Volume: 4, Issue: 5
Page:91-97
2020

ISSN 2520-4750 (Online) & ISSN 2521-3040 (Print)

International Journal of Science and Business

Properties of Good Unit Tests for Software Quality Assurance

Mustafa Nizamul Aziz

Abstract:
Software Quality is a vital element in Software development, therefore Software Engineering researchers and practitioners strive to show the importance of it. It is obvious to find the best tools to ensure Software Quality. One of the tools that prove the Software Quality is Unit Testing; which has shown its power to assess the Quality of Software. Good Unit tests must have to follow some certain attributes. A well- known working plan is 'A-TRIP' through which all the inevitable properties of good Unit Tests are checked. The purpose of this paper is to give words to A-TRIP and to show its practical impact on Software Quality Assurance. Documents analysis was used here as the data collection method to develop Research Questions (RQ). Qualitative method was used as the data analysis method in the research process to answer the RQs. The paper figured out the Cost of Tests as well as made arguments about each A-TRIP property and its implications in terms of costs in the software process.

IJSB
Accepted 07 June 2020
Published 08 June 2020
DOI: 10.5281/zenodo.3884317

Keywords: Unit Testing, A-TRIP, Software Quality, Software Development, Software testing

About Author (s)

Mustafa Nizamul Aziz, Senior Lecturer, East West University, Dhaka, Bangladesh.

1. Introduction

Unit Test is described (Hunt, 2003) as a piece of code written by a developer that exercises a very small, specific area of functionality in the code being tested. Usually a unit test exercises some particular method in a particular context. For example, one might add a large value to a sorted list, and then confirm that this value appears at the end of the list; or one might delete a pattern of characters from a string and then confirm that they are gone. Unit tests are performed to prove that a piece of code does what the developer thinks it should do.

"*Unit Testing Is the execution of a complete class, routine, or small program that has been written by a single programmer or team of programmers, which is tested in isolation from the more complete system"* (McConnell, 2004). It is good to mention and remind that Unit Testing itself will not improve the Software Quality directly rather than proving and assuring some characteristics which assess the Software Quality. In order to consider Unit Testing as 'good', it should satisfy some conditions (Hunt and Thomas, 2004). It should test the six specific areas of code (Right-BICEP); and as Hunt mentioned (2003) that a 'good' Unit Testing should have certain properties (A-TRIP) and the tests should cover correct boundary conditions. Software Quality is described by McConnell (2004) as it is an important part of programming that saves time and which is cost-effective; Software has both external and internal quality characteristics. The General Principle of Software Quality is that improving quality reduces development costs. The most obvious method of shortening a development schedule is to improve the quality of the product and decrease the amount of time spent debugging and reworking the software. In the next sections, it will be discussed how Unit Testing affects the Software Quality and next which properties are suitable for better Unit Testing. Research Questions of this study are: (1) What are the main Properties of Good Unit Tests and how Unit tests affect in Software Quality Assurance?, and (2) What is the essence of A-TRIP in Software Quality Assurance?

2. Unit tests for Software Quality Assurance

There are many techniques for software quality assurance. According to McConnell (2004), one of these good techniques is trying to find the program defects as early as possible during development time to ensure the software quality. Unit Testing has shown and proven as a vital element for Software Quality assurance. Execution testing can provide a detailed assessment of a product's reliability. Developers on many projects rely on testing as the primary method of both quality assessment and quality improvement. Testing does have a role in the construction of high-quality software, however, and part of quality assurance is developing a test strategy in conjunction with the product requirements, the architecture, and the design. Gao and Tsao have mentioned (2003) the importance of Unit Testing. Widespread reuse of a software component with poor quality may wreak havoc. Improper reuse of software components of good quality may also be disastrous. Testing and quality assurance is therefore critical for both software components and component-based software systems. Several recent books address the overall process of component-based software engineering or specific methods for requirements engineering, design, and evaluation of software components or component-based software. Not much resource was found focusing on testing, validation, certification, or quality assurance in general for reusable software components and component-based software systems. According to Kerry and Delgado (2009), testing has to be seen in a broader perspective of 'maximizing' customer satisfaction and providing feedback for process refinement, rather than just detecting and correcting errors in the software. According to McConnell (2004), Unit Testing is a powerful element to improve

