Bulk Fixing Coding Issues and Its Effects on Software Quality: Is It Worth Refactoring?

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

***Abstract*—The quality of a software system is mostly defined by its source code. Software evolves continuously, it gets modified, enhanced, and new requirements always arise. If we do not spend time periodically on improving our source code, it becomes messy and its quality will decrease inevitably. Literature tells us that we can improve the quality of our software product by regularly refactoring it. But does refactoring really increase software quality? Can it happen that a refactoring decreases the quality? Is it possible to recognize the change in quality caused by a single refactoring operation? In our paper, we seek answers to these questions in a case study of refactoring large-scale proprietary software systems. We analyzed the source code of 5 systems, and measured the quality of several revisions for a period of time. We analyzed 2 million lines of code and identified nearly 200 refactoring commits which fixed over 500 coding issues. We found that one single refactoring only makes a small change (sometimes even decreases quality), but when we do them in blocks, we can significantly increase quality, which can result not only in the local, but also in the global improvement of the code.**

***Keywords*-refactoring; software quality; maintainability; coding issues; antipatterns; ISO/IEC 25010**

I. Introduction

It is a typical nature of software systems that they evolve over time, and while they evolve, they get enhanced, modified, and adapted to new requirements. As an effect of this evolution, the code usually becomes more complex and drifts away from its original design, thereby the quality of the software erodes as time passes. This is one reason why the major part of the total software development cost (about 80%) is spent on software maintenance tasks [1].

To prevent software erosion, refactoring is used to improve the quality of a system (e.g. extensibility, modularity, reusability, maintainability, and efficiency). Fowler defines refactoring as "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*" [2]. The refactoring approach of Fowler et al. was proposed mainly for improving understandability and changeability. However, further research work show that the idea can also be applied to other purposes [3], such as improving performance, security, and reliability. In fact, as our previous research [4] indicates, developers often tend to do refactorings to fix coding issues that clearly affect the quality of the system, instead of refactoring code smells or antipatterns.

Many researchers study the relation between refactoring and software quality, and they usually investigate different refactoring methods (mostly defined by Fowler et al. [2]) and their effects on metrics, such as complexity or coupling [5], [6], [7]. However, most of the published studies were performed in a controlled, *in vitro* environment and we have only brief knowledge on how these methods are used by developers in the real world.

Our goal is to fill this gap at least partially and investigate refactorings in an *in vivo* environment. In order to achieve this, we study the developers of software development companies actively working on proprietary, large-scale, real-world software systems. In a project, we had a chance to work together with five companies from the ICT sector who faced maintenance problems every day and wanted to improve the quality of their products. These companies have over 5 million lines of code altogether, and by taking part in this project they got extra budget for refactoring their own code. In the end, they committed around 1,600 fixes where they used manual refactoring techniques to do the modifications.

In an earlier research work [4] we investigated questionnaires that the developers filled before and after they manually refactored the code. Now, we investigate how developers decided to improve the quality of their source code and what was the real effect of the manual refactorings on the quality. For this study, we analyzed the quality of five selected systems of 2 companies who participated in the project – the revisions before and after the developers applied refactoring operations – using a quality model based on the ISO/IEC 25010 standard. We show which code smells developers decided to fix and how each refactoring changed the quality of the systems.

The primary contribution of this paper is the experience report that we learned from a large-scale experiment which was carried out in an *in vivo* industrial environment on refactoring. We show how developers behave when they get time to improve the quality of their code and that it is not always true that a refactoring actually improves the quality.

In the following, we present the background of the motivating refactoring project and we briefly introduce the main concepts of the ColumbusQM probabilistic quality model that we used for analysis. Then, we present the evaluation of the results of the analysis including the discussion of threats to validity and some lessons that we learned during the experiments. Finally, we conclude our paper and show plans for future work.

II. Overview

A. Motivating Project

This research 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 quality. During the project, we developed an automatic/semi-automatic refactoring framework and tested it 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 tools, but they also tested the toolset on the source code of their own product. This provided a good chance for them to refactor their own code and improve its quality.

Table I
Systems that we examined

Company	LOC	Domain
Comp. I.	200k	Specific Business Solutions
Comp. II.	4,300k	Enterprise Resource Planning (ERP)
Comp. III.	170k	Integrated Business Management
Comp. IV.	128k	Integrated Collection Management
Comp. V.	100k	Web-based PDF Generation

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.

In the initial steps of the project we asked the companies to manually refactor their 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 using static code analyzers to help them identifying code parts that should be refactored in their code (antipatterns or coding issues, for instance). Developers had to fill out a survey with questions targeting the initial identification steps and explaining why, how and what did they refactor for each refactoring commit to their code. There were around 40 developers involved in this step of the project (5-10 on average from each company) who were asked to fill out the survey and carry out the modifications in the code. (After this manual refactoring phase and the development of the refactoring framework, the developers performed also lots of automatic refactorings; however, the study of those activities lays outside the scope of this paper.)

