How developers deal with Code Smells: the case of the SourceMiner Evolution team

Alcemir Rodrigues Santos[1], **Mário André de Freitas Farias**[1,3], **Eduardo Santana de Almeida**[1,2], **Manoel Mendonça**[1,2], **Claudio Sant'Anna**[1,2]

[1]Federal University of Bahia (UFBA)
Software Engineering Laboratory (LES)

[2]Fraunhofer Project Center (FPC)

[3]Federal Institute of Sergipe (IFS)

{alcemirsantos, marioandre, esa, mendonca, santanna}@dcc.ufba.br

Abstract. *Context. Identify code smells has occupied researcher mind all over the world. Goal. In this paper we tackled this problem aiming to gather evidence on software metrics use and the developers refactoring intention due the presence of code smells. Method. We performed a controlled experiment with the team of developers of a medium sized software system. Results. Our results showed that although metrics can help developers, which metric is most useful seems to be a personal choice. In addition, code smells may only affect the refactoring decision of less experienced developers.*

1. Introduction

A challenge for software engineers and programmers is to assure quality and maintainability for complex and large software systems [Sjoberg et al. 2013]. Such task often implies problems regarding communication, compatibility, and complexity issues [Lanza et al. 2005]. Therefore, ensuring the maintainability of software systems is not a trivial task, and improving this process is a continuous work of research in the software maintainability area.

In general, while software evolves with a complex design, developers may introduce design problems (*e.g.*, code smells). Sometimes, these code smells can cause maintainability obstacles to be transposed on the pursuit of code quality goals, such as understandability and changeability [Fowler 1999]. In this sense, researchers had been tackling such problems from different perspectives, either through *(i)* defining automatic strategy for detecting code smells [Lanza et al. 2005, Marinescu 2004], *(ii)* trying to understand how developers identify code smells [Santos et al. 2013], *(iii)* understanding software to improve the maintainability based on code smells evaluations [Sjoberg et al. 2013], or *(iv)* pursuing better ways to identify smells with metrics supporting develepers evaluation [Guo et al. 2010].

In this context, we performed an empirical study to investigate how developers identify code smells and to discuss the effectiveness of automatic code smells detection tools, such as inCode and inFusion[1]. Section 2 describes the study settings. In this sense, we enumerate the research questions guiding our study as follows.

[1]Both inCode and inFusion are available at: `www.intooitus.com`. We use InCode (v3.8). It takes the source code and rely on Lanza *et al.* [Lanza et al. 2005] strategies of smells detection.

RQ$_1$ Do developers converge with the automatic code smells identification?
RQ$_2$ What strategies do developers use to find code smells in their software?
RQ$_3$ Does the code smells identification imply in the developers intention of refactoring?

In summary, we used these questions to build the contributions (see Section 3) of this paper as follows. We fostered the discussion *(i)* on the factors affecting the code smell identification, *(ii)* on the developers reliance of software metrics in this process; and *(iii)* on the drivers of the developers refactoring intention. We concluded the paper by discussing the threats to validity (Section 4) and by answering the research questions (Section 6).

2. Study Settings

In this section, we present the setting of our study. More specifically, we discuss *(i)* the target system analyzed; *(ii)* the characterization of the subjects; *(iii)* the controlled experiment design; and *(iv)* the code smells we addressed in the investigation.

2.1. SourceMiner Evolution

SourceMiner Evolution[2] (SME) [Novais et al. 2013] is a software visualization tool that provides support to software evolution tasks, aiming to ease software maintenance. It was primarily implemented in Java as an Eclipse plugin in more than `20` KLOC. We choose SME because of convenience, since we could easily access the developers and mainly because it is not a trivial system and represent wide range of software systems.

Table 1 characterizes the SME tool by presenting software metrics values addressing different quality attributes, such as complexity, coupling and inheritance. We chose these metrics because they are well known [Chidamber and Kemerer 1994, Lanza et al. 2005], easy to understand and depict a wide view of our target system.

Table 1. Essential metrics values of SourceMiner Evolution.

Quality Attribute	Metric Acronym	Metric Name	Value
Size	NOP	Number of Packages	`36`
	LOC	Number of Lines of Code	`23.878`
	NOC	Number of Classes	`296`
	NOM	Number of Methods	`1.730`

2.2. Subjects Characterization

We called the seven SME developers to participate voluntarily. One developer did not answer the call and another could not attend due to lack of schedule, all others attended. Each developer worked on different parts of SME with eventual intersections. Those who attended the call have different formal education: a Ph.D. student (Dev2); a M.Sc. student (Dev1); and three undergraduate students (Devs 3, 4, and 5). Dev2 leads the project and advised the three undergraduate students during their work. We refer to them as DevX ($X = [1..5]$) for now on.

