UNIVERSITY OF WINDSOR

ELECTRICAL ENGINEERING CO-OP

SOFTWARE TESTING FUNDAMENTALS

CENTERLINE (WINDSOR) LTD.

PRODUCT DEVELOPMENT

WINDSOR, ON

Submitted to: Mr. Alex Bussiere

Submitted by: Eric Parker

Submitted on: January 8, 2018

Work Term Number: Fall 2017

University of Windsor
Faculty of Electrical and Computer Engineering
Windsor, ON N9B 3P4

January 8, 2018

Mr. Alex Bussiere
Product Developer
595 Morton Dr.
Windsor, ON N9J 3T8

Dear Mr. Bussiere,

Please accept this report entitled “Software Testing Fundamentals” as my submission to fulfil my work term requirements.

It was an honour to complete my final work term at CenterLine with the Product Development team. It was my pleasure to have the opportunity to contribute to a wide variety of projects that were important to the success of the corporation. During my time at CenterLine, I was fortunate enough to further develop a strong technical background in software development while also learning and fostering new skills related to circuit component selection and PCB layout.

Overall, this work term has provided me with valuable work experience, a deeper passion for embedded programming and IoT devices, and a better understanding of the product development process. My submitted report outlines the importance of implementing automated software testing procedures for quality assurance purposes and briefly describes how such measures could be implemented within the Product Development team at CenterLine.

Finally, I would like to thank you, my manager, for sharing your experience and knowledge with me throughout my work term. I enjoyed learning from you and I am very grateful that you were willing to share your time with me and answer my questions.

Sincerely,

Eric Parker

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

Executive Summary:

The purpose of this report is to explain the importance of implementing automated software testing procedures for quality assurance purposes and exemplify a few methods through which they could be deployed within the Product Development team at CenterLine. As consumers continue to demand more versatile and capable performance out of electronic products, the software applications behind these devices continue to grow in size and complexity. Unfortunately, linear growth in software complexity results in exponential growth in the amount of possible test cases. Any capable product available on the market today cannot be adequately tested via manual testing alone. Moreover, developers need the ability to easily (and completely) verify that incremental software changes do not adversely affect the performance of any previously developed system capabilities and features.

For this reason (among many others), it is important that software developers in any technical discipline understand how to design and implement simple automated tests to verify the performance of their product. This report will provide a basic understanding of fundamental software testing techniques and demonstrate how automation testing can prove indispensable to delivering quality software-driven electronic products.

Introduction:

This report demonstrates the importance of software testing and provides the reader with a basic understanding of software testing types, methods, and related terminologies. Testing (as it pertains to the software development life cycle) is the process of evaluating a system or its component(s) with the intent to conclude whether it satisfies predefined specific requirements or not. In other words, testing involves executing a system in order to identify any gaps, errors, or missing features. The main objective of software testing is to identify defects and errors during the development phases. This process is essential for establishing customer satisfaction and ensuring quality of product. One way to achieve this is to perform manual testing, whereby the tester takes over the role of the end-user and tests the software without using any automation tools or scripts to identify unexpected behaviour or bugs. Manual testing involves following test cases and test scenarios to ensure completeness as well as performing exploratory testing through which testers with little or no experience with the software or its internals attempt to learn the software in an effort to optimize usability.

Although manual testing is certainly required in order to deliver a quality product, there are many drawbacks associated with depending solely on this approach. Firstly, modern-day programs tend to be relatively large in size and very complex. With added complexity comes exponential growth in the amount of use cases that need to be tested. For example, just two independent `if` statements result in four possible test cases:

1. Both conditions are `false`.
2. The first condition is `true` and the second condition is `false`.
3. The first condition is `false` and the second condition is `true`.
4. Both conditions are `true`.