Software Quality; since it can find up to 50% of the software defects during the development time which causes coast- and time effective and these consider as a vital element that affect the quality of software according to what Jones mentioned (2000) "Software projects with the lowest levels of defects had the shortest development schedules and the highest development productivity. The software defect removal is the most expensive and time-consuming form of work for software". Also in his book it was shown how Unit Tests can improve the Software Quality indirectly through proving some internal and external quality characteristics like 'Correctness, Usability, Reliability, Accuracy, Robustness, Maintainability, Reusability, Readability, Testability, Understandability, and finally, the most valuable Reliability, each one of these characteristics play some role to improve the Software Quality positively for some extent, so by proven these elements the software quality will be enhanced and assured.

3. A-TRIP

Good tests have the following properties, which makes them A-TRIP: **A**utomatic, **T**horough, **R**epeatable, **I**ndependent, **P**rofessional (Hunt et. al., 2007, p. 117).

3.1 Automatic

should run without human intervention

Unit tests need to be run automatically. Invoking one more unit tests can't be any more complicated than pressing one button in the IDE or typing in one command at the prompt. We should not introduce a test that breaks the automatic model by requiring manual steps. A machine should be running all of the unit tests for all checked-in code continuously. This automatic check ensures that whatever is checked-in hasn't broken any test, anywhere. If tests are written poorly, either they won't help find defects or we'll spend too much time maintaining our tests as opposed to developing. Test must determine for itself whether it passed or failed. It means we don't have to think about it. No one needs to read the test results to check whether the code is working or not. We can even write Test Suites. A Test Suite is like a collection of Test Cases we run at the same time. In JUnit, any class can be used as a test suite. One of the big advantages of automated unit testing is the ability to safely modify existing code. Bad unit tests turn this advantage on its head. A unit test should only test a small piece of functionality. A litmus test is to ask us if any part of the unit test could stand alone in a separate unit test. Another feedback loop for our unit testing quality is the amount of time we spend with the debugger. If our unit tests are coarse, a test failure is more difficult to find. If the unit test exercises a small amount of code, the test failure cause can usually be spotted very quickly.

3.2 Thorough

should test every aspect of the system

The more unit-tested the code, the fewer problems it will have. Good unit tests are thorough; they test everything that's likely to break. These include every line of code, branch, exception or most likely candidates for error like boundary conditions, missing and malformed data, and so on. It's a question of judgment, based on the needs of project (time, effort, and budget). Although bugs tend to cluster around certain regions in the code, we have to ensure that we have checked all key paths and scenarios. Good tests test as much as is possible; the more we test the more defects we might find. How much our tests test is described as *coverage* and coverage can be broken down into the following three dimensions:

- **Code coverage**: Are all execution paths and methods tested? While it's hard to determine if all code paths are executed, we absolutely must have at least one test for every public method. Whether we need tests for non-public methods is debated.

- **Scenario coverage**: Are all possible situations covered? This is where CORRECT boundary conditions become important.
- **Specification coverage**: Are all the requirements covered? If our requirements are good, they'll all have clearly defined pass criteria, which we can test.

Tools are available practically which help us determine how much coverage we have achieved.

3.3 Repeatable

should lead to the same results no matter how often we run them

Just as every test should be independent of every other test; they must be independent of the environment as well. The goal remains that every test should be able to run over and over again, in any order, and produce the same results. This means that tests cannot rely on anything in the external environment. That includes obvious external entities such as databases, system time, network conditions, etc. Each test should produce the same results every time. If it doesn't, then that should tell us that there's a real bug in the code. Unit Tests should not require modification every time they run. How easily can we set up an environment to run the tests? In case, the answer is "not very," we may never get the tests running. As much as possible, we should decouple unit tests from outside dependencies, i.e. databases and web services through mocks or stubs. When we do test against an outside dependency, the dependency better be set up right by the automated build. We will not run the test just once. For the sake of continuous feedback, successful continuous integration, we have to make sure that our tests run quickly.

