2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)

Software Evolution and Quality Data from Controlled, Multiple, Industrial Case Studies

Aiko Yamashita†, S. Amirhossein Abtahizadeh*, Foutse Khomh*, and Yann-Gaël Guéhéneuc*
*Department of Computer and Software Engineering, Polytechnique de Montreal
{a.abtahizadeh,foutse.khomh,yann-gael.gueheneuc}@polymtl.ca
† Centrum Wiskunde & Informatica, and Oslo and Akershus University College of Applied Sciences
aiko.fallas@gmail.com

***Abstract*—A main difficulty to study the evolution and quality of real-life software systems is the effect of moderator factors, such as: programming skill, type of maintenance task, and learning effect. Experimenters must account for moderator factors to identify the relationships between the variables of interest. In practice, controlling for moderator factors in realistic (industrial) settings is expensive and rather difficult. The data presented in this paper has two particularities: First, it involves six professional developers and four real-life, industrial systems. Second, it was obtained from controlled, multiple case studies where the moderator variables: programming skill, maintenance task, and learning effect were controlled for. This data set is relevant to experimenters studying evolution and quality of real-life systems, in particular those interested in studying industrial systems and replicating empirical studies.**

***Index Terms*—software quality, software evolution, software defects, software replicability, case study, empirical study, moderator factors, industrial data, replication, code smells.**

I. Introduction

Many *moderator factors* can impact the outcome of a software engineering activity [1]. Experimenters must control them (or take them into account by measuring them) to avoid spurious relationships between the variables of interest. The moderator factors: programming skill and learning effect are important factors to consider because software development is intrinsically a human endeavor in which developers play a major role [2]. Equally, *type of task* have been shown to be an important moderator in program comprehension [3]. However, it is not possible to easily control for these factors in real-life settings because of their very nature: their values depend on the contexts of the systems, which are often beyond the experimenters' controls. In this paper, we present a data set that has two particularities. First, it involves six professional developers and four real-life, *industrial* systems. Second, it was obtained through a *controlled*, *multiple* case study designed to control for the moderator factors: programmer skill, maintenance task and learning effect. In the study, *programmer skill* was controlled by selecting developers from a pool of developers in the same range of programming skills assessed through a previous, independent study by Bergersen & Gustafsson [4]. *Maintenance task* was controlled by involving four systems with near-identical features, over which identical tasks could be performed. Finally, *learning effect* was controlled by having each developer replicate the same tasks on two different systems. The primary data set contains *multiple evolution histories*, in the form of git repositories, and multiple *sets of defects*, in the form of reports (excel files) extracted from issue tracking systems. The secondary data consists of a series of attributes extracted from the software systems (i.e., code smells) and their evolution (i.e., code churn), and a log specifying the dates on which developers worked on each of the systems/tasks, in the form of excel files. The remainder of this paper is organized as follows: Section II provides a brief background of the study from which the data was obtained. Section III describes the data being released. Section IV explains how the data can be accessed and used as well as caveats and limitations. Section V concludes and presents future work.

II. Background of the Data Set

This data set is derived from a *controlled, multiple*, industrial case study conducted by Simula Research Laboratory in 2008, with the objective of investigating the effect of code smells on the evolution and quality of real-life, industrial software systems. Previous work describes in detail the context and the methodology of the study [5].

A. Systems

The four systems forming part of this data set, and involved in the study described in [5] originally come from a previous study conducted by Simula Research Laboratory in 2003 [6] to investigate *software project replicability*. For the study, Simula issued a tender for the development of a new web-based information system. Based on the bids, they hired four Norwegian software consultancy companies to develop independently a version of the system, using the *same requirements specification*. This resulted in four systems (hereon denoted as A, B, C and D) with near-identical features, but with dissimilar size, design, and implementation. The systems encompass a

TABLE I
LOC per file type for all four systems.

System	A	B	C	D
Java	8205	26679	4983	9960
JSP	2527	2018	4591	1572
Others	371	1183	1241	1018
Total	11103	29880	10815	12550

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.44

Round	Developer 1	2	3	4	5	6
1	A	B	C	D	C	A
2	D	A	D	C	B	B

Fig. 1. Assignment of systems to developers in the study.

wide range of software artifacts such as Java, JSP, and other file types, such as XML and HTML as summarized in Table I.

The main feature of all four systems comprised of keeping a record of the empirical studies and related information at Simula, e.g., the researcher responsible for the study, participants, data collected, and publications resulting from the study. The systems were all deployed on Simula's Content Management System (CMS), which at that time was based on PHP and a relational database system.

