UNIVERSITY OF WATERLOO

Faculty of Mathematics

ACHIEVING QUALITY THROUGH AUTOMATED SOFTWARE TESTING

Ultimate Software

San Francisco, California

Prepared by

Nicholas Westbury

4A Computer Science

ID 20550430

August 24, 2018

MEMORANDUM

To:	Brian B.
From:	Nicholas Westbury
Date:	August 24, 2018
Re:	Work Report: Achieving Quality Through Automated Software Testing

I have prepared the enclosed report, "Achieving Quality through Automated Software Testing" for my fourth and final work report. The report is required by the Co-operative Education Program as part of Co-op degree requirements.

As you know, I worked as part of the software development team for Perception, a survey tool that allows employers to get feedback and insights from algorithms that analyze the employee survey data using a mix of standard statistical and machine-learning methods. My primary duty this work term was to build features and tests for the Perception web application. As the product matures, focus is shifting from feature to stability work. Part of this effort is increased testing, forming the basis for this report.

As part of this process, the Faculty of Mathematics requests that you evaluate this report for command of topic and technical content/analysis. Your evaluation will be submitted to the Math Undergrad Office for evaluation. Fifteen percent of the weight is determined by your evaluation, the remainder by the faculty. The combined marks determine whether the report will receive credit.

Thank you for your help in preparing this report,

Nicholas Westbury

TABLE OF CONTENTS

List of Figures

List of Tables

Executive Summary

This reports starts by looking at the theoretical motivation and objectives of testing and then considers some practical testing methods to attain these objectives. It will talk about the advantages, disadvantages, and trade-offs of a variety of software testing approaches and explains some of the considerations that goes into creating a broader company-wide testing strategy. After reading this report, the reader should understand why testing is important, why perfect testing is impossible, and be introduced to different classes of tests. Ultimately, the take-away will be understand the basis for testing and a sample implementation of this basis.

1.0 Introduction

Software testing has always been a core pillar of building reliable computer programs. At its core, software testing is simply running a program and determining if the output is the expected behaviour based on requirements. It is composed of several linked activities: verifying software matches agreed-upon specifications, verifying software completeness, identifying defects in software that can take the form of feature, usability, or performance bugs [1]. As computer applications take an ever-growing roles in our lives, reliability and quality is crucial.

The purpose of the report is to inform on a high-level about the why, what, and how of software testing. It will start by elaborating on why testing is important, background theory on different testing methods, and end with a sample practical application of these testing methods.

2.0 ANALYSIS

2.1 Theory

Exclduing mission-critical applications, software testing has traditionally been seen as afterthought for startups that prioritize shipping features before worrying about stability. Nonetheless, in the long term, it is important to reduce programming bugs thereby increasing quality, leading to happier customers and more revenue. Automated testing leads to less manual regression testing lower cost and developement time.

2.1.1 Motivation: Cost

The most easily measurable cost are the direct financial costs associated with software bugs. According to Tricentis, the cost was a staggering 1.1 trillion dollars in 2016 alone [2]. This figure includes direct expenses such as engineering costs to fix the bugs, the cost to customers being unable or impeded to use the product. Less frequently thought-about costs include the impact on the company's brand, customer loyalty, and level of annoyance customers report.

Moreover, the later a bug is found and repaired, the more expensive it is to fix. A 2008 IBM report found that after a bug makes its way through the design, implementation, testing, customer beta test and finally post-production release, it is roughly 30 times more expensive compared to finding it during the design phase [3]. The earlier throughout testing is done, the cheaper it is. The next figure illustrates the exponential cost nature of defect the later it is found in the development process.

