Discriminating Development Activities in Versioning Systems: A Case Study*

Juan Jose Amor, Gregorio Robles, Jesus M. Gonzalez-Barahona, Alvaro Navarro
GSyC, Universidad Rey Juan Carlos (Madrid, Spain)
{jjamor,grex,jgb,anavarro}@gsyc.escet.urjc.es

Abstract

When characterizing the coding activity of developers on a software project, a common technique is to study the transactions in the versioning system. However, not all transactions are equal, neither represent the same kind (both in quality and quantity) of activity. In this paper, a methodology for classifying the interactions of developers with versioning systems is proposed, based on the analysis of the textual descriptions (comments) attached to each transaction. Some results of applying this methodology to a specific case (the FreeBSD CVS repository) are presented, alongside with some ideas about how to use this technique for improving effort estimation and other research lines.

1. Introduction

Analyzing the activity of developers in software projects can be of interest for several reasons, being effort estimation one of the most clear case studies. Measuring activity can be difficult in the absence of timesheets or similar information. In libre software[1] projects this is usually the case, but fortunately, other sources of information are usually available to infer activity. Revision control systems, mailing list archives and bug tracking systems, for example, contain a huge amount of data about developer activity in several fields [1]. These data are generally publicly available and can be collected and analyzed in order to obtain a detailed view of the activity of developers.

In the specific case of versioning systems, the whole history of modifications to the code is available, including the changes for each modification, who performed it, and when it was done. Attached to each modification, a textual description of the change is also available. For many years, research groups in several domains have been using this information, although most of them have ignored the textual descriptions of the changes, probably because of their lack of structure.

Some authors, however, have used this information to characterize each modification, especially code transactions[2]. So, Mockus et al. [7] proposed a methodology for classifying code transactions based on the analysis of textual descriptions. They normalized the texts (that is, converted them to lower case, removed the most common words as they do not add semantic information, and extracted the stem of each word) and performed word frequency analysis and keyword clustering to establish a set of rules in order to classify each modification into one of the three types of maintenance (corrective, adaptive and perfective) [9]. German [4] also used descriptions to analyze maintenance (bug fixing) and comment modifications. Although a good starting point, these approaches have severe limitations if our intention is to have a finer-grained classification that allows to distinguish among the various types of corrective, adaptive and perfective maintenance activities.

In this paper, we present a methodology that uses different techniques (such as Bayesian analysis) to obtain a fine-grained classification of code transactions from their textual description, and provide some results of having applied this methodology on a case study. In the next section, we will introduce our detailed classification of code transactions. The third section contains the description of the methodology and the forth one, results from applying it on FreeBSD. Finally, conclusions are drawn, limitations are pointed out and hints for further research are suggested.

*This work has been funded in part by the European Commission, under the CALIBRE CA, IST program, contract number 004337.

[1] Through this paper the term "libre software" will be used to refer to code that conforms either to the definition of "free software" (according to the Free Software Foundation) or of "open source software" (according to the Open Source Initiative).

[2] A code transaction is an atomic commit which changes at least one source code file [4]. Another term for code transactions that can be found in literature is modification record (in general, in response to a modification request).