B. The Controlled, Multiple Case Study

In 2008, the Simula Research Laboratory introduced a new CMS called Plone (plone.org), thus all the four original systems needed to be adapted to the new environment, providing the opportunity to conduct and observe a real-life software maintenance project. The maintenance project was conducted in 2008 by outsourcing two European software consultancy companies. The entire study lasted four months, at a total cost of 50.000 Euros. The project comprised of three maintenance tasks: the first two tasks involved adapting the system to the new platform, and the third task adding new functionality.

Six software professionals were recruited from a pool of 65 participants of a previous study on programming skill [4]. All the selected developers, with the exception of one (developer 3) who had a slightly higher than average skill, had been evaluated to have a similar development skill. These developers completed all three maintenance tasks *individually* on one system, and then repeated the same tasks on a second system. We then distinguish "rounds" where *first round* corresponds to a case where the developer has not maintained any of the systems previously, and *second round* denotes a case where developers repeat the tasks on a second system. Consequently, we use "rounds" as a proxy for learning effect. Figure 1 describes the order in which the systems were assigned to each developer. This assignment resulted in 3 projects per system, i.e., 6 developers x 2 systems = 12 projects (cases) in total. The development took place entirely at the company sites and the first author of this paper was present in both sites for the entire duration of the project. The developers were given no information on what the study entailed. Eclipse was used as the development tool, together with MySQL (www.mysql.com) and Apache Tomcat (http://tomcat.apache.org). Subversion or SVN (http://subversion.apache.org/) was used as the versioning system. There were no unit tests available, due to the heterogeneity of the systems. Instead, acceptance testing was done to identify defects at system level, which were then registered in Trac (https://trac.edgewall.org/).

III. Data Description and Release Approach

A. Data Description and Collection

The dataset comprises of four parts: 1) Software evolution history, 2) Software defects, 3) Code smells and change-related metrics (of both the initial and final versions of the artifacts), and 4) Task dates.

a) Software evolution history: As mentioned previously, there were six developers involved in the study. Each was given access to one SVN repository containing the two systems they were assigned to. Consequently, there are six repositories in total. Each repository contains the evolution of two of the four systems, as depicted in Fig. 1. In total, there are 12 code evolution histories, each displaying a certain combination of the variables: system, developer and round. The repositories were initially in SVN, but later on were migrated to Git (see section III-B for further details).

b) Software defects: In the same way as each developer was given access to an SVN repository, they were given access to one issue tracking system (Trac). Each Trac project was in addition, integrated with the corresponding SVN repository. Each issue tracking system registered the defects corresponding to the two systems each developer worked on. In total, there are 12 excel files of software defects, each set displaying a certain combination of the variables: system and developer, where the file format is: "Defects_Dev{1/2/3/4/5/6}_Sys{A/B/C/D}.xlsx. Trac treats every issue as a "Ticket", and each Ticket contains the following information: Ticket ID, Status, Resolution, Severity, Priority, Created, Modified, Summary, and Description. The description of the maintenance tasks were not included as tickets in Trac, but rather given as a specification document to the developers.

c) Code smells and change metrics: The following code smells were detected in the four systems: Data Class, Data Clumps, Duplicated code in conditional branches, Feature Envy, God (Large) Class, God (Long) Method, Misplaced Class, Refused Bequest, Shotgun Surgery, Temporary variable used for several purposes, Use of implementation instead of interface, and Interface Segregation Principle (ISP) Violation. Note that these metrics are available only for Java files due to limitations in the tool. For each code smell instance identified, the following information was extracted: Smell name, smell description/details, Filename, and System. The code smells were extracted from the original (untouched) version before they were subjected to changes by the developers ("InitialSmells.xls"), and they were also measured on the 12 resulting versions after the developers finalized the maintenance tasks ("FinalSmells.xls"). Consequently, the last data contains in addition the following information: System, Developer, Round. For detecting the code smells, two commercial tools were used: Borland Together and *InCode*; they both use the detection strategies (metrics-based interpretations of code smells) by Marinescu [7]. As for code *change metrics* ("Changes.xls"), the following metrics were calculated for each commit revision: Programmer, Revision No., Date, Full path, Filename, File extension, System, Action Type (i.e., Added, Deleted, Modified, Renamed), No. lines added, No. lines deleted, No. lines changed, and Churn. We used the definition by Hall and Munson [8] for churn: the absolute number of changes (i.e., number of lines changed + added + deleted) made over a number of versions of a software unit. There were in total 2400 commits from which we extracted these metrics. The change-related metrics were calculated by writing a Java program that used SVNKit (http://svnkit.com), a library for extracting data from Subversion.

