Do Automatic Refactorings Improve Maintainability? An Industrial Case Study

Gábor Szőke, Csaba Nagy, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy
Department of Software Engineering
University of Szeged, Hungary
{gabor.szoke,ncsaba,hpeter,ferenc,gyimothy}@inf.u-szeged.hu

***Abstract*—Refactoring is often treated as the main remedy against the unavoidable code erosion happening during software evolution. Studies show that refactoring is indeed an elemental part of the developers' arsenal. However, empirical studies about the impact of refactorings on software maintainability still did not reach a consensus. Moreover, most of these empirical investigations are carried out on open-source projects where distinguishing refactoring operations from other development activities is a challenge in itself.**

We had a chance to work together with several software development companies in a project where they got extra budget to improve their source code by performing refactoring operations. Taking advantage of this controlled environment, we collected a large amount of data during a refactoring phase where the developers used a (semi)automatic refactoring tool. By measuring the maintainability of the involved subject systems before and after the refactorings, we got valuable insights into the effect of these refactorings on large-scale industrial projects. All but one company, who applied a special refactoring strategy, achieved a maintainability improvement at the end of the refactoring phase, but even that one company suffered from the negative impact of only one type of refactoring.

***Index Terms*—automatic refactoring; software maintainability; coding issues; ISO/IEC 25010**

I. Introduction

Refactoring has become an integral part of the software life cycle. Developers do these restructurings as a regular task and studies show that they really perform them often: about 70–80% of all structural changes in the code are due to refactorings [1], [2]. Since this term first appeared in the literature [3], [4], it has described smaller or larger structural changes in the source code to improve its quality, but keeping its original behavior. As Fowler defines it, refactoring is "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*" [4]. He proposes this technique mainly for improving understandability and changeability. However, further research work show that the idea can also be applied to other purposes [5], such as improving performance, security, and reliability. In fact, as our previous research [6] indicates, developers often tend to do refactoring to fix coding issues that clearly affect the quality of the system, besides refactoring code smells or antipatterns.

By definition, the intention of developers with refactoring is to improve comprehensibility, maintainability, hence the quality of the source code. However, there is a disagreement in the literature whether it truly improves quality or not. For instance, an 'extract method' operation may decrease the average complexity of methods in a class, but may increase other metrics, such as the number of methods, or even coupling or cohesion metrics. That is, we win some improvements on one hand, but we loose some on the other hand. It is hard to find a good balance. Therefore, many researchers investigate the impact of refactoring on source code metrics or source code quality [7]–[13], but there is no general agreement on the results. Most of these studies were performed on several small and/or open-source projects, and experience reports on large-scale proprietary software systems are still missing to help to understand how developers use – or should use – refactorings.

We had a chance to work together with 5 software development companies from the ICT sector in a project where they got extra budget to improve their source code by refactoring. First, they did the task manually, and later, after we developed a tool together to automate refactorings, they used the tool to commit thousands of automatic refactorings. Thanks to the project requirements, all the refactorings were thoroughly documented. In our previous work [6], we investigated what kind of manual restructurings were targeted by the developers (e.g. eliminating bad smells, improving metrics, fixing coding issues) and we found that they mostly decided to fix coding issues, which can cause faults. Later, we studied the effects of *manual refactorings* on source code maintainability [14].

In the work presented in this paper, we study the effects of *automatic refactorings* on source code maintainability in a fully industrial environment. As developers decided to fix coding issues, the refactorings studied here are also about these issues. For measuring source code maintainability, we employed the *ColumbusQM* maintainability model [15] which is implemented in the *QualityGate* tool [16].

We found that the resulting source code of automatic refactorings are different compared to those of manual refactorings, because developers tend to accept an automatically generated code modification even if it is not the best solution that they would use by performing manual refactoring. To study this situation, we address the following questions:

1) Does automatic refactoring increase the overall maintainability of a software system?
2) What is the impact of different automatic refactoring types on software maintainability?
3) What is the impact of different automatic refactoring types on the code metrics used in the maintainability model?

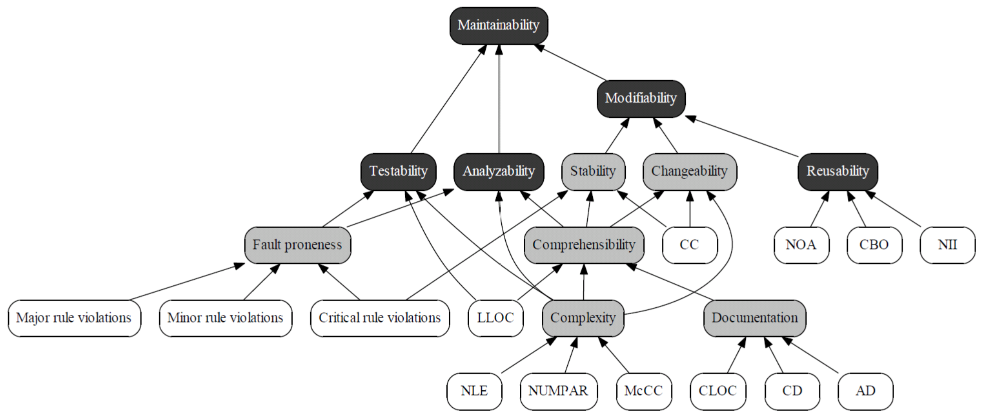

Figure 1. ColumbusQM maintainability model attribute dependency graph (ADG)

We will show in our *in vivo* case study that the answers to these questions are not as straightforward as one might think at first glance. *All the data we used is available as an online appendix to ease the reproduction of this study.*[1]

The rest of the paper is organized as follows. In Section II we give an overview of the project background and the maintainability model that we use for the analysis. In Section III, we introduce the methodology how we address our practical questions, and we present the results in Section IV. Finally, we give an overview of the related work in Section VI and draw our conclusions in Section VII.

II. Background

A. Motivating Project

This work was part of an R&D project supported by the EU and the Hungarian Government. The goal of the 2-year project was to develop a software refactoring framework, methodology and software tools to support the "continuous reengineering" methodology, hence provide support to identify problematic code parts in a system and to refactor them to enhance maintainability. During the project, we developed an automatic/semi-automatic refactoring framework and tested it on the source code of the 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 tools, but they also tested and used the toolset on the source code of their own products. This provided a good chance to them to refactor their code and improve its maintainability.

