UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

University of Limerick

CS4157 Software Quality

The Role & Importance of Testing in Facilitating the Development of Quality Software

Niall Dillane

CSIS

Submitted to

Dr. Valentine Casey
CSIS

2 May 2020

Abstract

Testing is an often neglected stage of the software development process, garnering less praise and compensation for its workers. Throughout this paper, we will research and demonstrate the various aspects of testing, as well as its importance in developing high quality systems.

Contents

1 Introduction

Software Testing is the investigation of software for the purpose of finding defects and providing feedback on the quality of the product (Kaner 2006). This typically involves running the program — either manually or automatically with some kind of script — and providing input with the intent of finding defects (colloquially known as bugs), or rather proving that there are no bugs. Of course, it is impossible to prove that a piece of software is entirely bug free (Pan 1999), at least if it has any degree of complexity, but testing aims to provide a reasonably complete picture of the state of the software and an avenue to reducing defects.

Testing has become more prominent in recent years (Tuteja et al. 2012), but I believe it still does not get the credit it deserves. Testing is a very broad area, which is key at each stage of the software development process: from specification, to implementation, to maintenance. As it consumes 40-50% of development efforts (Tuteja et al. 2012), it warrants significant consideration and planning from the beginning, so as to minimise costs and time needed later on.

2 Dynamic Analysis in Verification and Validation

There are two different types of analysis in software testing: those are static and dynamic analysis.

Static analysis simply involves analysing a piece of software without actually running it (Wichmann et al. 1995). This can entail human review of code in a text editor, tools used to provide statistics e.g. lines of code per file, or the use of a compiler. This can be useful in certain circumstances, such as finding security vulnerabilities in a codebase (Livshits 2006) and is naturally quicker than walking through a program, but generally it is limited from a "user experience" point of view.

Dynamic analysis is based on system execution, often using tools to automate the process (Ghahrai 2018). There are various stages, from individual unit tests of small portions of the system, to integrating these units and the system overall. Dynamic testing provides a more complete picture of the state of the system, and is a more "real life use" approach. This can be time consuming, and so more and more automation is sought out, but one must be wary of the false sense of security these tools can provide.

Verification and Validation are related concepts in software testing. In whole, they describe the process of: identifying if the system has satisfied the specification provided (verification); and determining if that specification is what the customer really wants or needs (validation). In short, verification answers the question: "Are we building the system right?", and validation answers "Are we building the right system?" (Boehm 1984).

Dynamic verification involves, as outlined previously, executing the code. At this point, developers wish to identify if the system has been built to specification, conducting a review of the program in various ways. These can be testing in the small — checking if inidividual functions perform as expected — up to testing the system as a whole, by executing modules and checking that they interact as expected. Non-functional requirements (discussed more in Section 3) may also be tested, such as stress-testing a server to see how many users it can handle.

Dynamic validation is an even higher level of testing (acceptance testing), wherein the development team and external stakeholders walk through the product and use it as a user would. This is black box testing (see Section 6), as the testers are performing these actions without looking at the internal workings of the system. A purely surface view.

In both verification and validation, dynamic analysis is very important, as it provides real-world feedback on the system and any defects found. Runtime execution is very different from compilation, so I believe static analysis alone is not enough in verifying that the system is built correctly. This is even more critical in the case of validation, since it is recommended to have external stakeholders take part in the testing. These may be non-technical people, so simply showing a series of code metrics will likely be insufficient. Much more preferable is having a dynamic experience, acting out pre-defined user stories and ensuring that the customer is satisfied with the experience.

Some have argued in favour of pure static analysis, pointing out that dynamic analysis is unreliable and cannot ensure complete coverage (Zhioua et al. 2014). However, even here the arguments are generally limited to certain aspects of the software, such as security.

3 Functional and Non-Functional Requirements

Functional requirements define behaviour that a system must include, some kind of function that takes input and/or provides output (Stellman and Greene 2005). These are generally found in use cases, e.g. "a user logs in to their profile". These are definite pieces of functionality that the system will fulfil, which may perform calculations, communicate with other services or utilise data.

