Cumulative Code Churn: Impact on Maintainability

Csaba Faragó, Péter Hegedűs, and Rudolf Ferenc
Department of Software Engineering,
University of Szeged, Hungary
Email: {farago,hpeter,ferenc}@inf.u-szeged.hu

***Abstract*—It is a well-known phenomena that the source code of software systems erodes during development, which results in higher maintenance costs in the long term. But can we somehow narrow down where exactly this erosion happens? Is it possible to infer the future erosion based on past code changes? Do modifications performed on frequently changing code have worse effect on software maintainability than those affecting less frequently modified code?**

In this study we investigated these questions and the results indicate that code churn indeed increases the pace of code erosion. We calculated cumulative code churn values and maintainability changes for every version control commit operation of three open-source and one proprietary software system. With the help of Wilcoxon rank test we compared the cumulative code churn values of the files in commits resulting maintainability increase with those of decreasing the maintainability. In the case of three systems the test showed very strong significance and in one case it resulted in strong significance (p-values 0.00235, 0.00436, 0.00018 and 0.03616). These results support our preliminary assumption that modifying high-churn code is more likely to decrease the overall maintainability of a software system, which can be thought of as the generalization of the already known phenomena that code churn results in higher number of defects.

***Index Terms*—code churn, ISO/IEC 25010, source code maintainability, Wilcoxon test**

I. Introduction

Software maintainability plays a crucial role in modern development projects. It is one of the six sub-characteristics of software quality, as defined originally in the ISO/IEC 9126 standard [1]. Software maintenance consumes huge efforts: based on the experiences, high proportion of the total amount of software development costs are spent on this activity. As maintainability is in direct connection with maintenance costs [2], our motivation is to investigate the effect of the development process on the maintainability of the code. Our goal is to explore typical patterns causing similar changes in software maintainability, which could either help to avoid software erosion, or provide information about how to better allocate efforts spent on improving software maintainability.

In previous works we already tackled this area of research. In a recent paper [3] we presented that there is a strong connection between the version control operations and the maintainability of the source code. We also performed a study [4] that revealed the connection of the version control operations and maintainability. It turned out that file additions have rather positive, file updates have rather negative effect on maintainability, while a clear effect of file deletions was not identified. Furthermore, in a more recent work [5] we presented the results of a variance analysis. File additions and file deletions increase the variance of the maintainability, and operation Update decreases it.

Up to now we considered the number of version control operations only. In this study we dig a bit deeper: we consider also on which file the change was performed on. Furthermore, we "look into" the source files, and try to assess the extent of the file changes. We analyze if historical changes have any long term effect on files in terms of maintainability changes caused by current modifications.

We introduce the concept of *cumulative code churn* and use its value to express the gross amount of past code changes. Khoshgoftaar et al. [6] defined code churn as *"the number of lines added to, deleted from, or modified in a source module"*. In this study we consider the total gross amount of changes from the very beginning of the development of a software system. The term cumulative code churn expresses the fact that the code churn values are accumulated (summed).

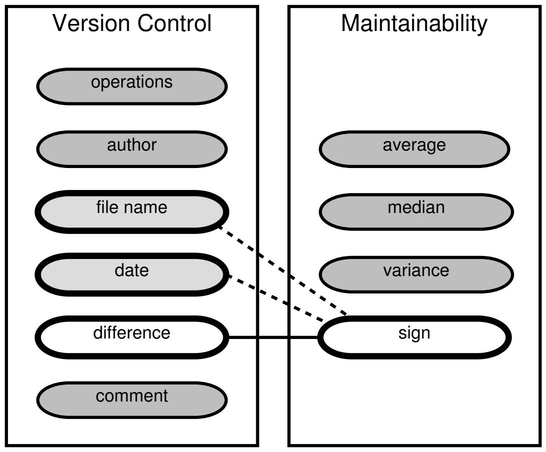

Figure 1. Overview of the research

Figure 1 demonstrates how this study fits into our long term research of revealing the connections between patterns in development activities and maintainability.

The left hand side rectangle indicates the information which can be gained directly from a version control system. These are the following, per commit: operations performed on files (e.g. file addition, modification, deletion or rename), along with the file names, the author, the date and the comment, and if we dig a bit deeper, then the difference between the previous and the current revision can also be extracted. Ellipses with dark background and narrow border line indicate information not used in this study (these are either analyzed in other works, or not yet analyzed). The white one (difference) is the most important part of this study. The ellipses with light gray color and wide border line indicate source control information which are implicitly used: the file name (as we calculate the historical code churn for each file) and the date (as the order of the commits matter).

The right hand side rectangle represents maintainability, which is actually the absolute change of the maintainability of the source code between the previous and current revision. Other studies dealt with the average, the mean and the variance of subsets of maintainability changes; in this study we considered the sign, i.e. the direction of the maintainability change, indicating if maintainability increases or decreases.

We performed our analysis on commit basis. We collected the historical data from the SVN source code version control system and estimated the maintainability with the help of the ColumbusQM probabilistic software quality model [7].

A practical use of the results could be the following. It would be useful to show information in the integrated development environments about cumulative code churn values (e.g. as defined in this study, or other, similar, available and easy to compute ones). This could warn the developers for the possible risk of maintainability decrease if the file is to be modified. Source code quality actions – for example, mandatory code review with stricter rules than usual – could be applied.