d) Task Dates: An important feature of the data on this study is the fact that developers perform identical tasks across systems. This means that the effect of the type of task could be ruled out for the purposes of secondary analyses. During the study, different features of the case study protocol were used, one of them was *daily interviews* with the developers and a *study log* that the PI (Principal Investigator) kept during the entire study. Based on these two sources, it was possible to determine per day, which tasks the developers were working on. This can provide contextual information when analysing the commit logs and the defect data in Trac. This data ("Task-Dates.xls") contains the following information: Programmer, System, Task (i.e., 1,2,3), and Date. Detailed description of each of the tasks is available in [5]. Note that there are some overlapping dates between tasks or even systems. These represent situations where the developer finished a task and moved on to the next task or system during the same day.

B. Process for Releasing the Data

We first converted the SVN repositories into Git to allow researchers to work off-line with the data and clone the repositories for their own needs easily. Also, Git is considerably more popular for repository analysis tools than SVN. The Git repositories were cloned on GitLab[1] instance deployed over a VM within the Polytechnique of Montreal infrastructure. The second task consisted of anonymising the data. We wrote a script to remove/change the history of Git, following instructions from[2]. Sensitive folders were removed from the history using "–filter-branch". Then, strings containing names of developers (including complete name, short names, nicknames or parts of the name), emails, phones, and references to Simula's employees or infrastructure were replaced (consistently) with a Python script using a mapping file which contained all the patterns for the sensitive strings and corresponding anonymised strings. The same script was used to change the authors of the commits and re-write the master HEAD branch (to anonymise names in the history). For the defects, it was not possible to reinstall the Trac server because it was previously integrated with SVN, and now we have moved to Git. Thus, we extracted the defect reports and published them in Zenodo.org, with the secondary data set.

IV. Data Usage

A. How to access the data?

The evolution history is available at the following url: http://opendata.soccerlab.polymtl.ca/git/explore/projects. The researcher should then create an account in order to access, but it should be granted automatically. The defect data, alongside the secondary data, can be accessed via Zenodo, at the following url: https://zenodo.org/record/293719. The script for anonymising git is available at: https://goo.gl/B2XM9z.

B. How has the data been used?

The data presented in this paper has been used[3] in the doctoral dissertation by Yamashita [5]. In addition, it has been used in the work by Yamashita & Counsell [9], where it was investigated if code smells can be used as system-level indicators of maintainability. Analyses reported in [10, 11] investigate whether code smells can be used as indicators of problematic artifacts, and to which extent can code smells uncover *maintenance problems* in general. Sjøberg et al., [12] and Soh et al., [13] also use this data set in conjunction to additional data (e.g., interaction traces) to quantify the effect of code smells on maintenance effort at different granularity levels. Finally, Yamashita & Moonen [14] also use this dataset to investigate the phenomenon of inter-smell relations in Object Oriented systems.

C. Potential usage scenarios

a) Analysis of "repeated defects" in a multiple case study: There is evidence that developers introduced similar defects while working on the same system, which hints that some defects are "meant to happen". It could be interesting to examine these defects and investigate the properties of the system leading to the introduction of those defects.

b) Studies on the impact of different metrics/attributes on software evolution: This data has primarily been used to investigate the effect of code smells on software maintainability, but other metrics/attributes can be extracted from the systems, and validated across the different cases.

c) Further studies of inter-smells: Explanatory and predictive models built based on the notion of inter-smells –by using techniques such as association mining or clustering can be contrasted to traditional, file-based analyses.

d) Cost-benefit analysis of smell removal: The repositories can be mined for the refactorings performed, and can be contrasted to evaluate which refactorings actually "paid off" from quality and–or change size perspectives.

e) Benchmarking of tools/methodologies: The data set and the underlying systems can be used for benchmarking purposes, when evaluating new tools for metrics detection, defect extraction, or any other methodologies.

f) Task/context extraction: A possible improvement on the data set concerns the accuracy of the time when a given task was performed. Currently, the data only defines the *date* on which a developer was working on a given task/system (with the already mentioned "overlap" issue). It could be valuable to experiment with techniques/tools/methods that can allow identifying the exact context (e.g., task) at the time of each commit. Such techniques have industrial applications such as the one reported by [15].

[1] https://about.gitlab.com/gitlab-com/ [2] https://goo.gl/LXtKUE

[3] In conjunction with other qualitative data collected during the same study.

