Empirical Study on Refactoring Large-Scale Industrial Systems and Its Effects on Maintainability

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

PII: S0164-1212(16)30155-8
DOI: 10.1016/j.jss.2016.08.071
Reference: JSS 9833

To appear in: *The Journal of Systems & Software*

Received date: 16 February 2015
Revised date: 10 August 2016
Accepted date: 23 August 2016

Please cite this article as: Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, Tibor Gyimóthy, Empirical Study on Refactoring Large-Scale Industrial Systems and Its Effects on Maintainability, *The Journal of Systems & Software* (2016), doi: 10.1016/j.jss.2016.08.071

Highlights

- We examine hundreds of manual refactoring commits from large-scale industrial systems.
- We study the effects of these commits on source code using a maintainability model.
- Developers preferred to fix concrete coding issues rather than fix code smells.
- A single refactoring had only a small impact (sometimes even negative effect).
- Whole refactoring process has significant beneficial effect on the maintainability.

Empirical Study on Refactoring Large-Scale Industrial Systems and Its Effects on Maintainability

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

Department of Software Engineering
University of Szeged, Hungary

Abstract

Software evolves continuously, it gets modified, enhanced, and new requirements always arise. If we do not spend time occasionally on improving our source code, its maintainability will inevitably decrease. The literature tells us that we can improve the maintainability of a software system by regularly refactoring it. But does refactoring really increase software maintainability? Can it happen that refactoring decreases the maintainability? Empirical studies show contradicting answers to these questions and there have been only a few studies which were performed in a large-scale, industrial context. In our paper, we assess these questions in an in vivo context, where we analyzed the source code and measured the maintainability of 6 large-scale, proprietary software systems in their manual refactoring phase. We analyzed 2.5 million lines of code and studied the effects on maintainability of 315 refactoring commits which fixed 1,273 coding issues. We found that single refactorings only make a very little difference (sometimes even decrease maintainability), but a whole refactoring period, in general, can significantly increase maintainability, 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

1. Introduction

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

Email addresses: gabor.szoke@inf.u-szeged.hu (Gábor Szőke), antal@inf.u-szeged.hu (Gábor Antal), ncsaba@inf.u-szeged.hu (Csaba Nagy), ferenc@inf.u-szeged.hu (Rudolf Ferenc), gyimi@inf.u-szeged.hu (Tibor Gyimóthy)

to prevent the negative effects of this *software erosion*, and to improve the maintainability is to perform refactoring tasks regularly.

After the term *refactoring* was introduced in the PhD dissertation of Opdyke [2], Fowler published a catalog of refactoring transformations, where he defined 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*" [3]. Researchers quickly recognized that this technique can also be applied to other areas, such as improving performance, security, and reliability [4]. Many researchers have started to studying the relation between refactoring and maintainability too, and they usually investigate different refactoring methods (mostly from Fowler's catalog [3]) and their effect on code metrics, such as complexity and coupling [5, 6, 7].

Kim et al. [8] found in their study that, in practice, developers' views on refactoring usually differ from the academic ones. As our previous study [9] indicates it too, developers often tend to do refactoring to fix coding issues (e.g. coding rule violations identified by static analyzers) that clearly affect the maintainability of the system, instead of refactoring code smells or antipatterns.

Empirical studies show contradicting findings on the benefits of refactoring. E.g., Ratzinger et al. [10] say that increasing the number of refactoring edits can decrease the number of defects, while Weißgerber [11] and Diehl say that a high ratio of refactoring edits is often followed by an increasing ratio of bug reports. Most of these studies were performed on open-source systems or in controlled *in vitro* environment, and there are relatively few studies in a large-scale, industrial context.

In this study, we investigate refactorings from the developers' point of view, in an *in vivo* environment by studying the developers of software development companies working on large-scale, proprietary software systems. In a project, we had a chance to work together with five software development companies who faced maintenance problems every day and wanted to improve the maintainability of their products. By taking part in this project, they got an extra budget to refactor their own source code. The systems of these companies, which we selected for our study, consisted of about 2.5 million lines of code altogether and in the end, their developers committed 1,273 source code fixes where they used manual refactoring techniques to make the modifications.

The primary contribution of this article is the experience report of what we learned from our large-scale experiment, which was carried out in this *in vivo* industrial environment on refactoring.[1] We explore the data set that we gathered by addressing the following motivating research questions:

- Is it possible to recognize the change in maintainability caused by a single refactoring operation with a probabilistic quality model based on code metrics, coding issues and code clones?

[1]Parts of the results of this study were first presented in our conference paper [12]. Here, we almost double the number of subject systems, show more details, draw further conclusions, and provide an online appendix to make our study reproducible.

- Does refactoring increase the overall maintainability of a software system?
- Can it happen that refactoring decreases maintainability?

In the following, we present the background of the motivating refactoring project in Section 2, where we also briefly introduce the main concepts of the ColumbusQM probabilistic maintainability model that we used to measure the maintainability changes in the source code. Then, in Section 3, we present the results of our analysis including some interesting observations that we obtained during the experiments. After, we discuss threats to validity in Section 4. In Section 5 we present related work, and finally, in Section 6 we draw some conclusions and describe plans for future work.

2. Overview

2.1. Motivating Project

This research work was part of an R&D project supported by the EU and the Hungarian State. The goal of the two-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 industrial partners, having an *in vivo* environment and live feedback on the tools. Hence partners not only participated in this project by helping to develop the refactoring tools, but they also tested and used the toolset on the source code of their own product. This provided a good opportunity for them to refactor their own code and improve its maintainability.

Five experienced software companies were involved in this project. They were founded in the last two decades and they started developing some of their systems before the millennium. The systems that we selected for this study consist of about 2.5 million lines of code altogether, are written mostly in Java, and cover different ICT areas like ERPs, ICMs and online PDF Generators (see Table 1).

Table 1: 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

In the initial steps of the project we asked the companies to manually refactor their code, and provide detailed documentation of each refactoring, explaining the main reasons and the steps of how they improved the targeted code fragment. We gave them support by using static code analyzers to help them identify code

parts that should be refactored in their code (antipatterns or coding issues, for instance). Developers had to fill out a survey for each refactoring commit. This survey contained questions targeting the initial identification steps and they also had to explain why, how and what they changed in their code. There were around 40 developers involved in this phase 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. Based on the results of this manual refactoring, we designed and implemented a refactoring framework with the companies. This framework helped them in the final phase of the project to perform automatic refactorings. In this study, we report data that we gathered during the manual refactoring phase.

In our previous study [9], we examined the questionnaires that were filled out by the developers before and after they manually refactored the code. We investigated which attributes drove the developers to select coding issues for refactorings, and which of these performed best. We found that these companies, when they had extra time and a budget, actually optimized their refactoring process to improve the maintainability of their systems (i.e., what they thought would improve maintainability). Here, we take a closer look at what they really did in the source code, and examine the impact of their refactoring commits on the maintainability of the system through static analysis.

We selected 6 systems, and for each system[2] we analyzed the maintainability of the revisions where developers committed refactorings and the revisions before these commits. For the maintainability analysis we used the SourceAudit tool, which is a member of the QualityGate[3] product family of FrontEndART Ltd. This tool measures the source code maintainability based on the ColumbusQM probabilistic quality model [13], where the maintainability of the system is determined by several lower level characteristics (e.g. metrics and coding issues). SourceAudit is a software quality management tool that allows the automatic and objective assessment of the maintainability of a system.

These maintainability analyses were performed after the manual refactoring phase of the systems, and were independent of the above mentioned reports of the static analyzers. The changes in maintainability were not shown to the developers during the refactoring phase. The goal was to observe the changes without affecting how the developers planned their manual refactorings.

2.2. Quality Model

We briefly introduce the ColumbusQM quality model[4], which is based on the ISO/IEC 25010 [14] international standard for software product quality.

[2]We ended up having only six system because Company V bankrupted after the manual refactoring phase and we were not able to get access to their code for the analysis, just the surveys. For this, we omit Company V from the rest of the article.

[3]QualityGate product home page – `http://quality-gate.com/`

[4]Detailed description of the ColumbusQM quality model is available in the work of Bakota et al. [13]

Thanks to the probabilistic approach, this model integrates the objective, measurable characteristics of the source code (e.g. code metrics) and expert knowledge, which is usually ambiguous. At the lowest level, the following properties are considered by the model:

- source code metrics (e.g. some C&K metrics),
- source code duplications (copy&pasted code fragments),
- coding rule violations (e.g. coding style guidelines, coding issues).

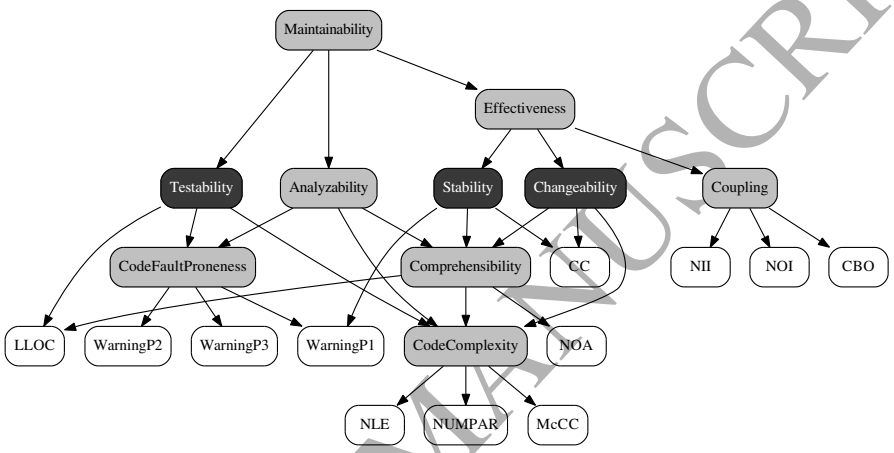

Figure 1: An overview of the attribute dependency graph of ColumbusQM [13]. Unfilled nodes represent the sensor nodes (code metrics, number of coding rule violations, number of code clones, etc.) in the model. Aggregated nodes (both light and dark gray nodes) are calculated from these sensor nodes or other aggregated nodes. They were either defined by the ISO/IEC 25010 standard (dark gray) or introduced for showing further external maintainability attributes (light gray).

The computation of the standard's high-level quality characteristics is based on a *directed acyclic graph* (DAG), whose nodes correspond to quality properties that can be considered low-level or high-level attributes (see Figure 1). The nodes without input edges are low-level nodes (*sensor nodes* – shown in white). These characterize a software system from the developers' view, so their calculation is based on 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[5] tool (by FrontEndART Ltd.), which builds an *abstract semantic graph* (ASG) from the source code, and it uses this graph to calculate metrics, find code clones (duplications) and to find coding issues such as unused code and empty catch blocks.

