Mining Version Control Systems for FACs (Frequently Applied Changes)

Filip Van Rysselberghe and Serge Demeyer
Lab On Re-Engineering
University Of Antwerp
Middelheimlaan 1
filip.vanrysselberghe@ua.ac.be

Abstract

Today, programmers are forced to maintain a software system based on their gut feeling and experience. This paper makes an attempt to turn the software maintenance craft into a more disciplined activity, by mining for frequently applied changes in a version control system. Next to some initial results, we show how this technique allows to recover and study successful maintenance strategies, adopted for the redesign of long–lived systems.

1. Introduction

As stated by Lehman's 1st law of software evolution, a system has to undergo continuous change in order to remain satisfactory for its stakeholders [8]. Adding new features, correcting faults and accommodating for new changes, are generally considered the prime reasons for these changes [12].

Unfortunately, the changes applied during maintenance are seldom documented. Even for refactorings –currently the best known approach towards a systematic catalogue of maintenance tasks–, there is no indication which refactoring is most suitable for a certain situation. This observation can easily be illustrated by looking through Fowler [4], which is considered standard work on the subject and counting the number of conditional sentences. Knowledge on which changes are most appropriate for an occurring problem or situation is therefore private to experienced software maintainers.

By making this knowledge general, software maintenance can be improved and lose its status of ad hoc discipline. In order to meet this goal, we propose a technique analogue to the idea of frequently asked questions or FAQs. These FAQs are summaries of frequent questions and corresponding answers to reduce the continual posting of the same basic question. Analogue, we propose to identify frequently applied changes (FACs) since these changes record general solutions to frequent and recurring problems.

To detect such frequently applied changes, a technique based on clone detection is used. Due to their central position in modern development processes and their ability to record a project's entire change history, versioning systems contain a wealth of change information. Therefore the data for the detection process is provided by a versioning system.

In the remainder of this paper we will introduce the technique, evaluate it and position it in a broader context. The first section is reserved for introducing the technique (section 2). Afterwards, the results of an initial case study to evaluate the technique are discussed in section 3. Section 4 on the other hand, explores how the resulting change sets can be used to study software maintenance. Our future directions are discussed in section 5. The last section (6) discusses related work.

2. The Technique

With our technique we want to focus on how a system changed during its maintenance. Therefore we are interested in systems which are able to tell which changes were made. Typically, version or change management systems offer this functionality.

Such versioning systems like CVS, ClearCase and SourceSafe, can be considered as a large source code repository containing all the versions of a program. However internally, most versioning systems store the entire source of one version only. CVS for example, stores the last version entirely since that is the most likely version to be checked out for additional editing. Other versions are re–constructed by means of delta's, relative to the one complete version. For CVS, these delta's record which lines have to be added, deleted or changed in order to get a previous version. Since these delta's record which changes were made, we can ex-

tract change information from a versioning system.

In practice, extracting change information from a versioning system is not difficult as we found out by our initial study. For this study we targeted the CVS versioning system since it is used for many successful open–source projects, providing a lot of changes to study. Using proper CVS commands, change information can be extracted and afterwards processed. We combined the "cvs log" and "cvs diff" commands to extract change data like the difference in code before and after the change, the date and time of the change, the file involved etc. Since this basic change data is stored in about any versioning system, other versioning systems than CVS can be used as well.

Being able to extract change information from a versioning system is only part of the technique. Processing the changes in order to locate frequently applied changes is therefore the second step in our technique. However, first, we define a frequently applied change as a change to the code which occurs multiple times in the evolution of a system. Since a source code level is targeted, two applied changes are equal if there is a similarity relation between the delta of both changes. Locating frequently applied changes therefore corresponds to identifying sets of similar code fragments.

