Moderator asks participants to discuss in their group if they accept/reject the smell, writing their arguments in post-its and fixing them in the board, as exemplified in Figure 1.

[1] https://anonymous.4open.science/r/FG-2426/README.md

(3) Moderator invites the accepters to present their arguments for accepting the incidence of the code smell.
(4) Moderator invites rejecters to argue on Accepters' opinion based on their annotations, starting the discussion.
(5) Moderator check whether the groups reach a consensus.

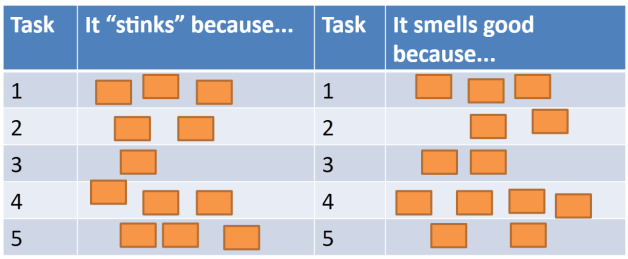

Figure 1: Representation of the board used in the focus group session with post-its fixed.

2.5 Execution

The focus group session was conducted in a classroom prepared for the study, without interruptions. We physically allocated the accepters and rejecters with their corresponding scribe in different corners of the room for mitigating the incidence of external noise during the internal discussions. Figure 2 presents an overview of the room during the meeting. During the study, all the planned tasks were performed. From the 12 subjects recruited, nine participated in the study. Despite this difference, the nine participants played the role previously planned for them, resulting in five rejecters and four accepters.

Figure 2: Overview of the room during the Focus Group Session.

The first phase of the study took 15 minutes. The time spent with the second phase (tasks) was approximately one hour and 30 minutes. This time was somehow balanced among the five tasks. In the end, the whole focus group meeting effectively took one hour and 45 minutes, extrapolating 15 minutes from the scheduled time. The groups were free to self-organizing during the internal discussions. The group of the rejecters opted by composing two subgroups, each one sharing a notebook containing the source code of each code snippet evaluated. The accepters opted by working together using a single notebook. As expected, the scribes and the observer took detailed notes addressing all the activities performed in the meeting using their own notebooks.

All the developers collaborated in all tasks. Leaderships naturally emerged for each group since the first task. The moderator acted on encouraging all the members to participate, avoiding concentrating the discussions too much in the leaders' arguments. During the four first tasks, the discussions were intense. In all of them, the moderator needed to alert the participants about the time planned. In the fifth task (Refused Bequest), we observed that most of the participants from both groups were exhausted, which had influenced their power of argumentation and the lack of discussions.

3 RESULTS

After the focus group session, the content of the notes taken by the scribes/observer was analyzed by two other researchers that identified the main arguments and behaviours observed in the groups. Based on this content, the first researcher performed open coding over the arguments identified. Then, he compiled a set of heuristics validated by the second one. The scribes/observer' notes and the complete list of heuristics are also available at the link about the study. Then, we used the heuristics for composing validation items for supporting the manual validation of the five types of code smells investigated. In the following subsections, we summarize the discussions performed by the groups in each validation task. We emphasize the arguments that emerged from each group. We also present the resulting validation items for each type of code smell. These items do not intend to indicate which decisions developers should take but stimulating them to reflect on relevant aspects of the source code.

3.1 Long Parameter List

3.1.1 Accepters. The group of accepters begun its internal discussions by assuming that one of the factors for arguing that the method analysed has a long parameter list is that this method *has six parameters*. Then, they had a brief discussion about the possibility of the method being a constructor, which would be another favorable argument for accepting the code smell. One participant highlighted that the method signature needs more than one line to be shown on the screen, which would indicate that the list of parameters is too long. This participant also argued that the list of parameters is composed of *too many complex data types*. After analysing the source code, another participant argued that the nature of the method's parameters leads the corresponding class to have *too many dependencies* with others. Besides, this participant inferred that *the parameters are not all necessary*.

3.1.2 Rejecters. After an initial reflection about the formal definition of long parameter list, one participant justified the absence of this code smell in the method analysed by arguing that *the parameters address different data types*. Besides, this participant also argued that the referred code element could not be classified as having a long parameter list once *it was a constructor*. Otherwise, it would be a code smell. After more analyses, the group concluded that *a list of five parameters cannot be considered long*.

3.1.3 Discussion. The group of rejecters started by presenting the following arguments: (i) the method's parameters have different data types; (ii) the method is a constructor; and (iii) having five parameters sounds acceptable. Immediately, the accepters exposed their strong disagreement by counter-arguing that (i) there are too many parameters, (ii) a line break is needed to list all parameters, and (iii) some of the parameters might not be needed. Then, one member of the accepters' group pointed out that rejecters miscounted the number of arguments (six instead of five). Besides, he argued that the rejecters' fail on counting was probably due to a long list of parameters that even required line break. We observed that this event somehow 'broke' the group of rejecters on their arguments, but they keep on discussing with the accepters. After some minutes, the group of rejecters had conceded to the accepters' arguments, agreeing with them that being a constructor is an additional issue to recognize a long parameter list.

