The Impact of Version Control Operations on the Quality Change of the Source Code

Csaba Faragó, Péter Hegedűs, and Rudolf Ferenc

University of Szeged Department of Software Engineering
Árpád tér 2. H-6720 Szeged, Hungary
{farago,hpeter,ferenc}@inf.u-szeged.hu

Abstract. The number of software systems under development and maintenance is rapidly increasing. The quality of a system's source code tends to decrease during its lifetime which is a problem because maintaining low quality code consumes a big portion of the available efforts. In this research we investigated one aspect of code change, the version control commit operations (add, update, delete). We studied the impact of these operations on the maintainability of the code. We calculated the ISO/IEC 9126 quality attributes for thousands of revisions of an industrial and three open-source software systems. We also collected the cardinality of each version control operation type for every investigated revision. Based on these data, we identified that operation Add has a rather positive, while operation Update has a rather negative effect on the quality. On the other hand, for operation Delete we could not find a clear connection to quality change.

Keywords: Software Maintainability, Software Erosion, Source Code Version Control, ISO/IEC 9126, Case Study.

1 Introduction

Software quality plays a crucial role in modern development projects. There is an ever-increasing amount of software systems in maintenance phase, and it is a well-known fact that software systems are eroding [14], meaning that in general their quality is continuously decreasing due to the ever-ongoing modifications in their source code, unless explicit efforts are spent on improvements [3]. Our aim is to check the connection between the developers' interactions and the quality change, in order to identify which patterns typically increase, and which decrease code maintainability.

Our longer term plan is to discover as much of these patterns as possible. In the beginning we focus only on the data found in the version control systems. Later we plan to include other available data, especially micro interactions performed within the IDE during development, and data found in issue tracking systems. Based on the results, we can hopefully formulate advices to software developers on how to avoid the maintainability decrease. Furthermore, this could also help to better allocate efforts spent on increasing quality.

In this research we focus on the *version control operations*. Specifically, we check the effects of file additions, updates and deletions on the maintainability

B. Murgante et al. (Eds.): ICCSA 2014, Part V, LNCS 8583, pp. 353–369, 2014.
© Springer International Publishing Switzerland 2014

of the source code. We checked how the higher number or higher proportion of a version control operation within a commit typically affects maintainability. Basically, we assumed that file additions have positive impact, as they introduce new, clean, reasoned code. Initially we expected the same result also for file deletions, as this operation is typically used during code refactoring, which is an explicit step towards better maintainability. On the other hand, we expected that file updates tend to decrease maintainability.

To summarize our goals, we formulated the following research questions:

RQ1: *Does the amount of file additions, updates and deletions within a commit impact the maintainability of the source code?*

RQ2: *Are there any differences between checks considering the absolute number of operations (Add, Update, Delete) and checks investigating the relative proportion of the same operation within commits?*

The paper is organized as follows. Section 2 introduces works that are related to ours. Then, in Section 3 we present the methodology used to test the underlying relationship between version control operations and maintainability changes. Section 4 discusses the results of the performed statistical tests and summarizes our findings. In Section 5 we list the possible threats to the validity of the results, while Section 6 concludes the paper.

2 Related Work

As a particular software quality related activity, refactoring is a widely researched field. Lots of works build models for predicting refactorings based on version control history analysis [18–20]. Moser et al. developed an algorithm for distinguishing commits resulted by refactorings from those of other types of changes [13]. Peters and Zaidman investigated the lifespan of code smells and the refactoring behavior of developers by mining the software repository of seven open-source systems [15]. The results of their study indicate that engineers are aware of code smells, but are not really concerned by their impact, given the low refactoring activity.

There are works which focus on the effect of software processes on product quality [11]. Hindle et al. deal with understanding the rationale behind large commits. They contrast large commits against small commits and show that large commits are more perfective, while small commits are more corrective [8]. Bachmann and Bernstein explore among others if the process quality, as measured by the process data, has an influence on the product quality. They showed that product quality – measured by number of bugs reported – is affected by process data quality measures [4].

Another group of papers focus on estimating some properties of maintenance activities (e.g. the effort needed to make changes in the source code, comprehension of maintenance activities, complexity of modification) [7, 12, 21]. Tóth et al. showed that the cumulative effort to implement a series of changes is larger than the effort that would be needed to make the same modification in

only one step [21]. The work of Gall et al. focuses on detecting logical couplings from CVS release history data. They argue that the dependencies and interrelations between classes and modules that can be extracted from version control operations affect the maintainability of object-oriented systems [6]. Fluri et al. examine the co-evolution of code and comments as a vital part of code comprehension. They found that newly added code – despite its growth rate – barely gets commented; class and method declarations are commented most frequently, but e.g. method calls are far less; and that 97% of comment changes are done in the same revision as the associated source code change [5]. Unlike these works, we do not use the version control data to predict refactorings or software quality attributes, but to directly analyze the effect of the way version control operations are performed on software maintainability.

