A Case Study of Refactoring Large-Scale Industrial Systems to Efficiently Improve Source Code Quality

Gábor Szőke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy

Department of Software Engineering, University of Szeged

Abstract. Refactoring source code has many benefits (e.g. improving maintainability, robustness and source code quality), but it takes time away from other implementation tasks, resulting in developers neglecting refactoring steps during the development process. But what happens when they know that the quality of their source code needs to be improved and they can get the extra time and money to refactor the code? What will they do? What will they consider the most important for improving source code quality? What sort of issues will they address first or last and how will they solve them? In our paper, we look for answers to these questions in a case study of refactoring large-scale industrial systems where developers participated in a project to improve the quality of their software systems. We collected empirical data of over a thousand refactoring patches for 5 systems with over 5 million lines of code in total, and we found that developers really optimized the refactoring process to significantly improve the quality of these systems.

Keywords: software engineering, refactoring, software quality

1 Introduction

With short deadlines or lack of resources, developers tend to neglect refactoring steps during development and if they see a quick and easy way to get a test working and a ten-minute way to get it working with a simpler design, they will go for the quicker way, although the correct choice should be to spend ten minutes on refactoring. This usually results in the deterioration of the software. One way to combat this deterioration is to continuously re-engineer the code. Continuous reengineering is not only mentioned by popular development principles such as eXtreme programming [3], but the software engineering community realized that instead of spending money on maintenance tasks periodically it may be cheaper and more effective to continuously maintain the code and check its quality. For instance, Demeyer et al. say in [5] that "there is good evidence to support the notion that a culture of continuous reengineering is necessary to obtain healthy, maintainable software systems."

In our paper, we investigate how programmers re-engineer their code base if they have the time and extra money to improve the quality of their software systems. In a project we worked together with five companies where one of the goals was to improve the quality of some systems being developed by them. It was

interesting to see how these companies optimized their efforts to achieve the best quality improvements at the end of the project. They are all profit-orientated companies, so they really tried to get the best ROI in terms of software quality. To achieve it, they had to make important decisions on what, where, when and how to re-engineer. We collected this information as experimental data and here we present our evaluation in the form of a case study. We found that developers really optimized the refactoring process to improve the quality of these systems; they usually went for the most critical but least risky types of refactorings. The results presented in this study could serve as a guideline for designing a re-engineering process.

The main contributions of this paper are:

- A case study on software refactorings with experimental data gathered from re-engineering large-scale proprietary software systems.
- Guidelines to re-engineer large-scale projects effectively.

The paper is organized as follows. In the next section we present related research work and then in Section 3 we introduce the motivational background of our case study. After, in Section 4 we present our results by answering our research questions. We discuss threats to validity and other results we got in Section 5 and finally we conclude the paper.

2 Related work

Refactoring has been a hot topic since the appearance of Fowler's book [10] and Opdyke's PhD thesis [15]. There are many papers published in this area and it is not the aim here to systematically summarize these studies. In this section, we will give a general overview of software refactoring, and present some case studies which are similar to ours.

Mens et al. published a survey to provide an extensive overview of existing research in the area of software refactoring [12]. They identified six main refactoring activities. These are:

- Identifying precisely where the software should be refactored
- Determining which refactoring(s) should be applied to the places identified
- Ensuring that the refactoring applied preserves behaviour.
- Appling the refactoring
- Assessing the effect of the refactoring on quality characteristics of the software (e.g. complexity, understandability and maintainability) or the process (e.g. productivity and cost effort).
- Maintaining a consistency between the refactored program code and other software artifacts such as documentation, design documents, software requirements specifications and tests

Our study can be viewed as a piece of research work which attempts to support decisions on the first five activities.

Many papers have been published on where and how software code should be refactored – e.g. by applying automatic tools to identify bad smells [2,11], change

smells [17] and by using static rule checkers such as CheckStyle*, FindBugs† and PMD‡ for Java. Code clones may be regarded as a special type of bad smells and they are also typical targets of refactorings [4,20,19].

To determine which refactoring(s) should be applied, most of the studies investigate the effects of refactorings on metrics or quality attributes. Alshayeb et al. studied how refactoring improves external quality attributes such as adaptability and maintainability [1]. Stroggylos et al. analyzed source code version control system logs of popular open source software systems to detect changes marked as refactorings and examine how the software metrics are affected by this process [18]. DuBois et al. studied the impact of refactorings on cohesion and coupling metrics in [7] and found that benefits can occur, and described how and when the application of refactoring could improve selected quality characteristics [6]. Fontana et al. studied the impact of refactoring applied to reduce code smells on the quality evaluation of the system [9].

Murphy et al. studied four methods to collect empirical data on refactorings [14]: mining the commit log, analyzing code histories, observing programmers and logging refactoring tool use. In our study, we combine these methods. A similar study was conducted by Moser et al. [13] as they observed small teams working in similar, highly volatile domains and assessed the impact of refactoring in a close-to industrial environment [13]. Pinto et al. investigated what programmers said about refactoring on the popular Stack Overflow site [16].