The added value of this approach to simply calculating maintainability with the ColumbusQM model is that it is light-weight. While ColumbusQM needs the source code, a complete benchmark and is very computation intensive, our approach requires VCS access only.

Formally, we investigated the following research question in this paper:

Research Question: *Do commits that involve files which were previously intensively modified have a different impact on the maintainability of the source code, compared to those commits affecting less intensively modified files?*

Null hypothesis: *The amount of the past modifications does not have any influence on the maintainability changes caused by future commits.*

Assumed alternative hypothesis: *Modifying files which have been modified intensively in the past is more likely to result in further maintainability decrease than modifying files that have been less intensively modified earlier.*

The null hypothesis basically claims that the maintainability changes can take arbitrary values: positive, negative and even zero, and the probability of being any value is independent from the past modification. Technically speaking, the assumed alternative hypothesis is not the perfect negation of the null-hypothesis, but it is stronger. We constructed the statistical tests not just to decide whether we can reject the null-hypothesis, but also to find evidences that support the assumed alternative hypothesis in such case.

The research question can be rephrased informally as follows: *Does high cumulative code churn have a bad effect on maintainability?* In our interpretation, the high value of cumulative code churn for a file expresses the fact that it has been modified heavily during its lifetime. We investigated this question by studying three open-source and an industrial system. According to the results presented in this paper, we could reject the null hypothesis, as the performed Wilcoxon rank test showed significant results in all cases. This supports the common intuition that modifying high-churn code is more likely to cause software degradation.

The remaining of the paper is organized as follows. Section II provides a brief overview of works that are related to this research. In Section III we present the methodology of how we collected the data, what kinds of tests we performed and how we illustrated the results. In Section IV we describe the results of the statistical tests. In Section V we list the possible threats to the validity of the results, while Section VI concludes the paper.

II. Related Work

Analyzing the effect of code churn on source code, especially for defect prediction, is an intensively investigated research area. We, on the other hand, studied the impact of cumulative code churn on future maintainability changes. Our research may be thought of as the generalization of the already published results, as we think the number of defects is certainly an aspect of maintainability, but not the only one.

Khoshgoftaar et al. [6] present a gross change prediction improvement using neural networks. Their measure of quality is the gross change of source code from the beginning of the testing phase to the end of maintenance phase. They executed their model on 8 software systems and concluded that their approach with neural networks resulted in a much improved quality prediction.

In another study, Khoshgoftaar et al. [8] assessed the reliability of telecommunication software systems. They considered a software module as fault prone if it exceeded a threshold of debug code churn. They defined code churn as the number of lines added or changed due to bug fixes. We considered the number of lines deleted as well.

In their article Munson et al. [9] presented how they calculated the code churn values, and proved that this was a proper fault surrogate. They synthesized the measurements, and defined code churn (as a new measure) by comparison of the complexity of sequential builds. They analyzed 19 builds of a large embedded system, with about 300 thousand lines of code, consisting of more than 3700 modules, written in C.

Ohlson et al. [10] analyzed the same phenomena as we did, the code erosion; they used the phrase "code decay." They refer to code churn as the number of defect fix reports for a component. The analysis was based on 8 releases of a legacy software with 130 components. They were able to identify the most fault-prone components with the help of code churn and other metrics. We executed our tests on source code basis, and not on component basis.

Hall et al. [11] presented their concept of code delta and code churn compared to a baseline with the help of a real, industrial software system. Our baseline was the first commit of the software, which was really the very beginning in one case, and in the other 3 cases the first commit was a migration from another source control system.

Eick et al. [12] investigated a huge project (containing about 100 million lines) written in C++ to find evidence for code decay. They found statistical evidence of this phenomena: the number of files touched per change increased; parallel to this, the modularity has been declined by changes touching multiple modules; furthermore they dealt with fault rates and effort prediction as well. On the other hand, they could not find evidence if the code decay can be fatal, i.e. not possible to change further. In our study we also analyzed how historical changes affect the maintainability, and we performed one step further: considering the code decay as an evidence, tried to identify why, when and where it occurs.

Nagappan et al. [13] presented a study about a defect prediction model, validated on the source code of Windows Server 2003. They defined 8 relative code churn measures, e.g. churned lines of code per total lines of code. They showed that these measures correlate with defect density. They also concluded that relative code churn measures are good and absolute code churn measures are poor defect density predictors. They found that these relative code churn measures are good predictors of system defect density, and they can be efficiently used to distinguish between fault-prone and non fault-prone binaries.

The same authors present also another approach of post-release defect prediction [14], considering software dependencies and code churn. They found that this combination is a good predictor of faults. They used a very big data set, but they validated their concept on a single project. We, on the other hand, targeted projects from different domains (although smaller ones) to lower the chance of casual results.

Ajila et al. [15] performed a research on a long term software life cycle (considering a six years period). They analyzed the effect of code delta, code churn and rate of change on software evolution. They found no relationship between the size of the code added and the number of designers required to develop and test it. In the current research we targeted the available commits only, and did not consider information other than the source code. However, considering other software development interactions, like the low level IDE interactions or the information available in issue tracking systems are in our long term plan.

In their study Shin et al. [16] describe the result of testing if complexity, code churn and developer activity metrics (28 all together) obtained from source code and development history are proper indicators of the location of software vulnerabilities. They validated their approach on the source code of Mozilla Firefox and Red Hat Enterprise Linux kernel.