Atkins et al. use the version control data to evaluate the impact of software tools on software maintainability [1]. They explore how to quantify the effects of a software tool once it has been deployed in a development environment and present an effort-analysis method that derives tool usage statistics and developer actions from a project's change history (version control system). We also try to evaluate the maintainability changes through version control data; however, we investigate the general effect of version control operations regardless of tool usage.

Pratap et al. in their work [16] present a fuzzy logic approach for estimating the maintainability, while in this research we use a probabilistic quality model.

3 Methodology

In this section we summarize the kind of data we collected and how we elaborated on them to gain the results.

3.1 Version Control Operations

We take the version control operations as predictors of source code maintainability. In this work we investigated the number of the version control operation types: we only checked how many Adds, Updates and Deletes existed in the examined commit. These three numbers of every commit formed the predictor input of the analysis. E.g., if a certain commit contains 2 file additions, 5 file updates and 1 file deletion, then the input related to that commit would be (2, 5, 1). The fourth version control operation – Rename – was not considered, because there were hardly any commits containing this operation.

As the used quality model handles Java files only, we removed the non-Java source related statistics. E.g., if a commit contained 3 updates, 2 of Java files and one of an XML file, then we simply treated this as a commit of 2 updates.

3.2 The Applied Quality Model

The dependent variable of the research was the quality of the source code. This was estimated by the ColumbusQM probabilistic software quality model [2], which is based on the ISO/IEC 9126 standard [10].

The model calculates a composite measure of source code metrics like logical lines of code, complexity, number of coding rule violations etc. The calculation is based on expert weights and a statistical aggregation algorithm that uses a so-called benchmark as the basis of the qualification. The resulting maintainability value is expressed by a real number between 0.0 and 1.0. The higher number indicates better maintainability. See the work of Bakota et al. [2] for further details about the model.

3.3 Maintainability Change

The system's maintainability change can be calculated as the difference of the maintainability values of the current revision and the previous one. However, a simple subtraction is not sufficient for two reasons:

- The quality model provides the quality value based on a distribution function. The absolute difference between e.g. 0.58 and 0.54 is not the same as between 0.98 and 0.94. The latter difference is bigger as improving a software with already high quality is harder than improving a medium quality system.
- The same amount of maintainability change (e.g. committing 10 serious coding rule violations into the source code) has a much bigger effect on a small system than on a large one.

To overcome these shortcomings, we applied the following transformations:

- We used the quantile function of the standard normal distribution to calculate the original absolute value from the goodness value. This is feasible because the goodness value is derived from a probability function with normal distribution. We performed this transformation with the `qnorm()` R function [17]. The transformed values served as the basis of the subtractions.
- We multiplied the results of the subtractions (the maintainability value differences) by the current size of the system, more specifically, the current total logical lines of code (TLLOC, number of non-comment non-empty lines of code).

Figure 1 illustrates why the quantile conversion is necessary. The same difference on the y axis is not the same after quantile conversion (x axis), as expected.

We defined the quality change of the first commit to be 0.0.

3.4 Two-Sample Wilcoxon Rank Tests

For investigating **RQ1** and **RQ2**, two subsets of the commits were defined in several ways detailed below. The partitioning was performed based only on the version control operations in each case. After the partitioning we examined the maintainability changes of the commits belonging to these subsets. To check if the differences are significant or not, we used the two-sample Wilcoxon rank test (also known as Mann-Whitney U test) [9]. The Wilcoxon rank test is a so-called paired difference test, which checks if the population mean ranks differ in two data sets. Unlike mean this is not sensitive to the extreme values.

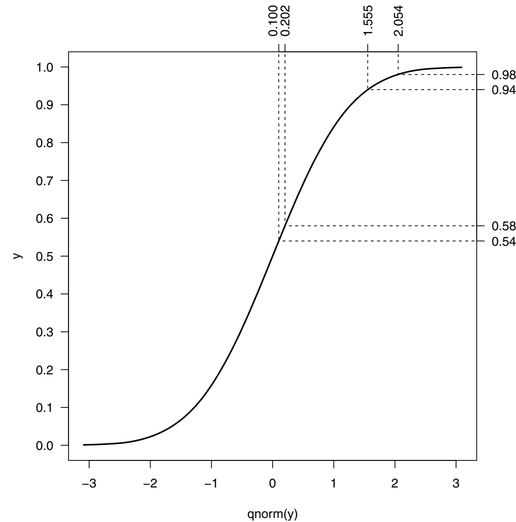

Fig. 1. Illustration why quantile conversion is necessary

The tests were performed by the `wilcox.test()` function in R [17]. The result of the test is practically the p-value, which tells us the probability of the result being at least as extreme as the actual one, provided that the null-hypothesis is true. In every case the null-hypothesis was that there is no difference between the distribution of the maintainability change values in the two commit sets. The alternative hypothesis was the following: the elements (maintainability differences) in one subset are less or greater than those in the other subsets.

