Science of Computer Programming 102 (2015) 44–56

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

The effect of refactoring on change and fault-proneness in commercial C# software

M. Gatrell [a], S. Counsell [b,*]

[a] MarketInvoice, Hogarth House, 136 High Holborn, London, UK
[b] Dept. of Computer Science, Brunel University, Uxbridge, Middlesex, UK

ARTICLE INFO

Article history:
Received 19 August 2011
Received in revised form 20 October 2014
Accepted 19 December 2014
Available online 24 December 2014

Keywords:
Refactoring
Change-proneness
Fault analysis
Evolution

ABSTRACT

Refactoring is a process for improving the internal characteristics and design of software while preserving its external behaviour. Refactoring has been suggested as a positive influence on the long-term quality and maintainability of software and, as a result, we might expect benefits of a lower future change or fault propensity by refactoring software. Conversely, many studies show a correlation between change and future faults; so application of a refactoring may in itself increase future fault propensity, negating any benefit of the refactoring. In this paper, we determine whether the refactoring process reaps future maintenance benefits and, as a consequence, results in software with a lower propensity for both faults and change. We studied a large, commercial software system over a twelve-month period and identified a set of refactored classes during the middle four months of the study; a bespoke tool was used to detect occurrences of fifteen types of refactoring. We then examined the fault- and change-proneness of the same set of refactored classes in the four months prior to, during, and after the period of refactoring to determine if change or fault activity was reduced either during or after the period of refactoring studied. We also compared these trends with remaining classes in the system that had not been refactored over the same periods. Results revealed that refactored classes experienced a lower change-proneness in the period after refactoring and were significantly less fault-prone during and after the period of refactoring, even when accounting for the effects of change. The study therefore presents concrete evidence of the benefits of refactoring in these two senses.

1. Introduction

Refactoring is the process of improving the internal characteristics of existing software while preserving its external behaviour [12]. Opdyke first documented the process and potential benefits of refactoring in [29], followed later by Fowler's text documenting seventy-two common refactorings [12]. Refactoring has been suggested as a positive influence on the long-term quality and maintainability of software and is also said to prevent (and even reverse) code decay [12]. A typical example of a refactoring is 'Rename Method' where a method is renamed to make its purpose more intuitive and the code more understandable as a result. While a range of recent empirical studies have used refactoring [4–6,19,27,30,31,37,36], a broad set of research questions still remain largely unanswered [25]. Since refactoring effort takes time and uses developer resources, one of these unanswered questions is whether or not future maintenance benefits are delivered as a result of

* Corresponding author.
E-mail address: steve.counsell@brunel.ac.uk (S. Counsell).

http://dx.doi.org/10.1016/j.scico.2014.12.002
0167-6423/

refactoring? Equally, are there quantifiable benefits in terms of reduced change- and fault-proneness? The stance taken in this paper is that if refactoring is undertaken, then later maintenance becomes easier. If later maintenance is easier, then by implication making changes to that method/class (whether as a result of faults of through other requested enhancements) is more likely to be understood by the maintainer. In this paper, we attempt to answer one single research question:

Does refactoring lead to less change-prone and fault-prone classes?

For example, if *ClassA* and *ClassB* exist in a software system (and exhibit a similar change and fault history) and we were to apply some refactorings to *ClassA* but not to *ClassB*, would we expect the change and fault activity of *ClassA* to increase or decrease relative to *ClassB* during the period of refactoring or during a period after refactoring? To this end, we examined a subset of a large, commercial software system over a twelve-month period. The part of the system studied was written in C# and consisted of 266,629 lines of code and 7489 classes.

