Maintainability of Source Code and its Connection to Version Control History Metrics

Csaba Faragó

Department of Software Engineering
University of Szeged

Szeged, 2016

Supervisor:
Dr. Rudolf Ferenc

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
OF THE UNIVERSITY OF SZEGED

University of Szeged
PhD School in Computer Science

"Programs must be written for people to read,
and only incidentally for machines to execute."

— Harold Abelson

Preface

Recently I found a plain old 650 MB compact disc, burnt at 20th June, 2001. On that disc I have found some materials about the preparation of the application to the Doctoral School of Computer Science in Szeged. I was already a co-author of a few ongoing or already presented articles; some of them were on that disc. However, at that time I made a hard decision: I withdrew, and my professional life changed its direction towards the industry.

Ten years later I was thinking about proceeding with the research. I have learned that there was the possibility of individual preparation. It was not necessary to follow the three-year study program, and I decided to renew my research activities in this form, parallel to my work. Most of the research was done on the train between Nagykőrös (my home town) and Budapest (where my work place is located).

Chronologically first, I would like to mention the great co-work with co-authors of the articles done on the field of dynamic slicing of source code. The excellent supervision of Dr. János Csirik and Dr. Tibor Gyimóthy helped all of us to be efficient in the research. I really enjoyed the great common work with Dr. Árpád Beszédes, Dr. Tamás Gergely and Zsolt Mihály Szabó. I gained very useful programming experiences with them.

After a decade Dr. László Vidács helped me in contacting the university again; without his help I might not reach this goal. I am grateful to Dr. Rudolf Ferenc for his big help at the restart and great supervision throughout all the research. The continuous cooperation with Dr. Péter Hegedűs was very fruitful from the very beginning; his great ideas were always helpful, especially in the cases when the research came into difficulties. The cooperation was excellent also with my casual co-authors: Dr. Tibor Bakota, Gergely Ladányi and Ádám Zoltán Végh.

I am in the lucky situation that my current workplace, the Lufthansa Systems Hungária Kft., is open for novelties and cutting edge technologies, therefore it provides a good atmosphere for learning and research. Furthermore, without their financial support and free time I would not have been able to participate in two great conferences.

I am indebted to Bruce Derby for grammatical revision of the thesis. Last but not least, this could not have been done without the encouragement, patience and support of my wife Lilla.

I thank everybody who helped my research activities!

Csaba Faragó, 2016

Contents

List of Tables

List of Figures

To my family.

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."

— Martin Fowler

1 Introduction

If a program would be written exclusively for the computers, and it would be 100% sure that no human will ever see the source code later, then the code quality would not be important. As indicated by the quote above, that would be indeed an easier situation. In that case also this thesis would deal with some completely different topic.

However, this is not true: the developer spends most of her/his work time on *reading* source code, and not *writing* it. Therefore, the developers should pay attention to the maintainability of the source code: to help the subsequent readers, who might be even the same developer after a longer while.

Software code quality is very important, because a too complex, hard-to-maintain code results in more bugs on one hand, and makes the further development more expensive on the other hand. Developers typically focus on the problem: to solve it, make it work, and the importance of the code quality in time pressure is frequently secondary. The motivation of the studies behind this thesis is to help developers to create source code which is easier to maintain.

The thesis consists of two main topics: *program slicing* and the *impact of version control history metrics on maintainability*. There is a strong connection between them: the *maintainability of program source code*.

One of the usages of **program slicing** is to improve the maintenance of the source code, helping the programmer by highlighting the relevant parts of it. With this technique the developer can eliminate the source code irrelevant from a certain problem viewpoint, and observe only those statements which really influence the erroneous part. In this thesis we focus on the unstructured statements handling in a certain dynamic program slicing algorithm. As there is a great number of slicing algorithms, and we deal only with a certain one; furthermore, that algorithm had several aspects and the unstructured statements handling is one of them, we can state that this part of the thesis is like a cog in the machine.

In the topic of **version control history analysis** we try to find evidence why code maintainability decreases. The *seventh law of Lehman* is about *declining quality*, and it states that *the quality of a real-world software system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes.* In

the second part of this thesis we deal with the connection between this code decay and some version control history metrics. Although both the topic of maintainability and the software repositories mining are thoroughly analyzed research fields, we are not aware of any study which deals with the, strictly speaking, connection between version control history metrics and the maintainability of the source code. So in this thesis we present a pioneer work of this young research field.

Beyond the logical one, technical connection also exists between the two topics; we illustrate this in Figure 1.1. The software developed during the study of dynamic program slicing was built directly on the top of a C++ analyzer tool called CAN. Later it was extended, renamed to Columbus, and it was presented by Ferenc et al. [47]. A few years later a Java analyzer was also implemented. The Columbus Quality Model was presented by Bakota et al. [11]. It was built on the top of the Columbus Java analyzer, and it calculates the maintainability of program source code. The research about version control history analysis uses the results calculated by the Columbus Quality Model.

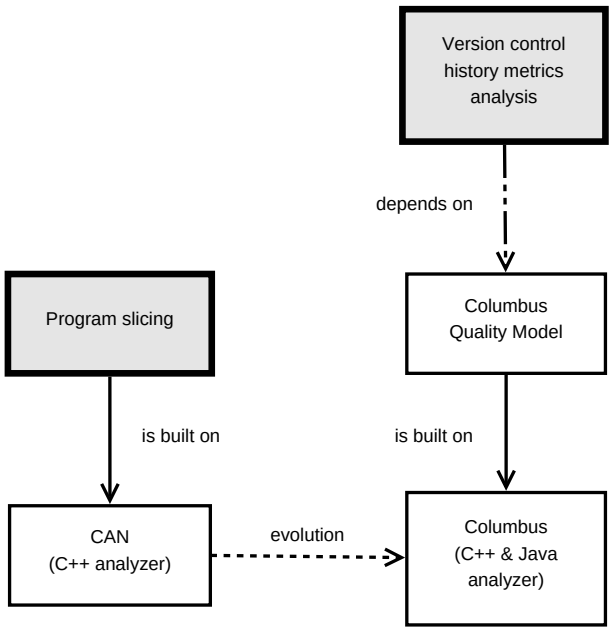

Figure 1.1. Technical connection between the two research fields

1.1 Structure of the Thesis

In this section (Section 1) we provide an overview of the results. We summarize the results, emphasize the author's contributions to those, and list the related publications.

The **first part** of the thesis contains one single chapter (Chapter 2). In that we provide an overview about the topic of dynamic program slicing of C programs, and present the results of the unstructured C statements handling.

The **second part** consist of three chapters. First, in Chapter 3 we introduce the version control history based maintainability analysis. Then in Chapter 4 we describe how version control operations affect the maintainability of the source code. First we show that connection between version control operations and maintainability

change exists. Then we deal with how each version control operation type affects the maintainability change, in terms of absolute change and the variance of the change. Finally, we present a new visualization means adequate for illustrating the results. In the last chapter of the second part (Chapter 5) we present how various version control related metrics are connected to maintainability. We check code ownership and code churn in detail. Finally we present 6 version control based metrics, and check their relation to maintainability one by one.

In Chapter 6 we conclude the thesis. In the Appendix we summarize the findings in English and Hungarian.

1.2 Summary of the Results

In this section we summarize the results: we list all the relevant publications and indicate the author's contribution to each one.

Part I – Dynamic program slicing

The first part of the thesis deals with dynamic slicing of C programs. A slice consists of all statements and predicates that might affect a set of variables at a program point. Slicing algorithms can be classified according to whether they only use statically available information or dynamic information as well, to static and dynamic one. In Thesis Point 1 we deal only with dynamic program slicing.

1. **Unstructured C statements handling in a dynamic slicing algorithm**

   Gyimóthy et al. [58] introduced a dynamic slicing method, which forward computes the backward slices. Unlike earlier dynamic slicing approaches, which require memory proportional to the execution history, the presented method's memory requirement is in proportion with those requirements of the original program. Therefore theoretically it is adequate to handle cases with very long execution history.

   However, the algorithm in its original form was inappropriate for slicing real programs, because it handled assignment, conditional and loop statements only. In their study Beszédes et al. [18, 19] adopted the algorithm on the C programming language. They solved several issues, e.g. the function calls or the pointer handling.

   One of the issues to be solved was the handling of unstructured statements in the C programming language, which were the following: `goto`, `break`, `continue` and `switch-case-default`. Other studies rate `return` statement as unstructured statement as well; we solved that problem along with function calls.

   In our solution [39, 40] we introduced so-called label variables, which get value at the point of execution, and the dependent lines of the source are those located after the label. In case of `goto` the dependent statements are all the statements after the declaration of the label, within the function. In case of `break` the dependent statements are all the statements after the related code block (e.g. after the `while` block). In case of `continue` the dependency should be introduced from the first statement of the related code block until the end of the function. This also means that the `continue` always depends from itself. In case of `switch-case-default`, the `switch` statement should be handled with a

predicate variable, similarly to e.g. in case of the `while` statement. If at least one internal statement is part of the result, then all the `case` labels, along with the `default`, should be put into the result.

Later, we extended the method [17], where we defined the relevant slice as the union of all possible executions. In case of significant code coverage the size of the resulting slice was a fraction of the result calculated by a static program slicing tool.

The author's contributions

The author elaborated and presented the details of the unstructured statements handling (`goto`, `break`, `continue`, `switch-case-default`). He participated in the implementation and he performed the tests.

Related papers

[39] **Csaba Faragó** and Tamás Gergely. Handling the Unstructured Statements in the Forward Dynamic Slice Algorithm. In *Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST)*, pages 71–83, 2001.

[40] **Csaba Faragó** and Tamás Gergely. Handling Pointers and Unstructured Statements in the Forward Computed Dynamic Slice Algorithm. *Acta Cybernetica*, 15(4):489–508, 2002.

[17] Árpád Beszédes, **Csaba Faragó**, Zsolt Mihály Szabó, János Csirik and Tibor Gyimóthy. Union Slices for Program Maintenance. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM)*, pages 12–21. IEEE Computer Society, 2002.

Part II – Version control system based maintainability analysis of the code

In the second part of the thesis we deal with version control system based maintainability analysis of the source code.

According to studies and experiences, code quality, especially code maintainability, has direct impact on development costs. On the other hand, the quality of the software source code declines if we do not invest to improve it. We were motivated to find out where and why this code erosion occurs: are there typical developer interactions causing similar change in the maintainability. Knowing it might help preventing the code decay.

In Thesis Point 2 we deal with the version control operations: how file addition, update and deletion affects the maintainability. In Thesis Point 3 we deal with other information found in version control system: how the level of code ownership and code churn affects the maintainability.

2. **Connection between version control operations and maintainability**

   In this thesis point we present the connection found between the number of operations in a version control commit and the maintainability change caused by that commit. First we present how we found the existence of the connection between these two independent series of data. Then we show how we revealed the effect of each version control operation to the value of maintainability change and variance on maintainability change. We illustrate the results with diagrams.

   We divided this thesis point into three parts.

2.A **Existence of the connection between version control operations and maintainability**

First we showed that a connection between version control operations and maintainability really exists, in spite of the fact that the data are coming from different sources [45].

For every commit we determined the maintainability change, and based on this we partitioned them into three disjoint subsets: maintainability increase, no change, and decrease. On the other hand, we divided them based on version control operations into the following four categories: (D) commits containing file deletion; (A) commits not containing file deletion, but containing file addition; (U+) commits containing updates only, at least two; (U1) commits consisting of exactly one update operation. This two dimensional division forms a contingency table.

For four analyzed systems we performed the contingency Chi-Squared test, which tells if the difference between the expected and the actual distribution is significant. According to the significant results we stated that there was a connection between the version control operation and the maintainability.

The author's contributions

The author performed the categorization of the commits on version control operation basis, based on the results of a Principal Component Analysis. He applied the Chi-Squared test on the *maintainability change x version control operation based commit category* matrix. He implemented the algorithm using R, executed all the tests and evaluated the results. Interpretation and illustration of the results is also the work of the author.

2.B **Impact of version control operations on value and variance of maintainability**

We considered file additions, file updates and file deletions one by one, and checked their impact on the size [41] and the variance of maintainability change [34, 36].

In the algorithm we divided the commits into subsets considering the number of operations, and we compared the related maintainability change values. We defined seven divisions, which considered the presence, the absolute number, and the proportion of the actually examined operation. Then we compared the related maintainability change values using Wilcoxon-test, and the variance of those values using F-test.

The tests of the value comparison resulted that file additions improved, or at least less eroded the maintainability than file modifications, while the file updates mainly eroded them. We could not establish the effect of the file deletion because we received contradictory results.

At the variance check we concluded that the file addition and file deletion increased, while the file update decreased the variance. As the amplitude was much bigger than the absolute change, as a final conclusion we stated that it was recommended to place special attention on file additions.

The author's contributions

The author constructed the process of the methodology. This includes the idea of the 7 subdivision, the Wilcoxon rank test, the F-test execution, and the methodology of the evaluation of the results. The idea of converting the difference of maintainability values into a comparable absolute value is also the work of the author. He implemented the algorithm, executed the tests and performed the result evaluation. All the explanatory diagrams are done by the author, including the idea and the implementation.

2.C **Cumulative characteristic diagram and quantile difference diagram**

We invented two visualization methods, which were adequate for presenting some of the published results visually [37].

The input of the *Cumulative Characteristic Diagram* is a set of numbers. We sort them non-ascendant, and for every index we calculate the sum from the first one up to that point. The indices represent the x-coordinate, and the sums represent the y-coordinate. The characteristic is created by connecting these points with lines. On the Composite Cumulative Characteristic Diagrams we draw two or more Cumulative Characteristic Diagrams.

Using the *Quantile Difference Diagram* we can compare two sets of numbers. First we sort the elements of both sets in non-descending order. Then we take the values of every centile from both sets, pairwise. We calculate the difference of every pair. Using the centiles as the x-coordinate and the differences as y-coordinate, finally connecting the points we gain the Quantile Difference Diagram. If the original data contains outliers, it is recommended these values not to depict; the default implementation does not consider the lower and upper 5%.

The CCD is suitable for illustrating the Chi-squared test, the Wilcoxon-test and the variance test, while the QDD is suitable for illustrating Wilcox test and variance test.

The author's contributions

The idea of the Cumulative Characteristic Diagram and Quantile Difference Diagram is the work of the author. He implemented this in R, he is responsible for the `vudc` R package [38]. The usage possibilities of these diagrams, i.e. the illustration of Chi-squared test, the Wilcoxon-test, and the variance test are also the idea of the author. All the diagram generation and the maintainability analysis related case study is also done by the author.

Related papers

[45] **Csaba Faragó**, Péter Hegedűs, Ádám Zoltán Végh, Rudolf Ferenc. Connection Between Version Control Operations and Quality Change of the Source Code. *Acta Cybernetica*, 21(4):585–607, 2014.

[41] **Csaba Faragó**, Péter Hegedűs, Rudolf Ferenc. The Impact of Version Control Operations on the Quality Change of the Source Code. In *Proceedings of the 14th International Conference on Computational Science and Its Applications (ICCSA)*, volume 8583 Lecture Notes in Computer Science (LNCS), pages 353–369. Springer International Publishing, 2014.

[34] **Csaba Faragó**. Variance of Source Code Quality Change Caused by Version Control Operations. In *Proceedings of the 9th Conference of PhD Students in Computer Science (CSCS)*, 12–13, 2014.

[36] **Csaba Faragó**. Variance of Source Code Quality Change Caused by Version Control Operations. *Acta Cybernetica*, 22(1):35–56, 2015.

[37] **Csaba Faragó**. Visualization of Univariate Data for Comparison. *Annales Mathematicae et Informaticae*, 45:39–53, 2015.

3. **Connection between version control history metrics and maintainability**

In the third thesis point we go further in analyzing the information located in version control systems. Unlike earlier, here we considered which piece of information is related to which file, therefore making a connection between different commit operations.

First we examined the effect of the past modification intensity of the source code, and of the level of code ownership on the later maintainability changes. After that we defined six version control history metrics, and considering all of them one by one we checked their connection with the maintainability, and, as a cross-check, with the number of post release bugs.

We divided this thesis point into two parts.

3.A **Impact of code modifications and code ownership on maintainability**

We checked the effect of past cumulative code churn [43] and the number of contributors [42] on the maintainability of the actual commits.

We calculated for each file and revision from the very beginning, how many lines have been added and deleted all together. On a certain commit we averaged these values. We divided these values into two subsets based on the maintainability change of the related commit, if it decreases or increases it (we omit the commits related to neutral maintainability changes). Finally we compared the values using Wilcoxon-test.

We performed similar steps in case of code ownership analysis. There we checked how many different developers contributed to the file, and at certain commit we took their geometric mean. For the comparison here we also used Wilcoxon-test. We illustrated those using Quantile Difference Diagrams.

We gained that the past intensive modifications and the lack of clean code ownership resulted in the decrease of the maintainability.

The author's contributions

The author took major part in framing the methodology of cumulative code churn and code ownership analysis. He elaborated the details, implemented it in R, executed the tests and evaluated the results. The author created the examples and the helper diagrams.

3.B **Correlation between version control history metrics and maintainability**

We defined six version control history metrics, and checked their connection with maintainability [44]. These metrics were the following: cumulative

code churn, number of modifications, ownership, ownership with tolerance, code age, and last modification time.

For a certain version of the analyzed system we sorted the source files based on every metric. We determined the order of files on the Relative Maintainability Index [68, 69] basis; furthermore, as a cross-check, we determined the order based on the number of post-release bugs as well. Then we tested the similarity of these orders with help of the Spearman's rank correlation test.

As a result we got that higher intensity of modifications, the higher number of code modifications and developers (without and with tolerance), the older code and the later last modification date resulted lower maintainability and higher number of post-release bugs.

The author's contributions

The author elaborated the methodology of the version control history metrics analysis. He defined the 6 metrics. He implemented the version control history metric extractor in Java, published it on the Github (`https://github.com/maintainability/hotspot`) and maintains it. He collected the necessary version control history metric and bug data. He participated in the mathematical formalism of the Relative Maintainability Index calculator. He implemented the algorithm in R, executed the tests and evaluated the results.

Related papers

[42] **Csaba Faragó**, Péter Hegedűs, and Rudolf Ferenc. Code Ownership: Impact on Maintainability. In *Proceedings of the 15th International Conference on Computational Science and Its Applications (ICCSA)*, volume 9159 Lecture Notes in Computer Science (LNCS), pages 3–19. Springer International Publishing, 2015.

[43] **Csaba Faragó**, Péter Hegedűs, and Rudolf Ferenc. Cumulative Code Churn: Impact on Maintainability. In *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 141–150. IEEE Computer Society, 2015.

[44] **Csaba Faragó**, Péter Hegedűs, Gergely Ladányi, and Rudolf Ferenc. Impact of Version History Metrics on Maintainability. In *Proceedings of the 8th International Conference on Advanced Software Engineering & Its Applications (ASEA)*, pages 30–35. IEEE Computer Society, 2015.

[69] Péter Hegedűs, Tibor Bakota, Gergely, Ladányi, **Csaba Faragó**, and Rudolf Ferenc. A Drill-Down Approach for Measuring Maintainability at Source Code Element Level. *Electronic Communications of the EASST*, pages 1–21, 2013.

In Figure 1.2 we illustrate how the publications and the topics presented in the second part of the thesis are related to each other. The left-hand side and the right hand side big rectangle illustrate the second and the third thesis points, respectively. The inner rectangles represent topics (i.e. publications), each containing a key phrase. The solid arrows represent the evolution of the research, while the dashed ones illustrate helper-like connections.

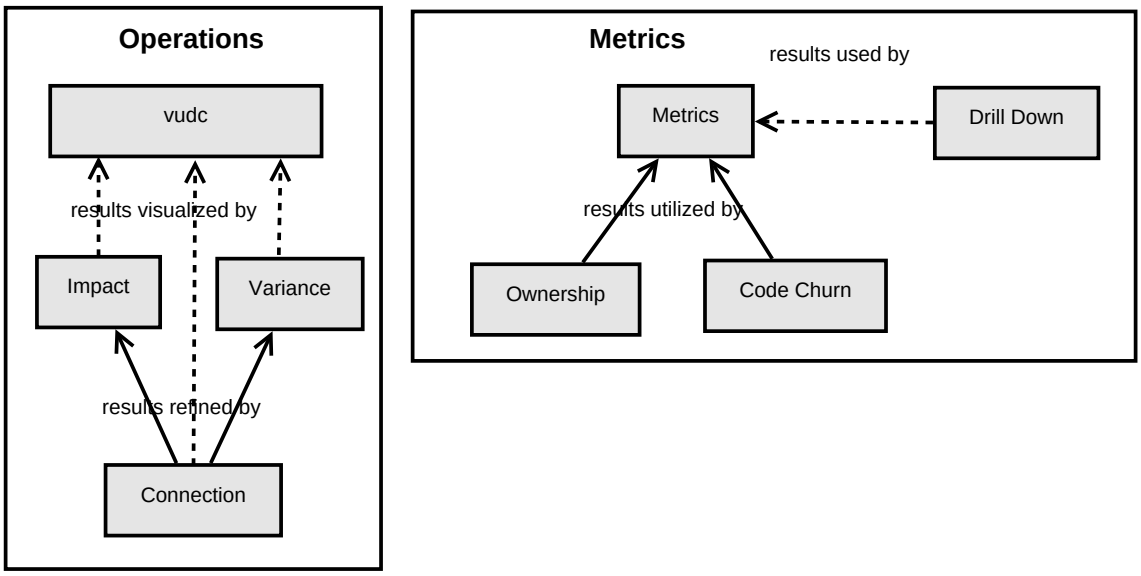

Figure 1.2. Overview of the publication related to maintainability analysis

1.3 Thesis Points Summary

In Table 1.1 we summarize the publications related to each thesis point.

	[39]	[40]	[17]	[45]	[41]	[34]	[36]	[37]	[43]	[42]	[44]	[69]
1.	●	●	●									
2.A				●								
2.B					●	●	●					
2.C								●				
3.A									●	●		
3.B											●	●

Table 1.1. Thesis points and supporting publications

Part I

Program Slicing

"Computer Science is no more about computers than astronomy is about telescopes."

— Edsger Dijkstra

2

Thesis Point 1: Unstructured C Statements Handling in a Dynamic Slicing Algorithm

Here we deal with a dynamic program slicing algorithm, with the focus on the unstructured statements handling.

In Section 2.1 we highlight the most important papers related to this topic. In Section 2.2 we provide an overview of the dynamic slicing algorithms. Afterwards, we present a certain dynamic slicing algorithm, and an extension of it, suitable for the C programming language. Neither the base algorithm, nor the base idea of its extension is the work of the author; however, they are necessary for understanding the author's contributions. Finally, we present the union slices algorithm, which is a generalization of the extended dynamic slice methodology, suitable for slicing real C programs. The author contributed to this work by taking part in the implementation, especially in the expression handling part, and by performing the tests.

In Section 2.3 we describe what the author contributed to the extended algorithm. This is the unstructured statements handling (`goto`, `break`, `continue` and `switch-case-default`) in C programming language. In other studies authors rate the `return` statement as unstructured statement [3, 113]; we, on the other hand, solved that problem along with function call handling.

It was necessary to solve other problems as well in order to adopt the algorithm to C programming language, like pointer handling, inter-procedural dependencies and so on. Detailed presentation of these aspects are out of the scope of this thesis.

In Section 2.4 we present some experimental results. The author took part both in implementation and in execution of the tests; however, the evaluation of the results was not done by the author.

Finally, in Section 2.5 we summarize the results.

Figure 2.1 provides an overview of how these parts fit together, and where the author's contribution is located in the system. The author has no contribution to the base algorithm, illustrated by the innermost dark rectangle. Light squares illustrate the

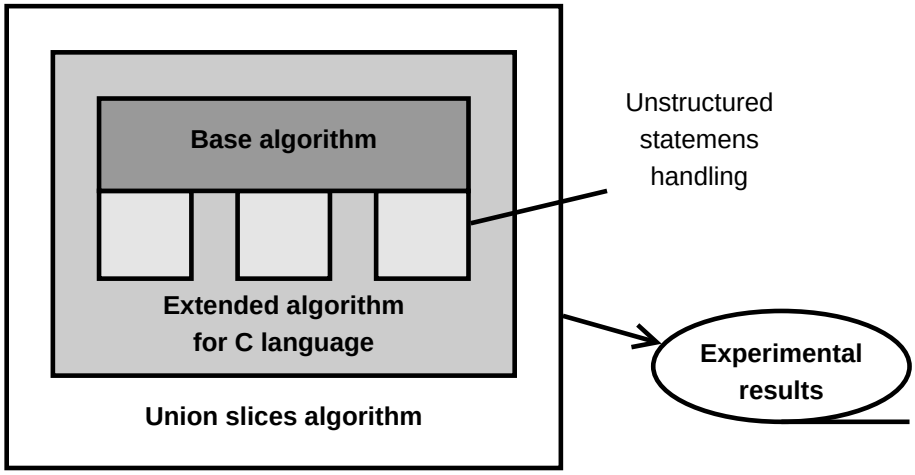

Figure 2.1. Overview of the forward computing dynamic slicing algorithms

C programming language specific extensions. One of them illustrates the unstructured statements handling. Note that the other rectangles just illustrate that there were other issues which had to be solved, and does not reflect the real number of them.

The big rectangle consisting of the base algorithm and their extensions illustrates the extended algorithm for C language. The outermost rectangle illustrates the union slices algorithm. The experimental results presented in this chapter are related to this algorithm.

Note that the figure has been created for illustration only. There is no connection between the size or tone of the objects and their importance.

2.1 Related Work

The research area of program slicing is very large. The number of citations of the "base article" [121] is close to 4000, therefore we can say that thousands of papers already appeared related to this topic. There are dozens of papers having a number of citations over 100.

The vast majority of the papers appeared up to 2002, when we published those articles on which this part of the thesis relies on. According to Google Scholar, the most relevant paper in this research field which was published after 2002 can be found – at the time of writing – on the 19^{th} place.

Excellent taxonomies exist [78, 116, 124, 113]; in this section we can just skim the surface of the related literature.

2.1.1 Overview

The term program slicing was proposed by Weiser [121] as follows: it reduces the program to a minimal form which still produces that behavior. He practically defined the backward slice, which consists of all statements and predicates that might affect a set of variables at a specific program point (the slicing criterion). Our approach is also based on this definition.

Two great taxonomy articles appeared in year 1995. Kamkar [78] characterized the actual state of the art of program slicing literature with the word "unwieldy". In the article the author attempted to bring order in the phrases. Clean and unambiguous

notions helps us better understand other papers, and it helps us to make our papers well understood by other researchers.

Tip [116] focused on various program slicing algorithms. He provided a classification and comparisons about the methodologies and papers appeared before. He focused especially on dynamic slicing algorithms, like ours. The number of the references in the bibliography (108) illustrates that the size of this area of research was already big two decades before writing this Thesis.

A more recent taxonomy was published by Xu et al. [124], summarizing hundreds of papers, tens of different algorithms, including both static and dynamic slicing algorithms and other slicing techniques as well. As this overview was published a few years after ours, it includes also our forward computing dynamic slicing algorithm as well.

In book Srikant and Shankar [113] present a taxonomy of program slicing methodologies; a separate section deals with slicing of unstructured programs. Our algorithm is also mentioned.

2.1.2 Slicing Methodologies

Ferrante et al. [48] defined the notion Program Dependence Graph, and that was adequate to calculate static program slices.

Korel and Laski [84, 85] proposed an extension of the original method using dynamic information. They defined the dynamic program slice as follows: is an executable part of the original program that preserves part of the program's behavior for a specific input with respect to a subset of selected variables, rather than for all possible computations. They refer what we call execution history as trajectory.

Agrawal and Horgan [4] also presented a dynamic slicing algorithm. They argued that a static slice contains all the statements of the program which might affect the value of a variable occurrence, but a dynamic slice contains all statements which really affect it. They presented a data structure dynamic dependence graph, which depends on the length of the program execution, therefore it is unbounded. They introduced the concept of a reduced dynamic dependence graph as well, which is proportional to the number of distinct dynamic slices arising during the current program execution. Our approach is also a dynamic one, but on the contrary, we do not use Dynamic Dependence Graph, but an approach whose memory requirement is practically proportional to the memory requirement of the original program.

Gallagher and Lyle [53] proposed extending the notion of program slice to decomposition slice. The original definition requires a variable and a line, but a decomposition slice requires a variable only, and it captures all computation on a given variable. Our approach uses the classic definition, i.e. using a line; however, in an extension we also used unions of several slices.

A hybrid static and dynamic slicing method was introduced in [118] to compute the quasi static slices where the value of some input variables was fixed while other variables vary.

Another example of a hybrid slicing method is the work of Rilling et al. [106]. They introduced a framework for the computation of both static and dynamic slices based on the notion of removable blocks (as in [86] and [83]). The objective of this work is again not to reduce the size of the parts of the program to be investigated, but to ease the computation of the dynamic slices by removing certain parts of the program, first using static slicing techniques.

Conditioned slicing, as proposed by Canfora et al. [27], computes a subset of the program which preserves the behavior of the original program with respect to a slicing criterion for a given set of execution paths. An extension of this notions, focusing on preconditions and post conditions as well, was presented by Harman et al. [64].

Another approach of reducing the slice sizes was the amorphous program slicing, as it was presented by Harman et al. [62]. This uses a general theoretical framework of program projections, with which equivalent but simpler program projections can be obtained, where the traditional static and conditioned slices can be seen as special kinds of projections. The main difference between conditioned slicing and amorphous slicing on one hand, and our approach on the other hand is that the former is primarily a static approach while trying to involve some dynamic information, but without actually performing the executions.

Zhang and Gupta [129] addressed the problem of the size of dynamic dependency graphs. They proposed a compact and rapidly traversable graph representation. The idea came from the recognition that all dynamic dependences (data and control) need not be individually represented. They identified sets of dynamic dependence edges between a pair of statements that can share a single representative edge. We, on the other hand, overcame the problem of the size of dynamic dependence graph by omitting it completely and following an approach without the necessity of that graph at all.

Szegedi et al. [115] adopted the forward computing dynamic slicing algorithm (presented in detail later in Section 2.2.2) to Java language. They investigated the sizes of backward and forward dynamic and union slices, and compared them to the corresponding static slices, considering real-world Java programs.

Vidács et al. [120, 119] focused on C/C++ preprocessing directives. The original algorithm considers the already preprocessed version (technically speaking, not the `.c` but the `.i` files, which are intermediary files during the compilation); in this paper the authors proposed slicing the preprocessing directives in the original source files in order to gain more precise results.

2.1.3 Unstructured Statements Handling in Slicing Algorithms

Typically papers presenting a novel slicing algorithm do not address the problem of unstructured statements handling, similarly to our case. As the unstructured statements handling is the main focus of this section of the thesis, we pay special attention to this topic.

Ball and Horwitz [13] and Choi and Ferrante [30] published similar solutions independently. They proposed to modify the Control Flow Graph and therefore the Program Dependence Graph as well.

Agrawal [3] proposed an alternative solution, leaving the original graphs intact, and using a separate graph to store the additional required information. This approach was efficient in case of structured jump statements, like `break`, `continue` and `return`. On the other hand, the resulting slices turned to be too large if a `goto` statement was present. Harman and Danicic [63] addressed this problem. They presented a post dominance based algorithm, which turned to be efficient in case of the presence of `goto` statements as well.

Harman et al. [65] considered three slicing attributes: the termination behavior, the size, and the syntactic structure. Along the termination dimension a slice may be either strong (the slice terminates if and only if the original program terminates) or

weak (if both terminate, then they produce the same result). Along the size dimension a slice may be equal to the Ottenstein's Program Dependency Graph [48] or larger than it. Along the syntactic structure dimension a slice might be syntax-preserving (the syntactic order of the remaining statements are preserved) on amorphous (where the slice might contain statements not present in the original program). This results 8 combinations. The authors proved that no unstructured statement handling algorithm existed for 2 of these combinations, while they presented algorithm for the remaining six combinations.

2.1.4 Related Theses

Other aspects of the C dynamic slicing algorithm summarized in this thesis were already part of other PhD theses.

In his thesis Árpád Beszédes [15] summarized the C++ source code analysis tool which was used by the dynamic slicing methodology at determining the static dependencies. In the second part the author presented the dynamic slicing algorithm of the C language, along with several aspects. He carved the surface of the unstructured statements handling as well, which serves the main part of the current thesis point.

In the second part of his thesis Tamás Gergely [54] provided a summary of program slicing algorithms, along with several implementation details of the C language dynamic slicing algorithm. He presented the Java implementation of the dynamic slicing algorithm as well.

2.2 Overview of the Forward Computation Dynamic Slicing Algorithm

2.2.1 Program Slices

To recall, a slice consists of all statements and predicates that might affect the variables in a set V at a program point p. A slice may be an executable program or a subset of the program code. In the first case the behavior of the reduced program with respect to a variable v and program point p is the same as the original program. In the second case a slice contains a set of statements that might influence the value of a variable at point p. Slicing algorithms can be classified according to whether they only use statically available information (*static slicing*) or compute those statements which influence the value of a variable occurrence for a specific program input (*dynamic slice*).

In many applications (e.g. debugging) the computation of dynamic slices is more preferable since it can produce more precise results, i.e. the dynamic slice is smaller than the static one. In this section we focus on dynamic slicing.

Gyimóthy et al. [58] introduced a method for the forward computation of dynamic slices i.e. at each iteration of the process, slices are available for all variables at the given execution point. However, the method presented was applicable only to very simple programs, with one procedure, scalar variables and simple assignment statements only. In study Beszédes et al. [18] presented the handling of the procedures and the implementation of the algorithm.

In our paper [40] we presented how to handle pointers and the jump statements in the C programs. In addition to the `goto` statement we solved the problem of `break` and

`continue` statements, which can be regarded as special cases of the `goto` statement. We also described the handling of the `switch-case-dafault` statement. That paper is the core of this thesis point.

2.2.2 The Base Algorithm

In some applications static program slices contain redundant instructions. This is the case for debugging, for instance, where we have dynamic information as well. Hence debugging may require smaller slices, which improves the efficiency of the bug finding process. The goal of the introduction of dynamic slices was to determine more precisely those statements that may contain program bugs, assuming that the failure has occurred for a given input.

```
     #include <stdio.h>
     int n, a, i, s;
     void main()
     {
 1.    scanf("%d", &n);
 2.    scanf("%d", &a);
 3.    i = 1;
 4.    s = 1;
 5.    if (a > 0)
 6.      s = 0;
 7.    while (i <= n) {
 8.      if (a > 0)
 9.        s += 2;
         else
10.        s *= 2;
11.      i++;
       }
12.    printf("%d", s);
     }
```

Figure 2.2. Example program

Consider the example program in Figure 2.2. The static slice of this code with respect to the variable s at vertex 12 contains all the statements.

Prior to the description of a new dynamic slice algorithm we introduce some basic concepts and notations.

A feasible path that has actually been executed will be referred to as an *execution history* and denoted by EH. Let the input be $a = 0$, $n = 2$ in the case of our example. The corresponding execution history is $\langle 1, 2, 3, 4, 5, 7, 8, 10, 11, 7,$
$8, 10, 11, 7, 12\rangle$. We can see that the execution history contains instructions which come in the same order as they have been executed, so $EH(j)$ gives the serial number of the instruction executed at the j^{th} step, referred to as *execution position* j.

To distinguish between multiple occurrences of the same instruction in the execution history we make use of the notion of *action*. It is a pair (i, j) which is written as i^j,

where i is the serial number of the instruction at the execution position j. For example 12^{15} is the action for the output statement of our example for the same input as above.

The *dynamic slicing criterion* is a triplet $(\mathbf{x}, i^j, V)$ where $\mathbf{x}$ denotes the input, i^j is an action in the execution history, and V is a set of the variables. For a slicing criterion a dynamic slice can be defined as the set of statements which may affect the values of the variables in V.

We apply a program representation which only considers the definition of a variable, and use of variables, and direct control dependencies. We refer to this program representation as a *D/U program representation*. An instruction of the original program has a D/U expression of the form:

$$i.\ d:\ U,$$

where i is the serial number of the instruction, and d is the variable that gets a new value from the instruction in the case of assignment statements. For an output statement or a predicate d denotes a newly generated "output variable" or "predicate-variable"-name of this output or predicate, respectively (see the example below). Let $U = \{u_1, u_2, ..., u_n\}$ such that any $u_k \in U$ is either a variable that is used at i or a predicate-variable from which instruction i is (directly) control dependent. Note that there is at most one predicate-variable in each U. (If the *entry* statement is defined, there is exactly one predicate-variable in each U.)

Our example has a D/U representation shown in Figure 2.3.

$i.$	d	:	U
1.	n	:	$\emptyset$
2.	a	:	$\emptyset$
3.	i	:	$\emptyset$
4.	s	:	$\emptyset$
5.	$p5$	:	$\{a\}$
6.	s	:	$\{p5\}$
7.	$p7$	:	$\{i, n\}$
8.	$p8$	:	$\{p7, a\}$
9.	s	:	$\{s, p8\}$
10.	s	:	$\{s, p8\}$
11.	i	:	$\{i, p7\}$
12.	$o12$	:	$\{s\}$

Figure 2.3. D/U representation of the program

Here $p5$, $p7$ and $p8$ are used to denote predicate-variables and $o12$ denotes the output-variable, whose value depends on the variable(s) used in the output statement.

Now we are ready to derive the dynamic slice with respect to an input and the related execution history based on the D/U representation of the program as follows. First, we process each instruction in the execution history starting from the first executed statement. Then after processing an instruction $i.\ d : U$, we derive a set $DynSlice(d)$ that contains all those statements which affect d when instruction i has been executed. By applying the D/U program representation the effect of data and control dependencies may be treated in the same way. After an instruction has been executed and the related *DynSlice* set has been derived, we determine the *last definition* (serial number of the instruction) for the newly assigned variable d denoted by

$LS(d)$. Put simply, the last definition of variable d is the serial number of the instruction where d is last defined (considering the instruction $i.\ d : U$, $LS(d) = i$). Clearly, after processing the instruction $i.\ d : U$ at the execution position j each $LS(d)$ has the value i for each subsequent executions until d is redefined next time. We also use $LS(p)$ for predicates, which denotes the last definition (evaluation) of predicate p. For example, if $EH(10) = 7$ (the current action is 7^{10}) then $LS(d) = 7$.

Now the dynamic slices can be determined as follows. Assume that we are running a program having an input t. After an instruction $i.\ d :\ U$ is executed at position p, $DynSlice(d)$ contains just those statements involved in the dynamic slice for the slicing criterion $C = (t, i^p, U)$. *DynSlice* sets are determined by using the relation below:

$$DynSlice(d) = \bigcup_{u_k \in U} \Big(DynSlice(u_k) \cup \{LS(u_k)\} \Big)$$

After $DynSlice(d)$ has been evaluated we determine $LS(d)$ for assignment and predicate instructions, i.e.

$$LS(d) = i$$

Note that this computation order is strict since when we determine $DynSlice(d)$, we have to consider whether $LS(d)$ occurred at a former execution position instead of p (like the program line `x = x + y` in a loop).

program DynamicSlice
begin
  Initialize *LS* and *DynSlice* sets
  ConstructD/U
  ConstructEH
  for $j = 1$ **to** *number of elements in EH*
    the current D/U element is $i^j.\ d :\ U$
    $DynSlice(d) = \bigcup_{u_k \in U} \big(DynSlice(u_k) \cup \{LS(u_k)\} \big)$
    $LS(d) = i$
  endfor
  Output *LS* and *DynSlice* sets for the last definition of all variables
end

Figure 2.4. Dynamic slice algorithm

A formal version of the forward dynamic slice algorithm is presented in Figure 2.4. Note that the construction of the execution history is achieved by instrumenting the input program and executing this instrumented code. The instrumentation procedure is discussed in [18].

We will illustrate how the above method works by applying it to our example program in Figure 2.2 with the execution history ⟨ 1, 2, 3, 4, 5, 7, 8, 10, 11, 7, 8, 10, 11, 7, 12 ⟩.

