An Exploratory Study on the Influence of Developers in Code Smell Introduction

Leandro Alves, Ricardo Choren, Eduardo Alves
Military Institute of Engineering - IME
Computer Science's Departament
RJ, Brazil
Email: leansousa@gmail.com, choren@ime.eb.br, eduaopec@yahoo.com.br

***Abstract*—A code smell is any symptom in the source code that possibly indicates a deeper maintainability problem. Code smell introduction is a creative task - developers unintentionally introduce code smells in their programs. In this study, we try to obtain a deeper understanding on the relationship between developers and code smell introduction on a software. We analyzed instances of code smells previously reported in the literature and our study involved over 6000 commits of 5 open source object-oriented systems. First, we analyzed the distributions of developers using specific characteristics to classify the developers into groups. Then, we investigated the relationships between types of developers and code smells. The outcome of our evaluation suggests that the way a developer participates in the project may be associated with code smell introduction.**

Keywords–Code smells; exploratory study; software development and maintenance; development teams

I. INTRODUCTION

Software development is a complex activity that does not end even when the software is delivered. Usually, a software needs to be modified to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment [1]. However, continuous change can degrade the system maintainability. The degree of maintainability of a software system can be defined as the degree of ease that the software can be understood, adjusted, adapted, and evolved, and comprises aspects that influence the effort required to implement changes, perform modifications and removal of defects [2]. There are several issues that decrease the maintainability of a software system, such as problems with design principles, lack of traceability between analysis and design documentation, source code without comments and code smells.

Code smells are characteristics of the software that may indicate a code or design problem that can make software hard to evolve and maintain [3]. For instance, the more parameters a method has, the more complex it is. It would be desirable to limit the number of parameters you need in a given method, or use an object to combine the parameters. The presence of code smells indicates that there are issues with code quality, such as understandability and changeability, which can lead to maintainability problems [4].

The code quality depends on how good the developers are. However, there is little knowledge about the influence of developers on the introduction of code smells in a software system. Previous work focus on code smell detection and removal [5][6] and other studies focus on the awareness about code smells on the developer's side [4][6]. The challenge is to further understand the relationship between developers and code smell introduction. As a result, software managers have little knowledge on how the development team affects the software maintainability.

There are still some questions regarding the interplay between developers and the existence of code smells in a source code. Can the way how a developer Works in a Project, be used to understand the frequency of some code smell introduction in a source code? What types of code smells a developer is more likely to introduce? Understanding these issues may help developers to improve their skills and to build team culture with the purpose of avoiding code smells.

This paper presents a study to assess the influence of developers in code smell introduction in software code. Our investigation focused on the study of five software maintenance projects. The projects were selected because of the following characteristics: they were open source projects; information about them were available in a Git repository [7]; they had a substantial number of commits (over a seven hundred each); and they were developed using an object oriented programming language (Java).

This paper is structured as follows: Section 2 presents the concepts related to Code Smells and the classification of the developers. Section 3 describes a proposed method to sort the developers in groups and assess the contribution on the variation of Code Smells in the source code of the software. Section 4 demonstrates a case study for the application of the method of classification of developers, evaluating the influence of each developer group in variation of *Code Smells*. Section 5 describes related work and finally, Section 6 presents the conclusion of this article.

II. STUDY PRELIMINARIES

This Section presents the definitions of code smells and of developer characteristics used in our study.

A. Code Smells

Webster [8] defined *antipatterns* in object-oriented development. An antipattern is similar to a pattern except it is an obvious but wrong solution to a problem. Nevertheless, these antipatterns will be tried again by someone simply because they appear to be the right solution [9]. *Code smells* refer to structural characteristics of a source code that indicate this code has problems, affecting directly on the maintainability of the software and resulting in a greater effort to carry out developments in this source code [10].

 ISBN: 978-1-61208-438-1

B. Developer Characteristics

Software development is a human activity [11]. Understanding the human factors of the developers allows software managers to organize them in groups, so that they can compose more efficient teams [12]. Whereas distinguishing and verifying the impacts of each developer individually is a very difficult task, developers can be categorized according to their involvement in a software project. The involvement of a developer can be measured in terms of level of participation and degree of authorship on the source code [13].