3 Background

3.1 Project

The research work presented here formed part of an EU project. The main goal of the project was to develop a software refactoring framework, methodology and software products to support the 'continuous reengineering' methodology, hence provide support to identify critical code parts in a system and to restructure them to enhance maintainability. During the project, we developed an automatic/semi-automatic refactoring framework and tested this technology on the source code of industrial partners, having an in-vivo environment and live feedback on the tools. So partners not only participated in this project to develop the refactoring framework, but they also tested the tool set on the source code of their own product. This provided a good chance for them to refactor their own code and improve its quality.

In the initial step of the project we asked them to manually refactor their own code, and provide a detailed documentation of each refactoring, explaining what they did and why to improve the targeted code fragment. We gave them support by continuously monitoring their code base and automatically identifying problematic code parts using a static code analyzer based on the Columbus technology of the University of Szeged [8], namely the SourceMeter product of

*http://checkstyle.sourceforge.net/

†http://findbugs.sourceforge.net/

‡http://pmd.sourceforge.net/

FrontEndART Ltd.[§] Companies had to fill in a survey with questions targeting the initial identification of steps; that is, evaluating the reports of SourceMeter looking for really problematic code fragments and explaining in the survey why that code part was actually a good target for refactoring. After identifying coding issues, they refactored each issue one-by-one and filled out another questionnaire for each refactoring, to summarize their experiences after improving the code fragment. There were around 40 developers involved in the project (5-10 on average from each company) who were asked to fill in the survey and carry out the refactorings.

3.2 Survey questions

The survey consisted of two parts for each issue. The developers had to fill in the first part before they began refactoring the code, and the second part after the refactoring. In the first part, they asked the following questions:

- Which PMD rule violations helped you identify the issue?
- Which Bad Smells helped you to find the issue?
- Estimate how much it would take to refactor the problem.
- How big is the risk in carrying out the refactoring? (1-5)
- How do you think the refactoring will improve the quality of the whole system's code? (1-5)
- How do you think the refactoring will improve the quality of the current local code segment? (1-5)
- How much improvement do you think the refactoring will make to the current code segment? (1-5)
- How many files will the refactoring have an impact on?
- How many classes will the refactoring have an impact on?
- How many methods will the refactoring have an impact on?

We asked some questions after developers had finished the refactoring task. These were the following:

- Which PMD rule violations did the refactoring fix?
- Which Bad Smells did the refactoring fix?
- How much time did the refactoring task take?
- Did any automated solution help you to fix the problem?
- How much of the fix for this problem could be automated? (1-5)

For most of the questions, we provided some basic options. For the first question for example we provided a list of PMD rule violations with their names, to help the developers answer the questions quickly. In the questions on the classes and methods impacted, we provided different ranges, namely 1-5, 5-10, 10-25, 25-50, 50-100, 100+. Each question had a text field where the developers could explain their answers and they could also suggest possible improvements and add comments.

[§] http://frontendart.com

3.3 Systems under investigation

In the study, we had chance to work together with five experienced companies in the ICT sector. These companies were founded in the last two decades and some of their projects were initiated before the millennium. The 5 given projects consisted of about 5 million lines of code altogether, written mostly in Java. The projects covered different ICT areas like ERPs, ICMS and online PDF Generation. More details can be found in Table 1.

Id	LOC	Domain
Company A	200k	Specific Business Solutions
Company B	4,300k	Enterprise Resource Planning (ERP)
Company C	170k	Integrated Business Management
Company D	128k	Integrated Collection Management Systems (ICMS)
Company E	100k	Web-based PDF Generation

Table 1. Systems that we examined

Each project had Web/online modules and some of them could run as standalone applications too. Companies A and B commenced their projects with the first releases of Java Enterprise Edition. At that time there were no application frameworks (like SpringFramework) available, so they implemented their own versions. Therefore the core of their systems can be regarded as legacy Java systems, but still under active development.

4 Case study

4.1 RQ1: What kinds of issues did the companies find most reasonable to refactor?

Our first research question focused on which issue types the companies considered the most important to refactor. We asked the companies which indicators helped them best in finding problematic code fragments in their systems. In our survey, companies could select Bad Code Smells and Rule Violations as indicators on how they found the issues.

In our evaluation, we distinguish a special kind of bad smell which suggests code clones in the system. In Figure 1, a distribution can be seen for the issues which helped the companies to identify the problematic code fragments in their code. The intersections in the figure came from the fact that developers could select more than one indicator per issue. The reason why bad smells and clones had no elements in their intersection was because a clone is a special kind of bad smell, as mentioned earlier. The same applies for the intersection of the former group and the rules group (an empty set cannot intersect anything).