2.3. Experimental Design

We designed our (quasi-)experiment following Wohlin [Wohlin et al. 2012] guidelines. First, we carried out a pilot study and then the experiment itself. The pilot study was

[2]Software developed at Software Engineering Lab (LES): `www.les.dcc.ufba.br`

carried out with two master students which did not know the SME source code in advance. They had different performances leading the pilot study to inconclusive results. The pilot study took too long so that we calibrate the experiment task by reducing the number of code smells instances analyzed by the participants. The study included two code smells: `God Class` (when a class tends to concentrate functionality from several unrelated classes, while at the same time increasing coupling in the system) and `Data Class` (when the class is a "dumb" data holders, without complex functionality, but which are usually heavily relied upon by other classes in the system) [Lanza et al. 2005]. These smells are easy to understand and wide used in the literature.

In the experiment itself, we split the subjects into *"trained"* and *"under-training"* developers relying on the characterization form data. We considered trained those developers with a long-term training in code smells and with a large software industry experience working on projects performing different roles, such as Quality Assurance Analyst, Project Manager, Requirements Analyst (Devs 1 and 2). The under-training developers are still looking forward to achieve their Computer Science degree and started to program back in high school (Devs 3, 4 and 5). These developers reported fewer than two years on average of experience with development and differently from the trained developers, they reported few knowledge about code smells and refactoring.

For the experiment we used inCode to provide classes with smells. We chose this tool because it provide better support among the tools analyzed. Next, we elaborated the list of classes (10 classes) submitted to the developers analysis by putting together classes with and without code smells. We randomly selected the classes whithout any smell. The task assigned to the developers was the inspection of SME and the fulfillment of the questionnaire (discussed in details in the next Section) accordingly.

Additionally, we provided printed support material to all of them before the execution of the experiment: the definition of the code smells addressed and the metrics available to their use. We allowed them to keep the material during the experiment. During the experiment, we provided the metrics values calculated by inCode and make clear that its use were optional. We did not performed a formal training to avoid bias [Santos et al. 2013]. The experiment itself consists of accomplish the assigned task of fulfill an questionnaire about code smells in SME, which we describe next.

2.4. Applied Questionnaire

We considered different aspects to build the questionnaire used in our experiment aiming to gather useful information to answer our research questions. More specifically, we take into consideration: *(i)* the agreement with the inCode smells identification for each class; *(ii)* the agreement between the developers groups regarding their own identification of code smells; *(iii)* their intention to refactor the classes they pinpointed as containing code smells; *(iv)* the grading of the severity of the code smells, accordingly with their own criteria to match with the inCode values; *(v)* the developers opinion if metrics helped them to perform the smells identification. To support replication, we provide the questionnaire in http://goo.gl/TkDPUd.

The applied questionnaire consists of nine questions. We enumerated the questions as follows. *(Q1)* Does this class contain this smell? *(Q2)* How much does this class seem a smell to you? *(Q3)* Would you refactor these classes? *(Q4)* If answered Yes in Q3,

grade this class according to the emergency of refactoring (order as you would refactor). *(Q5)* Did you work in this class? *(Q6)* What drove you to your answer in `Q1`? *(Q7)* Did metrics help you to judge the classification of each class? *(Q7.a)* Which metrics set was useful to judge "Data Class"? *(Q7.b)* Which metrics set was useful to judge "God Class"?

3. Results

In this section we *(i)* describe the collected data and *(ii)* analyze the gathered information through different viewpoints.

3.1. Collected Data

Table 2 shows developers judgement about the presence or absence of the code smells addressed in our controlled experiment. For each developer there is three columns relating developers and the classes analyzed: `GC`, `DC` and `W`. The GC and DC columns show whether the developer identified the `God Class` and `Data Class` code smells for each given class or not. The severity of the code smell setted by the developer fulfills each cell. The W column shows whether the developer worked in that class during the SME development or not. For example, Dev2 indicated the presence of God Class and Data Class in the `ProjectStatistic` class with severities equals to one and ten, respectively, and did not worked in the class. Dev5 and Dev4 did not point out the severity of some code smells they identified during the experiment, which we represent with a hyphen `(-)`. The column `O` presents an oracle regarding the automatic analysis provided by the inCode tool.

