Automated Acceptance Testing vs. Quality: A Case Study of an Open Source Project

John Noll
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland

Ermanno Pirotta
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland

Carlos Solis
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland

Xiaofeng Wang
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland

18 May 2011

Contact

Address	Lero International Science Centre University of Limerick Ireland
Phone	+353 61 233799
Fax	+353 61 213036
E-Mail	info@lero.ie
Website	http://www.lero.ie/

This work is partially supported by Science Foundation Ireland
under grant no. 03/CE2/I303-1

Automated Acceptance Testing vs. Quality: A Case Study of an Open Source Project

John Noll, Ermanno Pirotta, Carlos Solís, and Xiaofeng Wang

Lero, the Irish Software Engineering Research Centre,
University of Limerick,
Limerick, Ireland
{john.noll,ermanno.pirotta,carlos.solis,xiaofeng.wang}@lero.ie

Abstract. Automated accept testing is an emerging practice that is claimed to yield many benefits to software development projects, among which is higher quality of the software product itself. Yet there is little empirically grounded evidence to support such claims. In this paper an open source software project was studied to explore the link between automated acceptance testing and the quality of code. The findings of our study show that such link cannot be established readily. Further investigation is needed and our study provides a practical way to conduct such studies.

1 Introduction

Acceptance testing, defined in the IEEE Standard 1012-1986, is "a formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system". It is usually performed on the entire system or a large part of it [12]. Within short iterative software development processes (weekly, bi-weekly, etc.), if acceptance tests are performed manually, there will be lot of time allocated to testing within each iteration. In addition, manual acceptance testing will in most cases be tedious, expensive and time consuming [5, 7]. Automated acceptance testing has emerged as a promising initiative to ease and improve this process, the basic idea of which is to document requirements and desired outcome in a format that can be automatically and repeatedly tested.

There are reported benefits of the adoption of automated acceptance testing, among which is the improved software quality. However, these claims are based on industrial experience reports [2, 3, 6, 9, 15], with limited empirical evidence that is largely drawn from controlled experiments with students [11, 12, 14]. In particular, how the adoption of automated acceptance testing affects the quality of open source software has not been investigated. This observation has inspired our study.

The objective of this study is to explore the link between automated acceptance testing and the quality of code in open source software development. To this end, we adopted a case study approach to examine Zope3, an open source project that deployed automated acceptance testing in its development activities.

The remaining part of this paper is organized as follows. Section 2 is a review of related work that reports the benefits of adopting automated acceptance testing. In Section 3 we describe our research design. The findings of the study are reported in Section 4, which is followed by a discussion section where we interpret the findings and explicate the limitations of our study. The paper concludes with contributions and potential future work.

2 Related Work

The benefits of the adoption of acceptance testing have been espoused by both industry and research communities. In particular, the use of acceptance tests as an effective way to express and specify requirements has been reported in the literature. Miller and Collins [9, p.1] observe that acceptance tests "capture user requirements in a directly and verifiable way and they measure how well the system meets those requirements". In the same vein, Martin et al. [8, p. 610] state that "automated tests are unambiguous - there are clear criteria established for a successful test and the test results are checked automatically against them". This results in an overall improvement of the communication between the different stakeholders involved in a project [10] and their level of agreement [1]. Knowledge sharing is enhanced [9] and acceptance tests also represent a way to learn more about the specific domain of the application being developed. For instance, in their report on the use of the FIT acceptance testing framework, Prashant Ghandi et al. [3] note that "through our use of FIT documents, we evolved our understanding of the project's domain and found that our FIT documents led us to discover new domain concept" (p. 255) and "by collaborating closely on the FIT documents, the developers and customers reach a shared understanding of the domain and developer the ubiquitous language of the application (p. 253).

Acceptance tests have been also considered to improve the overall development process. They provide a good indicator of the project status against customer expectations [9, 13]. They also increase the overall confidence of the functionalities being developed by the development team [3].

Several authors report that the adoption of automated acceptance testing improves the quality of software products. Crispin and House [2] state that automated acceptance testing helps in finding bugs at the early stages and therefore helps avoiding overwork at the end of the iteration; Miller and Collins [9] suggest that automated acceptance tests expose problems that unit tests are missing. Hanssen and Haugset [5] also argue in favour of automated acceptance testing based on the fact that it contributes to improving the quality of the product by allowing more bugs to be detected before production release.

Nevertheless, the authors of this paper are not aware of any empirical study which investigates the link between the quality of code and automated acceptance testing. Most of the existing studies are in the form of industry experience reports [2, 3, 6, 9, 15] and controlled experiments with students [11, 12, 14]. They respectively report mainly best practices and how acceptance tests can bridge

the gap between requirements and deliverables, as well as the effort required to learn and implement them.