3.1.4 Validation Items. Based on the arguments identified and heuristics coded, we composed the following validation items:

- Does the method signature have too many parameters?
- Is the method signature easy to comprehend?
- Does the method parameters indicate strong dependence from external classes?
- Are there too many parameters composed of complex types?
- Are all parameters actually needed?
- Do the parameters have different data types?

3.2 Feature Envy

3.2.1 Accepters. Initially, one member of the group exposed his doubts about the code smell definition, promptly resolved by his colleagues. After analysing the class, the members concluded that it *does not use its own resources* while *it consumes too many external resources*. These arguments were then considered sufficient by the accepters to convince the rejecters.

3.2.2 Rejecters. The group analyzed the class while discussing the definition of the feature envy. Once they observed an uncommon behaviour in the method, they tried to put themselves in the authors of the source code shoes to understand why they would build the class in that way. Then, they concluded that*the code authors had applied the composite design pattern*, arguing that the concept of feature envy could not be applied in this case. Another argument raised from the group to discard the feature envy was that *the method uses methods belonging to its parent class.*

3.2.3 Discussion. Different from the first task, in the second one the discussion between groups started dynamically since the first argument was uttered. The rejecters opened the discussions by arguing that it was an inherited method, belonging to the parent class. Immediately, the accepters showed their disagreement by arguing that the referred method doesn't pass any parameter to the method from the parent class, only using methods from its parent. Therefore, the accepters understand that this method should be implemented in the superclass instead. Besides, accepters also argued that one of the methods inherited from the parent class was being used over 20 times by the method analysed. Then, the rejecters counter-argued that this behavior is acceptable. By the end of the discussion, both groups were more agreeable to each other's arguments. However, we observed that they were confident in their arguments. Finally, the groups opted by keeping their positions.

3.2.4 Validation Items. Based on the arguments identified and heuristics coded, we composed the following validation items:

- Does the method call external methods too frequently?
- Does the class structure address some design pattern?
- Does the class use methods inherit from its parents?

3.3 Middle Man

3.3.1 Accepters. After analysing the class, the first issue brought up by the accepters was that the class in question *has a single private method, probably never used.* The particular characteristic of a single-method class leads a less-experienced participant to feel confused about the scope of Middle Man. After solving that, the group identified two other relevant issues favorable to the code smell: once the single method *is merely part of a request chain*, it consequently *does not seems useful for other classes.*

3.3.2 Rejecters. The group showed some difficulty in raising arguments for rejecting the code smell, but they eventually realized that the analysed class *has the clear role of serializing data.* Then, they also concluded that a *small class composed of a single smell* could not be considered a middle man.

3.3.3 Discussion. The rejecters opened the discussions by arguing that the class is easy to read once it has a single method playing a specific role, which is serializing data. In their counter-argument, the accepters started showing to the rejecters that there's already an interface designed for serializing data. Then, they emphasized that the aforementioned role makes the class' objects becoming part of a request chain, merely passing a single string as parameter. The arguments given by the accepters led rejecters to rethink about the definition of the code smell and taking a second look at the source code. After that, they recognized the pertinence of the accepters' arguments. However, they concluded that the class is required due the programming language. For them, alternative implementations would harm the system's performance and readability.

3.3.4 Validation Items. Based on the arguments identified and heuristics coded, we composed the following validation items:

- Does the class perform any relevant logical task?
- Does the class clearly delegate its responsibilities to other classes?
- Is this class part of a request chain?

3.4 Primitive Obsession

3.4.1 Accepters. For this Smell the group quickly brought up the argument that it had *too many primitive types* and they could be *replaced by a single enum* that consolidated all of them.

3.4.2 Rejecters. The first argument that came from the discussion was that the method *had the specific role of being a parser* and the primitive types were needed for that. Other strong argument that was brought up was that the presence of said primitive types *improved the method's comprehensibility.*

3.4.3 Discussion. The rejector's group opened the discussion by arguing that primitive types are needed due to the role of parser

identified in the method analysed. The acceptor group counter-argument that even if this parsing is necessary, it could be consolidated into a single enum type to be externally stored. Then, a rejector claimed that the proposed modification would result in other anomalies, which led the groups to discuss cohesion and coupling concerns. After reading the definition of primitive obsession, the accepters asked whether the rejecters had analysed the class as a whole. By the end, both groups stand by their initial positions.

3.4.4 Validation Items. Based on the arguments identified and heuristics coded, we composed the following validation items:

- Is it clear the intention of using primitive types in the method?
- Does the adoption of primitive variables contribute to the comprehension of the method?
- May two or more variables be consolidated into a single complex type?