[5]SourceMeter product home page – `http://sourcemeter.com/`

High-level nodes (called *aggregate nodes*) characterize a software system from the end user's view. They are calculated as an aggregation of the low-level and other high-level nodes. In addition to the aggregate nodes which are defined by the standard (dark gray nodes), there are also some new ones that were introduced for showing further external maintainability attributes (light gray nodes). These 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 performed by an aggregation along the edges of the graph, which is called the *attribute dependency graph* (ADG).

Typically, we want to know how good or bad an attribute is in terms of maintainability. We use the term *goodness* to express this with the help of the model. To include some degree of uncertainty in the value of goodness, it is represented as a random variable with a probability density function, which is called the *goodness function*. The goodness function is based on the metric histogram over the code elements, as it characterizes the system from the aspect of one metric (from one aspect). As goodness is a relative term, it is expected to be measured by means of comparison with other histograms. After applying the distance function between two histograms, we get a goodness value for the subject histogram. This value will be relative to the other histogram, but the goal is to be independent. Although, the result will always depend on the histograms in the benchmark (see below), we can get a better estimate by repeating the comparison with a larger set of systems in the benchmark. For every comparison, we get a goodness value which can be basically regarded as a sample of a random variable over the range $[-\infty, \infty]$. Interpolation of the empirical density function leads us to the goodness function of the low-level nodes. There is also a way to aggregate the sensor nodes along the edges of the ADG. Bakota et al. [13] held an online survey, where they asked academic and industrial experts for their opinions about the weights of the quality attributes. The number assigned to an edge is considered to be the degree of contribution of source goodness to target goodness. Taking into account every possible combination of goodness values and weights, and the probability values of their result, they defined a formula to compute the goodness function for each aggregate node. Finally, the top-level node in the ADG, maintainability, will have an aggregated value over the interval $[0, 10]$.

As we mentioned before, each histogram gets compared to several 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 the comparison of the software system to be evaluated. By applying the same benchmark, quality becomes comparable among different software systems, or different versions of one system.

This qualification methodology is general and independent of the ADG and the votes of the experts. But the latter is language specific, resulting in the need for language-specific ADGs. The ADG for Java is shown in Figure 1, which was constructed based on the opinions of over 50 experts. The benchmark for Java contains the analysis results of over 100 industrial and open-source Java

systems.

3. Evaluation

3.1. Methodology

Figure 2 gives 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 provided support by analyzing their systems using a static source code analyzer, namely the SourceMeter tool (which is based on the Columbus technology [15]). Developers were aware of the results of these analyses and they had access to the reports including a list of problematic code fragments. This list pointed out concrete coding issues, antipatterns (e.g. duplicated code and long functions) and source code elements with problematic metrics at different levels (e.g. classes/methods with excessive complexity and classes with bad coupling or cohesion metric values).

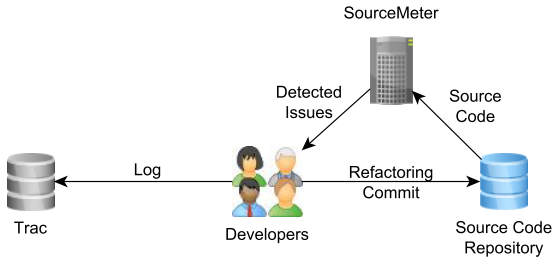

Figure 2: Overview of the refactoring process. SourceMeter provided a list of potential problems in the code. Developers could freely choose one of these, or identify a new one, which they fixed and committed to the version control system. They also had to fill out a survey for each refactoring in the ticketing system (Trac).

In the project, the companies' programmers were required to refactor their own code, hence improve its maintainability, but they were free to choose how they wanted to do it. They could freely choose any coding issues or metrics from the reported problems, and they were also free to identify additional problems in the code by themselves. However, the project required that they filled out the survey (in a Trac ticketing system) and that they gave a thorough explanation on what, why and how they refactored during their maintenance work. Besides filling out the survey, we asked them to provide revision information so we could map one refactoring to a Trac ticket and a revision in the version control system (Subversion, Mercurial).

After the manual refactoring phase, we analyzed the marked revisions to assess the change in the maintainability of the systems caused by refactoring commits. Figure 3 gives an overview of this process. It was not a requirement of 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 asked us in particular to commit these changes to the trunk

or development branches so they could 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}) that were reported in the Trac system as refactoring commits, and we analyzed all these revisions along with the revisions prior to them. As a result, we considered 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, which is actually not a reported refactoring commit.

We performed an analysis of these revisions of the source code via the QualityGate SourceAudit tool, mentioned earlier in Section 2.1, which uses the maintainability model described in Section 2.2. To be able to calculate the changes in the maintainability, we had to analyze the whole code base for each revision. That is, a commit with a small local change may also have an impact on some other parts of the source code. E.g., a small modification in a method may result in the appearance of a new clone instance, or changes in coupling metric values of some other classes. Besides analyzing the maintainability of these revisions, we collected data from the version control system as well, like diffs and log messages.

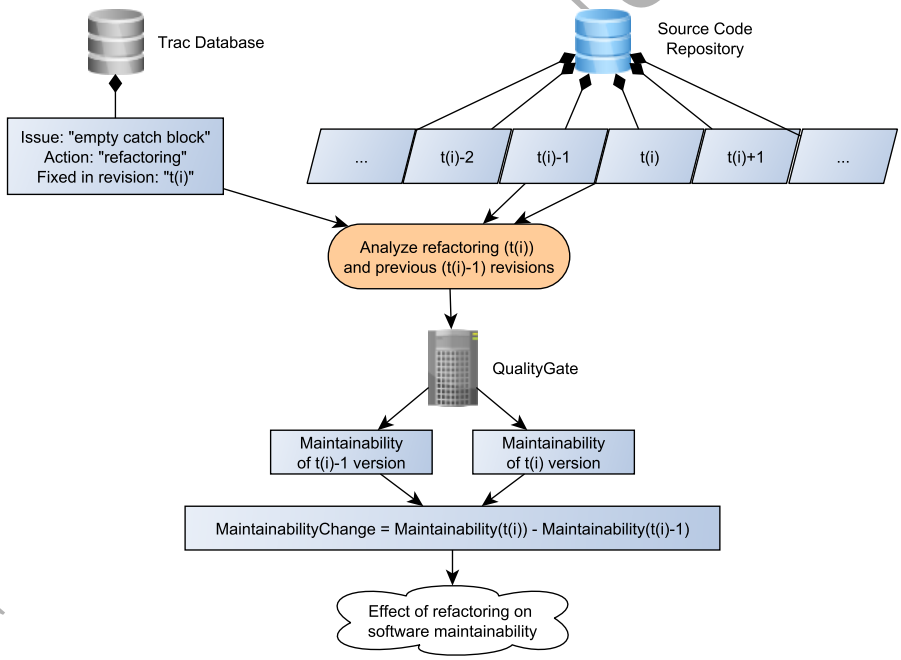

Figure 3: Overview of the analysis process. We identified the refactoring commits based on the tickets in Trac, and analyzed maintainability of the revisions before/after refactoring commits.

We will now illustrate the use of a simple refactoring through a coding issue that was actually fixed by the developers. In this example, we show the 'Position Literals First In Comparisons' coding issue. In Listing 1, there is a Java code

example with a simple String comparison. This code works perfectly until we call the 'printTest' method with a null reference. By doing so, we would call a method of a null object, and the JVM would throw a NullPointerException.

```
1  public class MyClass{
2   public static void printTest(String a){
3    if(a.equals("Test")) {
4     System.out.println("This is a test!");
5    }
6   }
7   public static void main(String[] args) {
8    String a = "Test";
9    printTest(a);
10   a = null;
11   printTest(a); // What happens?
12  }
13 }
```

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

To avoid this problem, we have to compare the String literal with the variable instead of comparing the variable with the literal. So to fix this issue, we simply swap the literal and the variable in the code, as can be seen in Listing 2. Thanks to this fix, one can safely call the 'printTest' method with a null object and we do not have to worry about a null pointer exception. This and similar refactorings are simple, but we can avoid critical or even blocker errors.

```
1  public class MyClass{
2   public static void printTest(String a){
3    if("Test".equals(a)) {
4     System.out.println("This is a test!");
5    }
6   }
7   public static void main(String[] args) {
8    String a = "Test";
9    printTest(a);
10   a = null;
11   printTest(a); // What happens?
12  }
13 }
```

Listing 2: Sample refactoring of the code in Listing 1

3.2. Overall Change of Maintainability of the Systems

Table 2 shows the size of the six selected systems and the number of analyzed revisions including the number of refactoring commits. Recall that we determined the refactoring revisions from the ticketing system as those revisions which were marked by the developers as refactoring commits. In addition, we analyzed the non-refactoring revisions prior to the refactoring revisions in order to calculate the change in maintainability (see Section 3.1). All in all, we analyzed around 2.5 million lines of code with 732 revisions, out of which 315 were refactoring commits. Developers made 1,273 refactoring operations with these commits. Notice that the project allowed the developers to commit more

refactorings together in one patch, but one commit had to consist of the same type of refactoring operations. So one commit possibly included the necessary code transformations to fix more Position Literals First issues, but it could not happen that a different type of coding issue was also fixed in it.

Table 2: Main characteristics of the selected systems: lines of code, total number of analyzed revisions, number of refactoring commits, number of refactoring operations.

System	Company	kLOC	Analyzed Revisions	Refactoring Commits	Refactorings
System A	Comp. I.	1,740	269	136	470
System B	Comp. II.	440	180	38	78
System C	Comp. III.	170	78	15	597
System D	Comp. IV.	38	37	16	18
System E	Comp. IV.	11	57	40	40
System F	Comp. IV.	50	111	70	70
	Total	**2,449**	**732**	**315**	**1,273**

The first diagram in Figure 4 shows the change in the maintainability (between each pair of refactoring and its predecessor) of *System A* during the refactoring period. The diagram shows that maintainability of the system increased over time; however, this tendency includes the normal development commits as well and not only the refactoring commits.

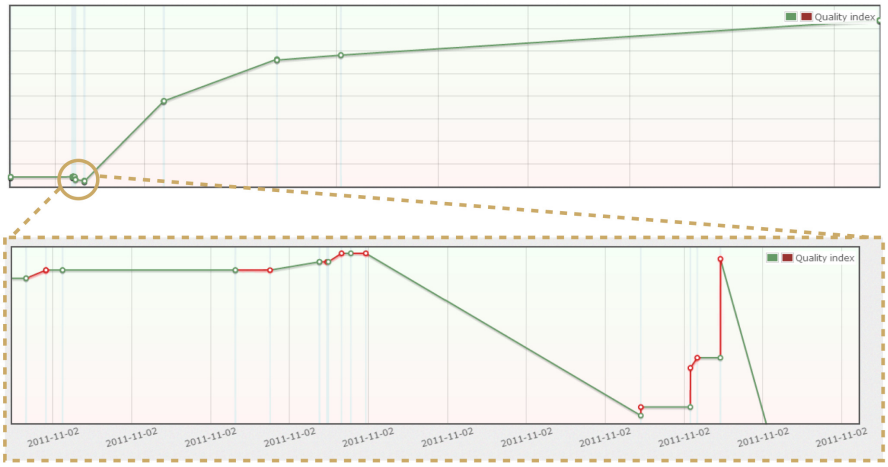

Figure 4: Maintainability of *System A* over the refactoring period and a selected subperiod where we highlighted in red the changes in maintainability caused by refactoring commits

The second diagram in Figure 4 shows a sub-period and highlights in red those revisions that were marked as refactoring commits, while the green part indicates the rest of the revisions (i.e, the ones preceding a refactoring commit) which were the normal development commits. It can be seen that those commits that were marked as refactorings noticeably increased the maintainability

of the system, but in some cases the change does not seem to be significant and the maintainability remains unchanged. However, commits of normal development sometimes increase and sometimes decrease the maintainability with larger variance.