Five experienced software companies were involved in this project. These companies were founded in the last two decades and some of their projects were initiated before the millennium. Their projects consisted of about 5 million lines of code altogether, written mostly in Java, and covered different ICT areas like ERPs, ICMS and online PDF Generation. An overview of these can be seen in Table I.

B. Estimation of the Maintainability Change

Several maintainability models exist which try to numerically express the maintainability of a software system. Most of these models rely on the observation that the increase of some code metrics (e.g. length of the code, or complexity) indicate decrease in the maintainability, thus software quality. Chidamber and Kemerer [17] defined several object-oriented metrics; these definitions are *de facto* standards used in many works. Gyimóthy et al. [18] validated empirically that the increase of some of the defined metrics (e.g. CBO) indeed increase the probability of faults.

[1]http://www.inf.u-szeged.hu/~ferenc/papers/ICSME2015/

Table I
Companies involved in the project

Company	Primary domain
Company I.	Enterprise Resource Planning (ERP)
Company II.	Integrated Business Management
Company III.	Integrated Collection Management
Company IV.	Specific Business Solutions
Company V.	Web-based PDF Generation

To calculate the absolute maintainability values for the refactored and prior revisions of all the systems we used the ColumbusQM probabilistic software maintainability model [15] that is based on the quality characteristics defined by the ISO/IEC 25010 [19] standard. In the current work we treat the estimation of maintainability as a "black box", but it is important to give a high level overview of how this maintainability model works internally.

The computation of the high-level quality characteristics is based on a directed acyclic graph (see Figure 1) 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 estimated 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 1). 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), there are also new ones introduced (light gray nodes). The used version of the maintainability model considers the metrics listed in Table II.

Table II
SOURCE CODE METRICS USED BY THE MAINTAINABILITY MODEL

Metric	**Abbr.**	**Description**
Logical Lines Of Code	LLOC	Number of non-comment and non-empty lines of code.
Number Of Ancestors	NOA	Number of classes that a given class directly or indirectly inherits from.
Nesting Level Else-if	NLE	The maximum of the control structure depth in a method. 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.
Coupling Between Object classes	CBO	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.
Clone Coverage	CC	A real value between 0 and 1 that expresses what amount of the item is covered by code duplication.
NUMber of PARameters	NUMPAR	The number of parameters of the methods.
McCabe's Cyclomatic Complexity	McCC	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 (?:)*.
Number of Incoming Invocations	NII	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 only once.
API Documentation	AD	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).
Comment Lines of Code	CLOC	Number of comment and documentation code lines of a method.
Comment Density	CD	Ratio of the comment lines of a method (CLOC) to the sum of its comment (CLOC) and logical lines of code (LLOC).
Critical / Major / Minor issues	–	Number of critical/major/minor coding rule violations in the methods.

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). A so-called *goodness value* (from the [0,1] interval) is calculated for each node in the ADG that expresses how good or bad (1 is the best) is the system regarding that quality attribute. These values are transformed into the [0,10] interval by the *QualityGate*[2] *SourceAudit* tool [16] which implements the model, therefore we will use this scale throughout the paper. 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 on ColumbusQM, please refer to the work of Bakota et al. [15].

III. METHODOLOGY

Figure 2 shows a brief overview of the automatic refactoring phase of the project. In this phase, developers of participating companies were asked to use the implemented Refactoring Framework[3] to execute refactoring tasks on their systems (semi)automatically. This framework is implemented as a server-side component providing three types of services:

- A static source code analyzer tool set to derive low-level quality indicators to be used for identifying refactoring targets.
- A persistence layer above a database for storing and querying analysis data (with a complete history).
- A set of web services capable of (semi)automatically executing various refactoring operations to eliminate certain coding issues and generating a source code patch to be applied on the original code base.

As can be seen from the above list, the framework was not only providing refactoring algorithms for the developers, but it gave them support to identify possible targets for refactoring by analyzing their systems using a static source code analyzer, namely the *SourceMeter*[4] tool which is based on the *Columbus* technology [20] (*Step 1*). Although the tool is able to give a list of problematic code fragments including coding issues, antipatterns (e.g. duplicated code, long functions) and source code elements with problematic metrics at different levels (e.g. classes/methods with too high complexity and classes with bad coupling or cohesion metrics), given that the framework only supports the refactoring of 21 different coding issues, the companies were asked to fix issues from this list.

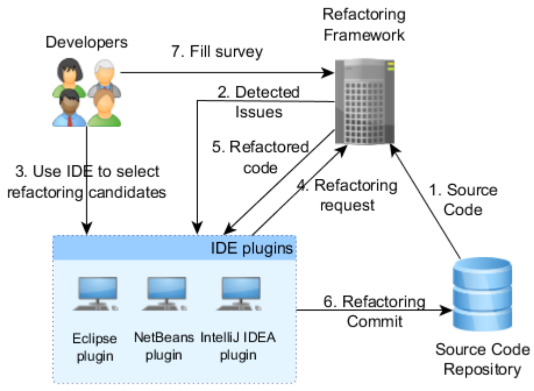

Figure 2. Overview of the refactoring process

It was a project requirement for the developers to refactor their own code, hence improve its maintainability, but they were free to select how they go through that. So it was the choice of the developers what kind of coding issues they fix with the help of the framework. The process of fixing a coding issue was to apply the appropriate refactoring operation offered by the framework through the developers' standard development environment. Plugins were created for all three IDEs the companies used (Eclipse, NetBeans, and IntelliJ IDEA) to easily utilize the functionalities provided by the Refactoring Framework. These plugins were able to load from the framework and to present in the IDEs all the detected coding issues that the developers were able to refactor (semi)automatically (*Step 2*). To apply a refactoring

[2]https://www.quality-gate.com/
[3]http://www.sed.inf.u-szeged.hu/FaultBuster
[4]https://www.sourcemeter.com/

operation, the developers selected an issue from the code and called the refactoring service through the IDE plugins (*Step 3*). Some of the refactoring algorithms could be executed fully automatically, like the refactoring of “Local Variable Could Be Final” coding issue (i.e. inserting the final keyword in front of a variable is unambiguous), while other refactoring types required user interaction, thus were executed semi-automatically (e.g. to refactor “Empty Catch Block” we asked the developers what to insert into the empty block). Regardless of the type of the refactoring, after gathering all the required information from the plugin, a request was sent to the Refactoring Framework to perform the refactoring step on the code (*Step 4*).