In previous work we investigated which attributes drove the developers to select coding issues for refactorings, and which of these refactorings performed best. We also found that when developers get the extra time and budget to refactor their own code they really optimize the refactoring process to improve the quality of these systems.

Here, we study the impact of refactorings on the quality of the source code of these systems. We selected 5 systems with 548 refactorings in 198 commits. We analyzed the quality of the revisions where developers committed refactorings and the revisions before these commits. For the quality analysis we used the SourceAudit tool, which is a member of the QualityGate[1] product family of FrontEndART Ltd. This tool measures the source code quality based on the ColumbusQM probabilistic quality model [8] where the quality of the system is determined by several lower level characteristics (e.g. metrics or number of coding issues). SourceAudit is a software quality management tool, which allows automatic and objective assessment of the quality of a system.

B. Quality Model

In this section, we briefly introduce the ColumbusQM quality model [8]. The computation of the ISO/IEC 25010 [9] high level quality characteristics is based on a directed acyclic graph, whose nodes (sensors) correspond to quality properties that can be considered low-level or high-level attributes (see Figure 1).

The nodes that have no input edges are low level nodes (sensor nodes). These nodes characterize a software product from the developers' view, which means they are usually estimated by using source code metrics, or other source code properties (e.g. violating coding conventions). These properties can be calculated by static source code analysis. For this analysis QualityGate uses the free SourceMeter[2] tool (by FrontEndART Ltd.), which builds an Abstract Semantic Graph (ASG) from the source code, and uses this graph to calculate metrics, find code clones and check for coding issues such as unused code and empty catch blocks.

High level nodes (called aggregate nodes) characterize a software product from the end user's view, and they are aggregated from the low level and other high level nodes. These aggregated nodes have input and output edges as well. The edges of the graph show the dependencies between sensor nodes and aggregated nodes. Evaluating all the high level nodes is made by an aggregation along the edges of the graph, which is called Attribute Dependency Graph (ADG).

Typically, we want to know how good or bad an attribute is, but this is not a trivial question. In the model, we use the term goodness to express how good or bad an attribute is. The value of goodness is not known precisely, because it is represented by a random variable with a probability density function, which is called goodness function. To create a goodness function, we use the metric histogram over the code elements, as it characterizes the system from the aspect of only one metric (from one aspect). As goodness is a relative term, it is expected to be measured by means of

[1]http://quality-gate.com/
[2]http://sourcemeter.com/

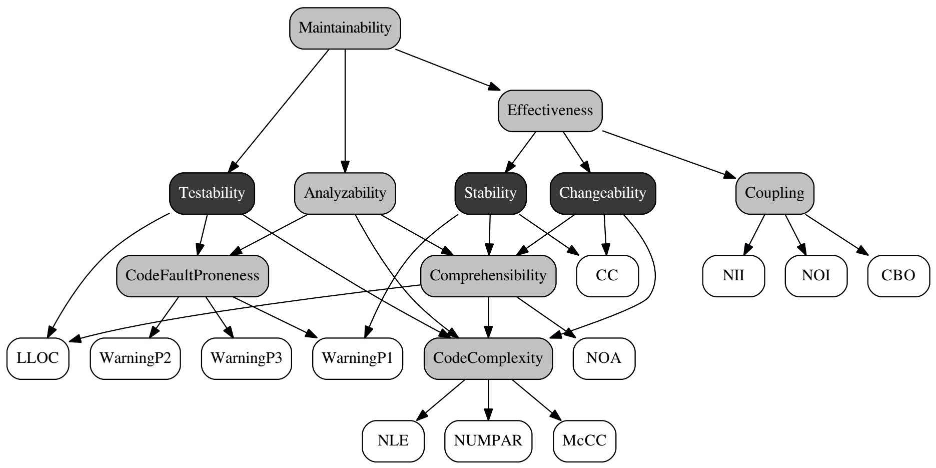

Figure 1. An overview of the attribute dependency graph of the Columbus Quality Model

comparison with other histograms. After applying the distance function between two histograms, we get one goodness value for the subject histogram. This is relative to the other histogram, however the goal is to get proper, independent goodness. It can be reached by repeating the comparison with lots of other, different histograms. With every comparison, we get a goodness value which can be basically regarded as a sample of a random variable from the range $[-\infty, \infty]$. Interpolation of the empirical density function leads us to the goodness function of the low level node. There is a way to aggregate the sensor nodes along the edges of the ADG. We held an online survey, where we asked many academic and industrial experts for their opinion about the weights between the quality attributes. The number assigned to an edge is considered to be the amount of contribution of source goodness to target goodness. Taking into account every possible combination of goodness values and weights, and the probabilities of their result, we defined a formula for computing goodness function for every aggregate node.

As we mentioned, every histogram gets compared with lots of other histograms. In order to do this, it is necessary to have a reference database (benchmark), which contains source code properties and histograms of numerous software systems. This benchmark is the basis for comparison of the software system to be evaluated. By using the same benchmark, quality becomes comparable among different software systems, or different versions of one system.

