On the Effects of Pair Programming on Thoroughness and Fault-Finding Effectiveness of Unit Tests

Lech Madeyski

Institute of Applied Informatics, Wroclaw University of Technology,
Wyb.Wyspianskiego 27, 50370 Wroclaw, Poland
Lech.Madeyski@pwr.wroc.pl
http://madeyski.e-informatyka.pl/

Abstract. Code coverage and mutation score measure how thoroughly tests exercise programs and how effective they are, respectively. The objective is to provide empirical evidence on the impact of pair programming on both, thoroughness and effectiveness of test suites, as pair programming is considered one of the practices that can make testing more rigorous, thorough and effective. A large experiment with MSc students working solo and in pairs was conducted. The subjects were asked to write unit tests using JUnit, and to follow test-driven development approach, as suggested by eXtreme Programming methodology. It appeared that branch coverage, as well as mutation score indicator (the lower bound on mutation score), was not significantly affected by using pair programming, instead of solo programming. However, slight but insignificant positive impact of pair programming on mutations score indicator was noticeable. The results do not support the positive impact of pair programming on testing to make it more effective and thorough. The generalization of the results is limited due to the fact that MSc students participated in the study. It is possible that the benefits of pair programming will exceed the results obtained in this experiment for larger, more complex and longer projects.

1 Introduction

Pair programming (PP) [1] is key software development practice of eXtreme Programming (XP) methodology [2] which has recently gained a lot of attention. Pair programming is a practice in which two distinct roles, called a driver and a navigator, are distinguished. They contribute to a synergy of the individuals in a pair working together at one computer and collaborating on the same development tasks (e.g. design, test, code). The driver is typing at the keyboard and focusing on the details of the production code or tests. The navigator observes the work of the driver, reviews the code, proposes test cases, considers the strategic implications [3,4] and is looking for tactical and strategic defects or alternatives [5]. The rule is that all production code is written by two people

J. Münch and P. Abrahamsson (Eds.): PROFES 2007, LNCS 4589, pp. 207–221, 2007.

sitting at one machine [2]. In the case of solo programming, all activities are performed by one programmer.

Test-driven development (TDD) [6,2], also known as test-first programming, is another important and well known software development practice of XP methodology, supposed to be used with pair programming. TDD is a practice based on specifying piece of functionality as a test (usually low-level unit test), before writing production code, implementing the functionality, so that the test passes, refactoring (e.g. removing duplication), and iterating the process. The tests are run frequently, while writing production code. Kobayashi et al. [7] suggested that pair programming, test-driven development and refactoring, which is the inherent part of TDD development cycle, had a very good synergy. Therefore, it seems reasonable to evaluate pair programming practice in the context of TDD.

Pair programming is supposed to be software development practice that can influence unit testing to make it more rigorous, thorough, and effective. The question is whether the impact of pair programming is significant or not.

2 Measures

Programmers who write unit tests should have a set of guidelines indicating whether their software has been thoroughly and effectively tested.

2.1 Code Coverage

Measuring code coverage is one of such guidelines which can be applied, as code coverage tools measure how thoroughly tests exercise programs [8]. However, it remains a controversial issue whether code coverage is a good indicator for fault detection capability of test cases [9]. Marick [8] shows that code coverage may be misused, but code coverage tools are still helpful if they are used to enhance thought, and not to replace it. Cai and Lyu [10] found that code coverage was a good estimator for fault detection of exceptional test cases, but a poor one for test cases in normal operations.

Kaner [9] lists 101 coverage measures. The important question is which code coverage measure should be used. Useful insights concerning this question are given by Cornett [11]. Statement coverage, also known as line coverage, reports whether each executable statement is encountered. The main disadvantage of statement coverage is that it is insensitive to some control structures. To avoid this problem, decision coverage, also known as branch coverage, has been devised. Decision coverage is a measure based on whether decision points, such as `if` and `while` statements, evaluate to both true and false during test execution, thus exercising both execution paths. Decision coverage includes statement coverage, since exercising every branch must lead to exercising every statement. However, a shortcoming of this measure is that it ignores branches within boolean expressions which occur due to short-circuit operators. For example, it can preclude calls to some methods. Unfortunately, the most powerful measures as Modified Condition/Decision Coverage (MC/DC), created at Boeing and required for aviation software, or Condition/Decision Coverage are not available