The description of the refactoring mechanism in detail is beyond the scope of this study, but in short, the framework performed graph transformations on the abstract semantic graph (ASG) built from the source code to produce the appropriate refactorings. After a refactoring operation was carried out on the ASG, the framework re-generated the transformed source code. To avoid the unnecessary alternations of the code layout, the framework generated only a patch for the affected code part, not the entire source code. This patch was sent back to the IDE plugins in which the developers were able to preview the modifications (with the help of the built-in diff viewers of the IDEs) before they applied it (*Step 5*). Of course, the developers had the possibility to discard the changes if they were not satisfied with the resulting refactored code. In that case, no changes were made in the code base. Note, that the framework allowed fixing multiple issues at once, but this type of batch refactorings had to be of the same type (for example, the framework was able to fix hundreds of *PositionLiteralsFirstInComparisons* issues in one patch, but mixing issues was not supported). If the provided patch got accepted, the developers applied them on the current code base and performed a commit to upload the refactored code into the source code repository (*Step 6*).

Besides applying concrete refactorings, the project required that the companies fill out a survey (that we collected in the same framework) after each refactoring and give an explanation on what and why they refactored during their work (*Step 7*). The survey contained revision related information as well, so we could relate one refactoring to a revision in the version control systems.

After this refactoring phase, we analyzed the marked revisions and investigated the change in the maintainability of the systems caused by refactoring commits. Figure 3 gives an overview of this analysis. It was not a requirement from the developers that they commit only refactorings to the version control system, or that they create a separate branch for this purpose. It was more realistic, and some developers particularly asked us to commit these changes to the mainline or development branches, so they could develop their system in parallel with the refactoring process. What was a requirement though is that a commit containing refactoring operations could not contain other code modifications. Hence, for each system we could identify the revisions (r_{t_1}, ..., r_{t_i}, ..., r_{t_n}) that were reported in the surveys collected by the Refactoring Framework after refactoring commits, and we analyzed all these revisions with the revisions prior to them. As a result, we considered for a system the set of revisions $r_{t_1-1}, r_{t_1}, ..., r_{t_i-1}, r_{t_i}, ..., r_{t_n-1}, r_{t_n}$ where r_{t_i} is a refactoring commit and r_{t_i-1} is the revision prior to this commit.

We performed the analysis of these revisions of the source code with the *QualityGate SourceAudit* tool [16], which uses the maintainability model explained in Section II-B. If a commit contained more than one refactoring of the same type – because the framework supported such way of bulk fixing the issues – we calculated the average amount of maintainability changes of a refactoring type by dividing the maintainability change caused by the whole commit with the number of actual refactorings contained in it. Everywhere in the paper, if we deal with maintainability change caused by a refactoring type, we use the average values of these changes. This is of course a small threat to validity, as it is not guaranteed that all the fixed issues in various places in the code affect the maintainability the same way. However, all the refactorings were performed by an automatic framework which resulted in very similar (though due to the possible manual steps not necessarily the same) fixes for the issues, therefore the chances that these refactorings had different impacts on maintainability is minimal. Besides analyzing the maintainability of the above revisions, we gathered data from the version control system as well, such as diffs and log messages.

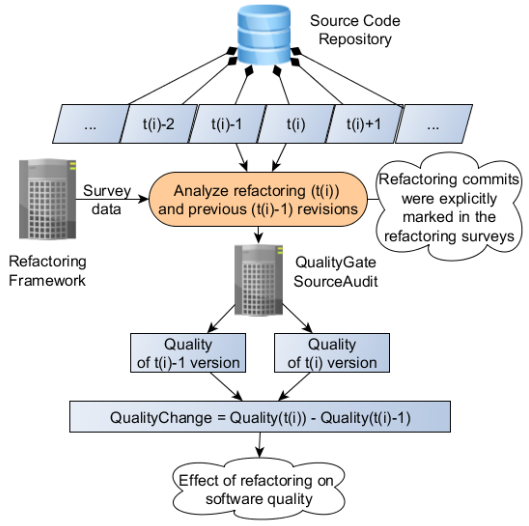
Source Code Repository
...
t(i)-2
t(i)-1
t(i)
t(i)+1
...
Survey data
Analyze refactoring (t(i)) and previous (t(i)-1) revisions
Refactoring commits were explicitly marked in the refactoring surveys
Refactoring Framework
QualityGate SourceAudit
Quality of t(i)-1 version
Quality of t(i) version
QualityChange = Quality(t(i)) - Quality(t(i)-1)
Effect of refactoring on software quality

Figure 3. Overview of the analysis process

IV. Results

Following the process described in Section III, the companies performed a large number of automatic refactorings on their own code base using the Refactoring Framework developed within the project. They uploaded almost 4,000 refactorings to the source code repositories in more than 1,000 commits altogether (see Table III). We analyzed 4 projects of 4 different companies and collected data according to the method depicted in Figure 3. That is, we calculated all the maintainability changes caused by refactoring commits and aggregated the data at various levels. As the utilized maintainability model takes the number of coding issues into consideration (see Figure 1) and the Refactoring Framework supports the refactoring of such coding issues, one might think that it is trivial that upon refactoring the maintainability of code will increase. Nonetheless, fixing an issue might cause code changes that lead to e.g. changes in code clones, new coding issues, or changes in metrics. So it is far from trivial to predict the complex effect of refactorings on code maintainability. It further complicates the task that the Refactoring Framework includes some semi-automatic steps, thus developers are able to configure the same refactoring operations somewhat differently. For example, fixing an *EmptyCatchBlock* issue begins with three options, namely *a*) add logger; *b*) use *printStackTrace()*; and *c*) leave a comment, where selecting one option may introduce new options (e.g.: comment text and logger kind).

Table III
SELECTED PROJECTS

Company	**Project**	**kLOC**	**Analyzed revisions**	**Refactoring commits**	**Refactorings**
Company I	Project A	1,119	299	217	1,444
Company II	Project B	962	868	449	1,306
Company III	Project C	206	1,313	316	404
Company IV	Project D	780	200	66	682
	Total	**3,067**	**2,680**	**1,048**	**3,836**

First, we show how the sum of all refactoring related maintainability changes turned out for the various projects. Next, we dig a bit deeper into the data to find out what is the average impact of the individual refactoring types on software maintainability. Finally, we go even one step further to explore the effect of refactoring types on software maintainability at the level of code metrics.