For the purpose of the study analysis, we decomposed the twelve-month period into three, four-month intervals. During the middle four months of the twelve-month period, we identified which, and how frequently, a set of fifteen refactorings had been applied to classes. We used a bespoke, automated tool to mine the fifteen refactorings and detect occurrences of each. We also collected fault and change data during that middle period. Results showed that classes in the system that had been refactored had slightly lower change proneness in the period after refactoring than before. The same set of classes was found to be significantly less fault-prone during and after the period of refactoring, compared with the same classes prior to refactoring. The implication of the research is that direct benefits can therefore be ascribed to the refactoring process. While some work has explored the relationship between refactoring and fault-proneness [3,15,39], the authors know of no other work that has made the link between reduced fault-proneness and refactoring over an evolutionary period in proprietary or open-source software.

The remainder of the paper is organised as follows. In the following section, we describe the motivation for the study and related work. In Section 3, we describe preliminaries such as the system studied and the metrics extracted by the tool. We then analyse the data and discuss the key issues raised by the study (Section 4). Section 5 discusses the findings of the study and its implications and suggests threats to study validity; finally, we conclude and point to further work in Section 6.

2. Motivation and related work

The motivation for the research in this paper arises from a range of sources. First, refactoring is now a widespread developer practice, accompanied by the recent proliferation of tool support and increasing numbers of research studies in the area. An important research and practical, industry-relevant question remains unanswered: can the benefits of refactoring be quantified and, if so, is it worth doing refactoring? Second, to our knowledge, there has been only limited research to determine how refactoring is applied in evolving commercial software and little research that we know of to show a relationship between refactored code and subsequent change or fault-proneness [3,15,39]. Software practitioners need to be able to decide on a refactoring strategy that results in suitable rewards, i.e., gains in the long-term maintenance of software in return for the initial investment of refactoring. It is to shed light on this aspect of the maintenance process that provides a further motivation for our research. Specifically, we attempt to show that refactoring results in lower future maintenance and fault-fixing in terms of change- and fault- proneness, respectively.

In terms of the limited research on fault-analyses and refactoring, Bavota et al. [3] recently explored the relationship between the two in releases of three large systems using the RefFinder tool [30]. Results indicated that some types of refactorings were deemed "un-harmful" while other refactorings involving inheritance hierarchies (e.g., the pull up method refactoring) more often caused faults. Weissgerber et al. [39] explored CVS archives of three open-source Java projects [7] and found that phases of the projects with a high ratio of refactorings were followed by an increasing ratio of faults; on the other hand, there were phases where no increases were found. Murphy-Hill et al. [26] documented a broad refactoring study over a source code base serviced by a large number of developers. A range of findings were presented showing that developers refactored frequently, but tended to stick to a subset of simpler types of refactorings that were repeated often. The research also showed that a) refactoring often occurred at the same time as other changes, rather than as a separate maintenance activity and that b) refactorings were rarely acknowledged by developers in the commit comments of the source control system. Elish et al. [10] investigated the impact of refactoring on subsequent testing effort by measuring both internal quality metrics and testing effort and provided a classification of refactorings based on the impact detected. They showed that some refactorings had no effect on testing effort, while others increased subsequent testing effort.

Murphy-Hill et al. [26] and Soares [32] posited that manual refactoring was both time-consuming and error-prone, while tool supported refactoring allowed incorrect transformations to occur. Better tool support was recommended showing 'safe' refactorings – i.e., refactorings that were less likely to have an adverse impact on the software in terms of introducing faults. Soares recommended an approach that combined safe refactoring support using advanced testing techniques such as mutation testing to provide immediate developer feedback of the risk of the refactoring. Equally, the user interface provided to developers (for refactoring) has been an under-researched area of refactoring. A 'drag-and-drop' approach was recently explored by Lee et al. [18] and empirically demonstrated an improvement in this technique over traditional IDE approaches. Usha et al. [38] proposed a model for software development with refactoring and evaluated the results with metrics to measure maintainability, reusability and comprehension. From experimental results, they showed that refactoring activity in the software development process lead to improvements in reusability and comprehension, thereby enhancing code maintainability. Stroggylos [34] showed that some quality metrics, such as the Lack of COhesion in Methods (LCOM) metric of