Giger et al. [17] showed that code churn defined simply by number of lines modified is not so good error predictor as fine-grained source code changes defined by them. They tested their concept on the source code of Eclipse.

Maintainability is also a very intensively investigated research area, and a complete overview of this topic exceeds the limits of this article. Therefore we just touch the surface of the most recent results.

Fry et al. [18] present their results on the comparison of the maintainability of human written and generated patches. They found that human written patches are slightly more maintainable than machine generated ones; however, they proposed a system which augment the machine generated patches with human readable documentation, and it changed the original tendency.

Yamashuta et al. discuss [19] how code smell interactions affect maintainability. They also provide evidences that code smells found in coupled artifacts have traceable effects on maintainability.

In their study [20], Hanenberg et al. present an experiment investigating if static type systems improve maintainability compared to dynamic type systems. They found that static type systems are beneficial in understanding source code and fixing type errors, but not in fixing semantic errors.

III. Methodology

This section describes how we performed statistical tests to answer our research question. Section III-A provides a high level overview about the methodology. Section III-B describes how we cleaned the input data, and why it was necessary. In Section III-C we describe how we calculated the maintainability. Then, in Section III-D we present how we determined the cumulative code churn values, while in Section III-E we demonstrate that the calculated maintainability and cumulative code churn values are independent. Section III-F describes the statistical tests we used for comparisons. In Section III-G we deal with the possible alternatives and describe why we selected the chosen approach. Finally, in Section III-H we present a method for validation cross-check.

A. Overview

We analyzed the connection between cumulative code churn and maintainability. In order to do this we needed the following information: cumulative code churn and maintainability both expressed as numeric values. Neither of them is trivial to obtain, there are no exact definitions on how to compute them. Sections III-C and III-D describe our approach of calculating these values.

We chose to work on a per commit basis as the maintainability values used as input data in this study were available for the revisions of the subject systems. This is a significant constraint with inevitable loss of precision, but we think this is the best strategy for using system level maintainability values. We are convinced that without this trade-off the results would be even more significant.

Technically, we were looking for a cumulative code churn and a maintainability value for all revisions of the analyzed software systems. The cumulative code churn values were calculated based on the change history from SVN. Regarding the maintainability of systems it was sufficient to have the information whether it increased, decreased or did not considerably change as a result of the commit operation in question.

As a result, we got a maintainability change sign – cumulative code churn value pair for each revision. These numbers are independent from each other extracted from different data sources. We performed a statistical test on these input data sets as described below.

B. Preliminary Steps

Before the actual computation we did some data cleaning. The analyzed software systems were all written in Java. The quality model we used (see below) considers Java source files only. Therefore we first removed the data related to the non-Java source files (e.g. xml files) from the input. After this step, some of the commits became empty. So we also removed the data about commits containing no Java source files at all. As a result, we worked on an input commit set containing Java source files, and each analyzed revision contained at least one committed Java file.

C. Calculation of the Maintainability Change

We calculated the maintainability value of every revision with the help of the ColumbusQM probabilistic software quality model [7]. This model is based on the fact that the increase of software metrics (e.g. object-oriented metrics defined by Chidamber and Kemerer [21]) decreases the maintainability. Gyimóthy et al. [22] empirically validated that the increase of some of these metrics increase the probability of faults.

The model itself considers the following metrics: logical lines of code, the number of ancestors, the maximum nesting level, the coupling between object classes, clone coverage, number of parameters, McCabe's cyclomatic complexity, number of incoming invocations, number of outgoing invocations, and number of coding rule violations. These metrics are compared with those of other systems in a benchmark, and then the results of the comparisons are aggregated using a probabilistic statistical algorithm utilizing also weights provided by experts.

The model was empirically validated, resulting that there is a correlation between the calculated maintainability values and the real development costs [2].

As a result, we get a sign (positive, zero, or negative) for each commit.

D. Cumulative Code Churn Calculation

This section describes how we calculated the cumulative code churn values for the revisions. First, we show how we defined the cumulative code churn of a file, then we define the churn value of a commit.

According to the literature [6], code churn is defined as follows: lines added, modified or deleted in a file from one version to another. We use a historical approach to extend this notion from the very beginning of the available revision history.

We initialized the cumulative code churn value for every file to zero. At each commit we performed the following on every file. We executed the SVN diff tool for the actual and the previous version of the file. Besides the change itself, it contains information where and how the changes occurred and how many lines were affected. The lines added are indicated with a plus ($+$) sign, and the removed ones are with minus ($-$) sign. Updates within lines are considered as a line removed and a line added.[1]

In the current work we considered the cardinality of line changes (both line additions and line deletions). These values are summed from the very beginning of the available version control history; this value forms the cumulative code churn of a file. As a result, we obtain how many lines have been added to the source code plus how many lines were removed in the history for every file in each commit.

As we already pointed out, the maintainability data was available commit-wise, so it was necessary to define the cumulative code churn value for a commit itself. A commit related to the revision in question may contain any number of files (to be more precise in our special case: it contains at least one Java source file). We somehow need to define the cumulative churn value of the commit itself.

First of all, during calculation we consider the value *before* the actual commit, i.e. not considering the current modifications. This means that we tried to find evidence on the effect of the actual commit without checking anything (except the affected files) of that commit.

