e-Informatica Software Engineering Journal, Volume 1, Issue 1, 2007

An Empirical Evaluation of Refactoring

Dirk Wilking*, Umar Farooq Khan*, Stefan Kowalewski*

*Embedded Software Laboratory, RWTH Aachen University

wilking@informatik.rwth-aachen.de, umar.khan@ixi.informatik.rwth-aachen.de, kowalewski@informatik.rwth-aachen.de

Abstract

This paper presents a process evaluation for the agile technique of refactoring based on the language C. The basis for this evaluation is made up by an experiment which is targeted on the aspects of increased maintainability and modifiability. Although the maintainability test shows a slight advantage for refactoring, results show no significant strength here. Concerning modifiability, the overhead of applying refactoring appears to even weaken other, positive effects. The analysis of secondary variables provides hints on advantages of the refactoring technique like reduced resource consumption and a reduced occurrence of complicated control structures.

1 Introduction

Maintenance of software is reported as a serious cost factor [24]. One solution proposed to reduce maintenance effort is refactoring [8] which is a method to continuous restructure code according to implicit micro design rules. Its new aspects are the smooth integration into an existing development process where it is used continuously in the background. Developers are forced to think about their code structure and to identify parts which "smell" - which is the best description that can be given for this subjective concept. After identification, the according code is changed based on a catalogue of change steps referring to the problem. These steps range from renaming of variables, extraction of methods to the extraction of complete classes from the existing code.

Refactoring is assumed to positively affect non-functional aspects, presumably extensibility, modularity, reusability, complexity, maintainability, and efficiency as stated in [24]. However, additional negative aspects of refactoring are reported, too. They consist of additional memory consumption, higher power consumption, longer execution time, and lower suitability for safety critical applications.

Most research concerning refactoring is done on the technical side in order to apply refactoring in a computer aided way. The general aim here is to either integrate a new technical aspect to refactoring like languages ([15, 21]), to support refactoring by a tool for analysis ([22, 30, 31, 35]) or to support the actual execution of refactoring ([10, 23]). Empirical evidence of the effect of refactoring is rarely to be found. One example for an empirical evaluation is the influence of refactoring on changeability as evaluated in [12] reporting a lower change effort. Other empirical results provide a taxonomy for bad smells as presented in [22].

Experience reports show a mixed picture of refactoring. One application of refactoring is reported to show non-satisfactory results [28]. It is reported that bad tool support along with the size of a legacy system created the problems. A code evolution analysis [19] investigates one of the main artifacts minimized by refactoring: copied code (code clones). It states that not every code clone should be subject to refactoring and that for some clones, appropriate refactorings are missing. One successful examination in terms of an increase in program performance due to refactoring is reported in [13]. A secondary, nonetheless interesting aspect mentioned there is the compliance to the design principle of information hiding after the application of refactoring.

Concerning agile methods in general, a limited empirical evaluation took place so far [2]. Most work has been done for pair programming [1] as this seems to be the most important aspect of extreme programming. As in addition refactoring is a major technique which can be used on its own, this report presents an experiment which intends to help assessing agile methods and this technique more precisely.

2 Design of the Experiment

The general approach followed by this experiment consisted of a group of 12 students using the same requirements specification to develop a program. Six students used refactoring continuously during development while the rest was asked to continuously document each function programmed. The assignment to a treatment group was done at random. The later treatment is regarded as a placebo in order to omit a Hawthorne effect [29] and to apply the same level of disturbance to this control group.

As the effects of refactoring were assumed to have an impact on non-functional aspects, two hypothesis were of special interest. The first one was the effect of refactoring on maintainability. Regarding this aspect, a direct evaluation method as proposed in [17] which is mainly based on a metric definition was not done. As in this case participants were available, a measurement with the help of the participants was done. Maintainability was tested by randomly inserting defects into the code and measuring the time needed to fix them (thus classified as corrective maintainability [3]). The second hypothesis was an improved modifiability caused by refactoring. In order to test this, small additions were added to the specification as new requirements and the time and physical lines of code (LOC) needed to implement them were measured.

2.1 Variables and Measurement

The independent variable of this experiment was the treatment which was a single, dichotomous factor. Either a participant was assigned to the refactoring or to the documentation treatment. In order to control the execution of the particular treatment, a simple tool was established disturbing every participant every 20 minutes. During each disturbance, the participant was asked to either work on a refactoring checklist or to document the last functions he programmed. In the case of documentation, changing the code was prohibited during this step.

One dependent variable of this experiment was the LOC metric together with the time to implement a new version based on additional features. LOC is considered to be a rough measure for the size of the resulting product. Both were used to measure the additional effort a developer needed to add new, unmentioned features to his code. These two thus were regarded as an indicator for system modifiability.