Non-functional requirements (NFRs) place constraints on the development of the system and its functionality, and may include: performance, reliability, security, etc, (Stellman and Greene 2005). These are not exactly behaviours that the system must allow, but rather quality attributes which define how the system exhibits these behaviours, e.g. "90% of pages must load within 1 second".

3.1 How to Test

Functional testing is a black-box (Section 6) approach to testing, wherein functions are passed input and have their output verified for correctness (Kaner et al. 1999). This is not the same as input testing, where we test a single method of the program, but rather a function of the system which could interact with many different parts. For example, in the case of testing "login" functionality, the input data prepared may be a username and password, while the output expected may be a success message and redirection. These are prepared, the test carried out and then verified, either manually or with an automated tool.

Non-functional testing is manifold, with different techniques required for performance, security, portability, etc. Each of these aspects will have their own methods, and are selected according to the NFRs of the specific system. Within performance, a common NFR is load testing: determining a system's performance and behaviour under normal circumstances as well as peak load (Eriksson 2019). Generally, there are tools available for testing these different aspects, but other more abstract NFRs, like usability, may require manual testing and surveying.

4 Software Testing Process

The software testing process can vary depending on the organisation and the software development lifecycle they employ. The latest developments in this space are Agile[1] and Extreme Programming[2], which often utilise Test Driven Development in writing tests first, then the code required to pass them (Alliance 2005). These tests are added to as the system develops, as well as additional code to satisfy them, hence the term "continuous integration".

In more traditional software development models, such as the Waterfall model[3] testing is carried out at various stages:

- Requirements are analysed and testable aspects are identified
- Tests are planned and scripts developed if necessary
- Tests are executed and results logged. Errors are reported to the development team
- Metrics and reports are generated on the state of the software, whether it is ready for release etc.
- Regression testing is carried out to check if defects have been fixed, or if new features pass the tests

(Pan 1999)

Note also that there are different levels of testing, as touched on previously.

[1] https://agilemanifesto.org/

[2] http://www.extremeprogramming.org/

[3] https://en.wikipedia.org/wiki/Waterfall_model

4.1 Unit Testing

Unit testing is the lowest level of testing, taking the smallest testable piece of an application which can be run. This is usually a single function (method), which is passed input and an associated expected output, or possibly an entire program (Pearson 2020). A white box approach (see Section 6) is taken, wherein the tester can see the inner workings of the code as it executes. The idea is that by testing each individual portion separately, tests are easier to write, provide more complete coverage, and ensure that every portion will work together.

4.2 Integration Testing

However, simply because units have been tested individually does not guarantee smooth interoperation. Integration testing combines units within one of the programs, or a collection of units, to discover if there are interface defects between them (Pearson 2020). This can also test the efficiency of the relationships, for example, if there is perhaps something slowing down or interrupting communication. Ultimately, it is no use if units work separately but not together.

4.3 Regression Testing

Regression testing is performed when changes have been made to the system, either to fix a bug identified during previous levels of testing, or to add additional functionality. This involves re-running all tests to make sure that the change has not broken the area affected or anything it communicates with (Basu 2015). It is typical for bug-fixing to create other bugs elsewhere in the system, so being able to double check is vital.

4.4 System Testing

Here, the complete application is tested, verifying (see Section 2) that the system has satisfied the specification and all requirements (Pearson 2020). This is typically done in a black-box fashion, by external testers who were not part of development and are not concerned with the inner workings. This is to mimic more of a customer experience.

4.5 Acceptance Testing

In contrast to System testing, Acceptance testing is done for the purpose of validating (see Section 2) the system, determining if it does the right things. These tests are typically written by the customer and represent their interests, assuring them that the system performs correctly (Davis 1985). This is entirely black-box, with less formal documentation, for example: "When I press delete on a friend in my contacts, they should disappear from the list".