Second, it was necessary to somehow find the common root of the calculated values, which should be a kind of an average of them. That was the proper choice (instead of considering for example just the maximum) because of the nature of the already available data, i.e. the maintainability. The sign of the maintainability change caused by a certain commit is the common impact of all the modifications of all the affected files of that commit, i.e. the final change is a kind of an average of the individual changes.

Therefore we chose the most straightforward approach and calculated the averages of the above churn values of the affected files.

We illustrate the cumulative code churn calculation on an artificial example. The example project contains 3 sources and 5 revisions. Let Table I contain the number of file modifications: lines removed and lines added to the files in the different revisions. For example, `Game.java` has been added at the third revision with 25 lines, and it was modified at fourth revision as follows: 3 lines has been removed and 7 added.

Table I
EXAMPLE FILE MODIFICATIONS

		Revision				
ID	**File name**	**1**	**2**	**3**	**4**	**5**
1	`Main.java`	0, 25	2, 3		10, 0	10, 15
2	`Data.java`	0, 30		0, 5	7, 23	15, 0
3	`Game.java`			0, 25	3, 7	

[1]More details about the unified diff format can be found on the pages http://en.wikipedia.org/wiki/Diff_utility#Unified_format and https://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html

For this case, Table II illustrates the calculated churn values for every file and every revision. They are initialized to 0 (representing the "0th" revision). Continuing the previous example, for `Game.java` this value remains 0 until it is added with 25 lines, then at the next update 3+7=10 lines have been modified, therefore the result in revision 4 will be 35.

Table II
EXAMPLE FILE CHURN VALUES

		Revision					
ID	**File name**	**0**	**1**	**2**	**3**	**4**	**5**
1	`Main.java`	0	25	30		40	65
2	`Data.java`	0	30		35	65	80
3	`Game.java`	0			25	35	

The commit related churn values are calculated based on the file related churn values. These values are listed in Table III. It contains which files were affected in each revision (Changed source ID, e.g. 3 for `Game.java`), the previous churn values and the calculated average. The average values calculated this way form the input of the statistical tests we performed.

Table III
EXAMPLE COMMIT CHURN VALUES

Revision	**1**	**2**	**3**	**4**	**5**
Changed source IDs	1, 2	1	2, 3	1, 2, 3	1, 2
Prev. churns	0, 0	25	30, 0	30, 35, 25	40, 65
Average	0.0	25.0	15.0	30.0	52.5

As result, we get a non-negative number representing the magnitude of the cumulative code churn of each commit.

E. Independence of the Values

The maintainability and the cumulative code churn values are entirely independent from each other. On one hand, the maintainability change for the nth revision is calculated as the sign of difference of maintainability values of the nth and the (n-1)th revisions (which is measured by utilizing source code metrics). Therefore its value is solely affected by the code change between the previous and the current revision.

On the other hand, the cumulative code churn value of the actual commit is affected by the modifications of the 1st, 2nd, 3rd, ..., (n-1)th revisions. The last considered piece of information in the calculation is (potentially) part of the code delta between the n-2th and n-1th revision (only if the same source files are affected by both commits).

Therefore the explanatory and response variables of the performed statistical tests are totally independent.

F. Comparison Tests

At this point we have 2 pieces of information for every commit:

- an indicator if the maintainability has been increased, did not change, or decreased; and
- a number illustrating the cumulative churn sizes of the source files in that commit.

We want to tell something about their connection. For that, we divide the commits into two subsets: commits with positive maintainability change and those of negative maintainability change. From now on we do not consider commits with zero maintainability change anymore. The zero maintainability changes are typically caused by small, one line modifications.

We take the churn numbers of each subset. Therefore, at this point, we have 2 sets of churn values: one for positive, and one for negative maintainability change commits. Our null hypothesis is that there is no difference in the magnitudes of these numbers. The alternative hypothesis is that there are significant differences in the churn values. Our assumption is that the churn values related to positive maintainability changes are significantly lower than those related to negative maintainability changes.

In order to verify this, we selected the Wilcoxon rank test (also known as Mann-Whitney U test). This is a nonparametric statistical test of the null hypothesis that two populations are the same, with the alternative hypothesis that one of the populations has significantly higher values than the other. In this case the alternative hypothesis is that commit churn values related to maintainability decrease are higher than the churn values related to maintainability increase.

The two major advantages of this test are the following: it does not require any specific distribution, and it is not sensitive to the outliers. Both constraints would be problematic in our case. If a statistical test requires a special distribution, that is normal distribution in most of the cases. But the defined churn values are not of normal distribution, but rather similar to an exponential distribution. The other problem is the presence of outliers. The uncommon commits (like merging huge amount of code from another branch, or renaming hundreds of files by removing and adding them) would cause significant bias in case of statistical tests sensitive to outliers.

There are two-tailed and one-tailed versions of this test. The two-tailed version tells if there is a significant difference between the values of the input data sets, regardless of its direction. We wanted to check the direction of difference explicitly (i.e. that values in subset A are greater than values in subset B), therefore we selected the one-tailed test. By using this approach, we would be able not just to reject the null-hypothesis, but to prove the assumed alternative hypothesis as well.

The most important result of this statistical test is the well-known p-value, indicating the probability of the result being at least as extreme as the observed, provided that the null-hypothesis is true. In the results section we present these p-values for the analyzed systems. We performed the test by employing the `wilcox.test()` function in the R [23] statistical software package.

G. Discussion

This section describes why we chose the presented approach which might help the reader to better understand it.