This qualification method is general and independent of the attribute dependency graph and the votes of experts. An attribute dependency graph for qualifying Java systems was developed by industrial and academic people. This is shown in Figure 1. More than 50 experts voted for the weights of the graph. After that, the reference database was built, containing the analysis results of more than a 100 industrial and open source Java systems. Now, objective qualifications for Java systems can be performed with this qualification method. In this paper when we refer to source code quality, we always mean maintainability.

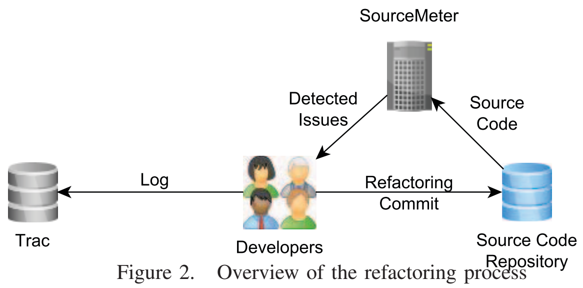

Figure 2. Overview of the refactoring process

III. Evaluation

A. Methodology

Figure 2 shows a brief overview of the manual refactoring phase of the project. In this phase developers of participating companies were asked to manually refactor their systems. For this manual refactoring we gave them support by analyzing their systems using a static source code analyzer, namely the SourceMeter tool (which is based on the Columbus technology [10]). Developers were aware of the results of these analyses and they had a thorough list of problematic code fragments. This list pointed out concrete 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). It was a project requirement to refactor their own code, hence improve its quality, but they were free to select how they go through that. So it was the developers' choice whether they fixed coding issues or improved the metrics of classes, for instance. However, the project expected that they fill out a survey in a ticketing system (Trac) and give a thorough explanation on what, why and how they refactored during their work. Besides filling out the survey, we asked them to provide the revision information so we could relate one refactoring to a Trac ticket and a revision in the version control system.

After the manual refactoring phase, we analyzed the marked revisions and investigated the change in the quality

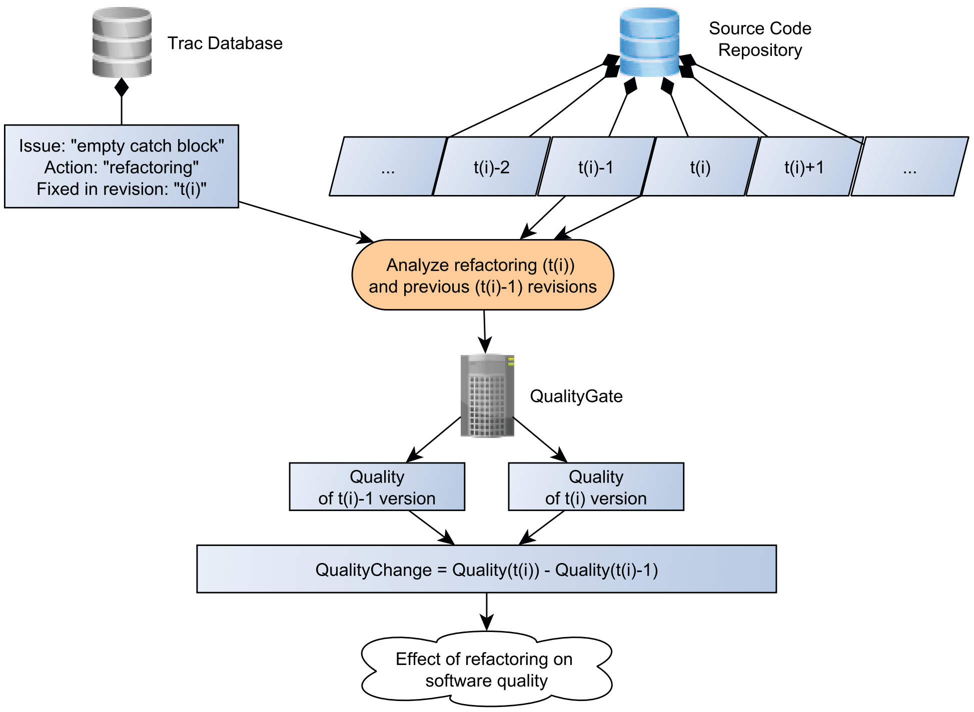
Trac Database
Source Code Repository
Issue: "empty catch block"
Action: "refactoring"
Fixed in revision: "t(i)"
...
t(i)-2
t(i)-1
t(i)
t(i)+1
...
Analyze refactoring (t(i)) and previous (t(i)-1) revisions
QualityGate
Quality of t(i)-1 version
Quality of t(i) version
QualityChange = Quality(t(i)) - Quality(t(i)-1)
Effect of refactoring on software quality

Figure 3. Overview of the analysis

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 can commit only refactorings to the version control system, or that they create a separate branch for this purpose. It was more realistic, and developers particularly asked us that they can commit these changes to the trunk or development branches, so they can develop their system in parallel with the refactoring process. Hence, for each system we identified the revisions (r_{t_1}, ..., r_{t_i}, ..., r_{t_n}) reported in the Trac system as refactoring commits and analyzed all these revisions with the revisions prior to each refactoring commit. As a result, the analyzed set of revisions for a system contained the 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, which is actually not a reported refactoring commit. Besides analyzing the quality of these revisions, we gathered data from the version control system as well, such as the patch of the commit and log messages.