For maintainability, a special test was prepared. It consisted of a time measurement for the fixing task of randomly induced syntactical and semantical failures. These were directly created in the participants source code by randomly removing lines of source code. The tests consisted of a short description of the failure (in case of a semantical failure) and the measuring consisted of the time needed to locate and fix them. The measuring was done in seconds and supervised by a member of the chair.

A measurement of a difference in the abstract syntax tree is currently executed in order to assess a general difference in the micro structure of the different versions (cf. [14, 18, 20]).

2.2 Hypothesis

The main hypothesis of an improved modifiability for different versions measured by the time t was formalized by

$$H_0 : \bar{t}_{mod_{Ref}} \geq \bar{t}_{mod_{Doc}}$$

with $\bar{t}_{mod_{Ref}}$ being a version's mean development time for the refactoring group and $\bar{t}_{mod_{Doc}}$ being the according value for the documentation group. Thus, the resulting alternative hypothesis was

$$H_1 : \bar{t}_{mod_{Ref}} < \bar{t}_{mod_{Doc}}$$

Concerning corrective maintainability, the corresponding hypothesis was that the measured time for maintainability $\bar{t}_{main}$ during the maintainability test was greater for the documentation group leading to the null hypothesis of

$$H_0 : \bar{t}_{main_{Ref}} \geq \bar{t}_{main_{Doc}}.$$

The expected hypothesis thus was

$$H_1 : \bar{t}_{main_{Ref}} < \bar{t}_{main_{Doc}}.$$

2.3 Procedure

The execution of the experiment started with a video introduction explaining the microcontroller, the development environment, and the general conditions of the experiment. Only the last video was different for each participant group as it either explained the refactoring or documentation task. By using videos it was made sure that each participant received the same introduction and that no treatment group was favored. After that, an initial survey was carried out in order to assess the participant's overall programming knowledge and knowledge about refactoring. In order to avoid motivation effects refactoring was named reorganization within the documents and videos. Additionally, the

participants were asked to develop the software without any additional software engineering techniques to avoid interference of other factors. At last, the participants were not told what kind of measurement was done in the end in order to avoid preparation for requirement additions in terms of architecture.

The development started by reading the requirements document which was the same for all participants. After that, programming started until each requirement was implemented. The development task consisted of a game based on a reaction and a memorization part. The reason for this type of application was the low domain knowledge required. In addition, different types of hardware programming were needed in order to use the buttons (with debounce), LCD, LEDs and interrupts.

Concerning the execution of refactoring, only a subset of applicable refactoring steps was chosen with the addition of macro refactorings as discussed in [9] and [10]. The reason for excluding certain refactorings is the utilization of the language C. Only non-object oriented programming features were used during this experiment.

As the participants were not supposed to be accustomed to refactoring, a special, controlled execution was intended. First, the frequency of refactorings was set to a rate of 20 minutes. This was done to assure continuous refactoring together with a reminder of executing refactoring at all. The disadvantage of this approach are the occasions where a refactoring was initiated without the actual need for it. As the execution of refactoring steps was uncommon and the perception of bad smells was not based on participant experience, a checklist based on [8] was used in order to control both aspects. The execution of a refactoring is regarded non problematic whereas the detection of bad smells is subject to personal interpretation because of the nature of this term. Thus, only an informal description of this basic concept was given.

The final code size differed between individuals and was not affected by the treatment. The size ranged from 745 to 2214 lines of code. For each version, an acceptance test was executed checking the basic functionality and new features which were added. In case of an imprecise requirement definition, the implementation was accepted in the way the participant understood the requirement.

2.4 External Conditions and Limitations

The time span for this experiment was 3 months. During this time, all participants worked on the tasks until they finished them or the maximum of 40 hours was reached. Each participant worked in a different room and a simple room management was done as only three different rooms were available. The event that a participant wanted to work and no room available could be circumvented by this. Files were separated on network drives so that no participant could see the results of the other. The complete development environment was accessible in each room and participants worked on their own. Interruption sometimes occurred, but the frequency was not very high. For questions, an instant messaging server was setup and all messages were logged which was known and had to be accepted by the students.

2.5 Participants

The experiment was carried out with twelve graduate students. All of them were students at the RWTH Aachen University. The experiment had been advertised on the university's mailing list, notice boards and in the courses. Applications from 14 students had been received of whom 12 students had been selected randomly. Their field of study was mainly computer science, with one participant working in the field of mechanical engineering. All participants were paid and received a forty hour student helper contract. The students had programming knowledge of Java, whereas the language C was new to some. As mentioned above, refactoring was new to them except for one student who had practical knowledge.