When we look at the results in Figure 1, we see that the companies found the majority of issues lay in the sets of rule violations and bad smells. It can also be seen that rule violations alone cover 85% of all the issues found. This also includes 75% of all the bad smells (because of the intersection). So the assumption is that rule violations are the best candidates for highlighting issues.

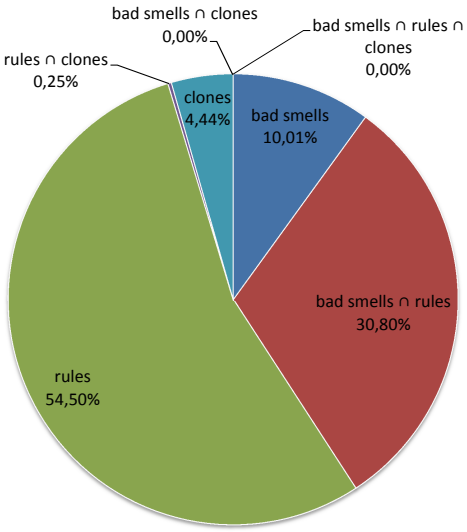

Fig. 1. Distribution of issue indicators

However to confirm this, we also had to look at how many issues the companies fixed in order to choose the best indicator of refactorings.

Figure 2 shows the percentage of each fixed issue found from our survey. When we examine the ratio of fixed issues, we see that the bad smells are mostly refactored issues. However if we include the total number of issues, it is clear that rule violations gave the most advance.

Based on the fact that 85% of all issues were rule violations and developers mostly fixed these issues instead of the others, in future RQs we will focus on rule violations.

4.2 RQ2: What are those attributes of refactorings that can help in selecting them?

The rule violations in the survey were provided by the *PMD source code analyzer* tool. In our study, we categorized and aggregated these rules into groups. The groups we used were the Rulesets taken from the PMD website. The companies filled in the survey for 961 PMD refactorings altogether. These 961 refactorings produced 71 different rule violation types over 19 rulesets.

Below, we will examine these rulesets based on different attributes. Based on our survey questions, we created the following attributes:

- **number of refactorings** indicates how many issues were fixed for a certain kind of PMD or ruleset.
- **average and total time required** tells us the total and average time that companies spent on a refactoring. (Values are in work hours.)
- **estimated time** shows how companies estimated the time that a refactoring operation would take. (Values were enumerated between 1 hour and 3-4 days.)

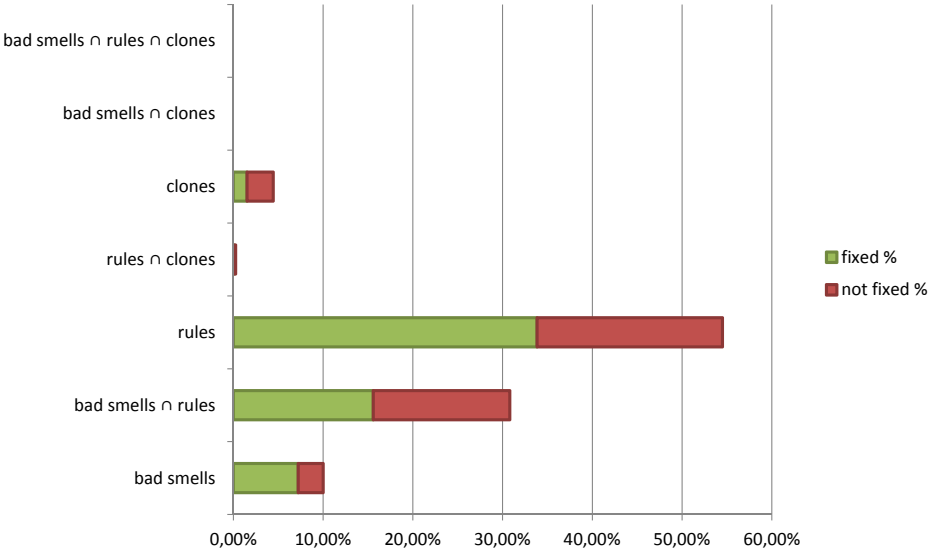

Fig. 2. Percentages of fixed issues for different problem types

- **local improvement** indicates the subjective opinion of developers on how much the local code segment was improved by the refactoring (Values are between 1-5.)
- **global improvement** indicates the subjective opinion of developers on how much the code improved globally. (Values are between 1-5.)
- **risk** indicates the subjective opinion of developers on how risky the refactoring is. (Values are between 1-5.)
- **impact** is an aggregated number that tells us how many files/classes/methods a refactoring affected. (Values are enumerated between 1-100.)
- **priority** tells us how dangerous a rule violation is, and how important it is to fix it. The priority attribute did not come from the survey; we used the prioritisation of the underlying toolchain. (Values lie between 1-3.)