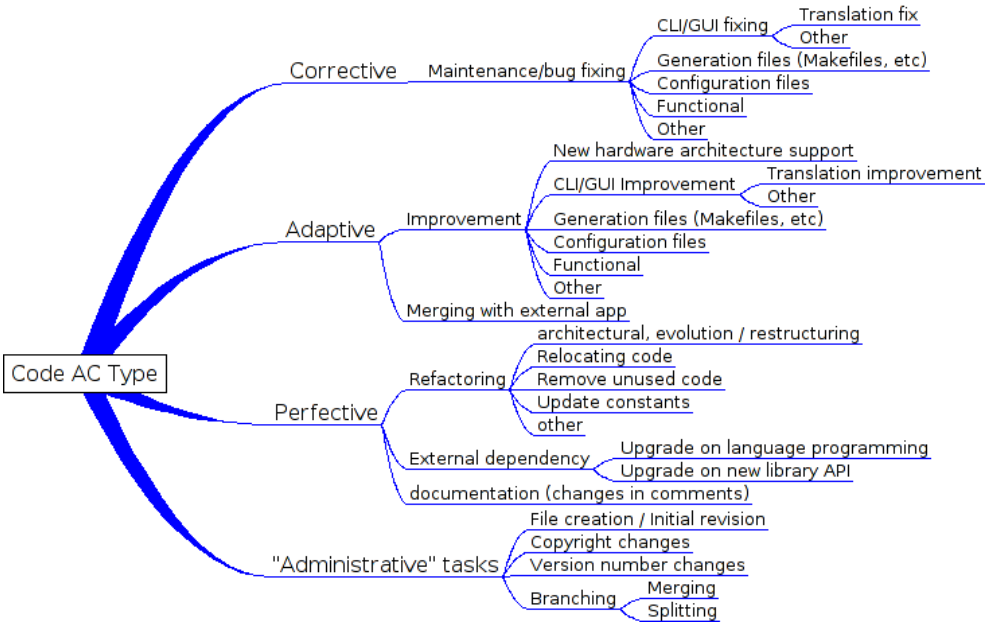

Figure 1. Classification schema for code transactions

2. Classification of transactions

Since we were interested in identifying kinds of transactions with a finer grain than in other studies, we first designed a schema for classifying code transactions (see figure 1). The starting points of the schema are the already-mentioned 'classical' types of maintenance (corrective, adaptive and perfective) [9]. In addition, a fourth type has been added (administrative). There we include several kinds of transactions usually performed not by developers, but by CVS[3] administrators, such as initial file creation, administrative changes in version number, administrative branch merging or splitting, among others.

With this classification in mind, we examined the CVS systems of several libre software projects, identifying subtypes for each type. Since the goal in the long term of our study is using activity to estimate effort, we considered specially those types which could have different impacts on effort. At the same time, we were interested in subtypes relevant in most projects, and not specific to one of them. This lead us to the following subtypes:

- Corrective transactions were classified according to the kind of fix or maintenance activity: CLI/GUI[4] (fixes which affect only to the user interface, either textual or graphical, and not to application functionality), functional (fixes to functional code), generation (fixes on makefiles, etc), configuration (fixes on 'config' files, etc.) and other (for instance, a spelling fix on a comment).
- Adaptive transactions were first classified in improvements (modifications for implementing new features) and merging of code from other projects (with the purpose of adding functionality)
- Perfective transactions were split in code refactoring, external dependency and perfective changes in code comments.
- Administrative tasks were classified in file creation (when a new file is created and cannot be classified in the previous categories), changes in copyright information (including license), change in version number, and merging/branching (eg., when the testing branch is merged on the stable one).

Many of these subtypes have also been divided into several categories as shown in figure 1. It is important to notice that this classification is not complete and can

[3] Although it is being phased out by other newer systems, CVS is still the most used source code management system in libre software projects. It is also the one used in the case study shown in this paper.

[4] CLI stands for Command Line Interface, while GUI is the acronym of Graphic User Interface.

be improved by studying more projects. However, it is good enough for the purposes of testing our classification methodology.

3. Methodology

The methodology we propose to classify the transactions can be summarized as follows:

1. We download information about all commits from the CVS repository, using the CVSAnalY tool [8, 5]. As a result of this process, code transactions are identified and stored in a database with all the information related to them.

2. We then classify the transactions in the database according to the textual information, using the *naive Bayes* classifier (in several steps, as will be detailed below). As a result, each transaction is classified into a type with a certain probability.

More in detail, CVSAnalY works by analyzing the CVS log entries, and grouping commits in code transactions (using the algorithm described in [4]). For each identified transaction, rich information is stored (file type, version, commiter, time, textual description, etc.). Therefore, it produces a database with an exhaustive description of the CVS repository, in a structured format.

For the classification of code transactions, the most popular text classification method based in computer learning is run on the textual description of each. It is the *naive Bayes* classifier [3], which is based in Bayes' theorem. Although it requires that the set of attributes are conditionally independent for each class, it usually performs well enough in real cases (for instance, it is currently used in many junk mail automatic filters [2]). The actual software used for running this algorithm is the Bow toolkit [6], a system implementing several text classification algorithms, being *naive Bayes* one of them.

The process starts with an expert classifying a very small, randomly selected fraction of the transactions identified by CVSAnalY (using a database editor). Then, a script produces files in the format required by *rainbow*, the main program of the Bow toolkit. *rainbow* learns from these files, producing its own database. Using other scripts, the textual information for all the unclassified transactions are given to *rainbow*, which classifies them with a certain probability. Both the decision of the classifier and the probability are stored in our database.

This first automatic classification was considered poor, since it produced a high fraction of wrongly classified transactions. Therefore, a refinement step was performed after it. A random sample of the transactions classified with lower probability, or which lead to a wrong classification, was selected, and classified again by an expert. The resulting information improved *Bow*'s Bayesian classification capabilities.