As explained in [33], this type of participants is sufficient for evaluating basic effects or an initial hypothesis. In addition, [16] states that at least last-year software engineering students have a comparable assessment ability compared to professional developers and in [4] no general difference could be found for different programming expertise between these groups.

2.6 Technical Background

The experiment used an ATMEL ATmega16 microcontroller clocked with 6MHz as development platform. The software was written in C and developed with WINAVR 2 and ATMEL AVR Studio 3. For the LCD programming, an additional C-header was given to the students as this was regarded standard. Some tools were used in the background which comprised the disturber mentioned above and a code gathering tool which copied the code base every time a compilation was done. This last step was done in order to study code evolution.

3 Validity

This section critically examines practices and ancillary conditions. The procedure, measurements and theoretical concepts are structured as proposed in [36].

3.1 External Validity

Although in general students can be regarded as average programmers, they do not represent the often demanded professional developers. As stated above, they are regarded sufficient to show an effect within an initial method evaluation [33]. Regarding the treatment, the use of additional, unmentioned features can be regarded as in favor for refactoring. The event of changing requirements and thus the need for new features is not regarded artificial but normal industrial development. Regarding the environment, especially the lack of an object oriented language might have changed the influence of refactoring. This is not regarded as an artificial interaction because refactoring is regarded a method that can be applied in general to improve the design of a program. Technical factors like an exceptional good development environment or a method specific language might blur a method's effect and thus the lack of it is not regarded problematic.

3.2 Internal Validity

One of the major internal threats is the application of refactoring itself consisting mainly of a checklist and a periodical call for the application of it. This artificial treatment was chosen because of the high control. The downside of this is that the concept of a bad smell may require much more experience than provided by the checklist and that the application of refactoring may require a higher degree of freedom for an individual developer than allowed by such a list.

History and maturation are not regarded a threat as there is no repeat in the sense of reoccurring tasks or measurements except for the maintainability test were the code knowledge may have increased for each test. As an additional precaution, the main measurement tasks (additional requirements and failure inducing) were not known to the participants so that they could not prepare their code for this.

As some participants could not fulfill the requirements for all versions, they may have suffered from demoralization effects. But because of the fact that each participant worked on his own and no results where revealed to others, social threats are regarded a minor threat.

Concerning the communication between participants, only a contract specifying the participants duties and rights could be used as controlling device. As the development took a few weeks per participant, the possibility of private communication could not be eliminated.

3.3 Construct Validity

A clear theory in the sense of an abstraction of the effects is not easy to define for refactoring. There are several effects which are accumulated in the term refactoring. One of the major points is the abstract design principle of “once and only once” suggested by the inherent term “factor” [8]. Another effect might be the constant rereading and rethinking of existing code. By this, a continuous awareness of all parts of the source code might by achieved revealing positive effects like simpler reuse of code and faster navigation. This might be considered a constant reviewing process, too. One last aspect is an implicit effect of refactoring with the existence of a good structure being indirectly postulated. This effect may force developers to maintain a certain quality for every part of the code which may not be the case for non-refactoring based development.

However, by following the combined approach of bad smell and collection of refactoring steps, the common usage of this technique is adopted and its general influence assessed. Consequently, an abstract construct was not used.

Concerning the outcome expected to be caused by refactoring, only the variables of maintainability and modifiability were measured. For other non-functional aspects like modularity, reusability, complexity, and efficiency no direct measurement construct could be found. The quality of the actual measurement consisted only of a single variable for maintainability, whereas multiple measurements in the form of LOC and time were done for modifiability.

The generalizability of the treatment suffers from the hard 20 minute interrupts. One the one hand, as refactoring is executed on demand when a problem has been discovered, this treatment is artificial. On the other hand, it is the only way of assuring a constant execution. In addition, the reminder of looking for bad code aspects is regarded helpful for unexperienced participants. The general idea of changing bad code continuously thus is considered as maintained.

3.4 Conclusion Validity

Concerning the experiment's power, the low number of participants ($n = 6$) is a problematic point. Power is described in [5] as the probability of rejecting the null hypothesis and thus directly describes how good the experiment can show an effect. As the importance of that aspect may be exaggerated (compare [26] to [25]) given the quality of variables for empirical software engineering, the value for n still is too low. As described in [7], a bootstrap power calculation can be done by sampling (with replacement) a higher number of participants based on the original data. Table 1 depicts the probability p of showing a difference of the mean fixing time of 12 seconds or more. This can be regarded a rough indicator, as a only point estimator is used and 12 seconds is a rather low difference (five percents regarding the mean fixing time of 240 seconds). Regarding a refactoring group size of 48 participants, the experiment starts to have an appropriate probability of showing the expected effect. An interesting application of this sample size oriented power calculation is proposed in [32] suggesting a continuous review of an experiment's power.