D. Challenges and Limitations

a) Context of the study: The external validity of any results stemming from this data are contingent to the context of the study, in this case: medium-sized, Java-based, three-layered architecture, web-based, information systems.

b) Tasks were individual: The software professionals completed the project individually, i.e., not in teams or pair programming. This can affect the applicability of results obtained from this data in highly collaborative environments.

c) Time frame: The data does not fully represent a long-term maintenance project with large tasks, given the size of the tasks and the shorter maintenance period covered in the study. However, tasks resemble backlog items in a single sprint/iteration within the Agile context.

d) The age of the systems: The technology used in this study is already nearly 10 years (14 if the original study [16] is considered). However, there are still many industrial systems which are even older than 14 years, and the technology involved is still quite relevant to current software projects.

e) Tool availability: Unfortunately, the tools used for detecting the code smells are not available anymore, thus researchers would need to resort to alternative tools for extracting code smells, with possibly different results.

f) No explicit corrective tasks: The tasks considered explicitly in the study design are only of adaptive and perfective nature. Thus, the corrective tasks manifested in the study are not controlled for (i.e., they were generated from the defects originally existing in the systems –which were of diverse nature, and side-effects from developers' changes).

g) Date accuracy for the task: As previously mentioned, the data specifies the dates for which a developer worked in a specific task/system. However, if the developer did several commits the same day, is not always straightforward to determine to which task the commit corresponds to (in particular commits concerning the same system).

h) Quality of the defect reports and commit logs: Although developers were instructed to report the defects with as much information as possible, this was not always the case. Also, not all the commit logs were associated with an issue (Ticket) ID. This may require in some cases, mining techniques to link a commit to a bug fix.

i) Realism of the study: Is natural to believe that there will be a trade-off between the degree of realism and the degree of control in such type of studies (for a more detailed discussion on this issue, see [5]). We believe the systems and tasks belong to a realistic setting, and special care was put in order to ensure as much as possible, a realistic project.

V. Conclusion and Future Work

We presented a data set that has two particularities: first, it involves six professional developers and four real-life, industrial systems. Second, it was obtained from *controlled*, *multiple* case studies designed to control for the factors: programming skill, task and learning effect. In future work, we plan to: 1) release more data from this study, 2) provide concrete guidelines for sharing diverse types of data from software engineering studies, and 3) present a proposal for a platform that could constitute a more intuitive approach for sharing research data.

References

[1] F. Shull, V. Basili, J. Carver, J. C. Maldonado, G. H. Travassos, M. Mendonça, and S. Fabbri, "Replicating Software Engineering Experiments: Addressing the Tacit Knowledge Problem," in *Proceedings of the 2002 International Symposium on Empirical Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2002.

[2] R. M. Belbin, *Management teams : why they succeed or fail*. Butterworth-Heinemann, 2010.

[3] J. Burkhardt, F. Détienne, and S. Wiedenbeck, "Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase," *Empirical Software Engineering*, vol. 7, no. 2, pp. 115–156, 2002.

[4] G. R. Bergersen and J.-E. Gustafsson, "Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective," *Journal of Individual Differences*, vol. 32, no. 4, pp. 201–209, 1 2011.

[5] A. Yamashita, "Assessing the Capability of Code Smells to Support Software Maintainability Assessments: Empirical Inquiry and Methodological Approach," Ph.D. dissertation, University of Oslo, 2012.

[6] B. C. D. Anda, D. I. K. Sjøberg, and A. Mockus, "Variability and Reproducibility in Software Engineering : A Study of Four Companies that Developed the Same System," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 407–429, 2009.

[7] R. Marinescu, "Measurement and Quality in Object Oriented Design," Ph.D. dissertation, Politehnica University of Timisoara, 2002.

[8] G. A. Hall and J. C. Munson, "Software evolution: code delta and code churn," *Journal of Systems and Software*, vol. 54, no. 2, pp. 111–118, 2000.

[9] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An Empirical Study," *Journal of Systems and Software*, 2013.

[10] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data," *Empirical Software Engineering*, vol. 19, no. 4, pp. 1111–1143, 3 2013.

[11] A. Yamashita and L. Moonen, "To what extent can maintenance problems be predicted by code smell detection? An empirical study," *Information and Software Technology*, vol. 55, no. 12, pp. 2223–2242, 12 2013.

[12] 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, pp. 1144–1156, 8 2013.

[13] Z. Soh, A. Yamashita, F. Khomh, and Y.-G. Gueheneuc, "Do Code Smells Impact the Effort of Different Maintenance Programming Activities?" in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 3 2016, pp. 393–402.

[14] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, pp. 682–691.

[15] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 134–144.

[16] B. C. D. Anda, "Assessing Software System Maintainability using Structural Measures and Expert Assessments," in *IEEE International Conference on Software Maintenance*, 2007, pp. 204–213.