The level of participation is related to the developer's involvement in the project and can be used, for example, to determine the degree of decision-making the developer has in the project team, allowing discover developers who exercise leadership in project [13][14]. The degree of authorship indicates the usual tasks the developer performs when acting on the software source code. It involves line code change, insertion or removal and file (e.g., class in an object-oriented system) insertion and removal.

III. Study Settings

The goal of our study is to investigate the influence of developers on the introduction of code smells in a software code. To do so, we analyzed the sequence of commits done in the repository of five different software projects. Merge (branches) were considered in the selection of the project commits.

First, we categorized the developers in different groups according to their characteristics in the project (participation and authorship). Then, for each commit, we searched for code smells in the source code. The quality focus was the analysis on the variation of the number of code smells along the time.

To categorize the developers, we used the k-means clustering algorithm [15]. The information used in the k-means algorithm was taken from the software repository and they were related to the participation level and degree of authorship of the developers in each selected project.

To find code smells in the source code, we used PMD [16], a static rule-set based Java source code analyser that seeks to evaluate aspects related to good programming practices.

A. System Characteristics

The first decision we made in our study was the selection of the target systems. We chose five medium-size systems. The first one, called Behave, is an automation tool for functional testing. It was first versioned in 2013 and we found 724 commits in its project. We selected 373 commits of Behave in our study. The second was JUnit, a unit-testing framework for the Java programming language. It was first versioned in 2000 and we found 1885 commits in its project. We selected 1203 commits of Junit in our study. The third one was Mockito, an open source-testing framework for Java, which allows the creation of test double objects (mock objects) in automated unit tests for the purpose of Test-driven Development or Behavior Driven Development. It was first versioned in 2007 and we found 1993 commits in its project. We selected 1561 commits of Mockito in our study. The fourth one, called RxJava, is a library for composing asynchronous and event-based programs using observable sequences for the Java VM. It was first versioned in 2012 and we found 2939 commits in its project. We selected 906 commits of RxJava in our study. The last system was VRaptor, a Java MVC Framework focused in delivering high productivity to web developers. It was first versioned in 2009 and we found 3385 commits in its project. We selected 2243 commits of VRaptor in our study. The projects selected for this study were taken from the Git repository on June 2014.

These systems were chosen because they met a number of relevant criteria for our study. First, these systems encompass a rich set of code smells (e.g., Dead Code, Long Method, Unhandled Exception). Second, they are non-trivial systems and their sizes are manageable for an analysis of code smells. Third, each one of them were implemented by more than 50 programmers with different levels of participation (the selected systems were all open source projects). Last, they have a significant lifetime, comprising of several commits. The availability of multiple commits allowed us to observe the introduction of code smell throughout their long-term development and evolution.

It should be noted that for this study, commits were discarded that altered documentation of source code, HTML pages and templates (css, imagens, javascript) changes because they have no relation with change of code smells.

B. Study Phases

Our study was based on the analysis of the developers' information and the systems' code smells. The main phases of our study are described next.

Recovering the Developers' Information. In this phase, we focused in gathering information about the level of participation and degree of authorship of a developers. The reason was that we needed to group the developers so that we could rely on general coding behaviour instead of trying to focus in each developer separately. We selected information from the data available in the Git repository. As a result, we concentrated on the analysis of information for each developer commit. For level of participation, we collect date and time of commit initial, date and time of last commit and interval (days) between commits. For degree of authorship, we collect amount of modified files (classes) in the commit (insertion, modification and deletion) and the amount of lines of code modified during the commit (insertion and deletion).

Classifying the Developers. The recovered information was used in the k-means clustering algorithm to identify groups of developers with similar characteristics, according to their participation and degree of authorship in the project. In this study, we used the k-means algorithm varying the value of k from four up to nine in order to verify the distribution of developers in the clusters.