N:	6	12	24	48
$p(d \geq 12)$	0.68	0.74	0.83	0.91

Table 1: Power calculation of a difference in means of 12 seconds for different sample sizes N

As the hypothesis and the assumed effects of refactoring have been clearly stated, "fishing for results" may only occur for secondary variables for this experiment. Nevertheless, these variables are investigated and interpreted, as they may give ideas for other effects caused by refactoring. Their unreliable nature (significant results cannot be regarded as such) is emphasized in the text.

The reliability of the measures is difficult to assess. LOC is always a point of discussion, but it nevertheless can be regarded a rough measure for system size. The measurement of relative time (compared to the first, full version) used to assess modifiability has the advantage that it includes the main benefit expected for refactoring: a decrease of effort when adding features. In addition, this variable is simple to measure. The special test for maintainability which randomly induces failures into the participant's code simulates the same effect as a real case of corrective maintainability: a system failure is reported, its cause has to be found in the code and it has to be fixed. Its reliability is regarded above average as time is used as main variable and the failure creation is based on a random process.

4 Analysis

4.1 Main Hypothesis

4.1.1 Maintainability

The measurement of maintainability, which consisted of a random insertion of 15 syntactical and 10 non-syntactical errors, was measured in seconds. The errors were created by removing lines of code randomly. The resulting error were divided into syntactical and non-syntactical nature. Because of the randomization and the rather uncommon test method, a more detailed rating of the severeness of an error was note feasible. The results were gathered for all twelve participants and the corresponding mean error correction times were aggregated into the box plot of figure 1. Here, a minor advantage for the refactoring treatment can be seen, but the results were not significant when a bootstrap test was executed for $\alpha = 0.05$. The assumption of better maintainability thus cannot be answered according to this, but the slightly lower value for the refactoring treatment lead to the impression of only a minor effect of refactoring.

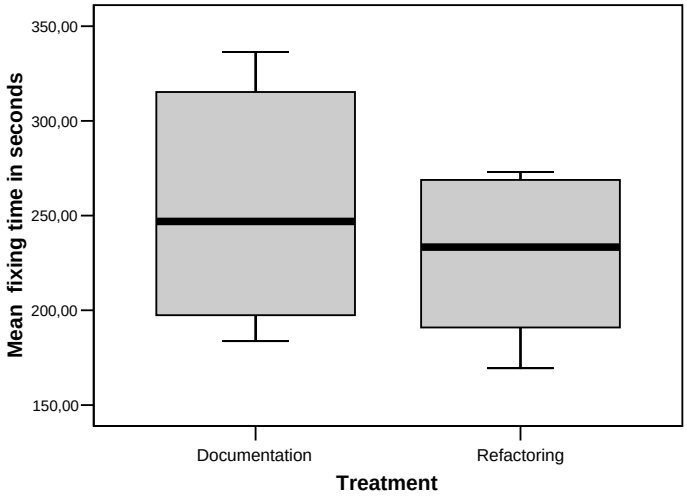

Figure 1: Box plot of mean fixing time of each participant divided by treatment group, 6 data points per group

4.1.2 Modifiability

Concerning modifiability, the measurement consisted of an additional implementation of minor, new requirements added to the main task. The effect of each addition was evaluated by counting the lines of code that were added, changed, and deleted for a version and by measuring the time needed to fulfill the new requirements. It must be noted that due to the different performance of the participants only 10 results were included for version 1.1, and only 9 participants could be included for version 1.2 and 1.3.

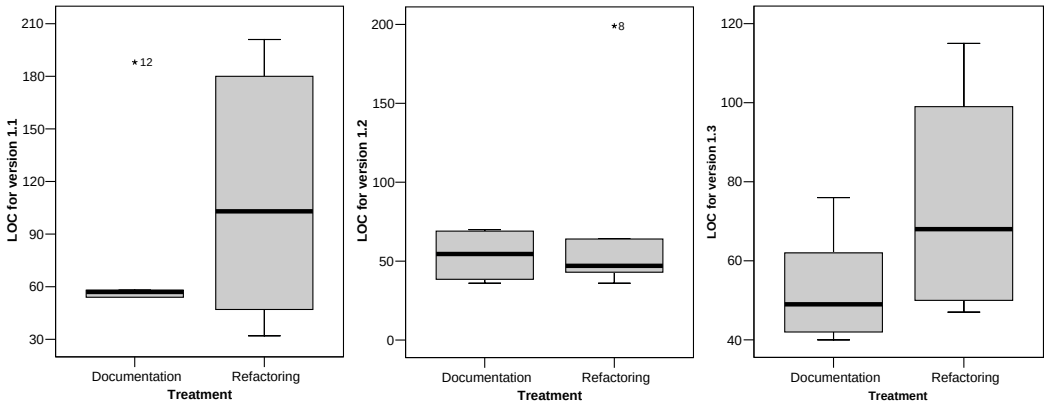

Figure 2: Box plots for changed LOC per version categorized by treatment, 6 data points per treatment