We demonstrate the use of a simple refactoring through a sample coding issue that was actually fixed by the developers. In this example, we use the coding issue: Position Literals First In Comparisons. In Listing 1 there is a Java code sample, with a simple String comparison. This code is perfectly working, until we call the "printTest" method with a null reference. When we try to do this, according to the code, we would try to call a method of a null object. Of course, it is impossible, so we get a NullPointerException.

```
public class MyClass{
 public static void printTest(String a){
  if(a.equals("Test")) {
   System.out.println("This is a test!");
  }
 }
 public static void main(String[] args) {
  String a = "Test";
  printTest(a);
  a = null;
  printTest(a); // What happens?
 }
}
```

Listing 1. A code with a Position Literals First In Comparisons issue

To avoid this, we have to compare the String literal to the variable, not the variable to the literal. Fixing it is quite easy, we only swap the literal and variable, and that is all (see Listing 2). From now on, one can call the "printTest" method with a null object and will never get a Null Pointer Exception, as he or she never tries to call a method of a null object. This and similar refactorings are simple, but one can avoid critical or even blocker errors using them properly.

```
public class MyClass{
 public static void printTest(String a){
  if("Test".equals(a)) {
   System.out.println("This is a test!");
  }
 }
 public static void main(String[] args) {
  String a = "Test";
  printTest(a);
  a = null;
  printTest(a); // What happens?
 }
}
```

Listing 2. Sample refactoring of the code in Listing 1

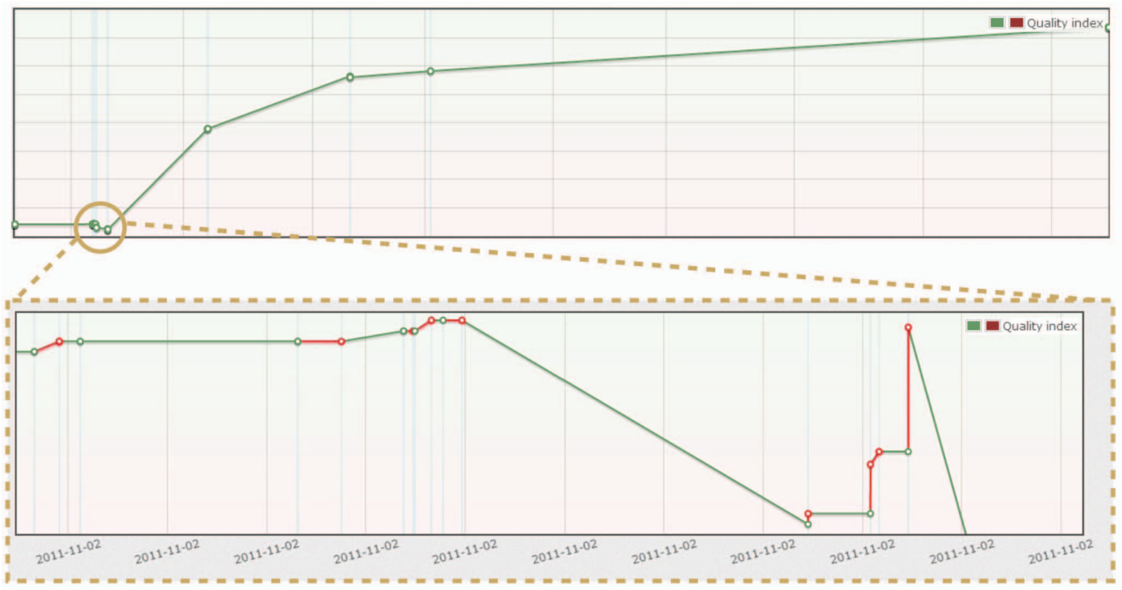

Figure 4. Quality of *Project A* over the refactoring period and a selected subperiod where we highlighted the changes in quality caused by refactoring commits with red color

B. Overall Change of Quality of the Systems

Here, we selected five systems that were involved in the project. The size of the selected systems and the number of analyzed revisions including the number of refactoring commits can be seen in Table II. All in all, we analyzed around 2.1 million lines of code with 450 revisions out which 198 were refactoring revisions. Developers made 548 refactorings with these commits. (It was allowed to commit more refactorings together in one patch – if it was really needed to do so).

Table II
SELECTED PROJECTS

Project	kLOC	Analyzed Revisions	Refactoring Commits	Refactorings
Project A	320	60	33	59
Project B	400	47	24	150
Project C	750	128	62	219
Project D	270	34	17	42
Project E	440	181	62	78
Total	**2180**	**450**	**198**	**548**

The first diagram in Figure 4 shows the overall change of the quality of *Project A* over the refactoring period. The diagram shows that the tendency of the change in quality was increasing during the period. However, this increasing tendency includes the normal development commits as well and not only the refactoring commits.