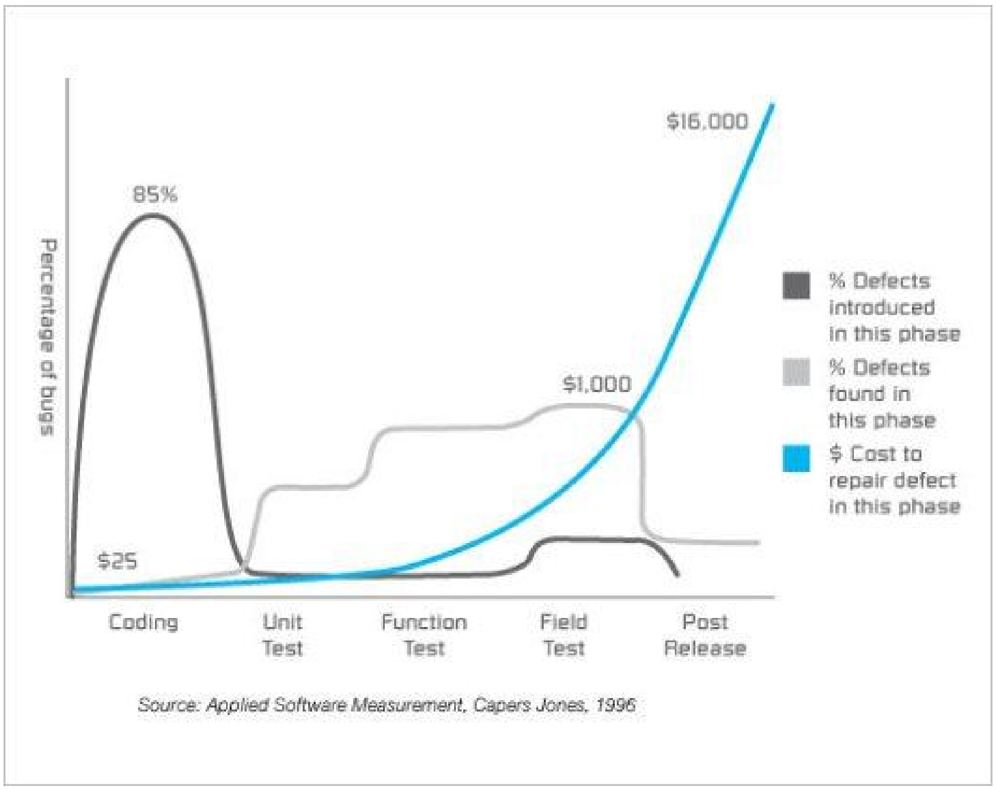

Figure 1: The cost of fixing a bug increases dramatically the later it is caught. [4]

In extreme cases, the costs are staggering. A high-profile case was Mariner 1, a 1962 NASA spacecraft with destination to Venus. Shortly after takeoff, it veered off course due to a software bug and $18 million was lost [5]. Amazon Web Services went down for 4 hours causing nearly $50 million dollars in lost revenue [6]. A software bug in the Therac-25 radiation machine caused larger-than-required dosage to be sent causing at least 3 deaths [7]. These examples are outliers, with thousands of smaller cases of bugs causing unexpected and expensive incorrect behaviour but they illustrate the need for software testing.

2.1.2 Motivation: Customer Happiness

Another common motivating factor is that better testing leads to happier customers. It is hard to measure customer happiness objectively but Medrad, a medical imaging equipment manufacturer, saw an annual revenue growth of 15% for four years, which it attributed to a growing customer base after instituting a quality assurance (QA) program. Similarly, Johnson Controls, an automobile company, found as part of their QA efforts that every 1% increase in customer satisfaction lead to a $13 million positive impact on their bottom line [8].

2.1.2 Why Testing is Difficult

Given that testing has benefits on both the company's bottom line and customer satisfaction, the natural question arises: why don't all software companies treat testing as a high priority? One reason is that testing is difficult. Even well-meaning companies that are willing to invest in QA might be afraid of the complexity.

To illustrate this point, consider the following simple pusedo-code:

Algorithm 1: Simple Divison

```
function DIVIDE(a, b)
    return a / b
end function
```

The number of aspects there is to analyze for test purposes is virtually limitless. For the snippet above, a type-conscience programmer might ask what types are *a* and *b* expected to be? Are they integers, floating-point numbers, or could they even be complex types like imaginary numbers or could a be an array and b an integer? Are these considerations that we should test for or these assumptions fair to make given the domain of the system? Do we need to test for every combination of number or is it sufficient to test certain edge cases? The answer to these questions will depend on an application's own error tolerance.