Instead of executing a two direction Wilcoxon rank test (which would consider only the absolute magnitude of the difference, and not the direction – i.e. which one is greater), we executed the one direction test twice: first considering that the values in the first subset are less than those in the second, and in the second case we checked the opposite direction. We chose this approach as we needed the direction as well (we were not satisfied with the answer that the values are different in one subset compared the other, we also wanted to know which of them are less and which are greater).

As we performed the test twice each time, two p-values resulted. Let us denote them with p_1 (in case of values in the first set are less than those in the second one) and p_2 (the opposite direction). E.g., in case of a concrete division it turned out that the p-value is 0.0046 being numbers in one subset greater than those in the other, which also means that the p-value of having smaller values in the first set is 0.9954. Please note that the sum of these p-values are always 1.0, i.e. 100%. From the two p-values we consider the better one, noting the direction this result was executed with. Therefore the result is practically always exactly twice as good as it would be in case of a two direction test. E.g., in case of comparing two exactly same datasets, the resulting p-value is 0.5. This is considered when analyzing the results.

In order to be able to publish the results in a concise format, we introduced an approximate approach: we calculated the number of zeros between the decimal point and the first non-zero digit of the p-value. More formally, if the canonical form of the p-value is $(a \cdot 10^b)$, the transformed value is the absolute value of the exponent minus one (i.e. $|b| - 1$). E.g., if the p-value is 0.0046, then the canonical form is $4.6 \cdot 10^{-3}$, so the absolute value of the exponent is 3, minus 1 yields 2.

Please note that at least one of the two exponents is 0. Therefore for an even more compact interpretation, the non-null value is taken with appropriate sign (positive if the values in the second dataset are greater than in the first one, and negative in the opposite case), which can be calculated as the difference of the two p-values. Formally, this transformation was calculated by the following function:

$$f = \left\lfloor log\frac{1}{p_1} \right\rfloor - \left\lfloor log\frac{1}{p_2} \right\rfloor \tag{1}$$

3.5 Divisions

This section describes how the two subsets of the whole commit set were defined. All of the below mentioned partitions were performed for every version control operation type (Add, Update and Delete).

First we define the notion of *main dataset* which can be one of the following:

- The whole dataset, including all the revisions.
- The subset of the commits where the examined commit operation type occurs at least once.
- The subset of the commits where all the commit operations are of the same type.

We partitioned the main dataset into two parts (*first dataset* and *second dataset*) in the following ways:

- Divide the main dataset into two, based on the median of the absolute number of the examined operations. The greater values go into the first dataset, the second dataset is the complementary of the first one considering the main dataset.
- Divide the main dataset into two based on the median of the proportion of the examined operations, with similar division.
- Take the main dataset as the first dataset, and the second dataset as its complementary considering the whole dataset. This division can be defined only if the main dataset is not the whole dataset.

After eliminating those combinations which are not relevant, we ended up with seven combinations for dataset division per commit operation type. All of these are illustrated with the example of operation Add and the assumption that the presence of this operation has positive impact on the maintainability.

DIV1: *Take all commits, divide them into two based on the absolute median of the examined operation.* It checks if commits containing high number of operation Add have better effect on maintainability than those containing low number of operation Add.

DIV2: *Take all commits, divide them into two based on the relative median of the examined operation.* It checks if the commits in which the proportion of operation Add is high have better effect on maintainability compared to those where the proportion of operation Add is low. To illustrate the difference between DIV1 and DIV2 consider a commit containing 100 operations, 10 of them are Addition (the absolute number is high but the proportion is low) and a commit containing 3 operations, 2 of them are Additions (the absolute number is low, but the proportion is high).

DIV3: *The first subset consists of those commits which contain at least one of the examined operations, and the second one consists of the commits without the examined operation.* It checks if commits containing file addition have better effect on the maintainability than those containing no file additions at all.

DIV4: *Considering only those commits where at least one examined operation exists, divide them into two based on the absolute median of the examined operation.* This is similar to DIV1 with the exception that those commits which does not contain any Add operation are not considered. This kind of division is especially useful for operation Add, as this operation is relatively rare compared to file modification, therefore this provides a finer grained comparison.