During the execution the following values are returned:

Action	d	U	$DynSlice(d)$	$LS(d)$
1^1	n	$\emptyset$	$\emptyset$	1
2^2	a	$\emptyset$	$\emptyset$	2
3^3	i	$\emptyset$	$\emptyset$	3
4^4	s	$\emptyset$	$\emptyset$	4
5^5	$p5$	$\{a\}$	$\{2\}$	5
7^6	$p7$	$\{i, n\}$	$\{1,3\}$	7
8^7	$p8$	$\{p7, a\}$	$\{1,2,3,7\}$	8
10^8	s	$\{s, p8\}$	$\{1,2,3,4,7,8\}$	10
11^9	i	$\{i, p7\}$	$\{1,3,7\}$	11
7^{10}	$p7$	$\{i, n\}$	$\{1,3,7,11\}$	7
8^{11}	$p8$	$\{p7, a\}$	$\{1,2,3,7,11\}$	8
10^{12}	s	$\{s, p8\}$	$\{1,2,3,4,7,8,10,11\}$	10
11^{13}	i	$\{i, p7\}$	$\{1,3,7,11\}$	11
7^{14}	$p7$	$\{i, n\}$	$\{1,3,7,11\}$	7
12^{15}	$o12$	$\{s\}$	$\{1,2,3,4,7,8,10,11\}$	12

The final slice is the union of $DynSlice(o12)$ and $\{LS(o12)\}$, see Figure 2.5.

```
#include <stdio.h>
int n, a, i, s;
void main()
{
 1.  scanf("%d", &n);
 2.  scanf("%d", &a);
 3.  i = 1;
 4.  s = 1;
 5.   if (a > 0)
 6.     s = 0;
 7.  while (i <= n) {
 8.    if (a > 0)
 9.       s += 2;
        else
10.      s *= 2;
11.    i++;
      }
12.  printf("%d", s);
}
```

Figure 2.5. The framed statements give the dynamic slice

Analysis of the algorithm

Let us analyze the duration and the memory requirement of the algorithm. It is very hard to figure out the exact requirements. We try to make an average-case analysis with referring to the worst-case, too.

First let us consider the duration. The initializations are approximately linear to the different memory locations. The `DU` construction is linear to the length of the

executable source code. One can ask why do not we say that it is linear to the statements? The reason is that the duration of one step is dependent to the length of the statement. For example it takes less time to build up the `DU` for `a=b+c` than for `a=b*c-f(b,b+c)-c`. The construction of the `EH` is linear to the execution of the original program. Unfortunately the constant multiplier hidden by "theta-notation" (it is used in analysis of the algorithms) is hardly predictable: it is dependent to the number of pointers etc., which can vary from zero up to the whole program. The duration of the first and the third pseudo-statement within the main `for` cycle is constant. The union statement's duration is critical within the algorithm, but unfortunately it is very hardly predictable. In the worst case the `U` set can hold all the variables (i.e. different memory locations + pseudo-variables (e.g. labels etc.)), and all the dynamic slices holds all the statements within the program. It this case the main cycle's duration is proportional to `<execution history> * <memory locations> * <number of statements>`. But in the most normal programs the size of the `U` set is not so big, in most cases it holds about 4-5 elements. There are hardly any such statements where the `U` set contains more than 10 elements. A dynamic slice in most cases contains not too much statement, but it seems in many cases it is linear to the size of the program. The duration of the output depends to the numbers of slice criteria etc., but it is less than the computation. According to these the average execution time of the algorithm is `O(|EH|*|statements|+|memory|)`.

Now let us analyze the memory requirements. The most relevant memory requirement takes the storage of the temporal slice results, i.e. the `U` sets; the others (e.g. memory requirements of the initialization part etc.) can be ignored. In the worst case every variable (i.e. every memory location) contains all the statements, so in this case the memory requirement is `O(<number of different memory locations> * <number of statements>)`. In fact it is very unlikely to use such a big memory. At bigger programs in most cases the memory requirement of the dynamic counting algorithm is linear to the memory requirement of the original program, with a bigger constant.

2.2.3 Dynamic Slicing Method for Large C Programs

In order to handle the pointers, the variables are identified by their addresses and not by their names. This approach has several good advantages. One is that it solves the problem of the variables with the same name but different program scope.

The address of a variable can only be determined dynamically after its declaration, but the DU is derived from the static source code. Hence there are two DU structures: a static DU which contains variable names, and a dynamically resolved DU (dynamic DU) which contains addresses. Note that the dynamic DU may change during the program execution due to a change in variable address, pointer value, etc. The necessary parts of the dynamic DU are computed at each step using the static DU and the (extended) execution history.

In a C program there may be several variables present with the same name but in different scopes. The address of a variable with a specific name may depend on the scope of the expression where the variable is used. So the algorithm must keep track of the scopes and maintain a stack structure for each function in order to store the addresses of the variables. Each time a new scope is begun, a new address table is created at the top of the stack, and when a new variable declaration occurs, the name

and address are recorded in this new table. For the address of a variable the address tables are searched from the top to the bottom of the stack of the actual function. When a scope is leaved, the top element of the stack is discarded. The first element at the bottom of each such stack is the same: the address table of global variables (these can be accessed by every function).

2.2.4 Union Slices

In paper [17] we proposed a compromise solution for program maintenance tasks where the static slices are unacceptable because of their lack of precision, and dynamic methods are unfeasible because of the lack of resources needed to conduct lots of test cases. We introduced the notion of *union slices* for the computation of the union of dynamic slices for many test cases. It was based on previous results where the authors introduced an efficient method for the computation of dynamic slices in real life situations [18]. They recommended the use of a combination of static and union slices to determine the responsible program parts with less effort. This means that the size of program parts that need to be investigated can be reduced by concentrating on the most important parts first. The basic idea for this comes from the fact that while the static slices are safe but large, union slices are smaller but, alas, unsafe (i.e. they do not contain all possible dependencies).

The concept of union slices is fairly obvious as the union of dynamic slices for a (finite) set of test cases. However, if we computed the union of dynamic slices for *all possible* executions, we would obtain a theoretical slice of the program that contains all realizable dependencies. Therefore, we refer to this slice as the *realizable slice*.

The precise slice is minimal and therefore can be smaller than the realizable slice, but then the latter can be approximated by practical means. If we consider the static slice as an upper bound of the realizable slice, the union slice can be seen, on the other hand, as the *lower bound* for it (see Figure 2.6).

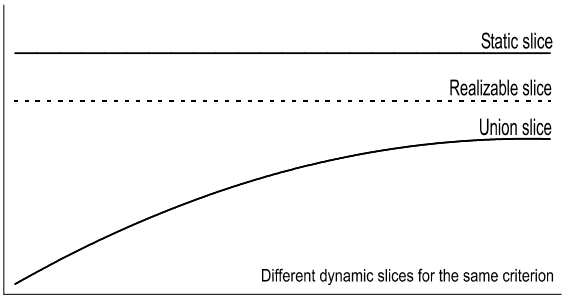

Figure 2.6. Approximation of the realizable slice

For the calculation of union slices, we modified the algorithm presented in [18] as follows:

We defined the union slices as the union of the dynamic slices for a variety of inputs. It is the following for a particular program P with different executions using the inputs $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$:

$$UnionSlice(X, i, V) = \bigcup_{\mathbf{x}_k \in X} DynSlice(\mathbf{x}_k, i, V)$$

Here, that specific *DynSlice* set is used which holds the dynamic slice for the last occurrence of instruction i in the execution trace.

program DynamicSlicer($P, \mathbf{x}$)
inputs: EH for program P with input $\mathbf{x}$
D/U representation for P
outputs: $DynSlice(\mathbf{x}, i^j, V)$ sets for all i^j actions
with the corresponding V sets as the used variables at i
begin
Initialize LS and $DynDep$ sets for all variables
for $j = 1$ **to** *number of elements in EH*
the current D/U element is i^j. $d: U$
$DynDep(d) = \bigcup_{u_k \in U} \big(DynDep(u_k) \cup \{LS(u_k)\}\big)$
$LS(d) = i$
Output $DynSlice(\mathbf{x}, i^j, U) = DynDep(d)$
endfor
end

Figure 2.7. Unions slice algorithm

2.3 Unstructured Statements Handling

An issue which must be dealt with is how we should handle the jump statements in the dynamic slicing algorithm. In this section we consider the C-specific jump statements, but the method could be used in other programming languages as well.

In the next part we describe the handling of the `goto` statement, along with the `break`, `continue`, and `switch` statements.

2.3.1 The `goto` Statement

Where a `goto` statement occurs, the D/U structure is built up as follows. First, so-called *"label variables"* are introduced. Let the defined variable (d) be the previously introduced label variable called the real name of the label. It could also be an ordinal number, but for the sake of simplicity we use the previous name here. The use set (U) contains no "extra" variables, just the appropriate predicate variable, and we will find that it can contain label variables too.

The previously defined label variable is inserted into the use set (U) of those statements which occur after the corresponding label within the function. It is important to do this to the end of the function, not just in the appropriate block.

If there are more labels, they are all handled in the same way. If the `goto` statement appears after the definition of the label, then of course it contains the just defined label variable. But this is not a problem because in the execution history it appears as a formerly defined variable. It can be defined by itself or by another `goto` statement. If no `goto` statement that jumps to a specific label is executed during the program, the last definition of that label remains undefined so it will not affect the result of the dynamic slice. The result contains all of the defined labels.

When the `goto` is executed during the program and the dynamic slice contains at least one of the statements after the definition of the label, then the result will at least contain the previous corresponding `goto` (and of course its predicate dependencies transitively). So it often unnecessarily increases the size of the dynamic slice, and using lots of `goto` statements will make it hard to analyze the program.

$i.$		def	:	USE
	`int i,j,k,l;`			
1.	`k=0;`	k	:	$\emptyset$
2.	`l=0;`	l	:	$\emptyset$
3.	`i=0;`	i	:	$\emptyset$
	`l1:`			
4.	`j=0;`	j	:	$\{l1\}$
	`l2:`			
5.	`k=k+i+j;`	k	:	$\{k, i, j, l1, l2\}$
6.	`l++;`	l	:	$\{l, l1, l2\}$
7.	`j++;`	j	:	$\{j, l1, l2\}$
8.	`if (j<2)`	$p8$	:	$\{j, l1, l2\}$
9.	`goto l2;`	$l2$	:	$\{p8, l1, l2\}$
10.	`i++;`	i	:	$\{i, l1, l2\}$
11.	`if (i<2)`	$p11$	:	$\{i, l1, l2\}$
12.	`goto l1;`	$l1$	:	$\{p11, l1, l2\}$
13.	`printf("%d",k);`	$o13$	:	$\{k, l1, l2\}$

Figure 2.8. Example program: handling the `goto` statement

We show an example in Figure 2.8, and its results in Figure 2.9. As one might expect, the use of `goto` statements resulted in a lot of dependencies.

2.3.2 The `break` Statement

The `break` statement is practically equivalent to `goto` statement, which jumps out from the block of the appropriate `while`, `do...while`, `switch` or `for` statement to the first statement after this block. This statement can be handled as follows. The defined variable at every occurrence of the `break` statement should be an individual label variable. One form might be `break<Nr>`, where `<Nr>` is the ordinal number of the `break` statement within the program. All of the statements after the corresponding block are dependent on the previously defined label variable, just like in the case of `goto` statement. Note that if a label is placed just after the corresponding block and the `break` is replaced with a `goto` which jumps to that label, then the effect is the same.

An example of the `break` statement and results are shown in Figure 2.10 and Figure 2.11, respectively.

2.3.3 The `continue` Statement

Like the `break` statement, we should define a separate label variable. This might be denoted by `continue<Nr>`, where `<Nr>` is the ordinal number of the `continue` statement within the program. It is defined in statements where `continue` occurs. The dependent statements are statements from the beginning of the block of the appropriate `for`, `while` or `do...while` statement to the end of the function. So the `continue` statement is always dependent upon itself.

Action (i^j)	*DynSlice*()	Action (i^j)	*DynSlice*()
1^1	$\emptyset$	16^{12}	$\{3,4,7,8,9,10,11\}$
2^2	$\emptyset$	17^4	$\{3,4,7,8,9,10,11,12\}$
3^3	$\emptyset$	18^5	$\{1,3,4,5,7,8,9,10,11,12\}$
4^4	$\emptyset$	19^6	$\{2,3,4,6,7,8,9,10,11,12\}$
5^5	$\{1,3,4\}$	20^7	$\{3,4,7,8,9,10,11,12\}$
6^6	$\{2\}$	21^8	$\{3,4,7,8,9,10,11,12\}$
7^7	$\{4\}$	22^9	$\{3,4,7,8,9,10,11,12\}$
8^8	$\{4,7\}$	23^5	$\{1,3,4,5,7,8,9,10,11,12\}$
9^9	$\{4,7,8\}$	24^6	$\{2,3,4,6,7,8,9,10,11,12\}$
10^5	$\{1,3,4,5,7,8,9\}$	25^7	$\{3,4,7,8,9,10,11,12\}$
11^6	$\{2,4,6,7,8,9\}$	26^8	$\{3,4,7,8,9,10,11,12\}$
12^7	$\{4,7,8,9\}$	27^{10}	$\{3,4,7,8,9,10,11,12\}$
13^8	$\{4,7,8,9\}$	28^{11}	$\{3,4,7,8,9,10,11,12\}$
14^{10}	$\{3,4,7,8,9\}$	29^{13}	$\{1,3,4,5,7,8,9,10,11,12\}$
15^{11}	$\{3,4,7,8,9,10\}$		

Figure 2.9. Results of `goto` statement handling example program

i.		*def*	:	*USE*
	`int a,b,i;`			
1.	`a=1;`	a	:	$\emptyset$
2.	`b=1;`	b	:	$\emptyset$
3.	`i=2;`	b	:	$\emptyset$
4.	`while (i>0) {`	$p4$	:	$\{i\}$
5.	`b--;`	b	:	$\{p4,b\}$
6.	`i--;`	i	:	$\{p4,i\}$
7.	`if (b==0)`	$p7$	:	$\{b\}$
8.	`break;`	$break8$	:	$\{p7\}$
9.	`a++;`	a	:	$\{p4,a\}$
	`}`			
10.	`printf("%d",a);`	$o10$	:	$\{a,break8\}$

Figure 2.10. Example program: handling the `break` statement

An example of the `continue` statement and results are shown in Figure 2.12 and Figure 2.13, respectively.

2.3.4 The `switch` Statement

After the handling of `break` statement, the handling of the `switch` statement is quite straightforward.

At the place where the `switch` statement occurs a predicate variable is defined, just like in the case of `while` or `if`. All of the statements within the `switch` block are dependent on this predicate variable. If at least one statement within the switch block is included in the slice result, all of the `case` labels and the `default` label are included. Here the `break` statements are handled in the same way as described before.

Action (i^j)	$DynSlice()$
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\{3\}$
5^5	$\{2,3,4\}$
6^6	$\{3,4\}$
7^7	$\{2,3,4,5\}$
8^8	$\{2,3,4,5,7\}$
9^{10}	$\{1,2,3,4,5,7,8\}$

Figure 2.11. Results of `break` statement handling example program

$i.$		def	:	USE
	`int a,b,i;`			
1.	`a=1;`	a	:	$\emptyset$
2.	`b=1;`	b	:	$\emptyset$
3.	`i=2;`	b	:	$\emptyset$
4.	`while (i>0) {`	$p4$	:	$\{i, continue8\}$
5.	`b--;`	b	:	$\{p4, b, continue8\}$
6.	`i--;`	i	:	$\{p4, i, continue8\}$
7.	`if (b==0)`	$p7$	:	$\{b, continue8\}$
8.	`continue;`	$continue8$	:	$\{p7, continue8\}$
9.	`a++;`	a	:	$\{p4, a, continue8\}$
	`}`			
10.	`printf("%d",a);`	$o10$	:	$\{a, continue8\}$

Figure 2.12. Example program: handling the `continue` statement

An example of the `switch` statement and its results are shown in Figure 2.14 and Figure 2.15, respectively.

2.3.5 Improvements on the Methodology

The unstructured statement handling methodology has been improved later, see the summary work by Beszédes [16].[1]

In the case of `goto` correction of the algorithm was necessary. If both forward and backward jumps exist in the source code, then the methodology of building the static dependency as described in section 2.3.1 turned to be not precise enough. The post-dominance algorithm determines precise dependencies, as described in a paper by Harman and Danicic [63].

In one specific execution only one of the predicate variables will be responsible for the actually realized control dependence, which we call the *active predicate*. When propagating the dependencies through the current instruction's use set, we must select just one predicate variable to continue with. If there are more predicate variables in the

[1]Some sentences of the next two paragraphs are copied from the article [16].

Action (i^j)	$DynSlice()$
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\{3\}$
5^5	$\{2,3,4\}$
6^6	$\{3,4\}$
7^7	$\{2,3,4,5\}$
8^8	$\{2,3,4,5,7\}$
9^4	$\{2,3,4,5,6,7,8\}$
10^5	$\{2,3,4,5,6,7,8\}$
11^6	$\{2,3,4,5,6,7,8\}$
12^7	$\{2,3,4,5,6,7,8\}$
13^9	$\{1,2,3,4,5,6,7,8\}$
14^4	$\{2,3,4,5,6,7,8\}$
15^{10}	$\{1,2,3,4,5,6,7,8,9\}$

Figure 2.13. Results of `continue` statement handling example program

$i.$		def	:	USE
	`int a,b;`			
1.	`b=0;`	b	:	$\emptyset$
2.	`a=2;`	a	:	$\emptyset$
3.	`switch (a) {`	$p3$	:	$\{a\}$
	`case 1:`			
4.	`b=5;`	b	:	$\{p3\}$
5.	`break;`	$break5$	:	$\{p3\}$
	`case 2:`			
6.	`b=3;`	b	:	$\{p3\}$
	`case 3:`			
7.	`b++;`	b	:	$\{p3,b\}$
8.	`break;`	$break7$	:	$\{p3\}$
	`default:`			
9.	`b=6;`	b	:	$\{p3\}$
	`}`			
10.	`printf("%d",b);`	$o10$	:	$\{b,break5,break7\}$

Figure 2.14. Example program: handling the `switch` statement

use set, our approach is to choose the one that has been defined most recently. In other words, for $i^j.\ d:\ U$, we will choose predicate p for which $LD(p) = max\{LD(r)|r \in U$ and r is a predicate variable$\}$, where $LD(v)$ is the last definition of variable v, i.e. the execution step at which v was defined just before the j^{th} step where i was executed.

Action (i^j)	*DynSlice*()
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\{2\}$
4^6	$\{2,3\}$
5^7	$\{2,3,6\}$
6^8	$\{2,3\}$
7^{10}	$\{2,3,6,7,8\}$

Figure 2.15. Results of `switch` statement handling example program

2.4 Experimental Results

Several experimental results confirmed that our dynamic slices were more precise than the static one. Among the test sources there are 3 medium sized: the `bzip` (a compression utility), the `bc` (a scientific calculator) and the `less` (this is a powerful text viewer program). The sizes of these programs is shown in the following table.

prog	lines	executable	files	bytes	functions
bzip	4495	1595	1	130 458	73
bc	11555	3220	20	312 722	138
less	21489	5400	43	639 036	363

The first column is the name of the program, the second one means the total lines of the source, the third is the size of the executable code (i.e. without comments etc.), the fourth is the number of source files, the fifth is the total length of the source code in bytes, and the last one means the number of the functions within the program.

With help of our program we made several executions on several slice criteria for all the 3 sources. The number of the different slice criteria and the number of the executions are shown in the next table.

program	criteria	executions	coverage
bzip	154	18	68%
bc	57	49	63%
less	50	14	45%

The last column shows the coverage of the program. A statement is defined to be covered if at least once is executed during all the tests. The coverage means the percentage of the covered statements related to the whole program.

With a static slice generator tool (CodeSurfer [31]) we made static slices, too. The results are shown in Figure 2.16.

The first column shows the size of the executable code, the second the coverage, the third the average static slice (result of the CodeSurfer) and the last one is average of the so-called union slices generated by our dynamic slice generator tool. The union slice means the union of the all the generated slices (several executions + more results within one execution) to a certain statement.

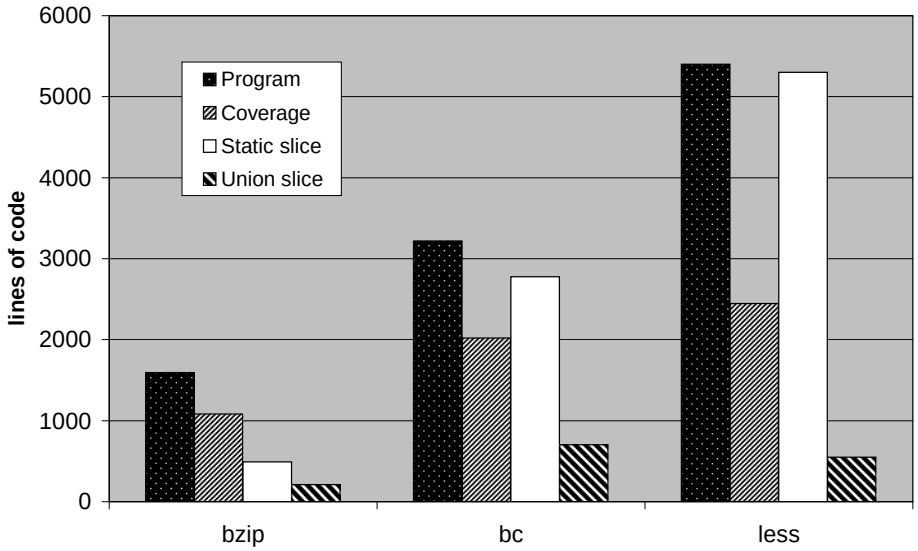

Figure 2.16. The average slice sizes

In Figure 2.17 the average growth tendency for the three programs can be seen using the same input as above. In order to obtain this diagram we computed the simple average of growth curves of all of the slicing criteria separately for the three programs (the same as for the diagrams in the previous figure). These curves are displayed in a normalized form relative to the maximum attained slice size. The curves were also stretched horizontally, the full width depicting the total number of the executions. Interestingly, the overall characteristics for the three different programs are quite similar.

2.5 Summary

In this chapter we presented a dynamic program slicing algorithm. It forward computes the dynamic slices. The algorithm is effective, as its memory requirement in practice is proportional with the different memory locations of the original program.

Then we presented an extension of the algorithm, suitable for the analysis of real C programs. In a generalization of the algorithm the notion of relevant slice was introduced, which is the union slice of all the theoretically possible executions.

One of the problems to be solved to apply the original algorithm to C was the unstructured C statements handling, which are the following: `goto`, `break`, `continue`, `switch-case-default`. The main idea was the introduction of virtual label variables (predicate variable in case on `switch`), with appropriate places of definition and dependency.

In case of significant code coverage the size of the resulting slice was a fraction of the result calculated by a static program slicing tool.

2.6 Contributions

The author contributed to the new results presented in this chapter as follows:

- Elaboration of the unstructured statements handling in the presented dynamic slicing algorithm (Section 2.3).

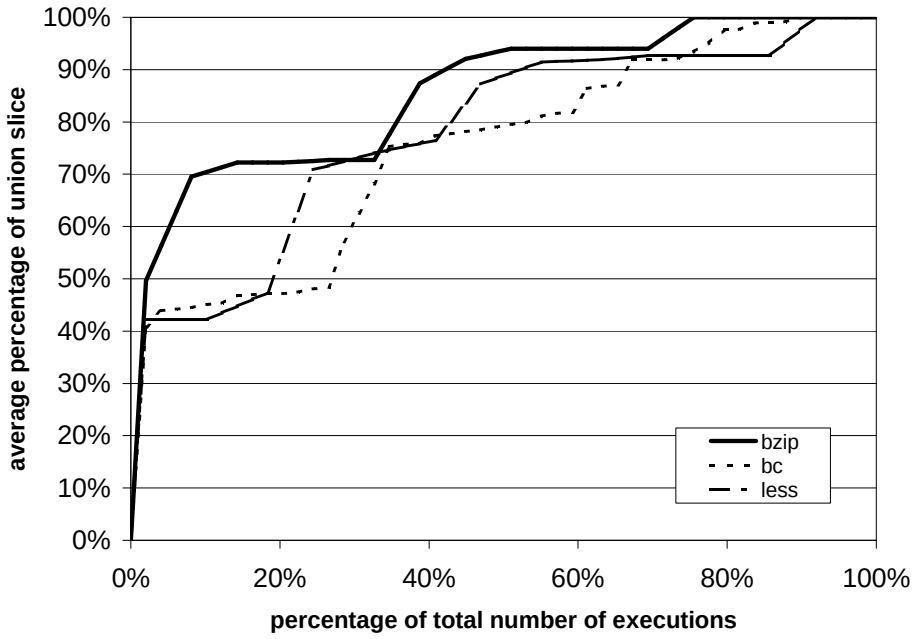

Figure 2.17. Average growth of the union slices

- Participating in the implementation.
- Execution of the tests (Section 2.4).

Part II

Version Control History Metrics

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

— Rick Osborne

3

Overview of Version Control History and Maintainability

Software erosion is a well-known phenomenon, meaning that software quality is continuously decreasing due to the ever-ongoing modifications in the source code. In this part of the thesis we present the results of finding the connection between the developers' interactions and the quality change.

We were motivated to perform this research for several reasons. Determining typical patterns which have significant effect on maintainability could help us better allocate software developer efforts. For example, a more strict code review is recommended for those commits which have statistically higher impact on maintainability.

In this section we present common information related to Section 4 and 5. First, in Section 3.1 we give a general overview of this research area. In Section 3.2 we present the quality model we use for measuring the maintainability of a systems. In Section 3.3 we present the analyzed systems we used. In Section 3.4 we describe how we performed random checks as a validation cross-check. Finally, in Section 3.5 we present some works related to this part of the thesis.

The Quality Model presented in Section 3.2.1 (including the class level version within Section 3.2.2) is not the work of the author of this thesis. The author took part in the mathematical formalism of the algorithm of drill down methodology (Section 3.2.2), but the algorithm itself is not the work of the author of this thesis. Collecting the input data, as described in Section 3.3 is also not done by the author. The rest of this chapter represents the work of the author.

3.1 General Overview

Figure 3.1 provides a brief overview about the current state of the art, the main topic of this part of the thesis, and highlights some possible future research directions.

Historically first, several source code metrics were defined, and their bug predictive values were proved. Then these metrics were aggregated to quality models which

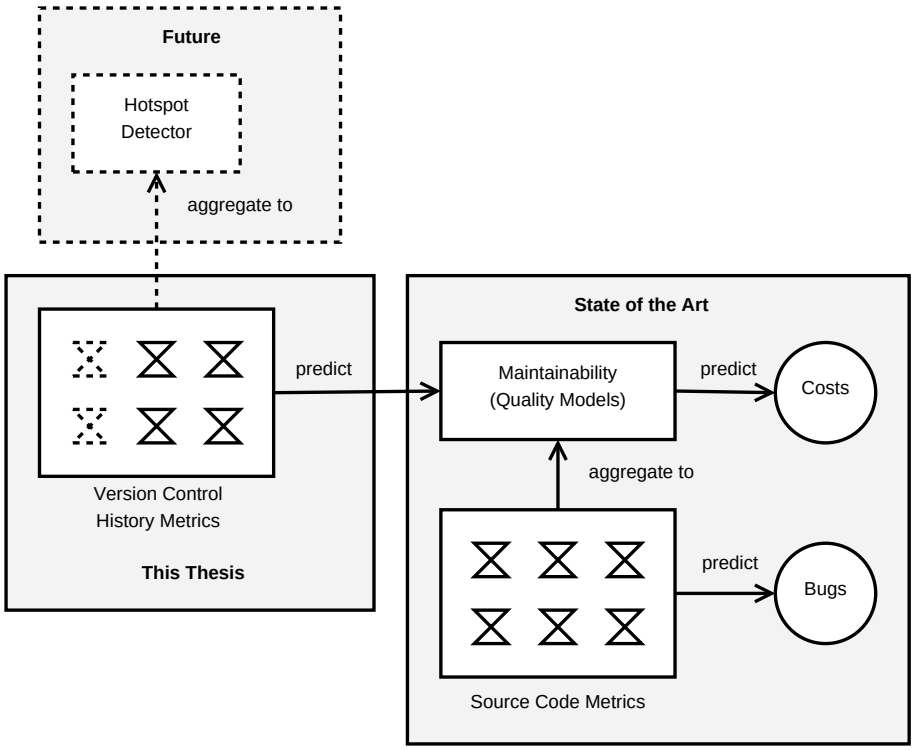

Figure 3.1. Overview of version control history based maintainability analysis

measure maintainability of the source code. As maintainability has direct impact on development costs, a good quality model predicts efforts well.

We in this thesis focus on version control history. We present some metrics and show their correlation with software maintainability, calculated by a certain quality model.

The possible future research directions are drawn with dashed lines. Other version control history metrics should be defined and analyzed as well. A mid-term goal is to summarize these results and create a version control history based hotspot detector.

3.2 Quality Model

3.2.1 The Columbus Quality Model

For the definition of software quality we refer to the ISO/IEC 9126 standard [76], which defines six high-level characteristics that determine the product quality of software: *functionality, reliability, usability, efficiency, portability*, and *maintainability*. Due to its direct impact on development costs [12], and being in close relation with the source code, maintainability is one of the most important quality characteristics.

To calculate the absolute maintainability values for every revision of the systems we used the Columbus Quality Model, a probabilistic software quality model that is based on the quality characteristics defined by the ISO/IEC 9126 [76] standard. The computation of the high-level quality characteristics is based on a directed acyclic graph (see Figure 3.2) whose nodes correspond to quality properties that can either be internal (low-level) or external (high-level). Internal quality properties characterize the software product from an internal (developer) view and are usually calculated by using source code metrics. External quality properties characterize the software

product from an external (end user) view and are usually aggregated somehow by using internal and other external quality properties. The nodes representing internal quality properties are called *sensor nodes* as they measure internal quality directly (white nodes in Figure 3.2). The other nodes are called *aggregate nodes* as they acquire their measures through aggregation. In addition to the aggregate nodes defined by the standard (dark gray nodes) we also introduce new ones (light gray nodes).

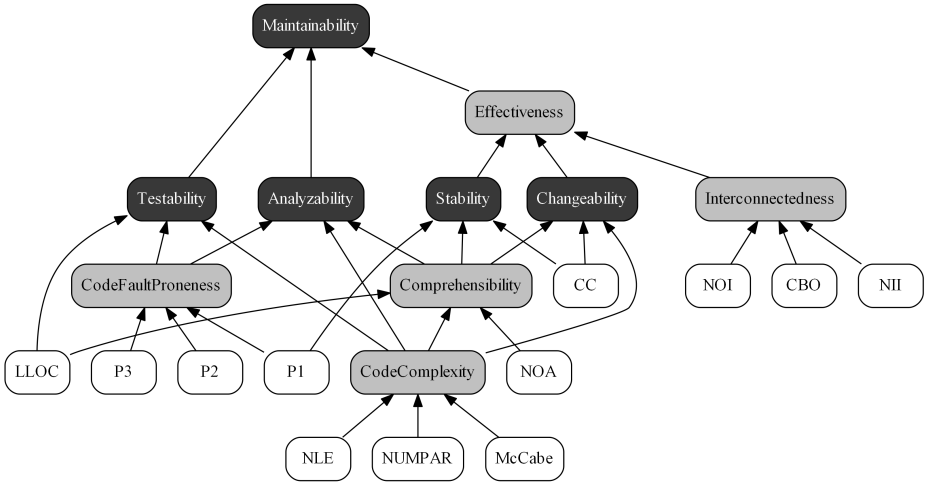

Figure 3.2. Attribute Dependency Graph of Columbus Quality Model

For the version of the model we used for the analysis for Section 4 and 5.1 the following source code metrics applies:

- *LLOC (Logical Lines Of Code)* – the LLOC metric is the number of non-comment and non-empty lines of code.
- *NOA (Number Of Ancestors)* – NOA is the number of classes that a given class directly or indirectly inherits from.
- *NLE (Nesting Level Else-if)* – NLE for a method is the maximum of the control structure depth. Only `if`, `switch`, `for`, `foreach`, `while`, and `do...while` instructions are taken into account and in the `if-else-if` constructs only the first `if` instruction is considered.
- *CBO (Coupling Between Object classes)* – a class is coupled to another if the class uses any method or attribute of the other class or directly inherits from it. CBO is the number of coupled classes.
- *CC (Clone Coverage)* – clone coverage is a real value between 0 and 1 that expresses what amount of the item is covered by code duplication.
- *NUMPAR (NUMber of PARameters)* – the number of parameters of the methods.
- *McCC (McCabe's Cyclomatic Complexity)* – the value of the metric is calculated as the number of the following instructions plus 1: `if`, `for`, `foreach`, `while`, `do-while`, `case` label (which belongs to a `switch` instruction), `catch`, conditional statement `(?:)`.

- *NII (Number of Incoming Invocations)* – the number of other methods and attribute initializations which directly call the method. If a method is invoked several times from the same method or attribute initialization, it is counted once.
- *NOI (Number of Outgoing Invocations)* – the number of directly called methods. If a method is invoked several times, it is counted once.
- *WarningP1/P2/P3 (Serious/medium/minor coding rule violations)* – the number of serious/medium/minor PMD (`http://pmd.sourceforge.net/`) rule violations in the class.

The edges of the graph represent dependencies between an internal and an external or two external properties. The aim is to evaluate all the external quality properties by performing an aggregation along the edges of the graph, called Attribute Dependency Graph (ADG). We calculate a so called *goodness value* (from the [0,1] interval) for each node in the ADG that expresses how good or bad (1 is the best) the system is regarding that quality attribute. The probabilistic statistical aggregation algorithm uses a so-called benchmark as the basis of the qualification, which is a source code metric repository database with 100 open source and industrial software systems.

For further details about Columbus Quality Model, see work by Bakota et al. [11].

Bakota et al. [12] showed that there was a converse exponential relationship between the maintainability of a software system and the overall cost of development, supported by an empirical validation. The QualityGate SourceAudit tool by Bakota et al. [10] is based on this quality model.

Demonstrating Maintainability Changes

In most of the statistic tests it was enough to determine the sign of the difference of subsequent commits. For demonstration purposes we present a short example for all the 3 types of maintainability changes; all of the examples were taken from the source code of open-source project Tomcat.

