The Journal of Systems and Software 123 (2017) 223–238

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Test coverage of impacted code elements for detecting refactoring faults: An exploratory study

Everton L.G. Alves*, Tiago Massoni, Patrícia Duarte de Lima Machado

Federal University of Campina Grande, Rua Aprigio Veloso, 882, Bairro Universitário, Campina Grande, PB, Brazil

ARTICLE INFO

Article history:
Received 30 November 2014
Revised 5 January 2016
Accepted 1 February 2016
Available online 11 February 2016

Keywords:
Testing
Refactoring
Coverage

ABSTRACT

Refactoring validation by testing is critical for quality in agile development. However, this activity may be misleading when a test suite is insufficiently robust for revealing faults. Particularly, refactoring faults can be tricky and difficult to detect. Coverage analysis is a standard practice to evaluate fault detection capability of test suites. However, there is usually a low correlation between coverage and fault detection. In this paper, we present an exploratory study on the use of coverage data of mostly impacted code elements to identify shortcomings in a test suite. We consider three real open source projects and their original test suites. The results show that a test suite not directly calling the refactored method and/or its callers increases the chance of missing the fault. Additional analysis of branch coverage on test cases shows that there are higher chances of detecting a refactoring fault when branch coverage is high. These results give evidence that a combination of impact analysis with branch coverage could be highly effective in detecting faults introduced by refactoring edits. Furthermore, we propose a statistic model that evidences the correlation of coverage over certain code elements and the suite's capability of revealing refactoring faults.

1. Introduction

Refactoring improves quality factors of a program while preserving its external behavior (Fowler et al., 1999; Mens and Tourwé, 2004). Refactoring edits are one of the foundations of agile software development. In the agile community, the refactoring activity is known to confine the complexity of a source code, improving non-functional aspects of a software such as decreased coupling and increased cohesion (Moser et al., 2008). Fowler et al. (1999) lists four advantages that refactoring brings in the context of Agile Methods: (i) it helps developers to program faster; (ii) it improves the design of the software; (iii) it makes software easier to understand; and (iv) it helps developers to find bugs.

Recent studies have evidenced that nearly 30% of the changes performed during software development are likely to be refactorings (Soares et al., 2011). For example, code clones spread throughout several methods of a class can be unified into a single method, then replacing the clones by a call to this new method; this is the Extract Method refactoring (Fowler et al., 1999), which is one of the most widely applied (Murphy et al., 2006).

Although there are several automatic refactoring tools in popular IDEs, developers still perform most refactorings manually. Murphy et al. (2006) find that about 90% of refactoring edits are manually applied. Negara et al. (2013) agree by showing that expert developers prefer manual refactorings over automated. Usability issues seem to have a negative impact on developers' confidence on those tools (Lee et al., 2013). Moreover, recent works show that incorrect refactorings – unexpectedly changing behavior – are present even in the most used tools (Daniel et al., 2007; Soares et al., 2013).

In such scenario, developers widely use *regression test suites* for validating manually-applied refactoring edits. As such, refactoring edits are error prone and require validation, as subtle faults may pass unnoticed. Dig and Johnson (2005) state that nearly 80% of the changes that break client applications are API-level refactoring edits. In addition, 77% of the participants from Kim et al.'s survey with Microsoft developers (Kim et al., 2012) confirm that refactoring may induce the introduction of subtle bugs and functionality regression. A regression test suite, however, may be ineffective in finding refactoring faults. Also, it may be impractical to rerun and analyze the execution results of the whole test suite after each refactoring edit. Techniques that minimize the test suite, while maintaining its effectiveness, are desirable.

Nevertheless, this intuition has little scientific evidence; it is important to distinguish which impacted methods, if called by the

* Corresponding author. Tel.: +55 83 3310 1122.
E-mail addresses: everton@copin.ufcg.edu.br (E.L.G. Alves), tiago@computacao.ufcg.edu.br (T. Massoni), patricia@computacao.ufcg.edu.br (P.D.d.L. Machado).

http://dx.doi.org/10.1016/j.jss.2016.02.001
0164-1212/

test suite, are most effective in detecting faults that might be led to by refactoring. In this paper, we present an exploratory study, performed on three real open-source Java projects, with seeded faults related to two of the most common refactoring edits, Extract Method and Move Method. Using the actual test suites from the selected projects, we measure the *direct calls (first-level coverage)* to several groups of methods possibly impacted by a refactoring edit, relating these data to the status of the test case – whether it detects or not the seeded fault.

Overall, only 67% of the seeded 270 faults were detected by the project's test suite. The lack of test cases calling the method whose body is changed seem to be very relevant – 70% of the unrevealed faults present this property. Similarly, 51% of the undetected faults are missed by test cases that directly call the callers of the changed method. On the other hand, for 78% of the detected faults, the test suite included at least one test case that calls the refactored method directly. Considering callers, this rate was also high (70%). In 62% of these suites there were test cases that cover, at first level, both the refactored method and its callers. The detection results did not present statistical dependence with the type of refactoring (same with the type of seeded fault). Based on our results, we propose a statistical model that uses first-level coverage data to foresee chances a test suite has to detect refactoring faults.