Following similar logic, it can easily be shown that three independent `if` statements result in eight possible unique outcomes. Thus, through this simple example it can be seen that the number of possible test cases grows exponentially with software complexity according to the relationship 2^n where n represents the number of independent `if` statements in a given program. This rudimentary example investigates how complexity grows via the use of simple `if` statements; there are far more complex programming constructs deployed by software developers to implement program features in modern software. Moreover, any given modern software application may contain several thousand conditional statements. This results in an enormous amount of test cases that couldn't possibly be tested via manual testing (e.g. if $n = 1000$, $2^{1000} = 1.07 \times 10^{301}$ possible test cases). The detriments associated with manual testing are particularly severe when new software changes may affect previously tested program features. If this is the case, complete testing of all features would need to be completed for each small addition to the software. For this reason, software producers need the ability to quickly verify that incremental software changes do not adversely affect the performance of previously developed system capabilities.

Considering the limitations associated with manual testing, it is often necessary to deploy additional software testing techniques that are capable of leveraging the speed and power of automation in order to adequately test software systems.

Overview

CenterLine (Windsor) Limited is a Canadian multinational engineering company with its international headquarters located in Windsor, Ontario. CenterLine also has locations in the USA, Mexico, Brazil, Germany, Romania, India, and China. CenterLine specializes in designing and building advanced automation machinery and products that satisfy resistance welding, metal forming, and cold spray application needs. Primary customers include Original Equipment Manufacturers (OEMs) and Tier suppliers operating within the automotive, mass transit, aerospace, and defense industries.

One of the primary research and development divisions of CenterLine is the Product Development department, who is responsible for designing and prototyping innovative product solutions to meet the needs of customers. Product Development plays a pivotal role in ensuring that CenterLine remains competitive in an ever-changing marketplace. This division of CenterLine continually propels the company toward finding and leveraging new markets. For example, CenterLine very recently began producing microprocessor control solutions and proprietary electronics with great success. Such endeavors have added significant new revenue streams for the company that are experiencing tremendous year over year growth. These changes have led to the Product Development team producing over 10,000 lines of proprietary code each year. This represents an entirely new area of expertise within the company. As the Product Development team grows to accommodate recent successes, they could certainly benefit from adopting standard software development best practices followed by larger software companies in order to maintain a high standard of software quality; automated software testing is certainly one of these.

This report will provide a high-level overview of software testing and its guiding principles, discuss when automated testing should and should not be conducted, explain different testing methods and levels, and then suggest popular language-specific testing tools to assist with automation.

I. Types of Testing

Software testing can be broadly categorized into two main types: manual testing and automation testing. Manual testing, as the name implies, involves testing software manually (i.e. without using any automation tool or script). The tester assumes the role of the end-user and executes test cases and test scenarios to ensure that the software functionalities meet the intended behaviour as defined by the requirements. Automation testing involves using another software and/or writing a custom automation script capable of testing a software product. This process involves automating manual testing and other testing procedures.

Both testing types offer specific advantages and disadvantages. The main benefit of manual testing is it approaches the task from the perspective of the end-user. This provides much needed subjective feedback involving how well a software meets high-level requirements as well as allows for qualitative assessment of the user experience that couldn't otherwise be automated. Moreover, manual testing methods such as exploratory testing allows testers the opportunity to suggest system-level adjustments that could improve overall ease of use even when a system may be objectively meeting requirements. The main disadvantages associated with manual testing are speed and cost. Since all manual testing must be performed by a person, testing procedures are very time consuming and costly due to the wage that the company must pay the tester. This means that it is not possible to test all possible use cases within a software using manual testing alone, which may inadequately identify many bugs and missing requirements resulting in defects being sent to the customer unknown to the development team. Such a circumstance can be very costly as it will lead to increased maintenance and correction costs later in the software development life cycle.