The second diagram in Figure 4 shows a sub-period and highlights with red color those revisions that were marked as refactoring commits; while the rest of the revisions – the normal commits of the development – are colored with green. It can be observed that those commits that were marked as refactorings noticeably increased the quality of the system, but in some cases the change does not seem to be significant and the quality remains unchanged. On the other hand, commits of normal development, sometimes increase and sometimes decrease the quality with larger variance. To further investigate these changes we study the impact of each type of refactoring one-by-one.

C. Effect of Different Types of Refactorings on the Quality

For each refactoring ticket we asked the developers to select what they wanted to improve with the commit:

- Did they try to fix a coding issue?
- Did they try to fix an antipattern?
- Did they try to improve a certain metric?

In some commits these may overlap, so it may happen, that a developer wants to fix a coding issue and he may improve a metric as well. However, this was not the case, and developers mostly handled these separately. On the other hand, there were metrics that they wanted to fix together.

1) Metrics: Table III shows the change in quality caused by refactoring commits whose purpose was to improve certain metrics. The first that we notice here is that the number of these kind of commits is very small compared to the total number of refactoring commits. It was definitely not the primary goal of the developers to improve the metrical values of their systems, although we reported them all the well-known complexity, coupling, and cohesion metrics at package, class and method levels. One might question how well trained were these developers and whether they were really familiar with the meaning of these metrics. To eliminate this factor, for each company, we held a training where we introduced them the main concepts of refactoring, code smells and gave them advanced introduction to metrics.

Table III
CHANGE IN QUALITY CAUSED BY COMMITS IMPROVING METRICS

Metrics	#	Avg. Change
McCC - McCabe's cyclomatic complexity, NOA - Number of ancestors	3	0.002299
NII - Number of incoming invocations	1	0.001645
NAni - Number of attributes (without inheritance)	1	0.001231
LOC - Lines of code	18	0.000458
NUMPAR - Number of parameters	5	0.000382
NMni - Number of methods (without inheritance)	1	0.000257
McCC - McCabe's cyclomatic complexity	1	0
NA - Number of attributes	1	0
U - Reuse ratio (for classes)	1	-0.00017

Most of the participating developers attended this training and not only juniors, but senior developers too.

Even with the small amount of commits whose purpose was to improve metrics, it can be seen, that complexity metrics (e.g. McCabe's cyclomatic complexity or Number of parameters) and size metrics (e.g. Lines of code) were the most familiar ones that developers intended to improve. The Avg. Change column of Table III shows the average of the measured changes in the quality caused by these commits. There were 3 refactorings where developers wanted to fix a class with high complexity and bad inheritance hierarchy at the same time; these commits had the greatest influence on quality. In 18 cases, developers wanted to decrease the LOC metric, and five times they fixed methods with too many parameters. It is also interesting to see that once they targeted the reuse ratio (e.g. to simplify the inheritance tree), and this resulted in a decrease in quality. One explanation is that if they want a better reuse ratio, they probably need to introduce a new class (inheriting from a superclass), which might add extra complexity or in the worst case additional coding issues or code clones to the code.

Table IV
CHANGE IN QUALITY CAUSED BY COMMITS FIXING ANTIPATTERNS

Antipattern	#	Avg. Change
Long Function, Duplicated Code	3	0.002299
Data Clumps	1	0.001231
Long Function	22	0.000517
Long Parameter List	5	0.000382
Large Class Code	2	0.000128
Duplicated Code	1	-0.00017

2) Antipatterns: Table IV shows the average of changes in quality when developers fixed antipatterns. Some antipatterns were identified with automatic analyzers (e.g. Long Function and Long Parameter List), but developers could spot antipatterns manually as well and report them to the ticketing system (Data Clumps is an example for an antipattern identified by a developer).

Like in the case of metrics, fixing antipatterns was not the primary concern of developers. Typically, they fixed Long Functions, Large Class Code or Long Parameter List. Most of these antipatterns could be also picked via metrics. Also, the resulting change in quality is similar to the corresponding metrics (see the average change for the LOC metric in Table III and for the Long Function antipattern in Table IV). It is interesting to see, that in one case where developers fixed copy&paste code, they decreased the quality. Another note is that developers fixed Long Function and Duplicated Code together three times, and a Data Clumps pattern once. Fixing these antipatterns might require a larger, global refactoring of the code (e.g. using Extract Class refactoring). These global refactorings indicated a significantly larger change in quality compared to others.

3) Coding Issues: Table V presents the average of measured quality changes where developers fixed coding issues. The relatively big number of refactorings shows that this was what developers really wanted to fix when they refactored their code base. Although, it is somehow uncertain whether fixing a coding issue can be considered as a refactoring or not. Fixing a Null Pointer Exception issue may perhaps change the execution (in a positive way), but it is questionable whether this change (fixing an unwanted bug) can be considered as a change in the observed external functionality of the program or not. On the other hand, it is clear that the purpose of fixing coding issues is to improve the quality of the code and not to modify its functionality. But, does it really improve the source code quality? Can it happen that fixing one issue may introduce another one? Is it possible that while the developer fixes an issue in a class, he ruins the metrics of the same class or some other classes too?