Table 3: Number of commits which increased or decreased the maintainability of the systems

System	Negative	Zero	Positive
System A	17	94	25
System B	3	18	17
System C	2	5	8
System D	1	7	8
System E	13	9	18
System F	8	30	32
Total	**44**	**163**	**108**

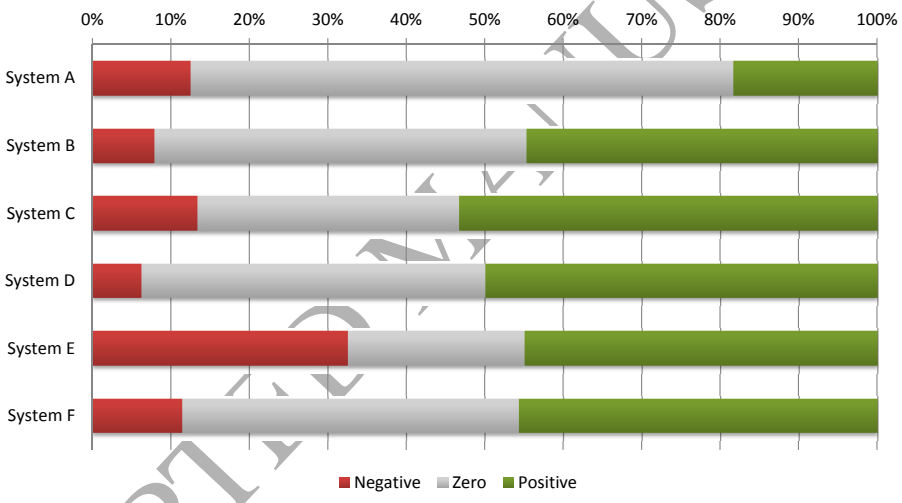

Figure 5: Normalized percentages of commits with a negative/zero/positive impact on maintainability (negative - red, zero - gray, positive - green)

Table 3 lists the number of commits for each system which had a positive (or negative) impact on maintainability. If a commit increased the maintainability value it had a positive (beneficial) impact; if it decreased, it had a negative (detrimental) impact; otherwise it did not affect the sensors of the quality model and its impact is considered zero (neutral). As can be seen in Figure 5, the results show that for all of the systems the beneficial effects outnumber the detrimental ones. Interestingly, it also indicates that a large proportion of the commits did not have an observable impact on maintainability. The main reason for this is that ColumbusQM does not recognize all the coding issues that were fixed by the developers. As the developers were not aware of the ColumbusQM model, their aim was simply to improve their code. This included some

fixes of coding issues that were detected by the ColumbusQM only when the refactoring affected some source code metrics. (Section 3.3.3 elaborates on these refactorings.)

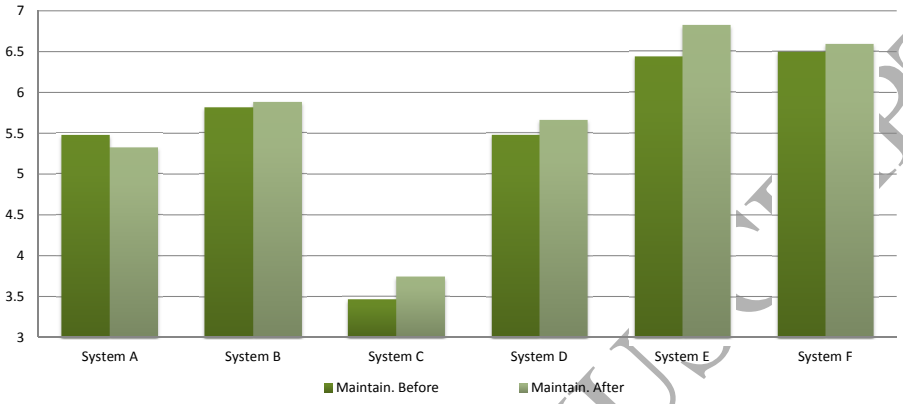

Figure 6: Maintainability of the projects before and after the refactoring period

Figure 6 shows the maintainability values that we measured before and after the refactoring period of each system in question, and Table 4 lists details on how the maintainability increased or decreased for these systems. Recall that the value of maintainability can be between 0 and 10, where 0 describes a system with the hardest maintainability, and 10 indicates a system which is very easy to maintain. The 'Metrics', 'Antipatterns' and 'Coding Issues' columns show for each system the number of different kinds of refactorings that were fixed. Note that they could have fixed more issues with one commit, so it might happen that the aim of a fix was to improve some metrics and eliminate antipatterns together. The 'Total Impr.' column shows the difference; that is, the maintainability improvement at the end of the project. 'Ref. Impr.' shows the total of the maintainability changes caused by refactoring commits only; hence it shows how refactoring commits improved the maintainability.

Table 4: Maintainability of the systems before and after the refactoring period

System	Metrics	Anti-patterns	Coding Issues	Maintain. Before	Maintain. After	Total Impr.	Ref. Impr.
System A	0	0	470	5.4699	5.3193	-0.1506	-0.0030
System B	32	34	43	5.8095	5.8762	0.0667	0.0135
System C	15	13	595	3.4629	3.7354	0.2725	0.0767
System D	3	0	17	5.4775	5.6594	0.1819	0.0151
System E	14	8	31	6.4362	6.8190	0.3828	0.0436
System F	15	11	42	6.4972	6.5926	0.0954	0.0716

We measured positive change in the maintainability of five systems out of six and in the case of System F, 75.05% of the maintainability improvement was caused by refactoring commits. Notice, however, that for System A, main-

tainability decreased by the end of the refactoring period (it had the biggest detrimental impact ratio in Table 3). Also, this system had the largest code base among the systems analyzed and its developers decided to fix only coding issues.

3.3. Effect of Different Types of Refactorings on the Maintainability

To further investigate the changes made during the refactoring period, we will study the impact of each type of refactoring. For each refactoring ticket, we asked the developers to select what they wanted to improve with the commit:

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

In practice, it may happen that a developer wants to fix a coding issue and he may improve a metric value as well in the same commit. Also, many metrics correlate with antipatterns (e.g. large class/long method correlate with LOC). However, in the project developers mostly handled these separately. For coding issues, we asked them in particular to commit refactorings of only one certain kind of issue per commit. But this also means that they were allowed to refactor more from the same kind of coding issue in one commit.

3.3.1. Metrics

Table 5 shows the change in maintainability caused by refactoring commits, where the goal of the developers was to improve certain metrics. (See Table 6 for a detailed description of these metrics.) The first thing that we notice here is that the number of these refactorings (74) is very small compared to the total number of refactorings (1,273). It was definitely not the primary goal of the developers to improve the metric values of their systems, although we told them about all the well-known complexity, coupling, and cohesion metrics at the package, class and method levels. One might doubt how well trained these developers were 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 the main concepts of refactoring and code smells, and then gave them an advanced introduction to metrics. Most of the participating developers attended this training session, including juniors and senior developers as well.

Among those refactorings which fix metrics, it can be seen that complexity metrics (e.g. McCabe's cyclomatic complexity and Number of parameters) and size metrics (e.g. Lines of code) were the most familiar ones that developers intended to improve. The *Average* column of Table 5 lists the average of the measured changes in the maintainability caused by these commits. The first entry in the table shows a refactoring which was performed because of the high value of the Number of defined methods metric. In this case, developers realized that they had similar methods in a few of their classes (methods for serialization and deserialization). They did a *Pull-up method* refactoring, which

Table 5: Change in maintainability caused by commits improving metrics

Metrics	#	Average	Min	Max	Deviation
NMD	1	0.005252	0.005252	0.005252	0.000000
COF	3	0.002691	0.000000	0.006546	0.003425
McCC + NOA	3	0.002299	0.002299	0.002299	0.000000
CLB	10	0.001662	-0.007803	0.017286	0.006616
NII	1	0.001645	0.001645	0.001645	0.000000
McCC	2	0.001323	0.000000	0.002647	0.001872
NA	1	0.001231	0.001231	0.001231	0.000000
LOC	38	0.001007	-0.007617	0.011233	0.003687
NUMPAR	5	0.000382	-0.000108	0.001113	0.000578
NM	1	0.000257	0.000257	0.000257	0.000000
NLE	4	0.000047	0.000047	0.000047	0.000000
NA	1	0.000000	0.000000	0.000000	0.000000
U	2	-0.000083	-0.000165	0.000000	0.000117
NOS	1	-0.000167	-0.000167	-0.000167	0.000000
NOI	1	-0.004062	-0.004062	-0.004062	0.000000

Table 6: Description of metrics

Abbreviation	Description
NMD	Number of defined methods
COF	Coupling factor
McCC	McCabe's cyclomatic complexity
NOA	Number of ancestors
CLB	Comment lines before class/method/function
NII	Number of incoming invocations
NA	Number of attributes (without inheritance)
LOC	Lines of code
NUMPAR	Number of parameters
NM	Number of methods (without inheritance)
NLE	Nesting level
NA	Number of attributes
U	Reuse ratio (for classes)
NOS	Number of statements
NOI	Number of outgoing invocations

reduced the number of defined methods in the code and had a beneficial impact on the maintainability. Developers also tried to decrease the Coupling factor in their systems with *Move method* and *Move field* refactorings (second row of the table). There were three refactorings where developers attempted to fix a class with high complexity and bad inheritance hierarchy at the same time. In 38 cases, developers wanted to decrease the LOC metric, and five times they fixed methods with too many parameters. It is also interesting to observe that once they targeted the reuse ratio (e.g. to simplify the inheritance tree) and this resulted in a decrease in maintainability. One explanation is that if they wanted a better reuse ratio, they probably needed to introduce a new class (inheriting from a superclass), which might increase the complexity or in the worst case introduce new coding issues or code clones.

3.3.2. Antipatterns

Table 7 shows the average of changes in maintainability 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.)