Language-specific considerations also need to be taken into account. For example, python2 and C will return the floor when dividing integers. For example, 1 divided by 2 gives 0. However, python3 will do float division with integers so $1/2 = 0.5$. A computer scientist might also point out edge cases when the result exceeds the $[-2^{31}, 2^{31} - 1]$ range for 32-bit signed integers leading to under or overflow. Finally a mathematician might ask what would happen for the cases $a/0$ or $0/0$. The amount of knowledge and proper discretion required to test even trivial functions is very high. So high in fact that throughout software testing through mathematical proofs is reserved for mission-critical applications such as space exploration or medical. It is unnecessary and wasteful to test every case for most applications.

2.2 Types

Traditionally, testing can be divided into two large categories: white and black box testing. White-box testing refers to testing when the implementation is known whereas black-box testing is testing where the code is unknown and product-behaviour testing is done. Each category can be further sub-divided into more specific categories.

2.2.1 White-box Testing

White-box includes any tests where the implementation is known to the developer. It is done at the unit testing phase. White-box testing has some advantages and disadvantages compared to black-box testing.

Generally white-box tests are more through than black-box since developers can put emphasis on more complex sections of the code. With knowledge of the underlying code, developers can test the code they know is most complex and therefore more risky. For this reason, performance issues are more easily identified here. Automating the tests is easier when the implementation is known.

There are downsides too. For one, only a programmer is able to write tests. The number of inputs can frequently be too high to test throughly and may not be how the software would be used by a cusomter. White-box testing also has bias to the existing code-base and is generally worst at finding faults rather than testing existing behaviour.

2.2.2 Black-box Testing

Black-box is roughly the opposite of white-box testing. A tester knows *what* the software should do but not *how* it does it. It has the benefit of being able to be tested by anyone even without particular programming knowledge. They are closer to the product specification so it can be easier to find business-use edge-cases, but harder to find code edge-cases compared to white-box testing.

2.3 Testing Phases

Different phases of testing are appropriate for different degrees of specificity of a software testing. Each has its advantages and drawbacks and therefore the phases are often combined as part of a testing pipeline.

2.3.1 Unit Testing

Unit tests code on a functional-level. Generally they are written to ensure function or class-level behaviour is correct. A unit test minimally a single branch of logic and test the most complex or edge cases. Multiple unit tests should ideally cover every logical branch. Unit testing is intented to be used along with other testing method to complete a full testing strategy.

Pros	Cons
• Simple to write • Can be written independently of other code • Serve as specification to other developers by giving concrete examples on how to use a particular function •Can be run in parallel because all external integration is mock •Easy to pinpoint exact line of broken code	• Misleading code coverage because 100% unit test coverage doesn't correspond to complete test coverage • Doesn't cover interaction between components

Table 1: Unit testing pros and cons

2.3.2 Integration Testing

Integration testing tests interaction between individual components with respect to a software specification. It combines unit-tested modules and tests the interaction between them.

Pros	Cons
• Closer to testing the application with user requirements	• Can be challenging to separate unit-test into integration tests dependent on code structure • Not as easy to trace-down mistakes compared to unit tests

Table 2: Integration testing pros and cons

2.3.3 System Testing

System testing is a class of black-box testing where the program requirements are tested based on the product requirements. System testing is done on the completed overall system.

2.3.4 Acceptance Testing

The final testing phase is done by real users. It will be the most accurate test because run tests in real environments. It is often the last step before a product release.

2.4 Sample Practical Implementation

At work, our testing stack consists of a few different technologies. The primary technologies are Jasmine/Karma for client-side unit and integration testing, Frisby for backend API system testing, nosetests for python unit testing, and Selenium for system testing. The choice of these particular technologies is specific to our software stack but is illustrative of a well-rounded testing strategy.

Jasmine and Karma are the testing framework and test runner that allow for unit-testing JavaScript code. We have framework-specific testing to allow Angular modules to be broken down into their natural components: controllers, services, and directives. These are akin to classes in object-oriented languages. Testing these components as a unit instead of individual function has the advantage that they are often closely coupled and directly dependent on another, making the testing closer to what is being run. One frustration is that these unit test require a considerable amount of mocking to seperate an individual component.

Figure 2: Angular/Jasmine/Karma is a popular testing stack for frontend testing. [9]

Python nosetests are similar to Jasmine and Karma unit tests but for the backend instead of the frontend. They are divided from each individual file into the three divisions of our server code: gateways, service, and endpoints. Python has a mature first-party testing framework and generally accomplishes the theoretical goals of unit tests to be discrete, independent, and complete. Nonetheless, one downside is the automated tests for database communication is lengthy because each unit tests clears the database.