4.3 RQ3: Which refactoring operations were the most desirable based on to the attributes defined above?

The attributes above tell us how risky a refactoring operation is and how much time it will usually take to fix. By combining these attributes, we can discover which rules or rulesets are the most beneficial or riskiest; or by aggregating the first two attributes with time required, we can see which rules will best return the effort we invested in refactoring. Next, we investigate the number of refactorings, time required, improvement and risk.

Number of refactorings Now let us look into the most obvious attribute, namely the number of refactorings the companies performed. The results in Figure 3 indicate that the companies dealt with almost every kind of rule violation. The majority of refactored rule violations were found in the *Design* ruleset. This

ruleset contains rules that flag suboptimal code implementations, so fixing these code fragments should significantly improve the software quality and perhaps even the performance. The *Design* ruleset is followed by the *Strict Exceptions*, *Unused Code* and *Braces* categories, which focuses on throwing and catching exceptions correctly, removing unused or ineffective code, and also the use and placement of braces. Some rule violations in the following categories were also fixed in large numbers under the *Basic, Migration, Optimization, String and StringBuffer* rulesets. The other rulesets scarcely came up (like *Empty Code*) or not at all (like *Android*). This is probably due to the fact that the projects did not contain these kinds of violations or contained only false positives.

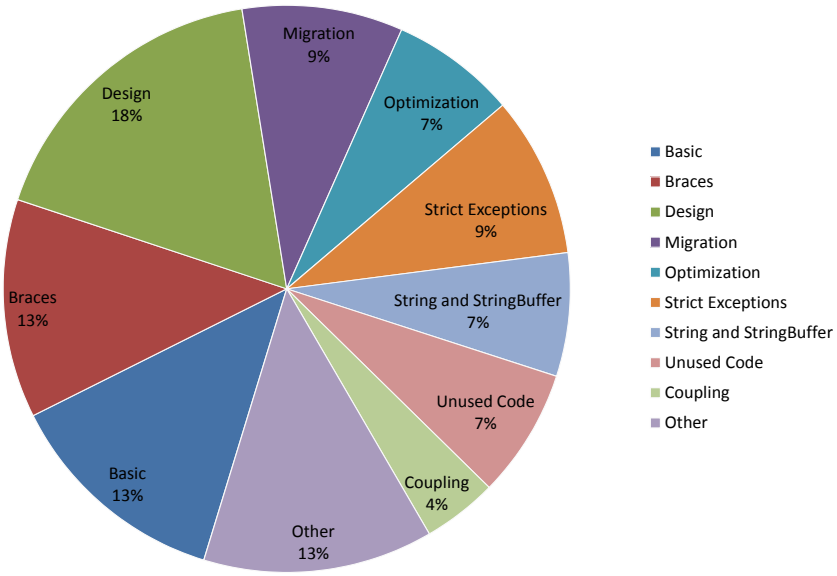

Fig. 3. Distribution of refactorings by PMD rulesets

Average and total time required After investigating how many refactorings the companies made, we will now examine how much time a refactoring operation took. (Here, we consider the time the developers spent on refactoring their source code, excluding the time they spent on testing and verifying the code.)

When we look at the total time needed for the categories in Figure 4, we see that the time distribution of the refactorings shows a similar tendency as the number of refactorings. A linear correlation can be seen between the number of refactorings and the total time spent on them. However, other interesting things were observed when we looked at the average time spent on the different kinds of PMD categories in Figure 5. It seems as if the companies spent most of the time on average on *Code Size*, *Security Code Guidelines* and *Optimization* rules. The least time was spent on average on Braces, *Import Statements* and *Java Beans* rules (excluding those rules where no time was spent at all). The *Code*