Table 7: Change in maintainability caused by commits fixing antipatterns

Antipattern	#	**Average**	**Min**	**Max**	**Deviation**
Duplicated Code	11	0.003527	-0.007803	0.011233	0.005195
Long Function, Duplicated Code	3	0.002299	0.002299	0.002299	0.000000
Large Class Code	5	0.001586	0.000000	0.006670	0.002872
Shotgun Surgery	1	0.001526	0.001526	0.001526	0.000000
Data Clumps	1	0.001231	0.001231	0.001231	0.000000
Long Parameter List	5	0.000382	-0.000108	0.001113	0.000578
Long Function	40	-0.000084	-0.007617	0.007097	0.002703

As in the case of metrics, fixing antipatterns was not the primary concern of developers. Typically, they fixed Duplicated Code, Long Functions, Large Class Code or Long Parameter List. Most of these antipatterns could be also identified via metrics. In practice, the greatest influence on the maintainability among antipatterns was caused by fixing Duplicated Code segments. Removing code clones can be done for example by using *Extract Method*, *Extract Class* or *Pull-up Method* refactoring techniques. Removing duplications reduces the LOC of the system, increases reusability and improves the overall effectiveness. Interestingly, fixing Duplicated Code sometimes reduced maintainability, as can be seen in the *Min* column of Table 7. For instance, in one case, it decreased the maintainability by 0.0078. Developers of Company IV performed an *Extract Superclass* refactoring on two of their classes to remove clones. At first it was not clear why it had a detrimental effect on the maintainability because in most of the other cases it had a beneficial effect. Further investigation showed that they fixed the Duplicated Code, which in fact increased the maintainability as usual, but they introduced two new OverrideBothEqualsAndHashcode coding issues, which together had a bigger detrimental effect than the fix itself. (Fortunately, they fixed the new coding issues in later commits.)

Developers fixed Duplicated Code antipatterns 11 times, Long Function with Duplicated Code 3 times altogether, Large Class Code 5 times, and Long Function antipattern 40 times. Fixing these antipatterns require a larger, global refactoring of the code (e.g. using Extract Method refactoring). These global refactorings induced a larger change in maintainability compared to others. It is also interesting that the deviation of the effects on maintainability were the largest in the case of fixing Duplicated Code, Large Class Code and Long Function antipatterns.

3.3.3. Coding Issues

Tables 8 and 9 list the average of measured maintainability changes where developers fixed coding issues. The relatively big number of refactorings tells us that this was what developers really wanted to fix when they refactored their code base. As we previously noted, it is ambiguous whether a code transformation which was intended to improve the maintainability, but slightly modifies the behavior, should be classified as a refactoring or not. Fixing a coding issue, for instance, 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) should be considered a change in the observed external functionality of the program. However, it is obvious that the purpose of fixing coding issues is to improve the maintainability of the code and not to modify its functionality. We will classify all these fixes as refactorings following the refactoring definition of Kim et al.[8], in which they say that refactoring does not necessarily preserve the semantics in all aspects. Nevertheless, we group the coding issues into two groups; namely (1) issues that can be fixed via semantic preserving transformations, and (2) issues which can be fixed only via transformations which do not preserve the semantics of the original code. The SP columns in Tables 8 and 9 show this information.

Table 8: Positive maintainability changes caused by commits fixing coding issues. (*SP* column shows whether a refactoring did a semantic preserving transformation or not.)

SP	Coding issue	#	Avg	Min	Max	Dev
✗	UseLocaleWithCaseConversions	4	0.008748	0.005894	0.012439	0.002938
✗	UnsynchronizedStaticDateFormatter	1	0.008618	0.008618	0.008618	0.000000
✓	AvoidInstanceofChecksInCatchClause	5	0.003825	0.000000	0.017286	0.007549
✓	ExceptionAsFlowControl	1	0.003139	0.003139	0.003139	0.000000
✗	NonThreadSafeSingleton	1	0.002977	0.002977	0.002977	0.000000
✓	AvoidCatchingNPE	3	0.002341	0.001627	0.003484	0.001000
✗	EmptyCatchBlock	11	0.002175	0.000000	0.007559	0.002849
✗	OverrideBothEqualsAndHashcode	8	0.001768	0.000000	0.005922	0.004241
✓	EmptyIfStmt	1	0.001286	0.001286	0.001286	0.000000
✓	UnusedPrivateField	9	0.000729	-0.004062	0.007533	0.003016
✓	PreserveStackTrace	11	0.000457	-0.000389	0.001942	0.000904
✗	SignatureDeclareThrowsException	23	0.000348	0.000000	0.001526	0.000692
✗	SwitchStmtsShouldHaveDefault	4	0.000323	-0.000167	0.000642	0.000364
✓	UseStringBufferForStringAppends	17	0.000289	-0.009357	0.012077	0.007609
✓	ArrayIsStoredDirectly	2	0.000273	0.000183	0.000363	0.000127
✓	UnusedLocalVariable	4	0.000223	-0.000247	0.000828	0.000463
✓	LooseCoupling	16	0.000212	0.000000	0.002647	0.000830
✓	AvoidDuplicateLiterals	454	0.000121	0.000121	0.000121	0.000000
✓	UnnecessaryLocalBeforeReturn	43	0.000108	0.000000	0.000585	0.000459
✓	UnnecessaryWrapperObjectCreation	118	0.000083	0.000083	0.000083	0.000000
✗	AvoidPrintStackTrace	32	0.000069	0.000000	0.000185	0.000304
✓	SimplifyConditional	39	0.000010	0.000000	0.000125	0.000061

Tables 8 and 9 show 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 manual refactoring period of any sys-

Table 9: Zero or negative changes in maintainability by commits fixing coding issues. (*SP* column shows whether a refactoring did a semantic preserving transformation or not.)

SP	Coding issue	#	Avg	Min	Max	Dev
✓	AvoidSynchronizedAtMethodLevel	8	0.000000	0.000000	0.000000	0.000000
✓	ConsecutiveLiteralAppends	1	0.000000	0.000000	0.000000	0.000000
✓	MethodReturnsInternalArray	8	0.000000	0.000000	0.000000	0.000000
✓	ReplaceHashtableWithMap	1	0.000000	0.000000	0.000000	0.000000
✓	UseIndexOfChar	48	0.000000	0.000000	0.000000	0.000000
✓	UnusedModifier	31	0.000000	0.000000	0.000000	0.000000
✓	BooleanInstantiation	47	-0.000016	-0.000273	0.000235	0.000305
✓	IntegerInstantiation	84	-0.000019	-0.000247	0.000014	0.000247
✓	IfElseStmtsMustUseBraces	117	-0.000111	-0.000456	0.000186	0.001406
✓	BigIntegerInstantiation	21	-0.000156	-0.003587	0.000974	0.001110
✓	InefficientStringBuffering	12	-0.000264	-0.002649	0.000128	0.000846
✓	UnusedPrivateMethod	2	-0.000863	-0.002729	0.001002	0.002638
✗	AvoidCatchingThrowable	2	-0.001654	-0.003307	0.000000	0.002339
✓	AddEmptyString	9	-0.001833	-0.004527	0.000677	0.002117

tem. Some of these coding issues are simple coding style guidelines which can be relatively easily fixed (e.g. IfElseStmtsMustUseBraces), while there are some issues which may indicate serious bugs and need to be carefully fixed (e.g. MethodReturnsInternalArray or OverrideBothEqualsAndHashCode). Issues that are easier to fix were refactored in larger quantities such as IntegerInstantiation and BooleanInstantiation. It is not that surprising that these issues had a relatively low impact on maintainability; however, it is interesting to observe that some of them induced a detrimental change in the maintainability.

The coding issue with the highest average maintainability improvement was UseLocaleWithCaseConversions. This issue warns the developer to use a Locale instead of simple String.toLowerCase()/toUpperCase() calls. This avoids common problems encountered with some locales, e.g. Turkish. The second highest average is the UnsynchronizedStaticDateFormatter issue, where the problem is that the code contains a static SimpleDateFormat field which is not synchronized. SimpleDateFormat is not thread-safe and Oracle recommends separate format instances for each thread. Company IV fixed this issue by creating a new SimpleDateFormat instance to guarantee thread-safety. However, using ThreadLocal would have been a better solution for both readability and performance.

In the case of the IfElseStmtsMustUseBraces issues, the reason for the detrimental change in maintainability is the increased number of the code lines in the modified methods. The sensors of the maintainability model will change at a low level; that is, the number of issues and the LOC metric. These changes will affect the higher level, aggregated maintainability attributes like CodeFaultProneness and Comprehensibility and also the Maintainability. A simple demonstration of this situation 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. What is more, this kind of issue has only minor priority so there is a good chance that the beneficial change in the number of issues will have a smaller influence on the maintainability than the detrimental change caused by the increased amount of

lines of code.

```
1  public static int doQuant(int n) {
2    if ( n >= 0 && n < 86) return 0;
3    else if (n > 85 && n < 170) return 128;
4    else return 255;
5  }
```

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

```
1  public static int doQuant(int n) {
2    if ( n >= 0 && n < 86)
3    {
4      return 0;
5    }
6    else if (n > 85 && n < 170)
7    {
8      return 128;
9    }
10   else
11   {
12     return 255;
13   }
14 }
```

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

In the case of InefficientStringBuffering, the reason for the detrimental change in maintainability is also the modified number of lines of code. Listing 5 demonstrates this kind of issue in a code sample that needs to be refactored. Some of the developers decided to fix this issue, as 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; namely, one coding issue vanishes.

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

Listing 5: A code with InefficientStringBuffering issue

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

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

Other developers preferred to fix the problem, as can be seen in Listing 7. This way, the issue vanishes as well, but there is a side effect: at least one new code line appears in the code, which again affects the lines of code metric, hence it has a slight impact on the maintainability.

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

Listing 7: Another way of refactoring Listing 5

Another interesting refactoring was the one where Company III refactored Avoid Duplicate Literals coding issues. This kind of issue tells us that a code fragment containing duplicate String literals can usually be improved by declaring the String as a constant field. Refactoring these flaws helps to eliminate dangerous duplicated strings, which should improve stability and readability. Although this was the manual phase of the project (where the companies could not yet use the refactoring tool that we intended to develop later), we spotted an interesting commit message where Company III refactored this coding issue with the help of the Netbeans IDE. Netbeans was able to assist them in finding and extracting duplicated string literals into constant fields. The fix was simple and straightforward so we decided keep these refactorings as valuable commits, and not to filter out them from the study. The developers eliminated 454 issues in one commit which covered more than 20,000 lines of code. The quality increase of this commit is quite large; and it improved the maintainability index of the whole system by 0.055.

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

3.4. Impact of Non-Refactoring Commits

We could analyze the systems only in their refactoring period when developers performed some refactoring tasks on their code. As a result, we have the analysis data for each system before and after a refactoring commit was submitted to the version control system. We did not analyze other commits, so we do not have analysis data for other non-refactoring commits. However, we have some possibilities here to study the impact of non-refactoring commits and compare them to refactorings. We analyzed the revisions before refactoring commits to explore the maintainability of a system between two refactoring commits. Suppose that r_i and r_j revisions are consecutive refactoring commits of a system and $j > i$. In this case, we analyzed the revisions r_{i-1}, r_i, r_{j-1} and r_j, following the same sequence of the commits. Change in the maintainability between the revisions (r_{i-1}, r_i) and (r_{j-1},r_j) are caused by two different refactoring commits, but the change in the maintainability between (r_i,r_{j-1})

is because of several non-refactoring commits. These changes measured between two consecutive refactoring commits, (which are caused by other, non-refactoring commits) makes it possible to compare the impact of refactoring and non-refactoring commits. Although these group together several smaller (non-refactoring) commits, we can consider them as normal development tasks. For simplicity, we will refer to these as *development commits* in the rest of this section.

