Dataset of Developer-Labeled Commit Messages

Andreas Mauczka, Florian Brosch, Christian Schanes, Thomas Grechenig
Vienna University of Technology
Institute of Industrial Software (INSO)
Karlsplatz 13, 1040 Vienna, Austria
{andreas.mauczka, florian.brosch,
christian.schanes, thomas.grechenig}@inso.tuwien.ac.at

***Abstract*—Current research on change classification centers around automated and semi-automated approaches which are based on evaluation by either the researchers themselves or external experts. In most cases, the persons evaluating the effectiveness of the classification schemes are not the authors of the original changes and therefore can only make assumptions about the intent of the changes. To support validation of existing labeling mechanisms and to provide a training set for future approaches, we present a survey of source code changes that were labeled by their original authors. Seven developers from six different project applied three existing classification schemes from current literature to enrich their own changes with meta-information, so the intent of the changes becomes more evident. The final data set consists of 967 classified changes and is available as an SQLite database as part of the MSR data set.**

I. Introduction

The study of software evolution has a long history and many efforts have been taken to automatically mine repositories to gain insight into the software evolution process. Specifically the Mining Software Repository community has a long running history of studying how software changes over time by mining version control systems or bug tracking systems. Over the years, several studies have been performed on automatic classification of repository artefacts, be it bug reports (e.g. Antionol et al. in [2]) or commit messages (e.g. [6], [7], [4], [8]). However, most of these studies face the challenge of evaluating the performance of the approach internally and externally. A commonly used approach to measure success of a classification mechanism is to evaluate precision and recall or related values. These measurements are often performed internally by the researchers themselves (e.g. see [3], [8] or [4]), or externally, by experts (see e.g. [7]). In both cases the evaluators are commonly not the authors of the changes and therefore cannot be sure of the intent of the change. Only the author of a change knows the desired effect of a commit.

Hence, our goal is to perform a survey on open source project contributors and have them classify their own changes by applying three different classification methods. We chose three classification schemes for our survey – more schemes would have reduced the number of classified commits, due to the time the developers were willing to invest. The first scheme is one implemented in e.g. [7] and is based on a slightly modified set of Swanson's maintenance task categories. The second scheme has been implemented by Hindle et al. in [6] and provides more detailed information on change intent by using non-functional requirements to classify commits. The third scheme is similar to the first scheme, but has been tailored specifically for software evolution in open source projects and was provided by Hattori and Lanza in [4]. Based on this data, future classification techniques might be evaluated using the gathered commit meta-information and existing approaches may be a posteriori re-evaluated for effectiveness.

II. Assembly of the data

To provide transparency on the process of data aggregation and transformation, the following section describes how the data for the final data set was assembled. We performed a four-steps process:

1) Selection of developers and projects with personal commitment
2) Assembly of commit data and creation of survey forms
3) Provisioning instructions and guidance
4) Aggregation of the data into a single data source

A. Selection of developers and projects

To perform the survey, we contacted developers that were or are regular committers to open source projects. All developers are senior developers (6+ years experience in open source or industry projects) and personal acquaintances of the authors of this paper. Due to the exhaustive scale of the survey, personal commitment of the developers was important, since the task could take more than two hours depending on the number of commits to be categorized. Seven developers including one of the authors of the study agreed to perform the categorization.

The projects involved are very diverse, which may be beneficial for later research based on this data (e.g. in evaluation of cross-domain valid approaches). For example, Vala[1] is a compiler in the GNOME context, while Mylyn Reviews[2] is an established review tool provided as an Eclipse plug-in. The list of the projects may be found in the data set.

B. Assembly of commit data and creation of survey forms

We used the automated mining tool SubCat[3] to aggregate all data necessary for the survey forms. SubCat offers, among other mining capabilities, functionality to mine GIT

[1]https://wiki.gnome.org/Projects/Vala
[2]https://projects.eclipse.org/projects/mylyn.reviews
[3]https://github.com/andreasmauczka/SubCat

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.71

repositories and stores the mined information into an SQLite[4] database, which allowed for fast data exports into our survey forms. Every participant received a survey form that contained all of his commits from his open source project. The survey form supplied columns for each of the classification schemes as well as commit-identifiers, the commit message and meta-data on the change (e.g. changed files). We kept the initially generated database and stripped the model of all entities and attributes that were not relevant for the survey to be able to easily integrate the returned survey forms into the final data set. We inserted a table that contains the survey results into the SQLite database. This ensures comfortable integration of the data into any future studies. Figure 1 shows the model for the survey data.