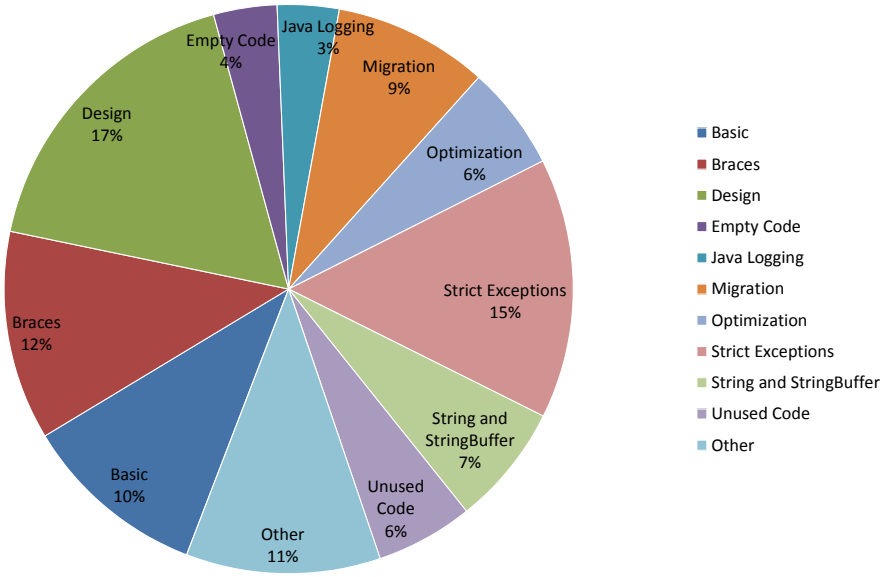

Fig. 4. Total refactoring durations by PMD rulesets

Size ruleset contains rules that relate to code size or complexity (e.g. CyclomaticComplexity, NPathComplexity), while the *Security Code Guidelines* rules check the security guidelines defined by Oracle. The latter guidelines describe violations like exposing internal arrays or storing the arrays directly. *Optimization* rules concern different optimizations that generally apply to best practices. Reducing the complexity of the code, making the application more robust or optimizing it takes time. Apparently, these take the most time. Removing unused import statements or adding or removing some braces usually can be performed quickly, but to find which independent statements to extract so as to reduce the complexity is a hard task.

Global and local improvement To learn which PMD rule violations fit the attributes best, we summarized and averaged both the global and local improvement values got from the survey. We ranked both sets of values by their position in their data set. The average of the two former values gave us a list of the best improving PMD rulesets. From our results, the best improvements locally and globally are given by the *Strict Exceptions, Coupling and Basic* PMD rulesets. However, rulesets contain a lot of different rules, and hence the categories alone did not give us the proper information we sought. To get further information, a per-rule statistic was required.

For the per-rule statistics, we filtered the results with those cases where the companies did fewer than 4 refactorings of a single kind of PMD rule. This ensured that only relevant data was included in the statistics, and a single-refactored PMD rule could not harm the average values.

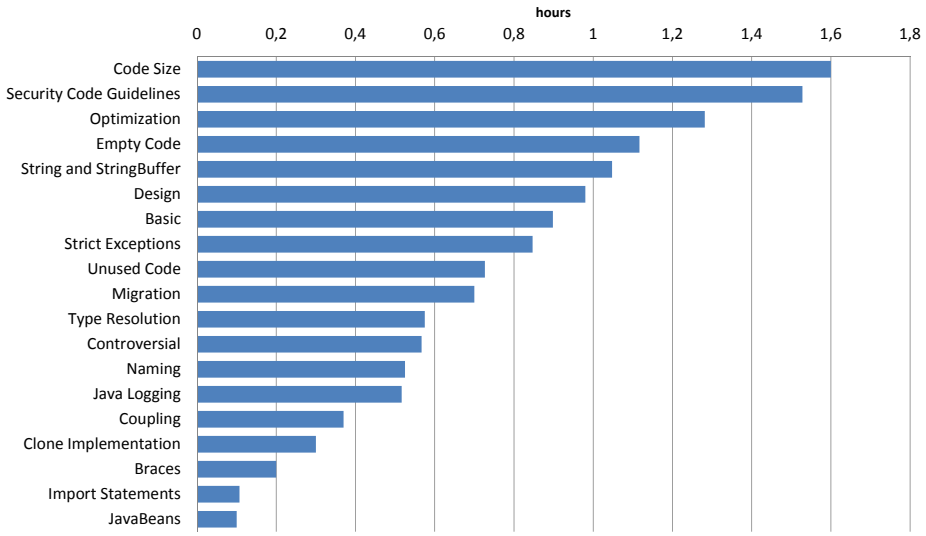

Fig. 5. Average refactoring durations by PMD Rulesets

Table 2 shows a top list of the best improving PMD rule violations. The top list was made by taking the average of the local improvements and summing the average of global improvements, in descending order.

PMD rule violation	Rank
PMD_LoC - LooseCoupling	1.
PMD_PLFIC - PositionLiteralsFirstInComparisons	2.
PMD_CCOM - ConstructorCallsOverridableMethod	3.
PMD_ALOC - AtLeastOneConstructor	4.
PMD_ATRET - AvoidThrowingRawExceptionTypes	5.
PMD_ULV - UnusedLocalVariable	6.
PMD_USBFSA - UseStringBufferForStringAppends	7.
PMD_OBEAH - OverrideBothEqualsAndHashcode	8.
PMD_AICICC - AvoidInstanceofChecksInCatchClause	9.
PMD_MRIA - MethodReturnsInternalArray	10.

Table 2. Top 10 PMD rules with the best improvements

Risk Table 3 shows the riskiest PMD rules used to refactor based on the replies by company experts. We can observe that in most cases the riskiest refactorings are for basic Java functionalities. The list includes rules concerning *java.lang.Object*'s *clone*, *hashCode* and *equals* method implementation, proper *catch* blocks and *throws* definitions, array copying and unused variables. All of the previous refactorings increased the quality of the software (by definition), but fixing these rule violations can have some unexpected consequences. These unexpected consequences are also caused by a previous improper implementation. Of course, if the software code had been written properly in the first place,

these unexpected results would have been appeared earlier, and could have been fixed during the development phase.