In this study we treat the used quality model as a black box, in a sense that both the theory behind it and the calculation itself was executed before independently from this study. For this level of abstraction it is enough to know that if all of the metrics increase, then the maintainability decreases; if they all decrease, then the maintainability increases; and if some of the metrics increase and others decrease, then the direction of the maintainability change depends on the benchmark and the aggregation algorithm applied by the model.

The quality model analyzes a complete system, i.e. a certain revision of a software system and calculates the result. We have no details about the individual software components, like source code elements. Furthermore, by comparing two subsequent revisions, we have no information about the individual impact of each file participating in the commit. Therefore the direction of the maintainability change is the compound result of the individual files. This is the reason why it was necessary to define a compound measure for the churn values as well. Averaging the file churns was the most straightforward choice, as the other part of the data series was also of averaging nature. Other approaches, e.g. considering just the maximum, would not be proper here because they would differ in nature from the already available information.

The way we calculated the churn values evolved over time. First we considered only the number of different places the user modified the code, then the information about the number of touched lines were taken into consideration. The next step could be to determine if a sequental file addition and file removal is a line modification or really a line removal and line insertion. There are no exact methods to determine this information from the diff files; several studies deal with this problem. A study [24] presents such a heuristics with their measurements of about 95% precision and recall. But based on our previous experiences we would expect similar, somewhat even lower p-values, therefore we decided not to implement such a heuristic to keep the presented straightforward approach.

It was a question how to deal with the commits with no traceable maintainability changes. We were thinking about the possibilities like adding them to one of the subsets or to both of them, but we rejected these ideas. We could not formulate any ideology why we should add them to either of the subsets, but adding to both of the subsets would effect the same result with lower strength. We can picture the situation with the following metaphore. We have two bottles filled with water and the question is which one is the colder. If we pour the same amount and same temperature water into both bottles, the relative order based on temperature would not change, but the difference would be lower.

It was not a trivial problem how to deal with deletions. Should it increase or decrease the code churn or maybe have no effect on it? Based on the way how other researchers treated this operation we chose to handle them similar to file additions.

Somehow aggregating the source code specific code churn values within each commit is inevitable; however, there are other possibilities as well. Beyond using the average, another candidate of aggregation could be the calculation of geometric mean of the values. Furthermore, in this case it should be taken into account that files just added (i.e. having past 0 churn value) must not to considered in the calculation (because multiplying anything with 0 would result in 0, and the n^{th} root will be 0 as well). Another dimension of the possibilities is whether the actual commit should be considered or not in the churn value calculation. We chose the more common average over geometric mean, and decided not to consider the actual commit, in order to strengthen the findings; however, we executed the tests with all the other combinations. We got similar p-values in all cases, all of them being lower than 0.05, and in several cases even lower than the results presented in this paper.

H. *Randomized Cross-Checks*

We wanted to be sure that the results are really valid, therefore we executed the tests on random data as well. This was done in the following way:

- we kept the churn values in their original order, and
- we also kept the signs of the maintainability changes, but we permuted randomly the order of the revisions they were originally assigned to, just like a pack of cards. The `sample()` R function was used to permute the order.

This way we got the same churn values and the same maintainability change signs as in the original case, but the connection which churn value belongs to which sign is randomized. We performed randomization several times, permuting the already permuted series. We executed the same analysis with the randomized data and checked the appropriate random results as well to be able to assess the significance of our original results.

As a result, we get several p-values performed on randomized data.

IV. Results

In this section the execution results of the previously detailed method are presented. First, in section IV-A we introduce the software systems we executed the tests on. Then in section IV-B the actual test results are presented. Section IV-C contains the results of the randomized data, for cross-checking the validity of the main test. Finally, section IV-D describes some important additional notes that help properly interpreting the results.

A. *Examined Software Systems*

The analysis was performed on the source code of the following four software systems (one of them is an industrial one and three of them are open-source):

- **Ant** – a command line tool for building Java applications.[2]
- **Gremon** – a proprietary greenhouse work-flow monitoring system.[3]
- **Struts 2** – a framework for creating enterprise-ready java web applications.[4]

[2]http://ant.apache.org
[3]http://www.gremonsystems.com
[4]http://struts.apache.org/2.x

- **Tomcat** – an implementation of the Java Servlet and Java Server Pages technologies.[5]

These systems were selected prior to the current work, and these are the same systems we used in other studies related to our research series [3]–[5].

Table IV
ANALYZED SYSTEMS

	Ant	Gremon	Struts 2	Tomcat
Maximum logical lines of code	106,413	55,282	152,081	46,606
Number of commits	6,102	1,158	1,749	1,292
Maintainability increases	1,482	456	498	269
Maintainability no change	3,051	365	710	704
Maintainability decreases	1,569	337	541	319

Table IV shows some basic properties of the systems. Around one third to half of the commits did not show a traceable maintainability change. It can also be observed that the number of commits causing maintainability decrease tend to be higher than the number of commits increasing maintainability. Figure 2 illustrates how the maintainability of the analyzed systems changed over time (the big ups and downs on the diagrams are typically caused by adding the source code of complete components to the analyzed one developed outside the analyzed source control system).

For demonstration purposes, we show a short example for all the 3 types of maintainability changes; all of the examples were taken from project Tomcat. The layout of some of the code snippets below are changed in order to fit the column width. All the directions of changes were correctly calculated by the used quality model.