The overall results provided six sets of developer clusters. Analysing these sets, we decided to use the results from k=5 (five clusters) because we wanted to avoid the presence of very scarce clusters. Then we used the apriori association algorithm to find correlations between different attributes in each cluster. The results identified the general association rules for the population of each cluster, as shown in Table I.

(a) *Group 01:* less frequent participation and line code deletion as general authorship behaviour;

(b) *Group 02:* less frequent participation and line code insertion and deletion as general authorship behaviour;

(c) *Group 03:* less frequent participation and file insertion, modification and deletion as general authorship behaviour;

(d) *Group 04:* more frequent participation and no particular general authorship behaviour (i.e., it performs all behaviours almost evenly);

(e) *Group 05:* more frequent participation and file insertion as general authorship behaviour.

TABLE I. POPULATION OF EACH CLUSTER

Gr.	Behave	JUnit	Mokito	RxJava	VRaptor
01	2	15	5	2	24
02	7	27	25	31	10
03	10	33	58	13	31
04	2	14	17	17	15
05	15	6	5	10	13

Selection of Code Smells. In this phase, we focused on selecting code smells that were previouly described in the literature. Moreover, we have not considered creating specific PMD rulesets to identify code smells. The reason was that we needed to rely on code smells that could be precisely identified in a systematic fashion, without any specialist assistance. As a result, we concentrated on the analysis of five existing code smells [17], which covered various anomalies related to object oriented programming. Those were: Dead Code (DC); Large Class (LC); Long Method (LM); Long Parameter List (LPL); and, Unhandled Exception (UE).

Identifying Occurrences of Code Smells. Code smells were identified using the PMD tool. Thus, code smells were detected using five ready-to-run PMD rulesets. We decided not to define specific rules for this study because we understand that code smells should be identified as simply as possible.

Analysis of Code Smell Introduction. The goal of the fifth phase was to analyse the behaviour of code smell introduction for the selected projects. The analysis aimed at triggering some insights for helping maintainers to understand the relationships between code smell introduction and the developers in the project team. To support the data analysis, the assessment phase was decomposed in three main stages. The first stage aimed at examining the occurrence frequency of each code smell in the analyzed commits. The second stage was concerned with observing the participation of the developers in the analyzed commits. The last stage focused on assessing the relationship of developers on a code smell manifestation. In this last stage, we calculated the average percentage of introduction and of removal of the selected code smell by each group of developers. The idea is to verify the general influence of each group in the project.

IV. STUDY FINDINGS

The first subsection below shows the total number of each investigated code smell in the target systems. The following five subsections report the findings associated with the characterization of code smells and the involvement of the developers. Finally, the last subsection presents some discussion about the results and the impact of developers in code smell introduction.

A. Occurrence of Code Smells

There was a significant difference on how often each investigated code smell occurred in the target systems. The results are summarized in Table II. The "I" column indicates the total number of times each code smell was inserted in each target system and the "R" column indicates the total number of times each code smell was removed. The "Tot" line presents the total number of smell instances detected (inserted and removed respectively). For "I" equal to 0 means that there was no inclusion of this code smell. For "R" equal to 0, means that no removal of said code smell. It is important to mention that not all code smells inserted in the analyzed commits were removed.

TABLE II. CODE SMELL OCCURRENCES

	Behave		JUnit		Mokito		RxJava		VRaptor	
CS	I	R	I	R	I	R	I	R	I	R
DC	91	81	208	262	230	335	168	224	311	517
LC	39	92	204	242	181	340	215	224	220	403
LM	46	61	98	161	112	353	149	225	150	409
LPL	0	0	3	9	0	0	63	112	0	1
UE	69	95	240	269	337	288	205	236	377	419
Tot	245	329	753	943	860	1316	800	1021	1058	1749

B. Dead Code

The *Dead Code* code smell refers to code that is not been used. These code smells were identified using the Empty Code, Unnecessary and Unused Code rulesets in PMD. These rulesets are composed of the following rules:

(a) *Empty Code:* this ruleset aims to check if there are empty statements of any kind (empty method, empty block statement, empty try or catch block, etc.);

(b) *Unnecessary:* this ruleset aims to determine whether there are unnecessary code (unnecessary returns, final modifiers, null checks, etc.);

(c) *Unused Code:* this ruleset aims to find unused or ineffective code (unused fields, variables, parameters, etc.).

