Inspecting Automated Test Code: A Preliminary Study

Filippo Lanubile and Teresa Mallardo

Dipartimento di Informatica, University of Bari,
70126 Bari, Italy
{lanubile,mallardo}@di.uniba.it

Abstract. Testing is an essential part of an agile process as test is automated and tends to take the role of specifications in place of documents. However, whenever test cases are faulty, developers' time might be wasted to fix problems that do not actually originate in the production code. Because of their relevance in agile processes, we posit that the quality of test cases can be assured through software inspections as a complement to the informal review activity which occurs in pair programming. Inspections can thus help the identification of what might be wrong in test code and where refactoring is needed. In this paper, we report on a preliminary empirical study where we examine the effect of conducting software inspections on automated test code. First results show that software inspections can improve the quality of test code, especially the repeatability attribute. The benefit of software inspections also apply when automated unit tests are created by developers working in pair programming mode.

Keywords: Automated Testing, Unit Test, Refactoring, Software Inspection, Pair Programming, Empirical Study.

1 Introduction

Extreme Programming (XP), and more generally agile methods, tend to minimize any effort which is not directly related to code completion [3]. A core XP practice, pair programming, requires two developers work side-by-side at a single computer in a joint development effort [21]. While one (the Driver) is typing on the keyboard, the other (the Navigator) observes the work and catches defects as soon as they are entered into the code. Although a number of research studies have shown that this form of continuous review, albeit informal, can assure a good level of quality [15, 20, 22], there is still uncertainty about benefits from agile methods, in particular for dependable systems [1, 17, 18]. In particular, some researchers propose to combine agile and plan-driven processes to determine the right balancing process [4, 19].

Software inspections are an established quality assurance technique for early defect detection in plan-driven development processes [6]. With software inspections, any software artifact can be the object of static verification, including requirements specifications, design documents as well as source code and test cases. However, test cases are the least reviewed type of software artifact with plan-driven methods [8], because

G. Concas et al. (Eds.): XP 2007, LNCS 4536, pp. 115–122, 2007.
© Springer-Verlag Berlin Heidelberg 2007

testing comes late in a waterfall-like development process and might be minimized if the project is late or out of budget.

On the contrary, testing is an essential part of an agile process. No user stories can be considered ready without passing its acceptance tests and all unit tests for a class should run correctly. With automated unit testing, developers write test cases according to the xUnit framework in the same programming language as the code they test, and put unit tests under software configuration management together with production code. In Test-Driven Development (TDD), another XP core practice, programmers write test cases first and then implement code which successfully passes the test cases [2]. Although some researchers argue that TDD is helpful for improving quality and productivity [5, 10, 13], writing test cases before coding requires more effort than writing test cases after coding [13, 14]. With TDD, test cases take the role of specification but this does not exclude errors. Test cases themselves might be incorrect because they do not represent the right specification and developers' time might be wasted to fix problems that do not actually originate in the production code.

Because of their relevance in agile processes, we posit that the quality of test cases can be assured through software inspections to be conducted in addition to the informal review activity which occurs in pair programming. Inspections can thus help the identification of "test smells", which are symptoms that something might be wrong in test code [11] and refactoring can be needed [23]. In this paper we start to examine the effect of conducting software inspections on automated test code. We report the results of a repeated case study in an academic setting where unit test cases, which have been produced by pair and solo groups, have been inspected to assess the quality of test code. The remainder of this paper is organized as follows. Section 2 gives background information about quality of test cases and symptoms of problems. Section 3 describes the empirical study and presents the results from data analysis. Finally, conclusions are presented in Section 4.

2 Quality of Automated Tests

Writing good test cases is not easy, especially if tests have to be automated. When developers write automated test cases, they should take care that the following quality attributes are fulfilled [11]:

Concise. A test should be brief and yet comprehensive.
Self checking. A test should report results without human interpretation.
Repeatable. A test should be run many consecutive times without human intervention.
Robust. A test should produce always the same results.
Sufficient. A test should verify all the major functionalities of the software to be tested.
Necessary. A test should contain only code to the specification of desired behavior.
Clear. A test should be easy to understand.
Efficient. A test should be run in a reasonable amount of time.
Specific. A test failure should involve a specific functionality of the software to be tested.