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);
}
```

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

For demonstrating maintainability decrease we consider 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.

Finally, for demonstrating the no traceable maintainability change category, we check revision 607483, at which source file `util/modeler/ManagedBean.java` was committed. The following line of code has been changed:

```
    object = this;
```

to the following:

```
    object = bean;
```

This change is very small and impacts no source code metrics at all, so it in itself does not affect the maintainability of the system (at least ColumbusQM does not recognize it).

3.2.2 A Drill-down Methodology

We extended the Columbus Quality Model with a drill-down methodology, as described by Hegedűs et al. [69]. This calculates the so-called Relative Maintainability Indices (the RMIs) for lower-level source code elements (e.g. classes or methods).

The base idea in a nutshell is the following: the maintainability analysis is performed on the whole system, and on the system without the analyzed source code element (which can be a package, a class or a function; in this thesis the classes are considered). The RMI is the difference between the original maintainability value and the maintainability value without that source code element. If the actual source code element is a hard to maintain code, compared to the rest of the system, then the maintainability value without that is higher than the original one; therefore the RMI of that element is negative.

First in this section we present a detailed, more formal description of the idea, and then we describe how we applied it on a model using class level sensor nodes only.

Calculating Relative Maintainability Index

To drill down to lower levels in the source code and to get a maintainability measure for the building blocks of the code base (like classes or methods), we define the Relative Maintainability Index (RMI) for the source code elements, which measures the extent to which they affect the system-level goodness values. The basic idea is to calculate the system-level goodness values, leaving out the source code elements one by one. After leaving out a particular source code element, the system-level goodness values will change slightly for each node in the ADG. The difference between the original goodness value computed for the system, and the goodness value computed without the particular source code element is called the *Relative Maintainability Index* of the source code element itself. The Relative Maintainability Index is a small number that is either positive when it improves the overall rating or negative when it decreases the system-level maintainability. The absolute value of the index measures the extent of the influence on the overall system-level maintainability. Also, a relative index can be computed for each node of the ADG, meaning that source code elements can affect various quality aspects in different ways and to a different extent.

Calculating the system-level maintainability is computationally very expensive. To get the relative indices, it is sufficient to compute just the goodness values for each node in the ADG; we do not need to construct the goodness functions. Luckily, computing the goodness values without knowing the goodness functions is feasible. It can be shown that calculating goodness functions and taking their averages is equivalent to just using the goodness values throughout the aggregation.

In the following, we will assume that $\omega(t)$ is equal to t for each sensor node (see the system-level quality computation in Section 3.2), which means that e.g. metric value twice as big means code twice as bad. While this linear function might not be appropriate for every metric, it is a very reasonable weight function considering the metrics used by the quality model. However, our approach is independent of the particular weight function used, and the formalization can be easily extended to different weight functions.

Next, we provide a step-by-step description of the approach used for a particular source code element.

1. For each sensor node n, the goodness value of the system without the source code element e can be calculated via the following formula:

$$g_{rel}^{e,n} = \frac{Kg_{abs}^{n} + m}{K-1} - \frac{1}{N}\sum_{j=1}^{N}\frac{M_j}{K-1}$$

where g_{abs}^{n} is the original goodness value computed for the system, m is the metric value of the source code element corresponding to the sensor node, K is the number of source code elements in the subject system, N is the number of systems in the benchmark, and $M_j\,(j = 1, \ldots, N)$ are the averages of the metrics for the systems in the benchmark.

2. The goodness value obtained in this way is transformed to the $(0, 1)$ interval by using the characteristic function which is the distribution function of the goodness values of a particular ADG node across the benchmark systems (see e.g. Figure 3.2 and Figure 3.3) of the sensor node n. For the sake of simplicity, we assume that from now on $g_{rel}^{e,n}$ stands for the transformed goodness value and it refers to the goodness value as well.

3. Due to the linearity of the expected value of a random variable, it can be shown that aggregating goodness functions in the original algorithm simplifies to a linear combination, provided that just the expected value needs to be computed. Therefore, the goodness value of an aggregate node n can be computed in the following way:

$$g_{rel}^{e,n} = \sum_{i} g_{rel}^{i} E\left(Y_v^i\right)$$

where $g_{rel}^{i}\,(i = 1, \ldots)$ are the transformed goodness values of the nodes that are on the other sides of the incoming edges and $E\left(Y_v^i\right)$ is the expected value of the votes on the i^{th} incoming edge. Note here that since $\sum_i E\left(Y_v^i\right) = 1$, and $\forall i, g_{rel}^{i} \in (0, 1)$, the value of $g_{rel}^{e,n}$ always lies in the $(0, 1)$ interval, hence no transformation is needed at this point.

4. The relative maintainability index for the source code element e and for a particular ADG node n is defined as

$$g_{idx}^{e,n} = g_{abs}^{n} - g_{rel}^{e,n}$$

The Relative Maintainability Index measures the effect of the particular source code element on the system level maintainability, computed via the probabilistic model. It should be mentioned here that this measure determines an ordering among the source code elements of the system, i.e. they become comparable to each other. What is more, because the system-level maintainability is an absolute measure of maintainability, the relative index values become absolute measures of all the source code elements in the benchmark. In other words, computing all the relative indices for each software system in the benchmark will produce an absolute ordering among them.

A Class Level Quality Model

For RMI calculation of study related to this thesis we used a modified version of quality model, containing exclusively class level metrics. We removed the metrics related to functions (like number of parameters), and we introduced new metrics as well, which were results of intermediary evolution of the quality model (e.g. comment density).

Figure 3.3 illustrates the Attribute Dependency Graph of the class level version of quality model.

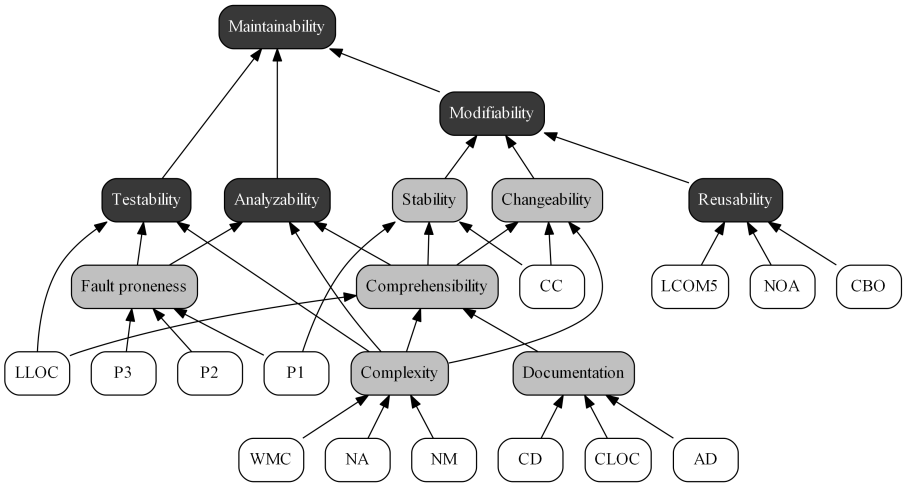

Figure 3.3. ADG of class level quality model

We recall the explanation to the Figure 3.2. The new sensor nodes compared to the original one are the following:

- *WMC (Weighted Methods per Class)* Complexity of the class expressed as the number of independent control flow paths in it. It is calculated as the sum of the McCabe's Cyclomatic Complexity (McCC) values of its local methods and initialization blocks.
- *NA (Number of Attributes)* Number of attributes in the class, including the inherited ones; however, the attributes of its nested, anonymous, and local classes are not included.
- *NM (Number of Methods)* Number of methods in the class, including the inherited ones; however, the methods of its nested, anonymous and local classes are not included. Methods that override abstract methods are not counted.
- *CD (Comment Density)* Ratio of the comment lines of the class (CLOC) to the sum of its comment (CLOC) and logical lines of code (LLOC).
- *CLOC (Comment Lines of Code)* Number of comment and documentation code lines of the class, including its local methods and attributes; however, its nested, anonymous, and local classes are not included.
- *AD (API Documentation)* Ratio of the number of documented public methods in the class (+ 1 if the class itself is documented) to the number of all public

methods in the class + 1 (the class itself); however, the nested, anonymous, and local classes are not included.

- *LCOM5 (Lack of Cohesion in Methods 5, number of functionalities of the class)* One of the basic principles of object-oriented programming is encapsulation, meaning that attributes belonging together and the operations that use them should be organized into one class, and one class shall implement only one functionality, i.e. its attributes and methods should be coherent. This metric measures the lack of cohesion and computes into how many coherent classes the class could be split. It is calculated by taking a non-directed graph, where the nodes are the implemented local methods of the class and there is an edge between the two nodes if and only if a common (local or inherited) attribute or abstract method is used or a method invokes another. The value of the metric is the number of connected components in the graph, not counting those which contain only constructors, destructors, getters, or setters.

3.3 Analyzed Systems

In this section we present how we selected the software system for data analysis, the result of the selection, and some basic information about these systems.

For Section 5.1 we used the same software systems for verification as in Section 4. On the other hand, for Section 5.2 we had to choose other systems for two reasons: first, because the different nature of the analysis (several concrete versions of each software, instead of the complete version control history of the main branch), and second, because the post release bug cross check.

In this section we present these software systems.

3.3.1 Analyzed Systems for Operations, Churn and Ownership Analysis

In order to gain as adequate results as possible, we considered those projects which had enough number of commits. Furthermore, the too small code increase could also have significant bias; therefore we considered those systems only where the code evolution was significant. To sum up, the selection criteria were the following:

- *Enough number of observations:* the available number of commits containing at least one Java-related operation should be at least 1,000.

- *Significant code increase:* the ratio of the maximum logical lines of code (this is typically the size of the system after the last available commit) and the minimum one (which is typically the size of the initial commit) should be at least 3.0.

We ended up with one industrial and three such open-source systems. For the industrial one we had all the information from the very beginning. For most of the open-source projects this is not the case; generally the initial source had been merged from another version control system. In order to neutralize this bias we defined the quality change of the first commit to be 0.0.

The 4 systems which fulfilled the above criteria were the following:

- **Ant** – a command line tool for building Java applications (`http://ant.apache.org`).
- **Gremon** – a greenhouse work-flow monitoring system (commercial; `http://www.gremonsystems.com`).
- **Struts 2** – a framework for creating enterprise-ready java web applications (`http://struts.apache.org/`).
- **Tomcat** – an implementation of the Java Servlet and Java Server Pages technologies (`http://tomcat.apache.org`).

Table 3.1 shows basic statistics of the selected systems.

	Ant	**Gremon**	**Struts 2**	**Tomcat**
Total commits	6,118	1,653	2,132	1,330
Commits containing Java source	6,102	1,158	1,452	1,292
TLLOC [1] minimum	2,887	23	39,871	13,387
TLLOC maximum	106,413	55,282	152,081	46,606
Number of developers	37	13	26	15
Total number of file adds	1,062	1,071	1,273	797
Total number of file updates	20,000	4,034	4,734	3,807
Total number of file deletes	204	230	308	485
Commits containing file adds	488	304	219	104
Commits containing file updates	5,878	1,101	1,386	1,236
Commits containing file deletes	55	89	94	77
Commits consisting of file adds	196	42	41	32
Commits consisting of file updates	5,585	829	1,201	1,141
Commits consisting of file deletes	19	8	12	23
Maintainability increases	1,482	456	498	269
Maintainability no change	3,051	365	710	704
Maintainability decreases	1,569	337	541	319
Maintainability mean with outliers	1.408	-5.505	-22.801	-4.719
Maintainability variance with outliers	15,274.2	11,136.0	985,767.8	138,819.5
Maintainability mean without outliers	1.782	-5.493	0.957	-0.555
Maintainability variance without outliers	5,663.9	8,662.5	6,762.7	3,431.7
Number of outliers	10	2	7	10
Percentage of outliers	0.164%	0.173%	0.400%	0.774%

Table 3.1. Systems for operations, churn and ownership analysis

- Version control history related statistics
  - Total number of available commits.
  - Number of commits affecting at least one Java source file.

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

  - Minimal / maximal logical lines of code (2 rows).
  - Number of developers.
  - Total number of Java source file additions / updates / deletes (3 rows).
  - Number of commits containing at least one Java source file additions / updates / deletes (3 rows).
  - Number of commits consisting of exclusively file addition / update / delete, considering Java source files only (3 rows).

- Maintainability related statistics
  - Number of commits with maintainability increase / no change / decrease (3 rows). The sum of these values result in the total number of commits affecting Java source files.
  - Average / variance of all the maintainability change values, i.e. considering the outliers as well (2 rows).
  - Average / variance of maintainability change values without outliers (2 rows).
  - Number / percentage of outliers (2 rows).

Considering the maintainability values, with the outliers both the means and the variances are very hectic. There are outliers in all of the projects with similar magnitudes. By removing the outliers both the means and the variances tend to have similar magnitude of values. Without the outliers the distribution of the data is close to normal.

Note that the maintainability change means are close to 0, compared with the magnitude of standard deviation. Furthermore, the means of all subdivisions are close to 0 as well. There are significant differences between the means, but these are much lower than their variances. This is important information for interpreting the results.

3.3.2 Analyzed Systems for Metric Analysis

For Section 5.2 we selected other software systems. The following factors were determinative at the analyzed systems and version selection:

- *Bug database.* We selected those versions of those systems where the bug data was available.
- *Programming language.* The RMI numbers were calculated with a tool based on the Columbus Quality Model, and it supports the Java programming language.
- *Source control.* As the version of the tool we used for extracting version control history metrics uses SVN, we considered only those software systems of which the revision history is publicly available in an SVN source control system.

Among the common intersection of the above criteria we chose widely known systems. These are the following: Ant, JEdit, Log4J, Xerces. Table 3.2 provides some information about the systems and versions we analyzed.

Name	Version	Date	Sources	Bugs
Ant	1.3	3rd March, 2001	108	33
	1.4	3rd September, 2001	162	45
	1.5	10th July, 2002	265	35
	1.6	18th December, 2003	310	166
	1.7	19th December, 2006	677	337
jEdit	4.0	12th April, 2002	275	226
	4.1	28th February, 2003	283	215
	4.2	8th August, 2004	332	106
	4.3	23rd December, 2009	434	12
Log4J	1.0	8th January, 2001	84	58
	1.1	19th April, 2001	72	82
	1.2	1st May, 2002	156	483
Xerces	1.3	31st January, 2001	301	186
	1.4	22nd May, 2001	189	580

Table 3.2. Systems for version control history metrics analysis

The table contains the name, the version, the release date, the number of considered source files (the common intersection) and the recorded number of bugs in the common intersection.

The input data, along with a function which calculates the results, can be found in the R package `hotspot` [35]. It can be installed as any other R package from the R GUI, or it can be downloaded from
`https://cran.r-project.org/web/packages/hotspot/`.
The format of the input data and the description how to perform the tests can be found in the help pages of the package. The recommended start is the main page of the package: `?hotspot`.

3.4 Random Checks

To validate the results, we performed random cross check in some cases. This was done in the following way:

- We kept the source control operation data as it is.
- We also kept the values of the quality changes, but we permuted randomly the order of the revisions it had been originally assigned to, just like mixing a pack of cards. The `sample()` R [103] function was used to permute the order.

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 primary results.

3.5 Related Work

The topic of this part of the thesis is related to the following areas of research: maintainability of program source code, mining software repositories, visualization, code churn, and code ownership. This section summarizes the most important works of these research areas, along with the theses related to this one.

3.5.1 Maintainability Related Work

Several papers deal with various software metrics based fault prediction, most of them are object-oriented ones. The quality model used by us relies on such metrics that have been proven to highly correlate with fault occurrences.

Studies by Li and Henry [89] and Chidamber and Kemerer [29] are among the earliest object-oriented metric proposals. Li and Henry [89] proposed the following metrics: coupling through inheritance, coupling through message passing, coupling through data abstraction, number of local methods, number of semicolons in a class, and number of attributes + number of local methods. Chidamber and Kemerer [29] proposed the following 6 metrics: depth of inheritance tree (DIT), number of children (NOC), coupling between objects (CBO), response for a class (RFC), lack of cohesion methods (LCOM) and weighted methods per class (WMC). The quality model used by us primarily relies on these metrics.

Brito and Melo [25] examined how metrics of object-oriented design can be used for fault prediction. Among the 6 checked metrics 4 showed a strong negative correlation (method hiding factor, method inheritance factor, attribute inheritance factor, polymorphism factor), with the fault numbers, one of them showed a strong positive correlation (coupling factor), and one (attribute hiding factor) did not show any significant correlation. The results were evaluated using programming languages C++ and Eiffel. The metrics used in that study were quite different from the ones we were using, however, this relatively early study in this area pointed out an important research direction.

Briand et al. [24] also examined the impact of the object-oriented metrics on faults. They made a case study with the help of 8 groups of students, who implemented the same task in C++. They found that the coupling and inheritance measures are strongly related to the probability of fault detection in a class. They did not find significant impact of cohesion on fault proneness. Among others, the quality model used by us uses these factors, and other studies showed significant correlation between all the above mentioned metrics and fault density in the Java programming language.

Subramanyan and Krishnan [114] examined the connection between the number of defects and the following object-oriented metrics: methods per class, coupling between objects, depth of inheritance tree and number of children. They validated the theory using both C++ and Java programming languages. All of these metrics were applied by the quality model used by us.

In their study Gyimóthy et al. [59] examined how various object-oriented metrics can be used for fault prediction. They found a strong positive correlation between the number of faults and the following metrics: number of methods per class, depth of inheritance tree, response for class, coupling between objects, lack of cohesion and number of logical lines of code. Although the validation was performed on C++ programs, the results can be applied for Java as well. The quality model used in this paper

relies heavily on these previous results.

Nagappan et al. [98] presented a universal quality model using software metrics. They used it for bug prediction; however, the method is adaptable to arbitrary measure of quality, i.e. maintainability as well. They found no single set of metrics that fitted all projects. The model used by us was shown to be an adequate one for several projects.

Moser et al. [92] presented a comparative analysis of the predictive power of two different sets of metrics for defect prediction. The described methodology provides a classification of Java sources, based on if they are defective or defect-free, with high precision and high recall. The authors found that process metrics are more effective in defect prediction than code metrics. The quality model used by us relies on code metrics only, and the above study shows us a possible direction for fine-tuning the model.

Hindle et al. [72] dealt with understanding the rationale behind large commits. They contrasted large commits against small commits and showed that large commits were more perfective, while small commits were more corrective.

Koch and Neumann [82] focused on the effect of software processes to the product quality, analyzing open source software development practices.

Bachmann and Bernstein [8] explored among others if the process quality, as measured by the process data, had an influence on the product quality. They showed that the product quality, measured by number of bugs reported, was affected by process data quality measures.

Fry et al. [51] presented their results on the comparison of the maintainability of human written and generated patches. They found that human written patches were slightly more maintainable than machine generated ones; however, they proposed a system which augmented the machine generated patches with human readable documentation, and it changed the original tendency.

Yamashita et al. [125] discussed how code smell interactions affected maintainability. They also provided evidences that code smells found in coupled artifacts had traceable effects on maintainability.

In their study [61], Hanenberg et al. presented an experiment investigating if static type systems improve maintainability compared to dynamic type systems. They found that static type systems were beneficial in understanding source code and fixing type errors, but not in fixing semantic errors.

Software quality had also a direct impact on the cost of software development and maintenance. Bakota et al. [12] showed a converse exponential relationship between the maintainability of a software system and the overall cost of development supported by an empirical validation. This result is very important for this thesis as well, as we point out where great maintainability decrease might happen, and the above study shows that this maintainability decrease will cost more in the future.

An earlier study of the cost of software quality was done by Slaughter et al. [112], financially justifying the investments in software quality. We went one step further by arguing how these investments could be used more efficiently.

In this research we analyzed Java source code, but a quality model for C# [70], Python, C++, and RPG also exist, which makes it possible to perform these studies in the future on projects written in these languages as well.

3.5.2 Mining Software Repositories Related Work

Mining software repositories is another large and evolving area that is related to this part of the thesis. In the annual conference of MSR [93] a great number of studies appear in connection with this research field. We can only scratch the surface of the MSR-related literature, which contains several hundreds of articles.

Kagdi et al. [77] provided a taxonomy of articles in this area, based on the following aspects: software evolution, purpose, representation and information sources. Hassan [66] also provided a detailed overview about the MSR field.

Atkins et al. [7] used the version control data to evaluate the impact of software tools on software maintainability. They explored how to quantify the effects of a software tool once it had been deployed in a development environment and presented an effort-analysis method that derived tool usage statistics and developer actions from a project's change history (version control system). We also tried to evaluate the maintainability changes through version control data; however, we investigated the general effect of version control operations regardless of tool usage.

Gall et al. [52] introduced their Relation Analysis that performed a deep analysis of logical coupling of modules. With the evaluation of 28 releases of an industrial software, without analyzing the source code, they were able to discover design flaws like god classes or spaghetti code.

Papers of Ying et al. [126] and Zimmermann et al. [130] described methodologies for determining change patterns (based on the change history), i.e. sets of files that had been were changed together frequently in the past. Similar investigation was done in the study by Rysselberghe and Demeyer [117] – frequently applied changes were mined from version control systems. This part of the thesis also focus on developer patterns.

Canfora et al. [26] presented a method to derive the set of source files impacted by proposed change requests, based on the data in version control (CVS) and issue tracking (Bugzilla) systems.

In the study Breu and Zimmermann [23] presented a history-based aspect mining method. They identified the aspect candidates (e.g. locking concerns) using version history. They increased the precision with the size of the projects, e.g. for Eclipse they reached 90% for the top ten candidates. The quality model used by us considers lower level metrics of source code, which is aggregated to assess the maintainability.

In the work Ratzinger et al. [105] presented an empirical study of predicting refactoring based on historical data found in version control system. We on the other hand tried to predict the future maintainability change instead of the refactors.

Robbes [107] presentsed an alternative information repository based on the IDE interactions, along with their implementation. It was especially useful for refactoring detection.

Zaidman et al. [128] investigated whether production code and the accompanying tests co-evolve by exploring a project's versioning system, code coverage reports and size-metrics. This study provides a good future direction candidate for further evolving the quality model used by us.

Peters and Zaidman [101] investigated the lifespan of code smells and the refactoring behavior of developers by mining the software repository of seven open-source systems. The results of their study indicated that engineers were aware of code smells, but they were not very concerned by their impact, given the low refactoring activity.

Giger et al. [56] explored prediction models for whether a source file would be affected by a certain type of source code change. For that, they used change data of

the Eclipse platform and the Azureus 3 project.

3.5.3 Visualization Related Work

Several diagram types exists for illustrating univariate, bivariate and multivariate data. The book by Chambers et al. [28] provides a summary of the most important possibilities. The book by Murrel [95] focuses on the diagram creating possibilities in R statistical programming language [103].

One of the most frequent diagram type for illustrating univariate numeric data is the box plot. However, as it became very popular, researchers faced its shortcomings. Several proposals appeared to make it better.

McGill et al. [90] suggested variable width and notched box plots. Not to forget that in those times the computers were expensive and slow, and the diagrams were mostly drawn by hand. Benjamini [14] exploited the capability of the computer. Frigge et al. [49] dealt mainly with the problem of outliers. Potter et al. [102] provided a summary of the variations of box plots.

Probably the most important problem with box plot is that it hides the local densities. To overcome this shortcoming, in R the density plot could be a good choice in several cases. Other popular diagram types handling this issue are violin plots (R function `vioplot()` in package `vioplot` [2], presented by Hintze and Nelson [73]) and bean plots (R function `beanplot()` in package `beanplot`, described by Kampstra [79]).

The problem of illustrating bivariate data is also very common. Goldberg and Iglewicz [57] presented an early proposal of a bivariate extension of boxplots. An interesting two dimensional extension of the box plots is the bag plot, as article by Rousseeuw et al. [108] suggests. For the implementation, see R function `bagplot()` in package `aplpack` [123].

Visualizing multivariate data is even harder. Hornik et al. [75] suggested a framework for visualizing multi-way contingency tables.

The presented R functions are mainly based on base package `graphics`. Another basic visualization related package in R is `grid`. The `lattice` package is based on `grid`; it was presented by Sarkar [109].

3.5.4 Code Churn Related Work

Analyzing the effect of code churn on source code, especially for defect prediction, is an intensively investigated research area.

Khoshgoftaar et al. [81] presented a gross change prediction improvement using neural networks. Their measure of quality was 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. [80] 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. [94] presented how they calculate 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. [100] analyzed the same phenomena as we did, the code erosion; they used the phrase "code decay." They referred 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.[60] presented their concept of code delta and code churn compared to a baseline with the help of a real, industrial software system.

Eick et al. [32] 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 could be fatal, i.e. not possible to change further. In this thesis we also analyze how historical changes affect the maintainability, and we go one step further: considering the code decay as an evidence, tried to identify why, when and where it occurs.

Nagappan et al. [96] 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 correlated with defect density. They also concluded that relative code churn measures were good and absolute code churn measures were poor defect density predictors. They found that these relative code churn measures were good predictors of system defect density, and they could be efficiently used to distinguish between fault-prone and non fault-prone binaries.

The same authors presented another approach of post-release defect prediction [97], considering software dependencies and code churn. They found that this combination was 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.[6] 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. [110] described 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. [55] showed that code churn defined simply by number of lines modified was not so good an error predictor as fine-grained source code changes defined by them. They tested their concept on the source code of Eclipse.

3.5.5 Code Ownership Related Work

There are several papers dealing with the topic of code ownership or developer related issues.

In their work Mockus et al. [91] presented a case study of the Apache Server open source development. Among others they considered the topic of code ownership as well. They analyzed a single project which had nearly 400 contributors and concluded that in the analyzed project no real code ownership was evolved. We analyzed 4 systems, with the magnitude of 1-3 dozens of developers each and analyzed the effects of the code ownership on future maintainability.

In a study, Nordberg [99] described four types of code ownership: product specialist, subsystem ownership, chief architect and collective ownership. They discussed the advantages and disadvantages of each models. Our findings support the base assumption of this study: in case of lack of well defined code ownership the code quality is likely to decrease. We did not consider code ownership models in such detail, but presented the most obvious developer related facts for the 4 analyzed systems.

LaToza et al. [87] presented the results of two surveys and eleven interviews conducted by software developers at Microsoft, regarding software development questions. Some of the questions were related to code ownership as well. An interesting statement of this article is that code ownership can also be wrong, as if a code is understood and maintained by a single developer, it makes individuals too indispensable. As an alternative of individual code ownership, the team code ownership was also investigated. Contrary to them, we examined the effect of the code ownership on the maintainability, i.e. study why code ownership was good, but from the organizational level the aspects could be different in the longer term.

Fritz et al. [50] investigated the frequency and recency of interactions on the code by developers: questions had been asked to find out if they were able to recall types of variables, types of parameters, method names, another method calls and methods which calls a specified method. They showed that according to the assumed hypothesis, the developers knew their own code better (that had been modified by him/her frequently and recently) compared to a foreign code. We, on the other hand, analyzed code ownership instead of code knowledge.

Weyuker et al. [122] investigated if their already presented fault prediction model could be enhanced by including the number of developers. They found that the achieved improvement was negligible, which might be surprising at a first glance. We, on the other hand, found a significant correlation by examining the number of different developers' effect on maintainability. The contradiction could be resolved by the following: an already well established model cannot be enhanced further significantly by including the number of developers predictor; but it itself is a good predictor of maintainability change and of defects as well.

In their study Bird et al. [20] investigated if there were significant differences in software quality following a distributed development model compared to a collocated development. They analyzed the development of Windows Vista and argued that the differences were hardly notable. As a complementary result they found a positive correlation between the number of developers and defects, which result is similar to ours. We did not consider the distance among development team members, but analyzed the effect of ownership on software maintainability.

The same authors [21] presented a fault prediction method, which combined social factors in development organizations and program dependency information. They

found that this was a better model than considering only one of the factors. They proved their concept on 2 huge projects: Windows Vista and Eclipse. We also used both social and technical networks implicitly: the social one was the number of developers of a module, and the technical one was the sources committed together.

The problem of code ownership, especially finding the hidden co-authors, was analyzed by Hattori et al. [67]. They created a tool called Syde, which recorded every change by every commit, and with the help of this information they were able to determine the code ownership more precisely. They validated their concept using a commercial system. We also analyzed code ownership, bud did not consider developer interaction information.

Rahman et al. [104] introduced a code ownership and experience based defect prediction model, but instead of just considering the modifications performed on source file itself, they introduced a fine-grained level by analyzing the contributions to code fragments. We on the other hand performed our analysis conventionally on source file basis.

The study by Bird et al. [22] targeted a similar goal to ours; as its title said: the effects of ownership on software quality. The authors investigated 2 huge projects: Windows Vista and Windows 7. We, on the other hand, investigated 4 smaller projects. They considered software quality in terms of pre-release faults and post-release failures; we considered code maintainability as an aggregated value of complexity metrics. They performed the analysis on binary and release level; our study was based on source code and commits. For a binary they defined the terms minor contributor (developers who contributed at most 5% of the total commits), major contributor (above 5%) and ownership (proportion of the commits of the highest contributor). Among others, they found that software components with many minor contributors had more failures than other software components. Moreover, the high level of ownership resulted in less defects. These findings are very similar to ours: by increasing the number of developers and therefore decreasing the ownership the software quality tends to be reduced.

3.5.6 Related Theses

There are already theses written in this field of research in the University of Szeged. Here we summarize those parts of these theses, which are related to this one.

As already mentioned in Part I, Árpád Beszédes [15] summarized the C++ source code analysis In his thesis Rudolf Ferenc [46] presented the details of the Columbus C++ analysis tool, adequate for modelling and reverse-engineering. In this work the author presented the design pattern recognizing algorithm in the C++ source code, and presented a fault proneness analysis as well. The connection between that thesis and this one is that the Columbus analyzer was later adopted to Java, which is the basis of the Quality Model, on which the second part of this thesis relies on.

In the thesis István Siket [111] summarized the studies about object oriented source code metrics, adequate for bug prediction. The analysis was still done using C++; however, that work was very important basis of the Columbus Quality Model.

In his thesis Tibor Bakota [9] summarized the Columbus Quality Model itself, which is based on the Columbus Java analyzer tool. See Section 3.2.1 of this thesis for a summary. In this work the author also presented the effect of the maintainability on development costs. In the third major part the author presented a summary about the code duplications analysis. Indeed, clone coverage is one of the sensor metrics the

Columbus Quality Model.

In his thesis Péter Hegedűs [71] presented advances of the quality model evolution. He adopted the Columbus Quality Model on C#. The he presented the source code element level software quality model, calculating the Relative Maintainability Index. See Section 3.2.2 of this thesis for an overview. Finally, he presented some applications of the Quality Models, like bug localization and revealing the effect of coding practices. At the end he carves the surface of the impact of version control operations on maintainability, which is indeed the Thesis Point 2.A of this thesis; see Section 4.1.

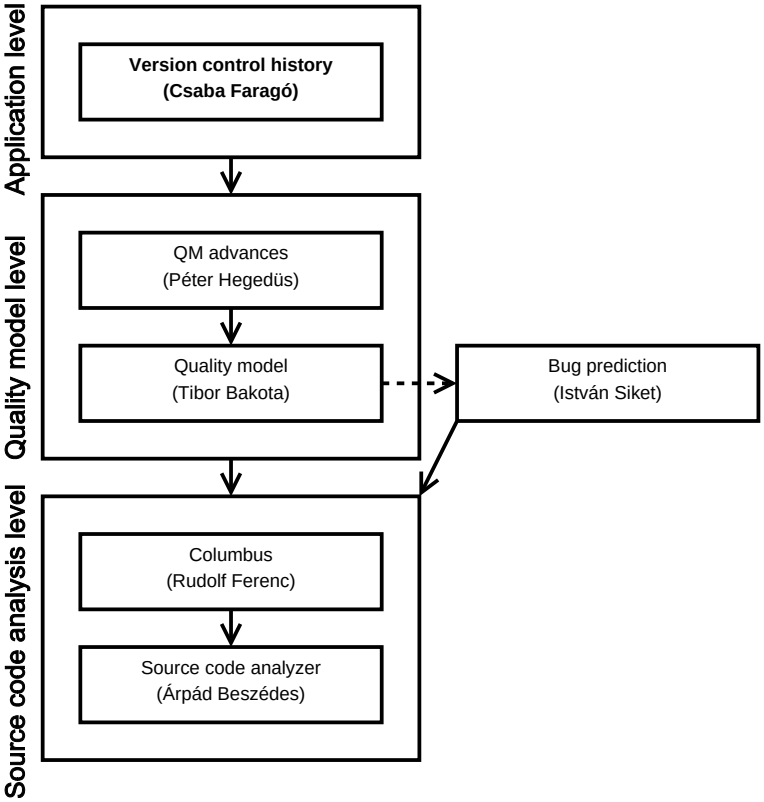

Figure 3.4. Related theses

Figure 3.4 presents an overview how these theses fit together, and how the present thesis fits into those theses.

3.6 Summary

This chapter contains information related to the remaining chapters of the second part of the thesis.

First in Section 3.1 we provided a general overview of the research field, pointing the actual state of the art, illustrating the position of this thesis, and highlighting the possible future directions of research.

Then in Section 3.2 we presented the quality model used for measuring the maintainability. In Section 3.2.1 we presented the Columbus Quality Model, which can be used for measuring a system level maintainability. The model considers source code metrics like logical lines of code, coding rule violations or code complexity, then compares them with other systems' values found in the benchmark, finally it aggregates

the results of the comparisons. The maintainability was calculated for every revision of the analyzed systems.

In Section 3.2.2 we presented an algorithm for determining the Relative Maintainability Index of any source code element (like class or function) within a system.

In Section 3.3 we presented the analyzed systems. We used two sets of analyzed systems, because of differences in methodology.

In Section 3.4 we presented the idea of cross check with randomized data. As we tried to find connection between two independent series of data, the main idea was the following: we kept one data series as it was, and we permuted randomly the other series, like shuffling a pack of cards.

Finally, in Section 3.5 we provided an overview about the related work of this research field. We summarized the work related to maintainability (Section 3.5.1), mining software repositories (Section 3.5.2), visualization (Section 3.5.3), code churn (Section 3.5.4) and code ownership (Section 3.5.5). At the end we summarized the related theses as well (Section 3.5.6).

3.7 Contributions

The results presented in this chapter are those on which the results presented in Chapters 4 and 5 are built upon. Therefore these are mostly not the work of the author of the thesis. The author's contributions are the following:

- Finding examples for demonstrating the maintainability changes (second part of Section 3.2.1).
- Taking part in the formalism of drill-down methodology (Section 3.2.2).
- Taking part in the selection of the analyzed systems, collecting some of the data and creating summary statistics (Section 3.3).
- Creating the idea of randomized cross check, along with the elaboration, implementation and evaluation (Section 3.4).
- Providing taxonomy of most of the related work (Section 3.5).

"If there are no ups and downs in your life, it means you are dead."
— Internet folklore

4

Thesis Point 2: Connection between Version Control Operations and Maintainability

Continuing the quote: if there are no ups and downs in your source code, it means your software is dead. Can we say anything about these ups and downs? In this section we deal with this topic.

In Section 4.1 (**Thesis Point 2.A**) we show that connection between version control operations and the quality change of the source code exists. In Section 4.2 (**Thesis Point 2.B**) we analyze the 3 operations types (file addition, update and delete) one by one, on two aspects of the maintainability change: its absolute change and its variance. In Section 4.3 (**Thesis Point 2.C**) we present how we overcome the problem of visualization of the found results.

4.1 Thesis Point 2.A: Existence of the Connection between Version Control Operations and Maintainability

For this first step, we checked the version control operations only; and within this set of information we focused exclusively on the mere number of various operations, i.e. how many files were added, updated and deleted within that commit. We were motivated by the question: did the way of introducing code changes (reflected by version control operations of different commits) have a traceable impact on software quality? Did all types of commit operations contribute to software erosion, or are there exceptions?

Other commit-related information, like the certain files affected, the change itself, the comment, the date and the author, are considered in Section 5.

4.1.1 Overview

The types of the version control operations and the maintainability of the code are at first glance remote concepts, more or less independent from each other. Furthermore, as no finer grained information is considered at this point (e.g. what was changed in the file, who made the change, or even on which file the change was performed), the distance between the maintainability change and the commit operations is even higher. Therefore, it is a non-trivial question if there is any connection between the two data sets at all.

Suppose that there was a connection between version control operations and maintainability changes in case of each examined projects, we were interested in finding out which were the common patterns, i.e. those connections which were significant for every examined project. These can be formed as general statements.

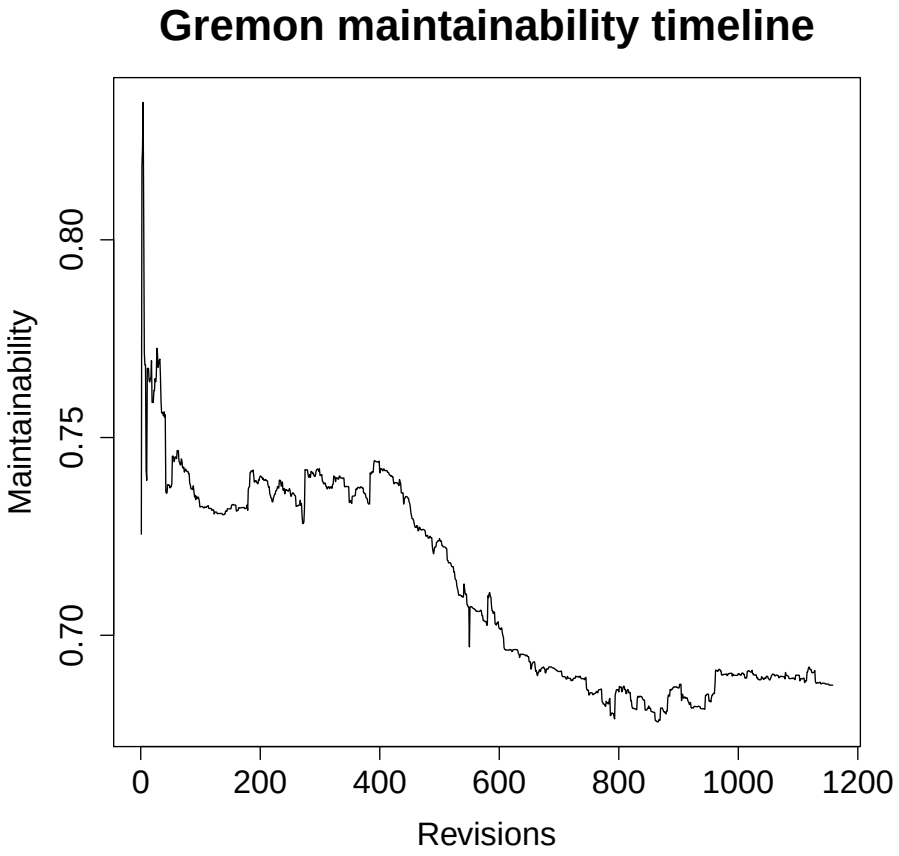

Figure 4.1. Maintainability values of the Gremon project

By performing experiments we tried to find evidences which support or reject some of our more concrete assumptions based on Figure 4.1. The beginning of the time line is very hectic. This is the start of the project with many additions of new parts. The maintainability becomes smoother later on, and the long-term tendency is negative. This is the phase when modifications on the existing sources are performed, and less new sources are added. Furthermore, based on our experiences, developers tend to pay more attention on the quality when adding new code than updating it later due to e.g. bug fixing, and this is especially true for the code originally developed by someone else. It is a hard task in itself to understand the code, reproduce the error, debug and find

the solution, therefore developers under time pressure are glad if they find a solution; finding a *nice* solution is often not reached.

Based on the above explained expectations we formulated the following research questions:

2.A.RQ1: *Do commits containing file additions to the system have a significant positive impact on its maintainability?*

2.A.RQ2: *Is it true that the commits containing source file updates only tend to significantly decrease the maintainability?*

2.A.RQ3: *Do commits containing file deletion improve the maintainability of the system?*

In Section 4.1.2 we present the methodology and in Section 4.1.3 we present the result

4.1.2 Methodology

In this section we describe the methodology of the analysis we perform on the input data. The data comes from 2 independent sources: measurement of the maintainability of each revision of the analyzed system on one hand, the related number of operations on the other hand. The quality model we used for calculating the maintainability is described in Section 3.2.

Version Control Operations

We considered the mere numbers of various operations, e.g. 2 files were added, 5 files were updated and 1 file was deleted within the examined commit. We analyzed Java source files only, so we skipped all other types of file system entries like directories or non-Java files (e.g. xml files). We did this because the actual version of the used quality model considers Java source files only.

Besides *Add*, *Update*, and *Delete*, there is a fourth version control operation: *Rename*. As there were hardly any Rename operations in the examined data (it occurred only in one of the analyzed projects with very low cardinality) this operation was not considered. Therefore, the input data collected from the version control system was an integer triple for each commit containing at least one Java source file:

- **A** - the total number of file additions,
- **U** - the total number of file updates,
- **D** - the total number of file deletions.

Contingency Table

The contingency table is a two-dimensional table with the maintainability changes in the rows and version control operation categories in columns, and the cells containing the total number of commits in the category causing that kind of maintainability change.

We partitioned the maintainability changes into three sets:

- +: positive change,
- 0 : no traceable change,

- $-$: negative change.

The maintainability change is positive if the calculated value of the actual commit is higher than the value of the previous commit, negative if it is lower and 0 if the two values are the same.

We divided the commits into several disjoint categories based on the version control operations they include. We defined the categories based on intuition coming from the principal component analysis (PCA) of the industrial project's data set. We defined the following categories:

- D: commits containing at least one *Delete* operation,
- A: commits containing no *Delete* operation, containing at least one *Add* operation,
- $U+$: commits containing *Update* operations only; the number of *Update* operations is at least 2,
- $U1$: commits consisting of exactly one *Update* operation.

Please note that the union of these commits is the full set of examined commits. Commits affecting no Java files do not have any effect on the calculated maintainability, therefore we omitted these from the calculation.

Bar Plot Diagrams

In order to visualize the data found in the contingency tables we used proportional bar plot diagrams. Each commit category was represented by a bar, which was divided into 3 parts: the proportion of positive, zero and negative maintainability changes within that category. For a better comparison we also presented the proportions of the full commit set.

We could also get intuitions about the answers of the research questions based on these diagrams. If there are spectacular differences among categories within a project, and there are similarities in the diagrams among projects, then it suggests that the connection between the version control operation types and the maintainability is quite strong.

Contingency Chi-squared Test

To give well-grounded answers to our research questions we performed Chi-squared tests [5] (similarly to the method presented by Ying and Robillard [127]) on the contingency tables.

This test calculates the expected values based on the sum of rows and columns, i.e. what were the values if there were no connection between version control operations and maintainability. Then it determines if the differences between the actual and the expected values are significant or not. The null-hypothesis is that these values are the same, and the reason of the differences are random. The final result of this test is practically the p-value, indicating the chance of the result being at least as extreme as the observed, provided that the null-hypothesis is true.

We performed the test using the `chisq.test()` R function [103]. This function calculates the standard residuals (*stdres*) as well for each cell, i.e. what would the

value be if the data were of standard normal distribution. E.g. if this value was -2.0, then it would mean that the number of the observed elements was less than the expected (see the negative sign), and the difference was as much likely to be random as a standard normally distributed variable is at least as extreme as 2.0 (i.e. less than -2.0 or greater than 2.0).