C. Provide instructions and guidance

Adequate instructions as well as guidance provided to the developers who participated in the survey are crucial for the data quality in the final data set. Since classification is often ambiguous, we tried to provide the same basis for all participants. First, we provided a set of instructions for every participant. Once the participants read these, we supplied them with the survey sheet. Afterwards, an author of the study would make an appointment for a guidance session with the participant. In this session the participant could ask questions about the classification schemes and ask for assistance on the first set of commits. These sessions were intentionally held open, so that the author would not influence the participant in his application of the scheme, i.e. the author would only recount the instructions if the participant obviously misunderstood a category. Naturally, some questions on the schemes arose which may be found in section III-B.

For the survey, we allowed for multi-labeling approaches, similarly to Hindle et al. in [6], since the primary goal of the survey is to capture as much of the intent of a change as possible. This is reflected in the instructions for all three schemes. We asked the participants to remove any changes where they were unsure or forgot about the intent of the change to increase the precision of the data set.

Instructions for Swanson's maintenance tasks: The first applied classification scheme is based on Swanson's maintenance tasks. It was applied in several studies (see e.g. [7]). We provided the participants with the following definitions of maintenance tasks (note that these definitions are altered from the original by Swanson to cover the open source development life cycle):

- **Corrective tasks** are corrective measures to the code base that address errors[5], faults[6] or failures[7] (according to the IEEE glossary for software engineering [1]) in the code base. This includes preventive maintenance steps taken to address latent faults that are not operational faults yet. E.g. a regression bug that is found during testing after the software was released. This does not include preventive measures that improve non-functional attributes.
- **Adaptive tasks** are changes that affect the business logic of the system, e.g. changes to implement new or alter existing functional requirements. These changes may stem from changes to the model of the software, but also from alterations of algorithms.
- **Perfective tasks** are enhancements to non-functional attributes of the system, e.g. boosting performance, refactoring or improving system documentation.

We asked the developers to classify all their commits according to this scheme. If a commit addressed a functional requirement, the developer was supposed to mark the commit as adaptive. They may assign multiple categories, if they thought a commit has addressed more than one category, e.g. a bug fix that also included some refactoring work would be marked as corrective and perfective. If all three categories were not fitting (e.g. actions like version tagging), we asked them to simply not mark the commit.

Instructions for NFR labeling: The second classification of commits involves the non-functional requirements (NFR) a commit addresses. This classification is based on the ISO9126 quality model and was proposed by Hindle et al. in [6]. A commit may possibly be assigned to multiple NFRs - a commit may implement new features like a search bar, along with a design that improves usability of the product. Similarly to the previous classification, we wanted developers to assign the commit to every NFR that they thought applied to their change. If they found all six categories not fitting, they were supposed to mark the commit in the *none* column. The possible categories were:

- **Functionality:** The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions. This includes *Suitability, Accuracy, Interoperability, Security and Functionality Compliance*.
- **Reliability:** The capability of the software product to maintain a specified level of performance when used under specified conditions. This includes *Maturity, Fault Tolerance, Recoverability and Reliability Compliance*.
- **Usability:** The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions. This includes *Understandability, Learnability, Operability, Attractiveness and Usability Compliance*.
- **Efficiency:** The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions. This includes *Time Behaviour, Resource Utilization and Efficiency Compliance*.
- **Maintainability:** The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional spec-

[4]http://www.sqlite.org/

[5]The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition

[6]An incorrect step, process, or data definition

[7]An incorrect result

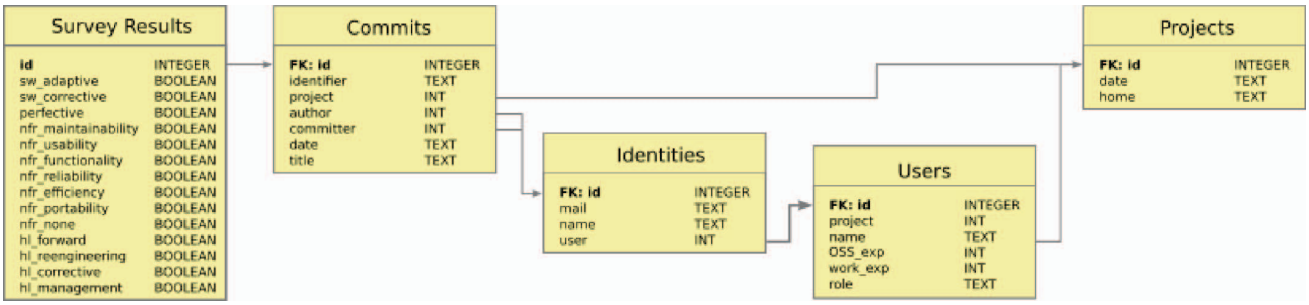

Fig. 1. Data set model for multi-projects task classification

ifications. This includes *Analyzability, Changeability, Stability, Testability and Maintainability Compliance*.