Independent. A test should produce the same results whether it is run by itself or together with other tests.
Maintainable. A test should be easy to modify and extend.
Traceable. A test should be traceable to and from the code and requirements.

Lack of quality in automated test can be revealed by "test smells" [11], [12], [23], which are a kind of code smells as initially introduced by Fowler [7], but specific for test code:

Obscure test. A test case is difficult to understand at a first reading.
Conditional test logic. A test case contains conditional logic within selection or repetition structures.
Test code duplication. Identical fragments of test code (clones) appear in a number of test cases.
Test logic in production. Production code contains logic that should rather be included into test code.
Assertion roulette. When a test case fails, you do not know which of the assertions is responsible for it.
Erratic test. A test that gives different results, depending on when it runs and who is running it.
Fragile test. A test that fails or does not compile after any change to the production code.
Frequent debugging. Manual debugging is required to determine the cause of most test failures.
Manual intervention. A test case requires manual changes before the test is run, otherwise the test fails.
Slow test. The test takes so long that developers avoid to run it every time they make a change.

3 Empirical Investigation of Test Quality

The context of our experience was a web engineering course at the University of Bari, involving Master's students in computer science engaged in porting a legacy web application. The legacy application provides groupware support for distributed software inspections [9]. The old version (1.6) used the outdated MS ASP scripting technology and had become hard to evolve. Before the course start date, the application had been entirely redesigned according to a four-layered architecture. Then porting to MS .NET technology started with a number of use cases from the old version successfully migrated to the new one.

As a course assignment, students had to complete the migration of the legacy web application. Test automation for the new version was part of the assignment. Students were following the process model shown in Fig. 1. To realize the assigned use case, students added new classes for each layer of the architecture, then they submitted both source code and design document to a two-person inspection team which assessed whether the use case realization was compliant to the four-layered architecture.

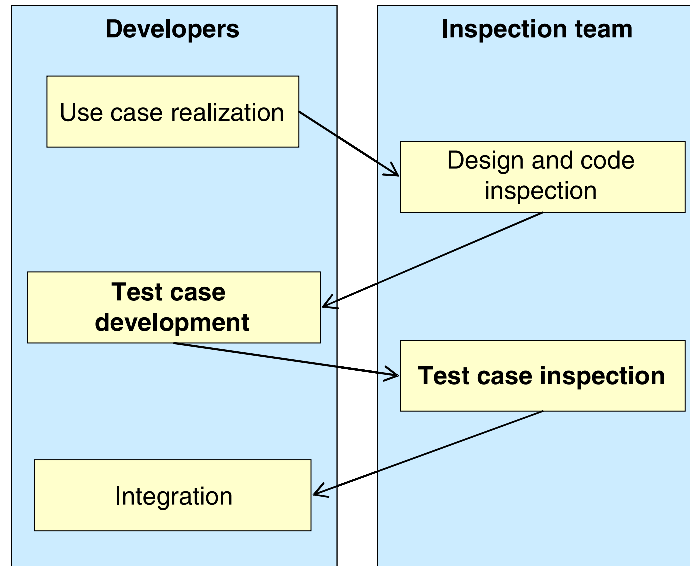

Fig. 1. The process for use case migration

In the test case development stage, students wrote unit test cases in accordance with the NUnit framework [16]. Students were taught to develop each test as a method that implements the Four Phases Test pattern [11]. This test pattern requires a test to be structured with four distinct phases that are executed in sequence. The four test phases are the following:

- *Fixture setup*: making conditions to establish the prior state (the fixture) of the test that is required to observe the system behavior
- *Exercise system under test*: causing the software we are testing to run.
- *Result verification*: specifying the expected outcome.
- *Fixture teardown*: restoring the initial conditions of the system in which it was before the test was run.

In the test case inspection stage, automated unit tests were submitted to the same two-person inspection team as in the previous design and code inspection. This time the goal of the inspection was to assess the quality of test code. For this purpose, the inspectors used the list of test smells as a checklist for test code analysis. Finally, the migrated use cases, which implemented all corrections from the inspections, could be integrated to the baseline.