Based on the review of related work, we have phrased the hypothesis of our study:

Hypothesis 1 *Automated acceptance testing has a positive effect on the quality of code.*

3 Research Design

In this section we detail the approach taken for our study, which comprised the following steps.

3.1 Select Metrics to Measure Quality of Code

A common measure of code quality is defect density: the number of defects per source line of code (defects/SLOC). This is typically measured at two points in time: at the transition from one phase to the next (for example, from functional to system testing), and at some fixed time (90 or 180 days) after release. The latter is sometimes referred to as "external" defect density.

Defect density is a relevant indicator of code quality because the defects are usually detected by actual failures, as revealed by tests or, in the case of external defect density, bugs reported by users. As such, faults that do not cause failures are not included in defect density, and so this metric provides a more user-centric measure of quality. In this study we used external defect density as an indicator of quality since this data is publicly accessible in the case project we studied.

3.2 Select Project and Release

The projects considered for our study are open source software (OSS) projects. The project selecting conditions are that a) the project has used automated acceptance testing; b) the bug repository is easy to access and contains necessary information to attribute each bug to source code; and c) it is possible to find a release version of the product to which the reported bugs can be attributed.

We have reviewed the websites of several major OSS Projects including Drupal (`http://drupal.org/`), Matterhorn `http://www.opencastproject.org/project/matterhorn`, NetBeans `http://netbeans.org/`, Plone `http://plone.org/` and Zope `http://www.zope.org/`. They are well-established projects with many collaborators and a significant amount of code. At the end the only project which satisfied the selecting conditions was Zope. Zope is a successful project with an active developer community and installed base. It is now in its third major release, Zope3, which has automated acceptance testing as a key component in the development process. A significant suite of automated functional test is part of the code base; and the suite includes scripts for calculating test coverage of classes. Zope3 is also old enough to have accumulated a significant collection of issue reports.

The sub-release we decided to study is Release 3.1.0-final of Zope3, which is the first stable release of Zope3 available to users. We have selected only the bugs which have been fixed, because they can be traced to source code through inspecting log files.

3.3 Identify Test-covered Code and Uncovered Code

Zope3 has a test suite that is configurable with many options, including options to run unit tests only, or all tests except unit tests; there is also an option to calculate and report test coverage. When the test suite is executed, each test logs the test name and number, the subsystem, class, and in some cases the method under test, and the result of the test; thus, by comparing the test log report of subsystems and classes tested with the Zope3 source code, each method in Zope3 can be classified as covered or uncovered by acceptance test(s).

When run with the coverage reporting option, the Zope3 test suite generates a coverage report for each python file in the Zope3 codebase; this file contains the number of times a test has called a line of the original code. In this way, a line of code is uncovered if it is called zero times, and it is covered if it is called at least once. In addition, a summary report which gives the percentage of lines covered by an acceptance test in each directory of the application. Therefore, it is possible to know if each method in a class is covered or uncovered.

3.4 Assess Quality of Test-covered and Uncovered Code

Issues reported on Zope 3 Release 3.1.0-final by both developers and users, and were collected in the publicly accessible Zope3 issue database at `https://bugs.launchpad.net/zope3/`. We have selected bugs that were reported from 2 October, 2005 (the release dates of 3.1.0-final) to 6 December, 2005 (the release date of 3.2.0 beta-1, the next release following 3.1.0-final). We assumed that bugs reported within this time range could be attributed to Release 3.1.0-final exclusively, as there is no stable (non-developer/experimental) release prior to 3.1.0-final.

The bug descriptions in the bug tracking system and the source code in the subversion system were analyzed in order to determine which lines were the roots of a bug. Each of the bug reports was examined by at least one of the authors; in the case of uncertainty, the conclusion was verified by another author.

Generally the bug reports include comments about the circumstances in which the unexpected behavior occurs, the class or file where the source of the bug is located, and possible solutions. Sometimes they include a trace of the exceptions thrown, which indicates files, methods, and line numbers. Other times, they include the revision version in which the bug was corrected, which allows the corrected files to be compared to their previous buggy versions. When a revision number containing a fix for a bug was not included, we had to explore the version system for the files described in the report, and look for a fix that

matched the bug description. In this way, the line(s) of code associated with each bug can be identified.

Since each line of code can be classified as covered or not covered by acceptance tests, bugs can also be divided into covered and uncovered, according to whether the source of the bug was in covered code or uncovered code.