A. Effect of Automatic Code Refactoring on Software Maintainability

The data presented in Table IV can bring us closer to find the answer to our first question. The rows of the table contain an overview of the quality properties of 4 systems of 4 companies participating in the automatic refactoring phase of the project. The Coding Issues column shows the overall number of issues that were fixed by (semi)automatic refactoring in a particular system. The Maintainability Before and Maintainability After columns hold the maintainability values of the systems before and after the automatic refactoring phase calculated as described in Section II-B (0 is the worst value, 10 is the best). The total improvement (*Total Impr.*) column reflects the difference between the maintainability values before and after the automatic refactoring phase, thus if this value is negative, the overall maintainability of the system has decreased during the refactoring phase, while positive difference means a maintainability improvement. Note, that the companies were allowed to perform any kind of code modifications during this phase, not just refactorings, so this number shows the combined effect of all the code changes on the system maintainability. The next column, refactoring improvement (*Ref. Impr.*) is the code improvement achieved solely by refactorings. This is calculated as the sum of the maintainability changes caused by commits containing refactoring operations only (i.e. sum of the maintainability differences between refactoring and prior commits). The last column (*Ref. Impr. %*) is simply the ratio of the refactoring and total improvement values. Its intuitive meaning would be the amount of code improvement caused by refactoring commits. Greater values than 100% can occur, which mean that the effect of refactorings is higher than the overall effect of all the code changes; however, this effect might be positive and negative as well.

In total, 3 out of 4 cases the overall system maintainability values increased during the refactoring phase. For all these 3 projects, the net effect of refactoring commits was also positive, meaning that the automatic refactoring phase increased the maintainability of the code. The only exception is *Project A*, where both the overall system maintainability and the net effect of refactoring commits were negative. But even in this case only a fraction (i.e. 80%) of the maintainability decrease was caused by the refactoring commits. This finding is more or less in line with the results of the manual refactoring phase of the project. As we described in Section I, the project had two refactoring phases; first, companies performed manual refactorings that was followed by a second phase where they performed (semi)automatic refactorings. The automated refactoring framework used to perform (semi)automatic refactorings was developed also within the same project. In the current paper this second, (semi)automatic refactoring phase is under investigation. We analyzed the results of the manual refactoring [14] and found that in most of the cases refactoring improved the overall maintainability of the systems with only a few exceptions. In the case of (semi)automatic refactoring, this finding also holds for 3 projects out of 4.

Table IV
QUALITY CHANGES OF THE SELECTED PROJECTS

Company – Project	**Coding Issues**	**Maint. Before**	**Maint. After**	**Total Impr.**	**Ref. Impr.**	**Ref. Impr. %**
Comp I – Proj A	1,444	4.449238	4.411970	-0.037268	-0.029822	80
Comp II – Proj B	1,306	6.039320	6.072320	0.032999	0.032999	100
Comp III – Proj C	404	4.132307	4.258933	0.126627	0.144507	114
Comp IV – Proj D	682	6.158691	6.161626	0.002935	0.003142	107

The seemingly negative results of *Project A* could be explained by a very project specific factor. The system which suffers from maintainability decrease belongs to a company where developers performed only two different types of refactorings, namely *PositionLiteralsFirstInComparisons* and *UnnecessaryConstructor*. Their motivation might have been that these refactorings required only local changes (i.e. they were low hanging fruits), therefore they were easier to manage and test the code after the modification. However, the effect of this limited set of refactoring types is completely different than a more balanced set of refactorings (see Table V).

The results of the other three companies support this hypothesis, as they performed a much wider range of refactoring

Table V
QUALITY CHANGES CAUSED BY REFACTORING CODING ISSUES

Coding Issue Type	**Project A**			**Project B**			**Project C**			**Project D**		
	#	**avg**	**ratio**	**#**	**avg**	**ratio**	**#**	**avg**	**ratio**	**#**	**avg**	**ratio**
AddEmptyString	0	0	0	150	0	1 ↑	1	0	1 ↑	270	0	1 ↑
ArrayIsStoredDirectly	0	0	0	33	0.000027	1.31 ↑	2	0.000184	3.15 ↑	0	0	0
AvoidPrintStackTrace	0	0	0	17	0	1 ↑	3	0.000061	1.71 ↑	0	0	0
AvoidReassigningParameters	0	0	0	30	0.000010	1.12 ↑	174	0.000007	1.08 ↑	10	0	1 ↑
AvoidThrowingNullPointerException	0	0	0	10	0	1 ↑	3	0.000540	7.32 ↑	0	0	0
AvoidThrowingRawExceptionTypes	0	0	0	3	0.000062	1.63 ↑	0	0	0	6	0	1 ↑
BooleanGetMethodName	0	0	0	125	0	1 ↑	1	0	1 ↑	9	0	1 ↑
EmptyCatchBlock	0	0	0	20	0.000058	1.68 ↑	14	0.000550	7.43 ↑	30	0.000044	1.51 ↑
EmptyIfStmt	0	0	0	32	0.000012	1.14 ↑	5	0.000074	1.87 ↑	5	0	1 ↑
MethodNamingConventions	0	0	0	2	0	1 ↑	21	0.000004	1.05 ↑	9	0	1 ↑
PositionLiteralsFirstInComparisons	409	0.000114	2.33 ↑	26	0.000054	1.42 ↑	5	0.000432	6.05 ↑	9	0.000060	1.70 ↑
PreserveStackTrace	0	0	0	90	0.000003	1.04 ↑	24	**-0.000001**	0.99 ↑	8	0	1 ↑
SimpleDateFormatNeedsLocale	0	0	0	141	0.000001	1.02 ↑	17	0	1 ↑	58	0	1 ↑
SwitchStmtsShouldHaveDefault	0	0	0	170	0.000016	1.19 ↑	47	0.000020	1.23 ↑	23	0.000010	1.12 ↑
UnnecessaryConstructor	1,035	**-0.000077**	0.10 ↑	41	**-0.000023**	0.73 ↑	7	**-0.000170**	**0.99 ↓**	40	**-0.000042**	0.51 ↑
UnnecessaryLocalBeforeReturn	0	0	0	149	0	1 ↑	13	0	1 ↑	135	**-0.000002**	0.98 ↑
UnusedLocalVariable	0	0	0	32	0.000012	1.14 ↑	30	0.000050	1.58 ↑	0	0	0
UnusedPrivateField	0	0	0	36	0.000005	1.06 ↑	0	0	0	4	0	1 ↑
UnusedPrivateMethod	0	0	0	29	0.000007	1.08 ↑	6	0.000285	4.33 ↑	5	0	1 ↑
UseLocaleWithCaseConversions	0	0	0	57	0.000108	2.26 ↑	0	0	0	44	0.000034	1.40 ↑
UseStringBufferForStringAppends	0	0	0	113	0.000008	1.09 ↑	30	0.000099	2.16 ↑	17	0.000008	1.09 ↑

tasks, and the maintainability of their systems increased in all cases. In the cases of *Project C* and *D*, it is even true that the refactoring commits caused a larger increase in the maintainability than the overall increase at the end of the phase, which means that code modifications other than refactorings decreased the maintainability. In the *UnnecessaryConstructor* line of Table V, we can see that all the values are negative, meaning that this type of refactoring caused a maintainability decrease in each and every system. Taking into consideration that out of the two types of refactoring performed by Company I, *UnnecessaryConstructor* was the absolute dominant by its number, it is now clear that the overall decrease in the maintainability of their system can be credited to this single type of refactoring. It is an interesting question in itself why removing an *UnnecessaryConstructor* decreases the maintainability, which we elaborate in Section IV-B.