Contrarily, the main benefit of automation testing is its speed and reusability. Since automation testing is capable of testing software much faster than any human could, this allows the automation scripts to test many more use cases of the software resulting in much more complete software verification. Generally, this leads to identification of bugs very early in the software development life cycle ultimately preventing bugs from reaching the customer. Another added benefit of automation testing is test reusability. That is, once the automation script has been created, it can be run at any time in order to fully test the software. This allows testers to completely retest a software application each time a change is made which is not possible with manual testing because it would be too time consuming and costly. This is extremely beneficial to the quality of the software since it ensures that new changes do not result in another fault being uncovered. The main disadvantage associated with automation testing is it is not possible to automate every test case in software. Some complex user-dependent processes cannot be accurately simulated using automation testing, especially when they involve interaction with the physical world (i.e. sensors, digital I/O, etc.). This is not always an impactful detriment considering that not all processes need to undergo automated testing (more on this later in the report). Another perceived disadvantage associated with automation testing is cost. In practice, this is seldom true since lack of testing during development often leads to increased maintenance and correction costs later on in the software development life cycle and early testing saves both time and cost in many aspects.

Since the pros and cons offered by both types of software testing differ greatly, optimal testing results can be achieved by deploying *both* testing methods as required. A particularly powerful combination that is often utilized by software companies is to use automation testing to verify

specific software feature behaviour and performance (including software functions with requirements that are not frequently changing and areas of application that require recurrent testing) combined with exploratory manual testing to improve user experience and validate the software against qualitative requirements.

II. Testing Levels

Just as software testing can be categorized as manual or automation testing according to how the testing is completed, it is often necessary to further classify testing procedures based on scope. That is, software testing levels are defined to identify distinct areas and prevent overlap and repetition between tests. There are four main software testing levels: unit testing, integration testing, system testing, and acceptance testing. These levels should generally be followed in a waterfall style (i.e. proceed to the next level of testing only when the previous level has been completed) and each level of testing increases in scope until the entire application is being tested as a complete unit.

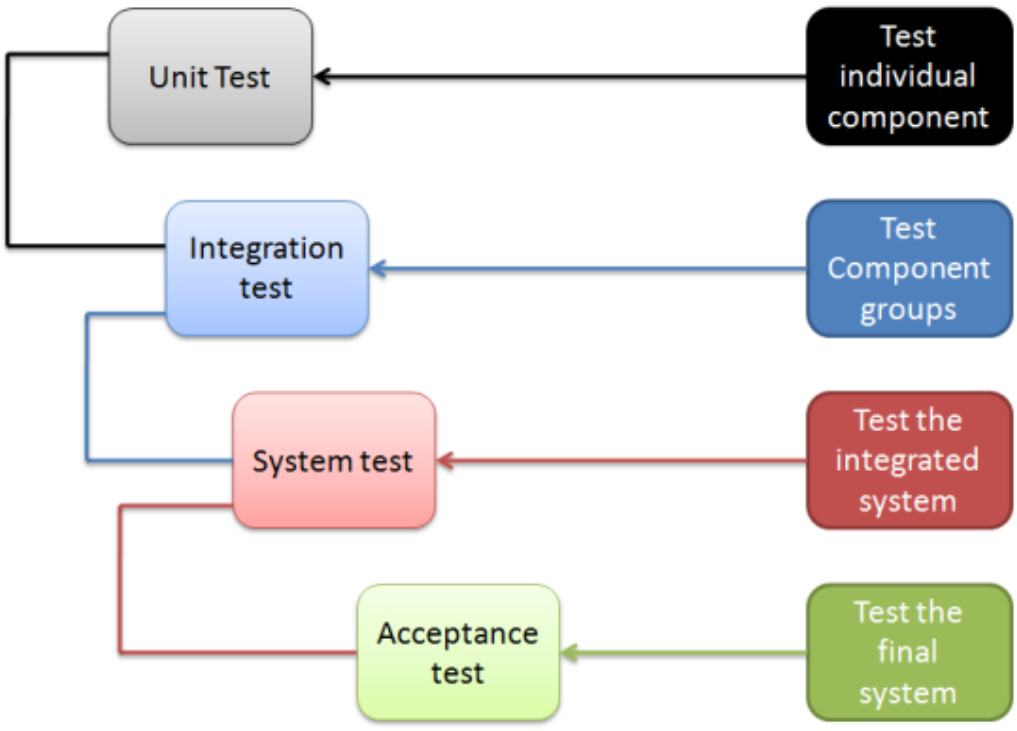

Figure 1: Overview of Software Testing Levels

Unit Testing:

This level of testing consists of separately testing individual units of source code (such as functions or class methods, while simulating all required dependencies). Unit tests are used to quantitatively verify that code is working as expected. Unit testing has the greatest effect on the quality of your code when it's an integral part of a team's software development workflow. Unit tests should be written as soon as the function or method is written and the unit test should verify the behaviour of the code in response to standard, boundary, and incorrect cases of input data. The main restrictions of unit testing are associated with the limited number of scenarios and test data that can be used to verify the source code; unit testing cannot catch each and every bug in an application. However, unit tests are very important for quantitatively verifying software performance and there are a number of tools available that make generating unit tests very quick and easy (more on this later in the report). According to the test driven development software development process, it is recommended to create unit tests before even implementing the code to be tested. This allows unit tests to serve as both documentation and functional specifications for the software. Although creating unit tests may seem like added overhead that will increase development time, in practice generating and utilizing unit tests throughout the development process are one of the best ways to prevent bugs from making it to the customer. For this reason, many of the best software companies in the world require them. Moreover, unit tests are great at catching defects very early in the software development life cycle which drastically reduces the cost and development time associated with fixing the defect.

Integration Testing:

Once all unit tests have been completed and the results are positive (i.e. no failures), the development team can move on to the next software testing level fully confident that each individual unit behaves as intended. Integration testing is one step larger in scope than unit testing, and is used to test how one or more units interact with each other and produce output in various scenarios. Integration testing implementation is highly dependent on the specific software architecture of the application being tested, since it is often not necessary to test the interaction of all units (e.g. if only some units need to interact in the software). There are two main approaches to integration testing: top-down or bottom-up. In bottom-up integration testing, testing begins with unit testing followed by tests of progressively higher-level combinations of units called modules. Top-down is exactly the opposite of bottom-up testing; it begins with the top-level modules and adds lower level modules one by one. In a comprehensive software development environment, it is recommended to begin integration testing via bottom-up testing and then proceed to top-down testing. This approach fully tests all individual units first and then considers their interactions as required.

System Testing:

The next logical progression in the hierarchy of testing levels is one in which the application is rigorously tested as a whole to ensure that it meets quality standards; this is known as system testing. System testing is purely black-box testing (hence the name *system* testing) and as such should require no knowledge of the inner design of the code or logic. System testing tests the design, behaviour, and the perceived expectations of the customer. As such, system level

testing (in contrast to unit testing and integration testing) involves qualitative analysis of the software as a unit. For this reason, various components of system level testing may or may not be automated, whereas unit and integration testing are almost exclusively automated processes. For example, colour schemes and other qualitative user experience characteristics are best assessed manually by a human user, however it is possible to quantify various perceived performance characteristics (such as response time of a software event) which makes automation possible and preferred in some scenarios. The software development team should make these decisions based on the requirements of the project, however it is important to ensure system level testing should test the entire software application as a whole with no knowledge of the inner design of the code.

Acceptance Testing:

Often considered the most important type of testing, acceptance testing is the final step before delivering the software to the customer. This level of testing gauges whether the application meets the intended specifications and satisfies the client's requirements. Generally, this involves a quality assurance team manually testing a pre-written set of scenarios and test cases. Acceptance tests are not only intended to identify cosmetic errors and interface gaps, but they also point out system-level bugs and performance inadequacies. This level of testing differs from system level testing because it is usually completed entirely via manual testing by a quality assurance person who is not directly on the development team and there are often legal or contractual requirements associated with the software that must be demonstrated during this testing level before the software will be accepted by the client.