It seems to be a reasonable assumption that refactoring commits often have a positive effect on maintainability, while this would not be true for development commits. To investigate this assumption, we count the commits which increased/decreased or had zero effect on maintainability. We use this data to study their independence with a *Pearson's chi-square test*. The input data is presented in Table 10.

Table 10: Number of refactoring and development commits which had negative/zero/positive effect on maintainability

Commit type	**Negative**	**Zero**	**Positive**
Refactoring	44	163	108
Development	63	167	139

We define the following null hypothesis: "*for each commit, its effect on maintainability is independent of the type of the commit (refactoring or development)*". Then the alternative hypothesis is: "*for each commit, its effect on maintainability is dependent on the type of the commit (refactoring or development)*". As a result of a chi-square test, we get a p-value of 0.2156, which is greater than the 0.05 significance level. Hence, we accept the null-hypothesis that the type of the commit and its effect on maintainability are independent.

Figures 7, 8, 9 show how the maintainability of the systems changed over time during the refactoring period. Revision numbers are obfuscated, but their order follows the original order of the commits. In the case of System A, developers refactored four submodules of the system, and we show these submodules separately in the diagram. The diagrams acknowledge that refactoring and non-refactoring results in varying changes in the maintainability. Just like we can spot refactoring/development commits which suddenly improve the measured values, we can spot their counterparts as well which suddenly decrease these values. It is easy to see, however, that the maintainability of the systems followed an improving tendency in the refactoring period.

We also have to notice that most of the refactorings presented in our study may be considered as small local changes and these commits are likely to have a small impact on the global maintainability. So the question arises whether the improving tendency is because developers take quality more into account in their new code, or the ColumbusQM model is more sensitive to larger code changes.

Besides maintainability, we measured the lines of code metric of the systems. Simply from the lines of code we can see the final difference in the newly added

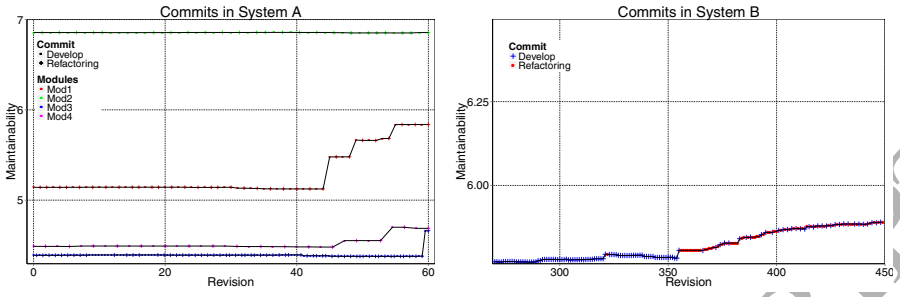

Figure 7: Maintainability of systems A and B during the refactoring period (revision numbers are obfuscated, but they are in their original order)

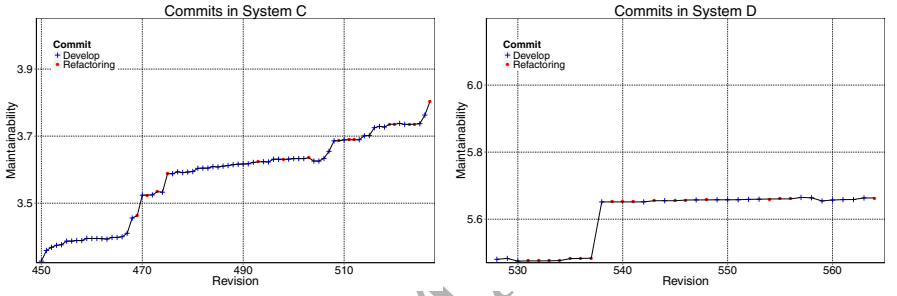

Figure 8: Maintainability of systems C and D during the refactoring period (revision numbers are obfuscated, but they are in their original order)

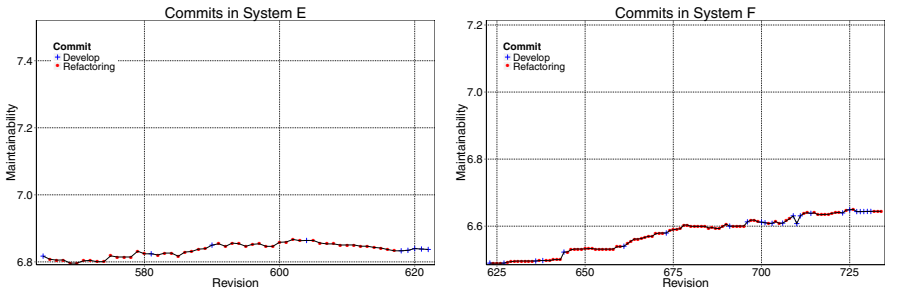

Figure 9: Maintainability of systems E and F during the refactoring period (revision numbers are obfuscated, but they are in their original order)

or deleted lines, but we cannot see the exact number of modified lines. Nevertheless, the difference in added/deleted lines is a good estimation of the size of the commit. Tables 11 and 12 show the average change in maintainability of refactoring and development commits normalized by the change in lines of code for each system. Recall that 'development commits' group together more commits. Hence, these are likely to be larger structural changes. Table 13

Table 11: Average change in maintainability of refactoring commits normalized by the change in lines of code for each system

System	Maint. Change Avg.	Change in LOC Avg.	Maint. Change Avg. per LOC
System A	-0.000087	30.55	0.000006
System B	0.000589	24.38	0.000099
System C	0.008362	64.73	0.000163
System D	0.000837	2.56	0.000266
System E	0.000092	6.12	-0.000504
System F	0.001441	6.28	0.000677

Table 12: Average change in maintainability of development commits normalized by the change in lines of code for each system

System	Maint. Change Avg.	Change in LOC Avg.	Maint. Change Avg. per LOC
System A	0.009068	-117.88	-0.000001
System B	0.000693	215.69	0.000018
System C	0.005652	19.31	0.000665
System D	0.009329	-48.00	0.000365
System E	0.001922	-5.75	0.000808
System F	0.001203	12.04	0.000088

shows the Pearson's r correlation coefficients and p significance levels between the change in lines of code and the change in maintainability for all the commits of each system. These results indicate a strong correlation between the size of the commit and its effect on maintainability. Hence, we have to acknowledge that the ColumbusQM model is more sensitive to larger code changes. Still, the smaller changes of the refactoring commits had a measurable impact on the global maintainability as well. Notice also, that for some systems the correlations are negative (also when we consider them all together). Moreover, in the case of System D, they indicate a perfect negative linear relationship between variables. The change in the lines of code can be negative (when they delete lines). Hence, this means that sometimes when they remove more lines, they improve the maintainability more notably. Indeed, in the case of System D, they had five commits (out of 36) where in the 'largest' commit they removed 906 lines reaching their best maintainability improvement of 0.1679 (see the online appendix for details).

On the other hand, when Bakota et al. evaluated the ColumbusQM model [13] on industrial software systems, they found that "the changes in the results of the model reflect the development activities, i.e. during development the quality decreases, during maintenance the quality increases." Here, we studied a refactoring period and in contrast to Bakotat et al. we found that normal development commits rather improved the quality. This acknowledges that developers tended to take quality more into account in their new code. What they also admitted us later at the end of the project.

Table 13: Pearson's r correlation coefficient and p significance levels between the change in lines of code and the change in maintainability

System	r	p
System A	-0.5	<0.01
System B	0.48	<0.01
System C	0.17	0.127
System D	-0.99	<0.01
System E	0.18	0.171
System F	-0.07	0.432
All	-0.42	<0.01

3.5. Discussion of Motivating Research Questions

Is it possible to recognize the change in maintainability caused by a single refactoring operation with a probabilistic quality model based on code metrics, coding issues and code clones?

We applied the ColumbusQM maintainability model to measure changes in the maintainability of large-scale industrial systems before/after refactoring commits. Our measurements revealed that the maintainability changes induced by refactoring operations can be seen in most of the cases. One particular change usually caused only a small change, which is quite natural considering that we analyzed 2.5 million lines of code altogether, and a particular refactoring operation usually affects only a small part of it. However, with some refactorings (mostly those involving fixing local coding issues) the model did not display any changes in the maintainability. This was due to the fact that these refactorings were very local, resulting that the sensors of the model did not recognize any changes in the metric values. By fine tuning the maintainability model, these cases might become detectable.

Does refactoring increase the overall maintainability of a software system?

After the refactoring period, the overall maintainability of the software systems improved and the maintainability model was able to measure this improvement in five out of the six systems. Commits which fixed more coding issues had a relatively higher impact on maintainability. Similarly, we observed in the tables that when developers fixed more metrics or antipatterns together, they induced a bigger change compared to others. Hence, a larger refactoring has a noticeable, positive impact on the maintainability, which is measurable using static analysis techniques.

Can it happen that refactoring decreases maintainability?

Measurements reveal that some refactoring operations might have a negative impact on the maintainability of the system, although its main 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 maintainability attribute, but we debase others.

3.6. Additional Observations

Overall, based on our results and analyses, there are some additional interesting observations that deserve to be discussed further.

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, like the small code smells, which they could fix quickly. 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 a thorough understanding of the code so developers can readily see the problem and fix it even if it was not written by themselves. In addition, testing is easier in these cases too. On the other side, a larger refactoring may have more difficulties: it requires better knowledge and understanding of the code; it must be designed and applied more carefully; or it may happen that permission is needed to change things across components/architecture. It remains a future research question as to which choice is better in the long term in such a situation. Should we fix as many small issues as we can, or perform only a few, but large, global refactorings and restructure the code?

Developers did not refactor just to improve metrics or avoid antipatterns

Our results suggest that developers did not really want to improve the metric values or 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 may not really be aware of the meaning of metrics and antipatterns. Though we are certain that they were aware of the definition of some metrics and code smells (because we trained them for the project), they probably had no experience in recognizing and fixing problematic classes with bad cohesion or coupling values, for instance. They were not maintainability experts who were experienced in studying reports of static analyzers. This seems to tie in with the previous finding that developers went for the low-hanging fruits, and chose the easier way of improving maintainability.