To summarize, we observe that the overall effect of the automatic refactoring phase tends to be positive, the only small bias is caused by the dominant number of a single type of refactoring (i.e. *UnnecessaryConstructor*) in *Project A*.

B. Impact of Automatic Refactoring Types on Software Maintainability

During the automatic refactoring period, developers fixed different kinds of coding issues, which had different impacts on software maintainability. In Table V we show for each system the number of fixed coding issues (column '#') and its average maintainability change (column 'avg') credited to the various kinds of coding issue types the developers fixed (semi)automatically. As the maintainability change of a single commit measured on the scale of 0 to 10 is extremely small, we also added a column to the table (column 'ratio') that reflects the number of times this change is bigger or less compared to an average maintainability change caused by a non-refactoring commit. We refer to this number as *nonRefactAvg* in the followings, and its value is 0.00005. The ↑ means that the actual change is bigger than the average maintainability change of the non-refactoring commits, while ↓ marks a worse effect than the average. Please note that the average maintainability change of the non-refactoring commits is negative, thus a maintainability decrease may still be marked with ↑ (meaning that the actual maintainability degradation is smaller than that of an average commit). For example, a ratio of 1.68 ↑ means that the actual maintainability improvement is greater than the average non-refactoring commit with 1.68 times of the absolute value of the average change:

$$avg = nonRefactAvg + ratio * |nonRefactAvg|$$

This is why a neutral change value (i.e. 0) is marked with 1 ↑, as 0 is better than the average maintainability change of non-refactoring commits, which is negative.

We can easily observe that Company I fixed only 2 types of coding issues in *Project A* as we already pointed it out in the previous section. The other companies fixed coding issues more diversely, 21 types altogether. Results indicate that in 55% of the cases refactoring increased the overall maintainability of the system, while it decreased the maintainability in only 10% of the cases (indicated with bold letters). In 35% of the cases it did not cause any traceable difference in maintainability measured by the model (i.e. the model was insensitive to the change). If we compare the results with the average maintainability changes of non-refactoring commits, we can see that only one value caused a larger maintainability decrease than an average non-refactoring commit. So even in those few cases where a refactoring type caused a maintainability decrease, it was much smaller than an average maintainability degradation introduced by a commit containing no refactorings. While the largest maintainability increases caused by some refactoring types are more than 7 times bigger than the average decrease caused by non-refactoring commits.

Looking closer into the results, we can see that fixing the *UnnecessaryConstructor* coding issue decreased the maintainability in each case. This issue occurs when a constructor is not necessary; i.e., when there is only one constructor, it is public, has an empty body, and takes no arguments. The automatic refactoring algorithm simply deleted these constructors. Intuitively, the maintainability of the source code should have been increased because we deleted unnecessary code and decreased the lines of code metric as well. However, ColumbusQM is not directly affected by the system size as it could lead to false conclusions like larger systems are necessarily harder to maintain, so the code reduction itself would not justify a maintainability increase anyway. Instead of the mere sizes of the systems, the maintainability model relies on the distribution of the method lengths. In this particular case the method length distribution is shifted towards the direction of longer methods as a lot of "good quality" code/methods got deleted. The removed constructors consisted of just a few lines, had no coding issues, had small complexity and they did not refer to other classes. Therefore, a maintainability decrease occur upon deleting such good quality methods due to the shift in the distribution of metric values like length, complexity or number of parameters of the remaining methods.

There are two other issues where the maintainability of a system decreased for some of the projects. One issue is the *UnnecessaryLocalBeforeReturn* that caused a decrease in maintainability for *Project D*. In this case the automatic refactoring algorithm simply inlined the value of the local variable into the return statement (which resulted in a line deletion as well). This should have increased the maintainability because it reduces the method length and removes a coding issue from the source code. However, it did not change the maintainability or it even decreased it (albeit the decrease was very small compared to other changes). Investigation of this phenomena revealed that a single change in lines of code or in the number of minor (low-priority) rule violations is so small that it has no noticeable effect. Additionally, in some cases fixing these issues introduced code clones as well (the only difference between two methods was the unnecessary local variable) which immediately decreased the measured maintainability.

The other issue causing a maintainability decrease is *PreserveStackTrace* in *Project C*. The typical fix of this issue is to add the root exception as a second parameter to the constructor of the newly thrown exception. However, Company III could not apply this strategy as their own exception classes did not override this two parameter constructor. So instead of the usual fix, they instantiated a new exception in a local variable, called its *initCause()* method with the root exception and threw the new exception. Besides the additional lines, the fix also introduced a new incoming call to the *initCause()* method of the exception objects. All these decrease the maintainability, which slightly outweigh the positive effect of removing a coding issue.

All in all, the results indicate that despite the seemingly counter-intuitive effects of fixing some issues, refactoring different types of coding issues usually increase code maintainability.

C. Impact of Automatic Code Refactoring on Code Metrics

Table VI shows all sensor nodes (internal quality properties) of the ColumbusQM ADG (see Figure 1), and the overall maintainability of a system as well. Sensor nodes represent *goodness* values of source code metrics. In the table we list two ratios for each sensor node. A ratio is the number of coding issue fixes when the refactoring caused a positive (column '+') or negative (column '–') change to the goodness value of the current sensor, divided by the number of all refactorings (positive, negative, and zero change). The values larger than 0.5 are highlighted with bold letters. The table also shows the priority (column *Pri.*) for each coding issue according to a scale between 1-3, and describes how dangerous an issue is (P1 – critical, P2 – major, P3 – minor).