DIV5: *Considering only those commits where at least one examined operation exists, divide them into two based on the relative median of the examined operation.* Similar to DIV2; see the previous explanation.

DIV6: *The first subset consists of those commits which contain the examined operation only, and the second one consists of the commits with at least one another type of operation.* This checks if commits containing file additions exclusively have better effect on the maintainability compared to those containing at least one non-addition operation. This division is also especially useful in case of file updates.

DIV7: *Considering only those commits where all the operations are of the examined type, divide them into two based on the absolute median of the examined operation.* This division is used to find out if it is true that commits which contain more file additions result better maintainability compared to those containing less number of additions. It is especially useful in case of file updates, as most of the commits contain exclusively that operation.

Please note that 2 of the theoretically possible 9 divisions were eliminated because they always yield trivial divisions (100% - 0%):

- All commits and its complementary. The complementary of all commits is always empty.

– Relative median division of commits containing the examined operation only. In these cases the proportion of the examined operation is always 100%, therefore one of the 2 datasets would be empty.

Table 1 illustrates these divisions.

Table 1. Divisions

	Complementary	**Absolute Median**	**Relative Median**
All Commits	-	DIV1	DIV2
Operation Exists	DIV3	DIV4	DIV5
Operation Exclusive	DIV6	DIV7	-

The tests were executed on all of these combinations during the experiment. In case of median divisions, if the median was ambiguous, both cases were tested (checking into which subset these elements should be added), and the better division (the more balanced division) is taken. In order to present the result in concise format the aforementioned exponent values were calculated for every possible combination and they were summed per system and operation. The mathematical background behind the addition is based on the exponents. If the p-values are independent, then the root probability is the product of the original probabilities, in which case the exponent of the resulting value would be approximately the sum of the exponents.

As a result we get a matrix with the version control operations in the rows and analyzed systems in the columns, and an integer value in each cell. We stress that this is only an approximation, fist of all, because the divisions are not independent. However, it is adequate for a quick overview, and for comparing the results of different systems. We also drill down in one case to illustrate how the calculated numbers were aggregated.

3.6 Random Checks

To validate the results, a random analysis was performed as well in the following way. We kept the original source control operations data and the values of the quality changes. But we permuted randomly the order of the revisions they were originally assigned to, just like a pack of cards (using the `sample()` R function). We performed randomization several times, permuting the already permuted series and executed the same analysis with the randomized data as with the original to assess the significance of our actual results.

The expected values of the exponents in random case can be derived from the diagram in Figure 2: 80% having 0, $9-9\%$ having -1 and 1, $0.9-0.9\%$ having -2 and 2, $0.09-0.09\%$ having -3 and 3, etc. With other words, the probability of the absolute value of the random exponents being at least 1 is 20%, 2 is 2%, 3 is 0.2%, etc.

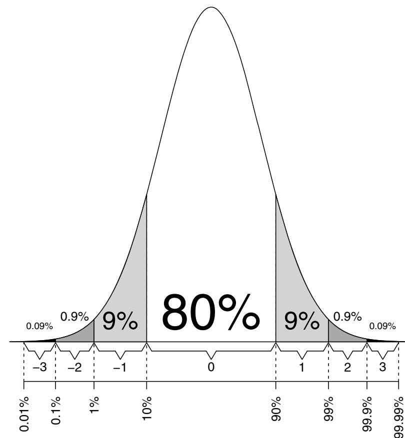

Fig. 2. Illustrating the calculated exponent values

As we have 3*7=21 tests per project all together, statistically 21*0.2=4.2 of them would be a non-null value, and 0.42 of them having an absolute value of at least 2. Therefore we set the acceptance criterion for the test that the absolute value of the exponents to be at most 2, which corresponds to the p-value 0.02. As we checked 4 projects (see below), statistically this means that 1 or 2 of the 4*21=84 cases would be false significant.

The expected absolute value in random case is about 1. Based on a check we found that the absolute value is at least 1 in about 66% of the cases, at least 2 in about 24%, at least 3 in about 7%, at least 4 in about 1.7%, at least 5 in about 0.35% of the cases, and so on. Based on this we accept the absolute values 4 and higher as significant.

4 Discussion

4.1 Examined Software Systems

The analysis was performed on the source code of 4 software systems. One of them was an industrial one, of which we had all the information from the very first commit. The others were open-source ones. Unfortunately, we did not find any open-source project of which we had all the commits from the beginning of the development in the same version control system. In order to gain as adequate results as possible, we considered only those projects for which we had at least 1,000 commits affecting at least one Java file. Furthermore, the too small code increase could also have significant bias, therefore we considered only those systems where the ratio of the maximal logical lines of code (typically the size of the system after the last available commit) and the minimal one (which was typically the size of the initial commit) was at least 3. We found 3 such systems which met these requirements.