Based on these standard residuals we calculated the p-value as follows. The R function `pnorm()` calculates the distribution of the given values, i.e. the proportion of elements less than or equal to the provided one. E.g., this value is 0.5 for 0.0, 0.023 for -2.0, 0.977 for 2.0 etc. Based on the definition of the p-value, the result for value 0.0 would be 1.0, i.e. there was no deviation from the expected value at all. To go on with the running example, for -2.0 we need to calculate the proportion left to -2.0 and right to 2.0, and sum it. As mentioned, the first value is 0.023, while the second one is also 1.0-0.977=0.023. Therefore the p-value is 0.046.

This process is illustrated in Figure 4.2. The size of both gray areas is 0.023. The lower dashed line is at 0.023, while the upper one is at 0.977.

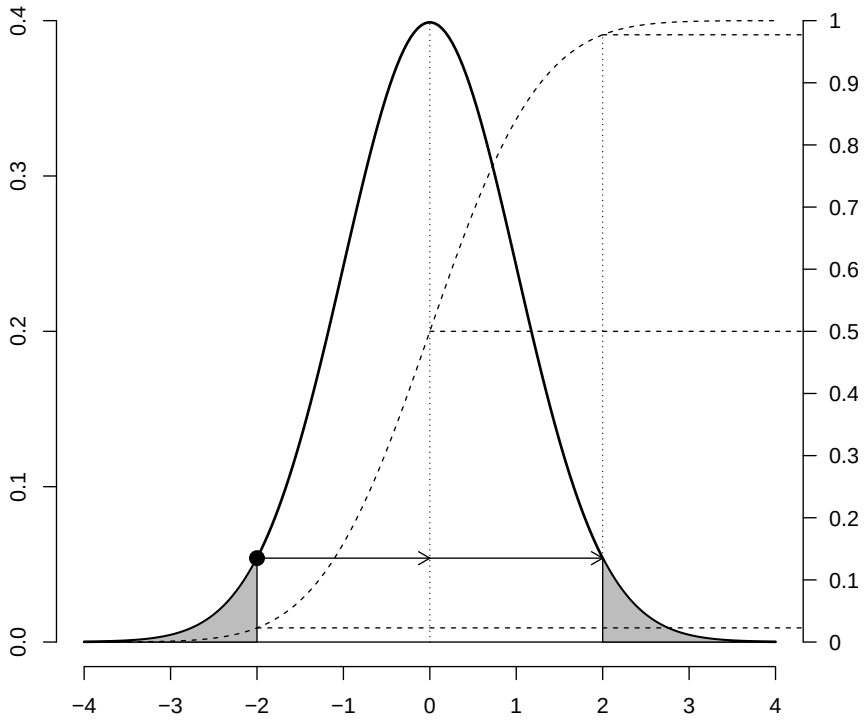

Figure 4.2. Standard normal distribution

To summarize, we had the following formula for calculation:

$$2 \cdot pnorm(-abs(x))$$

where x is the value of standard normal distribution. The cells containing small p-values can be considered as significant results.

In order to provide a quick and easy overview of the results, we performed one last step: we calculated the number of zeros between the decimal point and the first non-zero digit of the p-value, with the appropriate sign, denoting the direction of the deviation from the expected value (negative if it is less than the expected, positive if it is greater). More formally, if the canonical form of the p-value was $(a \cdot 10^b)$, the transformed value was the absolute value of the exponent minus one (i.e. $|b| - 1$), with the sign of the standard residual. E.g., in the above example the p-value in canonical form is $4.6 \cdot 10^{-2}$, and the sign of -2.0 is negative, therefore the transformed value is -1. 0 means that the random probability is at least 10%, 1 and -1 means that it is between 1% and 10% and so on. Formally, this transformation is calculated by the following function:

$$f = \left\lfloor log\frac{1}{p} \right\rfloor \cdot sign(stdres)$$

This test also gives a common p-value, i.e. not only cell based p-values. Having a low enough such p-value ($p < 0.01$) would answer positively the base question if there was a connection between version control operations and maintainability.

For answering the research questions formally, we took the last, transformed table. In case of the cell-based approach we considered those values significant, where the absolute values were at least 2 ($p < 0.01$) for all of the checked software systems.

4.1.3 Results

The Input Contingency Tables

The contingency tables created for the examined projects can be found in Tables 4.2, 4.1, 4.3 and 4.4.

	A	D	$U+$	$U1$	$\sum$
+	277	18	472	715	1482
0	13	12	625	2401	3051
-	172	25	467	905	1569
$\sum$	462	55	1564	4021	6102

Table 4.1. Contingency table of Ant

	A	D	$U+$	$U1$	$\sum$
+	118	43	122	54	337
0	13	3	126	223	365
-	109	43	198	106	456
$\sum$	240	89	446	383	1158

Table 4.2. Contingency table of Gremon

	A	D	$U+$	$U1$	$\sum$
+	123	43	183	149	498
0	17	25	166	503	711
-	82	46	233	179	540
$\sum$	222	114	582	831	1749

Table 4.3. Contingency table of Struts 2

	A	D	$U+$	$U1$	$\sum$
+	39	31	91	108	269
0	8	14	159	523	704
-	27	32	100	160	319
$\sum$	74	77	350	791	1292

Table 4.4. Contingency table of Tomcat

There are a couple of notable facts about the tables. First of all, the distributions of the positive, neutral and negative commits within each commit category are different. Second, these distributions seem to be similar in every project. This is promising, and worth the effort of the detailed analysis.

Figure 4.3 shows a graphical overview of the data, where we illustrated the proportions of each commit category on bar plot diagrams. The bars with different shades indicate the proportions of the positive (light gray), neutral (gray) and negative maintainability change related commits (dark gray) for each category, and we displayed the overall proportion as well. In order to see the differences between the random and the actual data, we included the results of random executions for each project (see Figure 4.4).

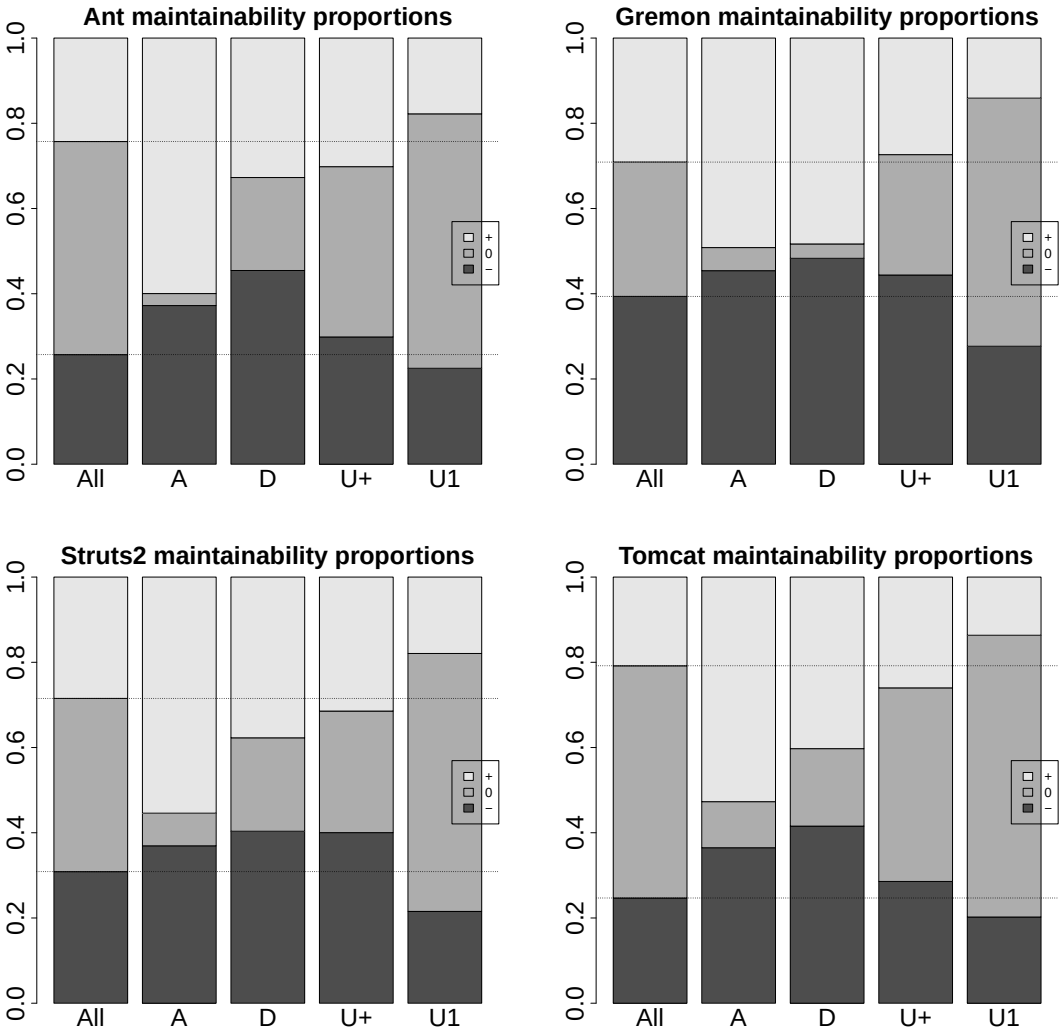

Figure 4.3. Maintainability change proportions of subdivisions

We can see the following on these diagrams:

- The middle bars (gray) are smaller than expected in case of A, D and U+, and higher in case of U1.
- The upper bar (light gray) is the tallest in case of A on every diagram.
- In case of U+ and U1 the lower bars (dark gray) are bigger than the upper ones (light gray) in most of the cases.

The relevance of these results are very spectacular if we compare them to the bar plots of the randomized data (see Figure 4.4). In case of randomized data, there are no

obvious differences in any category bar, compared to the bar of all commits (or with the bar of any other category). Furthermore, even the viewable small differences in the bars do not tend to be relevant: one difference on one diagram mostly differs on the other ones.

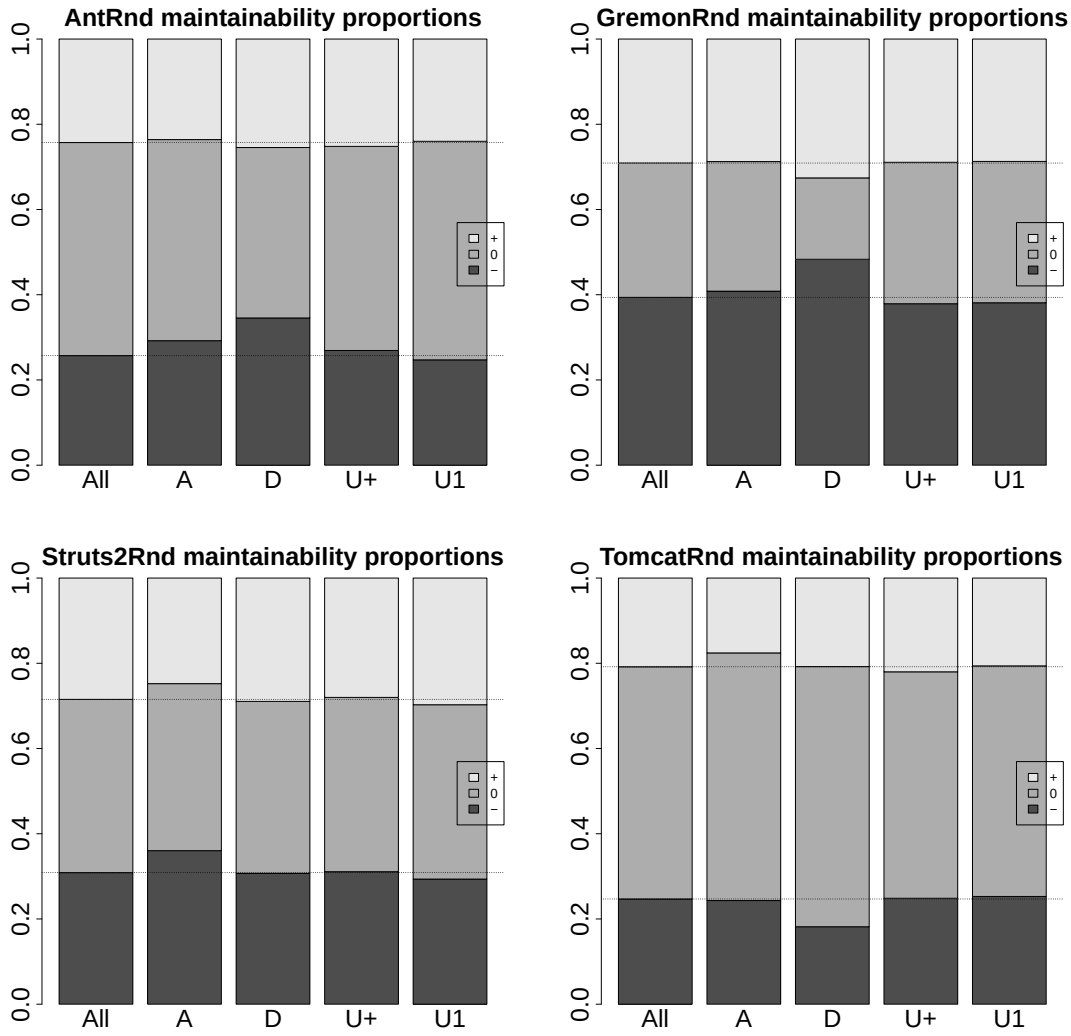

Figure 4.4. Maintainability change proportions in random cases

Results of Contingency Chi-Squared Tests

Based on the bar plot diagrams (see Figure 4.3) we already have an assumption about the answers to our research questions, but for a more grounded answer let us check the results of the Chi-squared tests on the contingency tables. In case of the Gremon project we present all the details. For the open-source systems we show only the input and the final results from which we can draw the main conclusions.

As already mentioned, Table 4.2 presents the original contingency table for the Gremon project on which the test was performed on. For example, the meaning of the upper left value (118) is the following: the total number of commits containing no deletion, containing at least one addition (i.e. belongs to category A based on the definition) and the maintainability change caused by that commit is positive. The last

row and the last column contains the sum of the values of the appropriate rows and columns, respectively. This is the consolidated input of the contingency Chi-squared test.

Table 4.5 contains the calculated expected values. Practically, this is the null-hypothesis: if the row and column sums would be the same as in the case of measured data, then in case of uniform distribution these were the cell values. The average values of random cases would tend to this matrix. The sums of rows and columns are the same as in the previous table. The meaning of the upper left value (69.8) is the following: if there was no connection between version control operations and maintainability change, and the number of commits in each category would be the same as in case of the input, furthermore, the total numbers of positive, neutral and negative maintainability changes were also the same, then this value would be an integer close to this number. In other words: the average value of this cell in the random cases would tend to this value. In this case the value 69.8 is much smaller than the value 118 found in the previous matrix (see Table 4.2).

Table 4.6 shows the standard residuals. This table illustrates if the previous difference is significant or not using the well-known standard normal distribution. The difference between the expected and the measured value is exactly as extreme as the difference between 0 and the values found in this table assuming a standard normal distribution. E.g., in the upper left case this is the chance of resulting in 7.69. Based on this, we already have a feeling that this is a very extreme value; the probability of resulting such value only by chance is very low.

	A	D	$U+$	$U1$	$\sum$
+	69.8	25.9	129.8	111.5	337
0	75.6	28.1	140.6	120.7	365
-	94.5	35.0	175.6	150.8	456
$\sum$	240	89	446	383	1158

Table 4.5. Expected values of Gremon

	A	D	$U+$	$U1$
+	7.69	4.15	-1.04	-7.90
0	-9.78	-5.95	-1.89	13.75
-	-2.15	1.80	2.77	-5.73

Table 4.6. Standard residuals of Gremon

Table 4.7 presents the p-values related to the standard normal distribution. These values answer the question of how low the previously mentioned chances are. Consider the upper left value again. The difference between the actual value (118) and the expected value (69.8) is 48.2. The other value with the same difference from the expected one is 21.6 (=69.8-48.2). The definition of the p-value is the following: the chance of the value being at least as extreme as measured, provided that the null-hypothesis is true. Therefore the meaning of the value in the upper left corner ($1.52 \cdot 10^{-14}$) is the following: the chance that the measured value is at least 118 or at most 21.6. Taking into consideration that its reciprocal is about $6.58 \cdot 10^{13}$ it means that in a random case this would statistically happen once in about every 66 trillion cases (and about once in every 132 trillion cases if the direction also matters).

Table 4.9 contains the exponents calculated as described in Section 4.1.2, Contingency Chi-squared Test part. Theoretically, the previous tables contain everything we need: the standard residuals provide the directions and the p-values table provide the absolute values; but the tables containing the exponents are easier to read and comprehend.

	A	D	$U+$	$U1$
+	$1.52 \cdot 10^{-14}$	$3.28 \cdot 10^{-5}$	$3.00 \cdot 10^{-1}$	$2.76 \cdot 10^{-15}$
0	$1.43 \cdot 10^{-22}$	$2.70 \cdot 10^{-9}$	$5.81 \cdot 10^{-2}$	$5.06 \cdot 10^{-43}$
-	$3.15 \cdot 10^{-2}$	$7.25 \cdot 10^{-2}$	$5.69 \cdot 10^{-3}$	$1.01 \cdot 10^{-8}$

Table 4.7. Cell-based p-values of Gremon

Table 4.9 is composed of the exponents and the directions. Consider the upper left value (13). The absolute value comes from the exponent (14) minus one (in order to convert the absolutely not significant results (having p-value > 0.1) to 0 instead of 1). The sign means the direction: the positive in this case means that the actual value is higher than the expected one. Also note that although this value is high, it is still far from the highest.

Tables 4.8, 4.10 and 4.11 show the resulted exponents for the Ant, Struts 2 and Tomcat projects, respectively.

	A	D	$U+$	$U1$
+	76	1	9	-60
0	-98	-4	-19	98
-	8	3	4	-14

Table 4.8. p-value exponents of Ant

	A	D	$U+$	$U1$
+	13	4	0	-14
0	-22	-8	-1	42
-	-1	1	2	-7

Table 4.9. p-value exponents of Gremon

	A	D	$U+$	$U1$
+	20	1	1	-19
0	-26	-4	-12	57
-	1	1	8	-15

Table 4.10. p-value exponents of Struts 2

	A	D	$U+$	$U1$
+	11	4	2	-14
0	-14	-10	-4	25
-	1	3	1	-5

Table 4.11. p-value exponents of Tomcat

Table 4.12 summarizes the overall p-values of each contingency Chi-Squared test (we calculated the previous p-values on a per cell basis).

System	**p-value**	**Significance**
Gremon	$1.19 \cdot 10^{-52}$	very strong
Ant	$1.60 \cdot 10^{-151}$	very strong
Struts 2	$4.47 \cdot 10^{-64}$	very strong
Tomcat	$4.84 \cdot 10^{-33}$	very strong

Table 4.12. Overall p-values of Chi-Squared tests

Based on these extremely low overall p-values in every case, we stated that there was a strong connection between the version control operations and the maintainability

changes.

For getting a better overview, we summed up the resulted exponents. Table 4.13 presents these sums, indicating those cells where the results are significantly similar for the systems. Dark cell means that the absolute value in every case was at least 2 ($p < 0.01$). The darkness indicates the degree of similarities in the significance. If there are 2 or 3 significant results and 1 not significant, it is indicated with a lighter color. 0 or 1 significant result is denoted by an even lighter cell fill. We reserved white for significant contradictions, i.e. if a cell would contain -2 or less in one case, and +2 or more in the other.

	A	*D*	*U+*	*U1*
+	120	10	12	-107
0	-160	-26	-36	222
-	9	8	15	-41

Table 4.13. Sum of the exponents of each cells

Half of the cells are dark; these indicate those results which are significant for every checked project. Please note that the table does not contain any white cells.

Random Check Result

We were also interested in the random case: does it also result in the same high numbers as presented previously or not. Based on the definition of the exponent table, theoretically, in a random case the proportion of 0 should be 90%, the proportion of absolute values 1 should be 9% (half of them negative and half of them positive), the proportion of absolute values 2 should be 0.9%, and so on. We received approximately the same kind of distributions in practice. Table 4.14 illustrates the results of one concrete execution with an overall p-value of 0.53. There are hardly any non-null values in these executions.

	A	*D*	*U+*	*U1*
+	0	0	0	0
0	0	0	0	0
-	1	0	0	0

Table 4.14. Exponents in case of randomized cross-check

Answers to the Research Questions

The answers to the research questions are primarily based on Table 4.13.

2.A.RQ1: *Do commits containing file additions to the system have a significant positive impact on its maintainability?*

The values in the first column are related to these commits. Value 120 and the dark color cell in the upper left cell indicates that the positive impact on the maintainability is very high for those commits which do not contain deletion but contain at least one

addition. *This supports our assumption that adding new source files to the system has a significant positive impact on its maintainability.*

On the other hand, the lower left cell of the table is also positive (+9), but the color is lighter. In 3 out of the 4 cases it contains a value close to 0, and one higher value. If we check the input, we see that the absolute number of commits in the positive cell is also higher than those in the negative cell in every case. Therefore we can also say that the overall effect of the add operation is positive.

2.A.RQ2: *Is it true that the commits containing source file updates only tend to significantly decrease the maintainability?*

The third and fourth columns are related to commits containing updates only. All the colors of the cells found in the fourth column (commits containing exactly one update) are dark and the values are negative both in the + and - cells. But the value found in the + row is much lower than the value found in the - row, and this is true for every input. We should also take into account that the maintainability tends to decrease, therefore if these values were equal, that would also mean maintainability decrease as an overall result. Thus *in the case of commits containing one update our assumption that the source file updates tend to decrease the maintainability is supported with high significance.*

The cell colors in the third column (commits containing exclusively at least 2 updates) are lighter. Both of the values found in the + and - rows are positive. However, the value found in the - cell is higher than the value in the + cell. Therefore *in the case of more updates our assumption is also supported, but with lower significance.*

2.A.RQ3: *Do commits containing file deletion improve the maintainability of the system?*

The second column is related to this research question. The values found in these cells are small in absolute values compared to those found in other columns and their colors are also not the darkest ones. The number in the + cell (10) is higher than the number in the - cell (8). Based on this we cannot formulate a general statement. Seemingly we could say that deletions have no positive effect on the maintainability as $10 > 8$. But that could be a false conclusion, because in general the number of commits causing negative maintainability change is in general higher than those causing positive change. Therefore 10 in the + cell does not necessarily mean higher number of absolute values than 8 in the - cell. And if we check the inputs, we see that just the opposite is true, i.e. the absolute number in the - cells in columns D are less than or equal with the values in the + cells. If we consider the input as well we find that there are more such commits of category D which resulted in maintainability decrease than those of increase. Therefore *the third assumption that commits containing deletion improve the maintainability of the system is not supported by the results.*

Other Results

Considering Table 4.13 we can read out other results as well, not covered by the original research questions.

First of all, the highest absolute value is 222, in row 0, column U1. All the other values in row 0 are negative. This means that *no traceable maintainability changes are primary related to small updates.* This is a trivial statement, of course, and it rather validates the used quality model than a real usable result of this research.

The second highest value in absolute is -160, also in row 0, but column A. Therefore *adding a new source code almost always has some traceable effect on the maintainability.*

Considering the negative (-) row alone, it would lead the false result that every commit category have negative effect on the maintainability, except the small updates. This is not true, because the value found in the positive (+) row should also be considered in case of every category. On the other hand, these values tell us that *with the exception of small updates there are too many maintainability decreases.* Eliminating some of these decreases would result in a well maintainable code, even without an explicit code quality increase campaign.

4.2 Thesis Point 2.B: Impact of Version Control Operations on Value and Variance of Maintainability

4.2.1 Overview

In this section we consider file additions, file updates and file deletions one by one, and check their impact on the size and the variance of maintainability change. The reason of the value check of maintainability is obvious: as we want to find typical patterns causing similar effect on maintainability, this analysis is a very important part of our long term vision.

The reason of the variance check is that if the net effect of one commit set is similar to another one, the difference in amplitudes can be important. The limited amount of efforts allowed to spend on source code quality improvements could be better allocated by focusing on those commits which cause higher maintainability change. This is analogous with the greenhouse effect. In the greenhouses the temperature is high because it does not decrease overnight. In software development, the elimination of the drastic maintainability decreases would result in a net maintainability increase. Therefore it is recommended to pay special attention to those commits which are likely to cause higher change of maintainability (high variance), compared to those likely causing lower change (low variance).

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

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

2.B.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?*

2.B.RQ3: *What is the impact of operation Add, Update and Delete on the variance of maintainability change?*

In Section 4.2.2 we describe the methodology, and in Section 4.2.3 we present the results.

4.2.2 Methodology

We estimated the maintainability as described in Section 3.2, and calculated the number of operations as described in Section 4.1.2. Here we need the maintainability change, not only its sign. First in this section we describe how we calculated this value. Then we show how all the commits are divided into disjoint subsets, on which we perform the tests. Then we present the two sample Wilcoxon rank test used for value

comparison, and variance test used for variance comparison. We performed randomized cross-check for value comparison; we present a few details for better understanding at the end of this section.

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 was feasible because the goodness value was derived from a probability function with normal distribution. We performed this transformation with the `qnorm()` R function [103]. The transformed values serves as the basis of the subtractions.
- We multiplied the results of the subtractions (the maintainability value differences) by the actual size of the system, more specifically, the actual total logical lines of code (TLLOC, number of non-comment non-empty lines of code).

Figure 4.5 illustrates why the quantile conversion was necessary. The same difference on the y axis is not the same after quantile conversion (x axis), as expected.

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

Divisions

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

We defined 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 three ways:

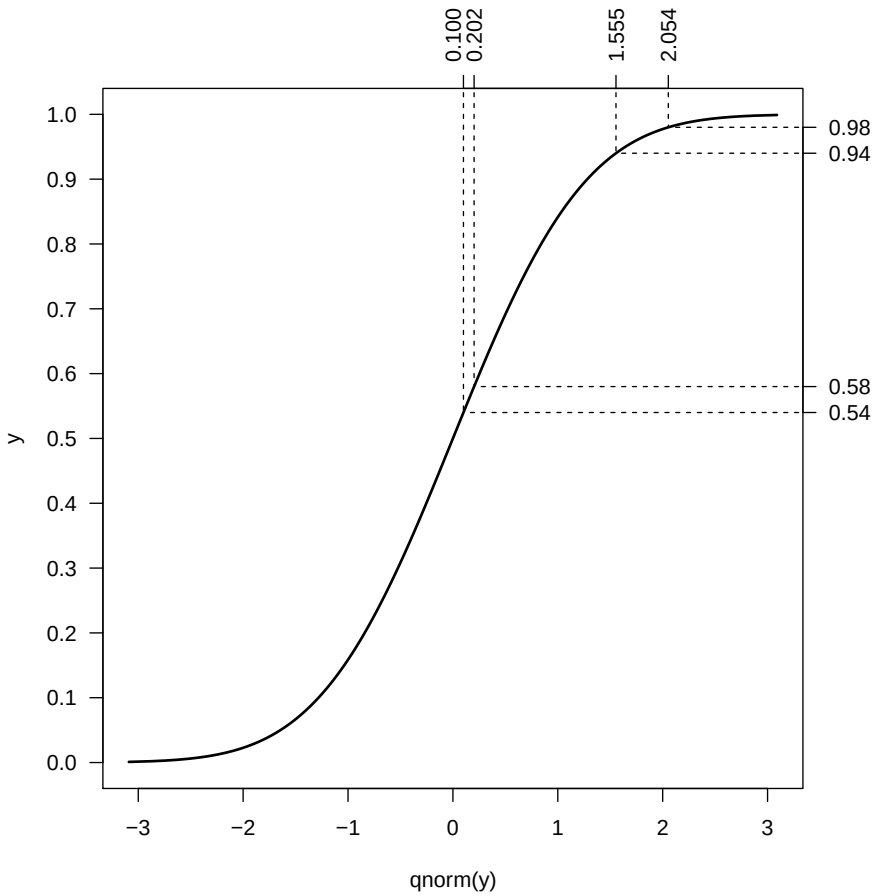

Figure 4.5. Illustration why quantile conversion is necessary

- We divided the main dataset into two, based on the median of the absolute number of the examined operations. The greater values went into the first dataset, the second dataset was the complementary of the first one considering the main dataset.
- We divided the main dataset into two based on the median of the proportion of the examined operations, with similar division.
- We took 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. We illustrate all of them 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 a 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 a 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 do 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 division is used to find out if it is true that commits which consist of more file additions result better maintainability compared to those consisting of less number of additions. This division is 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 in 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 4.15 illustrates these divisions.

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

Table 4.15. Overview of operation based commit divisions

We executed the tests (detailed later in this section) on all of these combinations during the experiment. In the case of median divisions, if the median was ambiguous,

we tested both cases (checking into which subset these elements should be added), and we considered the better division (the more balanced one).

Two-Sample Wilcoxon Rank Test for Value Comparison

After the partitioning detailed above we examined the maintainability changes of the commits belonging to these subsets. To check if the differences were significant or not, we used the two-sample Wilcoxon rank test (also known as Mann-Whitney U test) [74]. 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.

We performed the tests by the `wilcox.test()` function in R [103]. 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 was no difference between the distributions of the maintainability change values in the two commit sets. The alternative hypothesis was the following: the elements (maintainability differences) in one subset were 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 followed 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 were less and which were greater).

As we performed the test twice each time, the results were two p-values. We denoted them with p_1 (in case of values in the first set were 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 was 0.0046 being numbers in one subset greater than those in the other, which also meant that the p-value of having smaller values in the first set was 0.9954. Please note that the sum of these p-values are always 1.0, i.e. 100%. From the two p-values we considered the better one. Therefore the result was practically always exactly twice as good as it would be in case of a two direction test.

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 was $(a \cdot 10^b)$, the transformed value was 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 was $4.6 \cdot 10^{-3}$, so the absolute value of the exponent was 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:

$$f = \left\lfloor log\frac{1}{p_1} \right\rfloor - \left\lfloor log\frac{1}{p_2} \right\rfloor \tag{4.1}$$

In order to present the result in the table in concise format, we calculated these exponent values for every possible combination and we summed them 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 got a matrix with the version control operations in the rows and analyzed systems in the columns, and an integer value in each cell. This was only an approximation, first of all, because the divisions were not independent. However, it was adequate for a quick overview, and for comparing the results of different systems.

Random Checks

To validate the results, we performed a random analysis as well, as described in Section 3.4. Here we present what these checks mean from the exponent point of view.

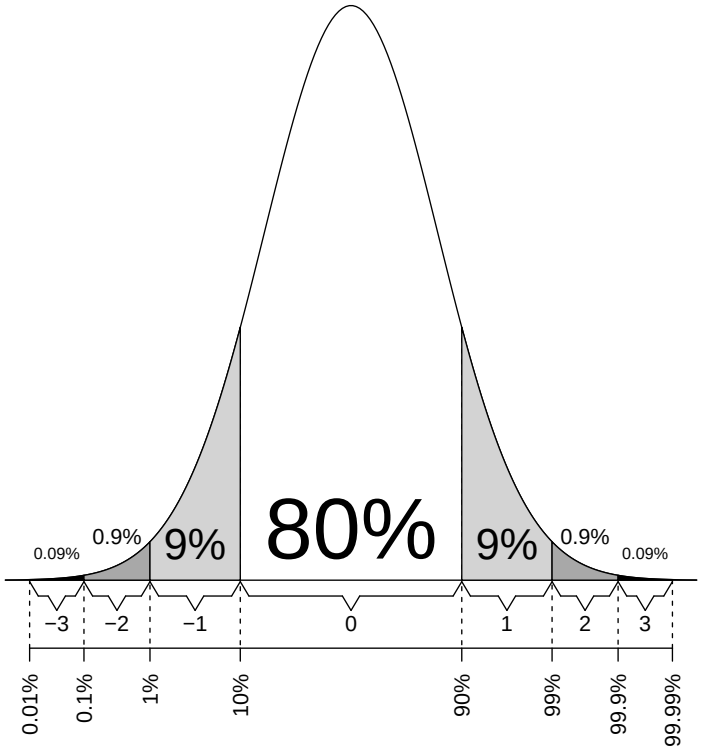

Figure 4.6. Illustrating the calculated exponent values

The expected values of the exponents in random case can be derived from the diagram in Figure 4.6: 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. In 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.

As we executed 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 was 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 accepted the absolute values 4 and higher as significant.

Variance Test

F-tests are a family of statistical tests. One of them is the F-test of equality of variances, which checks if two normal populations have the same or different variances.

We performed this test on each division combinations using the `var.test()` function in R [103]. This function calculates both the ratio of variances, and the p-value under the null-hypothesis that the variances are the same. The result of this test is the ratio of variances of the values in the sets defined by the divisions, along with the p-value. The p-value indicates the probability of the ratio of variance being at least as extreme as the calculated one, provided that the variances are equivalent. More specifically, the null-hypothesis is that the ratio of variances is 1.0.

For a better illustration we calculated the geometric mean of the ratios (per division basis, taking all the analyzed software) as well. We did not perform the variance test on the cases where the size of at least one of the subsets was below 5, because if the number of observations was low, then there was a high risk that the result was false.

Handling Outliers

The variance test is very sensitive to the outliers. A few, unusual commits (e.g. merging the resulting code of a development performed on another branch, adding code developed in another version control system or renaming a huge number of source files in two steps) cause drastic increase in variance. To neutralize this bias, we eliminated from the analysis the commits with very high absolute maintainability change values. Therefore we omitted commits where the absolute value of the maintainability change exceeded a considerably high value.

We checked the absolute maintainability change values of these extraordinary commits, and we found they were at the magnitude of 10,000. On the other hand, the typical absolute maintainability change value was at the magnitude of 100 or lower. Therefore we omitted commits having maintainability change value higher than 1,000.0. Only a few commits per software system caused higher absolute maintainability change than this value (see the outliers in Table 3.1). We performed the tests with other limits, like 500.0 or 2,000.0, and the results were similar, therefore we found that the limit of round 1,000.0 was a sound one.

4.2.3 Results

Summarized Results of the Wilcoxon Tests

We show the results of the methodology introduced in Section 4.2.2, Wilcoxon test part, in Table 4.16. The absolute number reflects the magnitude of the impact, while the sign gives the direction (maintainability increase or decrease). Figure 4.7 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.

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

Table 4.16. Version control operation check: sum of the exponents of p-values

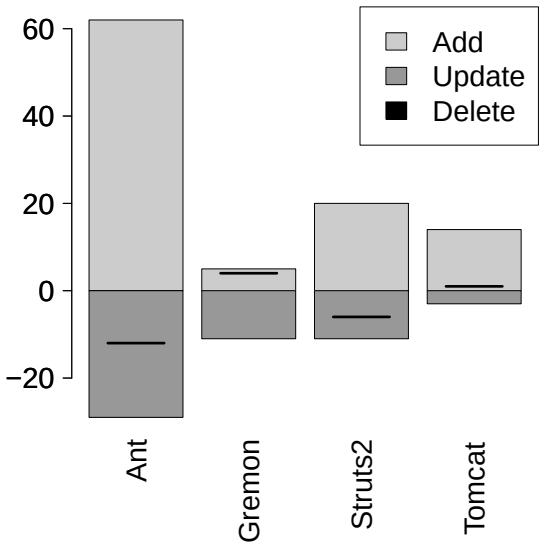

Figure 4.7. Illustrating version control operation check results with bars

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

Wilcoxon Tests Details

For details on the above numbers consider Table 4.17. We show the sum of the rows as well, which helps us drawing the attention on the most promising results. We recall the probabilities in random case, to illustrate the magnitude of the numbers in the first 3 columns.

The diagrams in Figure 4.8 illustrate the results of the Wilcoxon rank tests visually, where we present the values found in the summary column of Table 4.17. 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.

We also illustrate the hectic results of operation Delete (right bars, black).

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

Table 4.17. Version control operation check: exponent details

Now let us check the results in the tables.

Operation Add. 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. 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 yields significant result (exponent ≥ 2). This means that *commits containing additions have better effect on the maintainability compared to those containing no file additions.*

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

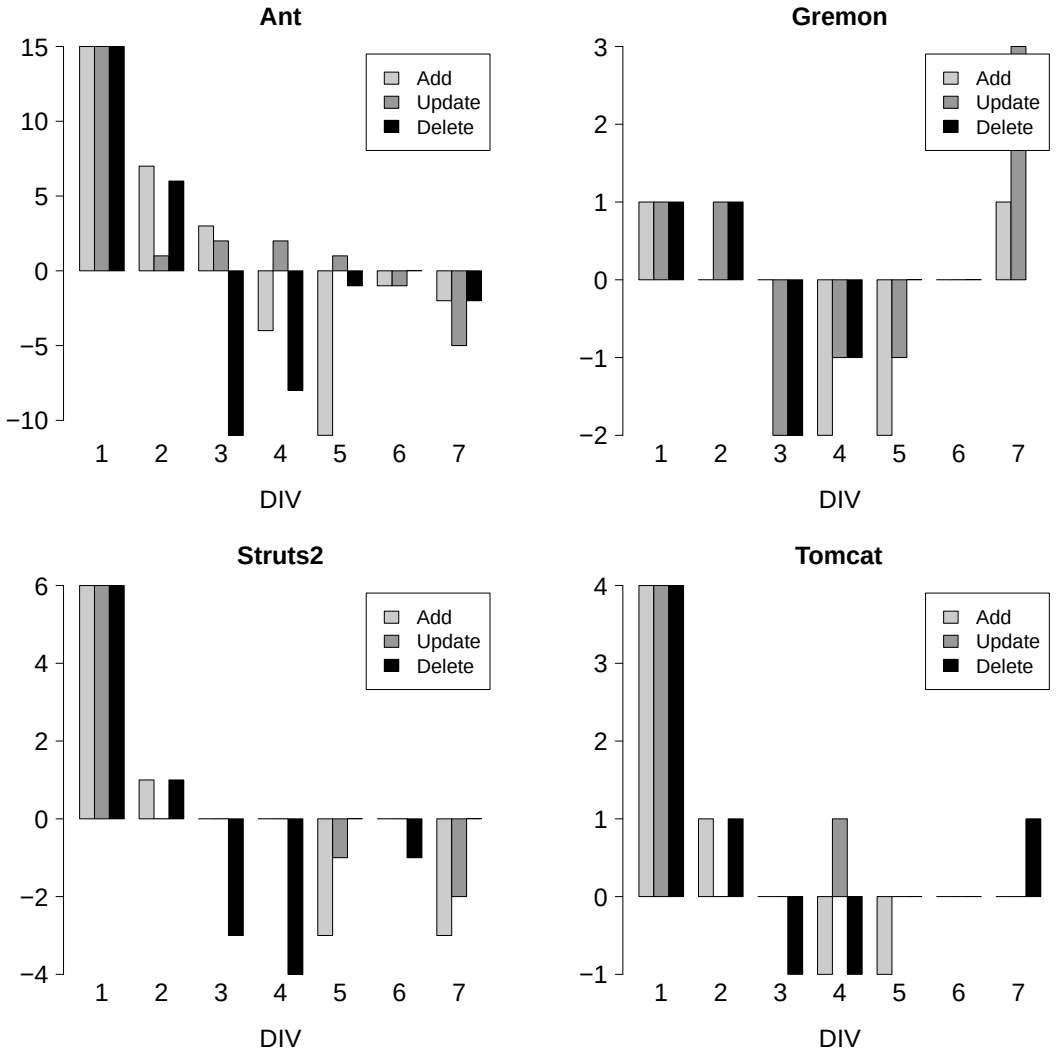

Figure 4.8. Version control operation check: exponent details illustrated with bars

connection between the number of operations and the maintainability is weak.

Operation 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 is 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.

Operation 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 contains 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.

Results of the Variance Tests

We performed also the variance tests on all the defined 21 combinations for all the 4 analyzed systems. First, we examined the visual representation of the results illustrated in Figure 4.9.

We generated the figure with the help of R and it contains 3 bar diagrams, one for each operation (Add, Update, Delete). The bars are divided into 7 subsets in all 3 cases, one for each division. In case of every operation division pair there are 5 bars: 4 indicating the results for each project (the thin gray ones), and one for their geometric

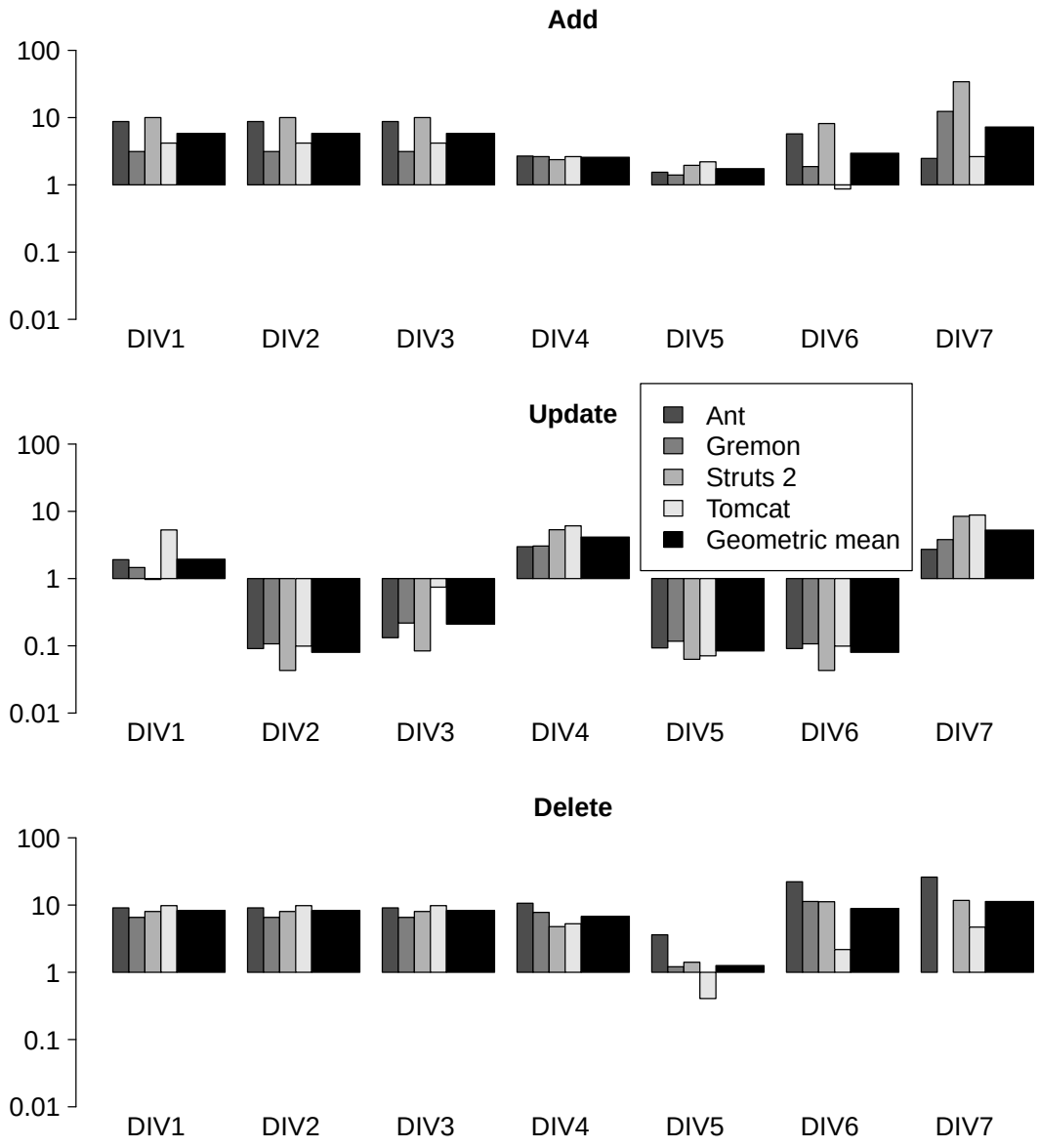

Figure 4.9. Illustration of variances

mean (the thick black one).

We illustrate the data on a logarithmic scale diagram. By using a normal scale, one could hardly see the difference between small absolute values, even could not see if it is above or below 1.0. The starting-point of the bars is at 1.0. This means that the ratio of variances higher than 1.0 are represented with a bar on the positive direction (above 1.0), and those of less than 1.0 are represented with a bar on the negative direction (below 1.0).

On this diagram the most important information, i.e. if the ratio of variances is above or below 1.0, is the most spectacular. Furthermore, one can really compare the same magnitude but opposite direction values: for example, the ratio of variance of 5.0 and 0.2 are practically the same magnitude with opposite direction; the logarithm scale diagram represents these values with the same absolute size of bars, one located above and the other located below 1.0.

The results can be learned informally even from this diagram, without the necessity

of studying the numerical values. In cases of operation **Add** and operation **Delete**, almost all the bars are located on the positive part of the diagram, meaning that these operations increase the variance. On the other hand, in case of operation **Update** the picture is mixed. Some of the bars (DIV1, DIV4 and DIV7) are positive, while the others are negative.

In almost all cases *the results are similar* (i.e. the sizes and the directions of the bars are similar) for all the 4 projects. However, there are some exceptions; the most spectacular one is DIV5 in case of operation Delete.

If the data is not available (operation Delete, DIV7, project Gremon), then a small empty gap can be found on this bar diagram. Note the difference between the value close to 1.0 (e.g. operation Update, DIV1, project Struts 2) and the missing result.

Now we examine the results more formally. Table 4.18 contains the ratio of variances, where the last columns show the geometric means (GM) of the values in the rows.

Operation	**Division**	**Ant**	**Gremon**	**Struts 2**	**Tomcat**	**GM**
Add	DIV1	8.74	3.13	10.01	4.18	5.82
	DIV2	8.74	3.13	10.01	4.18	5.82
	DIV3	8.74	3.13	10.01	4.18	5.82
	DIV4	2.68	2.63	2.37	2.63	2.57
	DIV5	1.54	1.4	1.95	2.2	1.74
	DIV6	5.73	1.87	8.16	0.868	2.95
	DIV7	2.47	12.36	34.17	2.63	7.24
Update	DIV1	1.91	1.46	0.97	5.29	1.94
	DIV2	0.091	0.107	0.043	0.099	0.08
	DIV3	0.132	0.218	0.084	0.745	0.21
	DIV4	2.98	3.05	5.32	6.08	4.14
	DIV5	0.093	0.117	0.063	0.071	0.084
	DIV6	0.091	0.107	0.043	0.099	0.08
	DIV7	2.72	3.8	8.43	8.8	5.26
Delete	DIV1	9.08	6.55	8.02	9.8	8.27
	DIV2	9.08	6.55	8.02	9.8	8.27
	DIV3	9.08	6.55	8.02	9.8	8.27
	DIV4	10.64	7.78	4.78	5.27	6.76
	DIV5	3.61	1.21	1.41	0.407	1.26
	DIV6	22.19	11.3	11.2	2.18	8.85
	DIV7	26.04	NA	11.72	4.7	11.28

Table 4.18. Ratio of variances

Table 4.19 contains the calculated p-values that represent the chances of the results being at least as extreme as in the table, provided that the null-hypothesis is true, i.e. the variances are the same. Please note that instead of executing a two-tailed test, we executed two times a one tailed test, and took the better result. The differences between the two methods can be neglected in most of the cases. We treated the p-values lower than 0.01 to be significant, which is indeed 0.02 in case of executing a two-tailed test.

0.0 means that the calculated p-value is so low that the R package is not able to

handle it and results in zero. The lower limit R can handle is about 10^{-350}. In case of very low values the exponential format is used as the exponent of 10. All the values are rounded up.

From this point on, to increase the readability, we refer the term *variance of maintainability change* is simply as *variance*.

Operation	Division	Ant	Gremon	Struts 2	Tomcat
Add	DIV1	0.0	0.0	0.0	0.0
	DIV2	0.0	0.0	0.0	0.0
	DIV3	0.0	0.0	0.0	0.0
	DIV4	10^{-13}	10^{-8}	10^{-6}	0.00037
	DIV5	0.00040	0.019	10^{-4}	0.0032
	DIV6	0.0	0.00098	0.0	0.34
	DIV7	10^{-4}	10^{-6}	0.0	0.039
Update	DIV1	0.0	10^{-5}	0.35	0.0
	DIV2	0.0	10^{-147}	0.0	10^{-124}
	DIV3	10^{-187}	10^{-22}	10^{-159}	0.062
	DIV4	0.0	0.0	0.0	0.0
	DIV5	0.0	10^{-126}	10^{-242}	10^{-131}
	DIV6	0.0	10^{-147}	0.0	10^{-124}
	DIV7	0.0	0.0	0.0	0.0
Delete	DIV1	0.0	0.0	0.0	0.0
	DIV2	0.0	0.0	0.0	0.0
	DIV3	0.0	0.0	0.0	0.0
	DIV4	10^{-8}	10^{-9}	10^{-8}	10^{-6}
	DIV5	0.0013	0.27	0.11	0.0049
	DIV6	0.0	10^{-11}	0.0	0.0029
	DIV7	10^{-4}	NA	10^{-4}	0.014

Table 4.19. p-values of the variance tests

Operation Add. All the values, with one exception, are higher than 1.0 and almost all of them are significant, meaning that operation Add increases variance. This is true for all kinds of occurrences – the simple presence, the high absolute number and high proportion as well.

For DIV1, DIV2 and DIV3 all the values within a project are the same, which is spectacular in the diagram as well. The reason is that the occurrence of operation Add is relatively low in the commits compared to operation Update. Therefore these are practically the cases where the first subset contains those commits which include at last one file addition, and the second one are those containing no file addition at all. This is the definition of DIV3. These values are very high (the geometric mean of the values is 5.82, which is among the higher values in the table), meaning that *the existence of operation Add heavily increases the variance.*

Among commits containing at least one file addition, the higher number of additions still increases the variance, see the values in row DIV4. The values found in this case (geometric mean of 2.57) is lower than those in the previous case; however, this is still significantly greater than 1.0.

Considering the high proportion of operation Add within commits containing at least one Add, it increases the variance a bit, see the results of DIV5. All the values found in that row are slightly higher than 1.0 with a geometric mean of 1.74. One of the values have a lower significance (one-tailed p-value of 0.019). It is interesting that these values are much lower than those in case of absolute median division.

Next row (DIV6) is a slightly contradictory: there is both a high value (8.16) and a value lower than 1.0 (0.868); however, the latter one is not significant. The relatively weak conclusion of this is that *commits containing operation Add exclusively have a higher variance compared to those commits containing at least one other operation.*

Finally, based on the results of DIV7, *among commits containing exclusively file additions the higher number of affected files resulted higher variance.* The values are located on a wide scale. The highest value in the table (34.17) is found here, while 2 of the 4 values are around 2.5. The smaller values could be the effect of the natural fact that higher number of any operation causes higher variance (i.e. higher amount of work is more likely to cause code quality change compared to a one line modification). Furthermore, this is a good example why calculating geometric mean (7.24) is a better choice than arithmetic mean (that would be 12.91); the former one expresses the common result much better.

Operation Update. In case of the Update operation there are many values significantly lower than 1.0. The first row (DIV1) presents hectic results, containing a high value (5.29) and a value lower than 1.0 (0.97, not significant) as well. The geometric mean (1.94) meaningfully expresses the results, namely that *the high number of operation Update slightly increases the variance.* This result is caused by a mixture of two factors. First, higher number of operations increase the variance – having more lines of code changed it is more likely that the net maintainability change would be bigger. Second, operation Update basically lowers the variance in itself; for comparison see the results of operation Add above and operation Delete below. This hectic behavior is the root of these two contradicting factors.

Next, DIV2 contains significantly lower values than 1.0, meaning that *commits containing higher proportion of Updates cause a lower variance in maintainability change, compared to those containing lower proportion of them.* For example, knowing the fact that the mean value of maintainability changes are close to 0, compared with their variance, this generally means that commits containing at least 80% Updates cause significantly lower absolute maintainability change than those of containing only at most 20% of Updates.

Values in the next (DIV3) row are still significantly lower than 1.0, indicating that *commits containing at least one file update cause lower variance compared to those containing no file updates at all.* However, the values are higher (i.e. closer to 1.0, meaning less significant) than those found in case of DIV2. This result is surprising, a lower value (higher reciprocal) was expected.

Based on values in row DIV4 it can be concluded that *among commits containing at least one update, those of containing higher number of updates cause higher absolute maintainability changes.* The values are relatively high (geometric mean is 4.14), and

we have similar values in all cases. Comparing the values with those of DIV1, in DIV4 only one of the 2 factors described above is present – the higher number of operations; the other one is not, as all the commits contain Updates per definition.

Values in row DIV5 are similar to values of DIV2, *among commits containing at least one file Update, high proportion of the operation reduces the variance.*

Examining values found in row DIV6 we find that these are the same as those in row DIV2. This is because most of the commits contain at least one Update, and the relative median division is made with a 100% threshold (DIV2). This is exactly the same as the definition of DIV6. Reformatting the sentence learned based on this definition, *commits containing exclusively Update operation causes lower variance compared with variance caused by those commits containing at least one other type of operation.*

Finally, the third kind of absolute median division (DIV7) also resulted values significantly higher than 1.0, meaning that among commits consisting of operation Update only, higher number of files affected causes significantly higher variance in maintainability change, compared with those of affecting lower number of files. Comparing the values first with other absolute median divisions of operation Update (DIV1 and DIV4), all the values are positive; second, with the results of DIV7 tests of operation Add, the variance of the values is similarly high.

Operation Delete. Operation Delete basically increases the ratio of variances. The first 3 division tests (DIV1, DIV2 and DIV3) give the same results in all cases; the reason for this is the same as described in the case of file addition. The resulting values are high, even higher than in the case of operation Add, meaning that *the presence of operation Delete causes even higher variance of maintainability change than caused by operation Add.*

Values in DIV4 are even higher, meaning that *among commits containing at least one delete those containing higher number of this operation cause significantly higher variance.*

The values in the next row (DIV5) are controversial, having one significantly higher value than 1.0, one significantly lower than 1.0, and 2 of non-significant results. This means that based on these data we *cannot formulate any statement about the variance caused by higher proportion of operation Delete among commits containing at least one of this operation.* The geometric mean of the values is slightly above 1.0.

Finally, DIV7 contains even higher values, meaning that *among those commits containing Delete operation exclusively the higher absolute number causes significantly higher variance.* The values are so high that it cannot be explained simply by "the more the higher" rule (i.e. more work causes higher variance), as it is the case with operation Update. Furthermore, the scale of variances is also high in this case. Please note the NA (not available) value in case of Gremon – since we found only 8 commits of this type (see Table 3.1), we decided not to include this result. We left it out also when calculating the geometric mean.

Utilization of the Results

This section illustrates how the results can be utilized in practice. We try to simulate the maintainability tendency of software development.

Suppose that we have a fixed budget for a deeper investigation of 10% of the program source code changes by an expert (beyond the normal code review). We expect that

with the help of the expert's hints the effect of maintainability decreasing changes will be reduced by 50%, but there will be no effect of the review on commits resulting maintainability increase anyway. The question is, how to use the limited budget most efficiently.

In the example below we have 1,000 commits of 2 types: high number of low variance commits with negative expected value, and low number of high variance commits with positive expected value. For this reason, we generated 200 random numbers of normal distribution with a mean of +1.0 and standard deviation of 10.0, and 800 random numbers of normal distribution with a mean of -0.25 and standard deviation of 1.0. In this example these numbers represent the numeric value of maintainability change of the commit in question, i.e. the normalized difference of the maintainability value of the new and the actual revision.

We can see that the sum of the low variance values is expected to be around -200, and the sum of the high variance values is expected to be around +200, therefore the sum of all the values together is expected to be about 0.

Now let us check the possibilities how the effort of reviewing 100 commits (the 10% of total commits) can be distributed.

1. In the naive case the distribution of the review budget will be random, at least from the variance point of view. In this scenario the reviewed commits could be influenced by the following factors: the availability of the expert, the actual approach of the management and so on. We simulate this case by randomly choosing 100 commits for review.

2. One could argue that it is most likely to gain useful information if we focus only on the cases with negative expected value. We simulate this by selecting 100 random cases out of the 800 values of low variance and negative expected value.

3. Another one could argue that the values of high variance should be considered, because by eliminating high decreases we gain much more, even along with the "missed shots", compared to the previous cases (provided that the expected value is close to 0, compared with the amplitude of the variance, i.e. it is likely to have large negative values). Therefore we select 100 random cases out of the 200 values of high variance and positive expected value.

We executed the simulation (programmed in R) of the above three strategies 10 times. Table 4.20 illustrates the results of these simulations. First column indicates the sequence number of the execution, the second one is the case without the code review (the total amount of maintainability change of the 1000 commit), the third one is the random case, the fourth focuses on the negative values, and the fifth focuses on the high variance.

We can see that it is a waste of efforts focusing on the low variance commits with negative expected values comparing with the totally random base case (compare the values of the third and fourth columns), but the best strategy is to concentrate exclusively on the values with high variance (compare values in the last column with the other ones). Therefore knowing the expected variance of the values can help us using the limited efforts more efficiently.

Nr.	Initial	Random	Negative Mean Value	High Variance
1	52.3	124.6	81.6	181.7
2	74.8	130.4	105.0	227.5
3	220.5	254.1	247.3	406.5
4	-170.5	-118.6	-140.4	7.8
5	118.5	170.9	145.7	286.7
6	-61.6	-9.9	-33.9	132.8
7	-159.0	-99.9	-130.7	52.7
8	91.3	155.4	114.0	289.8
9	11.4	57.6	35.6	156.7
10	-102.3	-56.1	-80.2	80.2

Table 4.20. Results of simulation for utilization of the results

Answers to the Research Questions

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

Consider Table 4.16. For interpretation of the magnitude of these values please consider the random probabilities (see explanatory text of Figure 4.6).

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 two positive and two negative results, containing low and high absolute values as well. Considering these data only we cannot formulate a valid statement for this operation.

2.B.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 4.17. 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.

2.B.RQ3: *What is the impact of operation Add, Update and Delete on the variance of maintainability change?*

Considering Table 4.18 and the related explanation, we can conclude the following.

The higher number of operation Add results in a higher variance in maintainability change.

The presence of operation Update decreases the variance of the maintainability change. Absolute median division is an exception – commits containing more updates increase the variance compared to those commits containing less updates.

Operation Delete increases the variance of maintainability change. The highest ratio of variances were caused by the presence of operation Delete.

4.3 Thesis Point 2.C: Cumulative Characteristic Diagram and Quantile Difference Diagram

4.3.1 Overview

In research data arise. Visualization of this is very important, as a diagram may reveal important characteristics. Furthermore, illustrating the statistic tests with proper diagrams might help understanding the results.

A great number of diagram types exist, but sometimes none of them are really adequate for visualization. In this study we present 2 new diagram types. One of them we call Cumulative Characteristic Diagram, and abbreviate CCD. The other one we call Quantile Difference Diagram, and abbreviate QDD.

These diagrams helped us in further research, and we found them useful in illustrating the results of Contingency Chi-Squared test, the Wilcoxon test and variance test. We implemented the diagrams in R [38] and published in our study [37] and case study [33].

Motivation

The motivating example came from the box plot diagram of the input data. Consider the divisions described in Section 4.1.2. The related box plot diagram is illustrated in Figure 4.10. Note that this version does not contain the outliers; the diagrams with outliers were even worse. On the diagram the leftmost box plot illustrates all the data, and the rest four represents the data falling into disjoint subsets.

Based on this example we framed the Cumulative Characteristic Diagram, which proved to be suitable for illustrating the results. Furthermore, this diagram type helped us to identify additional connections not discovered earlier. The analysis of these earlier unrevealed findings lead us to framing the Quantile Difference Diagrams.

4.3.2 Diagrams

Cumulative Characteristic Diagram (CCD)

The input of the base diagram is a set of numbers. In the first step, these numbers are sorted non-ascending. Then cumulatives are calculated for every element: the series starts with 0, the next element will be the value of the first element of the sorted array,

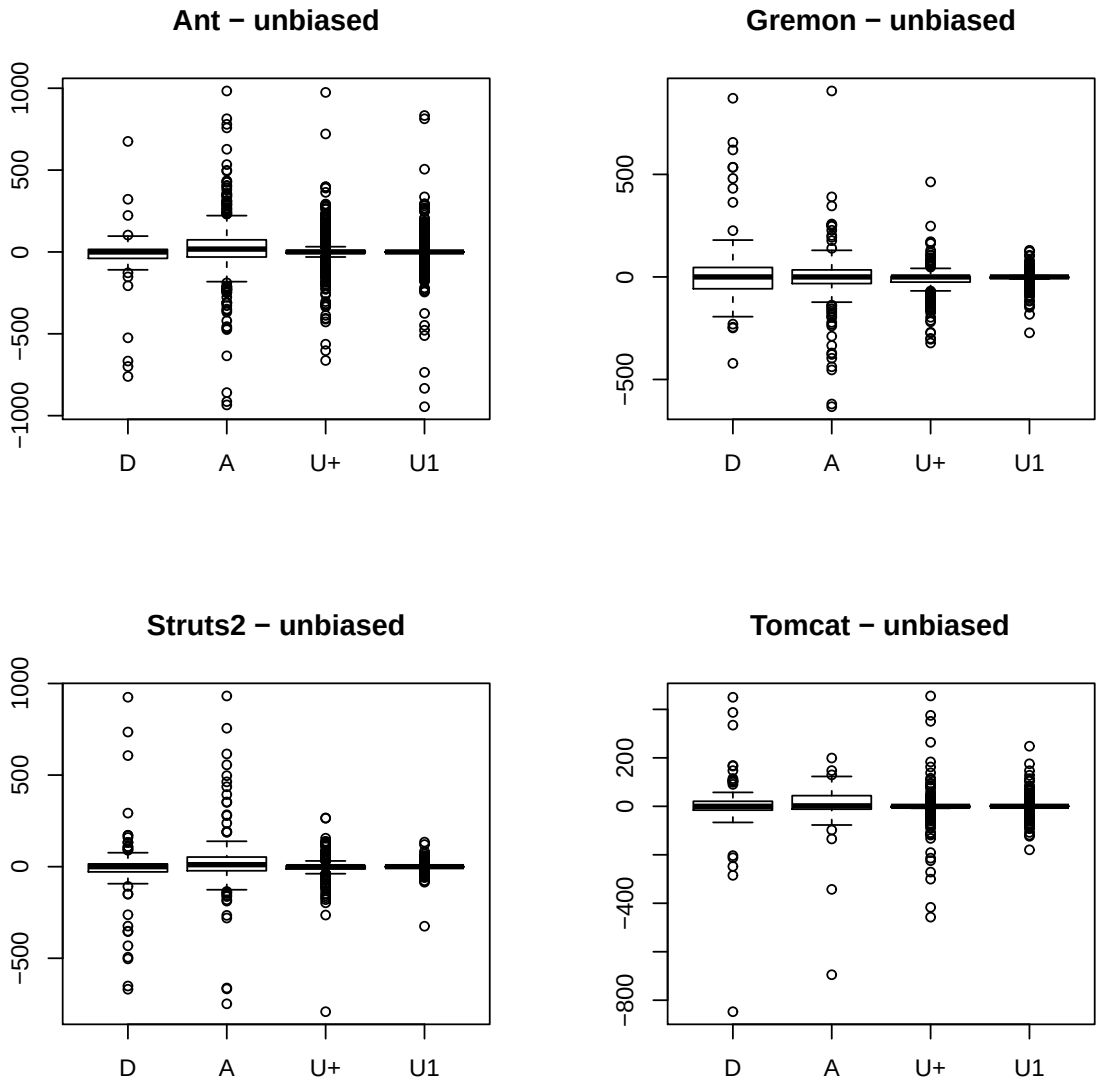

Figure 4.10. Plots with limited usefulness

the second element will be the sum of the first 2 elements, and so on. In the diagrams the x-coordinate represents the number of elements, and the y-coordinate represents the calculated cumulatives. Instead of drawing each point one by one, these points are connected with straight lines. If the number of elements is high enough, the result will look like a continuous line without bends.

The diagram type is mostly suitable for data of normal distribution with the mean close to 0. The diagram is applicable for quick comparison of several data sets: to illustrate the similarities and differences. It can be used to illustrate quickly two or more – seemingly similar – data sets if they are really similar or not. A CCD which contains two or more characteristics on the same diagram we call *Composite Cumulative Characteristic Diagram*.

We show examples later in this chapter in Figure 4.11.

Quantile Difference Diagram (QDD)

The idea behind the Quantile Difference Diagrams is to compare the same quantiles of two sets of numbers. This means the first element of the first set should be compared to the first element of the second one, similarly the 10% to the 10%, the median to the medial, the 90% to the 90%, highest to the highest and so on.

Therefore the input of the QDD is always two sets of numbers. Every centile is determined in both subsets, i.e. the 0% (which is the lowest one), the 1% (e.g. if the set contains 1000 elements, this is the 10^{th}) etc. This results 101 values in every case, either by omitting values, or taking the same values several times. Then the differences are calculated at every centile. On the the diagram these differences are displayed as a line. Examples for QDD can be found later in this chapter in Figure 4.12.

The `vudc` R Package

We implemented both diagram types as an R package [103], named `vudc` [38], which stand for *Visualization of Univariate Data for Comparison*. This can be installed as any other package, either directly from the R GUI, or by downloading from CRAN (`http://cran.r-project.org/web/packages/vudc/index.html`). After installation it should be loaded as a usual R package, as follows:

```
library(vudc)
```

The package contains two functions: `ccdplot()` and `qddplot()`; furthermore, data used in our research: `projectdata`. The user can obtain general information using R help command:

```
?vudc
```

`ccdplot()`

This function creates a Cumulative Characteristic Diagram. Figure 4.11 illustrates some examples.

The upper left graphics draws the *Cumulative Characteristic Diagram* of 100 random real numbers of standard normal distribution. This can be drawn with the following R function:

```
ccdplot(rnorm(100))
```

The upper right figure illustrates the *Composite Cumulative Characteristic Diagram*. This diagram contains characteristic diagram of two or more sets of numbers on the same scale, along with the (optional) CCD of the union of the numbers. The illustration contains numbers of normal distribution, with different size, different expected values and different variances. This can be created with help of R command

```
ccdplot(list(rnorm(400, 0.1, 1), rnorm(200, -0.1, 3)))
```

The differences in width, height and the right end are spectacular.

The lower left diagram illustrates that this diagram is sensible on the outliers. A mechanism is built in to remove the outliers automatically, either by providing an absolute threshold, or a percentage; the later one is applied on both ends. To reproduce a similar the diagram, use command

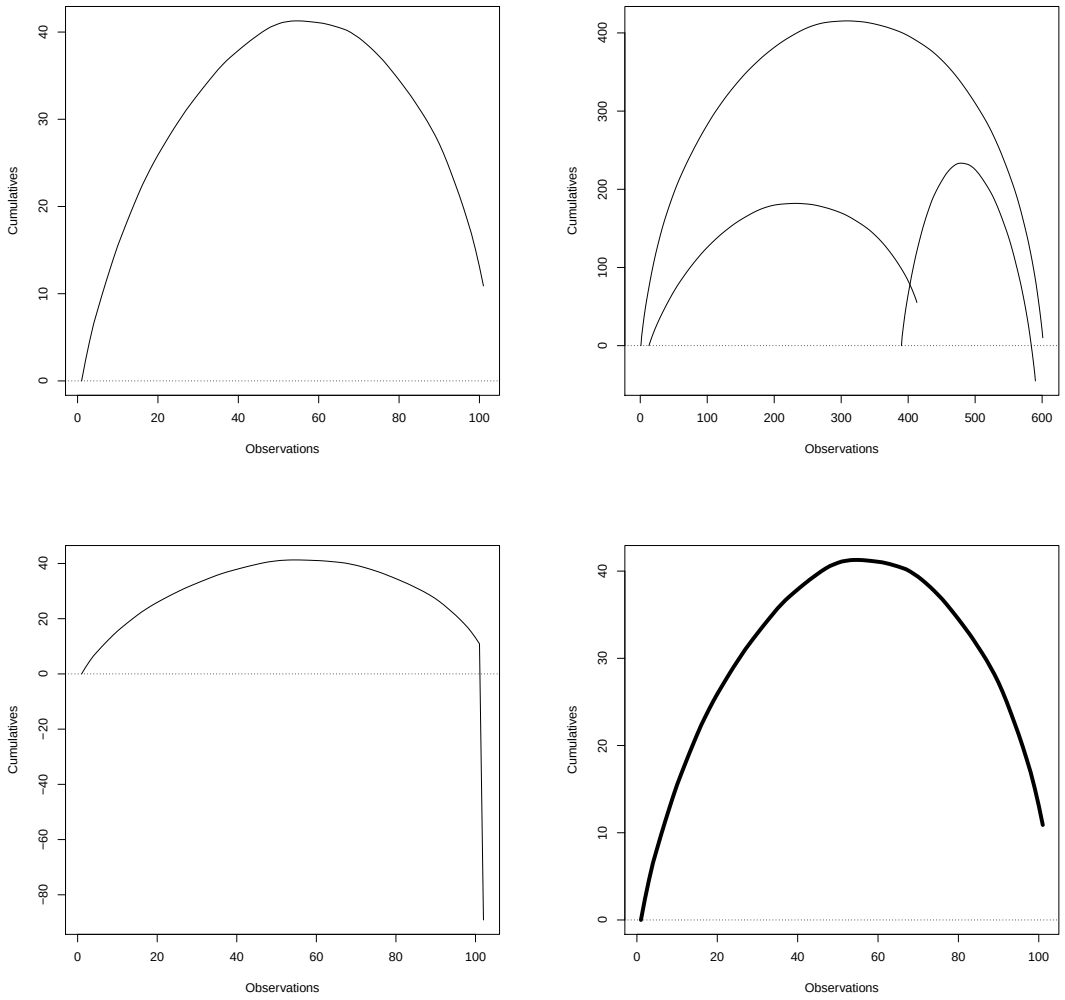

Figure 4.11. `ccdplot()` examples

```
ccdplot(c(rnorm(100), -100))
```

Finally, the lower right diagram illustrates that the function integrates into the standard R diagram functions, the standard parameters can be passed. The line is thick, which can be achieved with the following command:

```
ccdplot(rnorm(100), lwd=5)
```

Detailed information about all the possible parameters and further examples can be obtained using the R help command:

```
?ccdplot
```

`qddplot()`

This function creates Quantile Difference Diagrams. Figure 4.12 illustrates some examples.

The upper left diagram illustrates the comparison of two sets of random numbers of normal distribution, with different number of elements (100 vs. 200), different means

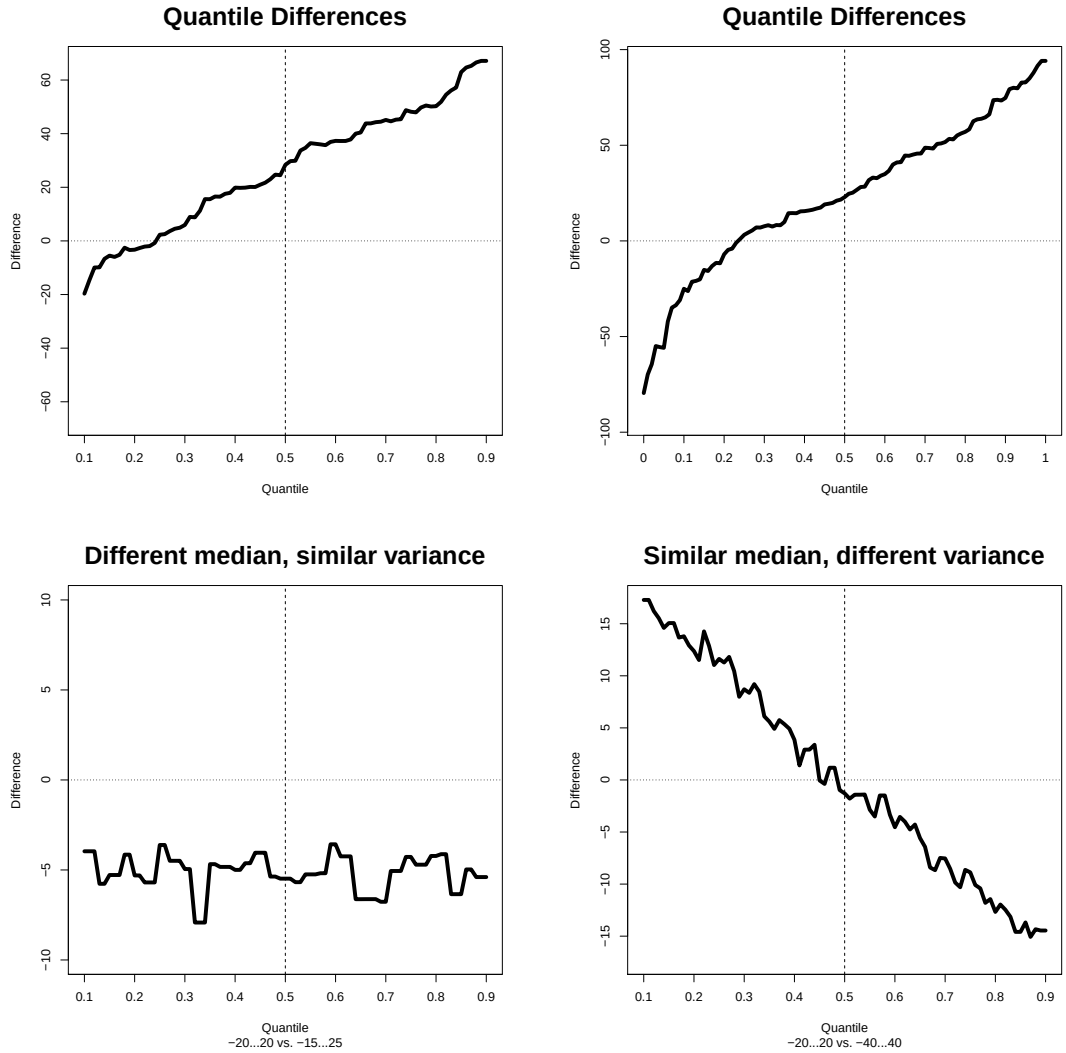

Figure 4.12. `qddplot()` examples

(30 vs. 10) and different standard deviation. Despite the fact that the number of elements in the second subset is twice as much as in the first one, the illustration was possible. The diagram illustrates that the numbers in the first subset are higher than those in the second: the territory above the abscissa (i.e. the x-coordinate) is higher than below it. On the other hand, it also illustrates that among the lowest elements the numbers in the second subset are higher than those in the first one. The diagram can be created using the following command:

```
qddplot(rnorm(100, 30, 50), rnorm(200, 10, 10))
```

The upper right diagram illustrates that the diagram is biased at both ends. This diagram is illustrated with numbers of the same distribution as above. By default, the diagram does not display the lower and the upper 5%. This can be fine-tuned using parameters. In this example the remove ratio is set to 0:

```
qddplot(rnorm(100, 30, 50), rnorm(200, 10, 10), remove.ratio=0.0)
```

The primary usage of this diagram is intended to illustrate the comparison of two sets of numbers of the same distribution and similar variance, but different expected

value. The first set contains 41 numbers, close to all the integers from -20 up to +20, and the second one similar numbers from -15 to +25. The difference is about 5, which is illustrated in the lower left diagram. The usage is intended to be converse: we have 2 sets of numbers, and the diagram reveals this property. The diagram was made using command

```
qddplot(seq(-20, 20) + rnorm(41), seq(-15, 25) + rnorm(41),
    main = "Different median, similar variance",
    sub = "-20...20 vs. -15...25")
```

The second most important usage of the diagram is intended to be the variance comparison. In this example the first set of numbers contain similar elements as above, and the second one contains 81 elements, around the integers from -40 up to +40. The comparison statistic tests, which compare the mean or median of the numbers would not show relevant deflection, however, the diagram, as shown in the lower right, indicates that there is a difference in variance. The medians are more or less the same (the difference is around 0 at the median), but in the ends the line is far from 0. Therefore such an illustration would indicate it is worth to compare the variances. The diagrams was created using the following command:

```
qddplot(seq(-20, 20) + rnorm(41), seq(-40, 40) + rnorm(81),
    main = "Similar median, different variance",
    sub = "-20...20 vs. -40...40")
```

Detailed information about all the possible parameters and further examples can be obtained using the R help command:

```
?qddplot
```

`projectdata`

The package contains information about the software systems described in Section 3.3.1: **Ant**, **Gremon**, **Struts 2** and **Tomcat**. In order to access the data first we need to issue the following command:

```
data(projectdata)
```

For each project a *data frame* is provided, containing information of every available commit. The rows of the data frame represent commits, and there are the following columns:

- `Revision`: the original revision number in the version control system
- `MaintainabilityDiff`: maintainability difference of the actual and the previous commit
- `A`: number of added Java files in the commit
- `U`: number of updated Java files in the commit
- `D`: number of deleted Java files in the commit
- `Churn`: a real number representing the code churn value of the commit

- `Ownership`: a real number representing the ownership of the commit

We removed the commits not containing Java files.

In order to be able to identify the commit, especially for the open source systems, we added the revision number to the data, exactly as it is located in the version control system.

The `MaintainabilityDiff` is the normalized difference of maintainability values of 2 subsequent revisions, as described earlier in this chapter. The final result is a real number.

The number of added, updated and deleted files are non negative integers, containing information about Java files (non Java files were removed).

The calculation of `Churn` and `Ownership` values are described later in this thesis, in Section 5.1.2.

This is an example excerpt of the data (information about the first 10 commits of project Ant):

```
> projectdata$Ant[1:10,]
   Revision MaintainabilityDiff  A U D    Churn Ownership
1    267549              0.00000 44 0 0     0.00  1.000000
2    267551            -14.55057  0 5 0  3960.60  2.000000
3    267554              0.00000  0 1 0  5706.00  2.000000
4    267557           -524.46238  0 2 1 12281.33  2.000000
5    267558            -19.55645  1 1 1  8343.00  1.587401
6    267559           -184.04878  0 3 0 11837.00  2.000000
7    267560            -15.25897  0 3 0 12300.67  3.000000
8    267561            -56.05360  0 1 0  4168.00  2.000000
9    267562             16.39003  0 2 0  5014.50  2.449490
10   267567            -71.82581  0 0 6     0.00  1.000000
```

The user can obtain detailed information using the help page of the project data, using R command

```
?projectdata
```

4.3.3 Illustrating the Statistic Tests

In this section we provide some examples about the usage of the defined diagrams, illustrating various statistic tests.

For the illustration, first we generate sets of numbers. Both sets are of normal distribution, containing 101 elements each. The first subset's (`x` in the example) mean is 1, and the standard deviation is also 1, and the second subset's (`y` in the example) mean is -1, and the standard deviation is 3. For the data generation first we set the random seed in order to be able to reproduce the results.

```
set.seed(1)
x <- rnorm(101, 1, 1)
y <- rnorm(101, -1, 3)
```

In this example we act as we just received these sets of numbers, and we do not know anything about them. First we generate the Cumulative Characteristic Diagram and the Quantile Difference Diagram, and then begin with the analysis. The diagram generation is performed with the following commands:

```
ccdplot(list(x, y))
qddplot(x, y)
```

We display the results in Figure 4.13.

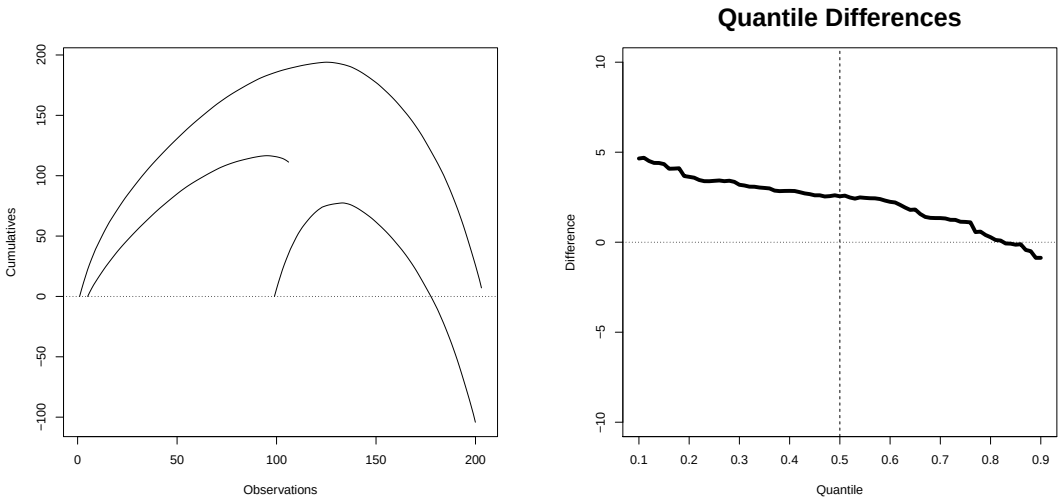

Figure 4.13. Examples for statistic test demonstrations

Wilcoxon Rank Correlation Tests and CCD

First let us check the Cumulative Characteristic Diagram (the left diagram of Figure 4.13). Based on the difference of the altitude of the right end of the characteristic lines (the first one is far above 0, and the second one is far below 0) it indicates that it is likely that the elements in the first subset are significantly higher than those in the second. To check this, we perform a one tailed Wilcoxon rank correlation test. This test compares each elements of the first subset with each in the second one.

This is the result of the Wilcoxon test:

```
> wilcox.test(x, y, alternative="greater")

        Wilcoxon rank sum test with continuity correction

data:  x and y
W = 7843, p-value = 2.046e-11
alternative hypothesis: true location shift is greater than 0
```

The preliminary assumption based on the CCD turned to be correct: the p-value is very low. Conversely, having a good result of Wilcoxon test, we can illustrate it with CCD.

Wilcoxon Rank Correlation Tests and QDD

The results so far indicates that the numbers in the first subset are greater than those in the second one. But can we tell more about them? To answer the question, consider the QDD (the right diagram of Figure 4.13).

The result of the Wilcoxon test on this diagram means the following: the signed territory between the line and the abscissa (i.e. the x-coordinate) is positive. However, on the right side the line is below 0, meaning that just considering the highest values, those in the second data set are higher than in the first one. In concrete cases this worth further analysis. Without QDD, this attribute could have been bypassed.

What does it mean in practice? Let the numbers denote the knowledge of students in mathematics in different countries. It can be higher in country A compared to country B in general, but the best students in country B might be better than in country A. On the Mathematics Olympics country B is likely to gain better results over country A. On the contrary: having a better results on the Olympics does not necessarily mean that the education is on the good way.

Variance Tests and CCD

Considering the CCD again (the left diagram of Figure 4.13) there is another spectacular difference between the left and the right curve to note: their width are the same, but the vertical lengths of the lines are different: the right hand side is much longer than the left hand side. This indicates differences in variance.

Now we perform the variance test.