To calculate the density of external bugs from both test-covered and uncovered code, we need to know the total lines of covered and uncovered code, from which the test coverage of Zope3 3.1.0-final can be calculated. Due to the large amount of source files in Zope3 and a lack of clearly defined boundary of Zope3 software package, we were not able to obtain the exact number of lines of code of Zope3 3.1.0-final. However, the test coverage of files tested by the Zope3 test suite is calculated by the Zope3 test coverage scripts. Since tested files are a subset of the files in the Zope3 codebase, this reported test coverage is the upper bound of the test coverage of the whole Zope3 package. It is calculated using the following formula:

$$UpperBoundCoverage = \frac{\sum_{i=1}^{n\_zopeapp\_files} cvLOC_i}{tLOC} \tag{1}$$

where *n_zopeapp_files* is the number of files tested by the test suite, $cvLOC$ is the number of covered lines of code and $tLOC$ is the total number of lines of code. It is an optimistic estimation since we have not taken into account the files without any coverage. This approximation of test coverage, even though not ideal, serves the purpose of our study.

4 Findings

Given the hypothesis stated in Section 1, we would expect that covered code would have lower defect density than uncovered code. Thus the null hypothesis would be that bugs are distributed randomly over covered and uncovered code. We can phrase this null hypothesis as:

Null Hypothesis 1 *The fraction of the total bugs that have their origin in covered code is approximately the same as the fraction of total lines of code that is covered by acceptance tests.*

$$\frac{bugs\ in\ covered\ code}{total\ bugs} \approx \frac{lines\ of\ covered\ code}{total\ lines\ of\ code} \tag{2}$$

In other words, we would expect bugs to be distributed evenly over the codebase, regardless of the degree of test coverage.

There were twenty-one bugs reported, fixed or committed before the date of release 3.2.0 beta-1. We found that five of them were caused by source lines covered by the acceptance tests, twelve by uncovered lines, one had origins in both covered and uncovered lines; the origin of the remaining three could not be determined.

We excluded the three unclassified bugs from our sample. The resulting sample size of bugs is eighteen, among which six bugs were caused by covered code, twelve by uncovered: we classified the one bug from mixed lines of covered and uncovered code into the category of bugs from covered code, because the fault in the covered code should have been exposed by the acceptance tests. Thus, the observed frequency of bugs from covered code is $O_c = 6/18 = 1/3$, while the observed frequency of bugs from uncovered code is $O_u = 12/18 = 2/3$.

Table 1. Observed and expected bugs

	Observed	Expected	Residual O-E
Bugs in uncovered code	12	8.28	3.72
bugs in covered code	6	9.72	-3.72
Total	18		

The upper bound of test coverage, calculated using Formula 1, is 54 percent. This is the expected distribution of covered bugs, that is $E_c = 0.54$, while the expected distribution of bugs from uncovered code is the complement $E_u = 1 - E_c$. According to the null hypothesis, we should expect that 54 percent of the 18 bugs are from covered code, and 46 percent from uncovered code.

To test how closely the observed distribution of bugs matches the expected distribution, we used a Chi-square test (x^2). Table 1 presents the total number of observed and expected bugs in covered and uncovered code, which is the input for the Chi-square test.

x^2 is calculated as follows:

$$x^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{3}$$

where O_i is the i-observed distribution, E_i is the i-expected distribution, and n the number of observations. The calculated Chi-square is 3.095.

The degrees of freedom ν is:

$$\nu = (r - 1) * (c - 1) \tag{4}$$

where r is the number of rows and c is the number of columns in the data table. For the data in Table 1 $\nu = 1$.

The calculated value x^2 must be lower than the distribution value of $x^2(\alpha, \nu)$ in order to accept the null hypothesis. The distribution value of $x^2(\alpha, \nu)$ is searched in a table with the critical values of Chi-square [4]. The table columns are the statistical significance levels, and the rows the degrees of freedom. For a statistical significance $\alpha = 5\%$ and $\nu = 1$, the Chi-square distribution value is $x^2(5, 1) = 3.841$.

The Chi-Square test results are summarized in Table 2.

Table 2. Chi-square test for bug distribution

Calculated Chi-square	x^2	3.095
Statistical significance level	α	0.05
Degrees of freedom	ν	1
Distribution value	$x^2(\alpha, \nu)$	3.841
Calculated Chi-square - Distribution value	$x^2 - x^2(\alpha, \nu)$	-0.7459

5 Discussion

The null hypothesis can be rejected if the calculated x^2 is greater than the x^2 distribution value with a statistical significance $\alpha = 0.05$, and 1 degree of freedom or $x^2(0.05, 1)$. In this case, $x^2 - x^2(\alpha, \nu)$ is equal to -0.7459; therefore, the null hypothesis cannot be rejected. As a result, we cannot conclude that the distribution of bugs is affected by the amount of covered code.

Using the upper bound test coverage (54 percent), we cannot determine if the acceptance tests have a positive or negative effect on the distribution of external defects. Further, we know that the actual test coverage is lower than 54 percent, because many files in the Zope3 distribution are not exercised by acceptance tests.

Our findings have to be considered preliminary due to several limitations that may impact the validity of our study. One limitation is the sample size of the reported bugs, which is very small. Even though the Chi-square test can be applied effectively to small samples, it would be more convincing if we had a significantly larger sample of reported bugs.