Table V shows the measured average, minimum, and maximum changes and the standard deviation. The coding issues in the rows are those issues which had at least one patch in the refactoring period of all the systems that we analyzed. Some of these coding issues are simple coding style guidelines which can be relatively easily fixed (e.g. IfStmtsMustUseBraces), while there are some issues which can indicate serious bugs and need to be fixed carefully (e.g. MethodReturnsInternalArray or OverrideBothEqualsAndHashCode). Issues that are easier to fix were refactored in larger numbers such as

Table V
CHANGE IN QUALITY CAUSED BY COMMITS FIXING CODING ISSUES

Coding issue	**#**	**Average**	**Min**	**Max**	**Deviation**
AvoidPrintStackTrace	32	0.00043140	0.00000000	0.00091337	0.00031712
BooleanInstantiation	47	-0.00009058	-0.00109062	0.00047086	0.00032689
BigIntegerInstantiation	21	-0.00015561	-0.00358699	0.00097444	0.00083454
ConsecutiveLiteralAppends	1	0.00000000	0.00000000	0.00000000	0.00000000
IfElseStmtsMustUseBraces	60	-0.00279348	-0.00387477	-0.00046913	0.00074460
IntegerInstantiation	84	-0.00034956	-0.00091754	0.00012738	0.00041582
InefficientStringBuffering	11	-0.00004679	-0.00064248	0.00012781	0.00020125
IfStmtsMustUseBraces	57	-0.00129054	-0.00374880	0.00052036	0.00171725
MethodReturnsInternalArray	8	0.00000000	0.00000000	0.00000000	0.00000000
OverrideBothEqualsAndHashcode	2	0.00000000	0.00000000	0.00000000	0.00000000
SimplifyConditional	39	0.00005174	0.00000000	0.00012700	0.00006285
SignatureDeclareThrowsException	18	0.00024588	0.00018497	0.00036990	0.00048685
UseIndexOfChar	45	0.00000000	0.00000000	0.00000000	0.00000000
UnnecessaryLocalBeforeReturn	43	0.00036712	0.00000000	0.00147897	0.00054782
UnusedModifier	31	0.00000000	0.00000000	0.00000000	0.00000000
UseStringBufferForStringAppends	14	0.00011752	0.00000000	0.00164534	0.00116343

IntegerInstantiation and BooleanInstantiation. It is not that much surprising that these issues had a relatively low impact on quality; however, it is interesting to see that some of them caused negative change in the quality.

In the case of the IfElseStmtsMustUseBraces and IfStmtsMustUseBraces issues the reason for the negative change in quality is the increased number of the code lines in the modified methods. The sensors of the quality model will change at low level: the number of issues and the LOC metric. These changes will affect the higher level, aggregated quality attributes like CodeFaultProneness and Comprehensibility and finally the Maintainability. A simple demonstration of this phenomenon is shown in Listings 3 and 4. A simple method with 5 lines could grow to 14 lines, if we apply all the necessary refactorings.

```
public static int doQuant(int n) {
  if ( n >= 0 && n < 86) return 0;
  else if (n > 85 && n < 170) return 128;
  else return 255;
}
```

Listing 3. Sample code with IfElseStmtsMustUseBraces issue. LOC: 5

```
public static int doQuant(int n) {
  if ( n >= 0 && n < 86)
  {
    return 0;
  }
  else if (n > 85 && n < 170)
  {
    return 128;
  }
  else
  {
    return 255;
  }
}
```

Listing 4. A sample refactoring of the code in Listing 3. LOC: 14

In the case of InefficientStringBuffering the reason for the negative change in quality is also the modified number of lines of code. There is an example code in Listing 5, that needs to be refactored.

```
String toAppend = "blue";
StringBuffer sb = new StringBuffer();
sb.append("The sky is" + toAppend);
```

Listing 5. A code with InefficientStringBuffering issue

Some of the developers fixed this issue, like it can be seen in Listing 6. This way, there are no new lines added to the code, and the effect of the refactoring is simple: one coding issue disappears.

```
String toAppend = "blue";
StringBuffer sb = new StringBuffer();
sb.append("The sky is").append(toAppend);
```

Listing 6. A sample refactoring of the code in Listing 5

But there were some developers, who preferred to fix the problem as it can be seen in Listing 7. This way, the issue disappears also, but there is a side effect: at least one new code line appears in the code, which affects again the lines of code metric, hence the quality.

```
String toAppend = "blue";
StringBuffer sb = new StringBuffer();
sb.append("The sky is");
sb.append(toAppend);
```

Listing 7. Another way of refactoring Listing 5

In some cases, the measured change in quality was 0. The reason for this lies in a pitfall of the quality model, as these minor priority issues were not taken into account by the quality model, hence when these issues were fixed, the model did not realize the change in the number of issues. Fixing these issues required only small local changes which did not influence other quality attributes either, so complexity and lines of code remained untouched, for instance. As a result, the measured change in quality was 0.