Figure 2 displays the change needed for each development version. Changing incorporates the actions of addition, deletion and modification of a line of source code. Concerning the difference in LOC, it becomes obvious that the refactoring treatment contradicts the initial assumption that refactoring has a benefit on system modifiability. The median of the changed lines for the refactoring group is above that of the control group in two cases.

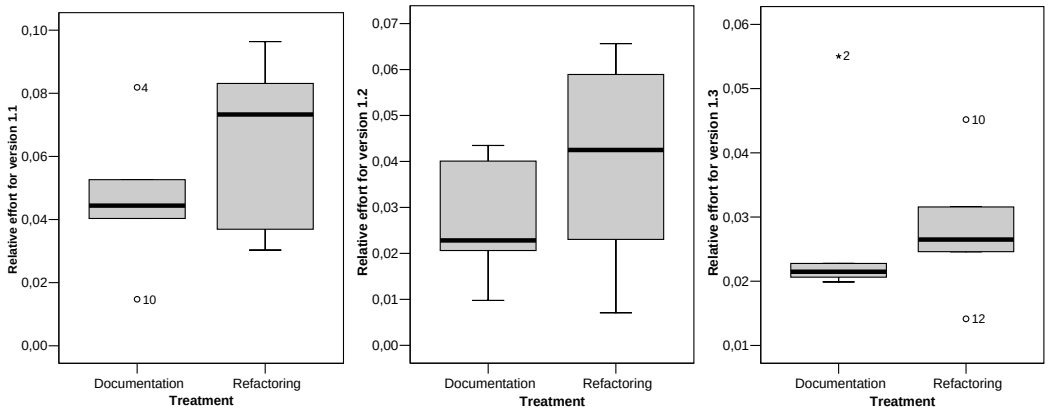

Figure 3: Box plots for fraction of development time compared to first version per modification categorized by treatment, 6 data points per treatment

The observation of figure 2 from above is supported by the time measurement for each treatment group as presented in figure 3. Although these two variables are linked together (more lines of code will take longer to write), the overall impression of additional effort for refactoring is strengthened. In this case, refactoring has a bad effect on all three versions.

Regarding the main hypotheses of better maintainable systems and a better overall modifiability, these results could not show an effect in favor of these non-functional aspects,

but rather give a hint on strong side effects of refactoring. An additional interpretation of these main results is given in section 5.

4.2 Analysis of Secondary Variables

Regarding the execution of the experiment, additional variables were measured during the programming procedure. One interesting aspect was the memory usage of a program which was reported by the compiler after each compile cycle. The memory types for the system consisted of SRAM and flash-RAM. SRAM is used for heap memory allocation, as stack memory and for initialized and non-initialized data fields. Flash is mainly used as program memory, meaning that the text of a program is stored here.

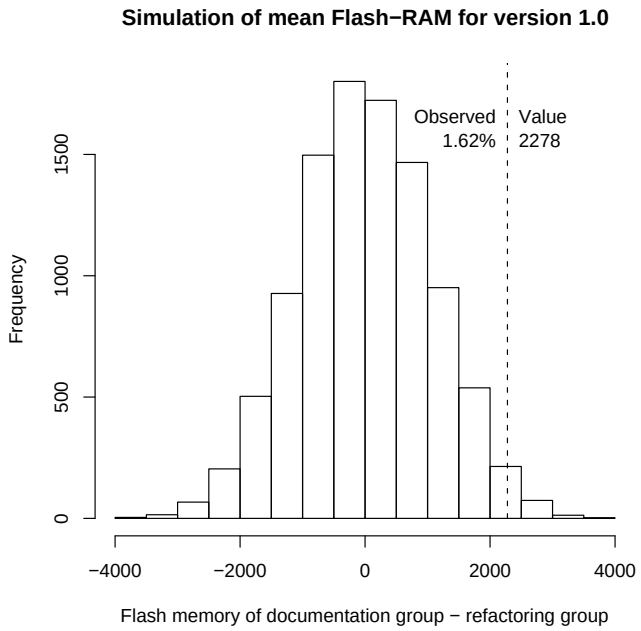

Figure 4: Bootstrap simulation of mean memory difference

When comparing the results of both groups, the difference of the mean flash memory usage for both groups was significantly different. While the documentation group needed less SRAM, the usage of flash was higher as shown by the bootstrap simulation [6] for the mean memory usage of both groups of figure 4. The simulation compares the observed memory difference to differences created by randomized groups. It starts by randomly dividing the observed memory values in two groups of the same size as in the original experiment. From each of these groups, the mean value is calculated and the values are subtracted. This is repeated 10000 times and the results are given in form of the histogram in figure 4. The original value of 2278 is rarely observed (only 1.62%) which can be interpreted as a non-random occuring event. Thus, memory consumption might

be effected by refactoring. The implication of this observation and possible causes for this difference are explained in section 5.