Fixing more complex design flaws (e.g. antipatterns or more complex coding issues) might have a better impact on the maintainability

In Figure 10, we show the effect of the average impact of different refactoring types (metrics, antipatterns, coding issues) on the maintainability among all the refactoring commits, and we list the corresponding min/max/deviation values in Table 14. As we saw previously, developers fixed mostly coding issues, but notice that those coding issues which required a fix that modified the semantics of the code had a larger impact on maintainability, just like that for antipatterns or metrics. Taking into account how the ColumbusQM calculates maintainability, this is mainly because fixing a more complex issue (antipattern) has a bigger impact on the full code base and not just some local parts of it. Another observation here is that we see developers fixed the Duplicated Code antipattern the most often, which is No. 1 in Martin Fowler's dangerous bad-smell list [3].

Table 14: Average, minimum and maximum impact on maintainability of different refactoring types

Types	Average Change	Minimum Change	Maximum Change	Deviation of Change
Metrics	0.000995	-0.007803	0.017286	0.003854
Antipattern	0.000832	-0.007803	0.011233	0.003524
Coding Issues	0.001080	-0.003307	0.012439	0.003393
Coding Issues (SP)	0.000074	-0.009357	0.017286	0.004392

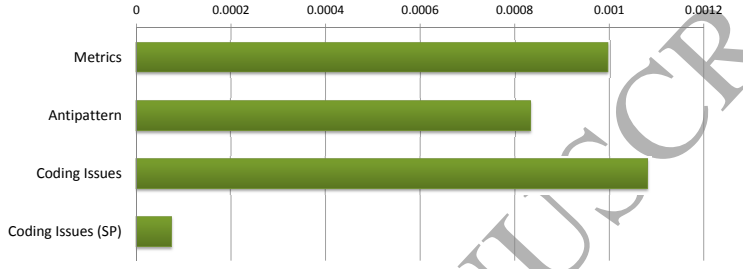

Figure 10: Average impact on maintainability of different refactoring types

Developers learned to write better code during the refactoring period

All the systems that we studied in the refactoring period displayed an improvement in source code maintainability, even if we only take into account the revisions where they did not refactor the code, but just committed normal development patches. Our analysis revealed us, that the number of newly introduced issues in the new code decreased. Indeed, developers admitted to us at the end of the project that they had learned a lot from performing a static analysis and from refactoring coding issues. They had learned how to avoid different types of potential coding issues. As a result they paid more attention to writing better code and avoiding new issues.

4. Threats to Validity

We made our observations based on hundreds of refactoring commits in six large-scale industrial systems. As 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 findings. Here, we give a brief overview of the most important ones.

Size of the sample set of refactoring commits investigated

The sample set was taken from a large-scale industrial environment compared to other studies, but, it is still limited to the systems that we analyzed. With a larger sample set of refactorings we might have an even better basis for conclusions and a more precise view on refactorings. In the future, we intend to extend the sample set with an analysis of automatic refactorings as well.

Maintainability analysis relies only on the Columbus Quality Model and Java

The maintainability model is an important part of the analysis as it also determines what we consider as an *effect on maintainability* of refactorings. Currently, we rely on the ColumbusQM model with all of its advantages and disadvantages. On the positive side this model has been published, validated and reflects the opinion of developers [13]; however, we saw in the evaluation section that the model might miss some aspects which would reflect some changes caused by refactorings. In particular, the model did not deal with some low priority local coding issues. The version of ColumbusQM used during the analysis relies on Java source code analysis. However, the same sensors could be applied to other object-orientated languages too.

Refactoring suggestions and quality analysis tool used to evaluate their effect come from the same toolkit

The Columbus technology was used for both the refactoring suggestions and by the quality analysis tool. I.e, the toolkit thinks that the changes made according to its own suggestions improve quality. This leads to the threat that the quality model used the same quality indicators as it suggested earlier as refactoring opportunities.

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 companies minimized the efforts that they spent on refactoring. Also, the actual state of a system, such as the size and quality of its test suite may influence the risk that a company would like to take during refactoring.

Limitations of the static analysis

We gave support to the developers in identifying coding issues with the help of a static analyzer. Naturally, this was a great help for them in identifying problematic code fragments, but might have led the developers to just concentrate 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.

5. Related Work

Since Opdyke introduced the term *refactoring* in his PhD dissertation [2] and Fowler published a catalog of refactoring 'bad smells' [3], many researchers have studied this technique to improve the maintainability of software systems. Just a few years later, Wake [16] published a workbook on the identification of 'smells', and indicated practices to recognize the most important ones and some possible ways to fix them by applying the appropriate refactoring techniques. Five years after the appearance of Fowler's book, Mens et al. [17] published a survey with over 100 related papers in the area of software refactoring.

There are several interesting topics studied today by researchers in which they examine refactoring techniques, such as program comprehension [18], impact of refactoring on regression testing [19], developers' opinion on refactoring tools [20], etc. Among these papers, there are some which investigate the positive or negative effects of refactorings on maintainability and software quality, but there are only a few empirical studies, especially studies that were performed on large-scale industrial systems. Below, we will present an overview of research work related to our study.

5.1. Guidelines on how to apply refactoring methods

One reason why researchers study the relations between maintainability and refactoring is to guide developers on when and how to apply refactorings.

Sahraoui et al. [5] investigated the use of object-oriented metrics to detect potential design flaws and to suggest transformations that handle the identified problems. They relied on a quality estimation model to predict how these transformations improve the quality. By validating their technique on some classes of a C++ project, they showed that their approach could assist a designer/programmer by suggesting transformations.

A visualization approach was proposed by Simon et al. [21]. Their technique was based on source code metrics of classes and methods to help developers in identifying candidates for refactoring. They showed that metrics can support the identification of 'bad smells' and thus can be used as an effective and efficient way to support the decision of where to apply refactoring.

Tahvildari et al. [22, 23] investigated the use of object-oriented metrics to detect potential design flaws and suggested transformations for correcting them. They analyzed the impact of each refactoring on object-oriented metrics (complexity, cohesion and coupling).

Yu et al. [24] adopted a process-oriented modeling framework in order to analyze software qualities and to determine which software refactoring transformations are most appropriate. In a case study of a simple Fortran program, they showed that their approach was able to guide the refactoring towards high performance and code simplicity while keeping implementing more functionalities.

Meananeatra [25] proposed the use of filtering conditions to help developers in refactoring identification and program element identification. They also proposed an approach to choose an optimal sequence of refactorings.

5.2. Refactoring and its effect on software defects

One way how researchers attempt to assess the effects of refactorings on maintainability is to study its effects on software defects.

Ratzinger et al. [10] analyzed refactoring commits in five open-source systems written in Java and investigated via bug prediction models the relation between refactoring and software defects. They found an inverse correlation between refactorings and defects: if the number of refactoring edits increases in the preceding time period, the number of defects decreases.

Görg and Weißgerber [26, 27] detected incomplete refactorings in open-source projects and they found that incorrect refactoring edits can possibly cause bugs.

Later, Weißgerber et al. [28, 11] analyzed version histories of open-source systems and investigated whether refactorings are less error-prone than other changes. They found that in some phases of their projects a high ratio of refactorings was followed by a higher ratio of bugs. They found also phases where there was no increase at all.

5.3. Refactoring and its effect on code metrics

Some researchers assess the effects of refactorings on source code metrics.

Stroulia and Kapoor [6] presented their experiences with a system that followed a so-called refactoring-based development. They found that size and coupling metrics of their system decreased after the refactoring process.

Du Bois and Mens [29] studied the effects of selected refactorings (Extract-Method, EncapsulateField and PullUpMethod) on internal quality metrics such as the Number of Methods, Cyclomatic Complexity, Coupling Between Objects and Lack of Cohesion. Their approach is based on a formalism to describe the impact of refactorings on an AST representation of the source code, extended with cross-references. Later, Du Bois et al. [7] proposed refactoring guidelines for enhancing cohesion and coupling metrics and they got promising results by applying these transformations to an open-source project. The Ph.D. thesis of Du Bois was also about the effects of refactoring on internal and external program quality attributes [30].

5.4. Empirical studies about refactoring and its effects on software quality/-maintainability

Empirical studies are those which are the closest to study. However, there are only a few large-scale *empirical studies* here.

Kataoka et al. [31] published a quantitative evaluation method to measure the maintainability enhancement effect of refactorings. They analyzed a single project and compared the coupling before and after the refactoring in order to evaluate the degree of maintainability enhancement. They found coupling metrics were effective for quantifying the refactoring effect and for choosing suitable refactorings. Their validation relied on a five-year-old C++ project of a single developer.

Moser et al. [32] studied the impact of refactoring on quality and productivity. They observed small teams working in similar, highly volatile domains and assessed the impact of refactoring in a 'close to industrial environment'. Their case study was about a Java project with 30 Java classes having 1,770 source code statements. Their findings indicated that refactoring not only increases software quality, but it also improves productivity.

Ratzinger et al. [33] observed the evolution of a 500 KLOC industrial Picture Archiving and Communication System (PACS) written in Java before and after a change coupling-driven refactoring period. They found that after the refactoring period, the code had low change coupling characteristics.

Demeyer [34] pointed out that refactoring is often blamed for performance reduction, especially in a C++ context, where the introduction of virtual function calls introduces an extra indirection via the virtual function table. He discovered, however, that C++ programs refactored this way often perform faster than their non-refactored counterparts (e.g. compilers can optimize better on polymorphism than on simple if-else statements).

Stroggylos et al. [35] assessed a similar question to ours, namely whether refactoring improves software quality or not. They analyzed version control system logs (46 revision pairs) of open-source projects (Apache, Log4j, MySQL connector and Hibernate core) to detect changes marked as 'refactoring' and how software metrics were affected. They found that "*the expected and actual results often differ*", and although "*people use refactoring in order to improve the quality of their systems, the metrics indicate that this process often has the opposite results.*"

Alshayeb et al. [36] studied the effects of refactorings on different external quality attributes, namely adaptability, maintainability, understandability, reusability, and testability. They analyzed a system developed by students and two open-source systems with at most 60 classes and less than 12,000 lines of code. They investigated how C&K metrics had changed after applying refactoring techniques taken from Fowler's catalog and estimated their effects on the external quality attributes. They found that refactoring did not necessarily improve these quality attributes.

Geppert et al. [37] studied the refactoring of a large legacy business communication product where protocol logic in the registration domain was restructured. They investigated the strategies and effects of the refactoring effort on aspects of changeability and measured the outcomes. The findings of their case study revealed a significant decrease in customer reported defects and in efforts needed to make changes.

Wilking et al. [38] investigated the effect of refactoring on maintainability and modifiability through an empirical evaluation carried out with 12 students. They tested maintainability by randomly inserting defects into the code and measuring the time needed to fix them; and they tested modifiability by adding new requirements and measuring the time and LOC metric needed to implement them. Their maintainability test displayed a slight advantage for refactoring, but regarding modifiability, the overhead of applying refactoring appeared to undermine other, positive effects.

Negara et al. [39] presented an empirical study that considered both manual and automated refactorings. They claimed that they analyzed 5,371 refactorings applied by students and professional programmers, but they did not provide further information on the systems in question.