A possible approach to locate identical and similar code fragments is by using clone detection techniques. Clone detection techniques are developed to identify duplicated or cloned code fragments within a program since duplication may hinder the program's evolution. Over the years different techniques are proposed to locate clones or fragments which share the same code but may differ in the naming of identifiers. Ducasse for example, proposes a detection technique to locate clones containing a certain amount of identical lines [2]. Baker on the other hand focusses on code fragments in which identifiers, which are likely to change during the duplication process, may differ as long as there is a one to one mapping between the identifiers [1]. Since these techniques are developed to locate similar code fragments in a scalable way, we use them to identify frequently applied changes.

By using clone detection techniques on the changes extracted from the versioning system, we are therefore able to identify frequently applied changes.

In our initial study, we started by extracting from the repository all changes made during the lifetime of a product. For each change the corresponding delta, which consists of the code before and after the change, was added to one, general text-file. This text–file was later analyzed by Kamiya's clone detection tool CCFinder [7] to locate recurring, similar changes. Since a similar change recurs a couple of times in the change history, it correspond to a frequently applied change or FAC.

3. Evaluating the technique

To evaluate the technique, we applied it to study a successful open–source system called Tomcat. After more than three years of development, Tomcat is considered the standard Java servlet container. Ever since the beginning, participating developers can contribute to the project by accessing a central CVS versioning system. Therefore the CVS system contains a wealth of maintenance information under the form of changes. Combined with the knowledge that the project experienced some major redesign phases, Tomcat shows an interesting case to explore the techniques possibilities.

3.1. Finding refactorings using clone detection

In a previous study, we showed how clone detection can be used to detect refactorings between two versions [11]. We observed how scalability hinders the application of this technique since comparing all successive versions of a large system is not feasible. The scalability problem was a direct result of comparing whole systems rather than the changes made between two versions. Since the technique proposed in this paper focusses on the changes, it does not suffer this problem, yet is able to detect similar refactorings.

3.2. Finding FACs

The goal of our initial study was to explore the kind of frequent change sets that can be formed when a parameterized clone detection technique as CCFinder is used [7]. Due to their focus on the detection of similar code fragments, parameterized clone detection techniques are expected to produce the most suitable FACs.

CCFinder is a token based detection technique which searches a specially constructed tree for maximal matches. Due to its token based nature, the detection process is not influenced by the code layout. Crucial in its detection process is the definition of a length threshold. This threshold defines the minimal number of tokens that should match before a token sequence is considered a match. Matches which fail to meet the length threshold, are not considered matching fragments. Increasing the threshold, therefore makes the detection of both positive as false matches less likely. In our study, the impact of this parameter was taken into account by exploring various thresholds. Based on the visual representation of the detected matches, we manually verified the relation between repeatedly found changes. For these matching changes make up a set of frequently applied changes.

The sets of identical changes, constructed with a high threshold value, are small sets of long changes which are almost identical. Accidentally matching, long changes are

less likely when a high threshold value is used. Therefore the changes which are identified as (frequently) recurring changes are related to each other by a copy relation. As we observed in the initial study, there are three possible causes for such relation to exist:

- A first reason is the introduction of duplicated code. Based on former experience, the maintainer applies a previously used solution in a different location. In our study, we for example noticed how the same exception handling code was introduced in different places. However, the maintainers were aware of the problems related to duplicating code since a few versions later, the duplicated statements were replaced by a function.
- Repositioning a code fragment is a second cause for the introduction of copy related changes. Moving code from one class to another results in deleting and adding the same piece of code at various locations. In our study, we for example observed how an utility function of RequestUtil was moved into the Parameter class.
- Temporarily adding a code fragment, is the last cause we observed. We noticed how many of the fragments added in one version, are deleted later, causing a copy relation between the addition and delete. As figure 1 illustrates, many similarities between add and delete changes exist.