The goal of the project was to increase the maintainability of the software systems. The column '*Maint.* +' shows the ratio of how many times a refactoring increased the overall maintainability of a system. For example, 0.86 means that *UseLocaleWithCaseConversions* fixes had a positive impact on maintainability in 86% of the cases. The column '*Maint.* –' shows the ratio of how many times a refactoring decreased the overall maintainability of a system. Looking at the same line as before, we see that the value is 0, which means that fixing this type of issues never decreased the maintainability. The remaining 14% did not affect the maintainability in neither positive nor negative way.

Looking at these values, we can see that fixing coding issues mainly increases the overall maintainability. However, there are a few issue types which did not change the maintainability at all, or they even decreased it. Increases happened mostly because of the expected behavior of the maintainability model, that is, decreasing the number of coding issues in the source code improves maintainability and stability, thus quality. This behavior can be observed mainly in the *P1*, *P2*, *P3* columns (the numbers of coding issues with different priorities, respectively). For example, *ArrayIsStoredDirectly* did not change any other sensors, just the number of *P2* coding issues and this increased the maintainability in all cases. Still, this pattern cannot be applied to every row in the table. For example *AddEmptyString*, *BooleanGetMethodName* coding issues increased the goodness of *P3* sensor in 6-7% of the cases but we cannot see any increase in maintainability. This is because the positive effect of *P3* sensor was so small that it increased the overall maintainability only in such a small amount that it is invisible due to rounding errors.

An interesting observation can be made on the *EmptyCatchBlock* where besides the 91% improvement of *P1*, one can see a 13% decrease in the *P2* sensor. A closer look into this case showed us that in some automatic *EmptyCatchBlock* refactorings developers choose to solve the issue with "put an e.printStackTrace() call into the catch block" option for the refactoring algorithm which resolved the *EmptyCatchBlock* but introduced a new *AvoidPrintStackTrace* issue at the same time.

Another compelling case is the *AvoidReassigningParameters* issue which shows a definitive improvement in the logical

Table VI
RATIOS OF QUALITY CHANGES ON INDIVIDUAL METRICS LEVEL

Coding Issue	**Pri.**	**#**	**AD**		**CBO**		**CC**		**CD**		**CLOC**		**LLOC**		**McCC**		**NII**		**NLE**		**NOA**		**PAR**		**P1**		**P2**		**P3**		**Maint.**	
			+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−	+	−
AddEmptyString	P3	421	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.06	0	0	0
ArrayIsStoredDirectly	P2	35	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0.97**	0	0	0	**0.97**	0
AvoidPrintStackTrace	P2	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.05	0	0	0	0.05	0
AvoidReassigningParameters	P3	214	0	0	0	0	0	0.11	0	0	0	0	0.02	0	0	0	0	0	0.12	0	0	0	0	0	0	0	0	0	0.16	0	0.29	0.02
AvoidThrowingNullPointerExcept.	P1	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.23	0	0	0	0	0	0.23	0
AvoidThrowingRawExceptionTypes	P2	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.11	0	0	0	0.11	0
BooleanGetMethodName	P3	135	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.07	0	0	0
EmptyCatchBlock	P1	64	0	0	0	0	0.05	0.06	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0.91**	0	0	0.13	0	0	**0.91**	0.02
EmptyIfStmt	P2	42	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.07	0	0	0	0.1	0.02
MethodNamingConventions	P3	32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.09	0	0.03	0
PositionLiteralsFirstInComparisons	P1	449	0	0	0	0	0.02	0.02	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0.88**	0	0	0	0	0	**0.88**	0
PreserveStackTrace	P2	122	0	0	0	0	0.01	0.03	0	0	0	0	0	0.01	0	0	0	0.02	0	0	0	0	0	0	0	0	0.25	0	0	0	0.25	0.06
SimpleDateFormatNeedsLocale	P3	216	0	0	0	0	0.04	0.01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.15	0	0.1	0.01
SwitchStmtsShouldHaveDefault	P2	240	0	0	0	0	0.01	0.01	0	0	0	0	0	0.01	0	0	0	0	0	0	0	0	0	0	0.04	0	0.35	0	0	0	0.4	0.01
UnnecessaryConstructor	P3	1,123	0.13	0.02	0	0	0	0.08	0	0	0.04	0.01	0	0.14	0	0.16	0.01	0.21	0	0.14	0	0	0	0.22	0	0.08	0	0.13	0.19	0	0.04	**0.73**
UnnecessaryLocalBeforeReturn	P3	297	0	0	0	0	0	0.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.12	0	0	0.1
UnusedLocalVariable	P2	62	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.15	0	0	0	0.15	0
UnusedPrivateField	P2	40	0	0	0.03	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.03	0
UnusedPrivateMethod	P2	40	0	0	0.03	0	0	0.05	0	0	0	0	0	0	0	0	0	0	0.05	0	0	0	0.08	0	0	0	0.05	0	0	0	0.1	0.05
UseLocaleWithCaseConversions	P1	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0.86**	0	0	0	0	0	**0.86**	0
UseStringBufferForStringAppends	P2	160	0	0	0	0	0	0	0	0	0	0	0.01	0	0	0	0	0	0	0	0	0	0	0	0	0	0.25	0	0	0.01	0.26	0

lines of code (LLOC) and nesting level (NLE) sensors. Fixing this reassignment involved removing some code parts that lowered the code lines and sometimes the complexity (i.e. the maximal nesting level) of the projects. Besides reducing the number of coding issues, these improvements caused a maintainability increase in 29% of the cases. However, in 2% of the cases, we observed a maintainability decrease though. That is because in 11% of the cases the removal of some code parts resulted in new code clones (CC), hence, two or more code parts differed only in the removed statements. So the effect of this refactoring is not trivial to predict, but in the majority of the cases we observed a maintainability increase.

UnusedPrivateField increased the goodness of the CBO sensor in 3% of the cases but it did not affect any of the other sensors. This happened mostly because the small number of fixes and also because it sometimes introduced *UnusedImports* coding issues as well.

Section IV-B showed why *UnnecessaryLocalBeforeReturn* coding issue decreased maintainability. Table VI shows that the introduction of code clones (CC) had bigger effect on maintainability than the fixes of the *P3* issue. Similarly, *UnnecessaryConstructor* is also referred in Section IV-B and its effects can be seen in Table VI in detail. Almost every sensor is affected by this coding issue fix, but this is mainly because the large number of refactorings.

As a summary, we observe that fixing coding issues via automatic refactorings does not have a significant impact on metrics in most of the cases, mainly because the changes are local. However, some fixes have an effect on metrics one would not think of at first glance.