An advantage of bootstrap is that for randomization of groups, the values measured during the experiment are taken. Thus, it reuses (bootstraps) its own data to compare the values to a more problem specific population. Compared to t-test and u-test, assumptions concerning the distribution of the data are lower making it more usable for smaller experimental groups [34].

4.3 Analysis of Refactoring Techniques

Another data source originates from the checklists of the participants. Here, each time a student was disturbed, the refactoring techniques applied during the process had to be checked. Based on the frequency of usage, a ranking of the importance for each refactoring technique could be created as shown in figure 5.

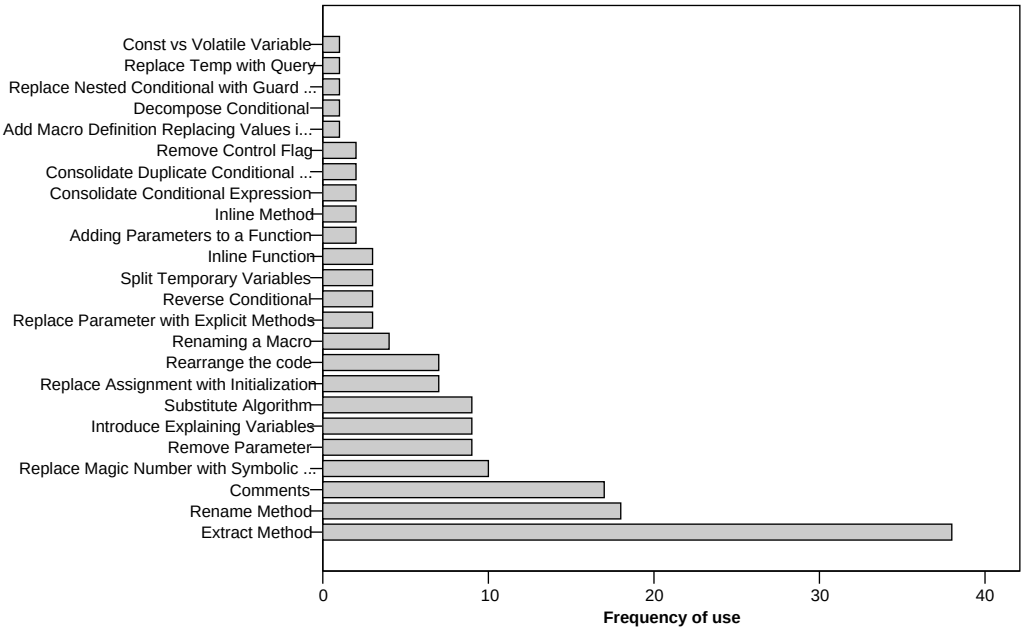

Figure 5: Accumulated occurrences of refactoring techniques for 6 participants

Regarding the techniques used, only some techniques may be of importance for average programming tasks. The extract method principle of aggressively dividing code blocks into smaller chunks appears to be by far the most important technique when refactoring is applied. Additionally, better naming schemes for methods appear to be important which might be understood as a change on the semantic level of the code. The addition of code comments seems to head for the same goal by giving a better explanation for blocks of source code. One single refactoring may have a high ranking only because of

the C language programming: replacing a magic number with a symbolic constant. Here again, a better explanation seems to be the aim of the refactoring technique.

This refactoring list may give hints on tool support for refactoring. One problem with this list is that even the most important technique (extract method) will be difficult to implement, as it requires syntactical knowledge of the source code for the according tool in oder to do the refactoring. This is regarded non standard for most source code editors.

4.3.1 Difference in Metrics

In order to describe the structural change that is caused by refactoring, the McCabe metric of cyclomatic complexity is used. Figure 6 shows the cyclomatic complexity plotted against the according lines of code for each function and each treatment group. While the midpoints of both groups do not differ, the number of functions with high cyclomatic complexity appears to be higher for the documentation group. About 11% of the functions created in the control group had a complexity of more than 10, while only 3% of the functions created with active refactoring had a higher value than 10. This may be a hint on the principle of simple design constituting one of the goals of refactoring.