- **Portability:** The capability of the software product to be transferred from one environment to another. This includes *Adaptability, Installability, Co-Existence, Replaceability and Portability Compliance*.

Instructions for software evolution tasks: The third classification of commits is the scheme for activities during software evolution in open source projects, presented by Hattori and Lanza in [4]. The same procedure as with the initial classification was followed. We asked the developers to classify all of their commits according to this scheme. If a commit involved a forward engineering activity, they were supposed to mark it as forward engineering. Again, it was possible to assign multiple categories, if they thought a commit addressed more than one activity, e.g. a bug fix that also included some refactoring would be marked as corrective engineering and re-engineering. If all four categories were not fitting, the developers did not mark them. Hattori and Lanza defined the following activities:

- **Forward engineering** activities are those related to incorporation of new features and implementation of new requirements.
- **Re-engineering** activities are related to refactoring, redesign and other actions to enhance the quality of the code without properly adding new features.
- **Corrective engineering** handles defects, errors and bugs in the software.
- **Management** activities are those unrelated to codification, such as formatting code, cleaning up, and updating documentation.

III. Results and Discussion

Table III shows an overview of the returned results. For each change that was assigned to a category in a schema, we added one point to the overall category.

A. Aggregation of the data into a single data source

Before we aggregated the data, we removed the first ten commits classified by the developer from each survey form. This was necessary, since the authors assisted during some of these commits and feedback form the developers suggested that ten commits is a sufficiently large training set to understand each classification scheme.

Once we removed the first ten commits, we aggregated the survey forms and imported them into the database. The database holds the raw data in the table `Commits` and the manually classified commits in the table `SurveyResults`. It also holds a table `Identitites` to store user and author information. Furthermore it stores general project information in `Projects`. Figure 1 shows the model for the final data set. Table `Users` is necessary due to compatibility reasons with SubCat so Bug Tracking System account information may be mapped to commit messages in future studies.

B. Discussion on the classification schemes

Feedback from the developers showed that all of the classification schemes were easy to apply and were perceived as coherent. Some problems occurred when changes did not fit any of the options provided by a scheme, e.g. developers reported that the category 'Management' of Hattori and Lanza [4] proved useful to identify changes they performed to tag versions or to create private branches. An example for this was the commit with the message "Creating private branch for the implementation of key refactoring functionality." which could be labeled with the third categorization scheme, while it could only be categorized as "None" in the second scheme and could not be categorized at all according to the first scheme.

One of the developers (Evan Nemerson) in the survey turned out to perform a specialized role in the Vala project. He is responsible for maintaining language bindings for Vala. His tasks are either triggered by an issue report, or by an update in a dependent API. Hence his task spectrum should be very one dimensional in all three schemes. This might be interesting for further research on developer roles.

Luca Bruno from the Vala project pointed out that a lot of his development tasks are also based on bug reports. Similarly to Evan, bug reports trigger his development work – sometimes his commit ends up including a bug fix as well as major new functionality. This means that the boundary between a corrective maintenance task and an adaptive maintenance task (analogously a corrective engineering activity and a forward engineering activity) are not clearly defined in some cases. The second classification scheme avoids this issue, since bug-fixing and adding functionality both fall into the *Functionality* category.

TABLE I
CLASSIFICATION OVERVIEW

		Swanson's Tasks (%)			**NFR Labeling** (%)							**SW Evolution Tasks** (%)			
	total	*adap.*	*corr.*	*perf.*	*func.*	*maint.*	*usab.*	*rel.*	*eff.*	*port.*	*none*	*forw.*	*corr.*	*reeng.*	*man.*
Florian Brosch[a]	200	51,98	29,7	18,32	60,08	7,98	22,05	6,84	0,0	0,76	2,28	57,64	29,56	8,37	4,43
Luca Bruno[b]	116	37,29	54,24	8,47	66,93	6,3	9,45	3,15	3,94	0,79	9,45	34,75	50,85	10,17	4,24
Evan Nemerson[b]	194	34,36	59,49	6,15	51,59	5,1	41,4	0,32	0,0	0,64	0,96	32,67	56,44	4,46	6,44
Thomas Seidl[c]	118	18,52	55,56	25,93	26,35	7,43	20,95	14,86	4,73	3,38	22,3	12,3	48,36	9,84	29,51
Martin Reiterer[d]	123	19,82	19,82	60,36	10,42	44,44	3,47	6,25	3,47	8,33	23,61	11,97	16,2	34,51	37,32
Kilian Matt[e]	81	34,83	15,73	49,44	25,71	20,95	19,05	9,52	1,9	2,86	20,0	28,87	15,46	16,49	39,18
Mark Struberg[f]	135	15,6	41,13	43,26	28,11	10,14	13,82	4,15	0,92	19,82	23,04	15,65	34,01	16,33	34,01