V. THREATS TO VALIDITY

Even in a case study which was carried out in a controlled environment, there are many different threats which should be considered when we discuss the validity of our observations. Here, we give a brief overview of the most important ones.

Heterogeneity of the commits: As we were interested in the effect of particular refactoring types on software maintainability, we filtered out those commits that contained different types of refactorings. Although the number of such commits was relatively low, it is obviously a loss of information. Additionally, when a commit contained multiple refactoring operations of the same type, we had to use the average of the maintainability changes to estimate the effect of an individual refactoring operation. This is also a threat to validity, as same refactorings may have different impact on the same system. However, its likelihood is minimal, as all the refactorings have been carried out (semi)automatically, thus resulting a very similar type of modifications in the code.

Maintainability analysis relies only on the ColumbusQM maintainability model: The maintainability model is an important part of the analysis as it determines also what we consider as an "effect on maintainability" of refactorings. Currently we rely on ColumbusQM with all of its advantages and disadvantages. On the positive side this model is published, validated and reflects the opinion of developers [15]; however, we saw that the model might miss some aspects which would reflect some changes caused by refactorings.

Limitations of the project: We claim that our experiment was carried out in an *in vivo* industrial context. However, this project might had unintentional effects on the study. For example, the budget for refactoring was not "unlimited" and some companies were seeking for fixes requiring the lowest amount of extra effort. A good example for this is Company I, who performed basically only two such types of refactorings.

Limitations of the supported refactoring types: The supported automatic refactorings focus on fixing 21 different coding issues. It is only a fraction of the possible and widely used set of refactoring operations, therefore the final conclusions are limited to these type of refactorings. However, most of these refactorings are simple yet powerful tools to improve the code structure agreed by all the companies involved in the project.

VI. Related Work

Since the term ‘refactoring’ was introduced [3], [4], many researchers studied the role of it in software development. Some studies estimate that about 70–80% of all structural changes in the code are due to refactorings [1], [2], which clearly indicates its importance in software evolution. Mens et al. published a survey to provide an extensive overview of research work in the realm of software refactoring [21] and cited over 100 studies. But the popularity of the topic has been further increasing.

Automation techniques can support the regular task of refactoring and are intensively studied by researchers. Ge et al. implemented the BeneFactor tool which detects developers’ manual refactoring and reminds them that automation is available, then it completes the refactoring automatically [22], [23]. Vakilian et al. proposed a compositional paradigm for refactoring (automate individual steps and let programmers manually compose the steps into a complex change) and implemented a tool to support it. Henkel et al. implemented a framework which captures and replays refactoring actions [24]. Jensen et al. used genetic programming for automated refactoring and the introduction of design patterns [25]. Also, there are many approaches to support specific refactoring techniques, e.g. extract method [26], [27], refactoring to design patterns [28] or clone refactoring [29].

There seems to be, however, a disagreement among researchers whether refactoring truly improves software maintainability or not. Stroulia and Kapoor [7] investigated how metrics were affected and found that size and coupling metrics of their system decreased after the refactoring process. Du Bois and Mens [9] studied the effects of refactoring on internal quality metrics based on a formalism to describe the impact of a representative number of refactorings on an AST representation of the source code. Du Bois wrote his dissertation on studying the effects of refactoring on internal and external program quality attributes [30] and previously Du Bois et al. [31] proposed refactoring guidelines for enhancing cohesion and coupling metrics; they obtained promising results by applying these transformations to an open-source project. Kataoka et al. [8] provided a quantitative evaluation method to measure the maintainability enhancement effect of refactorings. Yu et al. [32] adapted a modeling framework in order to analyze software qualities to determine which software refactoring transformations are the most appropriate. Moser et al. [11] studied the impact on quality and productivity as observed small teams working in similar, highly volatile domains and assessed the impact of refactoring in a close to industrial environment. Their results indicate that refactoring not only increases software quality, but also improves productivity. One of the few industrial case studies investigating the typical use and benefits of refactorings is carried out by Kim et al. [33] at Microsoft. Their survey revealed that the refactoring definition in practice is not confined to a rigorous definition of semantics-preserving code transformations and that developers perceive that refactoring involves substantial cost and risks. They found that the top 5 percent of preferentially refactored modules in Windows 7 experience higher reduction in the number of inter-module dependencies and several complexity measures but increase size more than the bottom 95 percent. This indicates that measuring the impact of refactoring requires multi-dimensional assessment.

A large-scale study was carried out by Murphy-Hill et al. [12] where they studied manual refactorings from Fowler’s catalog, and their data set spans over 13,000 developers with 240,000 tool-assisted refactorings of open-source applications. Similarly, Negara et al. [13] presented an empirical study that considered both manual and automated refactorings. They reported that they analyzed 5,371 refactorings applied by students and professional programmers, but they did not provide further information on the systems under question.

Most of the above studies were performed on either several small projects and/or open-source systems, which is one important difference compared to our work, as we observe a *large amount of automatic refactorings on proprietary software*. Another difference is that we use the ColumbusQM to objectively measure changes in the maintainability, while earlier studies rely only on internal code metrics. It makes us able to compare different refactorings and draw conclusions which might help developers in planning refactoring tasks or inspire research projects.

VII. Conclusions and Future Work

In this paper we described the outcome of a large-scale *in vivo* investigation of the impact of automatic refactorings on software maintainability. Taking advantage of an R&D project ensuring an extra budget for several industrial companies to carry out refactoring tasks on their software systems, we were able to collect and analyze real data regarding the maintainability of the involved systems before and after the refactorings. Given that developers are rarely allowed to spend enough time on refactoring, most of such activities are ad-hoc and undocumented, so the collected data is an asset in itself.

Employing the QualityGate SourceAudit tool [16] (which implements the ColumbusQM quality model [15]), we analyzed the maintainability changes caused by the different refactoring tasks. The analysis revealed that from the supported coding issue fixes all but one type of refactoring operation had a consistent and traceable positive impact on the software systems in the majority of the case. 3 out of the 4 involved companies reached a more maintainable system at the end of the refactoring phase. We observed however, that the first company preferred low-cost modifications, therefore they performed only two types of refactorings from which removing unnecessary constructors had a controversial effect on maintainability. Another observation was that it was sometimes counter productive to just blindly apply the automatic refactorings without taking a closer look at the proposed code modification. It happened several times that the automatic refactoring tool asked for user input to be able to select the best refactoring option, but developers used the default settings because this was easier. Some of these refactorings then introduced new coding issues, or failed to effectively remove the original issue. So human factor is still important, but the companies could achieve a measurable increase of maintainability by applying only automatic refactorings.