Therefore, all together we performed the analysis on the following 4 systems:

- **Ant** – a command line tool for building Java applications[1]
- **Gremon** – a greenhouse work-flow monitoring system.[2] It was developed by a local company between June 2011 and March 2012.
- **Struts 2** – a framework for creating enterprise-ready java web applications.[3]
- **Tomcat** – an implementation of the Java Servlet and Java Server Pages technologies.[4]

Table 2 shows the basic properties of them.

Table 2. Analyzed systems

Name	Min.	Max.	Total	Java	Total number of			Rev. with 1+			Rev. with only		
	TLLOC[5]		Commits		A	U	D	A	U	D	A	U	D
Ant	2,887	106,413	6,118	6,102	1,062	20,000	204	488	5,878	55	196	5,585	19
Gremon	23	55,282	1,653	1,158	1,071	4,034	230	304	1,101	89	42	829	8
Struts 2	39,871	152,081	2,132	1,452	1,273	4,734	308	219	1,386	94	41	1,201	12
Tomcat	13,387	46,606	1,330	1,292	797	3,807	485	104	1,236	77	32	1,141	23

4.2 Summarized Results of the Wilcoxon Tests

The results of the methodology introduced in Section 3.4 are shown in Table 3. The absolute number reflects the magnitude of the impact, while the sign gives the direction (maintainability increase or decrease). Figure 3 illustrates the same results as follows: the upper light gray bars represent the file additions, the lower darker gray bars the file updates, and the black vertical lines the file deletions. The file additions are all located on the positive part, the file updates on the negative, and deletions are hectic, with lower absolute values.

Table 3. Sum of the exponents

	Gremon	Ant	Struts 2	Tomcat
Add	5	62	20	14
Update	-11	-29	-11	-3
Delete	4	-12	-6	1

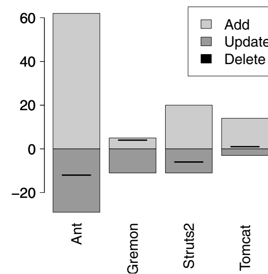

Fig. 3. Exponents with bars

These results cannot be interpreted on their own, they only provide a rough overview. The divisions are not independent; furthermore, in some cases the exactly same divisions are checked several times.

[1] `http://ant.apache.org`

[2] `http://www.gremonsystems.com`

[3] `http://struts.apache.org/2.x`

[4] `http://tomcat.apache.org`

[5] Total Logical Lines Of Code – Number of non-comment and non-empty lines of code

4.3 Wilcoxon Tests Details

For details on the above numbers consider Table 4. The sum of the rows are also shown, which helps us drawing the attention on the most promising results. We recall the probabilities in random case (see Subsection 3.6), to illustrate the magnitude of the numbers in the first 3 columns.

Table 4. Exponent details

Operation	Division	Gremon	Ant	Struts 2	Tomcat	$\sum$
Add	DIV1	1	15	6	4	26
	DIV2	1	15	6	4	26
	DIV3	1	15	6	4	26
	DIV4	0	7	1	1	9
	DIV5	1	1	0	0	2
	DIV6	1	6	1	1	9
	DIV7	0	3	0	0	3
Update	DIV1	-2	2	0	0	0
	DIV2	-2	-11	-3	-1	-17
	DIV3	-2	-4	0	-1	-7
	DIV4	-1	2	0	1	2
	DIV5	-1	-8	-4	-1	-14
	DIV6	-2	-11	-3	-1	-17
	DIV7	-1	0	-1	0	-2
Delete	DIV1	0	-1	0	0	-1
	DIV2	0	-1	0	0	-1
	DIV3	0	-1	0	0	-1
	DIV4	0	0	-1	0	-1
	DIV5	1	-2	-3	0	-4
	DIV6	3	-5	-2	0	-4
	DIV7	NA	-2	0	1	-1

The diagrams in Figure 4 illustrate the results of the Wilcoxon rank tests visually, where the values found in the summary column of Table 4 are illustrated. High absolute length of a bar means high significance within the project. Comparison is also interesting between the projects: high absolute lengths on the same place are considered as a strong result.

In case of operation Add (left bars, light gray) all of the bars are non-negative for every system. The bars related to DIV1, DIV2 and DIV3 are the tallest, and in 3 out of the 4 cases the bars for DIV4, DIV5, DIV6 and DIV7 are similar.

The bars for operation Update (middle bars, dark gray) are a bit more hectic; in general we can say that the height of most of the bars are negative. Furthermore, in case of DIV2, DIV5 and DIV6 we have long negative bars.