3.5 Refused Bequest

3.5.1 Accepters. The accepters rapidly concluded that *the class use few resources from its parent class*, which would be a sufficient argument for accepting the smells.

3.5.2 Rejecters. At the beginning, some members exposed their opinion that refused bequest is a "boring" code smell type to analyse. Then, they also stated that they're exhausted. Besides that, they reached a consensus that the behaviour observed is acceptable in that case analysed due to the *principle of polymorphism*.

3.5.3 Discussion. Since the beginning of this task, we observed that members from both groups showed signals of exhaustion. However, they made efforts to provide valuable arguments. The rejecters combined the argument about polymorphism along with an analogy to defend their position. Besides, they also showed the source code documentation, claiming that the source code was intentionally implemented in that way. All these arguments convinced the accepters to reject the code smell, despite their understanding that few resources from the parent class are used.

3.5.4 Validation Items. Based on the arguments identified and heuristics coded, we composed the following validation items:

- Does the class inherit methods that are never used?
- Does the inheritance conceptually make sense?

4 THREATS TO VALIDITY

One relevant threat to validity addresses the influence of the code snippets' over the validation items composed. To mitigate it, we intentionally selected code perceived as hard on reaching a consensus in previous work. Besides, we understand that the pre-defined roles played by developers combined with their unfamiliarity with the source code enabled them to combine their knowledge with their creativity for providing convincing but also comprehensive sets of arguments supporting their positions.

Another important threat address the researchers' bias on composing the validation items. To mitigate this threat, we avoided involving the scribes in the first moment of the data analysis. In this way, the moderator of the focus group and an external researcher worked on the composition of the heuristics. Then, all authors collaborated for composing the validation items.

5 CONCLUSION AND FUTURE WORK

It is undeniable that developers should give the final word about the incidence of code smells. In this paper, we report our experience conducting a focus group session for exploring in-depth the arguments of developers for validating the incidence of code smells. Based on the heuristics obtained from these arguments, we composed sets of validation items for the types of code smell investigated. By these items, we do not intend to determine when a code is/is not smelly but instead enabling a more comprehensive reflection of developers before taking their final decisions.

This initial study indicates that running focus group sessions is an effective strategy for reaching our goal. However, we need to conduct future studies for validating and evolving the proposed set. In this sense, we are currently working on combining the findings from the presented focus group with those obtained from a controlled study in which developers individually argued about the incidence of code smells from a larger set of code snippets.

6 ACKNOWLEDGEMENTS

This work is supported by PIBIC-Cefet/RJ and CNPq 152179/2020-8.

REFERENCES

[1] Keith Bennett and Vaclav Rajlich. 2000. Software Maintenance and Evolution: A Roadmap. (2000).
[2] Rafael de Mello, Anderson Uchôa, Roberto Oliveira, Willian Oizumi, Jairo Souza, Kleyson Mendes, Daniel Oliveira, Baldoino Fonseca, and Alessandro Garcia. 2019. Do Research and Practice of Code Smell Identification Walk Together? A Social Representations Analysis. https://doi.org/10.1109/ESEM.2019.8870141
[3] Rafael Maiani de Mello, Roberto Oliveira, and Alessandro Garcia. 2017. On the influence of human factors for identifying code smells: A multi-trial empirical study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 68–77.
[4] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. 1–12.
[5] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.
[6] Mário Hozano, Alessandro Garcia, Baldoino Fonseca, and Evandro Costa. 2018. Are you smelling it? Investigating how similar developers detect code smells. *Information and Software Technology* 93 (2018), 130–146.
[7] Jyrki Kontio, Johanna Bragge, and Laura Lehtola. 2008. The focus group method as an empirical tool in software engineering. In *Guide to advanced empirical software engineering*. Springer, 93–116.
[8] Rodrigo Lima, Jairo Souza, Baldoino Fonseca, Leopoldo Teixeira, Rohit Gheyi, Márcio Ribeiro, Alessandro Garcia, and Rafael de Mello. 2020. Understanding and Detecting Harmful Code. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*. 223–232.
[9] Roberto Oliveira, Rafael de Mello, Eduardo Fernandes, Alessandro Garcia, and Carlos Lucena. 2020. Collaborative or individual identification of code smells? On the effectiveness of novice and professional developers. *Information and Software Technology* 120 (2020), 106242.
[10] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development* 5, 1 (2017), 1–28.
[11] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.
[12] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. 2019. Comparing heuristic and machine learning approaches for metric-based code smell detection. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 93–104.
[13] Jilles Van Gurp and Jan Bosch. 2002. Design erosion: problems and causes. *Journal of systems and software* 61, 2 (2002), 105–119.
[14] Uwe Van Heesch, Theo Theunissen, Olaf Zimmermann, and Uwe Zdun. 2017. Software specification and documentation in continuous software development: a focus group report. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. 1–13.