Figure 1. As each dot in the plot corresponds to a similarity between an add and a delete change of at least 40 tokens, it illustrates the relation between both categories

High threshold values therefore allow the identification of recurring, product specific changes. By establishing the motivation behind these changes more general maintenance conclusions can be derived. For example we might learn why code is duplicated or when a temporary solution can be suitable. Also note that we used a simple comparison scheme, comparing any part of the delta, with any other delta. A comparison scheme in which only the pre-change code of delta's is compared, would not find many of these copy related changes. For high threshold values, it is therefore better to use a simple comparison scheme which just compares anything.

Low threshold values on the other hand, lead to the identification of frequently applied, generic changes. Due to the reduced impact of the threshold on accidental matches, more changes which are only accidentally syntactically identical, are detected. This leads to sets of frequently applied changes which correspond to low level code changes as e.g. changing a function call or changing a variable. In our case study, we for example had a set of package definition changes, while another set bundled an extensive set of import statement changes. However before low threshold detection can be used in practice, two problems should be solved.

When no special precautions are taken, different kinds of generic changes are identified as one frequently applied change. Since clone detection only demands that the syntactical structure of two fragments matches, changes sharing an identical structure, yet differing in semantics, are wrongly classified as identical changes. Figure 2 which serves as an example, illustrates how renames of function calls and variables are structurally identical. This problem can be solved by taking more specific change information into consideration.

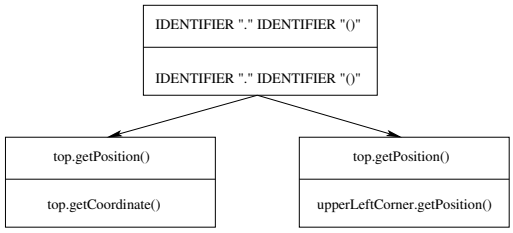

Figure 2. This figure shows how two generic changes can be identical from a syntactical perspective. In this example, renaming a function–call (left) and renaming a variable (right) share the same structure (above), however differ semantically. In the syntactic representation, non-terminals are quoted.

A similar problem was caused because many changes contain similar sub-units which match with each other. In our simple evaluation setting, all changes were added to one text-file which was afterwards analyzed for the presence of similarities. However this allows a statement, part of a larger change, to match a statement of an other change. To avoid this problem, the comparison of changes has to be

done in a more intelligent way. Identical changes are then no longer changes which just share a similarity, but changes in which both the code before the change as the one after the change are similar. In case of a low threshold detection process, more care has to be taken when comparing changes. The frequently applied changes are however more generic and can be used to automatically find generic maintenance strategies.

4. Study of frequent changes

Finding frequently applied changes on itself does not solve any maintenance issues. Each frequently applied change is rather a building block to identify generic maintenance strategies. Based on the kind of sets formed, different approaches can be taken.

Frequently applied changes identified with a high threshold and are therefore specific to one product, can be used as a starting point to study the motivation and success of a change. Afterwards this study may be generalized in a (set of) general rule. The motivation behind the removal of a recently added code fragment in a product, may teach us for example, why a change may fail in general. Similarly, studying the duplication of a solution, may point us to general problems or teach us how duplication grows, which in turn allows us to improve design patterns to cope with this duplication. However, due to their tight product coupling, these high threshold FACs can be used to improve and understand the current product as well.

The generic FACs found with a low threshold on the other hand, can be used to derive maintenance strategies automatically. In such automatic process, FACs are used as building blocks in a data mining process to identify frequent change patterns. All changes, classified according to their change set, are added to a kind of change transaction database. After composing such change database, a data mining process searches for frequent change patterns. However we can not just search all the changes for recurring patterns since change strategies are composed of both generic changes and relations between them. Therefore related changes, for example two changes which share an identical code fragment or because one change introduces a function used by the other, are considered as one change transaction. By comparing these various change transactions, recurring patterns are identified. These change patterns allow us to find generic maintenance operations like the refactorings that are currently described in literature [4]. By evaluating the situation in which these maintenance operations are used, we also might identify when and why they should be applied.