The hectic results of operation Delete (right bars, black) are also illustrated.

Now let us check the results in the tables.

Addition. The results in the first 3 divisions (DIV1, DIV2, and DIV3) are in all cases the same, because addition exists in less than half of the commits (see the definitions in Section 3). The overall result of the Wilcoxon test (26) is very high for these divisions, the highest absolute value in the table. On 3 out of the 4 projects the test yielded significant result (exponent ≥ 2). This definitely means that *commits containing additions have better effect on the maintainability compared to those containing no file additions.*

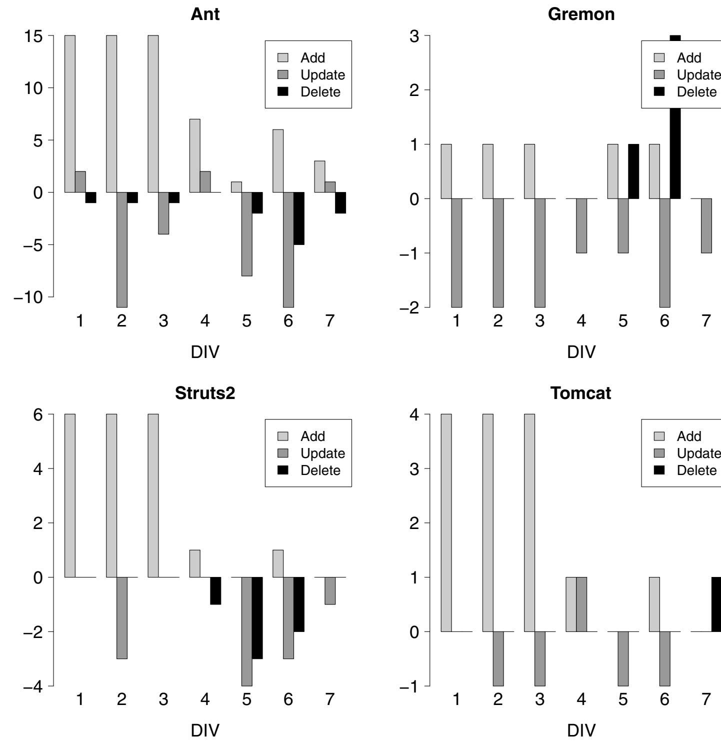

Fig. 4. Wilcoxon test result bars

The result of DIV4 is also relatively high (9), however, this is caused by a high value for one project, with less support of the others. This means *that for commits containing an addition, in some cases the higher absolute number of addition results better maintainability, comparing with the commits of lower number of additions*. It is interesting that this is not the case if the proportion of additions in taken (DIV5 with result of 2): *higher proportion of file addition does not result in significantly better maintainability.*

DIV6 checks if commits containing exclusively file additions have significantly different effect on maintainability compared to those containing other operations as well. The overall result (9) is remarkably high; however, the result is modulated by the fact that in case of 3 projects the connection is weak. The reason could be the low number of commits containing file additions only (the p-value is affected also by the number of elements: if the same result is supported by a higher number of elements, the p-value is lower). The result of DIV7 tells that if all file operations are file additions, then the connection between the number of operations and the maintainability is weak.

Update. Unlike in case of file addition, file update results of the first 3 divisions are quite different. Based on DIV1 (final result is 0) there is no difference between the effects of the commits with low and high absolute number of Update

operations on maintainability. However, it contains a contradiction: besides having 2 zeroes, the result contains a +2 (meaning that the high number of Updates significantly improves the maintainability) and a -2 (meaning it significantly decreases). We have not found the reason of this contradiction, possibly further investigations of other data is necessary.

On the other hand, DIV2 provides a very significant result (-17), and this is supported by every analyzed system with varying degree. This suggests that the proportion of operation Update really matters from maintainability point of view. In general, the higher the proportion of the operation Update within a commit, the worse its effect on the maintainability. DIV7 resulted in exactly the same values, because the divisions were the same: the commit either contains exclusively update or contains other operations as well.

Based on DIV3 we can say that the mere existence of operation Update has a negative effect on maintainability (comparing with those not containing any Update). This was significant in 3 out of the 4 systems, with no contradiction on the 4th. DIV4 is similar to DIV1 (absolute median division of those commits which contain at least one Update) with similar low significance and small contradiction. DIV5 is similar to DIV2 (relative median division of those commits which contain at least one Update) with similar but lower exponents.

The result of DIV6 (-17) is significant, which is supported by most of the checked systems. This means that in general, the presence of operation Update has a negative effect on the maintainability. DIV7 is similar to DIV1 or DIV4 (absolute median division of commits containing exclusively Update), and the result is not significant at all.