Regression Testing:

Although this type of testing is not considered one of the four main software testing levels, regression testing is one of the most powerful types of software testing that is used ubiquitously by top software companies. Regression testing is the process of verifying that software which was previously developed and tested still performs the same way after changes are made or new features are added. The motivation behind regression testing comes from the fact that whenever a change in a software application is made, it is quite possible that other areas within the application have been inadvertently affected by this change. There are many types of changes that may generate this circumstance including bug fixes, adding new features, modifying function implementations, making user interface changes, and many more. The intent of regression testing is to ensure that a change should not result in another fault being uncovered in the application. When regression tests are run at the application level, this gives developers and managers alike complete quantitative assurance that new changes or incremental developments will not adversely affect the performance of any critical feature thus allowing the team to quickly and confidently move changes into production. For this reason, regression testing is one of the most common types of software testing performed by software companies. Regression testing is also an excellent candidate for test automation because the testing process involves retesting virtually the entire application each time a small incremental change is made, which is not feasible using manual testing for obvious reasons.

III. When to Automate

Although there are clear benefits to automated testing, it is important to automate in a manner that will maximize return on investment (ROI) according to the needs of the development team. Since it is not possible to automate everything in software, this process includes making decisions about exactly what should be tested and how. Although there may be many factors to consider, the primary criteria should that dictate which areas of an application should be automation tested are:

- Frequently used features
- Areas of software that require frequent retesting
- Tests that are impossible (or take a lot of time or effort) to perform manually
- Program features with requirements that do not change frequently

Firstly, it is important to ensure that critical features including those that are frequently used by the user are tested thoroughly via automation testing especially for regression testing purposes. This ensures that core features fundamental to proper software operation are maintained throughout the development process. Frequently retested functionalities are also a strong candidate for automation to prevent team members from repeating work which improves efficiency and helps maximize ROI. Moreover, automation testing can also be applied to improve test quality as well; as shown previously, it is sometimes not physically possible to manually test all possible use cases in a software. For example, if you have 1000 individual `if` statements in a given application, this means $2^{1000} = 1.07 \times 10^{301}$ possible test cases. For obvious reasons, it is not possible to test all use cases unless thousands of test cases were being performed each second which is only possible via automation testing. Thus, automation testing can be deployed to greatly improve the thoroughness of a software testing process. However, it

is important not to overuse automation testing in order to maximize ROI. Therefore, automation testing should only be used for features or functionalities with relatively concrete requirements. If the requirements for a particular feature are constantly changing, the unit test for that feature will also have to change accordingly. If it is known in advance that a feature is likely to change, it is often in the best interest of the development team to delay implementing automated tests for this feature and use manual testing in the interim until a more definite requirement is determined.

IV. Popular Automation Testing Tools

There are many test automation tools designed to assist development teams with generating automation tests quickly and easily. Using a capable test automation tool enables development teams to test substantially more code, improve test accuracy, and decrease development time to improve overall productivity and ROI. Some testing solutions are proprietary while many others are available for free. Most software testing tools are programming language-specific however many platforms support many languages. Thus, once the development team is familiar with a particular automation tool's API, the team can easily jump between programming languages as required.

Popular test automation tools for the C++ programming language include:

- Boost.Test
- Google Test
- CppTest

For embedded system design, it may seem advantageous to use a platform emulator in order to more easily simulate hardware inputs and outputs. However, there are many detriments to this approach, namely: reduced computation speeds, increased development time, fewer number of test automation tools, and reduced testing capabilities. Instead, the preferred solution is to simulate machine I/O using only software while running the code under test locally on the developer's personal computer. Consider the fact that most commercially available microprocessor devices are programmed using high-level programming languages such as C++. This means popular microprocessor development "languages" (such as the Arduino programming language) are created using common high-level programming languages and

adding platform-specific custom libraries. Due to this, it is not only possible to simulate desired microprocessor-specific language elements using a common high-level programming language, but it is preferred for many reasons: it enables faster development time, allows for the use of much more powerful test automation tools, and completely isolates volatile emulation factors in order to test only the quality of the source code. Note that this method requires that the developer provide mock-up replacements for any platform-specific functions in their test program (such as Arduino functions, Particle functions, etc.). Although this may seem like added development overhead, consider the simple example demonstrating how to simulate the Arduino `millis()` function shown in pure C++ below.

```
unsigned long millis() {
    timeb t_now;
    ftime(&t_now);
    return (t_now.time  - t_start.time) * 1000 + (t_now.millitm - t_start.millitm);
}
```