Table 1 characterizes the results of students' work. Six students had redeveloped four use cases, two of which in pair programming (PP) and the other two use cases in solo programming (SP). Class methods include only those methods created for classes in the data and domain layers. Students considered only public methods for being tested. For each method under test, test creation was restricted to one test case, with the exception of a method in UC4 which had two test cases.

Table 1. Characterization of the migration tasks

	UC1	*UC2*	*UC3*	*UC4*
Programming Model	solo programming (SP)	pair programming (PP)	pair programming (PP)	solo programming (SP)
Class methods	26	42	72	31
Methods under test	12	23	35	20
Test cases	12	23	35	21

Table 2 reports which test smells were found by test case inspectors and their occurrences for each use case.

The most common indicator of problems was the need for manual changes before launching a test. Test cases often violated the Four Phases Test pattern, and this occurred in all the four use cases. In particular, we found that the fixture setup and teardown phases were missing some critical actions. For example, in UC3 and UC4, developers were testing class methods that delete an item in the repository. However, the fixture setup phase were not adding to the repository the item to be deleted, while the fixture teardown phase was missing at all. More generally, when a test case modified the application state permanently, tests failed and manual intervention was required to restore the initial state of the system. This negatively affected the repeatability of tests.

Two other common smells found in the test code were assertion roulette and conditional test logic. The root cause for these issues were developers' choice of writing one test case for each class method under test. As a consequence, a test case verified different behaviors of a class method using multiple assertions and conditional statements. Test case overloading hampered the clarity and maintainability of tests.

Another common problem was test code duplication which was mainly due to "copy and paste" practices applied to the fixture setup phase. It was easily resolved by extracting instructions included in the setup part from the fixture of a single test case to the shared fixture.

Table 2. Results from test case inspections

	UC1 (SP)	*UC2 (PP)*	*UC3 (PP)*	*UC4 (SP)*
Manual intervention	10	10	17	14
Assertion roulette	2	16	15	4
Conditional test logic	1	8	2	6
Test code duplication	1	7	6	1
Erratic test	1	1	2	0
Fragile test	0	0	1	3
Total issues	15	42	46	28
Issue density	1.2	1.8	1.3	1.3

Erratic tests were also identified as they were caused by test cases which depended on other test cases. When these test cases were running isolated they provided different results from test executions which included coupled test cases. Test case inspections allowed to identify those test code portions in which the dependencies were hidden.

Finally, there were few indicators of fragile tests because of data sensitivity, as the tests failed when the contents of the repository was modified.

The last two rows of Table 2 report, respectively, the total number of issues and issue density, that is the number of issues per test case. Results show that there were more test case issues in UC2 and UC3 than in UC1 and UC4. However, this difference is only apparent. If we consider the issue density, which takes into account size differences, we can see that pair programming and solo programming provide the same level of test quality.

4 Conclusions

In this paper, we have reported on an empirical study, conducted at the University of Bari, where we examine the effect of conducting software inspections on automated test code. Results have shown that software inspections can improve the quality of test code, especially the repeatability of tests, which is one of the most important qualities of test automation. We also found that the benefit of software inspections can be observed when automated unit tests are created by single developers as well as by pairs of developers.

The finding that inspections can reveal unknown flaws in automated test code, even when using pair programming, is in contrast with the claim that quality assurance is already included within pair programming, and then software inspection is a redundant (and then uneconomical) practice for agile methods. We can rather say that, even if developers are applying agile practices on a project, if a product is particularly high risk it might be worth its effort to use inspections, at least for key parts such as automated test code.

The results show a certain tendency but are not conclusive. A threat to validity of our study is that we could not observe the developers while working, so we cannot exclude that pairs effectively worked as driver/observer rather than splitting the assignment and working individually. Another drawback of this study is that it represents only a small study, using a small number of subjects in an academic environment. Therefore, results can only be preliminary and more investigations have to follow.

As further work we intend to run a controlled experiment in the next edition of our course to provide more quantitative results about benefits of test cases inspections. We also encourage researchers to replicate the study in different settings to analyze the application of inspections in agile development in more detail.

Acknowledgments. We would like to thank Domenico Balzano for his help in test case inspections.