Last but not least, this study shed light on some important aspects of measuring software maintainability. Some of the unexpected effects of refactorings (like the negative impact of removing unnecessary constructors on maintainability) are caused by the specialties of the applied maintainability model. Repeating the study with other maintainability models could open interesting research challenges and opportunities, which lied outside of the scope of this paper. Nonetheless, most of the existing approaches use the same source of information (i.e. source code metrics) to assess maintainability, which mostly differ in the way of combining metrics, thus we would expect similar results in terms of maintainability.

In the future, we plan to compare the effects of manual and automatic refactoring phases of the project. We would like to investigate differences between the spent time and maintainability gain on a manual refactoring against an automatic refactoring. Is it worth to give up manual refactoring for the sake of the performance gain we can achieve with automatic refactoring?

Acknowledgment

This research work was supported by the EU supported Hungarian national grant GOP-1.2.1-11-2011-0002. Here, we would also like to thank to all the participants of this project for their help and cooperation, and especially to Gábor Antal for his valuable work in the maintainability measurements.

References

[1] Z. Xing and E. Stroulia, "Refactoring practice: How it is and how it should be supported - an eclipse case study," in *Proc. of the 22nd IEEE Int. Conference on Software Maintenance (ICSM2006)*. IEEE Comp. Soc., 2006, pp. 458–468.

[2] D. Dig and R. Johnson, "The role of refactorings in api evolution," in *Proc. of the 21st IEEE Int. Conference on Software Maintenance (ICSM2005)*. IEEE Comp. Soc., 2005, pp. 389–398.

[3] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois, 1992.

[4] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and using nonfunctional requirements: A process-oriented approach," *IEEE Transactions on Software Engineering*, vol. 18, no. 6, pp. 483–497, 1992.

[6] G. Szőke, C. Nagy, R. Ferenc, and T. Gyimóthy, "A case study of refactoring large-scale industrial systems to efficiently improve source code quality," in *Proc. of Computational Science and Its Applications–ICCSA 2014*. Springer, 2014.

[7] E. Stroulia and R. Kapoor, "Metrics of refactoring-based development: An experience report," in *Proc. of the 7th Int. Conf. on Object-Oriented Information Systems (OOIS2001)*. Springer, 2001, pp. 113–122.

[8] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *Proc. of the Int. Conference on Software Maintenance*. IEEE, 2002, pp. 576–585.

[9] B. Du Bois and T. Mens, "Describing the impact of refactoring on internal program quality," in *Proc. of the Int. Workshop on Evolution of Large-scale Industrial Software Applications*, 2003, pp. 37–48.

[10] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Proc. of the 5th Int. Workshop on Software Quality*. IEEE Comp. Soc., 2007, p. 10.

[11] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *Balancing Agility and Formalism in Software Engineering*. Springer, 2008, pp. 252–266.

[12] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *Proc. of the 31st Int. Conference on Software Engineering (ICSE2009)*. IEEE Comp. Soc., 2009, pp. 287–297.

[13] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *Proc. of the 27th European Conference on Object-Oriented Programming (ECOOP2013)*. Springer-Verlag, 2013, pp. 552–576.

[14] G. Szőke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?" in *Proc. of the 14th Int. Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Comp. Soc., 2014, pp. 95–104.

[15] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy, "A probabilistic software quality model," in *Proc. of the 2011 27th IEEE Int. Conference on Software Maintenance (ICSM2011)*. IEEE Comp. Soc., 2011, pp. 243–252.

[16] T. Bakota, P. Hegedűs, I. Siket, G. Ladányi, and R. Ferenc, "Qualitygate sourceaudit: a tool for assessing the technical quality of software," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 440–445.

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

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

[19] ISO/IEC, *ISO/IEC 25000:2005. Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. ISO/IEC, 2005.

[20] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus – Reverse Engineering Tool and Schema for C++," in *Proc. of the 18th Int. Conference on Software Maintenance (ICSM2002)*. IEEE Comp. Soc., Oct. 2002, pp. 172–181.

[21] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[22] X. Ge and E. Murphy-Hill, "Benefactor: A flexible refactoring tool for eclipse," in *Proc. of the ACM Int. Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA2011)*. ACM, 2011, pp. 19–20.

[23] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proc. of the 34th Int. Conference on Software Engineering (ICSE2012)*. IEEE Press, 2012, pp. 211–221.

[24] J. Henkel and A. Diwan, "Catchup!: Capturing and replaying refactorings to support api evolution," in *Proc. of the 27th Int. Conference on Software Engineering (ICSE2005)*. ACM, 2005, pp. 274–283.

[25] A. C. Jensen and B. H. Cheng, "On the use of genetic programming for automated refactoring and the introduction of design patterns," in *Proc. of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO2010)*. ACM, 2010, pp. 1341–1348.

[26] A. Feldthaus and A. Møller, "Semi-automatic rename refactoring for javascript," *SIGPLAN Not.*, vol. 48, no. 10, pp. 323–338, Oct. 2013.

[27] D. Silva, R. Terra, and M. T. Valente, "Recommending automated extract method refactorings," in *Proc. of the 22nd Int. Conference on Program Comprehension (ICPC2014)*. ACM, 2014, pp. 146–156.

[28] A. Christopoulou, E. A. Giakoumakis, V. E. Zafeiris, and V. Soukara, "Automated refactoring to the strategy design pattern," *Information and Software Technology*, vol. 54, no. 11, pp. 1202–1214, Nov. 2012.

[29] N. Yoshida, E. Choi, and K. Inoue, "Active support for clone refactoring: A perspective," in *Proc. of the 2013 ACM Workshop on Workshop on Refactoring Tools (WRT2013)*. ACM, 2013, pp. 13–16.

[30] B. Du Bois, "A study of quality improvements by refactoring," Ph.D. dissertation, University of Antwerp, 2006.

[31] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *Proc. of the 11th Working Conference on Reverse Engineering*. IEEE, 2004, pp. 144–151.

[32] Y. Yu, J. Mylopoulos, E. Yu, J. C. Leite, L. Liu, and E. D'Hollander, "Software refactoring guided by multiple soft-goals," in *Proc. of the 1st Workshop on Refactoring: Achievements, Challenges, and Effects, in conjunction with the 10th WCRE conference 2003*. IEEE Comp. Soc., 2003, pp. 7–11.

[33] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, July 2014.