Figure 2: Arduino millis() Function Mock-Up in Pure C++

As shown in Figure 2, it is very simple to simulate common platform-specific functionalities. Hardware inputs, outputs, and transmission protocols such as serial communication can also be simulated in a similar fashion. Overall, this solution is preferable to running local emulators for reasons described previously.

In general, the decision as to which test automation tool is used by a given development team should be made based on feature demands, the team members' background knowledge, and budget.

Conclusion:

In conclusion, adequate software testing is an important factor in determining the quality of software produced by any development team. As software applications continue to increase in complexity, manual testing alone cannot adequately test all critical use cases especially during the development phases. Developers and managers alike need the ability to easily and accurately verify that incremental software changes do not adversely affect the performance of any previously developed system capabilities and features. This can easily be achieved by utilizing one of the many powerful test automation tools available today. Although automated testing scripts may seem like added overhead that will increase development time, using test scripts throughout the development process is one of the best ways to prevent bugs from making it to the customer and generally lead to cost savings due to reduced maintenance costs and manual testing hours. These are some of the numerous reasons why many of the best software companies in the world have entire teams devoted to automation testing.

CenterLine is doing an excellent job of thoroughly performing manual testing procedures throughout the software development life cycle. It is recommended that CenterLine's Product Development division strongly consider implementing some form of automation testing especially for unit testing and regression testing purposes. Such practices would allow the team to thoroughly test all software and would greatly increase the number of bugs caught during the development phases ultimately resulting in fewer bugs reaching the customer. In order to achieve this goal, the team would also need to develop and document formalized software requirements for each project. Especially when developing firmware for embedded systems, small bugs can easily go unnoticed when using manual testing procedures alone.

REFERENCES

"Automated Testing Best Practices and Tips." *SmartBear*. Last Modified December 2017.

https://smartbear.com/learn/automated-testing/best-practices-for-automation/

"Boost.Test." *boost*. Last Modified August 2017.

http://www.boost.org/doc/libs/1_65_1/libs/test/doc/html/index.html

"Bringing manual and automated testing together." *PractiTest*. Last Modified January 2015.

http://qablog.practitest.com/bringing-manual-and-automated-testing-together/

"Exploratory testing." *Wikipedia*. Last Modified August 2017.

https://en.wikipedia.org/wiki/Exploratory_testing

"How can I unit test Arduino code?" *StackOverflow*. Last Modified May 2013.

https://stackoverflow.com/questions/780819/how-can-i-unit-test-arduino-code

"Regression testing." *Wikipedia*. Last Modified January 2018.

https://en.wikipedia.org/wiki/Regression_testing

"Should I write unit tests? Yes and I'll tell you why." *Hubstaff*. Last Modified February 2014.

https://blog.hubstaff.com/why-you-should-write-unit-tests/

"Software Testing." *Tutorialspoint*. Last Modified January 2016.

https://www.tutorialspoint.com/software_testing/software_testing_tutorial.pdf

"Software Testing Levels." *Test Institute*. Last Modified January 2018.

http://www.test-institute.org/Software_Testing_Levels.php

"Test Driven Development (TDD): Example Walkthrough." *Technology Conversations*. Last Modified December 2013.

https://technologyconversations.com/2013/12/20/test-driven-development-tdd-example-walkthrough/

"Unit Test Basics." *Microsoft*. Last Modified January 2016.

https://msdn.microsoft.com/en-us/library/hh694602.aspx

"Why is software testing necessary?" *UTest*. Last Modified May 2016.

https://www.utest.com/articles/why-is-software-testing-necessary