```
> var.test(x, y, alternative="less")

        F test to compare two variances

data:  x and y
F = 0.095434, num df = 100, denom df = 100, p-value < 2.2e-16
alternative hypothesis: true ratio of variances is less than 1
95 percent confidence interval:
 0.0000000 0.1328177
sample estimates:
ratio of variances
        0.09543427
```

It turns out that the difference (indeed: the ratio) between the variance of the two subsets are really significant with extremely low p-value (meaning: it is very unlikely that this happened by chance).

It was not a big surprise for us as we generated the values to have different variances; however if we act we do not know anything about the nature of the input data, this could be helpful. In our study such a diagram helped us to perform analysis in this direction, and we presented the result in article [36].

Variance Tests and QDD

How does the difference in variance look like on the QDD?

If the line on the QDD is more or less horizontal, it indicates that there is no real difference in variance. On the other hand, if it has a slope, it is a sign of difference in variance. Considering the right diagram of Figure 4.13, we conclude that the line has a slope, indicating the probable significant difference of variances.

Contingency Chi-Squared Tests and CCD

In the basic case of Chi-Squared tests we have a null hypothesis about the number of elements of some subsets, and real observations. For example, consider the genres of students in a university. The null hypothesis is that 50% are male and 50% are female. In the fictive example of a Technical University there are 257 students, among them 243 boys and 14 girls. With the Chi-Squared test we can check if the difference is casual, or we should reject the null hypothesis, and state an alternative one, that in the technical universities there are more males then females.

In a more general case we have a matrix of any dimension. Every observation belongs to exactly one cell in the matrix. The null hypothesis is that the observations are distributed evenly in the matrix. It does not exactly mean that the number of elements are the same in each cell, but it is calculated based on the row and the column sums.

In our example we consider a matrix of dimensions 2x2, containing the number of positive and negative elements in both subsets. The following listing contains how we created it, and then we display the values. Then we perform the Chi-Squared test, and display the expected values, the global result of the test and the standard residuals on each cells. The meaning of the standard residual of a cell in nutshell is the following: what was the difference between the expected and the actual value if it was a number of standard normal distribution. Based on this value, p-values can be calculated for each cell.

```
> sign <- matrix(c(length(x[x>0]), length(x[x<0]),
                   length(y[y>0]), length(y[y<0])),
    2, 2, dimnames=list(c("positive", "negative"), c("x", "y")))
> sign
          x  y
positive 90 34
negative 11 67
> chisq.test.result <- chisq.test(sign)
> chisq.test.result$expected
          x  y
positive 62 62
negative 39 39
> chisq.test.result