First-level coverage reports on a direct need for the agile developer – identify the calls that must be made in a test case for improving its chance of detecting refactoring faults. On the other hand, indirect coverage of impacted elements may not be applicable in a agile context. The reason is that it can be tricky and demand high costs for a developer to assess fault detection capability in indirect calls. For instance, after applying an Extract Method, it seems intuitive that tests directly calling the changed method, its callers, and callees present good chances of detecting any newly-introduced fault. When considering first-level coverage, we are also focusing on test case expressiveness regarding refactoring edits. When a test case that calls directly a method fails, it may be more helpful to locate the fault.

Considering that several faults were missed by test cases even with first-level coverage of impacted methods, we additionally analyzed test cases that exercise the modified method and their callers. If at least one test case in the suite called the changed method, and the branch coverage of this method was greater than 75%, 91% of the faults were detected. If callers of the changed method were directly accessed, in 88% of the cases, the faults were detected with high branch coverage. For suites with low branch coverage (less than 25%), detection dropped to 66% and 62%, respectively. These results provide a good case for tests with direct calls combined with high branch coverage.

As another additional study, we explored the relationship between first-level coverage of impacted elements and binding issues with refactored variables within class hierarchies. Previous research (Soares et al., 2011) reported on several subtle faults in manual and automated refactoring being due to homonymous variables or methods being confused by refactored statements, so we extended our investigation for relating test cases with this kind of refactoring fault. Similarly to the other studies, this investigation showed that when a test suite covers the refactored method and its callers better are the chances of detecting binding-related faults introduced when refactoring.

We published a preliminary version of this study (Alves et al., 2014a) in which a single refactoring type and refactoring fault are analyzed. The current paper extends our previous study by investigating new refactoring types, new refactoring faults, by adding statistical validation to the conclusions, and by proposing new artifacts to help the evaluation of a test suite regarding its detection of refactoring faults.

Section 2 brings a motivating example for the problem of test cases that miss refactoring bugs. Next, we present the setup and research questions investigated by the experimental studies (Section 3), then Section 4 includes the results and discussion for the main experimental study. In Section 5, we extend the study to relate its results with branch coverage within the exercised methods, while Section 6 presents an exploratory study for binding-related refactoring faults. Section 7 discusses threats to validity. The last two sections cover the related work and concluding remarks, respectively.

2. Motivating example

In agile methodologies, even simple solutions may need improvement when requirement changes must be incorporated into the code base, so manual refactoring is frequent. Automation of refactoring is common, but here we focus on manual refactoring.

Opportunities for code improvement often involve code duplication, and its minimization or elimination is often desirable. For this task, the Extract Method refactoring (Fowler et al., 1999) encompasses small changes that group together multiple code fragments into a new method; the new method has a name explaining its purpose, and the original fragments are then replaced by a call to this method. When applying this edit, developers must be cautious: the new method must receive parameters that correspond to the variables manipulated by the grouped fragment, and a return value must be correctly provided to the callers; also, the new method could be changing the behavior of the target class.

Suppose that, after working on several tasks, a developer notices an opportunity of reducing code duplication. Fig. 1 presents two fragments of her code before and after the Extract Method refactoring; Lines 5–8 from `Elementm(boolean)` – Fig. 1(a) – are extracted into the `n` method in Fig. 1(b). Following principles for validating refactorings, and assuming that her test suite passes before and after the edits, also, no compilation error was found. Thus, she may become confident that the modification is correct, and commits the code to the code base. The *behavior*, however, is undesirably modified. If `b` is true, `x` finishes with value *23*, `x` is updated before throwing the exception. After extracting the method, the exception is thrown with the global `x` with its initial state *42*.

This example shows a very subtle refactoring fault that may easily go undetected, especially if the test suite does not exercise the `test` method. By examining previous research on change impact analysis (Ren et al., 2004; Zhang et al., 2011; Ryder and Tip, 2001), we can establish that, for an Extract Method refactoring edit, the methods most likely to be impacted are: (i) *the original method* (*M*), in this case, the `m` method; (ii) *the callers of the method under refactoring* (*C*), as methods that call `m()` might be negatively influenced in case they use `m`'s return value, and/or any variable handled by `m`; (iii) *the callees of the method under refactoring* (*Ce*), so if given methods that `m` calls require as pre-requisite the program to be in a certain state, then `m` must be run according to its previous behavior; and (iv) *methods with similar signature to the newly added one* (*O*). An extracted method may break or introduce overriding/overloading contracts causing a behavior change; for instance, `m` could already be declared within `Element`'s hierarchy.

It is expected that, by presenting first-level coverage of the methods potentially impacted by the refactoring, a test case is expected to detect such fault – this is an easy to follow guideline for writing appropriate test cases. Nevertheless, first-level coverage alone may mislead developers/testers (Inozemtseva and Holmes, 2014). More specifically, with refactoring faults, there is no evidence whether this type of coverage is a good quality measure in this context, or which methods a test suite should call to