The first example is revision 640897 of the source file `util/http/Parameters.java`. Its original content was the following:

```
public void processParameters( MessageBytes data,
                               String encoding ) {
  if( data==null || data.isNull() || data.getLength() <= 0 )
    return;

  if( data.getType() == MessageBytes.T_BYTES ) {
    ByteChunk bc=data.getByteChunk();
    processParameters( bc.getBytes(), bc.getOffset(),
                       bc.getLength(), encoding);
  } else {
    if (data.getType()!= MessageBytes.T_CHARS )
      data.toChars();
    CharChunk cc=data.getCharChunk();
    processParameters( cc.getChars(), cc.getOffset(),
                       cc.getLength());
  }
}
```

Content after modification:

```
public void processParameters( MessageBytes data,
                               String encoding ) {
  if( data==null || data.isNull() || data.getLength() <= 0 )
    return;

  if( data.getType() != MessageBytes.T_BYTES ) {
    data.toBytes();
  }
  ByteChunk bc=data.getByteChunk();
  processParameters( bc.getBytes(), bc.getOffset(),
                     bc.getLength(), encoding);
}
```

[5]http://tomcat.apache.org

The code has been obviously simplified as indicated also by the maintainability increase calculated by ColumbusQM.

For demonstrating maintainability decrease we selected revision 647307, where source file `util/buf/B2CConverter.java` was affected as follows:

```
public  final void recycle() {
}
```

After modification:

```
public  final void recycle() {
  try {
    // Must clear super's buffer.
    while (ready()) {
      // InputStreamReader#skip(long)
      // will allocate buffer to skip.
      read();
    }
  } catch(IOException ioe){
  }
}
```

Originally it was an empty (unimplemented) function. Considering that the function is only a few lines, the implementation is rather complex (compared to a typical sequential 3 lines long function), containing a coding rule violation (silently catching an exception) as well.

B. Results of the Statistical Tests

Table V shows the results of the Wilcoxon rank test, described in Section III-F.

Table V
RESULTS

System	p-value	Significance
Ant	0.00235	very strong
Gremon	0.00436	very strong
Struts 2	0.00018	very strong
Tomcat	0.03616	strong

We found three very strong (Ant, Gremon and Struts 2) and a strong (Tomcat) evidence for rejecting the null-hypothesis and accepting the alternative one.

Answer to the Research Question: we reject the null-hypothesis, and accept the alternative one, that maintainability increases are mostly related to lower cumulative code churn values, while maintainability decreases are related to higher cumulative code churn values.

C. Randomized Cross-Check Results

To exclude the possibility of getting the above promising results by accident, we executed a random cross-checks 10 times for each project. Table VI contains the resulted p-values. As it was expected, most of the p-values fall between 0.1 and 0.9, indicating either not significant or contradictory results.

These random check results ensure that the original calculation is correct, and the original results are realistic. With the help of this cross-check we can reject the null-hypothesis with higher confidence.

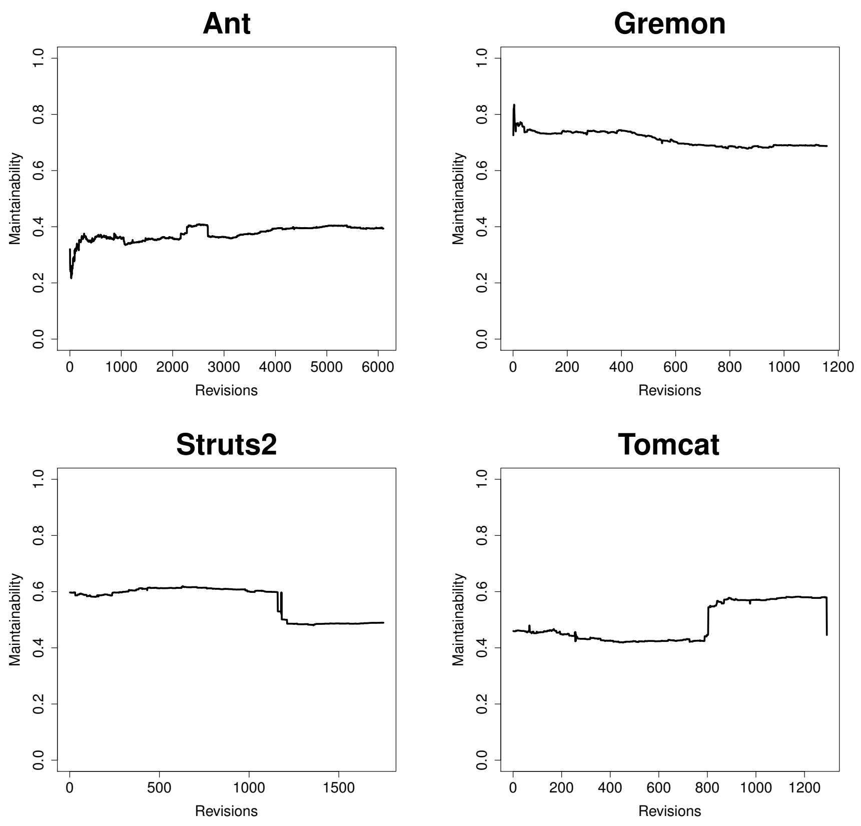

Figure 2. Maintainability timelines

Table VI
RESULTS OF RANDOMIZED DATA

Execution	Ant	Gremon	Struts 2	Tomcat
1	0.651	0.339	0.336	0.567
2	0.517	0.088	0.891	0.375
3	0.506	0.591	0.549	0.856
4	0.371	0.083	0.188	0.758
5	0.106	0.754	0.577	0.114
6	0.651	0.435	0.286	0.477
7	0.230	0.829	0.459	0.077
8	0.816	0.724	0.250	0.789
9	0.116	0.881	0.838	0.935
10	0.222	0.400	0.933	0.548

D. Discussion

It is quite easy to misinterpret the results, therefore in this subsection we present some important notes for a more adequate interpretation.