The Update operation has negative effect on maintainability, but the way how it appears (alone or together with other operation) really matters. It seems that the presence of other operations suppresses the effect of the update.

Delete. The effect of the operation Delete seems a bit contradictory. In case of Ant and Struts 2 we have non-positive results only. In case of Gremon and Tomcat the values are non-negative but with lower absolute values than the others. The NA means not available; in case of Gremon there were not enough commits which contained exclusively Delete operations and we could not perform a division based on the number of operations so that both sets would contain enough number of elements to be able to compare. There were 8 such commits, and we executed the test only if at least 5 elements in both subset existed.

The highest absolute values can be found in case of DIV6, meaning that there is a significant difference in the effect on the maintainability between commits containing exclusively Delete operations compared to those containing other operations as well, but the values are very contradictory. In 2 out of the 4 cases the results suggest that deletion significantly decreases the maintainability. This is a bit strange because it suggests that it is more likely that the more maintainable code is removed than those harder to maintain. Deletion could typically occur in case of refactoring, and we would expect that the hard-to-maintain code is removed and better-to-maintain code appears instead, but it seems that this is not the case.

On the other hand, in case of Gremon just the opposite is true with relatively high confidence. We have not found any explanation to this contradiction, and we cannot be certain that the reason is the fact that Gremon is an industrial software, implemented by paid programmers, while the others are not, implemented by volunteers, or something entirely different.

4.4 Answers to the Research Questions

RQ1: *Does the amount of file additions, updates and deletions within a commit impact the maintainability of the source code?*

Consider Table 3. For interpretation of the magnitude of these values please consider the random probabilities in Section 3.6. In case of operation Add all the values are positive, and all of them can be considered to be significant. Therefore, we can state that operation Add has positive impact on the maintainability. In case of operation Update all the values are negative. The absolute value of one of them (-3 in case of Tomcat) is relatively low which would not be convincing in itself. However, along with the others we can state that operation Update has negative impact on the maintainability. In case of operation Delete we have 2 positive and 2 negative results, containing low and high absolute values as well. Considering these data only we cannot formulate a valid statement for this operation.

RQ2: *Are there any differences between checks considering the absolute number of operations (Add, Update, Delete) and checks investigating the relative proportion of the same operation within commits?*

Consider Table 4. The values found in DIV1 (absolute median) should be compared with DIV2 (relative median) and those in DIV4 with DIV5. In case of operation Add there is no difference between DIV1 and DIV2. In case of comparing DIV4 with DIV5 we find that the values in the sum column are 9 and 2, respectively. This seems to be a good result at first glance; however, this is caused by only one value, and all the other 7 values are not significant. Therefore, based on these values only we cannot formulate anything for operation Add. In case of operation Update the relative median (DIV2 and DIV5) results in significantly lower values than those of absolute median (DIV1 and DIV4). Therefore, we can state that in case of Update the high proportion of the operation causes the maintainability decrease, rather than the absolute number of it. In case of operation Delete we again cannot formulate any statement.

5 Threats to Validity

The facts which might potentially threaten some of the validity of the results are the following.

The data are not fully complete, which is true for many of the researches of course; ours is not an exception either. The commit data for the Gremon project is complete: all the commits are available from the very beginning. On the other hand, large amount of data in case of open-source projects are missing. E.g., in

case of every project the initial commit contained a large amount of development which came from another version control system. To alleviate this problem we chose projects having plenty enough code enhancements during the development. In some cases a lot of development was done in another branch, and this appears as a huge merge in the examined branch. This could also have significant bias.

The results of the different systems show similar tendency in most of the cases; however, in some cases the results are diverging. This is especially true for operation Delete. We have not found the reason of these divergences.

There is no definition to the quality of the source code expressed by a number. The ColumbusQM tool used by us is one of several approaches, with its own advantages and drawbacks. We find this model as a good, well founded one, but we are aware that it is not perfect: it is being improved continuously. We treat this as an external threat and hope that more and more precise information about the software quality will support the results with higher confidence instead of threatening it.

6 Conclusions and Future Work

This research is part of a longer term study, aimed to identify the patterns of the developers' behavior which causes significant impact on the source code quality.

We studied the impact of version control commit operations on the maintainability change. Only the bare number of operations was considered, nothing else. We received some interesting answers, which are the following.

We found that file additions have positive, or at least better impact on maintainability, compared to the effect of those commits containing no or small number of additions.

On the other hand, the research showed that file updates have significant negative effect on the maintainability.

We found no clear connection between file deletions and maintainability. Based on this research the net effect of this operation is rather negative than positive, which contradicts with our initial assumption.