The results are summarized in Table III, which shows the percentage of insertion and removal of the Dead Code smell for each target system by developer group.

TABLE III. RESULTS FOR DEAD CODE

	Behave		JUnit		Mokito		RxJava		VRaptor	
Gr.	I%	R%	I%	R%	I%	R%	I%	R%	I%	R%
01	0	0	0	0	74	78	40	39	0	0
02	76	67	25	31	10	12	9	9	15	17
03	24	33	58	53	3	3	29	32	34	26
04	0	0	17	17	0	0	0	0	51	56
05	0	0	0	0	13	7	22	20	0	0

C. Large Class

The *Large Class* code smell refers to classes that are trying to do too much, often showing up as too many instance variables. These code smells were identified using a subset of the Code Size ruleset in PMD. The rules used to identify this code smell were:

 ISBN: 978-1-61208-438-1

(a) *Excessive Class File Length:* usually indicates that the class may be burdened with excessive responsibilities that could be provided by external classes or functions;

(b) *Excessive Public Count:* seeks for large numbers of public methods and attributes.

(c) *NCSS Type Count:* uses the NCSS (Non-Commenting Source Statements) algorithm to determine the number of lines of code for a given type;

(d) *Too Many Fields:* determines if a class has too many fields in its code;

(e) *Too Many Methods:* determines if a class has too many methods in its code.

The results are summarized in Table IV, which shows the percentage of insertion and removal of the Large Class smell for each target system by developer group.

TABLE IV. RESULTS FOR LARGE CLASS

	Behave		JUnit		Mokito		RxJava		VRaptor	
Gr.	I%	R%	I%	R%	I%	R%	I%	R%	I%	R%
01	0	0	0	0	70	79	35	37	0	0
02	69	72	28	29	15	10	10	8	19	15
03	31	28	58	53	2	4	34	30	39	34
04	0	0	14	18	0	0	0	0	42	51
05	0	0	0	0	12	7	20	25	0	0

D. Long Method

The *Long Method* code smell refers to methods that are trying to do too much, often presenting too much code. These code smells were identified using a subset of the Code Size ruleset in PMD. The rules used to identify this code smell were:

(a) *Excessive Method Length:* seeks for methods that are excessively long;

(b) *NCSS Method Count:* uses the NCSS algorithm to determine the number of lines of code for a given method;

(c) *NCSS Constructor Count:* uses the NCSS algorithm to determine the number of lines of code for a given constructor;

(d) *NPath Complexity:* determines the NPath complexity of a method (the number of acyclic execution paths through that method).

The results are summarized in Table V, which shows the percentage of insertion and removal of the Long Method smell for each target system by developer group.

TABLE V. RESULTS FOR LONG METHOD

	Behave		JUnit		Mokito		RxJava		VRaptor	
Gr.	I%	R%	I%	R%	I%	R%	I%	R%	I%	R%
01	0	0	0	0	71	77	42	36	0	0
02	72	64	24	29	13	12	8	10	16	16
03	28	36	58	43	4	3	28	30	39	25
04	0	0	17	28	0	0	0	0	45	60
05	0	0	0	0	13	8	23	24	0	0

E. Long Parameter List

The *Long Parameter List* code smell refers to methods that present a long parameter list usually involving global data. These code smells were identified using a single rule of the Code Size ruleset in PMD:

(a) *Excessive Parameter List:* seeks for methods with numerous parameters.

The results are summarized in Table VI, which shows the percentage of insertion and removal of the Long Parameter List smell for each target system by developer group.