A frivolous understanding of the results would be the following: “the more you work the more you err”. We, on the other hand, state that if one modifies a source file which has been intensively modified in the past, then it is more likely to make it even more complex, compared to modifying source files less intensively modified earlier.

Other possible misinterpretation could be the following: “the more a file has been changed, the more complex it will be.” We consider this statement trivial (see Lehman’s law of increasing complexity [25]) and this is not what we want to express; our statement is much stronger. For example, let us consider the McCabe cyclomatic complexity (McCC) in the following case. There are 2 source files: `A.java` with a longer modification history, having a higher cumulative code churn value, and its current McCC value is 7; and `B.java` with shorter modification history, having a lower cumulative code churn value, with current McCC value of 3; both before the actual commit. On this level of abstraction we state that it is more likely that the complexity of `A.java` will increase to 8 due to the effect of a future commit on that file, than the likelihood of complexity `B.java` being increased to 4 caused by a future commit on that file.

Another important note, which is a significant difference compared to the existing works, is the following. We examined the impact of cumulative code churn on the maintainability of the source code and not on the defects. Although we did not check the number of defects revealed later, we considered how the code maintainability is likely to change. The correct interpretation of this (i.e. the notion maintainability instead of error) would be the following: if a source code fragment has been extensively modified in the past, the next modification affecting it is more likely to make it even more problematic (more complex, introduce more coding rule violations, etc.) than those changes affecting source code that has not been modified so extensively.

As the test was performed from the maintainability change perspective, we should be careful when formulating the final conclusion. Even the above conclusion is not entirely precise. The absolutely correct conclusion can be stated as follows: if the maintainability was increased as the result of the *current modification* (e.g. the average complexity of the developed system has been reduced), then it is more likely that the modifications were performed on files with smaller cumulative code churn values (i.e. files that have been less intensively modified *in the past*) than churn values of files of a commit decreasing the maintainability.

Lastly, it is not stated (and not true, of course) that all the cumulative code churn values related to maintainability increase are less than all the cumulative code churn values of maintainability decrease. The correct summary of the results is the following: the cumulative code churn values related to maintainability decrease are significantly larger than those related to maintainability increase. If we executed the t-test (instead of the Wilcoxon test), which compares the averages, we could formulate the following straightforward statement: the average cumulative code churn values of the two subsets differ. With the help of Wilcoxon test such an easy statement cannot be formulated. In this case the t-test is unfortunately not applicable, as the average operation is very sensitive to the extreme values, furthermore, it assumes a normal distribution of the underlying data. Therefore this trade-off resulted in a somewhat more complicated interpretation.

V. Threats to Validity

Although the results are promising, there are some factors that threaten the validity of our findings.

The base hypothesis was tested on a relatively low number of data. We used only 4 systems with available data, and large enough number of observations. Furthermore, only two of them had more than one thousand commits causing maintainability changes. Higher number of software systems having longer revision histories could provide more accurate results.

The revision history was complete only in one case, furthermore, only the main branches were analyzed. More accurate results could be gained if the whole revision history would be analyzed, furthermore, even all the branches.

We eliminated the commits resulting non-traceable maintainability changes. The cardinality of these operations is between one third and half of the full number of commits. This is a relatively great amount of data excluded. An enhanced model considering also the commits with no maintainability change could provide slightly different results. However, we do not expect a different final conclusion even in the case of such a model.

The code churn calculation in case of line modifications could be more precise. Based on our earlier experiences, with exact code churn calculation we would expect an even slightly lower p-values.

There are several quality models, and there is no such unique model which is accepted by the whole industry. Using another model could provide a different result. We know that no quality model (including the one used by us) is perfect. However, as most product quality models rely on a similar source of information (i.e. source code metrics) we do not expect that the results are so much dependent on the actual quality model used.

This study focuses on the Java programming language only, but according to our strong assumption the results would be valid for other programming languages as well. However, focusing to Java exclusively threatens the generality of the results.

The cumulative code churn values are based on files, but the statistical test is based on commits. Therefore we needed a conversion from file level values to commit based values. We selected the straightforward average value; however, one could argue that another approach could be better (e.g. removing the lowest and highest 10%, taking just the median etc.). We executed tests with median and geometric mean as well, but we got basically the same results.

VI. Conclusions and Future Work

In this work we examined if cumulative code churn has any connection with source code maintainability. We were interested in the following: considering only the history of the files which were just committed, i.e. not considering the current modification, are we able to tell anything about the possible outcome of the current commit from maintainability point of view?

We divided the commits based on the sign of maintainability changes they cause (i.e. if it was positive or negative), and compared the cumulative code churn values of the commits of both sets. We found that the cumulative code churn values belonging to negative maintainability changes are significantly higher than those of belonging to positive maintainability changes. We executed the comparison test on 4 software systems; all of the cases resulted in a strong correlation (3 of them was very strong).