We identified the similarities and the differences between the high number and the high proportion of existence of a version control operation within commits. Later on this fact might also be an important part of the formula which will hopefully explain the influence of the developer's interactions on the source code quality.

Answers to these any maybe many other questions might help improving the knowledge where exactly the software erosion decreases.

Acknowledgments. This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0006, and the European Union and the State of Hungary, co-financed by the European Social Fund in the framework of TÁMOP 4.2.4. A/2-11-1-2012-0001 „National Excellence Program".

References

1. Atkins, D.L., Ball, T., Graves, T.L., Mockus, A.: Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. IEEE Transactions on Software Engineering 28(7), 625–637 (2002)
2. Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., Gyimóthy, T.: A Probabilistic Software Quality Model. In: Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011), pp. 368–377. IEEE Computer Society, Williamsburg (2011)
3. Bakota, T., Hegedus, P., Ladányi, G., Körtvélyesi, P., Ferenc, R., Gyimóthy, T.: A Cost Model Based on Software Maintainability. In: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012), pp. 316–325. IEEE Computer Society, Riva del Garda (2012)
4. Bernstein, A., Bachmann, A.: When Process Data Quality Affects the Number of Bugs: Correlations in Software Engineering Datasets. In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories, MSR 2010, pp. 62–71 (2010)
5. Fluri, B., Wursch, M., Gall, H.C.: Do Code and Comments Co-evolve? On the Relation between Source Code and Comment Changes. In: 14th Working Conference on Reverse Engineering, WCRE 2007, pp. 70–79. IEEE (2007)
6. Gall, H., Jazayeri, M., Krajewski, J.: CVS Release History Data for Detecting Logical Couplings. In: Proceedings of the Sixth International Workshop on Principles of Software Evolution, pp. 13–23. IEEE (2003)
7. Hayes, J.H., Patel, S.C., Zhao, L.: A Metrics-Based Software Maintenance Effort Model. In: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR 2004), pp. 254–260. IEEE Computer Society, Washington, DC (2004)
8. Hindle, A., German, D.M., Holt, R.: What Do Large Commits Tell Us?: a Taxonomical Study of Large Commits. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008, pp. 99–108. ACM, New York (2008)
9. Hollander, M., Wolfe, D.A.: Nonparametric Statistical Methods, 2nd edn. Wiley-Interscience (January 1999)
10. ISO/IEC: ISO/IEC 9126. Software Engineering – Product quality 6.5. ISO/IEC (2001)
11. Koch, S., Neumann, C.: Exploring the Effects of Process Characteristics on Product Quality in Open Source Software Development. Journal of Database Management 19(2), 31 (2008)
12. Mockus, A., Weiss, D.M., Zhang, P.: Understanding and Predicting Effort in Software Projects. In: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), pp. 274–284. IEEE Computer Society, Washington, DC (2003)
13. Moser, R., Pedrycz, W., Sillitti, A., Succi, G.: A Model to Identify Refactoring Effort during Maintenance by Mining Source Code Repositories. In: Jedlitschka, A., Salo, O. (eds.) PROFES 2008. LNCS, vol. 5089, pp. 360–370. Springer, Heidelberg (2008)
14. Parnas, D.L.: Software Aging. In: Proceedings of the 16th International Conference on Software Engineering, ICSE 1994, pp. 279–287. IEEE Computer Society Press, Los Alamitos (1994)

15. Peters, R., Zaidman, A.: Evaluating the Lifespan of Code Smells using Software Repository Mining. In: Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, CSMR 2012, pp. 411–416. IEEE Computer Society, Washington, DC (2012)
16. Pratap, A., Chaudhary, R., Yadav, K.: Estimation of software maintainability using fuzzy logic technique. In: 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), pp. 486–492 (February 2014)
17. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2013), `http://www.R-project.org/`
18. Ratzinger, J., Sigmund, T., Vorburger, P., Gall, H.: Mining Software Evolution to Predict Refactoring. In: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, ESEM 2007, pp. 354–363. IEEE Computer Society, Washington, DC (2007)
19. Schofield, C., Tansey, B., Xing, Z., Stroulia, E.: Digging the Development Dust for Refactorings. In: Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC 2006, pp. 23–34. IEEE Computer Society, Washington, DC (2006)
20. Stroggylos, K., Spinellis, D.: Refactoring–Does It Improve Software Quality? In: Fifth International Workshop on Software Quality, WoSQ 2007: ICSE Workshops 2007, p. 10. IEEE (2007)
21. Tóth, G., Végh, Á.Z., Beszédes, Á., Schrettner, L., Gergely, T., Gyimóthy, T.: Adjusting Effort Estimation Using Micro-Productivity Profiles. In: Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST 2011), pp. 207–218 (October 2011)