Quality Improvement with Testing

Jiayi Zheng

jiayi.zheng@student.uni-tuebingen.de

Abstract. Testing is the most frequently used method to improve software quality and in many cases it is the only method in software-quality program. This paper gives answers to the following questions from a developer's point of view: "Why should we test?", "What do we possibly do wrong?" and "How should we test effectively?"

Introduction

Testing is a complex topic which by itself forms a discipline alongside other software development activities. Unlike a software engineer or software developer, a test manager or test engineer deals exclusively with testing activities during the process of software development [AO, pp.4-5]. This paper however focuses mainly on testing from a developer's point of view.
The paper contains three sections. The first one **Testing in the Context of Software Quality** gives an overview about the role of testing in improving software quality and discusses about the effectiveness of testing. The second one **Problems with Testing** shows common behaviors and practices of developers which damage the effectiveness of testing. The last one **Strategies for Effective Testing** contains a summary of selected tips and suggestions given by Steven McConnel, Andrew Hunt and David Thomas, whose books *Code Complete* and *Pragmatic Programmer* serve as main references for the paper.
The purpose of this paper is to answer the following three questions:

1. Why should we test?
2. What do we possibly do wrong?
3. How should we test effectively?

Testing in the Context of Software Quality

It is difficult to explain the necessity of testing without talking about software quality first. The quality of a software can be defined by different characteristics. McConnel categorizes those characteristics in internal quality characteristics and external quality characteristics [MC, Sec.20.1]. External characteristics are characteristics of the software that a user cares about, such as correctness or usability [1]. Internal characteristics are those of the software that a developer cares about, such as maintainability or readability [2]. To improve the quality of a software means to maximize a set of those characteristics so that the product eventually meets the expectation of both developers and users [MC, Sec.20.2].
Testing is just one among many techniques for improving software quality. In general, it helps developers in detecting errors in software and provides estimations of certain quality characteristics mentioned above. Computer scientists and test engineers traditionally use two different systems to categorize software testing activities[AO, p.5]. One system is based on software development activity represented by the so called "V Model" with Acceptance Testing corresponding to the Requirement stage of software development, System Testing to the Architecture stage, Integration Testing to the Design stage and Unit Testing to the Implementation stage [AO, pp.5-7]. The other system uses the test process

[1] McConnel listed 8 external characteristics: correctness; usability; efficiency; reliability; integrity; adaptability; accuracy; robustness [MC, Sec.20.1].
[2] McConnel listed 7 internal characteristics: maintainability, flexibility, portability, reusability, readability, testability, understandability.

maturity levels with each level representing one certain attitude of the testers. Five different levels are described in [AO, pp.8-10]:

Level 0 Testing is the same as debugging.
Level 1 The purpose of testing is to show correctness.
Level 2 The purpose of testing is to show failures.
Level 3 Testing can show presence, but not the absence, of failures. Therefore the purpose of testing is to reduce the risk of using the software.
Level 4 Testing is a mental discipline that increases software quality.

To summarise, those kinds of categorization indicate that the testing process is an ongoing process which may contain different strategies and techniques depending on the different stages of software development, and also the attitude or thinking of the testers can have an impact on it.

Many software projects rely on testing as the primary method of both quality assessment and quality improvement [MC, Sec.20.2]. However testing is not the most efficient technique for defect removal and by no means provides sufficient quality assurance performed alone. As Jones states in [J, p.7], though testing has been the primary software defect removal method for more than 50 years, most forms of testing are only about 35% efficient or find only one bug out of three. The evaluation of efficiency results from taking the percentage of defects, detected by the used method, out of the total number of defects exist in the software. Statistics of Defect-Detection Rates in [MC, Sec.20.3] show that none of the tested defect removal techniques achieved an efficiency above 75% performed alone. For developers, the most common kinds of strategies: unit testing and integration testing each are only 30% and 35% efficient. Even the combination of those testing strategies often is less than 60% efficient [MC, Sec.22.1]. Moreover, testing is an expensive, labor intensive practice, as stated in [AO, p.10]. It requires up to 50% of software development costs and even more for safety-critical applications. In comparison, other techniques like collaborative development practices in their various forms can perform better and cost less [MC, Sec.22.1].