Despite our investigation, we still cannot answer with an absolute unambiguous yes to the thought provoking question, if it is possible to narrow down where code erosion occurs, but – similarly to our earlier findings – we can conclude that some types of commits are more likely to erode the source code than the others. Specifically in this case we found that great amount of past code changes (expressed by cumulative code churn) are correlated with code decay; and that modifications performed on intensively changing code have worse effect on software maintainability than those affecting less intensively modified code. Therefore we can conclude that committing files with higher cumulative code churn values (i.e. those of longer change history) is more likely to result in negative maintainability change, compared to those of lower cumulative code churn values.

Note that typically the sizes of maintainability changes caused by commits are individually not huge; rather relatively small, but they are significant. Not a simple commit will ruin the whole maintainability, but it decreases slowly and almost certainly; like gaining weight. Similarly to losing weight, the best way is to pay special attention to problems, try to avoid the wrong tendency and try to make one step in the good direction, but doing that persistently.

Unfortunately, a project rarely allocates resources for refactoring. Cumulative code churn values could help to better allocate these: the best would be to start with files with highest churn values.

This study is an important step of a longer term study. We showed in a previous article [3] that connection between version control operations and maintainability change does exist. Then we analyzed the operations one by one in further studies [4], [5]. Up to now we considered the number of operations only. In this study we extended the research to other version control data: the file name and change history.

A deeper analysis of the code churn is still open, for example treating the deletions differently, or focusing directly on commits causing no maintainability change. Furthermore, the analysis of the combination of code churn and other information (like time factor) could also be interesting.

In the next steps we plan to take even more version control information data into consideration, like author, date, or the comment. As a final step, we plan to aggregate the results, and then implement a tool which identifies the hot areas of the source code of the analyzed system. An IDE plug-in, which visually highlights these areas could be useful for architects, and also warn the developers automatically.

In longer term we plan to take into consideration other software development data, like integrated development environment interactions or data found in issue tracking systems. Moving to other languages and comparing them with Java could also be an interesting future research topic. For example, the quality model used for this research was adopted to C# by Hegedűs [26].

Our long term goal is to fine-tune the formula of code erosion as much as possible in order to understand why it happens, and with this information in hand we could give hints how to avoid it with the least additional effort.

Acknowledgment

This research work is partially supported by the European Union projects titled "Telemedicine-focused research activities on the field of Mathematics, Informatics and Medical sciences", project number: TÁMOP-4.2.2.A-11/1/KONV-2012-0073 and "REPARA – Reengineering and Enabling Performance And poweR of Applications", project number: 609666, co-funded by the European Social Fund.

References

[1] ISO/IEC, *ISO/IEC 9126. Software Engineering – Product quality 6.5*. ISO/IEC, 2001.

[2] T. Bakota, P. Hegedűs, G. Ladányi, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy, "A cost model based on software maintainability," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 316–325.

[3] C. Faragó, P. Hegedűs, Á. Z. Végh, and R. Ferenc, "Connection between version control operations and quality change of the source code," *Acta Cybernetica*, vol. 21, pp. 585–607, 2014.

[4] C. Faragó, P. Hegedűs, and R. Ferenc, "The impact of version control operations on the quality change of the source code," in *Computational Science and Its Applications (ICCSA)*. Springer, 2014, pp. 353–369.

[5] C. Faragó, "Variance of source code quality change caused by version control operations," *Acta Cybernetica*, vol. 22, pp. 35–56, 2015.

[6] T. M. Khoshgoftaar and R. M. Szabo, "Improving code churn predictions during the system test and maintenance phases," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 1994, pp. 58–67.

[7] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy, "A probabilistic software quality model," in *2011 27th IEEE International Conference on Software Maintenance*. IEEE, 2011, pp. 243–252.

[8] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *Proceedings of the 7th International Symposium on Software Reliability Engineering*. IEEE, 1996, pp. 364–371.

[9] J. C. Munson and S. G. Elbaum, "Code churn: A measure for estimating the impact of code change," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 1998, pp. 24–31.

[10] M. C. Ohlsson, A. Von Mayrhauser, B. McGuire, and C. Wohlin, "Code decay analysis of legacy software through successive releases," in *Proceedings of the Aerospace Conference*, vol. 5. IEEE, 1999, pp. 69–81.

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

[12] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[13] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference onSoftware Engineering (ICSE 2005)*. IEEE, 2005, pp. 284–292.

[14] ——, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 2007, pp. 364–373.

[15] S. A. Ajila and R. T. Dumitrescu, "Experimental use of code delta, code churn, and rate of change to understand software product line evolution," *Journal of Systems and Software*, vol. 80, no. 1, pp. 74–91, 2007.

[16] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[17] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 83–92.

[18] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 177–187.

[19] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 682–691.

[20] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1335–1382, 2014.

[21] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[22] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[23] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.R-project.org/

[24] G. Canfora, L. Cerulo, and M. Di Penta, "Identifying changed source code lines from version repositories." in *MSR*, vol. 7, 2007, p. 14.

[25] M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle," *Journal of Systems and Software*, vol. 1, pp. 213 – 221, 1980.

[26] P. Hegedűs, "A probabilistic quality model for C# – an industrial case study," *Acta Cybernetica*, vol. 21, no. 1, pp. 135–147, 2013.