The Journal of Systems and Software 000 (2016) 1–20

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

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

A R T I C L E I N F O

Article history:
Received 16 February 2015
Revised 10 August 2016
Accepted 23 August 2016
Available online xxx

Keywords:
Refactoring
Software quality
Maintainability
Coding issues
Antipatterns
ISO/IEC 25010

A B S T R A C T

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 1273 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.

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 (Lientz et al., 1978). One solution 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 (1992), 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*" (Fowler, 1999). Researchers quickly recognized that this technique can also be applied to other areas, such as improving performance, security, and reliability (Mylopoulos et al., 1992). Many researchers have started to study the relation between refactoring and maintainability too, and they usually investigate different refactoring methods (mostly from Fowler's catalog (Fowler, 1999)) and their effect on code metrics, such as complexity and coupling (Sahraoui et al., 2000; Stroulia and Kapoor, 2001; Du Bois et al., 2004).

Kim et al. (2012) found in their study that, in practice, developers' views on refactoring usually differ from the academic ones. As our previous study (Szőke et al., 2014) 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. For example, Ratzinger et al. (2008) say that increasing the number of refactoring edits can decrease the number of defects, while Weißgerber and Diehl (2006a) 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

* Corresponding author.
E-mail addresses: gabor.szoke@inf.u-szeged.hu (G. Szőke), antal@inf.u-szeged.hu (G. Antal), ncsaba@inf.u-szeged.hu (C. Nagy), ferenc@inf.u-szeged.hu (R. Ferenc), gyimi@inf.u-szeged.hu (T. Gyimóthy).

http://dx.doi.org/10.1016/j.jss.2016.08.071
0164-1212/© 2016 Published by Elsevier Inc.

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 1273 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?
- 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 re-engineering’ 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).

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.

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 our previous study (Szőke et al., 2014), 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 six 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 (Bakota et al., 2011), 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 25,010 (ISO/IEC, 2005) international standard for software product quality. Thanks to the probabilistic approach, this model integrates the objective, measurable characteristics of the source code (e.g. code metrics) and expert knowledge, which

[1] Parts of the results of this study were first presented in our conference paper (Szőke et al., 2014). 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.

[2] We ended up having only six systems 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. (2011).