Additionally, Table 2 presents *precision (p)*, *recall (r)* and f_1*-score* of the developers' judgements. Such metrics rely on the values of "true positives (tp)" — when the code smell assigned by the participant was also pointed out by inCode —, "false positives (fp)" — when the code smell assigned by the participant was not pointed out by inCode —, and "false negative (fn)" — when inCode pointed out the code smell to the class but the participant did not. In fact, while *precision* indicates the fraction of code smells correctly identified ($tp/(tp+fp)$), *recall* indicates the fraction of the existing occurrences identified ($tp/(tp+fn)$). Moreover, the f_1-score is an harmonic mean between precision and recall ($2*p*r/(p+r)$), *i.e.*, as much as they get higher, the F_1-score increases itself.

Table 2. Developers judgement on the code smells addressed and its respective values of precision, recall and accuracy.

Classes	O	Dev1			Dev2			Dev3			Dev4			Dev5		
		GC	DC	W	GC	DC	W	GC	DC	W	GC	DC	W	GC	DC	W
TreeMapView	GC	(10)		✔	(7)		✔	(10)		✔	(8)		✔	(-)		✔
PackagesFilter	DC			✔					(7)			(9)		(-)	(-)	
TreeMapModel	GC			✔	(10)		✔	(8)		✔	(9)		✔	(-)		✔
ConcernDiffModel					(2)					✔						
CouplingLayoutView							✔	(5)		✔						✔
ConcernFilterView	GC	(10)			(7)			(5)					✔			
CouplingModel	GC				(7)		✔	(10)		✔	(-)		✔	(10)		✔
ProjectStatistic	DC				(1)	(10)			(10)			(10)			(10)	
PolimetricLayoutView					(2)		✔			✔			✔			✔
InterfacesFilter	DC								(7)			(9)			(10)	
Precision		1	#		.57	1		.80	1		1	1		.75	1	
Recall		.50	0		1	.33		1	1		.75	1		.75	1	
F_1-score		.67	#		.73	.50		.89	1		.86	1		.75	1	

O: inCode oracle; **GC:** God Class; **DC:** Data Class; **W:** Worked?

Table 3(a) shows another excerpt of the collected data: the developers intention to refactor the classes analyzed in the experiment. The checked cells indicate that the

developer intends to refactor the class. The numbers inside the parentheses depict the ordering of refactoring assigned by the developer. The (-) means that the Dev2 did not inform the refactoring sequence. The questionnaire results for all questions is available in http://goo.gl/TkDPUd.

Table 3. (a): Developers refactoring intention and (b): Metrics use ocurrences.

(a)

Classes	Dev1	Dev2	Dev3	Dev4	Dev5
TreeMapView	✔(1)	✔(-)	✔(1)	✔(4)	✔(1)
PackagesFilter				✔(1)	✔(4)
TreeMapModel		✔(-)	✔(3)	✔(3)	✔(2)
ConcernDiffModel		✔(-)			
CouplingLayoutView					
ConcernFilterView	✔(2)	✔(-)	✔(4)		
CouplingModel		✔(-)	✔(2)	✔(5)	✔(3)
ProjectStatistic			✔(5)	✔(6)	✔(5)
PolimetricLayoutView					
InterfacesFilter				✔(2)	✔(6)

(b)

	GC	DC
LOC	3	-
WOC	2	1
CW	1	2
NOACCM	-	4
NOPUBA	-	1
CPFD	2	-
TCC	3	1

GC: God Class; **DC:** Data Class.

3.2. Data Analysis

The collected data built a rich set of information, from which we present a rationale through this section. For instance, we discuss the data taking into consideration *(i)* the profile and previous experience of the developers, *(ii)* the developers self-evaluation, and *(iii)* the developers refactoring intention.

3.2.1. Developers Profile and Experience

The values of F_1-score (Table 2) show that the under-training developers achieved closer results to the oracle than the other group. In other words, developers with lower formal education level and less industry experience obtained more correct answers according to the oracle (inCode). Similar experience produced similar beliefs [Passos et al. 2013], which might influenced on the similar judgement. Thus, both groups might have developed different beliefs from their work and academic environment, which may have positively influenced the judgement of the under-training developers. We also believe that, in contrast with the developers of the under-training group, the trained developers have different industry and academia experiences and such difference contributed to different beliefs, which might have increased the disagreement inside the latter group.