5 Test Strategy

A Test Strategy is a plan which defines the approach to the testing lifecycle of a piece of software (*How to create Test Strategy Document (Sample Template)* n.d.). This defines coverage and scope, as well as providing testers with a clear overall picture at all times. This is used in the hope of minimising the possibility of missing any test activity. Essentially, it is a guideline to achieve test objectives set out in the Test Plan (see Section 5.2).

Key principles of an effective test strategy are:

- Scope: The testing activities to be carried out, with their timelines, as well as responsibility for approving this test strategy.
- Test Approach: The levels of testing (e.g. Unit, Integration) and types (e.g. load testing for NFRs), along with the roles and responsibilities of each team member.
- Test Environment: Reflecting who will use the system, identify what the requirements are, e.g. Windows 10 operating system.
- Testing Tools: Which tools will be used, whether open source or if certain licenses are required for commercial tools. Potential for automation is also identified at this stage.

- Release Control: Management of version history and ensuring that tests are executed after all modifications, before release.
- Risk Analysis: Identify, create a plan to mitigate, and provide contingencies for all risks possible.
- Review: This document should be approved by all stakeholders on the development side, and changes noted.

(*How to create Test Strategy Document (Sample Template)* n.d.)

5.1 Test Procedures

Test procedures are formal specifications of test cases to be applied to one or more pieces of a program (Panzl 1976), facilitating rigorous coverage of the system by ensuring that individual portions of the system are tested in isolation. Similar to Unit Testing (see Section 4.1), these are self-contained and often automated, run with each iteration of the code.

Naturally, by having extensive and well-documented test procedures in place, this can lead to massive time savings over the course of developing a piece of software. Rather than manually stepping through code every time is change is made, trying to remember what needs testing, test procedures can be drawn on either to manually check off, or even automated to run in the background. Testing can consume a significant portion of development time and resources with hiring relevant personnel, but this can be mitigated by the use of documented test procedures.

5.2 Test Plan

The test plan is derived from the product description, requirements and test cases. It describes what to test, how to test it, who will perform the test and when (Ghahrai 2019). This is part of the Test Strategy (Section 5), and is vital in coordinating test activities.

Test plans are comprehensive, and include: test items, what will and won't be tested, techniques for testing, tasks part of the testing, pass/fail/suspension criteria, test environment and staff needs, deliverables and responsibilities, and finally the schedule.

These are all relatively intuitive concepts, but ensuring that all of this information is gathered in an organised way, documented from beginning to end, reduces the risk of missing a certain feature or confusion with deadlines and requirements. In short, this keeps all members of the development team on the same page, allowing for traceability of testing requirements as wel as the functional and non-functional requirements they tackle.

6 White Box and Black Box Testing

Software testing techniques are typically broken down into two categories: black box and white box testing. These describe different approaches and levels of depth to the testing process, although they can sometimes cross over and be utilised at the same level of a system.

White box testing (also known as structural testing, or perhaps more aptly, glass box testing) is a technique in which test cases are designed with knowledge of the underlying source code, often the developer themself (Nidhra and Dondeti 2012). This is concerned with the inner workings of the code, with tests typically devised for very small portions (See Section 4.1), with input and expected output, although it is sometimes used at the Integration and System levels.

Useful for examining the syntactic correctness of code, white box testing is most useful for verification of the code, rather than validation (see Section 2). That is, while it can provide wide coverage and confidence that the system is performing according to the relevant requirement, it does not guarantee that the specification is correct, nor that all the requirements have been implemented.

Black box testing, meanwhile, does not require any knowledge of the program and is typically carried out by a tester separate from the development team (Limaye 2009). The tester is only aware of the input and output, based on the requirements and with no concern for how the program works internally. This can be carried out at any level of testing, but is essential in Acceptance testing, where the system is validated to make sure that it fits customer needs.