PMD rule violation	Rank
PMD_PCI - ProperCloneImplementation	1.
PMD_ALOC - AtLeastOneConstructor	2.
PMD_SDTE - SignatureDeclareThrowsException	3.
PMD_ACNPE - AvoidCatchingNPE	4.
PMD_LoC - LooseCoupling	5.
PMD_OBEAH - OverrideBothEqualsAndHashcode	6.
PMD_AICICC - AvoidInstanceofChecksInCatchClause	7.
PMD_ULV - UnusedLocalVariable	8.
PMD_AISD - ArrayIsStoredDirectly	9.
PMD_ATNPE - AvoidThrowingNullPointerException	10.

Table 3. Top 10 riskiest PMD rules to refactor

4.4 RQ4: Which refactoring operations give the best ROI?

In the above we saw the most beneficial and riskiest PMD rules, but which rule violations should we fix to improve the code the most with the least risk and as speedily as possible? To discover this, we defined an index value which indicates the 'return of investment' or ROI for short. To calculate this index, we ranked the averages of each attributes according to their percentage values with all averages in the same attribute using the *percentrank*[¶] function.

$$\begin{aligned} ROI_{refactoring} = &\ \text{percentrank}(\text{average}(improvement_{local})) \\ &+ \text{percentrank}(\text{average}(improvement_{global})) \\ &- \text{percentrank}(\text{average}(risk)) - \text{percentrank}(\text{average}(time)) \end{aligned}$$

After we got the index number, we ordered the rule violations and took the first 15, which are listed in Table 4. Based on our findings the best ROI is indicated by mostly small, local refactorings of those possible errors that can cause big inconsistencies in future development or other parts of the software.

ROI statistics can tell developers which rule violations need to be fixed in order to get the most out of their refactoring efforts. They can improve the effectiveness of the software maintenance process, and can fix more issues; thus they can help to make the system more robust and also reduce the overall maintenance costs.

4.5 RQ5: How can we schedule refactoring operations efficiently?

Now we will describe a way of scheduling refactoring operations. First, we will examine how the industrial partners scheduled their refactorings and then we will make recommendations based on these observations.

[¶]Here, **percentrank** returns the rank of a value in a data set as a percentage of the data set.

PMD rule violation	Rank
PMD_LoC - LooseCoupling	1.
PMD_ATRET - AvoidThrowingRawExceptionTypes	2.
PMD_USBFSA - UseStringBufferForStringAppends	3.
PMD_OBEAH - OverrideBothEqualsAndHashcode	4.
PMD_PLFIC - PositionLiteralsFirstInComparisons	5.
PMD_MRIA - MethodReturnsInternalArray	6.
PMD_LVCBF - LocalVariableCouldBeFinal	7.
PMD_ALOC - AtLeastOneConstructor	8.
PMD_SDTE - SignatureDeclareThrowsException	9.
PMD_CCOM - ConstructorCallsOverridableMethod	10.
PMD_PST - PreserveStackTrace	11.
PMD_UPF - UnusedPrivateField	12.
PMD_ULV - UnusedLocalVariable	13.
PMD_AICICC - AvoidInstanceofChecksInCatchClause	14.
PMD_CC - CyclomaticComplexity	15.

Table 4. Top 15 PMD rules to refactor with the best ROI values

How did companies schedule their refactorings? We asked the companies how they scheduled their refactoring operations when fixing rule violations. Each of the companies used the priority attribute that was given for each kind of rule violation, by using the toolchain that was used to extract the rule violations. Priorities were 1, 2, 3, indicating different levels of threat for each rule violation.

- **Priority 1** indicates dangerous programming flows.
- **Priority 2** indicates not so dangerous, but still risky or unoptimized code segments.
- **Priority 3** indicates violations to common programming and naming conventions.

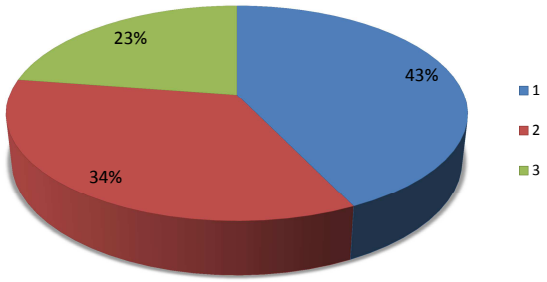

Fig. 6. Fix rate according to Priority

In Figure 6, we can see the percentage values of all the issues that were fixed for each priority level. They reveal that companies fixed Priority 1 issues the most and Priority 2 issues the second most. This means that companies here opted to fix the most threatening rule violations detected in the code.