3.4 Independent

should be independent from the environment and each other, should test only one thing at a time

Tests need to be kept neat and tidy, which means keeping them tightly focused and independent from the environment. We have to make sure that we are only testing one thing at a time. It doesn't mean that we use only one assert in a test, but that one test method should focus on a method, or a small set of production methods that, together, provides some feature. 'Independent' also means that no test relies on any other tests. We should be able to run any individual test at any time, and in any order. We don't want to have to rely on any other test having run first. Using per-test and per-class setup/teardown methods is better to get a fresh start. To say tests should be independent is to say they should be small. It must not be tested everything all at once. If we have several methods that equate to one feature which needs testing, we have to make sure to test those methods separately. The situation that too much is happening at the same time and there is no way to determine exactly which method is the source of the defect is a common major problem. There shouldn't be an order dependency, intended or not, between unit tests. Problems appear when a unit test leaves some kind of bad data around. A unit test is supposed to be a valuable form of documentation. At best, a unit test should explain the intended use and function of a class. At worst cases, the unit test should still be easy to debug in the time of a regression failure. Excessive data setup can complicate a unit test beyond any hope of comprehension.

3.5 Professional

*should be written and maintained to the same professional standards as our production cod*e

The test code must be written and maintained to the same professional standards as production code. All the usual rules of good design—maintaining encapsulation, honoring the DRY principle, lowering coupling, etc.—must be followed in test code just as in production code. It's easy to fall into the trap of writing very linear test code; that is, code that just plods

along doing the same thing over and over again, using the same lines of code over and over again, with nary a function or object in sight. That's a bad thing. Test code must be written in the same manner as real code. That means we need to pull out common, repeated bits of code and put that functionality in a method instead, so it can be called from several different places. Troubleshooting unit tests and understanding the unit test should be easier if all the pieces of the test data is visible in the same file. We should avoid redundant code and not to copy-paste the same code from one test method to the next one. It's a bad practice. Only we will write tests for relevant, non-trivial code; we're not writing tests just for the sake of writing tests, but we need them as a means by which to find defects. For example, we will not waste our time writing unit tests for trivial methods such as Getters and Setters; our other tests will be using those anyway and if they're broken we should notice very quickly.

4. Cost of Tests

In accordance with Hunt (2004), unit tests are very powerful magic, and if they used badly they can cause an enormous amount of damage to a project by wasting time. If unit tests aren't written and implemented properly, testers can easily waste so much time maintaining and debugging the tests themselves that the production code and the whole project suffers. Time is money. The faster the tests are, the more productive are the programmers. For all the monetary observations, it is important to focus on long term cost reduction.

4.1 Automatic

Unit tests need to be run automatically in at least two ways: invoking the tests and checking the results. Humans aren't very good at those repetitive tasks. Programmers make mistakes in the testing and checking, and waste time investigating a bug that may not exist, or not catch a new bug that will go on to cause additional damage. Another reason for automating test is that computing time is cheap, so manual testing is expensive. That was different forty years ago so manual tests were cheaper and the better choice back then. Now it's the other way round for most cases.

4.2 Thorough

Poor test coverage leads to bugs in production. Those are way more expensive than good test coverage. It is important to realize that bugs are not evenly distributed throughout the source code. Instead, they tend to clump together in problematic areas; so the more unit-tested the code, the fewer problems it contains.

4.3 Repeatable

Without repeatability testers and programmers might be in for some surprises at the worst possible moments test. What's worse, these sorts of surprises are usually bogus- it is not really a bug, it is just a problem with test. One can't afford to waste time chasing down phantom problems. A test that is not repeatable costs money but doesn't have any business value.

4.4 Independent

At any rate, Programmers want to achieve a traceable correspondence between potential bugs and test code. In other words, when a test fails, it should be obvious where in the code the underlying bug exists without looking at the test code itself. The name of the test should tell us all we need to know. Otherwise, we've got to go hunting for it, and that will just waste our time. Therefore, unit testing should be independent.