Nevertheless, for developers testing is still a beneficial practice. Developer testing can assess the reliability of the software under construction. The test results describe, how reliable the software is, in the current stage of development and guide to bug fixes. With Unit Testing developers can make debugging job easier by writing unit tests for each routine. In this way, if the program breaks they can easily find out which routine causes the defect. Finally, if the errors and test results are recorded over time, they can reveal common errors, as well as error-prone components or routines of the software, so that the developers can prevent this kind of problems in the future [MC, Sec.22.1].
The benefits of testing listed above surely do not compensate its deficiency in error detection compared to other techniques, but part of the problem with the

effectiveness of testing is caused by the testers or developers themselves [MC, Sec.22.1].

Problems with Testing

For developers, the testing activity doesn't fit into the natural course of software development which is to build and keep the software from breaking, whereas testing causes the software to break [MC, Sec.22.1]. So it is understandable that most developers hate testing, they tend only to test the part of the program that works and subconsciously avoid the weak spots [HT, Sec.43, p.1]. Tests for whether the code works are so called "clean tests". Their counterparts — "dirty tests" are tests for all the ways the code breaks [MC, Sec.22.2].
The preference of the developers for "clean tests" is one issue that affects the effectiveness of testing. McConnel mentioned an experiment carried out by Glenford Myers where a group of experienced programmers had to test a program with 15 known defects [MC, Sec.22.1]. The average defect-detection rate of the group is 30% or 5 out of 15 errors. None of them found more than 9 errors. This experiment shows that the effectiveness of testing depends on the willingness of the developers to find errors. With the assumption that the program is correct and the aversion to "dirty tests", developers will likely overlook errors in the program and thus affect the effectiveness of testing.

An other problem with testing is that developers tend to have an optimistic view about test coverage. The statistics show that the coverage rate the developers believe to be achieving (95%) is much higher than the coverage rate they are truly achieving (50 - 60% in average) [MC, Sec.22.2]. Moreover, most developers consider the achievement of 100% of "statement coverage" as sufficient, due to the fact that a lot of coverage analysis tools on the market apply this kind of test coverage criterion [HT, Sec.43, p.7]. But in fact, "statement coverage" is hardly sufficient, a better coverage criterion to meet would be "branch coverage". The following example with the corresponding graph demonstrates the difference between "statement coverage" and "branch coverage":

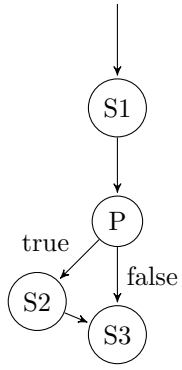
```
Statement1;

if (Predicate) {

  Statement2;

}

Statement3;
```



testcase1 = [S1, P, S2, S3]
testcase2 = [S1, P, S3]

Let T_i be all possible sets of testcases for this example. $T_1 = \{testcase1\}$ satisfies statement coverage and $T_2 = \{testcase1, testcase2\}$ satisfies branch coverage. In other words, "100% statement coverage" requires that every statement, which is every node in the graph, is executed at least once, whereas "100% branch coverage" requires that every path the code may take, which is every edge in the graph, is executed at least once. Since "branch coverage" subsumes "statement coverage" [AO, p.20], testing process aiming for "100% branch coverage" can discover more errors than that aiming for "100% statement coverage".

There are other factors besides those described above, which can affect the effectiveness of testing. Jones for example, named bad test cases with defects in them as one of the major defect origins in the U.S. software industry. About 6% of test cases have bugs of their own [J, pp.2-3]. And Amman and Ouffut pointed out in [AO, p.225], that many organizations tend to postpone all testing activities to the end of the implementation or after the implementation has started. But the later in the process the fewer resources remain for the testers or developers to thoroughly plan and design tests. As a consequence, the testing process becomes compressed and insufficient. Overall, the effectiveness of testing depends heavily on the attitude and the behavior of the developers or testers themselves.

Strategies for Effective Testing

This section presents some strategies for developers to perform testing effectively.