TABLE VI. RESULTS FOR LONG PARAMETER LIST

	Behave		JUnit		Mokito		RxJava		VRaptor	
Gr.	I%	R%	I%	R%	I%	R%	I%	R%	I%	R%
01	0	0	0	0	0	0	40	35	0	0
02	0	0	33	67	0	0	14	19	0	0
03	0	0	67	22	0	0	22	26	0	0
04	0	0	0	11	0	0	0	0	0	0
05	0	0	0	0	0	0	24	21	0	0

F. Unhandled Exceptions

The *Unhandled Exceptions* code smell refers to pieces of code containing malformed throw/try/catch statements. These code smells were identified using a single ruleset in PMD:

(a) *Strict Exceptions:* provides some strict guidelines about throwing and catching exceptions.

The results are summarized in Table VII, which shows the percentage of insertion and removal of the Unhandled Exceptions smell for each target system by developer group.

TABLE VII. RESULTS FOR UNHANDLED EXCEPTIONS

	Behave		JUnit		Mokito		RxJava		VRaptor	
Gr.	I%	R%	I%	R%	I%	R%	I%	R%	I%	R%
01	0	0	0	0	71	82	38	39	0	0
02	71	73	27	33	15	8	13	7	16	14
03	29	27	58	49	3	2	25	33	31	33
04	0	0	15	18	0	0	0	0	53	53
05	0	0	0	0	11	8	24	21	0	0

G. Discussion

Tables 2 to 6 presented the percentage of participation of each group of developers in the insertion and removal of code smells for the five studied systems, represented by the $\%I$ column and the $\%R$, respectively. In the analyzed set of commits of the Behave system, in general, groups 2 and 3 were responsible for inserting and removing such code smells. Group 2 inserted more smells but also removed in an even proportion. For JUnit and VRaptor, groups 2, 3 and 4 were responsible for inserting and removing code smells. Four groups inserted and removed code smells in the Mokito system, but the results point out to group 1 as been the one group with more impact on the insertion and removal of code smells. Finally, for the RxJava, the results indicate that groups 1, 3 and 5 were more responsible for inserting and removing code smells.

 ISBN: 978-1-61208-438-1

In the selected set of commits analyzed in this study, all code smells were decreased (had more removals than insertions). This is an indication that the occurrence of code smells depends on the software evolution. It seems that the code smells in the study tend to appear in preliminary releases with more frequency. We did not use the initial commits in our study to avoid the "cold start" problem as we believed these data would not have a proper indication of code smell removal.

Code Smells with Highest Frequencies. The code smells associated with the problem of dead code and unhandled exceptions fell in the group of highest insertion frequency for the analyzed target systems. A closer look made us to suspect that this probably occurred because groups 1 and 2 were more involved in these code smells. Such groups do not present a high level of participation and have a common authorship behavior, which is line code removal. We understand that, in some cases, lines may have been removed without the appropriate care, resulting in dead code and unhandled exceptions. The code smells associated with the problem of long method fell in the group of highest removal frequency for the analyzed target systems. This finding suggests that the development team for the target systems may have done proper refactoring as to decrease the size of the methods.

No Influence on Code Smells. The classification process found members for all groups in the development teams of every target system. However, there were groups that were not involved with code smells in some systems. For instance, group 4 did not insert nor remove code smells in the Mokito system. Groups 1, 4 and 5 did not insert nor remove code smells in the Behave system. We suspect that this occurred because there were few members in these groups for such systems. We used the whole dataset to classify the developers and when we took a deeper look in a system by system basis, some groups were scarce.

Developers vs. Code Smells. In general, groups 1 to 3 (groups whose members have fewer participation in the code development) tended to have a higher engagement in the introduction and removal of code smells. Initially, we thought that the developers in the groups with higher participation frequency would have more impact in code smell removal. This was not observed. We believe that, in the context of our study, this may have happened due to the fact that groups 4 and 5 were more more responsible for in adding functionality to the target systems whereas the other groups were more involved in fault correction.

Recommendations. Considering the results, it is necessary to evaluate the quality of the source code, taking into account the inclusion and removal of problematic code snippets. Thus, the developers assessment process (Group) must be reevaluated constantly, based on data related to the project's commit history. In addition, it is recommended that there is a mixture of different groups, considering the features that contribute to remove code smells.