Another limitation is that we were not able to obtain an accurate measure of the test coverage of Zope3, and so had to work with the upper bound of test coverage instead. This affected the statistical results and the interpretation of the findings accordingly.

Finally, the way we identified the acceptance tests of Zope3 can also be a potential limitation. We excluded the unit tests from the test suite and considered all the rest, including so-called "integration" tests, as part of the acceptance test suite; thus the upper-bound of test coverage is increased.

6 Conclusions

Our study set out to explore the benefit of automated acceptance testing in terms of the quality of a software product. We have studied Zope3, an open source software project, to reveal the possible link between automated acceptance tests and quality. Our findings cannot support the claim that automated acceptance testing can improve the quality of code. Bearing in mind that the validity of the findings is limited by several factors discussed in the previous section, we cannot reject the claim either. However, though our study is at the early stage, and the results should be considered preliminary, the research design we used in

the study can be considered a useful framework for other researchers to conduct other similar studies.

Several potentially valuable research directions are indicated by our study. One direction is to extend the current study to additional open source and closed source software projects which can increase the generalizability of our findings. Another direction is to investigate automated acceptance testing in more depth, to measure the level of automation using metrics such as coverage and degree of automation, and establish the links between automation level of acceptance testing and code quality. Yet another interesting study could be using the number of bug-affected lines of code, rather than just the number of bugs, as an indicator of code quality. This would allow us to take into consideration the severity of bugs, which has implications to the maintenance of code, and therefore a better indicator of the quality of code.

Acknowledgments

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (`http://www.lero.ie`).

References

1. Bruno C. Araújo, Anne Caroline Rocha, Arthur Xavier, Ana Isabella Muniz, and Francilene P. Garcia. Web-based tool for automatic acceptance test execution and scripting for programmers and customers. In *EATIS '07: Proceedings of the 2007 Euro American conference on Telematics and information systems*, pages 1–4, New York, NY, USA, 2007. ACM.
2. L. Crispin and T House. Testing in the fast lane: Automating acceptance testing in an extreme programming environment. In *Proceeding of XP Universe Conference*, 2001.
3. Prashant Gandhi, Nils C. Haugen, Mike Hill, and Richard Watt. Creating a living specification using fit documents. In *ADC '05: Proceedings of the Agile Development Conference*, pages 253–258, Washington, DC, USA, 2005. IEEE Computer Society.
4. Frederick J Gravetter and Larry B. Wallnau. *Statistics for the Behavioral Sciences*. Wadsworth Publishing, 8 edition, 2008.
5. Geir Kjetil Hanssen and Borge Haugset. Automated acceptance testing using fit. In *Hawaii International Conference on System Sciences*, pages 1–8. IEEE Computer Society, 2009.
6. David Kessler and Timothy J. Andersen. Herding cats: Managing large test suites. *AGILE Conference*, 0:375–380, 2009.
7. Martin Kropp and Wolfgang Schwaiger. Reverse generation and refactoring of fit acceptance tests for legacy code. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 659–664, New York, NY, USA, 2009. ACM.

8. David Martin, John Rooksby, Mark Rouncefield, and Ian Sommerville. 'good' organisational reasons for 'bad' software testing: An ethnographic study of testing in a small software company. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 602–611, Washington, DC, USA, 2007. IEEE Computer Society.
9. R.W. Miller and C.T. Collins. Acceptance testing. In *Proceedings of the XP Universe Conference*, 2001.
10. Shelly S. Park and Frank Maurer. The benefits and challenges of executable acceptance testing. In *APOS '08: Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral*, pages 19–22, New York, NY, USA, 2008. ACM.
11. Kris Read, Grigori Melnik, and Frank Maurer. Examining usage patterns of the fit acceptance testing framework. In Hubert Baumeister, Michele Marchesi, and Mike Holcombe, editors, *XP*, volume 3556 of *Lecture Notes in Computer Science*, pages 127–136. Springer, 2005.
12. Filippo Ricca, Marco Torchiano, Massimiliano Di Penta, Mariano Ceccato, and Paolo Tonella. Using acceptance tests as a support for clarifying requirements: A series of experiments. *Inf. Softw. Technol.*, 51(2):270–283, 2009.
13. R. Owen Rogers. Acceptance testing vs. unit testing: A developer's perspective. In *XP/Agile Universe*, pages 22–31, 2004.
14. Bojan Tomić and Siniša Vlajić. Functional testing for students: a practical approach. *SIGCSE Bull.*, 40(4):58–62, 2008.
15. Richard J. Watt and David Leigh-Fellows. Acceptance test driven planning (experience paper). In Carmen Zannier, Hakan Erdogmus, and Lowell Lindstrom, editors, *XP/Agile Universe*, volume 3134 of *Lecture Notes in Computer Science*, pages 43–49. Springer, 2004.