Such different beliefs contributed to the heterogeneous strategies used by the developers to identify the code smells. While, some rely on the use of metrics, others prefer code inspection, and there were also those who to mix both approaches. We found that most developers resort on metrics, Dev1 was the exception. He is the most experient and used source code inspection and his experience to assess code smells. In contrast, all the others perform an intensive use of metrics. Developers 3, 4 and 5 pointed out the metrics LOC and TCC as important to identifying God Class and developers 3 e 4 pointed out CW as important to identify Data Class. Such metrics were the most cited among the developers. Despite developers reliance on the software metrics, Table 3(b) shows that the choice of the metrics is personal, since developers reported different sets of metrics to assess each code smell. The metrics reported were ”Lines of Code (LOC)”, ”Weighted Operation Count (WOC)”, ”Class Weight (CW)”, ”Number of Accessor Methods (NOACCM)”, ”Number of Public Attributes (NOPUBA)”, Capsules Providing Foreign Data (CPFD), and ”Tight Capsule Cohesion (TCC)”.

3.2.2. Developers Self-Evaluation

Another finding in the experiment was about the self-evaluation. We identified that even though the trained developers said they had a good level of knowledge about smells, the under-training developers – whose claim otherwise – achieved better results. One possible explanation to their lack of confidence is their low level of formal education.

By investigating the developers characterization data collected we reached a consensus that under-training developers resort to the use of metrics to reinforce his judgement about the code smells. The fact of the under-training developers lack of confidence in their knowledge in code smells might have lead them to rely his judgement on metrics. Moreover, such behavior guide them to higher agreement with the tool.

3.2.3. Refactoring Intention

Regarding the refactoring intention, we proceeded with individual analysis for each developer as follows. **Dev1** intended to refactor only the classes pointed out as containing code smells, which might imply that he only considered code smell an anomaly that may be harmful to the system. Differently from Dev1, **Dev2** intended to refactor mainly the classes he identified a code smell which he attributed high severity. The only exception was the `ConcernDiffModel` class, which he justifies its refactoring because he knows the class presented too many problems in its evolution history.

Dev3 only intended to refactor the Data Classes with higher severity and all God classes, excepting `CouplingLayoutView`. Although he assigned the same severity for the god classes `CouplingLayoutView` and `ConcernFilterView`, he gave no explanation why he would refactor only one of them. In fact, none of the developers intend to refactor `CouplingLayoutView`. **Dev4** decided to refactor all the classes identified either with "God Class" or "Data Class". **Dev5** also intended to refactor all classes that contain code smells.

It seems that code smells do not drive developers with higher experience to refactoring. Instead, they would apply refactoring in the pieces of code with defect reincidence whether it contain a code smell or not. On the other hand, less experienced developers seem clearly guided by the code smells identification. In fact, their lack of experience might lead them to hold their judgement based on the existence of code smells instead of actual defects or flaws.

4. Threats to Validity

Despite all the caution in controlled (quasi-)experiments, they still present threats to validity [Wohlin et al. 2012]. In this section we discuss those we identified during our study as follows.

Conclusion validity. We believe the questionnaire was properly built to achieve the expected answers to our research questions. For instance, it allow us to detect the reliance of the unexperienced developers on the use of software metrics as support for the classes assessment. Therefore, we tried to bypass such threat to conclusion validity relying our analysis only in the information gathered with the subject's answers.

Construct validity. Regarding the experiment planning we made an effort to minimize communication among the subjects aiming to reduce the interference among their answers. In addition, we tried to include all the developers of the system analysed in order to get a wide analysis of the system regarding the code smells from different perspectives. Moreover, we clearly explained the process and introduced the support material – available during the experiment execution – to avoid misguidance of the subjects. In fact, we elaborated a simple and direct questionnaire to ease the understanding of the assigned task.

Internal validity. We only discuss data gathered with the questionnaire to assure a causal relationship between the treatment and the result. Otherwise, other factors – uncontrolled and unmeasured – may influence in the analysis. Therefore, to avoid influence from external factors (*e.g.* participants comunication) we choose to carry the questionnaire as a controlled (quasi-)experiment. Such decision allows us to draw more confident conclusions, despite the fact we interviewed a small sample of developers.

External validity. Our study interviewed a small sample of developers what is the biggest threat to the external validity. Therefore, we can not generalize our conclusions. However, in the same time, the reinforced internal validity allows us to draw insights to guide further investigations.