D. Discussion

After all, based on the results and the analysis, there are some additional observations that we found and discuss here.

Developers went for the easy refactorings: Although each participating company could take their time to perform large, global refactorings on their own code, numbers show that they did not decide to do so. They went for the easy tasks, for the small code smells, which they could fix rapidly. There might be several reasons for it, as fixing these code smells was relatively easy compared to others. Fixing a small issue which influences just the readability does not require thorough understanding of the code so a developer can easily see the problem and fix it even if it was not written by himself. In addition, testing is easier in these cases too. On the other hand, a big refactoring on the code requires better knowledge and understanding. It must be designed and applied carefully. It remains as a research question for future work that which choice is better in the long term in such a situation: fixing as many small issues as we can, or perform few, but large, global refactorings and restructure the code?

Developers did not refactor just to improve metrics or avoid antipatterns: Results show that developers did not really want to improve the metrics or to avoid certain antipatterns in their code; they simply went for the concrete problems and fixed coding issues. One reason that we must consider here is that developers probably did not really understand metrics and antipatterns. Although we are sure that they were aware of the meaning of some metrics and code smells (because we trained them for the project), they probably had no experience in identifying and fixing problematic classes with bad cohesion or coupling values, for instance. They were not quality experts who were experienced in studying reports of static analyzers. This somehow relates to the previous finding that developers chose the easier way and decided to fix concrete coding issues.

Developers learned to write better code during the refactoring period: All the systems that we studied in the refactoring period showed an improvement in source code quality even if we only take into account the revisions where they did not refactor the code, but just committed normal development patches. Developers admitted that they learned a lot from the static analysis and from refactoring coding issues. As an outcome they payed more attention to writing better code and to avoid these issues. The number of newly introduced issues in the new code was decreasing and they committed more simple and shorter classes and methods to their code.

Refactoring could be avoided if developers payed more attention to writing better quality code: According to the patches from the version control systems, one thing is clearly visible: if the developers pay a little more attention to the quality of their code (for example, they don't use double quotes for character literals), the process of refactoring could be avoided as these problems would never exist.

It is not guaranteed that one refactoring will improve the quality of the code: Quality measurements show that one refactoring might have a negative impact on the quality of the code, although its purpose is to improve it. It is not easy to decide how to fix an issue and balance its effects as it might happen that we want to improve one quality attribute, but we debase others.

Bulk fixing coding issues has a positive impact on the overall quality of the code: After the refactoring period, the overall quality of the system was always improving. Quality diagrams showed us that those commits which fixed more coding issues had a relatively higher impact on quality. Similarly, we saw in the tables that when developers fixed more metrics or antipatterns together they achieved a bigger change compared to others. Hence, bulk fixing coding issues has a positive impact on the quality which is measurable with static analysis techniques too.

E. Threats to Validity

We made our observations based on some hundreds of refactoring commits in five large-scale industrial systems. Like in similar case studies which were not 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.

Size of the sample set of refactoring commits investigated: The current sample set is definitely more realistic and larger than in similar research studies; however, with a larger sample set of refactorings we might have even a better basis for conclusions and a more precise view on refactorings. For future work, we plan to extend the sample set with the analysis of automatic refactorings too.

Quality analysis relies only on the Columbus Quality Model: The quality model is an important part of the analysis as it determines also what we consider as an "effect on quality" 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 [8]; however, we saw in the evaluation section that the model might miss some aspects which would reflect some changes caused by refactorings. Particularly, the model missed dealing with some low priority coding issues.

Limitations of the project: We claim that our experiment was carried out in an *in vivo* industrial context and developers were free to make their decisions for refactorings. On the other hand, the experiment was carried out at an initial phase of a project. This project might had unintentional effects on the research. E.g. the budget for refactoring was not "unlimited".

Limitations of the static analysis: We gave support to the developers in identifying coding issues with the help of a static analyzer. Of course, this was a great help for them in identifying problematic code fragments, but might have led the developers to concentrate only on the issues we reported. There is a risk here, that by using other analyzers or by not using any at all, we might get different results.

IV. Related Work

Restructuring the source code of an object-oriented program without changing its observed external behavior is called refactoring. Since Opdyke's PhD dissertation [11] – where the term was introduced – and Fowler's book [2] – where refactoring is used on "bad-smells" – many researchers studied refactoring as a technique to improve the maintainability of a software system. Mens et al. published a survey to provide an extensive overview of existing research in the area of software refactoring [12]. „Refactoring" is defined as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." That is to say, if applied well, refactorings improve the design of software, make it easier to understand, help to find bugs in it, and help to program faster [2], [13]. Unfortunately, it is unclear how specific quality factors are affected.