A large-scale study, with similar findings, was carried out by Murphy-Hill et al. [40]. They applied refactorings taken from Fowler's catalog, and their data sets spanned over 13,000 developers with 240,000 tool-assisted refactorings of open-source applications. Our study is complementary, as we analyzed industrial systems instead of open-source ones and we mostly dealt with coding issues instead of refactorings from the catalog.

Kolb et al. [41] reported on the refactoring of a software component called Image Memory Handler (IMH), which was used in Ricoh's current products of office appliances. The component was implemented in C and it had about 200 KLOC. They evaluated software metrics of the product before and after a refactoring phase and found that the documentation and implementation of the component had been significantly improved.

Kim et al. [42] reported on an empirical investigation of API-level refactorings. They studied API-level refactorings and bug fixes in three large open-source projects, totaling 26,523 revisions of evolution. They found an increase in the number of bug fixes after API-level refactorings, but the time taken to fix bugs was shorter after refactorings than before. In addition, they noticed that a large number of refactoring revisions included bug fixes at the same time or were related to later bug fix revisions. They also noticed frequent 'floss refactoring' mistakes (refactorings interleaved with behavior modifying edits).

In their study, Kim et al. [8] presented a study of refactoring challenges at Microsoft through a survey, interviews with professional software engineers and a quantitative analysis of version history data (of Windows 7). Among several interesting findings, their survey showed that refactoring definition in practice seemed to differ from a rigorous academic definition of behavior-preserving program transformations and that developers perceived that refactoring involved substantial cost and risks.

5.5. Code smells and maintenance

Another topic close to ours is the effect of (fixing) code smells on maintenance problems.

Yamashita and Moonen [43] found that the effect of code smells on the overall maintainability is relatively small. They observed 6 developers working on 4 Java systems and only about 30% of the problems that they faced were related to files containing code smells.

In another study, Yamashita and Counsell [44] found that code smells were not good indicators for comparing the maintainability of systems differing greatly in size. They evaluated four medium-sized Java systems using code smells and compared the results against previous evaluations on the same systems based on expert judgment and C&K metrics.

In a recent study, Yamashita [45] assessed the capability of code smells to explain maintenance problems on a Java system which was examined for the presence of twelve code smells. They found a strong connection between the Interface Segregation Principle and maintenance problems.

Similarly, Hall et al. [46] found that some smells do indeed indicate fault-prone code in some circumstances, but that the effects that these smells had on faults were small. As they said, "*arbitrary refactoring is unlikely to significantly reduce fault-proneness and in some cases may increase fault-proneness*".

Ouni et al. [47] claimed that most of the existing refactoring approaches treated the code-smells to be fixed with the same importance; and they proposed a prioritization of code-smell correction tasks. Another prioritization approach

was proposed by Guimaraes et al. [48] based on software metrics and architecture blueprints.

Khomh et al. [49] investigated the impact of antipatterns on classes in object-oriented systems and found that classes participating in antipatterns were more change and fault-prone than others.

Abbes et al. [50] investigated the effect of Blob and Spaghetti Code antipatterns on comprehension in 24 subjects and on three different systems developed in Java. Their results showed that the occurrence of one antipattern did not significantly make its comprehension harder, hence they recommend to avoid a combination of antipatterns via refactoring.

D'Ambros et al. [51] studied the relationship between software defects and a number of design flaws. They also found that, while some design flaws were more frequent, none of them could be considered more harmful in terms of software defects.

Chatzigeorgiou et al. [52] studied the evolution of code smells in JFlex and JFreeChart. They noticed that only few code smells were removed from the projects and in most cases their disappearance was not the result of targeted refactoring activities, but rather a side-effect of adaptive maintenance.

Tsantalis et al. [53] examined refactorings in JUnit, HTTPCore, and HTTPClient. Among several interesting findings, they found that there was very little variation in the types of refactorings applied on test code, since most of the refactorings were about reorganization and the renaming of classes.

5.5.1. Summary

In contrast to the above-mentioned studies, in ours (1) we observed a *large amount of manual refactorings* (1,273 refactoring operations in 315 commits, counting also a commit with 454 operations); (2) we studied the effect of refactorings on maintainability in real-life, large-scale industrial systems with over 2.5 million total lines of code; (3) these commits fixed *different design flaws* including code smells, antipatterns and coding issues; (4) lastly, we applied a *probabilistic quality model* (ColumbusQM) which integrates different properties of the system like metrics, clones, coding issues. Our study was carried out in a large-scale *in vivo* (industrial) environment.

A detailed feature comparison matrix of the above research papers is shown in Table 15. The rows of the table represent the studies and the columns indicate different features and research topics. The columns open-source, industrial, and students show the scope of the study in question. The next three columns refer to the code size of the studied projects, while the following two columns compare the number of metrics the authors involved in their work. The remaining columns show the research topics of the articles.

Figure 11 shows a graphical representation of the articles in our comparison. The blue diamond shapes represent the selected topics. An arrow between an article (indicated by gray ellipses or rectangles) and a topic (blue diamond) means that the article covers that topic. Ellipses without borders show works where the research was conducted with students or where this information was

Table 15: Comparison matrix of related studies based on public features and research topics

article	open-source	industrial	students	kLOC < 150	150 <= kLOC < 500	500 <= kLOC	metrics < 7	7 < metrics	maintainability	refactoring	bugs, defects	smells	refact. recommend.
Yamashita2013a [44]	-	✓	-	-	✓	-	-	✓	✓	-	-	✓	-
Yamashita2013 [43]	-	✓	-	-	✓	-	-	-	✓	-	-	✓	-
Yamashita2014 [45]	-	-	-	-	-	-	-	-	✓	-	-	✓	-
Hall2014 [46]	-	-	-	-	-	-	-	-	-	-	✓	✓	-
Ouni2015 [47]	✓	-	-	-	✓	-	-	-	-	✓	-	✓	-
Guimaraes2013 [48]	-	-	-	-	-	-	-	-	-	-	-	✓	-
Khomh2012 [49]	✓	-	-	-	✓	-	-	-	-	-	-	✓	-
Abbes2011 [50]	✓	-	-	-	✓	-	-	-	-	-	-	✓	✓
DAmbros2010 [51]	✓	-	-	-	✓	✓	-	-	-	-	✓	✓	-
Chatzigeorgiou2014 [52]	✓	-	-	✓	-	-	-	-	-	-	-	✓	-
Tsantalis2013 [53]	-	-	-	✓	-	-	-	-	-	✓	-	-	-
Opdyke1992 [2]	-	-	-	-	-	-	-	-	-	✓	-	-	-
Fowler1999 [3]	-	-	-	-	-	-	-	-	-	✓	-	✓	-
Wake2003 [16]	-	-	-	-	-	-	-	-	-	✓	-	✓	-
Mens2004 [17]	-	-	-	-	-	-	-	-	-	✓	-	-	-
DuBois2005 [18]	-	-	-	-	-	-	-	-	-	✓	-	-	-
Rachatasumrit2012 [19]	-	-	-	-	-	-	-	-	-	✓	-	-	-
Pinto2013 [20]	-	-	-	-	-	-	-	-	-	✓	-	-	✓
Sahraoui2000 [5]	-	-	-	✓	-	-	-	✓	✓	✓	✓	-	✓
Simon2001 [21]	-	-	-	✓	-	-	✓	-	-	✓	-	✓	✓
Tahvildari2003 [22]	-	-	-	-	-	-	✓	-	-	-	✓	-	✓
Yu2003 [24]	-	-	-	✓	-	-	-	-	-	✓	-	-	✓
Meananeatra2012 [25]	-	-	-	-	-	-	-	-	-	✓	-	-	✓
Ratzinger2008 [10]	✓	-	-	-	✓	-	-	-	-	✓	✓	-	-
Goerg2005 [26]	✓	-	-	✓	-	-	-	-	-	✓	✓	-	-
Weissgerber2006 [11]	✓	-	-	✓	-	-	-	-	-	✓	✓	-	-
Stroulia2001 [6]	-	-	-	✓	-	-	✓	✓	-	✓	-	-	-
DuBois2003 [29]	-	-	-	-	-	-	-	✓	-	✓	-	-	-
DuBois2004 [7]	✓	-	-	-	-	-	-	✓	-	✓	-	-	-
DuBois2006 [30]	-	-	-	-	-	-	-	✓	-	✓	-	-	-
Kataoka2002 [31]	-	-	✓	✓	-	-	✓	-	✓	✓	-	-	-
Moser2008 [32]	-	-	✓	✓	-	-	-	-	✓	✓	-	-	-
Ratzinger2005 [33]	-	✓	-	-	-	✓	✓	-	-	✓	-	-	-
Demeyer2005 [34]	-	-	-	-	-	-	✓	-	-	✓	-	-	-
Stroggylos2007 [35]	✓	-	-	-	✓	-	-	✓	✓	✓	-	-	-
Alshayeb2009 [36]	✓	-	✓	✓	-	-	-	✓	✓	✓	-	-	-
Geppert2005 [37]	-	✓	-	-	-	-	-	-	-	✓	✓	-	-
Wilking2007 [38]	-	-	✓	✓	-	-	✓	-	✓	✓	-	-	-
Negara2013 [39]	-	-	✓	-	-	-	-	-	-	✓	-	-	-
Murphy-Hill2009 [40]	✓	-	-	-	-	✓	-	-	-	✓	-	-	-
Kolb2006 [41]	-	✓	-	-	✓	-	-	✓	-	✓	-	-	-
Kim2011 [42]	✓	-	-	-	-	✓	-	-	-	✓	✓	-	-
Kim2012 [8]	-	✓	-	-	-	✓	-	-	-	✓	-	-	-
Our study	-	✓	-	-	-	✓	-	✓	✓	✓	-	✓	-