In turn, these maintenance operations can be considered as frequently applied changes and used to identify even higher level change strategies. One possibility is to redefine the data mining process to these higher level FACs. In this redefined process, a transaction groups all the change patterns applied in one version. Change patterns identified by such process may show us how a generic problem situation is removed by performing a sequence of lower level maintenance operations. Software maintainers can use this knowledge when handling a similar problem.

An other possibility is to relate changes with bug reports as proposed by Fischer [3]. By combining this information, frequently occurring bugs as well as their solutions may be identified. Consider for example a case where many bug reports are related to changes belonging to the "change loop condition" FAC. This would empirically show that looping conditions tend to lead to errors and therefore should be thoroughly tested. Similarly changes could be linked with maintenance reports or feature requests, to identify requests with similar solutions. This would not only allow maintainers to apply requests faster, but also helps designers to anticipate changes better.

Frequently applied changes, allow us to start with a set of small, generic changes and incrementally build up a whole set of maintenance operations which enrich our current software maintenance knowledge with wide-spread operations and techniques.

5. Future Directions

In the immediate future, we want to further explore the technique, introduced in this paper. This includes further evaluation of the suitability of various clone detection techniques and clone detection in general. For example, we want to study how the low threshold related problems (3.2) can be reduced or even removed.

Next to improving the technique through evaluation, we will apply it to study software evolution. In this context, we plan to evaluate the techniques and ideas presented in section 4. By means of these techniques, we want to investigate various projects to come to general maintenance strategies.

6. Related Work

To our knowledge, little effort has been spent to compose sets of frequently occurring changes and use those to study software maintenance. One could argue that a runtime change classification scheme as the one by Gustavsson [5], fits this context too since the items in such classification correspond to sets of changes. However FACs work on different levels and work from an exploratory perspective. Furthermore, the technique allows to automatically categorize changes based on any classification scheme.

In the context of studying changes, work has been done to identify the motivation for a change [12]. The goal was

verifying whether a change was made to fix a problem, prepare for future change or to insert new user functionality. Related with this work, Mockus used word frequency analysis of log messages to not only identify the purpose of changes, but relate it to change size and time between changes as well. Mockus and De Hondt, who both studied change log information, state that a textual description of a change is necessary to understand the real motivation behind a change [10, 6]. Our technique, does not differ from this point of view. Change information should be linked with other maintenance information to fully understand the motivation behind these strategies.

Concerning the detection of patterns using data mining, there is some relation to Michail [9] who detects reuse patterns based on a data mining approach. We propose a similar approach to detect change patterns over different versions.

References

[1] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Computing*, 26(5):1343–1362, October 1997.

[2] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings of Int. Conf. on Software Maintenance (ICSM)*, pages 109–118. IEEE Computer Society, September 1999.

[3] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings Int. Conf. on Software Maintenance (ICSM)*, pages 23–32. IEEE Computer Society, September 2003.

[4] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] J. Gustavsson. A classification of unanticipated runtime software changes in java. In *Proceedings Int. Conf. on Software Maintenance (ICSM)*, pages 4–12. IEEE Computer Society, September 2003.

[6] K. D. Hondt and P. Steyaert. Exploiting classification for software evolution. In *ECOOP 2000 Workshop on Objects and Classification*, 2000.

[7] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multi-linguistic token-based code clone detectionsystem for large scale source code. *IEEE Trans. Software Engineering*, 28(7):654–670, July 2002.

[8] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[9] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd Int. Conf. on Software Engineering*, pages 167–176. ACM Press, 2000.

[10] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings Int. Conf. on Software Maintenance (ICSM)*, pages 120–130. IEEE Computer Society, October 2000.

[11] F. V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In M. W. G. Tommi Mikkonen and M. Saeki, editors, *Proceedings Int. Workshop on principles of software evolution (IWPSE)*, pages 126–130. IEEE Computer Society, September 2003.

[12] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd Conf. On Software Engineering*, pages 492–497, 1976.