Many researchers studied refactoring but only a few papers analyze the impact of refactoring on software quality. Sahraoui et al. [5] use quality estimation models to study whether some object-oriented metrics can be used for detecting code parts where a refactoring can be applied to improve the quality of a software system. They do not validate their findings within an industrial case study or experiment. Tahvildari and Kontogiannis [14] investigate the use of metrics for detecting potential design flaws and suggest transformations for correcting them. Stroulia and Kapoor [6] investigate how size and coupling metrics behave after refactoring. They show that size and coupling metrics of a system decrease after refactoring; however, they only validate this in an academic environment. Bois et al. [7] propose refactoring guidelines for enhancing cohesion and coupling metrics and obtain promising results by applying them on an open-source project. Simon et al. [15] follow a similar strategy, they use a couple of metrics to visualize classes and methods which help the developers to identify the candidates for refactoring. Demeyer [13] shows that refactoring can have a beneficial impact on software performance (e.g. compilers can optimize better on polymorphism than on simple if-else statements). Bois and Mens [16] develop a framework for analyzing the effects of refactoring on internal quality metrics, but again, they do not provide an experimental validation in an industrial environment. Yu et al. [17] use a modeling framework for non-functional requirements and to study refactorings. They perform a case study, which shows that refactoring can be measured as a transformation on the state of the program in the quality space. Kataoka et al. [18] provide a quantitative evaluation of maintainability enhancement by refactoring. For the purpose of validation they analyze a project developed by a single developer, but do not provide any information on the development environment. Murphy et al. studied four methods to collect empirical data on refactorings [19]: mining the commit log, analyzing code histories, observing programmers and logging refactoring tool use. 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 [20]. Finally, Moser et al. [21] 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.

In our paper we observed a large number of refactorings and their effect on quality. These refactorings fixed different kinds of coding issues so we could investigate the work of developers applying different types of patches. Our work was carried out in an *in vivo* industrial environment which is an important difference compared to previous studies.

V. Conclusions and Future Work

Bakota et al. claim that the quality of a software product erodes over the years and if developers do not periodically and intentionally refactor the source code, then its quality will not improve [22].

In this paper we studied hundreds of refactoring commits from the refactoring period of five large-scale industrial systems developed by two companies, and we investigated the effects of these commits on source code quality using quality measurements of the QualityGate product of FrontEndART Ltd. We found interesting observations based on what and how developers refactored in the project. Among these observations we found that developers preferred to fix concrete coding issues rather than fixing code smells indicated by metrics or automatic smell detectors. It somehow reinforces the conclusion of our previous research [4], where we found that when developers get the extra time and budget to refactor their code, they really optimize their process to improve the quality effectively. As a final conclusion we claim that the effect of one refactoring on the global quality of the software product is hardly predictable; moreover, it might sometimes have a negative effect on the quality. However, when we refactor our code systematically, it has a significant positive effect on the quality. The reason for that is not only because we improve the quality of our software, but also the developers who do the refactorings will pay more attention on writing better quality code.

These five systems and their manual refactorings represented only a small portion of the full code base that we investigated during this research project. We gathered additional data from the developers and from the automatic tool guided refactoring period as well. This information is from an *in vivo* environment and we can learn a lot from it.

For future work, we plan to further investigate and seek answers for more questions that arise when developers start to work with refactoring. Just a few examples are: What should I refactor? How should I do this? Can I automate it somehow? What should I take care of or be afraid of? How much time will it take? Is it actually worth to do it?

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 all the participants of this project for their help and cooperation.

References

[1] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.

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

[3] 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.

[4] 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 *Proceedings of Computational Science and Its Applications–ICCSA 2014*. Springer, 2014.

[5] H. A. Sahraoui, R. Godin, and T. Miceli, "Can metrics help to bridge the gap between the improvement of oo design quality and its automation?" in *Proceedings of International Conference on Software Maintenance*. IEEE, 2000, pp. 154–162.

[6] E. Stroulia and R. Kapoor, "Metrics of refactoring-based development: An experience report," in *OOIS 2001*. Springer, 2001, pp. 113–122.

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

[8] T. Bakota, P. Hegedus, P. Kortvelyesi, R. Ferenc, and T. Gyimothy, "A probabilistic software quality model," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ser. ICSM '11. IEEE Computer Society, 2011, pp. 243–252.

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

[10] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus – Reverse Engineering Tool and Schema for C++," in *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*. IEEE Computer Society, Oct. 2002, pp. 172–181.

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

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

[13] S. Demeyer, "Refactor conditionals into polymorphism: what's the performance cost of introducing virtual calls?" in *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05.* IEEE, 2005, pp. 627–630.

[14] L. Tahvildari and K. Kontogiannis, "A metric-based approach to enhance design quality through meta-pattern transformations," in *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*. IEEE, 2003, pp. 183–192.

[15] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*. IEEE, 2001, pp. 30–38.

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

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

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

[19] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin, "Gathering refactoring data: A comparison of four methods," in *Proceedings of the 2nd Workshop on Refactoring Tools*, ser. WRT '08. ACM, 2008, pp. 7:1–7:5.

[20] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Proceedings of the 5th International Workshop on Software Quality*. IEEE Computer Society, 2007, p. 10.

[21] 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.

[22] T. Bakota, P. Hegedus, G. Ladanyi, P. Kortvelyesi, R. Ferenc, and T. Gyimothy, "A cost model based on software maintainability," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2012, pp. 316–325.