Given these attributes, the most efficient way is to start refactoring those issues that had Priority 1 level rule violations. To find out how the companies actually scheduled their refactorings, we split the refactorings into two sets. The first set contains refactorings which were made in the first half of the project, and the other set contains refactorings made in the second half. The results of these experiments are represented in Figure 7. They tell us in percentage terms how much was fixed for each priority level in the first half and second half of the project. They indicate that the companies fixed most Priority 1 rule violations in the first half of the project and fixed most Priority 2 rules in the second half. This is consistent with what the companies told us and they provided good feedback on how they scheduled their refactoring process.

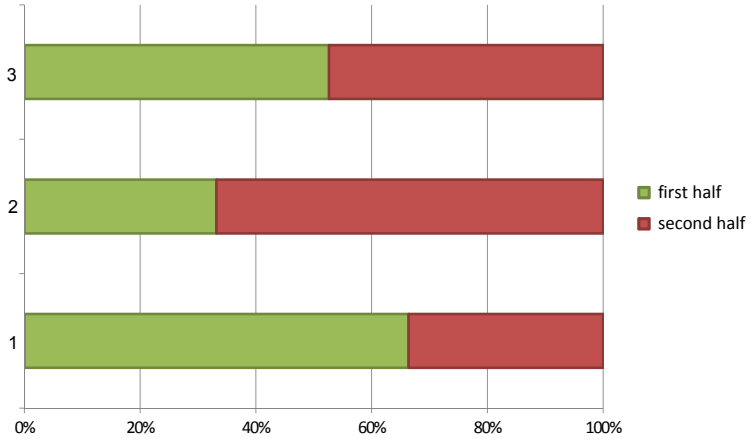

Fig. 7. Fixes in the first and second half according to Priority

How should the refactoring be scheduled? Learning from the experiences that the participating companies gained in the project, and from the results presented in Section 4.4, we suggest two kinds of scheduling. They are:

- Schedule refactorings by the priority-level of the issue, starting with the most threatening ones.
- Schedule refactorings by fixing issues with the best ROI score (see Table 4).

Choosing either of the above approaches should give an effective refactoring procedure. Scheduling by priority-level concentrates on fixing the most threatening issues, while concentrating on the ROI score should bring about the best improvement with the least effort and time. Moreover, combining the these former methods can also lead to a very efficient refactoring procedure, which offers the best of both approaches.

5 Discussion

Next, we will elaborate on potential threats to validity and some other interesting results that we obtained from our survey.

Threats to Validity

We identified some threats that can affect the construct, internal and external validity of our results.

The first one we encountered was the subjectivity of the survey. The answers to our survey questions were given by developers on a self-assessment basis. We did not measure the time needed or enhancement of refactorings with any automated solution; instead we let the developers answer the survey freely. Nevertheless, we carried out the survey with five industrial partners and therefore with many experts, which surely makes the results statistically relevant.

Another threat that we anticipated was that developers got 'unlimited' extra money and time to do the refactorings, so we could monitor how they refactored their system without any budget pressure. Although they got extra time and money in part of the project, there were still limits that might affect the results and the refactoring process.

Turning to external validity, the generalizability of our results depends on whether the selected programming language and rule violations are representative for general applications. The Java programming language was selected in the assessment together with the companies. These refactorings were made mostly on issues identified by PMD rule violations, hence they were Java specific. However, most of these rules could be generalized to abstract Object-Orientated rules, or they can be specifically defined for other programming languages.

Another threat is that whether fixing PMD rule violations can be viewed as refactoring or not. PMD refactorings are not like traditional refactoring operations that most studies examine (e.g. pull up, push down, move method, rename method, replace conditional with polymorphism). Despite this, Fowler [10] defined refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." During the project we encountered several PMD rule violations and our general experience is that the refactoring of these violations does not alter external behavior, so they can by definition be treated as refactoring.

Overall, our methods were evaluated on large-scale industrial projects, with contributions from expert developers, on a big set of data, which is a rather unique case study in the refactoring research area.

Other results

In our case study (see Section 4) we summarized our results based on research questions addressed to experts working in five ICT companies. However we ran into several interesting cases which were worth mentioning, but could not be incorporated into our research questions.

One of the interesting cases we found was when we searched for the longest-lasting refactorings. We found that *Company A* carried out a *SignatureDeclareThrowsException* refactoring, which lasted 16 hours. The issue occurred in a method of a widely implemented interface, and the problem was that the method threw a simple *java.lang.Exception* Exception-type. This is not recommended because it hides information and it is harder to handle exceptions. The developer assigned to the issue estimated that the work took 1-2 days, and said

that the risk was high because it impacted 10-25 files, but it was worth refactoring because the extra information they gained after the refactoring helped improve the maintainability of the source code.