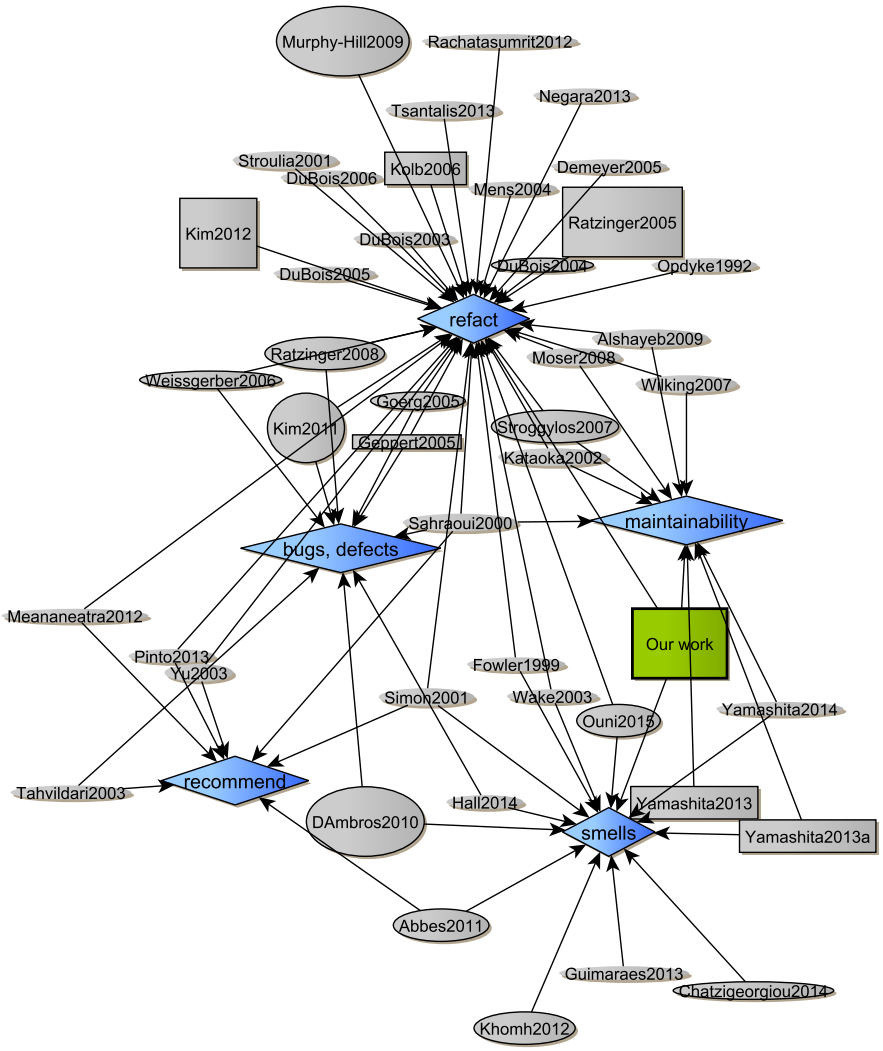

Figure 11: Graphical view of articles based on extracted features and relations to topics.

missing from the papers. Bordered ellipses show articles that investigate open-source projects. A bordered rectangle shows that the research was based on industrial projects. The height of the nodes represents the code size of the investigated project. Small-scale projects have smaller shapes, and large-scale projects have larger.

The graph was created with the *Organic Layout* tool of the *yEd Graph Editor*. This layout created a nice landscape where articles take place depending on their research topic. As it can be seen, our work (indicated with green color)

fills a big hole among smells, maintainability and refactoring, and does this in a large-scale industrial setting which is rather unique among the related works.

6. Conclusions and Future Work

Bakota et al. claim that the maintainability of a software product erodes over the years and if developers do not periodically and intentionally refactor the source code, then its maintainability will not improve [54].

In this study, we investigated hundreds of refactoring commits from the refactoring period of six large-scale industrial systems developed by four companies, and we investigated the effects of these commits on source code maintainability using maintainability measurements based on the ColumbusQM maintainability model [13]. We obtained 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 fix code smells indicated by metrics or automatic smell detectors. It reinforces the conclusion of our previous study [9], where we found that when developers had the extra time and budget to refactor their code, they optimized their process so as to improve the maintainability of a system.

In summary, we claim that the outcome of one refactoring on the global maintainability of the software product is hard to predict; moreover, it might sometimes have a detrimental effect. However, a whole refactoring process can have a significant beneficial effect on the maintainability, which is measurable using a maintainability model. The reason for this is not only because the developers improve the maintainability of their software, but also because they will learn from the process and pay more attention to writing more maintainable new code.

The six systems in our study and their manual refactorings represented only a portion of the full research project. We gathered additional data from the developers and from the automatic tool guided refactoring period as well. This information came from an *in vivo* environment and we can learn a lot from it. In the future, we plan to further investigate and seek answers to more questions that arise when developers start working on refactoring. Such questions include: 'What should I refactor?', 'How should I do it?', 'Can I automate it somehow?', 'What should I take care of or be afraid of?' and 'How much time will it take?'

Online Appendix

The data set which serves as the basis for this paper is available as an online appendix at: `http://www.inf.u-szeged.hu/~ferenc/papers/JSS2014-Is-It-Worth-Refactoring/`

This data set contains for all projects the analysis results of the refactoring commits. For each commit, it includes the type of the fixed issue, and the change in the maintainability caused by the commit.

Acknowledgment

This research work was partially supported by the Hungarian national grant GOP-1.2.1-11-2011-0002 and the EU FP7 project REPARA (project number: 609666). 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, G. E. Tompkins, Characteristics of application software maintenance, Communications of the ACM 21 (6) (1978) 466–471.

[2] W. F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois (1992).

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

[4] J. Mylopoulos, L. Chung, B. Nixon, Representing and using nonfunctional requirements: A process-oriented approach, IEEE Transactions on Software Engineering 18 (6) (1992) 483–497.

[5] H. A. Sahraoui, R. Godin, 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, R. Kapoor, Metrics of refactoring-based development: An experience report, in: OOIS 2001, Springer, 2001, pp. 113–122.

[7] B. Du Bois, S. Demeyer, 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] M. Kim, T. Zimmermann, N. Nagappan, A field study of refactoring challenges and benefits, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, ACM, New York, NY, USA, 2012, pp. 50:1–50:11.

[9] G. Szőke, C. Nagy, R. Ferenc, 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.

[10] J. Ratzinger, T. Sigmund, H. C. Gall, On the relation of refactorings and software defect prediction, in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08, ACM, 2008, pp. 35–38.

[11] P. Weißgerber, S. Diehl, Are refactorings less error-prone than other changes?, in: Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR '06, ACM, 2006, pp. 112–118.

[12] G. Szőke, G. Antal, C. Nagy, R. Ferenc, T. Gyimóthy, Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?, in: Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on, IEEE Computer Society, 2014, pp. 95–104.

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

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

[15] R. Ferenc, Á. Beszédes, M. Tarkiainen, 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, 2002, pp. 172–181.

[16] W. C. Wake, Refactoring Workbook, 1st Edition, Addison-Wesley Longman Publishing Co., Inc., 2003.

[17] T. Mens, T. Tourwé, A survey of software refactoring, IEEE Transactions on Software Engineering 30 (2) (2004) 126–139.

[18] B. Du Bois, S. Demeyer, J. Verelst, Does the "refactor to understand" reverse engineering pattern improve program comprehension?, in: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering, CSMR '05, IEEE Computer Society, 2005, pp. 334–343.

[19] N. Rachatasumrit, M. Kim, An empirical investigation into the impact of refactoring on regression testing, in: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, 2012, pp. 357–366.

[20] G. H. Pinto, F. Kamei, What programmers say about refactoring tools?: An empirical investigation of stack overflow, in: Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools, WRT '13, ACM, 2013, pp. 33–36.

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

[22] L. Tahvildari, 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.

[23] L. Tahvildari, K. Kontogiannis, J. Mylopoulos, Quality-driven software re-engineering, Journal of Systems and Software 66 (3) (2003) 225–239.

[24] Y. Yu, J. Mylopoulos, E. Yu, J. C. Leite, L. Liu, 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.

[25] P. Meananeatra, Identifying refactoring sequences for improving software maintainability, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, ACM, 2012, pp. 406–409.

[26] C. Görg, P. Weißgerber, Error detection by refactoring reconstruction, in: Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR '05, ACM, 2005, pp. 1–5.

[27] C. Görg, P. Weißgerber, Error detection by refactoring reconstruction, SIGSOFT Softw. Eng. Notes 30 (4) (2005) 1–5.

[28] P. Weißgerber, S. Diehl, Identifying refactorings from source-code changes, in: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06, IEEE Computer Society, 2006, pp. 231–240.

[29] B. Du Bois, 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.

[30] B. Du Bois, A study of quality improvements by refactoring, Ph.D. thesis (2006).

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

[32] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, 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.

[33] J. Ratzinger, M. Fischer, H. Gall, Improving evolvability through refactoring, SIGSOFT Softw. Eng. Notes 30 (4) (2005) 1–5.

[34] 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.

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

[36] M. Alshayeb, Empirical investigation of refactoring effect on software quality, Inf. Softw. Technol. 51 (9) (2009) 1319–1326.

[37] B. Geppert, A. Mockus, F. Rossler, Refactoring for changeability: A way to go?, in: Proceedings of the 11th IEEE International Software Metrics Symposium, METRICS '05, IEEE Computer Society, 2005, pp. 13–.

[38] D. Wilking, U. F. Kahn, S. Kowalewski, An empirical evaluation of refactoring, e-Informatica Software Engineering Journal Vol. 1, nr 1 (2007) 27–42.

[39] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, D. Dig, A comparative study of manual and automated refactorings, in: Proceedings of the 27th European Conference on Object-Oriented Programming, ECOOP'13, Springer-Verlag, 2013, pp. 552–576.

[40] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, 2009, pp. 287–297.

[41] R. Kolb, D. Muthig, T. Patzke, K. Yamauchi, Refactoring a legacy component for reuse in a software product line: A case study: Practice articles, J. Softw. Maint. Evol. 18 (2) (2006) 109–132.

[42] M. Kim, D. Cai, S. Kim, An empirical investigation into the role of api-level refactorings during software evolution, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, 2011, pp. 151–160.

[43] A. Yamashita, L. Moonen, To what extent can maintenance problems be predicted by code smell detection? - an empirical study, Inf. Softw. Technol. 55 (12) (2013) 2223–2242.

[44] A. Yamashita, S. Counsell, Code smells as system-level indicators of maintainability: An empirical study, Journal of Systems and Software 86 (10) (2013) 2639 – 2653.

[45] A. Yamashita, Assessing the capability of code smells to explain maintenance problems: An empirical study combining quantitative and qualitative data, Empirical Softw. Engg. 19 (4) (2014) 1111–1143.

[46] T. Hall, M. Zhang, D. Bowes, Y. Sun, Some code smells have a significant but small effect on faults, ACM Trans. Softw. Eng. Methodol. 23 (4) (2014) 33:1–33:39.

[47] A. Ouni, M. Kessentini, S. Bechikh, H. Sahraoui, Prioritizing code-smells correction tasks using chemical reaction optimization, Software Quality Control 23 (2) (2015) 323–361.

[48] E. Guimaraes, A. Garcia, E. Figueiredo, Y. Cai, Prioritizing software anomalies with software metrics and architecture blueprints: A controlled experiment, in: Proceedings of the 5th International Workshop on Modeling in Software Engineering, MiSE '13, IEEE Press, 2013, pp. 82–88.

[49] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Softw. Engg. 17 (3) (2012) 243–275.

[50] M. Abbes, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, CSMR '11, IEEE Computer Society, 2011, pp. 181–190.

[51] M. D'Ambros, A. Bacchelli, M. Lanza, On the impact of design flaws on software defects, in: Proceedings of the 2010 10th International Conference on Quality Software, QSIC '10, IEEE Computer Society, 2010, pp. 23–31.

[52] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of code smells in object-oriented systems, Innov. Syst. Softw. Eng. 10 (1) (2014) 3–18.

[53] N. Tsantalis, V. Guana, E. Stroulia, A. Hindle, A multidimensional empirical study on refactoring activity, in: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '13, IBM Corp., 2013, pp. 132–146.

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