1. **Hope to Find Bugs**
   As explained in the previous sections, the effectiveness of testing depends on the developers' willingness to find errors. Allthough, it might appear to be an unnatural act, but developers should hope to find bugs in their software. The message in both books is similar: "We are *driven* to find our bugs *now*, so we don't have to endure the shame of others finding our bugs later." in [HT, Sec.43, p.1] and "[...], but you should hope that it's you who finds the errors and not someone else." in [MC, Sec.22.1].
2. **Aim for Better Test Coverage**
   Many coverage analysis tools on the market keep track of which lines of codes or statements in a program have been executed and which have not. This kind of test coverage is hardly sufficient for detecting all possible errors in the program. Instead of aiming for "100% statement coverage", McConnel suggested to achieve "100% branch coverage" by testing, where every path of the program is executed at least once. Hunt and Thomas, on the other hand, suggested to use the "state coverage" where the number of states that

the program may have is the concern.
As an example, for the following function with two integer input parameters, each of which can be a number from 0 to 999, there are 1,000,000 logical states [3]. One state from those causes the divide-by-zero error where a = 0 and b = 0 [HT, Sec.43, p.8].

```
int test(int a, int b) {
  return a / (a + b);
}
```

In general, it is impossible to have 100% test coverage of a program, even with good coverage criteria, there are still areas where errors could possibly be and are hard to be detected. Nevertheless, effective use and good choice of coverage criteria make developers and testers more likely to find bugs in the program and thus increase the effectiveness of testing.

3. **Test as Early as Possible**
   As Thomas and Hunt recommended in [HT, Sec.43, p.1], developers should start testing as soon as they have code, because the earlier a bug is found, the cheaper it is to fix it. Statistics about the increase of defect-cost over the stages of software development in [MC, Sec.3.1] show that the longer the defect stays in the software development process, the more damage it causes and the higher the cost will be to fix it. For example, for the following workflow: requirements, architecture, construction, system test and post-release, a defect inserted in the architecture stage of development costs $1000 to fix during the architecture stage. But for its removal in the system test stage, the cost will increase to $15,000.
   One good approach for testing as early as possible is test-first programming or test-driven development(TDD). By writing testcases first problems or errors in the software would be exposed ealier and developers tend to produce better code as they have to think or rethink about requirements and design before coding.
4. **Test Often and Automatically**
   When changes are made to a software or components of the software, it is necessary to rerun tests that have been successfully completed in the past on regular basis, to make sure that the changes did not insert any new defects. This kind of tests are called regression tests which needed to be performed automatically. The idea behind regression testing is that small changes to one part of a program often cause problems in some other distant parts of the program. To find this kind of problem, developers should systematically retest the program using automated testing tools [AO, p.215].
   One common usecase for regression testing is to add one regression test for each bug or problem reported by human tester and rerun it each time changes have been made to the program [AO, p.216]. Thomas and Hunt also

[3] This is where techniques like Equivalence Partitioning and Boundary Analysis come in handy for reducing the amount of test cases by picking meaningful states to test. More about Equivalence Partitioning and Boundary Analysis in [MC, Sec.22.3].

supported this approach, they state that it is very likely that bugs once found would appear again sometime later and it is a waste of time to chase after known bugs. Moreover, customers or users are more willing to be saddled with new problems than with the same problem over and over [AO, p.216], so it is also a beneficial practice from a usability perspective.

5. **Testing the Tests**
   As mentioned in the previous section, bad testcases with errors in them also represent a major origin of defects. Hunt and Thomas suggested that after writing a test to detect a particular bug, developers should cause the bug deliberately to make sure that the test actually works as intended. A more sophiscated version of this approach requires a saboteur who is responsible for introducing bugs on purpose to test whether the testcases will catch them [HT, Sec.43, p.7].

Conclusion

In conclusion, we can now answer the three questions introduced in the beginning of this paper:

1. Why should we test?
   Because we care about software quality. Though, it is hard to achieve high quality with testing alone, as developer, we can still benefit from the testing practice and at least improve our code.
2. What do we possibly do wrong?
   We usually are not willing to find bugs and are too optimistic about the test coverage.
3. How do we test effectively?
   We should hope to find bugs.
   We should aim for better test coverage by choosing for example branch coverage or state coverage over statement coverage.
   We should test as early as possible, because in the end it will turn out to be cheaper for us and writing tests first is a beneficial practice.
   We should test often and automatically, use regression testing for finding bugs only once.
   We also should test the testcases to make sure that they work as intended.

References

[MC] McConnel, S.: Code Complete, Second Edition. Microsoft Press, 2004

[HT] Hunt, A., Thomas, D.: The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, 1999

[AO] Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, 2008

[J] Jones, C: Software Defect Origins and Removal Methods, 2012. published online `http://www.ifpug.org/Documents/Jones-SoftwareDefectOriginsAndRemovalMethodsDraft5.pdf`