        Pearson's Chi-squared test with Yates' continuity correction

data:  sign
X-squared = 63.177, df = 1, p-value = 1.889e-15

> chisq.test.result$stdres
                  x         y
positive  8.092926 -8.092926
negative -8.092926  8.092926
>
```

The result of the Chi Squared test indicates that the number of positive and negative elements in sets `x` and `y` is significant. This is indeed not surprising for us, as we know the nature of numbers in the sets. But in general this is not known.

The connection between the result of the Chi-Squared test and CCD is the following: if the shapes of the curves does not resemble to each other, with proper division it is likely that Chi-Squared test will show significant deflection from the null hypothesis (which in terms of CCD it means the shapes of the curves are similar).

In practice the Chi-Squared test is suggested to be executed specially in the following case: consider several observations (e.g. technical universities of different cities), and the curves on the CCD diagrams are different, but the CCD diagrams themselves are similar.

4.3.4 Illustrating the Results

In this section we illustrate the usage of the diagrams using the data of this Section.

Connection Between Version Control Operations and Quality Change of the Source Code

Let us consider data and results described in Section 4.1.

In general, the outliers have significant bias on the diagrams. Some unusual commits, like merging a whole branch to the trunk, or renaming files in 2 major steps by accident (first remove, and then in another commit add again) results in huge outliers. We remove the effect of these extraordinary commits by removing the huge values. Without these values we receive diagrams presented in Figure 4.14.

The curves within diagrams are obviously different, and there are similarities between the diagrams. We performed Contingency Chi-Squared test to check these differences between curves, and the results are convincing. In Figure 4.3 we illustrated the results with help of bar plot diagrams; Figure 4.14 illustrates the same results using Composite Cumulative Characteristic Diagrams.

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

Now consider the results of value comparison, as presented in Section 4.2.

Here we present the following check: we divided all the commits into two based on the existence of operation Add. The first subset contains commits where at least one new source file was added, and the second one contains the remaining. We considered the related maintainability changes, performed the Wilcox test on the mentioned sets of numbers, and concluded that the maintainability change of commits containing file addition is significantly higher than those of not containing file addition.

In Table 4.17 we presented the results in tabular format, now it is time to visualize the results.

We present the results in Figure 4.15. The right end of the curves for commits containing add are located spectacularly higher than these of the other curve. The difference was shwon to be significant using Wilcox test.

However, Quantile Difference Diagrams revealed some more important details.

We present the resulting diagrams in Figure 4.16. The territory above abscissa is spectacularly higher in case of Ant, Struts 2 and Tomcat. The Wilcox test resulted that it is true also in case of Gremon, but with weaker significance.

Indeed, the diagrams support the findings based just on the Wilcox test, however, it forevisioned the differences in variance.

Variance of Source Code Quality Change Caused by Version Control Operations

One of the results is that commits containing file additions have significantly higher maintainability change compared to those of not containing file additions at all. This result we can imagine as follows: this statement is true in all the magnitudes of maintainability changes. However, based on the diagrams in Figure 4.15 and especially in Figure 4.16, this is not true.

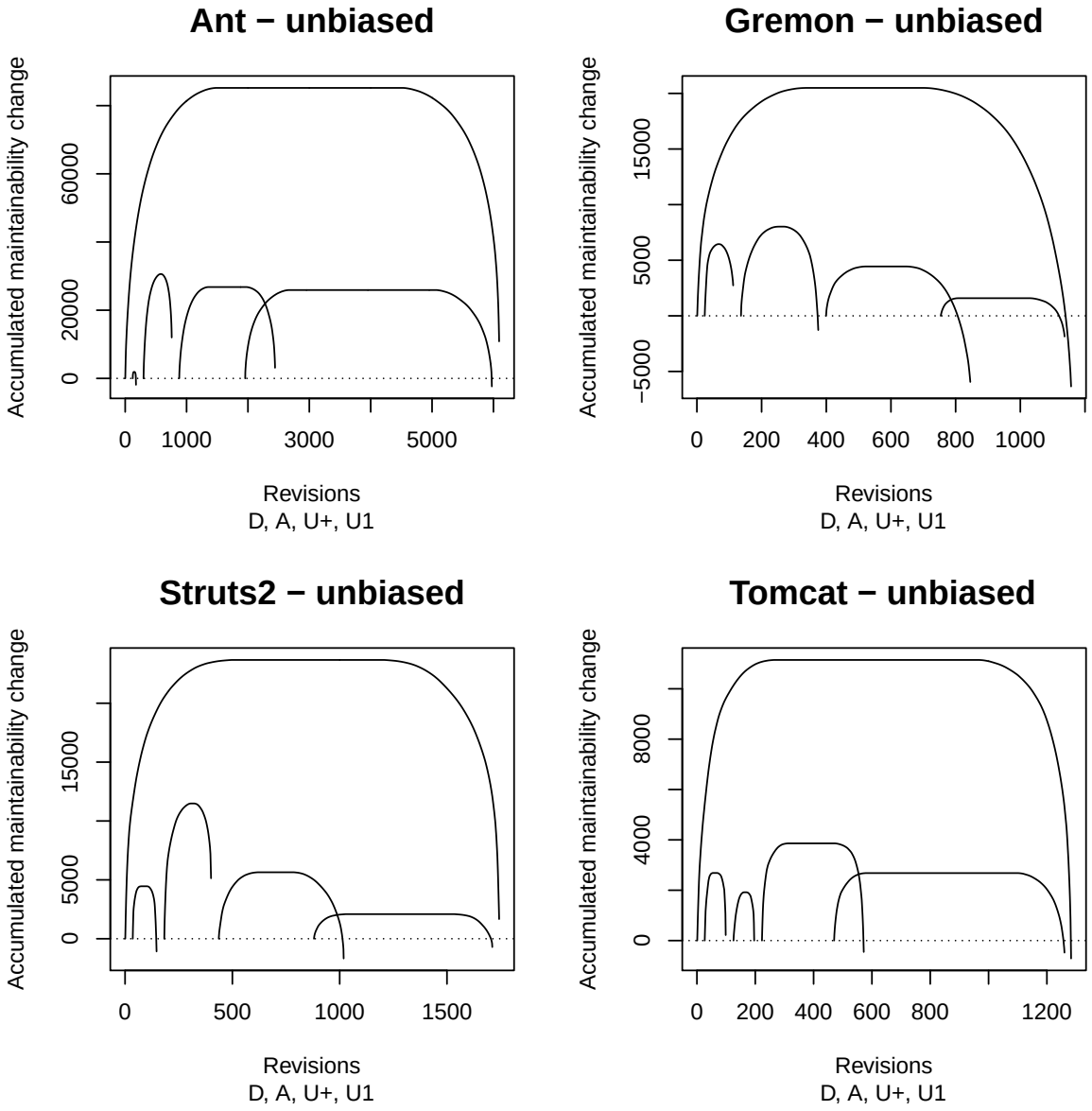

Figure 4.14. Composite cumulative characteristic diagrams about maintainability

In case of low values (high maintainability decreases) the values are much lower in case of commits containing file additions on the same quantile, compared to the commits not containing file additions.

The most important thing to see is it is not true – as one would expect just based on the preliminary test results – that the maintainability change values are higher in case of all magnitude of values. This means file additions really do their bits of the code erosion, and if it erodes, its erosion is much higher than the others. Without the QDD, just using the results of Wilcox test, this would not have been revealed.

We decided to analyze this phenomena further, which lead us the variance related results, as presented in Table 4.18 (for detailed explanation see the appropriate parts of the Section 4.2).

Based on the CCD the difference in variance is indicated by the following: the ratio of the horizontal width (i.e. the number of observations) and the vertical width are different. This is especially apparent in Figure 4.15 at project Gremon: the heights of the 2 lower curves are similar, but the width of the left one is spectacularly lower than the width of the right one.

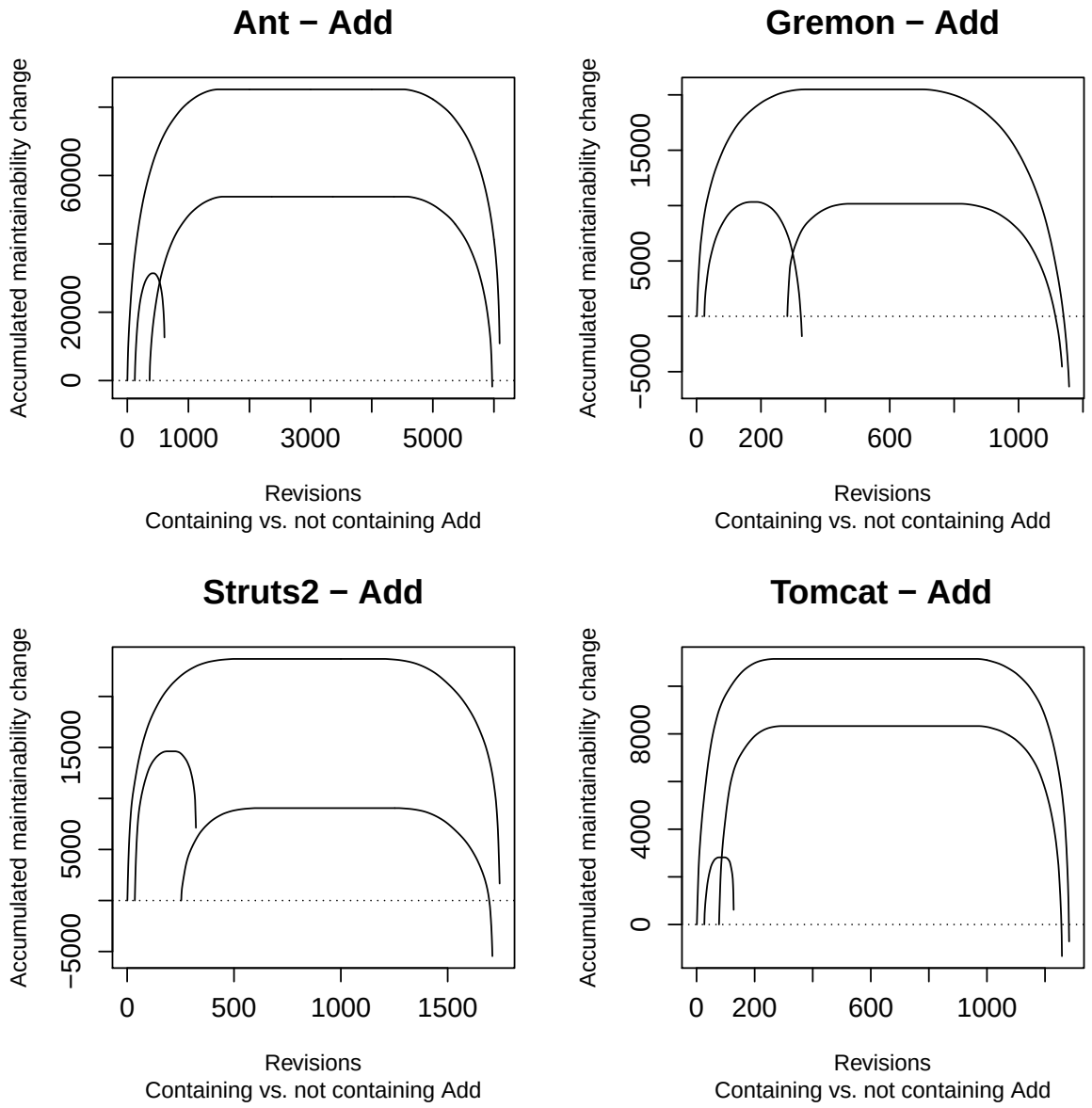

Figure 4.15. CCD: maintainability changes of commits with and without file additions

Considering the QDD (Figure 4.16) it is apparent in all the 4 cases that the lines have significant slopes, and their main shapes are not horizontal.

4.4 Summary

In this chapter we presented an analysis of the impact of version control operations on maintainability.

In Section 4.1 we showed the existence between version control operation and maintainability change of the source code. For every analyzed systems we considered all the available commits on the main branch, then we classified first on maintainability change bases, and then on number of operation basis. This resulted a matrix; each cell containing the number of commits confirming the both classification. Then we performed the Contingency Chi-Squared test to check if there are significant differences in the table between the actual and the expected values. The test resulted similar significant deviance for all the analyzed systems for

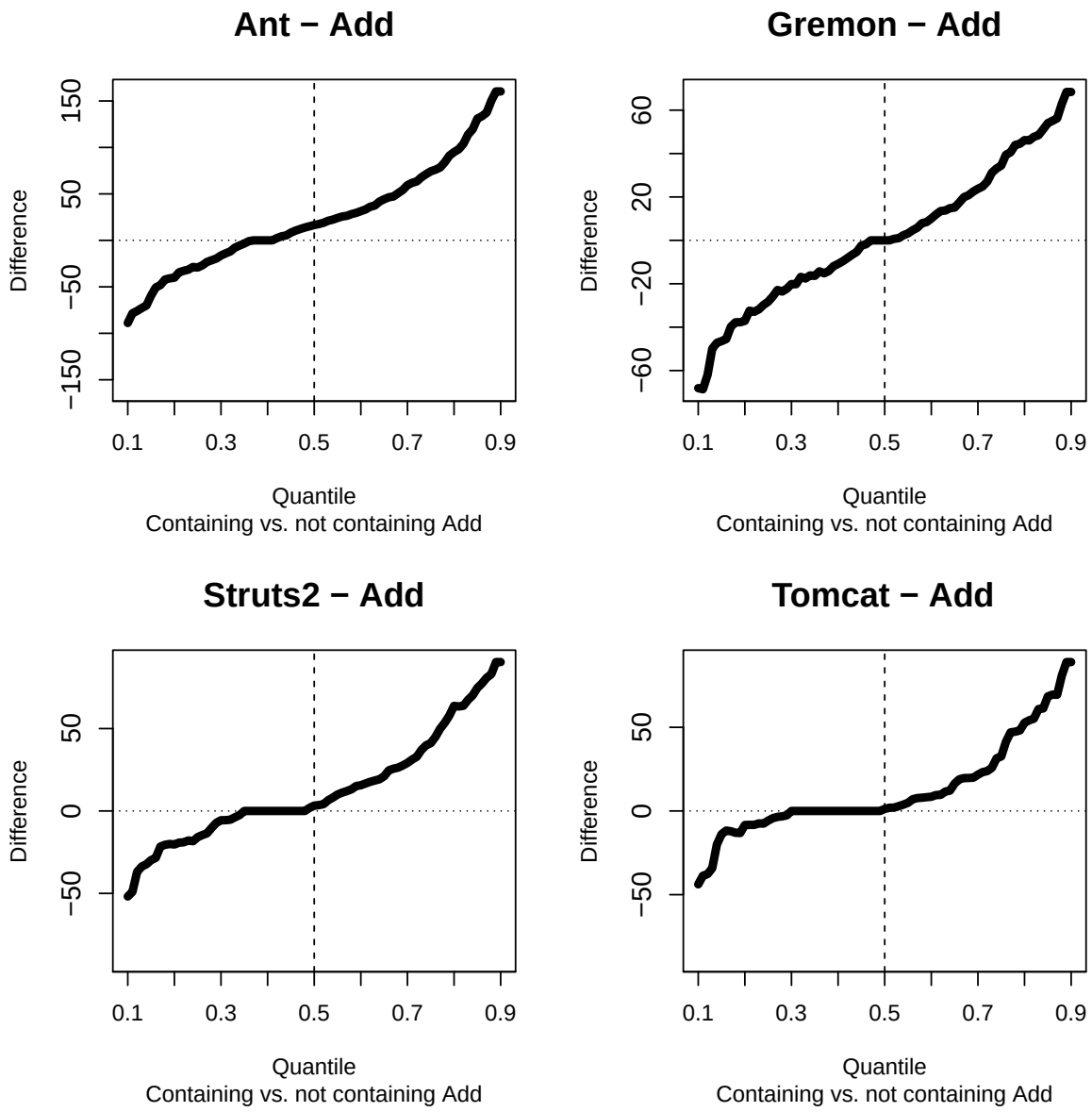

Figure 4.16. QDD: maintainability changes of commits with and without file additions

most of the cases, therefore we concluded that there was a connection between version control operations and maintainability change of the source code.

In Section 4.2 we analyzed the file addition, file update and file deletion version control operations, and checked their impact on the maintainability. We selected two subsets of all the commits on version control operation basis; we performed this 7 ways for every operations, for each analyzed systems. Then we took the related maintainability change values, and compared them considering the values and the variance of the values as well. For comparing the values we performed the Wilcoxon-test, and for variance comparison we performed the T-test. The tests revealed that file addition had better effect on maintainability than file update. We did not find a clear connection between file delete and maintainability. On the variance check the tests resulted that add and delete increased, while update decreased the maintainability.

The findings are thorough; however, potentials are still left for the usage of the results. The results could be used by a code review tool, which could automatically label each commit, indicating if it is "dangerous" or not. That information can be used by the architects of that

project, indicating the necessity of a thorough code review. If there are dozens of commits on that project every day, such an automation could be very useful.

In Section 4.3 we presented two diagrams which turned to be suitable for visualize the results of the used statistic tests.

The input of the *Cumulative Characteristic Diagram* is a set of number. We sort them non-ascendant, and for every index we calculate the sum from the first one up to that point. The indexes represent the x-coordinate, and the sums represent the y-coordinate. The characteristic is created by connecting these points with lines.

Using the *Quantile Difference Diagram* we can compare two sets of numbers. First we sort the elements of both sets in non-descending order. Then we take the values of every centile from both sets, pairwise. We calculate the difference of every pair. Using the centiles as the x-coordinate and the differences as y-coordinate, finally connecting the points we gain the Quantile Difference Diagram.

4.5 Contributions

The author contributed to the new results presented in this chapter as follows:

- The methodology of finding the connection between version control operations and maintainability, as presented in Section 4.1, along the implementation, the execution and the evaluation of the results. This methodology includes the following: the idea of categorization the commit on version control operation basis, using PCA; application of the Contingency Chi-Squared test on *maintainability change x version control operation based commit category* matrix. The evaluation of the results include the idea of interpretation of the results using the exponents, visualized by bar plot diagrams.

- The methodology of examining the impact of version control operation on the maintainability, as we described in Section 4.2, along with the implementation, the execution and the evaluation of the results. The methodology includes the 7 divisions of commits on version control operation basis, and application of the Wilcoxon-test and the F-test on these subdivisions. The evaluation of the results include the idea of calculating with the standard residuals, and visualization them with bar plot diagrams.

- The idea of the Cumulative Characteristic Diagram and Quantile Difference Diagram, as we described in Section 4.3.2, implementation in R, maintenance of the `vudc` R package.

- All of the helper diagrams in the chapter.

"One of my most productive days was throwing away 1000 lines of code."

— Ken Thompson

5 Thesis Point 3: Connection between Version Control History Metrics and Maintainability

Up to now we considered the number of each version control operation only, and we treated the commits independent from each other. In this section we reveal some connection between version control history related metrics and maintainability. First, in Section 5.1 (**Thesis Point 3.A**) we check how the intensity of past code modification and the level of the code ownership affects the future maintainability change. Then in Section 5.2 (**Thesis Point 3.B**) we present 6 version control history metrics, along with their connection with maintainability and post-release bugs.

5.1 Thesis Point 3.A: Impact of Code Modifications and Code Ownership on Maintainability

5.1.1 Overview

We checked the connection between maintainability change and the following 2 version control history metrics: cumulative code churn and code ownership. We performed the analysis on commit basis. We collected historical data form SVN version control system and estimated the maintainability with the help of ColumbusQM probabilistic software quality model, as we described in Section 3.2.

Formally, we investigated the following research questions:

3.A.RQ1: *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?*

3.A.RQ2: *Does the number of developers modifying the same code in the past have any affect on the maintainability change of future commits?*

The null-hypothesis was the following: the past does not have any influence on the future;

the future maintainability time line of the source code is independent of the past modifications.

The assumed alternative hypotheses were the following. 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. Modifying files without clear ownership (i.e. those of which have been modified by several different developers in the past) is more likely to result in further maintainability decrease than modifying files with clear ownership (i.e. those modified by only one or by a very few number of developers).

5.1.2 Methodology

We used the same maintainability data as in Section 4. The calculation is described in Section 3.2. For the statistic performed tests we needed the *sign* of the change of maintainability of the subsequent revisions.

In this section we present how we calculated the code churn and the code ownership data. Then we argue that these and the maintainability data are independent, and then present the statistic tests we performed.

Calculating Cumulative Code Churn

In this section we present how we calculated the cumulative code churn values for the revisions. First, we define the cumulative code churn of a file, then we define the churn value of a commit.

According to the literature [81], code churn is defined as follows: lines added, modified or deleted in a file from one version to another. We used 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]

We considered the cardinality of line changes (both line additions and line deletions). We summed these values from the very beginning of the available version control history; this value formed the cumulative code churn of a file. As a result, we obtained how many lines had been added to the source code plus how many lines had been 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 contained at least one Java source file). We somehow needed to define the cumulative churn value of the commit itself.

First of all, during calculation we considered the value *before* the actual commit, i.e. not considering the actual 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 was the

[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

common impact of all the modifications of all the affected files of that commit, i.e. the final change was 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 5.1 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.

		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	

Table 5.1. Example file modifications

For this case, Table 5.2 illustrates the calculated churn values for every file and every revision. They are initialized to 0 (representing the "0^{th}" 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.

		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	

Table 5.2. File churn values example

The commit related churn values are calculated based on the file related churn values. These values are listed in Table 5.3. It contains which files are 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 perform.

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

Table 5.3. Commit churn values example

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

Calculating Code Ownership

We used the following method to express the code ownership numerically. In a particular commit, we considered all the affected source files one by one. As indicated in Section 4.1.2 there was at least one Java file in every analyzed commits. For every source file, we calculated how many different developers committed on that file at least once from the beginning of the available history, including the actual commit as well. Therefore this value was at least 1.

At this point we had a positive integer number for every affected source files of the commit in question. But for further analysis we needed a value describing the ownership of the actual commit. For this we chose to calculate the geometric mean of the collected values for files. This expressed well the overall ownership of the file based actual ownership values.

For example, consider a small artificial project with 4 java files: `A.java`, `B.java`, `C.java` and `D.java`. This project have been developed by the following developers: `sulley`, `mike`, `randall` and `celia`. In Table 5.4 the rows represent commits. The first column contains the revision number, while the second one contains the author of that commit. Then the odd columns indicate if the actual file was affected by the commit in question, and the even columns contain the number of different developers of that file up to the actual commit. The last column contains the calculated ownership value.

Rev.	Author	A		B		C		D		Own.
1	`sulley`	+	1	+	1					1.00
2	`mike`	+	2			+	1			1.41
3	`sulley`	+	2	+	1					1.41
4	`randall`							+	1	1.00
5	`celia`	+	3					+	2	2.45
6	`randall`	+	4					+	2	2.83
7	`sulley`	+	4	+	1					2.00
8	`mike`	+	4							4.00

Table 5.4. Ownership related example

In the first revision `sulley` added files `A.java` and `B.java`. The ownership values are initialized to 1 for both of the files, and the geometric mean of 1 and 1 is 1.

In the second revision `mike` modified file `A.java` and added file `C.java`. At this point the ownership value of source file `A.java` has been increased to 2 (`sulley` and `mike`), and the value of file `B.java` was initiated to 1. The ownership value of the second revision is $\sqrt{1 \cdot 2} \approx 1.41$.

With these 2 examples the other 6 revisions are easy to understand.

From this scenario we can see the following:

- File `A.java` is the "hot spot" of the "project", modified by every developer.
- File `B.java` is an example of intensive modification by a certain developer (`sulley` in this case), therefore having a clean ownership.
- File `C.java` is an example of adding a source file once and never modifying later.
- File `D.java` is an example of a common code of two developers (`randall` and `celia` in this case).

As a result of the above described method, we get an ownership value for every revision.

Independence of the Values

At this point we had a maintainability change sign, a cumulative code churn value and an ownership value for each commit. Before going further to the statistical tests performed on these data, we argue that the maintainability change sign and the other two values were totally independent.

We calculated the maintainability value of the system solely from the source code; we did not consider version control data. We calculated the maintainability change for the n^{th} revision as the sign of difference of maintainability values of the n^{th} and the $(n-1)^{th}$ revisions, which we measured by utilizing source code metrics. Therefore its value was solely affected by the code change between the previous and the actual revision. We took into account neither the code history nor the author.

The modifications of the 1^{st}, 2^{nd}, 3^{rd}, ..., $(n-1)^{th}$ revisions affected the cumulative code churn and the ownership values of the n^{th} commit. In case of cumulative code churn calculation the last piece of information in the calculation we (potentially) considered part of the code delta between the $n\text{-}2^{th}$ and $n\text{-}1^{th}$ revision (only if both commits affected the same source files). We calculated the ownership value solely from version control historical data, particularly we considered the author of the earlier and the actual commits.

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

Statistic Tests

At this point we had the following 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, and another number illustrating the code ownership of the source files in that commit.

We wanted to tell something about their connection, independently for cumulative code churn and code ownership. For that, in both cases we divided the commits into two subsets: commits with positive maintainability change and those of negative maintainability change. From this point on we did not consider commits with zero maintainability change anymore. The zero maintainability changes are typically caused by small, one line modifications.

At this point we had 2 subsets of cumulative code churn values, and 2 subsets of code ownership values. We performed the comparison of the numbers belonging to these subsets, independent from each other. This means we performed the comparison of cumulative code churn values related to positive maintainability change (i.e. code quality increase) and those related to negative maintainability change (i.e. code quality decrease). We performed the same comparison for code ownership values, independent from cumulative code churn values.

The null hypothesis in both cases was that there was no significant difference between these values.

In case of cumulative code churn comparison the alternative hypothesis was that the cumulative churn values related to positive maintainability changes were significantly lower than those related to negative maintainability changes. In case of code ownership comparison, the alternative hypothesis was that the ownership values related to commits with positive maintainability changes were significantly lower than those related to negative maintainability changes.

In order to verify this, we used the Wilcoxon rank test. 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 have been problematic in our case. If a statistical test

requires a special distribution, that is normal distribution in most of the cases. But neither the defined cumulative churn values nor code ownership values were of normal distribution, but rather similar to an exponential distribution.

The other problem – especially in case of cumulative code churn – was 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 have caused 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 were greater than values in subset B), therefore we selected the one-tailed test. By using this approach, we were 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 [103] statistical software package.

We interpreted the p-values as follows:

- below 0.01: very strong significance,
- between 0.01 and 0.05: strong significance,
- between 0.05 and 0.1: significant,
- between 0.1 and 0.5: not significant,
- between 0.5 and 0.9: contradiction, and
- above 0.9: the opposite statement is true.

For the illustration of Wilcoxon test execution consider our running example in case of code ownership (the example for cumulative code churn would be very similar). Table 5.5 shows how the maintainability changed in each revision. For a better overview we repeat the ownership values in this table as well.

Revision	Ownership	Maintainability change
1	1.00	positive
2	1.41	neutral
3	1.41	negative
4	1.00	positive
5	2.45	negative
6	2.83	positive
7	2.00	neutral
8	4.00	negative

Table 5.5. Example maintainability changes for ownership

Now we have the following ownership value sets:

- Ownership values related to positive maintainability changes: $\{1.00, 1.00, 2.83\}$

- Ownership values related to negative maintainability changes: $\{1.41, 2.45, 4.00\}$

Considering all the comparison combinations (there are $3*3 = 9$ cases) we get the following. In 7 cases (the two 1.00 in all comparisons and comparing 2.83 with 4.00) the elements in the first data set are less than the elements in the second one, and in 2 cases (comparing 2.83 from the first data set with 1.41 and 2.45 from the second one) the result of the comparison is just the opposite. The p-value in this example is about 0.19, indicating that the elements in the first subset is less, but not significantly, than the elements in the second set. The obvious reason for this is the small number of observation.

Another example, resulting a p-value of 0.028 (i.e. significant), is comparing the following two subsets with the alternative of less: $\{1, 3, 5, 7, 9\}$ and $\{6, 8, 10, 12, 14\}$. In this example, the values in the first subset are spectacularly smaller than those in the second one, which is supported by the mentioned p-value.

5.1.3 Results

First we present a few code ownership related diagrams to highlight some details about the contributions. Then we present the results of the comparisons, finally we formally answer the research questions.

Code Ownership Related Diagrams

To provide an overview about some interesting aspects of the analyzed systems we present a few diagrams.

First let us consider Figure 5.1. The small empty circles on this strip chart represent the commits in a system. On the y-coordinate we list the developers in a decreasing order according to their number of contributions. The topmost developer is always the one with the largest contribution. We display the user IDs of the developers on the left of the diagrams. In case of Gremon – as it is an industrial project – the real user IDs are masked. On the right of the diagrams we display the portions of the total contributions. The x-coordinate represents the revisions of a system.

The black lines are actually several empty circles over one another; those are the periods when the developer in question was the most active.

Figure 5.2 illustrates the number of commits per author in a descending order. In case of Tomcat more then 80% of the commits were committed by a single developer. In projects Ant and Gremon there is again a clear single main developer who performed about 40% of the total commits. On the other hand, in case of Struts 2, it seems that there are 3 main contributors with a more or less similar impact on the project. From the strip chart it seems that 2 of them were main developers mainly in parallel, and then the third one took over the responsibility.

Figure 5.3 shows how many files a developer committed at least once. For example, if the small black circle above a developer is at the height of 100, it means that the developer committed in 100 different files. In case of Gremon and Tomcat the domination of the main developer is obvious from this diagram as well, but in case of Ant and Struts 2 it seems that several developers had a contribution affecting a large amount of files.

Figure 5.4 illustrates the number of different developers per files, which can be thought as a kind of ownership. This is the inverse of Figure 5.3, namely how many different developers committed to a single file. The black circles seem to be lines here as there are many files with the same values. If the lines at the lower values are longer, that indicates clearer separation of responsibility. Higher values indicate the hot spots: these are the files that were modified by several developers.

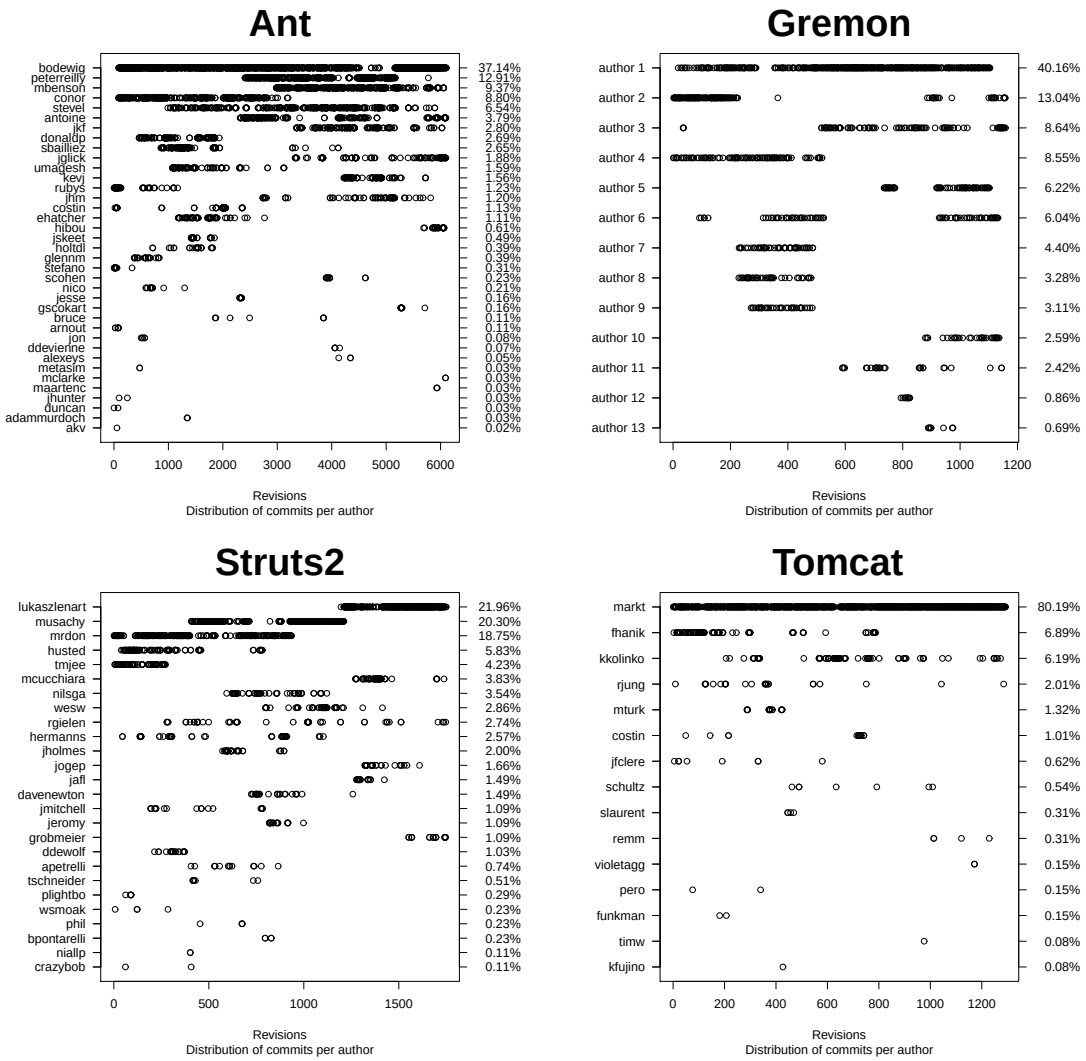

Figure 5.1. Commits per authors

Figure 5.5 is similar to the first one, but it contains a relaxation: we did not count the commits of a developer if the number of contributions of that developer on the source file in question was at most 1. We applied this rule because we wanted to eliminate the possible bias caused by a directory rename or a branch merge for example, which affected several source files by the contributor without real modifications of the source code. On these diagrams lines at 0 also appear, e.g. containing those sources which have been added once but never modified.

It is spectacular that the separation of responsibility is the best – based on the earlier statistics not surprisingly – in case of Tomcat. The separation of responsibility in case of Struts 2 and Gremon seems sufficient, but in case of Ant it is spectacularly bad. As the number of commits in this project is higher than the total number of commits in the rest of the 3 projects all together, we checked if such mess in separation of responsibility was caused just because of the long revision history, or this was a true tendency. We took the first 2,000 commits and found that the same lack of responsibility separation existed even considering similar magnitude of commits as in the other 3 cases.

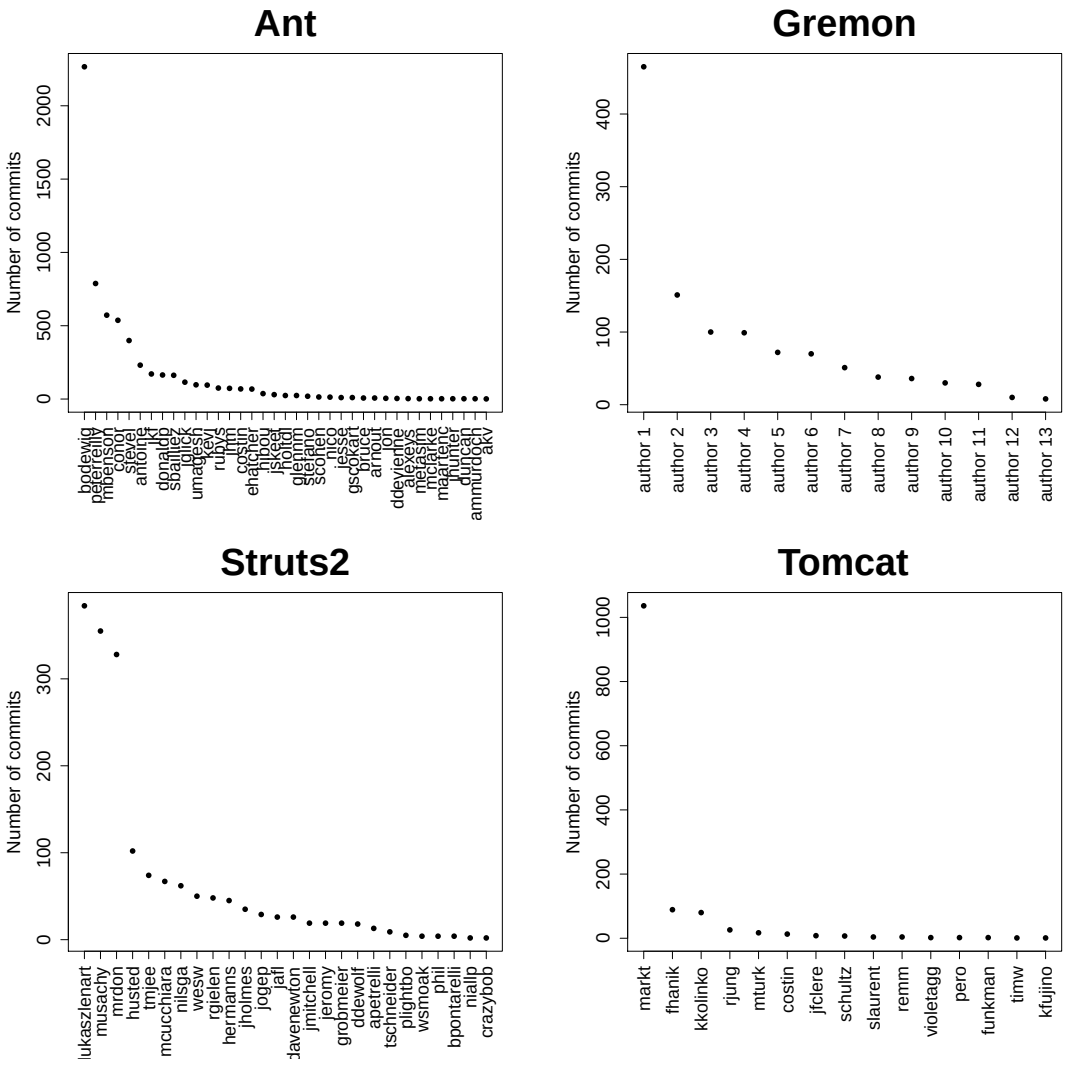

Figure 5.2. Commits per author

Results of the Comparisons

Table 5.6 shows the results of the Wilcoxon rank tests.

	Cumulative Code Churn		Code Ownership	
System	**p-value**	**Significance**	**p-value**	**Significance**
Ant	0.00235	very strong	0.03347	strong
Gremon	0.00436	very strong	0.05960	significant
Struts 2	0.00018	very strong	0.00001	very strong
Tomcat	0.03616	strong	0.21384	not significant

Table 5.6. Results of the cumulative code churn and code ownership tests

For cumulative code churn comparisons 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.

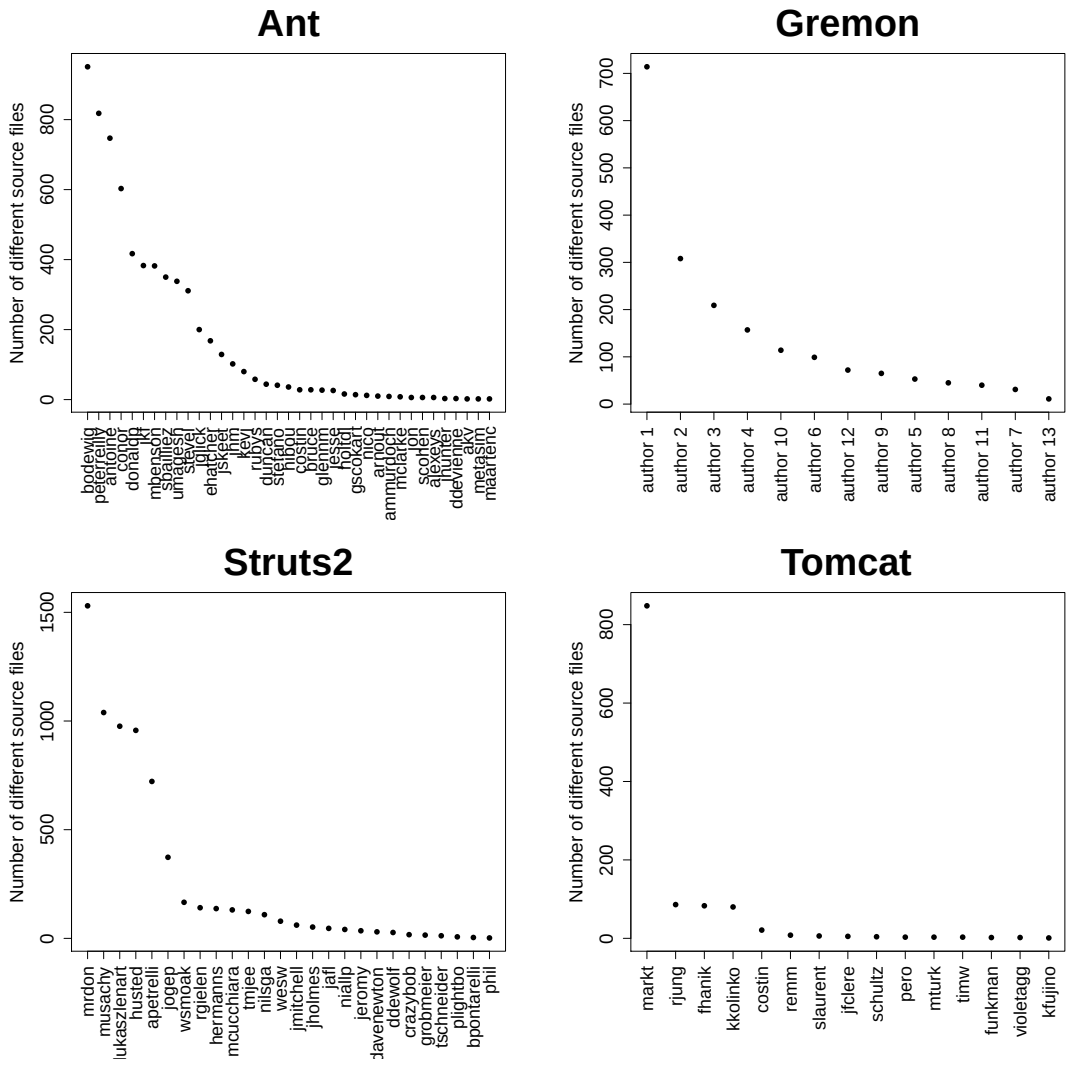

Figure 5.3. Files per author

For code ownership comparisons the picture is more hectic.

The result for project Ant is solid, above 0.01 but below 0.05. Indeed, the result for the first 2,000 commits is 0.002897, which is even below 0.01. It seems that the results weaken in the later phase of the project, when already too many developers contributed too many sources.

The results for project Gremon is somewhat above 0.05, but still significant.

On the other hand, results for Struts 2 is very strong.

The test for Tomcat shows absolutely no significance; however, the results are not contradicting either. The reason for this might be the fact that the same author performed more than 80% of the commits, which caused a huge bias compared to the other projects. Therefore Tomcat is an atypical project from this respect.

Now let us illustrate the results using Quantile Difference Diagrams, as presented in Section 4.3.2.

Figure 5.6 illustrates the cumulative code churn comparisons. The cumulative churn values are less, or at most equal on every quantile, for the commits related to maintainability increase, compared to those of maintainability decrease. The tendency of the line is negative (at higher values the line is located lower), and at higher values (from about 0.7) the amplitude is also higher than in case of lower values.

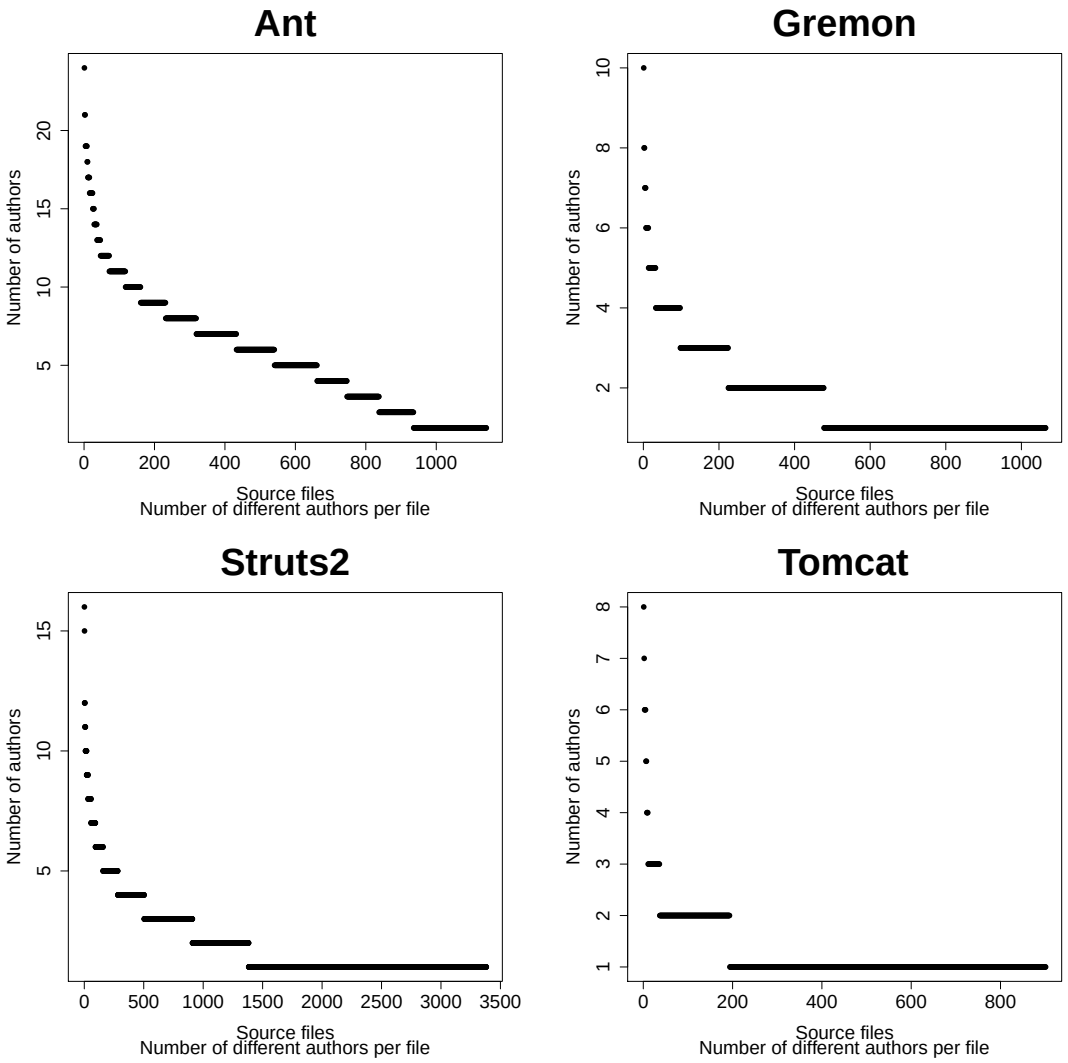

Figure 5.4. Number of authors per file

Figure 5.7 illustrates the ownership comparisons. Here also all the differences are non-positive, however, there are much more zeros, compared to the cumulative code churn related diagrams. The greatest difference can be found in case of Struts 2, which is not surprising according to the statistic test results (see the right column of Table 5.6).

Answers to the Research Questions

Based on the results above we can answer the research questions raised in Section 5.1.1:

3.A.RQ1: *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?*

The maintainability increases are mostly related to lower cumulative code churn values, while maintainability decreases are related to higher cumulative code churn values.

3.A.RQ2: *Does the number of developers modifying the same code in the past have any affect on the maintainability change of future commits?*

The maintainability increases are mostly related to clear code ownership, while maintainability decreases are related to lack of code ownership.

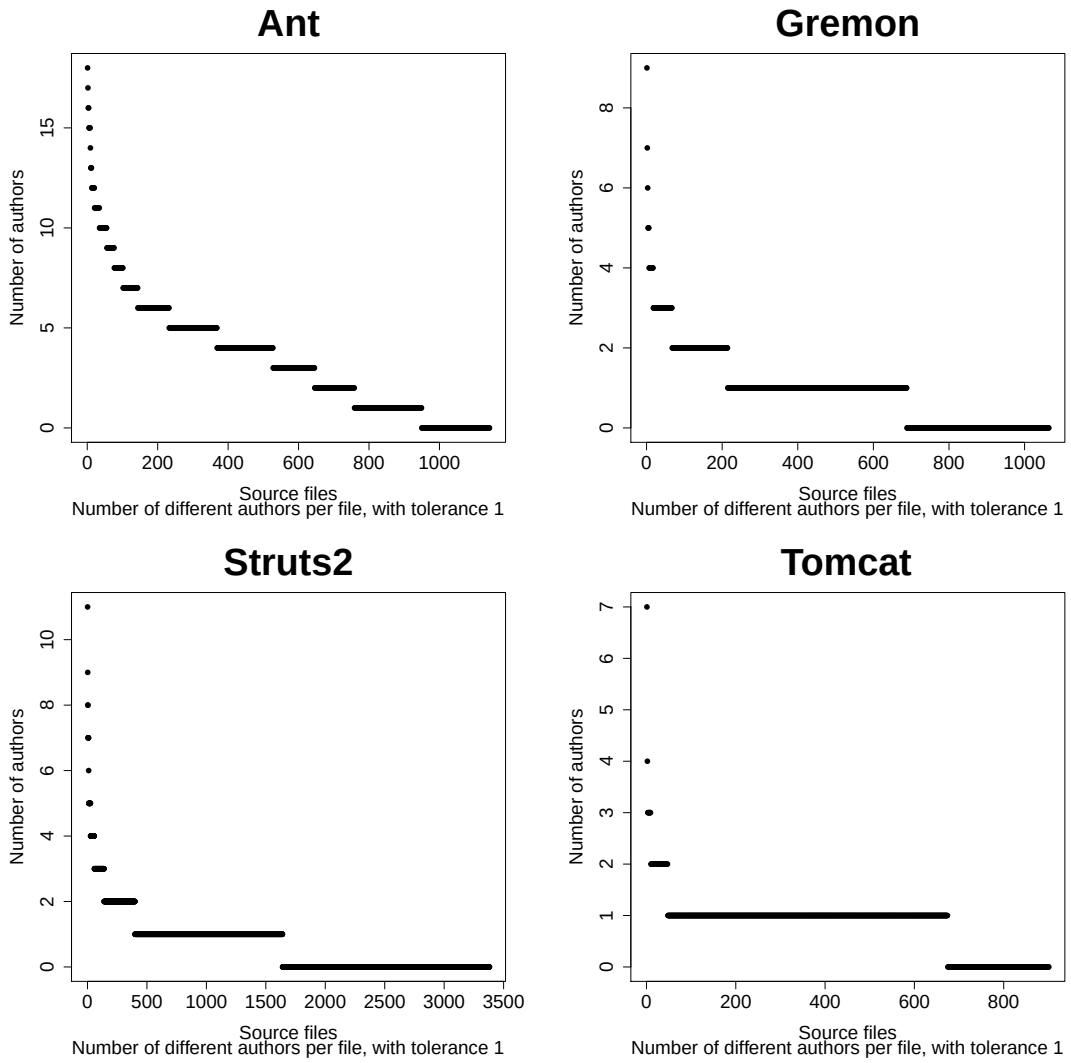

Figure 5.5. Number of authors per file with tolerance

Discussion

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

Regarding to *cumulative code churn results* 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 [88]) 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

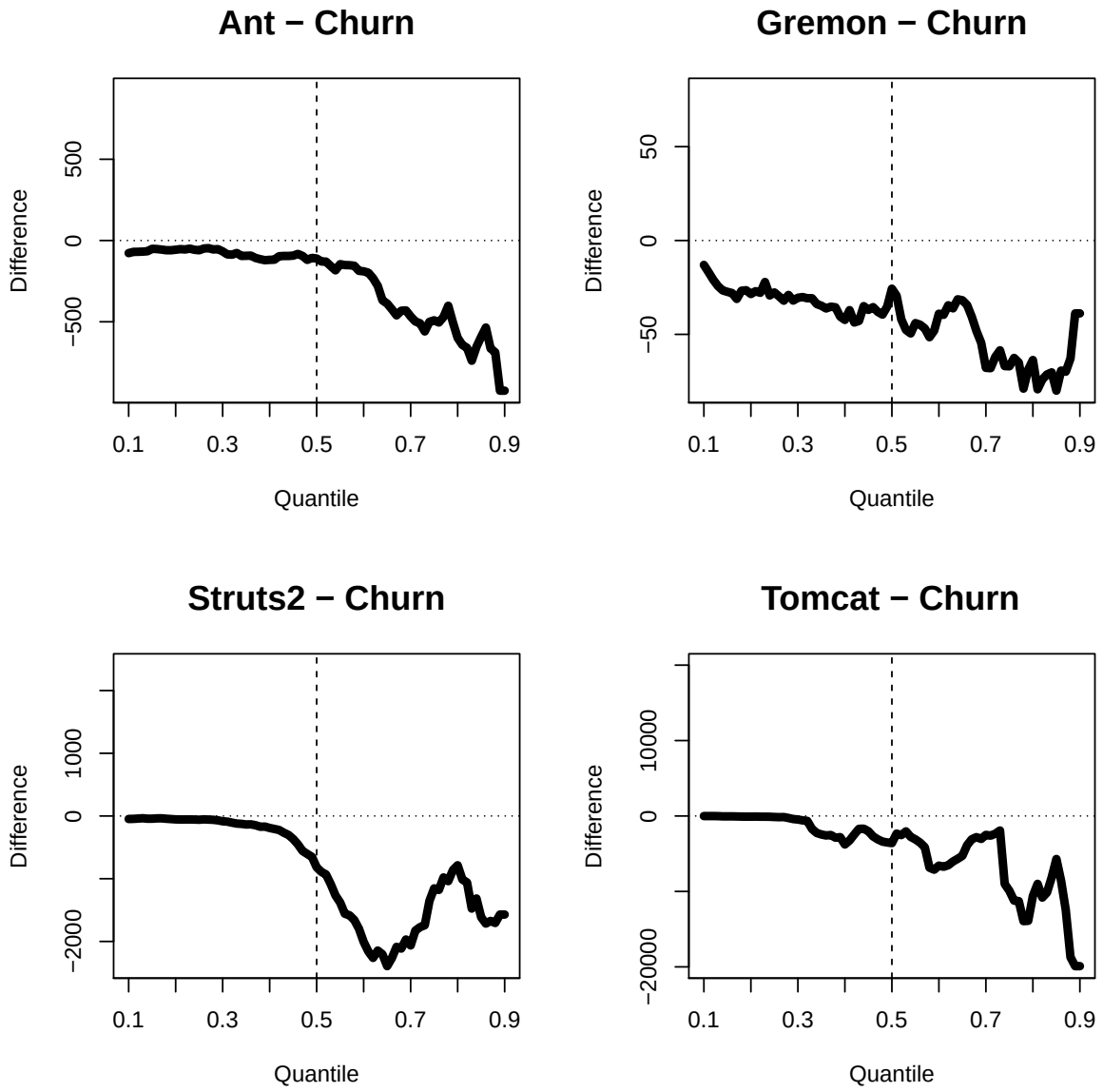

Figure 5.6. Quantile difference diagrams of cumulative code churns

on that file.

Another important note, which is a significant difference compared to the existing works, is the following. We examine the impact of cumulative code churn on the maintainability of the source code and not on the defects. Although we do not check the number of defects revealed later, we consider 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

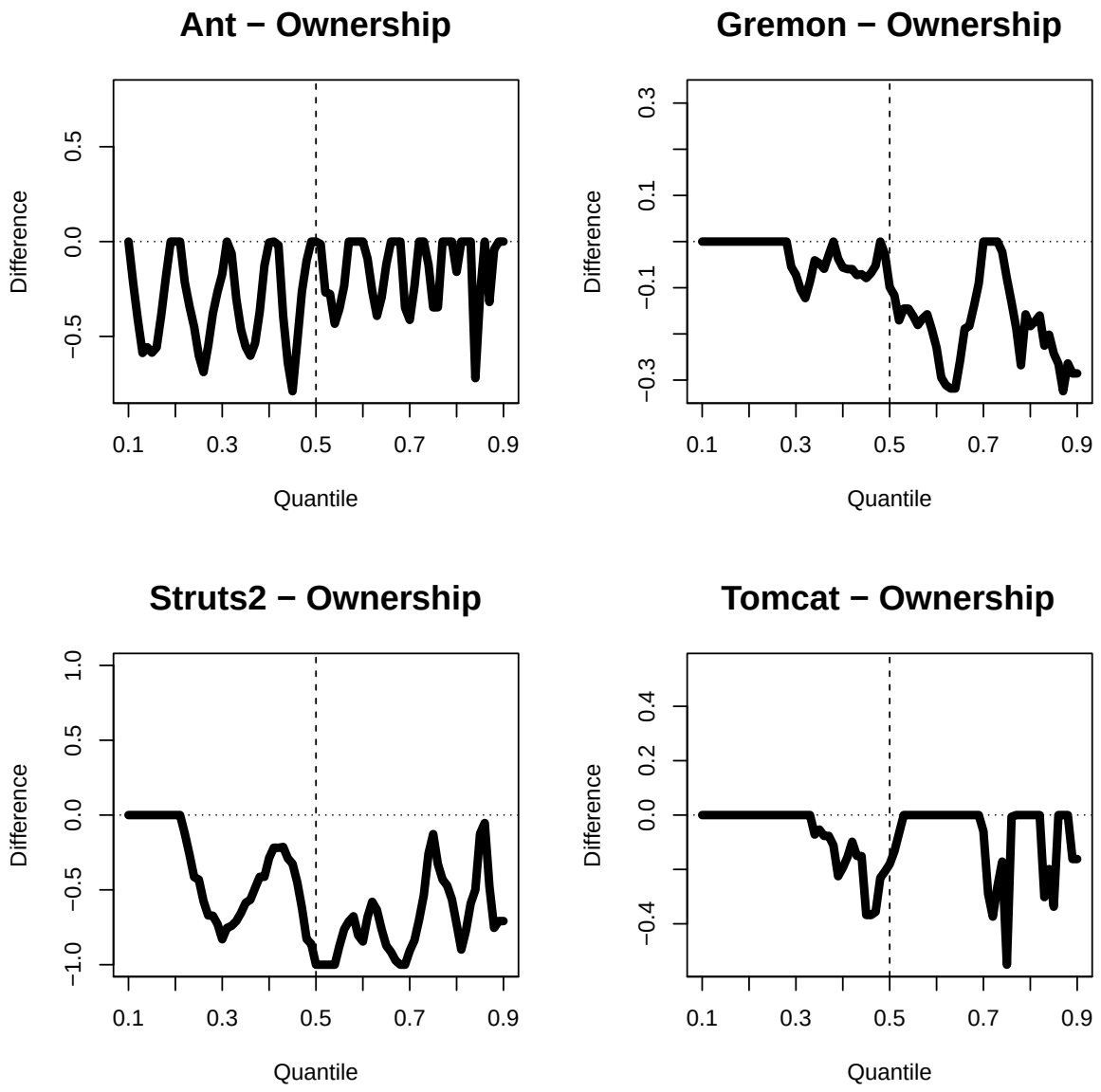

Figure 5.7. Quantile difference diagrams of code ownerships

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.

Regarding to *code ownership*, one could simplify the method as follows: the more people work on a system, the more complex it will be, and the more complex a systems is, the harder to maintain it. We consider this relationship as already known and did not even check it. On

the other hand, we state that the effect of the *future* modifications on source files changed by more developers in the *past* is more likely to lower the maintainability compared with modifications on files that have been changed by less number of developers earlier. Note that this is not trivial: a source file with several earlier contributor is likely to be more complex than those with clear ownership, and our statement is that the already low quality source code is more likely to become even worse than that of higher quality.

For the sake of better understanding we interpret our result as follows: source files which have been modified by more developers in the past is more likely to become more complex in the future than those with less number of contributors. An even more precise statement would be the following, which is harder to conceptualize: the number of earlier contributors of modifications resulting in code quality increase is more likely to be lower than those resulting in code quality decrease.

Now we highlight the limitations of the results. It would be an inappropriate interpretation that the quality of the code with clear ownership increases, and the quality of common code decreases. There are quality decreases in the sources with clear ownership, and quality increases even in the hottest code spots. The presented results are much more modest, but significant.

Regarding the strength of the results, we stress that the 4 analyzed systems have been fixed before the case studies.

5.2 Thesis Point 3.B: Correlation between Version Control History Metrics and Maintainability

5.2.1 Overview

In Section 5.1 we checked the impact of the past modification and ownership values on the future maintainability changes. In this part we define six version control history metrics: two modification intensity related, two code ownership related and two code aging related ones, and check their correlation with the maintainability.

Note the important difference between the first and the second (i.e. this) part. Considering the code churn, the first statement says that modifying source files which has been intensively modified in the past are more likely to cause further maintainability decrease, compared to those of less past modification. On the other hand, in this part we try to prove that if a source file has been intensively modified in the past, then it itself is less maintainable, compared to less intensively modified ones.

The first statement is somewhat stronger and less trivial. One could expect that the maintainability converge: if something is good, it is more likely to erode, and if something is bad, then there is a potential to get better.

Here we examine the more naturally expected behavior. But this – maybe weaker – statement is more important in our future vision: defining other metrics, aggregate them into a hotspot detector, which could be utilized by an IDE plug-in.

Putting the two statements together we can claim that the code maintainability does not converge: what is already wrong, it is likely to get even worse, compared to those of good maintainability.

This is similar to the experience in economy: people expected that the economic development would converge the countries to the same economic level. Phrases like "developing" and "developed" countries indicate this, meaning that for the already "developed" countries there is nowhere to develop further; on the other hand, the "developing" countries would get closer to the "developed" ones. In practice, just the opposite turned to be true. The countries we call

"developed" are developing further, but the politically correctly claimed "developing" counties are – with some nice exceptions – really poor countries, with less intensive development, or in several sad cases they declined.

Understanding the results presented in Section 5.1, considering the results of the second part, the result is similar to the country development analogy.

It might be unusual to the reader that the overview part starts with a result, but it might help an easier understanding the point of this section. From now on, until presenting the result, let us go back to the natural order of a study, do not consider the presented result. Let us consider the null hypothesis the following: there is no correlation between the version control history based metrics and the related source files' maintainability, because the source of information are independent.

In this section formally we investigate the following research questions:

3.B.RQ1: *How modifications intensity affects maintainability?*

3.B.RQ2: *How code ownership affects maintainability?*

3.B.RQ3: *How code aging affects maintainability?*

5.2.2 Methodology

Overview

We performed the analysis on a few certain versions of 4 software systems, as presented in Section 3.3.2. These systems differ from those used in Section4 and Section 5.1.

For a certain version of an analyzed system we determined the order of source files based on 3 independent factors: the version control history (which indeed resulted 6 orders of files, as we analyzed 6 metrics), the relative maintainability and the number of post-release bugs. We calculated the correlation between these orders.

Version Control History Metrics

We calculated the orders of files based on the following version control history metrics (each metric defined an own order).

- **Modification intensity** related metrics include cumulative code churn and number of modifications.
  - *Cumulative code churn* is the absolute sum of number of added and removed lines of code so far.
  - *Number of modifications* is the number of times the file in question has been modified so far.

  Ownership related metrics include contributors and the contributors with tolerance.
  - *Contributors* is the number of different contributors of the file so far.
  - *Contributors with tolerance* is the number of different contributors of the file so far, but if someone contributed to the file only once, then that contribution is not considered.
- **Aging** related metrics include age and last modification date.
  - *Age* is date when the file was added.
  - *Last modification date* is the date of the last modification.

Each of these metrics determines an order of files.

Relative Maintainability Indices

We calculated the Relative Maintainability Index for every class in the source code of the system as described in Section 3.2.2. The reason of this choice was the fact that in Java the source files and the classes are strongly correlated with each other: most of the source files contain exactly one class.

Number of Bugs

We considered bug data found in the PROMISE bug database [1], where the number of post release bugs of each source files of given release are made public.

Correlation Tests

We performed the Spearman's rank correlation check on every combination of version control history metrics on one hand, and RMIs and number of bugs on the other hand. This resulted $6 \cdot 2 = 12$ combinations of every analyzed versions. We performed the analysis using the R statistical software [103], using the `cor.test()` function.

5.2.3 Results

Results of the Correlation Tests

Tables 5.7 and 5.8 contain the results of the correlation tests between the version control history metrics, and RMI and bug, respectively.

System	Version	Churn	Modifs.	Ownership	Own.tolerance	Added	Modified
Ant	1.3	−0.861	−0.598	−0.392	−0.556	0.239	−0.563
	1.4	−0.867	−0.656	−0.475	−0.609	0.339	−0.373
	1.5	−0.747	−0.631	−0.550	−0.628	0.269	−0.592
	1.6	−0.852	−0.719	−0.636	−0.704	0.276	−0.464
	1.7	−0.702	−0.612	−0.560	−0.532	0.279	−0.268
jEdit	4.0	−0.712	−0.506	NA	−0.160	0.098	−0.442
	4.1	−0.681	−0.552	−0.515	−0.461	0.105	−0.466
	4.2	−0.713	−0.505	NA	−0.103	0.091	−0.478
	4.3	−0.302	−0.570	−0.488	−0.553	0.226	−0.044
Log4J	1.0	−0.823	−0.351	NA	−0.055	0.221	−0.283
	1.1	−0.873	−0.779	−0.556	−0.504	0.227	−0.535
	1.2	−0.854	−0.410	−0.481	−0.362	0.167	−0.102
Xerces	1.3	−0.660	−0.468	−0.217	−0.430	0.069	−0.100
	1.4	−0.481	−0.523	−0.322	−0.455	0.151	−0.355

Table 5.7. Spearman's correlation ρs of RMI comparison

The rows of the tables contain the name of the analyzed software system, its version, and the results of correlation between RMI (or bug) based order, and the order of the following: cumulative code churn, number of modifications, ownership, ownership with tolerance, added date and last modified date.

The correlation test between cumulative code churn and RMI resulted a very strong negative value, meaning the higher the cumulative code churn of a file is (i.e. it has been more

Name	Version	Churn	Modifs.	Ownership	Own.tolerance	Added	Modified
Ant	1.3	0.371	0.358	0.197	0.364	−0.142	0.398
	1.4	0.067	0.080	−0.029	0.108	0.138	0.326
	1.5	0.316	0.314	0.275	0.321	−0.176	0.231
	1.6	0.517	0.394	0.277	0.332	−0.174	0.393
	1.7	0.534	0.400	0.319	0.362	−0.172	0.236
jEdit	4.0	0.502	0.585	NA	0.063	−0.190	0.462
	4.1	0.546	0.653	0.539	0.536	−0.164	0.558
	4.2	0.405	0.501	NA	0.141	−0.177	0.345
	4.3	0.145	0.059	0.087	0.105	0.073	0.162
Log4J	1.0	0.589	0.388	NA	0.116	−0.304	0.301
	1.1	0.738	0.722	0.457	0.307	−0.204	0.531
	1.2	0.409	0.426	0.427	0.387	−0.294	−0.063
Xerces	1.3	0.365	0.357	0.203	0.270	−0.082	0.124
	1.4	0.255	0.408	0.090	0.484	−0.044	0.389

Table 5.8. Spearman's correlation ρs of bug comparison

intensively modified in the past), it is more likely that the RMI is lower (i.e. its maintainability is worse). The bug comparison also supported this result with positive correlation (higher cumulative code churn results in higher number of post-release bugs), with weaker correlation in absolute values.

The problem with cumulative code churn calculation is that it is very slow. On the other hand, we can express the intensity of past modification by the mere number of past modifications. The connection is similar to cumulative code churn, with weaker correlation.

The correlation between ownership and RMI was similar to churn or modification comparison, with somewhat moderate results. In 3 cases the comparison was not applicable, as all the affected files had the same number of contributors. In one case of bug comparison there was a contradiction. In our opinion this was casual; the small number of post release bugs of that version and the small number of contributors lead to this result.

The ownership with tolerance comparison resulted in more significant correlation. All the tests resulted a value, and there was no contradictory result.

In case of added date analysis the results indicated that the later added source files had better maintainability. We found that result surprising. Somewhat ironic explanation of this result can be the following: an early added source file had enough time to erode. However, the correlation in absolute value is weak, with two contradictions in case of bug comparison.

Finally, we found that the recently modified files are more likely to have worse maintainability, with a higher number of bugs, and the correlations are significantly higher than those of file addition dates. In case of bug comparison cross check there was one contradicting result.

Answers to the Research Questions

3.B.RQ1: *How modifications intensity affects maintainability?*

Higher intensity of file modifications result in worse maintainability and higher post release bugs. The cumulative code churn turned out to result in the highest correlation values. The number of file modifications also resulted in high, but lower correlation values.

3.B.RQ2: *How code ownership affects maintainability?*

Source files of lacking clean code ownership result in worse maintainability and higher post

release bugs. The mere number of contributors so far has a moderate strength of predicting maintainability and post release bugs, and it is somewhat higher if we apply a tolerance, not considering only one contributions.

3.B.RQ3: *How code aging affects maintainability?*

Earlier added and later last modified files result in worse maintainability and higher post release bugs. However, the strength of these correlations are the smallest, with a few contradictory results in case of bug comparison.

5.3 Summary

In this chapter we tried to find connection between version control history metrics and maintainability.

In Section 5.1 we checked the effect of past code modification intensity and of the ownership on the maintainability of the present commits. For the file modification intensity check we calculated for each file and revision from the very beginning how many lines were added and deleted all together. For code ownership check we considered how many different developers contributed to the file. At certain commit we took their mean, separately for code churn and code ownership. We divided these values into two subsets, based on the maintainability change of the related commit, if it decreased or increased it. We omitted the commits related to neutral maintainability changes. Then we compared the values using Wilcoxon-test. As the result we gained that the past intensive modifications and the lack of clean code ownership foretell the decrease of the maintainability.

In Section 5.2 we defined the following six version control history metrics: cumulative code churn, number of modifications, ownership, ownership with tolerance, code age, and last modification time. We tried to find a connection between them and the maintainability, one by one, and as a cross-check we tried to find their connection with the number of post-release bugs. We performed the test as follows. For a certain version of the analyzed system we sorted the source files based on every metrics. We determined the order of files on the Relative Maintainability Index basis (see Section 3.2.2); and we determined the order based on the number of post-release bugs as well. Then we tested the similarity of these orders with help of the Spearman's rank correlation test, using `cor.test()` R function. As result we got that higher intensity of modifications, the higher number of code modifications and developers (without and with tolerance), the older code and the later last modification date resulted lower maintainability and higher number of post-release bugs.

Now let us compare the differences of the two statements, considering the code modification intensity. In Section 5.1 we concluded that if a file was intensively modified in the past, it was likely that a future commit causes maintainability decay, compared to those of less intensive past modifications. In Section 5.2 we concluded that if a file was intensively modified in the past, it was likely that its maintainability was worse, compared to those of less intensive past modification. In the first statement we try to foretell the future based on the past. In the second one we tell how the analyzed metrics go with maintainability hand by hand. Therefore the second statement is somewhat weaker than the first one; however, they are different, and the second one could be more useful for utilizing the results.

What can be this utilization? We would like to aggregate the results, and implement a version control history based hotspot detector. We already tried to aggregate them, but the trivial methods have not provided better result than the modification intensity on its own. Therefore we have to study other aggregation methodologies.

There are version control history related information left intact or not fully explored. For example, we have not yet considered the comments at all, furthermore, there might be potentials in other aspects of developers and modification intensity, and especially in dates of the commits.

Therefore the border with the unknown is quite long, and we plan to try to discover this unknown in the near future.

5.4 Contributions

The author contributed to the new results presented in this chapter as follows:

- Elaborating the methodology of code churn and code ownership analysis, as described in Section 5.1. Performing the statistic tests in R, and evaluating the results.
- Defining the six version control history metrics as described in Section 5.2. Implementing the version control history metrics collector in Java. Elaborating the methodology of correlation test with Relative Maintainability Index and post-release bugs, implementing it using R, executing the tests and evaluating the results.
- All the illustrative diagrams and examples are the work of the author.

"Software is a great combination between artistry and engineering."
— Bill Gates

6 Conclusions

In this thesis we concerned two major topics: the program slicing and the maintainability analysis.

The research area of *program slicing* is huge and mature. To summarize the results presented in this thesis in one phrase could be the following: it is related to the unstructured statements handling of a specialized version of a certain dynamic program slicing algorithm. Therefore the work presented in this thesis is like a cog in the machine.

The topic of the *connection between version control history and software maintainability* is, strictly speaking, a young research area. This research field is located between software maintainability and mining software repositories.

The area of software maintainability is the elder and bigger one, having about half of the most relevant articles appeared before the millennium. The studies typically consider source code metrics and bug prediction. The field of mining software repositories is a more recent and evolving one; the vast majority of the articles appeared after 2000. A few of them also deal with bug prediction.