After that, all the transactions in the database were classified again, using *rainbow*. In the cases we have studied, this second run has given results which we have considered good enough: an average above 60% for the probabilities estimated by the Bayesian classifier for the more likely classification of each transaction, and also a verification by an expert of a random sample, with a success rate over 70%. If these percentages are not enough, further refining steps could be performed.

4. Case study: FreeBSD

We have tested the described methodology with the CVS repositories of several libre software projects. To illustrate how it works, we present here the process of applying it to FreeBSD, a complete operating system, for which both kernel and user-land utilities (module *src* of the CVS repository) are held in the same CVS repository, using a multi-branch approach:

- FreeBSD-CURRENT (HEAD) includes work in progress, experimental changes, and transitional mechanisms that might or might not be present in the next official release of the software. This branch is generally associated with major version numbers.

- FreeBSD-STABLE contains the more conservative code which generally have first gone into -CURRENT.

Around 4.x, FreeBSD introduced a third class of branch, a "release branch", used for a number of things, including the initial release engineering, security patches, and errata patches. Therefore, a schema of the FreeBSD branch tree is as follows:

```
HEAD              CVS HEAD
  RELENG_6        6-STABLE devel branch
    RELENG_6_0    6.0 eng branch
  RELENG_5        5-STABLE devel branch
    RELENG_5_0    5.0 eng branch
    ....
    RELENG_5_5    5.5 eng branch
  RELENG_4        4-STABLE devel branch
```

```
RELENG_4_11   4.11 eng branch
...
```

We have performed a separate analysis of each CVS branch, storing the extracted information in separate databases. Table 1 provides some general statistics about the branches under study. Even though the following study considers only branches, the first row of the table shows all the modifications performed on the versioning system, including those files that do not belong to any branch or that have been removed[5].

Tag	ACs	Commits	Commiters
–	139190	533806	440
HEAD	9052	36404	290
RELENG 6.1	101	175	37
RELENG 6.0	89	229	34
RELENG 6	2417	7565	142
RELENG 5.5	19	23	5
RELENG 5.4	140	285	36
RELENG 5.3	94	270	19
RELENG 5.2	167	318	38
RELENG 5.1	65	141	22
RELENG 5	231	12899	164
RELENG 4.11	96	249	28
RELENG 4.10	74	198	18
RELENG 4	13742	57358	263
RELENG 3	3530	18813	140

Table 1. CVS branches in FreeBSD.

From the analysis of the various branches, a database with a total number of 33,335 code transactions was obtained, from 1993-06-18 (in the HEAD branch) to 2006-05-29 (also in HEAD).

The starting random set of transactions classified by an expert amounted to 300 (for all the branches). After the first *naive Bayesian* classification, the average probability of the most likely type for each transaction was of about 50%. For the second refining step, a sample of 100 more transactions were classified by an expert. This step raised the average probability to above 65%.

After that, we did an automatic classification of each branch and discovered that the average probability of success in classification was low (around 50%), so we selected a second random set of 100 transactions classified with low (under 40%) probability and classified again them manually. We also examined random sets of records and, when we observed bad classification, we reclassified them. After feeding these newly classified records to the Bow toolkit, the average probability of success went up to around 65%, and the classification was correct in more than 70% of the cases for a random sample verified by an expert (this sample was of 30 transactions for each branch).

In fact, it was verified that most of the wrongly classified records were too ambiguous, even for a human. For example, while a text such as "Fixed range address bug" was clearly classified as a bug fix, texts such as "beforeinstall - SCRIPTS" or "mdoc(7) police: sweep" or "Cosmetics" are clearly ambiguous and cannot be classified (without inspecting the actual code changed). The classification of those transactions requires the consideration of other parameters, such as list of files committed or the specific lines added or changed, which are out of the scope of this paper.

The results shown in table 2 consider transactions that have been classified with a probability of success over 70% (as estimated by the Bayesian classifier). About 25% of the transactions in the HEAD branch were classified as corrective activity (maintenance), while 25% were classified as adaptive/improvement activity, almost 50% as perfective/refactoring, and the rest (less than 2%) as administrative, adaptive/merging or perfective/external dependency. Our approach is especially limited when classifying administrative transactions. This is because these transactions are not common and their textual description is not very rich. On the other hand, those administrative transactions which are difficult to identify with this approach contain generally a large amount of files, so including this information in our methodology could give better results.