These two classifications often correspond, and to fully test a system we must ensure that both are utilised (Nidhra and Dondeti 2012). White box testing is perhaps more useful at the Unit level, as part of Test Driven Development and automated to provide sanity checks on the code. However, it is not much use developing perfectly functioning code if it does not perform the actions outlined in the specification or that the customer desires. For this, more manual, black box testing is essential.

7 Conclusion

Throughout this paper, I aimed to research and outline the various aspects of testing, in order to provide a complete picture and walkthrough of the process. By doing so, we wished to demonstrate the importance of the role played by testing in developing quality software. This is naturally a goal of every software company, but testing too often receives little attention or documentation, in favour of rapid development.

While this is understandable in the short-term, I believe I have made the case for the value of investing early in testing, with the long-term benefits it brings. One must consider the techniques most suitable for their product and at which levels of testing they should be utilised, and create comprehensive documentation to ensure traceability of tests with their associated requirements. This documentation also enables much easier and safer automation, something which is appealing from time and cost saving perspectives, but can be dangerous in proving a false sense of security if planned improperly.

I believe that any company with long-term ambitions should place a strong emphasis on testing from an early stage, if they wish to produce quality software.

References

Alliance, A. (2005), ‘Tdd’, *agilealliance. org [Online]. Available: http://guide. agilealliance. org/guide/tdd. html.[Accessed: Apr. 27, 2015]* .

Basu, A. (2015), *Software quality assurance, testing and metrics*, PHI Learning Pvt. Ltd.

Boehm, B. W. (1984), ‘Verifying and validating software requirements and design specifications’, *IEEE software* (1), 75–88.

Davis, F. D. (1985), A technology acceptance model for empirically testing new end-user information systems: Theory and results, PhD thesis, Massachusetts Institute of Technology.

Eriksson, U. (2019), ‘All about load testing, understand the load tests’.
URL: *https://reqtest.com/testing-blog/all-about-load-testing/*

Ghahrai, A. (2018), ‘Static analysis vs dynamic analysis in software testing’, *Testing Excellence* .

Ghahrai, A. (2019), ‘Test strategy and test plan’.
URL: *https://devqa.io/qa/test-strategy-and-test-plan/*

How to create Test Strategy Document (Sample Template) (n.d.).
URL: *https://www.guru99.com/how-to-create-test-strategy-document.html*

Kaner, C. (2006), Exploratory testing, *in* ‘Quality assurance institute worldwide annual software testing conference’.

Kaner, C., Falk, J. and Nguyen, H. Q. (1999), *Testing computer software*, John Wiley & Sons.

Limaye, M. G. (2009), *Software testing*, Tata McGraw-Hill Education.

Livshits, B. (2006), *Improving software security with precise static and runtime analysis*, Vol. 67.

Nidhra, S. and Dondeti, J. (2012), ‘Black box and white box testing techniques-a literature review', *International Journal of Embedded Systems and Applications (IJESA)* **2**(2), 29–50.

Pan, J. (1999), 'Software testing', *Dependable Embedded Systems* **5**, 2006.

Panzl, D. J. (1976), Test procedures: A new approach to software verification, *in* 'Proceedings of the 2nd international conference on Software engineering', IEEE Computer Society Press, pp. 477–485.

Pearson, L. (2020), 'The four levels of software testing'.
URL: *https://www.seguetech.com/the-four-levels-of-software-testing/*

Stellman, A. and Greene, J. (2005), *Applied software project management*, " O'Reilly Media, Inc.".

Tuteja, M., Dubey, G. et al. (2012), 'A research study on importance of testing and quality assurance in software development life cycle (sdlc) models', *International Journal of Soft Computing and Engineering (IJSCE)* **2**(3), 251–257.

Wichmann, B., Canning, A., Clutterbuck, D., Winsborrow, L., Ward, N. and Marsh, D. (1995), 'Industrial perspective on static analysis', *Software Engineering Journal* **10**(2), 69–75.

Zhioua, Z., Short, S. and Roudier, Y. (2014), 'Towards the verification and validation of software security properties using static code analysis', *International Journal of Computer Science: Theory and Application* **2**.