We are not aware of any publication which deals with the connection between version control data and the values a software maintainability model calculates. Therefore the second part of the thesis can be considered as a new and still small field, but with big potentials.

First we opened the box and showed that connection between version control history data and software maintainability exists. At the beginning we considered version control operations only, but later we calculated also with other data as well, like file name, name of the developer or the date of the commit.

We think there are still big potentials left in this research field. First of all, the version control history data are far not fully exploited: there are still several metrics left to be defined and analyzed. Aggregating the values – similarly to the quality models which aggregate source code metrics data – is still a fully open task.

To summarize, in the first part we made a small step forward in a big research area, while in the second part we made a pioneer work in a young research area.

Appendices

A

Summary in English

The thesis consists of two main topics: **dynamic program slicing** and the **impact of version control history metrics on maintainability**. There is a strong connection between them: the *maintainability* of program source code.

The thesis contains three thesis points: the first one enhances the topic of dynamic program slicing, while the other two are related to the version control history metrics. These are the following:

1. Unstructured C statements handling in a dynamic slicing algorithm
2. Connection between version control operations and maintainability
   - 2.A Existence of the connection between version control operations and maintainability
   - 2.B Impact of version control operations on value and variance of maintainability
   - 2.C Cumulative characteristic diagram and quantile difference diagram
3. Connection between version control history metrics and maintainability
   - 3.A Impact of code modifications and code ownership on maintainability
   - 3.B Correlation between version control history metrics and maintainability

Dynamic Program Slicing

A slice consists of all statements and predicates that might affect a set of variables at a program point. Slicing algorithms can be classified according to whether they only use statically available information or dynamic information as well, to static and dynamic program slicing. In this thesis we dealt only with dynamic program slicing.

Gyimóthy et al. [58] introduced a method for the forward computation of dynamic slices. The point of the methodology in a nutshell is the following. We determine statically for every line of code which variable gets value, and on which variables it depends on. We handle the conditional and cycle statements as virtual predicate variables. We instrument the program and execute it to gain the execution history. For every executed step, computing forward we calculate from which lines the actual line depends on. This is the union of the last

modification places and their dependent lines of those variables which the actually calculated variable depends on.

In practice we can consider the memory requirement of the methodology to be linear to the memory requirement of the original program, which was a significant improvement compared to the big memory requirements of earlier methodologies. However, the algorithm in its original format was inappropriate for slicing real programs, because it practically handled assignment, conditional and loop statements only.

In their study Beszédes et al. [18, 19] adopted the algorithm on the C programming language. They solved several issues, e.g. the function calls or the pointer handling.

One of the issues to be solved was the handling on unstructured statements in the C programming language, which are the following: `goto`, `break`, `continue` and `switch-case-default`. In our solution [39, 40] we introduced so-called label variables, which get value at the point of execution, and the dependent lines of the source are those located after the label. In case of `goto` the dependent statements are all the statements after the declaration of the label, within the function. In case of `break` the dependent statements are all the statements after the related code block (e.g. after the `while` block). In case of `continue` the dependency should be introduced from the first statement of the related code block until the end of the function. This also means that the `continue` always depends from itself. In case of `switch-case-default`, the `switch` statement should be handled with a predicate variable, similarly to e.g. in case of the `while` statement. If at least one internal statement is part of the result, then all the `case` labels, along with the `default`, should be put into the result.

Later we extended the method [17], where we defined the relevant slice as the union of the all possible executions. In case of significant code coverage the size of the resulting slice was a fraction of the result calculated by a static program slicing tool.

Overview of Version Control History and Maintainability

Measuring Maintainability

We used the ColumbusQM for calculating the maintainability. This was published by Bakota et al. [11]. It is based on the ISO/IEC 9126 standard [76]. The algorithm considers metrics like logical lines of code, coding rule violations, complexity and others. Earlier studies (like the article by Gyimóthy et al. [59]) state that the higher these values are (e.g. the longer a function is), the higher number of bugs it is likely to contain. It compares the metric values with other systems of a benchmark, and finally it aggregates the results.

An extension of the methodology was published by Hegedűs et al. [68, 69], which can determine the maintainability based relative position of every source code element (class, function) within the system. The base idea in a nutshell is the following: the maintainability analysis is performed on the whole system, and on the system without the analyzed source code element. The Relative Maintainability Index is difference between the original maintainability value and the maintainability value without that source code element.

The Systems Analyzed

In the major part of our study we analyzed the following software systems: Ant, Gremon, Struts 2, Tomcat. On the other hand, in a smaller part we had to find other systems to analyze, due to methodology reasons. These were the following: 5 version of Ant, 4 versions of jEdit, 3 versions of Log4J and 2 versions of Xerces; all together we analyzed 14 versions of 4 systems.

Connection between Version Control Operations and Maintainability

Here we checked the impact of the version control operations (file addition, update and deletion) on the maintainability. First we presented how we found the existence of the connection between these two independent series of data. Then we checked the effect of each version control operation to the value of maintainability change and variance on maintainability change. We illustrated the results with diagrams.

Existence of the Connection between Version Control Operations and Maintainability

First we showed that connection between version control operations and maintainability really existed, in spite of the fact that the data were coming from different sources [45].

For every commit we determined the maintainability change, and based on this we partitioned them into three disjoint subsets: maintainability increase, no change, and decrease. On the other hand, we divided them based on version control operations into the following four categories: (D) commits containing file deletion; (A) commits not containing file deletion, but containing file addition; (U+) commits containing updates only, at least two; (U1) commits consisting of exactly one update operation. The combination of the two forms a matrix having 12 cells. Each cell contains the number of commits with matching conditions.

For every analyzed system we performed the contingency Chi-Squared test, which tells if the difference between the expected and the actual distribution is significant. We used the `chisq.test()` R [103] function for performing the test.

The test resulted a significant difference for almost all the cells. It deflected from the expected almost always in the same direction, and mostly with similar magnitude.

Therefore we stated that there was a connection between version control operation and the maintainability. Some certain values indicated the connection, which we analyzed further, as detailed below.

Impact of Version Control Operations on Value and Variance of Maintainability

We considered file additions, file updates and file deletions one by one, and checked their impact on the size [41] and the variance of maintainability change [34, 36].

In the algorithm we divided the commits into subsets based on several aspects, and we compared the related maintainability change values. We considered the following divisions for all the three operations. First, we defined the main data set in three ways: all commits; commits containing the examined operation; commits consisting exclusively of the examined operation. In each case we performed the following divisions: division based on median of the number of that operation; division based on the median of the proportion of that operation; the main data set and its complementary. Therefore theoretically we defined 9 divisions, practically, by eliminating the trivial ones, we had 7 divisions.

So we performed the division of the commits on version control operation basis, and we considered the related maintainability change values. We defined the maintainability change values not simply as the differences of the subsequent revisions, but we considered their characteristic and the actual size of the code as well.

We compared these values. At the value comparison we used the Wilcoxon-test, which is not sensitive on the outliers. We performed the comparison using the `wilcox.test()` R function. As result, we gained an overview for each operations, in what circumstances what kind of effect they had on the maintainability.

We compared the variances belonging to the two sets of maintainability change values. The variance tests are extremely sensitive on the outliers, and the non-standard commits, like merging a branch into the master, might cause huge deflections. Therefore we omitted these commits at the comparison of the variances.

A spectacular method of comparing the variances is the ratio of variances. We determined if the difference was significant or not using the `var.test()` R function.

The tests resulted that file additions improved, or at least less eroded the maintainability than file updates. The file updates mainly eroded them. We could not establish the effect of the file deletion, we received contradictory results.

At the variance check we concluded that the file addition and file deletion increased, while the file update decreased the variance. As the amplitude was much bigger than the absolute change, as a final conclusion we stated that it was recommended to pay special attention on file additions.

Cumulative Characteristic Diagram and Quantile Difference Diagram

We presented two visualization methods, which were adequate for presenting some of the published results visually [37, 38].

The input of the *Cumulative Characteristic Diagram* is a set of number. We sort them non-ascendant, and for every index we calculate the sum from the first one up to that point. The indices represent the x-coordinate, and the sums represent the y-coordinate. The characteristic is created by connecting these points with lines.

On the Composite Cumulative Characteristic Diagrams we draw two or more Cumulative Characteristic Diagrams.

This diagram type turned to be suitable for illustrating the Chi-squared test, the Wilcoxon-test and the variance test.

Using the *Quantile Difference Diagram* we can compare two sets of numbers. First we sort the elements of both sets in non-descending order. Then we take the values of every centile from both sets, pairwise. E.g. the median will be paired with the median, the 90% with the 90% from the other one etc. We calculate the difference of every pair. Using the centiles as the x-coordinate and the differences as y-coordinate, finally connecting the points we gain the Quantile Difference Diagram. If the original data contains outliers, it is recommended these values not to depict; the default implementation does not consider the lower and upper 5%.

We showed that this diagram type was suitable for illustrating Wilcox test and variance test.

Connection between Version Control History Metrics and Maintainability

We went further in analyzing the information located in version control systems. Unlike earlier, here we considered which piece of information was related to which file, therefore making a connection between different commits.

First we examined the effect of the intensity of past modification intensity of source code, and of the level of code ownership, on the later maintainability changes. After that we defined six version control history metrics, and considering all of them one by one we checked their connection with the maintainability, and, as a cross-check, with the number of post release bugs.

Impact of Code Modifications and Code Ownership on Maintainability

Here we checked the effect of past cumulative code churn [43] and the number of contributors [42] on the maintainability of the present commits.

We calculated for each file and revision from the very beginning, how many lines had been added and deleted all together. On a certain commit we averaged these values. We divided these values into two subsets, based on the maintainability change of the related commit, if it decreased or increased it (we omitted the commits related to neutral maintainability changes). Finally we compared the values using Wilcoxon-test.

We performed similar steps in case of code ownership analysis. There we checked how many different developers contributed to the file, and at certain commit we took their geometric mean. For the comparison here we also used Wilcoxon-test.

As the result we gained that the past intensive modifications and the lack of clean code ownership foretold the decrease of the maintainability.

Correlation between Version Control History Metrics and Maintainability

Finally we defined six version control history metrics, and checked their connection with maintainability [44]. These metrics were the following: cumulative code churn, number of modifications, ownership, ownership with tolerance, code age, and last modification time.

For a certain version of the analyzed system we sorted the source files based on every metrics. We determined the order of files on the Relative Maintainability Index [68, 69] basis; furthermore, as a cross-check, we determined the order based on the number of post-release bugs as well. Then we tested the similarity of these orders with help of the Spearman's rank correlation test, using `cor.test()` R function.

As result we got that higher intensity of modifications, the higher number of code modifications and developers (without and with tolerance), the older code and the later last modification date resulted lower maintainability and higher number of post-release bugs.

B

Magyar nyelvű összefoglaló

Az értekezés két fő részből áll: a *program szeletelésből* és a *verziókövető történeti metrikák hatása a karbantarthatóságra*. A két téma közötti szoros kapcsolatot a **szoftver karbantarthatóság** képezi.

Az értekezés összesen három tézis pontot tartalmaz, melyből egy a programszeletelés témakörét, míg a másik kettő a verzió követő metrikákat érinti. Az utóbbi két tézis pont alpontokra van osztva.

Az értekezés tézis pontjai az alábbiak:

1. A nem strukturált C utasítások kezelése egy dinamikus szeletelési algoritmusban
2. A verziókövető műveletek és a karbantarthatóság kapcsolata
   - 2.A A verziókövető műveletek és a karbantarthatóság kapcsolatának létezése
   - 2.B A verziókövető műveleteknek a karbantarthatóság értékére és varianciájára gyakorolt hatása
   - 2.C Halmozódó karakterisztika diagram és kvantilis különbség diagram
3. Verziókövető történeti metrikák és a karbantarthatóság kapcsolata
   - 3.A A kód módosításainak és a kód tulajdonlásnak a karbantarthatóságra gyakorolt hatása
   - 3.B Verziókövető történeti metrikák és a karbantarthatóság korrelációja

Dinamikus programszeletelés

A program szelet egy program utasításainak és predikátumainak egy részhalmaza, amelyek hatással vannak adott változókra adott ponton. Függően attól, hogy a szelet meghatározása során csak statikus vagy dinamikus információt is figyelembe veszünk, beszélhetünk statikus vagy dinamikus programszeletelési eljárásról. E tézispontban dinamikus programszeleteléssel foglalkozunk.

Gyimóthy Tibor szerzőtársaival tanulmányukban [58] egy előre haladó számolási technikát alkalmazó dinamikus programszeletelési algoritmust tettek közzé. A módszer lényege dióhéjban az alábbi. Mindegyik programsorban statikusan meghatározzuk, hogy mely változó kap

értéket, és az mely változók értékétől függ. Az elágazó és ciklus utasításokat virtuális predikátum változóként kezeljük. A programot instrumentáljuk és úgy futtatjuk, megkapva ezzel azt, hogy mely programsorok milyen sorrendben hajtódtak végre. Előre számolva minden egyes végrehajtási lépésben kiszámoljuk azt, hogy az adott sor aktuálisan mely más soroktól függ. Ez azon változók utolsó módosítási helyeinek és aktuális függőségeinek uniója, melyektől a kérdéses sorban kiszámolt változó függ.

A módszer memória igénye a gyakorlatban lineárisnak mondható az eredeti program memória igényével, ami jelentős javulás a korábbi módszerek hatalmas memóriaigényéhez képest. Ugyanakkor az algoritmus eredeti formájában alkalmatlan volt valós programok szeletelésére, mivel az csak értékadó, elágazó és ciklus utasításokat kezelt.

Beszédes Árpád szerzőtársaival cikkükben [19, 18] a módszert valós C programokra illesztették rá. Ebben számos problémát kellett megoldani, például a függvényhívásokét vagy a mutatók kezelését.

Az egyik megoldandó probléma a C programozási nyelvben jelen levő nem strukturált utasítások kezelése, melyek a következők: `goto`, `break`, `continue` és `switch-case-default`. Megoldásunkban [39, 40] úgynevezett címke változókat vezettünk be, melyek értéket a kérdéses utasítás végrehajtásakor kapnak, ettől a változótól pedig az a címke helyét követő utasítások függnek. A `goto` esetén a függő utasítások halmaza az összes, címkét követő utasítás, adott függvényen belül. A `break` esetén a függő utasítások a vonatkozó blokk (pl. a `while` ciklus belseje) utáni összes utasítás. A `continue` esetén a vonatkozó egység első utasításától kezdve az függvény végéig bele kell helyeznünk a függőséget. Ez egyébként azt is jelenti, hogy a `continue` mindig függ saját magától. A `switch-case-default` kezelésénél a `switch` ugyanolyan predikátum változóként kezelendő, mint pl. a `while`. Ha legalább egy belső utasítást tartalmaz az eredmény, akkor az összes `case` címkét az eredménybe kell tenni, a `default` címkével együtt.

A módszer kiterjesztését Beszédes Árpád publikálta szerzőtársaival [17], melyben a releváns programszeletet az összes lehetséges lefutás uniójaként definiálták. Jelentős programsor lefedettség mellett is az eredmény töredéke lett annak, amit egy statikus programszeletelő program kiszámolt.

A verziókövető adatok és a karbantarthatóság áttekintése

A karbantarthatóság mérése

A szoftver egy adott verziójának karbantarthatóságát a ColumbusQM szoftver karbantarthatóságot kiszámító programmal határoztuk meg. Ezt Bakota Tibor publikálta szerzőtársaival [11], ami az ISO/IEC 9126 szabványra épül. Az algoritmus olyan forráskód metrikákat vesz figyelembe, mint a vizsgált komponensek programsorban mért hossza, kódolási szabálysértések, komplexitás, és még számos egyéb. Korábbi tanulmányok alapján (ilyen pl. a Gyimóthy Tibor és szerzőtársai által publikált cikk [59]) állíthatjuk, hogy minél nagyobbak ez az érték (pl. minél hosszabb egy függvény), annál valószínűbb, hogy hibákat tartalmaz. A metrika értékeket összehasonlítja egy külső rendszerekből álló szoftver halmaz hasonló értékeivel, végül az eredményeket összegzi.

A módszer kiterjesztését Hegedűs Péter publikálta szerzőtársaival [68, 69], melynek segítségével minden forráskód elemre (osztályra, függvénye) meg tudjuk mondani annak a helyét karbantarthatósági szempontból a vizsgált rendszeren belül. A módszer lényege az, hogy sorban egyesével kivéve a forráskód elemeket elvégezzük a rendszer minősítését, és a relatív karbantarthatósági index a teljes rendszerre kiszámított érték és az adott forráskód elem nélküli érték különbsége.

Az elemzett rendszerek

Vizsgálatunk nagyobbik részében a következő négy szoftver rendszert elemeztük: Ant, Gremon, Struts 2, Tomcat. Kisebb részt viszont módszertani okok miatt újabbakat kellett keresnünk, melyek az alábbiak: az Ant öt verziója, a jEdit négy verziója, a Log4J három verziója, a Xerces-nek pedig 2 verziója, összesen tehát 4 rendszer 14 verzióját elemeztük.

A verziókövető műveletek és a karbantarthatóság kapcsolata

Először azt vizsgáltuk, hogy a verziókövető műveleteknek (fájl hozzáadás, módosítás, törlés) milyen hatásuk van a karbantarthatóságra. Kimutattuk a kapcsolat létezését e két független adatsor között. Ezután megvizsgáltuk mindegyik verzió követő utasításnak a karbantarthatóság megváltozására és annak varianciájára gyakorolt hatását. Az eredményeket diagramokkal illusztráltuk.

A verziókövető műveletek és a karbantarthatóság kapcsolatának létezése

Megmutattuk, hogy létezik kapcsolat a verziókövető utasítások számossága és a karbantarthatóság megváltozása között, annak ellenére, hogy két különböző adatforrásból származó adatokról van szó [45]. Ezt az alábbi módon tettük. Minden egyes módosításra meghatároztuk a karbantarthatóság változást, és e szerint három csoportba soroltuk őket: nőtt, nem változott vagy csökkent a karbantarthatóság. Másrészt a verziókövető műveletek szeriont a következő négy kategóriába soroltuk: (D) az adott módosítás tartalmaz törlést; (A) nem tartalmaz törlést, de tartalmaz hozzáadást; (U+) kizárólag több módosítást tartalmaz; (U1) egyetlen módosítást tartalmaz. A kettő kombinációja egy 12 cellát tartalmazó mátrixot alkot. A mátrix cellái az adott feltételeknek megfelelő módosítások számát tartalmazzák.

Mindegyik vizsgált rendszerre végrehajtottuk a kontingencia Khi-négyzet tesztet, mely megmondja, hogy szignifikáns-e az eltérés a várható és a tényleges eloszlás között. A teszt végrehajtásához a `chisq.test()` R [103] függvényt használtuk.

A legtöbb cellára azt kaptuk, hogy szignifikáns az eltérés. Majdnem mindig ugyanabba az irányba, és többnyire hasonló mértékben tért el a várttól.

Ezáltal kijelentettük, hogy van kapcsolat a verziókövető műveletek és a karbantarthatóság között, egyes konkrét értékek pedig előre vetítették a konkrét kapcsolatot, melyet az alább leírt módon részletesen megvizsgáltunk.

A verziókövető műveleteknek a karbantarthatóság értékére és varianciájára gyakorolt hatása

Egyenként megvizsgáltuk a fájl hozzáadást, módosítást és törlést, és megvizsgáltuk a hatásukat a karbantarthatóság megváltozásának értékére [41], valamint annak varianciájára [34, 36].

A módszer során a módosításokat különböző szempontok szerint részhalmazokra bontottuk, és a hozzájuk tartozó karbantarthatóság változás értékeket hasonlítottuk össze. Mindhárom műveletre az alábbi felosztásokat vettük. Először is háromféleképpen definiáltuk a fő adathalmazt: az összes módosítás; azok a módosítások, melyben előfordul a kérdéses művelet; azok a módosítások, melyek kizárólag az adott műveletből állnak. Mindegyik esetben a módosításokon a következő felosztásokat hajtottuk végre: felosztás a vizsgált művelet száma szerinti medián alapján; felosztás a vizsgált művelet arányának mediánja alapján; a fő adathalmaz és annak komplementere. Ezzel elvileg 9, a triviális felosztásokat nem számolva valójában 7 felosztást definiáltunk.

A módosítások felosztása tehát a verziókövető műveletek szerint történt; most mindegyik felosztáshoz vettük a vonatkozó karbantarthatóság változás értékeket. A karbantarthatóság érték megváltozását nem egyszerűen az egymást követő verziók karbantarthatósági értékeinek

különbségeként definiáltuk, hanem figyelembe vettük annak karakterisztikáját, valamint a program aktuális méretét is.

Ezeket az értékeket összehasonlítottuk. Az értékek összehasonlításánál a Wilcox tesztet alkalmaztuk, mely nem érzékeny a szélsőséges értékekre. Az összehasonlítást a `wilcox.test()` R függvénnyel végeztük el. Ezzel képet kaptunk arról, hogy mely műveleteknek milyen körülmények között milyen hatásuk van a karbantarthatóságra.

Összehasonlítottuk a két módosítás halmazhoz tartozó karbantarthatóság változás értékek varianciáit is. A variancia számítás rendkívül érzékeny a szélsőséges értékekre, márpedig a nem szokványos módosítások (például ilyen egy külön ág fejlesztéseinek beolvasztása a fő ágba) hatalmas kilengést okozhatnak. Emiatt ezeket a módosításokat a varianciák összehasonlítása során nem vettük figyelembe.

A variancia összehasonlítás szemléletes módja a varianciák hányadosa. Annak megállapítását, hogy az eredmény szignifikáns-e, a `var.test()` R függvény segítségével végeztük el.

Eredményül azt kaptuk, hogy a fájl hozzáadások javítják a karbantarthatóságot, legalábbis kevésbé rontják, mint a fájl módosítások. A fájl módosítások ugyanis nagyrészt rontják azt. A fájl törlés hatását nem sikerült egyértelműen kimutatnunk, ott ellentmondó eredményre jutottunk.

A variancia vizsgálat során megállapítottuk, hogy a fájl hozzáadása és a törlése növeli, míg a módosítás csökkenti a varianciát. Mivel a kilengés sokkal nagyobb, mint az abszolút megváltozás, végső konklúzióként arra jutottunk, hogy elsősorban a fájl hozzáadásokra érdemes leginkább odafigyelni.

Halmozódó karakterisztika diagram és kvantilis különbség diagram

Bemutattunk két vizualizációs módszert, amely alkalmas egyes publikált eredmények képi megjelenítésére [37, 38].

A *halmozódó karakterisztika diagram* bemenete egy számhalmaz. Ezeket nem növekvő sorrendbe rendezzük, majd minden egyes indexre kiszámoljuk azok összegét az elsőtől addig a pontig. A indexek képezi az x-tengelyt, az összegek pedig az y-t. A karakterisztika úgy keletkezik, hogy ezeket a pontokat összeköltjük. Az összetett halmozódó karakterisztika diagramon kettő vagy több halmozódó karakterisztikát ábrázolunk.

E diagram típus alkalmasnak bizonyult a kontingencia Khi-négyzet teszt, a Wilcox teszt és a variancia teszt illusztrálására.

A *kvantilis különbség diagram* segítségével két számhalmazt tudunk összehasonlítani. Ehhez külön-külön lerendezzük mindkét halmaz elemeit nem csökkenő sorrendben. Majd mindkét halmazból páronként vesszük mindegyik centilishez tartozó értékeket. Tehát például a medián a mediánnal lesz párban, a 90% a másik 90%-kal stb. Mindegyik pár esetén képezzük azok különbségét. A centilisekből képezve az x-koordinátát, a különbségekből az y-koordinátát, a keletkező pontokat pedig egyenes szakaszokkal összekötve kapjuk a kvantilis különbség diagramot. Ha az adathalmaz tartalmaz szélsőséges értékeket, célszerű a széleket nem ábrázolni; az alapértelmezett megvalósítás nem veszi figyelembe az alsó és a felső 5%-ot.

Megmutattuk, hogy ez a diagram alkalmas a Wilcox teszt és a variancia teszt illusztrálására.

Verziókövető történeti metrikák és a karbantarthatóság kapcsolata

Tovább léptünk a verziókövető rendszerben található információk elemzésével kapcsolatban. A korábbival ellentétben itt figyelembe vettük azt, hogy az adott információ mely fájlra vonatkozik, ezáltal kapcsolatot teremtve különböző forráskód módosítási egységek között.

Először a forráskód múltbeli módosítások intenzitásának, valamint a kód tulajdonlás mértékének a hatását vizsgáltuk a későbbi karbantarthatóság megváltozására. Ezt követően definiáltunk 6 verziókövető történeti metrikát, és mindegyikre megvizsgáltuk annak kapcsolatát a karbantarthatósággal, és – mintegy ellenőrzésképpen – az utólagosan javított hibák számával.

A kód módosításainak és a kód tulajdonlásnak a karbantarthatóságra gyakorolt hatása

Itt azt vizsgáltuk, hogy a múltbeli kód módosítás halmozódó intenzitása [43], valamint a módosítók száma [42] milyen hatással vannak a jelenlegi módosításnak a karbantarthatóságra gyakorolt hatására.

Minden egyes fájlra és verzióra kiszámoltuk azt, hogy kezdettől fogva összesen hány sort adtak hozzá és hány sort töröltek. Adott módosításnál ezeket az értékeket átlagoltuk. Az így kiszámolt értékeket két csoportba osztottuk aszerint, hogy a vonatkozó módosítás csökkentette vagy növelte a karbantarthatóságot (a karbantarthatóság változás szempontjából semleges esetekkel nem foglalkoztunk). Végül Wilcox teszt segítségével hasonlítottuk össze az értékeket.

A kód tulajdonlás esetén is hasonlóan jártunk el. Ott azt vizsgáltuk, hogy adott fájlt összesen hány különböző fejlesztő módosított, adott módosításnál pedig a geometriai közepét vettük. Az összehasonlítás itt is a fentihez hasonlóan, a Wilcox teszt segítségével történt.

Eredményül azt kaptuk, hogy a múltbéli intenzív módosítás és az egyértelmű kódtulajdonlás hiánya is a karbantarthatóság csökkenését vetíti előre.

Verziókövető történeti metrikák és a karbantarthatóság korrelációja

Végül definiáltunk hat verziókövető metrikát, és megvizsgáltuk azok kapcsolatát a karbantarthatósággal [44]. Ezek a metrikák a következők: halmozódó változás intenzitás, módosítások száma, módosítók száma, módosítók száma toleranciával, a kód kora és az utolsó módosítás időpontja.

A vizsgált szoftver adott verziójára mindegyik metrika szerint sorba rendeztük a forrásfájlokat. A fájlok sorrendjét megállapítottuk a relatív karbantarthatóság index [68, 69] alapján is, valamint ellenőrzésképpen az adott verzióban talált hibák szerinti sorrendet is. Majd a sorrendek hasonlóságát megállapítottuk a Spearman sorrend korrelációs teszt segítségével, melyhez a `cor.test()` R függvényt használtuk.

Eredményül azt kaptuk, hogy a nagyobb változás intenzitás, a módosítások valamint a módosítók (tolerancia nélküli és toleranciával számított) magasabb száma, a régebbi kód és a friss utolsó módosítás rosszabb karbantarthatóságot és nagyobb számú hibát eredményez.

Bibliography

[1] Promise defect database. `http://openscience.us/repo/defect/`.

[2] Daniel Adler. *vioplot: Violin plot*, 2005. R package version 0.2.

[3] Hiralal Agrawal. On slicing programs with jump statements. In *ACM Sigplan Notices*, volume 29, pages 302–312. ACM, 1994.

[4] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. In *ACM SIGPLAN Notices*, volume 25, pages 246–256. ACM, 1990.

[5] Alan Agresti. *An introduction to categorical data analysis*, volume 135. Wiley New York, 1996.

[6] Samuel A Ajila and Razvan T Dumitrescu. Experimental use of code delta, code churn, and rate of change to understand software product line evolution. *Journal of Systems and Software*, 80(1):74–91, 2007.

[7] David L Atkins, Thomas Ball, Todd L Graves, and Audris Mockus. Using version control data to evaluate the impact of software tools: A case study of the version editor. *Transactions on Software Engineering*, 28(7):625–637, 2002.

[8] Adrian Bachmann and Abraham Bernstein. When process data quality affects the number of bugs: Correlations in software engineering datasets. In *Proceedings of the 7th Working Conference on Mining Software Repositories (MSR)*, pages 62–71. IEEE Computer Society, 2010.

[9] Tibor Bakota. *Evaluating the effect of code duplications on software maintainability.* PhD thesis, SzTE, 2013.

[10] Tibor Bakota, Péter Hegedűs, István Siket, Gergely Ladányi, and Rudolf Ferenc. Qualitygate sourceaudit: a tool for assessing the technical quality of software. In *Proceedings of the CSMR-WCRE Software Evolution Week*, pages 440–445. IEEE Computer Society, Febr 2014.

[11] Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A probabilistic software quality model. In *Proceedings of the 27th International Conference on Software Maintenance (ICSM)*, pages 243–252. IEEE Computer Society, 2011.

[12] Tibor Bakota, Péter Hegedűs, Gergely Ladányi, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A cost model based on software maintainability. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*, pages 316–325. IEEE Computer Society, 2012.

[13] Thomas Ball and Susan Horwitz. *Slicing programs with arbitrary control-flow.* Springer International Publishing, 1993.

[14] Yoav Benjamini. Opening the box of a boxplot. *The American Statistician*, 42(4):257–262, 1988.

[15] Árpád Beszédes. *Forráskód analízis és szeletelés a programmegértés támogatásához*. PhD thesis, SzTE, 2005.

[16] Árpád Beszédes. Global dynamic slicing for the c language. *Acta Polytechnica Hungarica*, 12(1), 2015.

[17] Árpád Beszédes, Csaba Faragó, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Union slices for program maintenance. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM)*, pages 12–21. IEEE Computer Society, 2002.

[18] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, Janos Csirik, and Tibor Gyimothy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 105–113. IEEE Computer Society, 2001.

[19] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, Csaba Faragó, and Tibor Gyimóthy. Forward computation of dynamic slices of c programs. Technical report, Technical Report TR-2000-001, RGAI, 2000.

[20] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality?: an empirical case study of windows vista. *Communications of the ACM*, 52(8):85–93, 2009.

[21] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Proceedings of the 20th International Symposium on Software Reliability Engineering (ISSRE)*, pages 109–119. IEEE Computer Society, 2009.

[22] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 4–14. ACM, 2011.

[23] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *Proceedings of the 21st International Conference on Automated Software Engineering (ASE)*, pages 221–230. IEEE Computer Society, 2006.

[24] Lionel C Briand, Jürgen Wüst, John W Daly, and D Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.

[25] Fernando Brito e Abreu and Walcelio Melo. Evaluating the impact of object-oriented design on software quality. In *Proceedings of the 3rd International Software Metrics Symposium*, pages 90–99. IEEE Computer Society, 1996.

[26] Gerardo Canfora and Luigi Cerulo. Impact analysis by mining software and change request repositories. In *Proceedings of the 11th International Symposium Software Metrics*, pages 9–pp. IEEE Computer Society, 2005.

[27] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11):595–607, 1998.

[28] John M Chambers, William S Cleveland, Beat Kleiner, and Paul A Tukey. Graphical methods for data analysis. *Wadsworth, Belmont, CA*, 1983.

[29] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Transactions on Software Engineering*, 20(6):476–493, 1994.

[30] Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1097–1113, 1994.

[31] Homepage of `CodeSurfer`. `http://www.grammatech.com/products/codesurfer/`.

[32] Stephen G Eick, Todd L Graves, Alan F Karr, J Steve Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *Transactions on Software Engineering*, 27(1):1–12, 2001.

[33] Csaba Faragó. Case study for the `vudc` R package. *Annales Mathematicae et Informaticae (submitted)*.

[34] Csaba Faragó. Variance of source code quality change caused by version control operations. In *Proceedings of the 9th Conference of PhD Students in Computer Science (CSCS)*, pages 12–13, 2014.

[35] Csaba Faragó. *hotspot: Software Hotspot Analysis*, 2015. R package version 1.0.

[36] Csaba Faragó. Variance of source code quality change caused by version control operations. *Acta Cybernetica*, 22(1):35–56, 2015.

[37] Csaba Faragó. Visualization of univariate data for comparison. *Annales Mathematicae et Informaticae*, 45:39–53, 2015.

[38] Csaba Faragó. *vudc: Visualization of Univariate Data for Comparison*, 2016. R package version 1.1.

[39] Csaba Faragó and Tamás Gergely. Handling the unstructured statements in the forward dynamic slice algorithm. In *Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST)*, pages 71–83, 2001.

[40] Csaba Faragó and Tamás Gergely. Handling pointers and unstructured statements in the forward computed dynamic slice algorithm. *Acta Cybernetica*, 15(4):489–508, 2002.

[41] Csaba Faragó, Péter Hegedűs, and Rudolf Ferenc. The impact of version control operations on the quality change of the source code. In *Proceedings of the 14th International Conference on Computational Science and Its Applications (ICCSA)*, volume 8583 Lecture Notes in Computer Science (LNCS), pages 353–369. Springer International Publishing, 2014.

[42] Csaba Faragó, Péter Hegedűs, and Rudolf Ferenc. Code ownership: Impact on maintainability. In *Proceedings of the 15th International Conference on Computational Science and Its Applications (ICCSA)*, volume 9159 Lecture Notes in Computer Science (LNCS), pages 3–19. Springer International Publishing, 2015.

[43] Csaba Faragó, Péter Hegedűs, and Rudolf Ferenc. Cumulative code churn: Impact on maintainability. In *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 141–150. IEEE Computer Society, 2015.

[44] Csaba Faragó, Péter Hegedűs, Gergely Ladányi, and Rudolf Ferenc. Impact of version history metrics on maintainability. In *Proceedings of the 8th International Conference on Advanced Software Engineering & Its Applications (ASEA)*, pages 30–35. IEEE Computer Society, 2015.

[45] Csaba Faragó, Péter Hegedűs, Ádám Zoltán Végh, and Rudolf Ferenc. Connection between version control operations and quality change of the source code. *Acta Cybernetica*, 21(4):585–607, 2014.

[46] Rudolf Ferenc. *Modelling and reverse engineering C++ source code*. PhD thesis, SzTE, 2005.

[47] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus-reverse engineering tool and schema for c++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM)*, pages 172–181. IEEE Computer Society, 2002.

[48] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[49] Michael Frigge, David C Hoaglin, and Boris Iglewicz. Some implementations of the boxplot. *The American Statistician*, 43(1):50–54, 1989.

[50] Thomas Fritz, Gail C Murphy, and Emily Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 341–350. ACM, 2007.

[51] Zachary P Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2012.

[52] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 13–23. IEEE Computer Society, 2003.

[53] Keith Brian Gallagher and James R Lyle. Using program slicing in software maintenance. *Transactions on Software Engineering*, 17(8):751–761, 1991.

[54] Tamás Gergely. *Programok Statikus és Dinamikus Analízise*. PhD thesis, SzTE, 2010.

[55] Emanuel Giger, Martin Pinzger, and Harald 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 (MSR)*, pages 83–92. ACM, 2011.

[56] Emanuel Giger, Martin Pinzger, and Harald C Gall. Can we predict types of code changes? an empirical analysis. In *Proceedings of the 9th International Working Conference on Mining Software Repositories (MSR)*, pages 217–226. IEEE Computer Society, 2012.

[57] Kenneth M Goldberg and Boris Iglewicz. Bivariate extensions of the boxplot. *Technometrics*, 34(3):307–320, 1992.

[58] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 303–321. Springer International Publishing, 1999.

[59] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Transactions on Software Engineering*, 31(10):897–910, 2005.

[60] Gregory A Hall and John C Munson. Software evolution: code delta and code churn. *Journal of Systems and Software*, 54(2):111–118, 2000.

[61] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 2014.

[62] Mark Harman and Sebastian Danicic. Amorphous program slicing. In *Proceedings of the 5th International Workshop on Program Comprehension (IWPC)*, pages 70–79. IEEE Computer Society, 1997.

[63] Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance*, 10(6):415–441, 1998.

[64] Mark Harman, Rob Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. In *Proceedings of the 17th International Conference on Software Maintenance (ICSM)*, page 138. IEEE Computer Society, 2001.

[65] Mark Harman, Arun Lakhotia, and David Binkley. Theory and algorithms for slicing unstructured programs. *Information and Software Technology*, 48(7):549–565, 2006.

[66] Ahmed E Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance (FoSM)*, pages 48–57. IEEE Computer Society, 2008.

[67] Lile Hattori and Michele Lanza. Mining the history of synchronous changes to refine code ownership. In *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR)*, pages 141–150. IEEE Computer Society, 2009.

[68] Péter Hegedűs, Tibor Bakota, Gergely Ladányi, Csaba Faragó, and Rudolf Ferenc. A drill-down approach for measuring maintainability at source code element level. In *Proceedings of the 7th International Workshop on Software Quality and Maintainability (SQM)*, page 20, 2013.

[69] Péter Hegedűs, Tibor Bakota, Gergely Ladányi, Csaba Faragó, and Rudolf Ferenc. A drill-down approach for measuring maintainability at source code element level. *Electronic Communications of the EASST*, 60:1–21, 2013.

[70] Péter Hegedűs. A probabilistic quality model for C# – an industrial case study. *Acta Cybernetica*, 21(1):135–147, 2013.

[71] Péter Hegedűs. *Advances in Software Product Quality Measurement and its Applications in Software Evolution.* PhD thesis, SzTE, 2015.

[72] Abram Hindle, Daniel M German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 5th International Working Conference on Mining Software Repositories (MSR)*, pages 99–108. ACM, 2008.

[73] Jerry L Hintze and Ray D Nelson. Violin plots: a box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998.

[74] Myles Hollander and Douglas A. Wolfe. *Nonparametric Statistical Methods, 2nd Edition.* Wiley-Interscience, 2 edition, January 1999.

[75] Kurt Hornik, Achim Zeileis, and David Meyer. The strucplot framework: Visualizing multi-way contingency tables with vcd. *Journal of Statistical Software*, 17(3):1–48, 2006.

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

[77] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[78] Mariam Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31(3):197–214, 1995.

[79] Peter Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software*, 28(1):1–9, 2008.

[80] Taghi M Khoshgoftaar, Edward B Allen, Nishith Goel, Amit Nandi, and John 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*, pages 364–371. IEEE Computer Society, 1996.

[81] Taghi M Khoshgoftaar and Robert M Szabo. Improving code churn predictions during the system test and maintenance phases. In *Proceedings of the 10th International Conference on Software Maintenance (ICSM)*, pages 58–67. IEEE Computer Society, 1994.

[82] Stefan Koch and Christian Neumann. Exploring the effects of process characteristics on product quality in open source software development. *Principle Advancements in Database Management Technologies: New Applications and Frameworks*, page 132, 2009.

[83] Bogdan Korel. Computation of dynamic program slices for unstructured programs. *Transactions on Software Engineering*, 23(1):17–34, 1997.

[84] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[85] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[86] Bogdan Korel and Satish Yalamanchili. Forward computation of dynamic program slices. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 66–79. ACM, 1994.

[87] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, pages 492–501. ACM, 2006.

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

[89] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993.

[90] Robert McGill, John W Tukey, and Wayne A Larsen. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978.

[91] Audris Mockus, Roy T Fielding, and James Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 263–272. Acm, 2000.

[92] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 181–190. IEEE Computer Society, 2008.

[93] Mining Software Repositories (MSR) conference. `http://www.msrconf.org`.

[94] John C Munson and Sebastian G Elbaum. Code churn: A measure for estimating the impact of code change. In *Proceedings of the 14th International Conference on Software Maintenance (ICSM)*, pages 24–31. IEEE Computer Society, 1998.

[95] Paul Murrell. *R Graphics*. CRC Press, 2005.

[96] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference onSoftware Engineering (ICSE)*, pages 284–292. IEEE Computer Society, 2005.

[97] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 364–373. IEEE Computer Society, 2007.

[98] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 452–461. ACM, 2006.

[99] Martin E Nordberg III. Managing code ownership. *Software*, 20(2):26–33, 2003.

[100] Magnus C Ohlsson, Anneliese Von Mayrhauser, Brian McGuire, and Claes Wohlin. Code decay analysis of legacy software through successive releases. In *Proceedings of the Aerospace Conference*, volume 5, pages 69–81. IEEE Computer Society, 1999.

[101] Ralph Peters and Andy Zaidman. Evaluating the lifespan of code smells using software repository mining. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 411–416. IEEE Computer Society, 2012.

[102] Kristin Potter, Hans Hagen, Andreas Kerren, and Peter Dannenmann. Methods for presenting statistical information: The box plot. *Visualization of Large and Unstructured Data Sets, s*, 4:97–106, 2006.

[103] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[104] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM, 2011.

[105] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. Mining software evolution to predict refactoring. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 354–363. IEEE Computer Society, 2007.

[106] Juergen Rilling and Bhaskar Karanth. A hybrid program slicing framework. In *Proceedings of the 1st International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 12–23. IEEE Computer Society, 2001.

[107] Romain Robbes. Mining a change-based software repository. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, page 15. IEEE Computer Society, 2007.

[108] Peter J Rousseeuw, Ida Ruts, and John W Tukey. The bagplot: a bivariate boxplot. *The American Statistician*, 53(4):382–387, 1999.

[109] Deepayan Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer International Publishing, 2008.

[110] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *Transactions on Software Engineering*, 37(6):772–787, 2011.

[111] István Siket. *Szoftver termék metrikák alkalmazása a szoftverkarbantartás területén*. PhD thesis, SzTE, 2010.

[112] Sandra A Slaughter, Donald E Harter, and Mayuram S Krishnan. Evaluating the cost of software quality. *Communications of the ACM*, 41(8):67–73, 1998.

[113] YN Srikant and Priti Shankar. *The compiler design handbook: optimizations and machine code generation*. CRC Press, 2007.

[114] Ramanath Subramanyam and Mayuram S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *Transactions on Software Engineering*, 29(4):297–310, 2003.

[115] Attila Szegedi, Tamás Gergely, Arpad Beszedes, Tibor Gyimóthy, and Gabriella Toth. Verifying the concept of union slices on java programs. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 233–242. IEEE Computer Society, 2007.

[116] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[117] Filip Van Rysselberghe and Serge Demeyer. Mining version control systems for facs (frequently applied changes). In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR)*, pages 48–52, 2004.

[118] Guda A Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Notices*, volume 26, pages 107–119. ACM, 1991.

[119] László Vidács, Árpád Beszédes, and Tibor Gyimóthy. Combining preprocessor slicing with C/C++ language slicing. *Science of Computer Programming*, 74(7):399–413, 2009. Special Issue on Program Comprehension (ICPC 2008).

[120] László Vidács, Judit Jász, Árpád Beszédes, and Tibor Gyimóthy. Combining preprocessor slicing with c/c++ language slicing. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC)*, pages 163–171. IEEE Computer Society, 2008.

[121] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.

[122] Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.

[123] Hans Peter Wolf and Uni Bielefeld. *aplpack: Another Plot PACKage: stem.leaf, bagplot, faces, spin3R, plotsummary, plothulls, and some slider functions*, 2014. R package version 1.3.0.

[124] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[125] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 682–691. IEEE Press, 2013.

[126] Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. Predicting source code changes by mining change history. *Transactions on Software Engineering*, 30(9):574–586, 2004.

[127] Annie TT Ying and Martin P Robillard. The influence of the task on programmer behaviour. In *Proceedings of the 19th International Conference on Program Comprehension (ICPC)*, pages 31–40. IEEE Computer Society, 2011.

[128] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. Mining software repositories to study co-evolution of production & test code. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, pages 220–229. IEEE Computer Society, 2008.

[129] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *ACM SIGPLAN Notices*, volume 39, pages 94–106. ACM, 2004.

[130] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *Transactions on Software Engineering*, 31(6):429–445, 2005.