[a] Valadoc, http://valadoc.org/, 50.709 LoC
[b] Vala, http://vala-project.org/, 236.071 LoC
[c] Drupal Search API, http://drupal.org/project/search_api, 21.696 LoC
[d] TapiJI, http://code.google.com/a/eclipselabs.org/p/tapiji/, 19.611 LoC
[e] MyLyn, http://eclipse.org/mylyn/, 76.464 LoC
[f] DeltaSpike, http://deltaspike.apache.org/, 35.202 LoC

A discussion that was brought up by Florian Brosch during his classification run was the ambiguity of *Usability* improvements. Even he, as a co-author of this paper, had troubles to determine whether a change was solely *perfective* or *adaptive* or both. The second scheme almost always resulted in *Functionality* and *Usability*. The third scheme is difficult as well, since it may be constructed that a *Usability* framework may be implemented without any features. In this case it might be considered *forward-engineering*, however if previously features existed, it may be argued that the same feature could now be considered *re-engineering*.

IV. CONCLUSION

The data set we provide contains 967 classified commits. Every commit has been enriched with meta-information whether it addresses functionality requirements or non-functional requirements. Further, a fine-grained mapping for non-functional requirements addressed by a change is provided, as well as whether the change was refactoring related or a repository management elicited change. The provided meta-information allows to evaluate existing labeling and classification approaches, be it machine learners or otherwise generated schemes. More importantly though, the information may be used to train new classification techniques, since all necessary information to match the survey to the existing project repositories is available. Even more complex approaches that leverage not only code repository information but also e.g. bug tracking systems may be applied, due to the available author information.

The data is provided as an SQLite database and can easily be imported or statistically analyzed (e.g. by using RSQLite[8]). The model of the data is simple and intuitive and may easily be integrated into existing approaches that leverage more than one repository type (e.g. by using the user table to match code repository and bug tracker system users). Since one of the authors of the paper classified his own changes, we suggest to treat his data set differently, since his more thorough knowledge of the classification mechanisms might introduce bias.

[8] http://cran.r-project.org/web/packages/RSQLite/index.html

As for the classification schemes, developer acceptance was high. Developers did not require further explanation of the schemes beyond the initial instructions and the guidance session. Three preferred to step together through the first changes, while the rest did not. Overall though, feedback on all three schemes was that the schemes felt coherent and applicable.

ACKNOWLEDGMENT

The authors would like to thank Martin Reiterer, Markus Strutzenberg, Kilian Matt, Evan Nemerson, Luca Bruno and Thomas Seidl for their time to perform our survey.

REFERENCES

[1] IEEE Standard Glossary of Software Engineering Terminology, 1990.
[2] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement? In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research meeting of minds - CASCON '08*, page 304, New York, New York, USA, October 2008. ACM Press.
[3] Ying Fu, Meng Yan, Xiaohong Zhang, Ling Xu, Dan Yang, and Jeffrey D. Kymer. Automated classification of software change messages by semi-supervised Latent Dirichlet Allocation. *Information and Software Technology*, June 2014.
[4] Lile P. Hattori and Michele Lanza. On the nature of commits. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, pages 63–71. IEEE, September 2008.
[5] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. pages 392–401, May 2013.
[6] Abram Hindle, Neil A. Ernst, Michael W. Godfrey, and John Mylopoulos. Automated topic naming to support cross-project analysis of software maintenance activities. In *Proceeding of the 8th working conference on Mining software repositories - MSR '11*, page 163, New York, New York, USA, May 2011. ACM Press.
[7] Andreas Mauczka, Markus Huber, Christian Schanes, Wolfgang Schramm, Mario Bernhart, and Thomas Grechenig. Tracing your maintenance work–a cross-project validation of an automated classification dictionary for commit messages. In *Fundamental Approaches to Software Engineering*, pages 301–315. Springer, 2012.
[8] A Mockus and L Votta. Identifying reasons for software changes using historic databases. *Software Maintenance, 2000. Proceedings. International Conference on*, pages 120 – 130, Sep 2000.