5. Related Work

Different work have already tackled the problem of code smells identification at source code level [Lanza et al. 2005, Katzmarski and Koschke 2012, Santos et al. 2013]. For instance, Lanza *et al.* [Lanza et al. 2005] proposed strategies to help the automation of such identification. Their strategies were implemented into the inCode tool, which performs automatic identification of code smells. Katzmarski and Koschke [Katzmarski and Koschke 2012] try to measure the agreement between developers and different metrics. More specifically, the question addressed is whether program aspects measured by software metrics can be used to determine program complexity from the perspective of programmers. Santos *et.al.* [Santos et al. 2013] carried a controlled experiment study on detection of the `God Class` code smell. They concluded that subjects take different *drivers* to judge a whether a class is a `God Class` or not. Differently our study focus on the agreement between the developers and the automatic detection with a tool support.

6. Final Remarks and Future Work

We are aware of the threats to validity and we answer the research questions to foster the discussion on the topic rather than make a generalizable claim. With regard to RQ_1 we found by the values of precision and recall that "under-trainning" developers judgement seems more likely to agree with the inCode automatic judgement. Instead of adopt a systematic approach, it was the "experience" and "knowledge confidence on software design and code harmfulness" that guided the "trained" developers. Such different factors produce different strategies of inspection for each developer. Therefore, we prefer to suggest further investigation concerning RQ_2 since our study lack of evidence on this matter, and whichever claim would be only researchers opinions. In addition, we noticed that the presence of code smells only affect the less experienced developers refactoring intention (RQ_3).

By taking into consideration the evidence gathered we can draw some research opportunities on the topic: *(i)* what factors influence the beliefs of developers? *(ii)* what factors drive developers to refactoring? *(iii)* how much the developers' previous knowledge in the source code and their experience can help them to identify smells? *(iv)* how researcher can discuss in-depth about developers' beliefs on whether a code smell exists or not; *(v)* replicate the experiment with a larger sample and other systems. Besides that, there is also a need to discuss whether code smells reported are really a design problem in the developers' point of view.

Acknowledgemnts: This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq, grants 573964/2008-4, Universal Project (grants 486662/2013-6) and is made possible by the Cooperation Agreement #1/2012 between the Science, Technology and Innovation Secretariat of the State of Bahia, the Fraunhofer-Gesellschaft and the State of Bahia.

References

[Chidamber and Kemerer 1994] Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *Transactions of Software Engineering*, pages 476–493.

[Fowler 1999] Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Guo et al. 2010] Guo, Y., Seaman, C., Zazworka, N., and Shull, F. (2010). Domain-specific tailoring of code smells: An empirical study. In *Proc. of the 32nd Int'l. Conf. on Software Engineering - Vol.2*. ACM.

[Katzmarski and Koschke 2012] Katzmarski, B. and Koschke, R. (2012). Program complexity metrics and programmer opinions. In *Proc. of the 20th Int'l. Conf. on Program Comprehension*, pages 17–26.

[Lanza et al. 2005] Lanza, M., Marinescu, R., and Ducasse, S. (2005). *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Marinescu 2004] Marinescu, R. (2004). Detection strategies: metrics-based rules for detecting design flaws. In *Proc. of 20th IEEE Int'l. Conf. on Software Maintenance*, pages 350–359.

[Novais et al. 2013] Novais, R., Nunes, C., Garcia, A., and Mendonça, M. G. (2013). Sourceminer evolution: A tool for supporting feature evolution comprehension. In *Proc. of 29th IEEE Int'l. Conf. on Software Maintenance*, pages 1–4.

[Passos et al. 2013] Passos, M. C. M., Cruzes, D. S., Hayne, A., and Mendonça, M. G. (2013). Recommendations to the adoption of new software practices: A case study of team intention and behavior in three software companies. In *Proc. of the Int'l. Symp. on Empirical Software Engineering and Measurement*, pages 1–10, Baltimore, USA.

[Santos et al. 2013] Santos, J. A. M., de Mendonça, M. G., and Silva, C. V. A. (2013). An exploratory study to investigate the impact of conceptualization in god class detection. In *Proc. of the 17th Int'l. Conf. on Evaluation and Assessment in Software Engineering*, pages 48–59.

[Sjoberg et al. 2013] Sjoberg, D., Yamashita, A., Anda, B., Mockus, A., and Dyba, T. (2013). Quantifying the effect of code smells on maintenance effort. *Transactions on Software Engineering*, 39(8):1144–1156.

[Wohlin et al. 2012] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer Berlin Heidelberg.