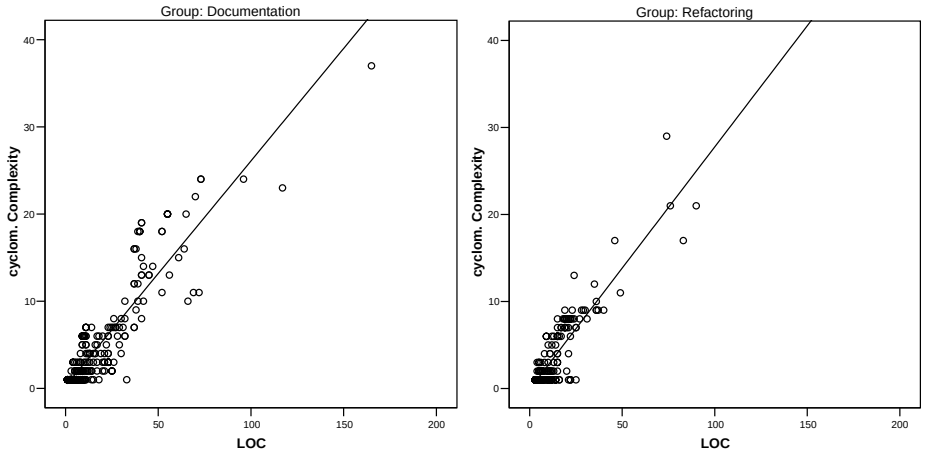

Figure 6: Cylomatic complexity versus LOC scatter plot for functions of both groups

5 Interpretation

The direct effect of an increased maintainability and a better modifiability caused by refactoring could not be shown within this experiment. Although rigid control of the application of refactoring techniques took place, the resulting system did not seem to have a better structure in terms of ability to understand the structure faster for maintainability. Modifiability, which might benefit from the idea of “once and only once”, and simplicity, did not seem be of significantly different, either. Instead, the results rather hint to an overhead when refactoring is applied leading to actually more effort when new requirements

are added to a system. The question arising from that overhead is if the accumulated time needed for refactoring pays off in bigger systems with more complex architectural aspects. For short projects, the probable benefit of refactoring may reveal itself too late and the resulting overhead may be a waste of time. For long projects, refactoring may have a more positive effect.

Regarding other variables measured during this experiment, the aspect of lower memory usage for program memory is a positive side effect. The basic principle of "once and only once" directly pays off as similar code is reused more often or, in other words, copied code for similar programing tasks is omitted. As this was not part of any hypothesis, this observation has to be regarded carefully.

The main criticism regarding this experiment is its size. The time frame of 40 hours is more than in other experiments, but not sufficient in terms of process assessment. The number of 12 participants is low, too, but as the modifiability results point into the opposite direction, the length of the experiment is regarded more problematic. One other source of criticism might be the use of refactoring without unit tests ([11, 27]). As this can be regarded a major technique to control side effects when a refactoring is executed, it is most often regarded a necessary addition to refactoring. It was omitted, because of the effect this kind of testing might have on software development. Its application would have made an evaluation of refactoring as a single factor more difficult.

One last argument against the experiment is that expert developers would constitute a much better evaluation basis. Their knowledge concerning better system design and areas of "smells" might increase the effect of refactoring. This argument is somewhat misleading, as first of all the effect of refactoring should occur even in the case of average programmers. In addition, using experts in the sense of 1% of available developers appears as an unrealistic modification compared to normal software development.

6 Conclusions and Future Works

6.1 Future Works

Regarding the long term effect of refactoring, a more indirect approach may be beneficial in the future. For example, instead of the execution of refactoring, the effects countered by refactoring might be subject to investigation. The habit "of copy and paste code" may be regarded as development laziness. If the occurrence of this behavior could be shown, the negative effects on the system might be measured leading to an indirect justification for refactoring. Another point closely related to the general application of agile methods is whether the first development solution found is the optimal one in terms of simplicity, understandability, performance, future-applicability and so on. If shortcomings in this area can be shown, refactoring might be the technique to give an increase in these variables. Another starting point for research is the underlying aim of refactoring: what are the reasons to change code, when to change it and, ultimately, do these reasons accumulate for a given code basis? This would need a formalization of the subjective term "smell".

A rather different approach more related to code evolution or metrics as proposed in [17] is done on an additional data source collected during the experiment. It consists of

the complete code basis of every participant which was saved every time a compilation was executed. The idea here is to analyze the abstract syntax tree of the code in order to find the cause for a specific refactoring.