References

1. Ambler, S.W.: When Does(n't) Agile Modeling Make Sense? http:// www.agilemodeling.com/essays/whenDoesAMWork.htm
2. Beck, K.: Test Driven Development: By Example. Addison-Wesley, New York, NY, USA (2002)
3. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, New York, NY, USA (2000)
4. Boehm, B., Turner, R.: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley, New York, NY, USA (2003)
5. Erdogmus, H., Morisio, M., Torchiano, M.: On the Effectiveness of the Test-First Approach to Programming. In: IEEE Transactions on Software Engineering, vol. 31(3), pp. 226–237. IEEE Computer Society Press, Los Alamitos, CA, USA (2005)
6. Fagan, M.E.: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, vol. 15(3), Riverton, NJ, USA, pp. 182–211 (1976)
7. Fowler, M.: Refactoring. In: Improving the Design of Existing Code, Addison-Wesley, New York, NY, USA (1999)
8. Laitenberger, O., DeBaud, J.M.: An encompassing life cycle centric survey of software inspection. In: The Journal of Systems and Software, vol. 50(1), pp. 5–31. Elsevier Science Inc, New York, NY, USA (2000)
9. Lanubile, F., Mallardo, T., Calefato, F.: Tool Support for Geographically Dispersed Inspection Teams. In: Software Process: Improvement and Practice, vol. 8(4), pp. 217–231. Wiley InterScience, New York (2003)
10. Maximilien, E.M., Williams, L.: Assessing Test-Driven Development at IBM. In: Proceedings of the International Conference on Software Engineering (ICSE'03), pp. 564–569 (2003)
11. Meszaros, G.: XUnit Test Patterns: Refactoring Test Code. Addison Wesley, New York, NY, USA (to appear in 2007). Also available online at http://xunitpatterns.com/
12. Meszaros, G., Smith, S.M., Andrea, J.: The Test Automation Manifesto. In: Maurer, F., Wells, D. (eds.) XP/Agile Universe 2003. LNCS, vol. 2753, pp. 73–81. Springer, Heidelberg (2003)
13. Muller, M.M., Tichy, W.E.: Case Study: Extreme Programming in a University Environment. In: Proceedings of the International Conference on Software Engineering (ICSE'01), pp. 537–544 (2001)
14. Muller, M.M., Hagner, O.: Experiment about Test-First Programming. In: Proceedings of the International Conference on Empirical Assessment in Software Engineering (EASE'02), pp. 131–136 (2002)
15. Muller, M.M.: Two controlled experiments concerning the comparison of pair programming to peer review. In: The Journal of Systems and Software, vol. 78(2), pp. 166–179. Elsevier Science Inc., New York, NY, USA (2005)
16. Nunit Development Team: Two, M.C., Poole, C., Cansdale, J., Feldman, G.: http://www.nunit.org
17. Paulk, M.: Extreme Programming from a CMM Perspective. In: IEEE Software, vol. 18(6), pp. 19–26. IEEE Computer Society Press, Los Alamitos, CA, USA (2001)
18. Rakitin, S.: Letters: Manifesto Elicits Cynicism. In: IEEE Computer, vol. 34(12), IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 4, 6–7 (2001)
19. Reifer, D.J., Maurer, F., Erdogmus, H.: Scaling Agile Methods. In: IEEE Software, vol. 20(4), pp. 12–14. IEEE Computer Society Press, Los Alamitos, CA, USA (2003)

20. Tomayko, J.: A Comparison of Pair Programming to Inspections for Software Defect Reduction. Computer Science Education, vol. 12(3). Taylor & Francis Group, pp. 213–222 (2002)
21. Williams, L., Kessler, R.R.: Pair Programming Illuminated. Addison-Wesley, New York, NY, USA (2002)
22. Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R.: Strengthening the Case for Pair Programming. In: IEEE Software, vol. 17(4), pp. 19–25. IEEE Computer Society Press, Los Alamitos, CA, USA (2000)
23. van Deursen, A., Moonen, L., van den Bergh, A., Kok, G.: Refactoring Test Code. In: Proceedings of the 2nd International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP'01) (2001)