H. Limitations

Some limitations or imperfections of our study can be identified and are discussed in the following.

Construct Validity. Threats to construct validity are mainly related to possible errors introduced during specific data processing from the repository. The repository did not provide an unique identification data for a developer, thus, it was not possible to determine whether a developer performed commits with different identifications. In this sense, each developer (responsible) identified in the repository was treated as a different developer. However, the study was not intended to focus on the contribution of a specific developer.

Conclusion Validity. We have three issues that threaten the conclusion validity of our study: the number of evaluated systems; the evaluated code smells (and their relation to the PMD rules), and; discarding the data from the commits that did not increase nor decrease the number of code smells. Five open source projects from Git were analyzed. A higher number of systems is always desired. However, the analysis of a bigger sample in this kind of study could be non-practical. The number of systems with all the required information available to perform this kind of study is bare. We understand that our sample can be seen as appropriate for a first exploratory investigation [18]. Related to the second issue, our analysis used the PMD tool. Regarding the set of code smells used in the study, code smells reported in the literature were considered in our study. Finally, we discarded data from commits that maintained the amount of code smells. Although the study focused on associating developer profiles to improving or lessen the quality of the code, we understand that this limitation does not allow us to make a conclusion for a specific code smell.

V. Related Work

There are several approaches available in the literature for detecting Code Smells. Mantyla investigated as developers identify and treat Code Smells in the source code to compare with automated detection methods [19]. There are also several approaches available in the literature for investigation of the effects of Code Smells in aspects related to software maintainability [20], such as defects [21], effort [22] and requests for changes [23]. In addition, few studies have focused on the detection of Code Smell through mining activities in software repositoryc [24].

Regarding the classification of developers in groups, there are several existing approaches in the literature. In this context, one of the proposals is based on data extracted from the repository in relation to the time of performing the commit. Thus, the model proposes to assess in which the range of hours developers insert more bugs in your commits [25]. Another approach is to sort the developers on the basis of the records related to quantity, time, and type of actions and activities that these developers come true, working on the project, and the data extracted from the version control system and other tools, such as mailing list and bug tracker tools [26][27].

VI. Concluding Remarks

This work presented a study to assess the influence of developers on the introduction of code smells in a software system. We classified the developers into five categories and verified their contributions (increasing or decreasing) in the number of code smells in a set of consecutive software versions. This exploratory study revealed, within the limits of the threats to its validity, the conjecture that the team member behaviour (participation frequency, authorship and development activity - feature development or fault correction) impacts in the insertion and removal of code smells.

 ISBN: 978-1-61208-438-1

Finally, it is important to highlight that we have analyzed commits of five systems. Then, the relationships of code smells and developers should be tested in broader contexts in the future. It would also be desirable to use the development activity of the developers in the classification and association of developers.

ACKNOWLEDGMENT

The authors thank everyone who provided knowledge and skills that really helped the search. The result is a compilation of ideas and concepts throughout the development of this work.

REFERENCES

[1] IEEE, IEEE Standard for Software Maintenance, IEEE Std 1219-1998. IEEE Press, 1999, vol. 2.

[2] "Software engineering - product quality, ISO/IEC 9126-1," International Organization for Standardization, Tech. Rep., 2001.

[3] F. A. Fontana and M. Zanoni, "On investigating code smells correlations," in ICST Workshops'11, 2011, pp. 474–475.

[4] A. F. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey." in WCRE, R. Lammel, R. Oliveto, and R. Robbes, Eds. IEEE, pp. 242–251. [Online]. Available: http://dblp.uni-trier.de/db/conf/wcre/wcre2013.html (access date: September 2015)

[5] I. M. Bertran, "Detecting architecturally-relevant code smells in evolving software systems," in Proceedings of the 33rd International Conference on Software Engineering, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 1090–1093. [Online]. Available: http://doi.acm.org/10.1145/1985793.1986003 (access date: September 2015)

[6] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "Balancing agility and formalism in software engineering," B. Meyer, J. R. Nawrocki, and B. Walter, Eds., 2008, ch. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, pp. 252–266.