	HEAD	RELENG_6	RELENG_6_1
Corrective/Bug fix	25,87	59,04	88,05
Adaptive/Improvement	25,17	16,63	7,46
Adaptive/Merging	0,32	1,07	0
Perfective/Refactoring	47,00	22,96	2,99
Perfective/Ext. Dep.	1,54	0,08	1,49
Perfective/Doc.	0	0	0
Admin/File C.	0,08	0,23	0
Admin/Copyright	0	0	0
Admin/Versions	0	0	0
Admin/Branching	0	0	0

Table 2. Classification for three branches (probability of success: 70%).

Although with a minor number of transactions, other branches are shown in table 2. For comparison, if we considered all the development branches aggregated (RELENG_N), over 50% of the transactions correspond to maintenance, around 35% to perfective/improvement

[5] Actually, there is no file removal in CVS; *removed* files are located in the attic and can be restored anytime.

and less than 20% to adaptive activities. For release engineering branches (RELENG_N_M), higher values of corrective activity (75% to 90%) are obtained.

This different behavior for the different types of branches is no surprise. In release engineering branches most activity is focused on critical bug fixing and security issues, while most improvement activity is done in HEAD branch. Some checks (inspection by an expert) have verified that, in fact, some improvement activity is merged in release engineering and development branches from HEAD, and also critical bug fixes are applied, not only to all supported stable branches but also to the main HEAD branch. Another observed behavior, also expected, is that corrective transactions tend to be smaller (in number of commits) than other types (see figure 2).

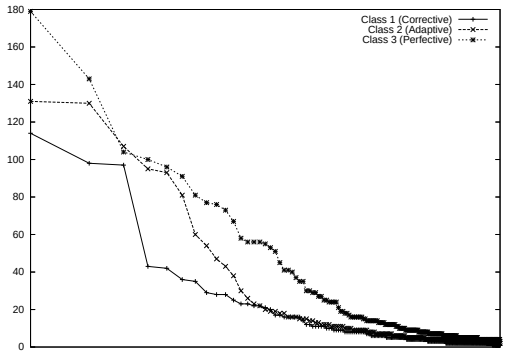

Figure 2. Number of files involved for several types of transactions (HEAD branch).

5. Conclusions

In this paper we have proposed a set of development activities in which we can classify code transactions in versioning systems. We have therefore used an approach that is automatic, although human-aided, that does not rely on the manual classification of keywords which shows to be only usable when classifying a small set of disjunctive activities. We have tested the methodology with the CVS branches of the FreeBSD project and have observed some preliminary behaviors that indicate that this approximation is very promising.

However, we also have identified a set of limitations. As our approach is text-based and depends on training, it does not work well with ambiguous texts or with very infrequent cases. A possible solution for this is to complement the analysis with other available data about the transaction, such as its size or its most common file type.

An interesting future line of research could be to apply this methodology on other information exchange tools used in development such as mailing lists and bug-tracking systems, where natural language is also common and makes automatic classifications difficult. This could extend the main idea behind this paper, that we need to classify various types of activity in order to weight activities different for the estimation of effort [1], to other data sources.

6. Acknowledgments

We thank Robert Watson (member of FreeBSD, now at the University of Cambridge), for his invaluable help in the interpretation of the FreeBSD case study.

References

[1] J. J. Amor, G. Robles, and J. M. Gonzalez-Barahona. Effort estimation by characterizing developer activity. In *8th Intl. Workshop on Software Engineering Economics*, pages 3–6, Shanghai, China, May 2006.

[2] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, G. Paliouras, and C. D. Spyropoulos. An evaluation of naive bayesian anti-spam filtering. In *Workshop on Machine Learning in the New Information Age*, pages 9–17, June 2000.

[3] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2 - 3):131–163, November 1997.

[4] D. M. Germán. An empirical study of fine-grained software modifications. In *Proc Intl Conference in Software Maintenance*, Chicago, IL, USA, 2004.

[5] B. Massey. Longitudinal analysis of long-timescale open source repository data. In *Proc Intl Workshop on Predictor Models in Software Engineering (PROMISE 2005)*, St.Louis, Missouri, USA, 2005.

[6] A. K. Mccallum. Bow: A toolkit for statistical language modeling, 1996. http://www.cs.cmu.edu/˜mccallum/bow.

[7] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proc Intl Conf Softw Maintenance*, pages 120–130, October 2000.

[8] G. Robles, S. Koch, and J. M. González-Barahona. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. In *Proc 2nd Workshop on Remote Analysis and Measurement of Software Systems*, pages 51–56, Edinburg, UK, 2004.

[9] E. B. Swanson. The dimensions of maintenance. In *Proc 2nd International Conference on Software Engineering*, pages 492–497. IEEE Computer Society Press, 1976.