Another intriguing example was with the same search as before. We found that *Company D* performed several *AvoidDuplicateLiterals* refactorings, which took them 7 hours on average to do; and each of the refactorings impacted on more than 100 classes. According to the comments in the survey, they used NetBeans IDE[‖] to fix these kinds of issues. NetBeans IDE has a integrated refactoring suite that helps developers to refactor their source code. Here, they used this suite to extract duplicated literals to constant variables. The survey comments revealed that the refactoring suite really helped them in this refactoring task, and it would be great help if automated solutions could be devised and implemented to tackle other of issues as well.

6 Conclusions and Future Work

In our study, we evaluated five research questions on refactoring in Java programs. The main goal of our experiments was to learn how developers refactor in an industrial context when they have the required resources (both time and money) to do so.

Our experiments were carried out on 5 large-scale industrial Java projects of different sizes and complexity. We studied refactorings on these systems, and learned which kinds of issues developers fixed the most, and which of these refactorings were best according to certain attributes defined in Section 4.2.

We also found that developers tried to optimize their refactoring process to improve the quality of these systems. We recommended two methods to schedule refactorings; one was based on priority and the other was based on a return in investments. Forming a refactoring process from either of these or simply combining them should lead to a very efficient refactoring process, making the system more robust, more maintainable, and most of all with lower costs.

In our experiments we gathered really big data on manual refactorings in an in-vivo industrial context. In this case study, we limited the context to the numerical evaluation of these results and investigated how to best select code fragments to effectively refactor our code base so as to improve software quality. In the future, with the data we obtained, we would like to investigate the effects of refactorings on source code quality and implement automatic techniques based on these results. We would also like to investigate the usage of these automatic algorithms as well.

Acknowledgements

This research was supported by the Hungarian national grant GOP-1.2.1-11-2011-0002. Here, we would like to thank all the participants of this project for their help and cooperation.

[‖] https://netbeans.org/

References

1. Alshayeb, M.: Empirical investigation of refactoring effect on software quality. Inf. Softw. Technol. 51(9), 1319–1326 (Sep 2009)
2. Arcelli Fontana, F., Braione, P., Zanoni, M.: Automatic detection of bad smells in code: An experimental assessment. Journal of Object Technology 11(2), 1 – 38 (08 2012)
3. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, first edn. (1999)
4. Choi, E., Yoshida, N., Ishio, T., Inoue, K., Sano, T.: Extracting code clones for refactoring using combinations of clone metrics. In: Proceedings of the 5th International Workshop on Software Clones. pp. 7–13. IWSC '11, ACM (2011)
5. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Morgan Kaufmann (2002)
6. Du Bois, B.: A Study of Quality Improvements by Refactoring. Ph.D. thesis (2006)
7. Du Bois, B., Gorp, P.V., Amsel, A., Eetvelde, N.V., Stenten, H., Demeyer, S.: A discussion of refactoring in research and practice. Tech. rep. (2004)
8. Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: Columbus – Reverse Engineering Tool and Schema for C++. In: Proceedings of the 18th International Conference on Software Maintenance (ICSM'02). pp. 172–181. IEEE Computer Society (Oct 2002)
9. Fontana, F.A., Spinelli, S.: Impact of refactoring on quality code evaluation. In: Proceedings of the 4th Workshop on Refactoring Tools. pp. 37–40. WRT '11, ACM (2011)
10. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc. (1999)
11. Maiga, A.: Impacts and Detection of Design Smells. Ph.D. thesis (2012)
12. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. 30(2), 126–139 (Feb 2004)
13. Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., Succi, G.: Balancing agility and formalism in software engineering. chap. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, pp. 252–266. Springer-Verlag (2008)
14. Murphy-Hill, E., Black, A.P., Dig, D., Parnin, C.: Gathering refactoring data: A comparison of four methods. In: Proceedings of the 2nd Workshop on Refactoring Tools. pp. 7:1–7:5. WRT '08, ACM (2008)
15. Opdyke, W.F.: Refactoring Object-oriented Frameworks. Ph.D. thesis (1992)
16. Pinto, G.H., Kamei, F.: What programmers say about refactoring tools?: An empirical investigation of Stack Overflow. In: Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools. pp. 33–36. WRT '13, ACM (2013)
17. Ratzinger, J., Fischer, M., Gall, H.: Improving evolvability through refactoring. SIGSOFT Softw. Eng. Notes 30(4), 1–5 (May 2005)
18. Stroggylos, K., Spinellis, D.: Refactoring–does it improve software quality? In: Proceedings of the 5th International Workshop on Software Quality. pp. 10–. WoSQ '07, IEEE Computer Society (2007)
19. Tairas, R.: Clone detection and refactoring. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. pp. 780–781. OOPSLA '06, ACM (2006)
20. Zibran, M.F., Roy, C.K.: Towards flexible code clone detection, management, and refactoring in IDE. In: Proceedings of the 5th International Workshop on Software Clones. pp. 75–76. IWSC '11, ACM (2011)