6.2 Conclusions

In this paper, a controlled experiment is presented assessing the effect of refactoring on non-functional aspects. However, a general effect of refactoring on maintainability or modifiability could not be shown. Instead, an overhead for the modifiability aspect seems to exists as refactoring itself needs a certain amount of time for its execution. A positive aspect of refactoring might be found in the "once and only once" design principle, as this seems to reduce the memory requirements of a system. As an addition, the three most important refactoring found during this experiment appear to be "extract method", "rename method", and "comments" which might be a starting point for basic refactoring support in software tools. In addition, a different approach to assess the importance of refactoring is presented focusing on indirect assumptions of why refactoring is applied and what problems it might solve.

References

[1] P. Abrahamsson and J. Koskela. Extreme Programming: A Survey of Empirical Data from a Controlled Case Study. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE04)*, 2004.

[2] P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen. New directions on agile methods: a comparative analysis. *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 244–254, 2003.

[3] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett. Understanding and Predicting the Process of Software Maintenance Releases. In *Proceedings of the 18th International Conference on Software Engineering*, 1996.

[4] J.-M. Burkhardt, F. Deétienne, and S. Wiedenbeck. Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase. *Empirical Software Engineering*, 7:115–156, 2002.

[5] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Lawrence Erlbaum Associates, 1988.

[6] A. C. Davison. *Bootstrap Methods and their Application.* Cambridge University Press, 1997.

[7] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap.* Chapman & Hall/CRC, 1998.

[8] M. Fowler. *Refactoring. Improving the Design of Existing Code.* Addison Wesley, 1999.

[9] A. Garrido and R. Johnson. Challenges of refactoring c programs. *IWPSE: International Workshop on Principles of Software Evolution*, 2002.

[10] A. Garrido and R. Johnson. Refactoring c with conditional compilation. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, Montreal, Canada, 2003.

[11] B. Georgea and L. Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46:337–342, 2004.

[12] B. Geppert, A. Mockus, and F. Rößler. Refactoring for changeability: A way to go? In *11th IEEE International Software Metrics Symposium (METRICS 2005)*.

[13] B. Geppert and F. Rosler. Effects of refactoring legacy protocol implementations: A case study. In *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium on (METRICS'04)*, pages 14–25, Washington, DC, USA, 2004. IEEE Computer Society.

[14] D. M. German. An empirical study of fine-grained software modifications. In *20th IEEE International Conference on Software Maintenance, 2004*, 2004.

[15] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, 2004.

[16] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects – a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5:201–214, 2000.

[17] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance (ICSM02)*, 2002.

[18] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[19] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, New York, NY, USA, 2005. ACM Press.

[20] R. Leitch and E. Stroulia. Understanding the economics of refactoring. In *The 7th International Workshop on Economics-Driven Software Engineering Research*, 2005.

[21] H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 27–38, New York, NY, USA, 2003. ACM Press.

[22] M. Mäntylä, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance (ICSM03)*, 2003.

[23] B. McCloskey and E. Brewer. Astec: a new approach to refactoring c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 21–30, New York, NY, USA, 2005. ACM Press.

[24] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2), 2004.

[25] J. Miller. Statistical significance testing – a panacea for software technology experiments? *The Journal of Systems and Software*, 73:183–192, 2004.

[26] J. Miller, J. Daly, M. Wood, M. Roper, and A. Brooks. Statistical power and its subcomponents – missing and misunderstood concepts in empirical software engineering research. *Journal of Information and Software Technology*, 1997.

[27] M. Müller and O. Hagner. Experiment about test-first programming. *IEE Proceedings Software*, 149(5):131–136, October 2002.

[28] M. Pizka. Straightening spaghetti code with refactoring? *Software Engineering Research and Practice*, 2004.

[29] L. Prechelt. *Kontrollierte Experimente in der Softwaretechnik*. Springer, 2001.

[30] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, New York, NY, USA, 2005. ACM Press.

[31] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, 2001.

[32] J. L. Simon. *Resampling: The New Statistics*. Resampling Stats, 1999.

[33] D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanović, and M. Vokáč. Challenges and recommendations when increasing the realism of controlled software engineering experiments. *ESERNET 2001–2003, LNCS 2765*, pages 24–38, 2003.

[34] J. B. Todman and P. Dugard. *Single-Case and Small-N Experimental Designs: A Practical Guide to Randomization Tests*. Lawrence Erlbaum Associates, 2000.

[35] B. Walter and B. Pietrzak. Multi-criteria detection of bad smells in code with uta method. In H. Baumeister, M. Marchesi, and M. Holcombe, editors, *Extreme Programming and Agile Processes in Software Engineering, XP 2005*, Sheffield, UK, 2005. Springer.

[36] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering. An Introduction.* Kluwer Academic Publishers, 2000.