4.5 Professional

Because the written code for unit test is real, some may argue that it's even more real than the code developer ships to customers. And because poor code quality slows down developers,

it's cheaper to invest in professional tests than having programmers wasting time analyzing poor code.

5. Findings and Conclusion

As Unit testing has been mentioned as one way to check the reliability of Software, and good Unit tests have been considered as effective tools of SQA, proper use of Unit testing assists developing software comfortably and correctly. After conducting a literature review on this field, it can be wrapped up that:

- A Unit-Test should produce a deterministic result.
- A Unit-Test should be independent and valid.
- Tests should reduce risk.
- Tests should be easy to run.
- Tests should be easy to maintain.
- Tests should only fail because of one reason. Tests should only test one thing, avoid multiple asserts for example.
- There should only be one test that fails for that reason. This keeps test base maintainable.
- Minimizing test dependencies; no dependencies on databases, files, and user interface, etc.

This paper has mainly concentrated on the properties of A-TRIP and the motivation of this study was to find out the practical effect of A-TRIP on ensuring Software Quality. Attributes of A-TRIP have been discussed and the essence of these attributes has been mentioned in the paper. Software Quality and its importance in systems development have also been described.

5.1 Future works

To make a real-time comparison of A-TRIP with other such kinds of structures to assess A-TRIP can be interesting future research topics.

References

Armour, P. (2011). Testing: failing to succeed. *Communications of the ACM*, 54(10):30-31.

Berndtsson, M., Hansson, J., Olsson, B., Lundell, B. (2008) *Thesis Project*, Springer.

Freeman, E., Freeman, E., Sierra, K., and Bates, B. (2004). *Head First design patterns*. O'Reilly Media.

Gao, Jerry., Tsao, H.-S. J. and Wu, Ye. (2003). *Testing and quality assurance for component-based software.* Boston: Artech House.

Gregor & Jones (2007) The anatomy of design theory. *Journal of the Association for Information Systems,* 8(5) 2007, 312-335.

Hevner, A., March, S., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, 28(1):75-105.

Hunt, Andy and Thomas, David (2003). *Pragmatic unit testing: in C# with NUnit*. Raleigh: The pragmatic programmers. p.3.

Hunt, Andy and Thomas, David (2003). *Pragmatic unit testing: in C# with NUnit*. Raleigh: The pragmatic programmers. p. 117.

Hunt, A. and Thomas, D. (2004). Three essential tools for stable development©. CrossTalk: *The Journal of Defense Software Engineering*, 17(11):22-25.

Jones, Capers (2000). *Software assessments, benchmarks, and best practices*. Boston, Mass.: Addison-Wesley.

Kerry, E.; Delgado, S.; (2009) "Applying software engineering practices to produce reliable, high-quality and accurate automated test systems," *AUTOTESTCON*, IEEE, vol. no. 69-71, pp.14-17.

McConnell, Steve (2004). *Code complete: [a practical handbook of software construction]*. 2. ed. Redmond, Wash.: Microsoft Press. Chapter 22, p. 1.

Robson, C. (2011). *Real World Research,* A Resource for Users of Social Research Methods in Applied Settings, 3rd ed., Blackwell Publishing.

Sjöström, J. and Ågerfalk, P. J. (2009) *An Analytic Framework for Design Oriented Research Concepts*. In Proceedings of the15th Americas Conference on Information Systems (AMCIS 2009), pages 302–310. San Francisco, CA, USA, August 2009.

Sørensen, C. (2002). *This is not an article-just some food for thoughts on how to write one*.

Cite this article:

Mustafa Nizamul Aziz (2020). Properties of Good Unit Tests for Software Quality Assurance. ***International Journal of Science and Business,*** *4*(5), 91-97. doi: https://doi.org/10.5281/zenodo.3884317

Retrieved from http://ijsab.com/wp-content/uploads/533.pdf

Published by

IJSAB International