[7] GitHub, "Git repository," https://github.com, 2014.

[8] B. F. Webster, Pitfalls of object-oriented development. M And T, 1995.

[9] J. Long, "Software reuse antipatterns," SIGSOFT Softw. Eng. Notes, vol. 26, no. 4, Jul. 2001, pp. 68–76. [Online]. Available: http://doi.acm.org/10.1145/505482.505492 (access date: September 2015)

[10] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:10. [Online]. Available: http://doi.acm.org/10.1145/1852786.1852797 (access date: September 2015)

[11] R. S. Pressman, Software Engineering: A Practitioner's Approach, 5th ed. McGraw-Hill Higher Education, 2001.

[12] E. Di Bella, A. Sillitti, and G. Succi, "A multivariate classification of open source developers," Inf. Sci., vol. 221, Feb. 2013, pp. 72–83. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2012.09.031 (access date: September 2015)

[13] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in Proceedings of the 6th International Conference on Predictive Models in Software Engineering, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 18:1–18:9. [Online]. Available: http://doi.acm.org/10.1145/1868328.1868356 (access date: September 2015)

[14] M. Zhou and A. Mockus, "Developer fluency: achieving true mastery in software projects," in Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010, 2010, pp. 137–146. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882313 (access date: September 2015)

[15] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability, L. M. L. Cam and J. Neyman, Eds., vol. 1. University of California Press, 1967, pp. 281–297.

[16] InfoEther, "Pmd is a source code analyzer," http://pmd.sourceforge.net/, 2014.

[17] M. Fowler, Refactoring: Improving the Design of Existing Code. Boston, MA, USA: Addison-Wesley, 1999.

[18] B. Kitchenham, H. Al-Khilidar, M. A. Babar, M. Berry, K. Cox, J. Keung, F. Kurniawati, M. Staples, H. Zhang, and L. Zhu, "Evaluating guidelines for empirical software engineering studies," in Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ser. ISESE '06. New York, NY, USA: ACM, 2006, pp. 38–47. [Online]. Available: http://doi.acm.org/10.1145/1159733.1159742 (access date: September 2015)

[19] M. Mantyla and C. Lassenius, "What types of defects are really discovered in code reviews?" IEEE Trans. Software Eng., vol. 35, no. 3, 2009, pp. 430–448. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2008.71 (access date: September 2015)

[20] D. I. Sjoberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," IEEE Transactions on Software Engineering, vol. 39, no. 8, 2013, pp. 1144–1156.

[21] F. Rahman, C. Bird, and P. T. Devanbu, "Clones: What is that smell?" in MSR, J. Whitehead and T. Zimmermann, Eds. IEEE, 2010, pp. 72–81. [Online]. Available: http://dblp.uni-trier.de/db/conf/msr/msr2010.html (access date: September 2015)

[22] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension." in CSMR, T. Mens, Y. Kanellopoulos, and A. Winter, Eds. IEEE Computer Society, 2011, pp. 181–190. [Online]. Available: http://dblp.uni-trier.de/db/conf/csmr/csmr2011.html (access date: September 2015)

[23] S. M. Olbrich, D. Cruzes, and D. I. K. Sjberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems." in ICSM. IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dblp.uni-trier.de/db/conf/icsm/icsm2010.html (access date: September 2015)

[24] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on. IEEE, 2012, pp. 411–416.

[25] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" in Proceedings of the 8th Working Conference on Mining Software Repositories, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 153–162. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985464 (access date: September 2015)

[26] W. Poncin, A. Serebrenik, and M. van den Brand, "Process mining software repositories," in Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on, March 2011, pp. 5–14.

[27] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," in Proceedings of the International Workshop on Principles of Software Evolution, ser. IWPSE '02. New York, NY, USA: ACM, 2002, pp. 76–85. [Online]. Available: http://doi.acm.org/10.1145/512035.512055 (access date: September 2015)

 ISBN: 978-1-61208-438-1