Frisby is used for system testing. It allows for HTTP requests to be made to different endpoint. For example, a test case could be to create a question and add it to a survey using the (`/api/create_survey` and `/api/add_survey_question/1` endpoints). Its purpose is to test endpoint backend system.

Finally, there is Selenium, a web browser automation tool. It allows for a realistic recreation of how the client would see and use the website. It is used for full system testing. It allows for nearly arbitrary complex tests but the downside it is considerable more difficult to write, and often more brittle than than other testing technologies. For example, a graphical overhaul would require re-writing Selenium test cases.

Each of these technologies has their limitations but together they cover the full-spectrum of tests. Karma/Jasmine cover the frontend. Frisby/Nosetests cover the backend. Selenium acts as the bridge between them. The figure below visually illustrates where each technology fits in the overall testing strategy.

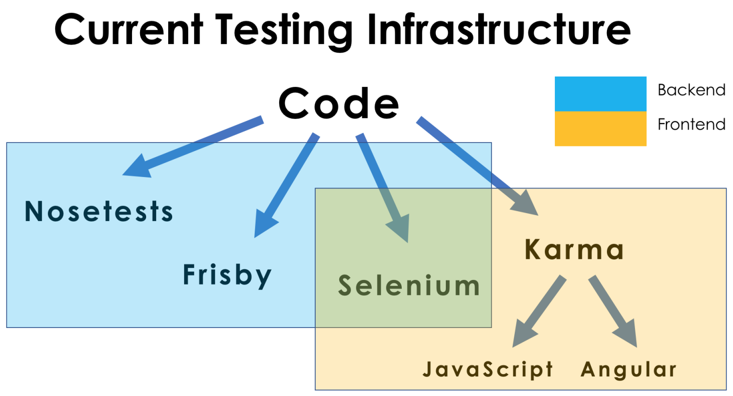

Figure 3: Four main technologies forming our testing infrastructure

3.0 Conclusion

As Perception matures, so does the importance for stability and thereby full-fledged automated tests. Increasing testing reduces the odds of a defect, increases customer satisfaction, and is correlated with increased revenues. Though perfect testing is practically impossible, the unit, integration, system, and acceptance testing pipeline is an effective way of having practical and useful tests. The explanation of our application of this theoretical model on practical product testing should highlight some of the strengths and weaknesses in the real-world. Automated software testing is a powerful tool towards achiving unparalleled quality.

REFERENCES

1. Pan, Jiantao. "Software Testing". Carnegie Mellon University, Spring 1999.st Newspaper, 25 Feb. 2012, https://users.ece.cmu.edu/k̃oopman/des_s99/sw_testing/.

2. Tricentis. "Software Fail Watch: 2016 in Review". Tricentis, 2016. https://www.tricentis.com/resource-assets/software-fail-watch-2016/

3. International Business Machines. "Minimizing code defects to improve software quality and lower development costs", Oct. 2008. ftp://ftp.software.ibm.com/software/rational/info/do-more/RAW14109USEN.pdf

4. Manna, Antonella. "An Introduction to Testing in Production". Belatrix, 18 Aug. 2016. http://www.belatrixsf.com/blog/an-introduction-to-testing-in-production/

5. National Aeronautics and Space Administration (NASA). "Mariner 1", 2018. https://nssdc.gsfc.nasa.gov/nm

6. Weise, Elizabeth. "Massive Amazon cloud service outage disrupts sites" https://www.usatoday.com/story/t
cloud-service-goes-down-sites-scramble/98530914/

7. Levson, Nacy. "Medical Devices: The Therac-25". University of Washington, 1995. http://sunnyday.mit.edu/papers/therac.pdf

8. Turner, Jamie. "How Quality Impacts Your Bottom Line". SmartBear, 11 Apr. 2013. https://smartbear.com/blog/test-and-monitor/how-quality-impacts-your-bottom-line/

9. Rosa, Santiago. "Angular: Unit Testing Jasmine, Karma". 30 Nov. 2017. https://medium.com/frontend-fun/angular-unit-testing-jasmine-karma-step-by-step-e3376d110ab4