AUTOMATED DEVELOPER TESTING

Baseline Training

Andrew Ray

4/25/2019

Table of contents

[1 Introduction 3](#_Toc6767224)

[2 Goals and Benefits 3](#_Toc6767225)

[3 Prerequisites 3](#_Toc6767226)

[4 Project 3](#_Toc6767227)

[5 Course Outline 4](#_Toc6767228)

[1.1. *Day 1 Subject* 4](#_Toc6767229)

[1.2. *Day 2 Subject* 4](#_Toc6767230)

[6 Member’s Responsibilities 4](#_Toc6767231)

[7 Reviewer’s Responsibilities 4](#_Toc6767232)

[8 Schedule Time Line 5](#_Toc6767233)

[**NOTE:** Please insert the proper document classification in the footer of this template (i.e., “CGI Internal,” “Confidential” or “Highly Confidential.” Refer to CGI’s [Information Classification Policy](https://ensembleca.ent.cgi.com/business/91292/SecurityDataPrivacy/CGI_Info_Classification_Policy.pdf) for help in determining the proper classification for your document.]

# Introduction

*{Give a brief description of the training. Describe objects and give a quick overview of the material.}*

Upon completion of this course, an engineer should be able to demonstrate an intermediate level of competence in the technology.

# Goals and Benefits

The goal of this course is to provide members with an understanding of the fundamentals to development and solution architecture. Through the completion of this course, members will develop a solid understanding of architecture and solution development basics. Completing the course and attaining endorsement from the reviewer will aid in CGI’s resource management and allocation, as management will have a more quantifiable way of measuring member’s skill levels.

The benefits of receiving the *Automated Developer Testing* course endorsement from the reviewer to the member include:

* Expansion of technical knowledge
* Recognition within the organization

The benefits of receiving the *Automated Developer Testing* course endorsement from the reviewer to CGI include:

* Fundamental architecture knowledge of across the consulting pool
* Accurate evaluation of skills across the consulting pool
* An increased culture of information sharing and peer accountability in the consulting pool

# Prerequisites

* Bootcamp or MVC 101
* SQL 101

# Project

All details surrounding starting and completing the project associated with the training.

# Course Outline

In order to demonstrate competency in *Automated Developer Testing*, a member must demonstrate knowledge of the following topics:

* 1. *Benefits of Automated Testing*

There are many types of automated tests, but in this document we will be focusing strictly on automated developer tests. Automated tests, as the name suggests, are a method of automatically testing and verifying the correctness of a system. The scope of the test size can span from a single method, to an entire system. These tests will return a result which will be a success or failure based on an assertion.

Benefits of Automated Testing

Why do we want to create automated tests? There is no such thing as a free lunch, and this is certainly true within the realm of automated tests.

Benefit 1: Confidence

What does fear do to a software developer?

* Makes you tentative
* Makes you want to hole up and communicate less
* Makes you shy away from feedback
* Makes you unhappy

**[Beck2014]**

Automated testing allows you to program courageously. Move forward confidently knowing that every step of the way your software still works. In programming we have a constant balance between fear and confidence. Anytime we become fearful, we write more tests. If we approach boredom, we write fewer tests. We never want to reach a point where we are writing tests for the sake of writing tests. Each test should be carefully evaluated from a value, cost, and risk perspective.

Automated testing allows tech leads to sleep at night after a production deployment , knowing that their software is stable. It allows team dynamics to remain free from stress, not only from within the project team but also between marketing, management, or any other project stakeholders.

In a study performed by Google in 2016 to determine traits of highly effective teams, Google found that the number one factor was an environment where team mates had “Psychological safety” **[Google2016]**. Automated developer testing can the confidence it provides play into this significantly, and I do not believe it’s effects can be understated.

This idea is supported by other experts in the field. In the 2017 State of Devops report provided by Puppet and DORA. They highlight four key points:

* Establishing and supporting generative and high-trust cultural norms.
* Implementing technologies and processes that enable developer productivity, reducing code deployment lead times and supporting more reliable infrastructures.
* Supporting team experimentation and innovation, to create and implement better products faster.
* Working across organizational silos to achieve strategic alignment.

**[Puppet2017]**

These points are presented within the context of how leadership can have significant positive impact on a team, but the underlying effects are just as powerful when provided via other sources as well. Jez Humble of DORA draws the comparison of this report and the Google study himself during one of his presentations at the 2017 Goto conference **[Humble2017]**.

The Devops report also finds a correlation between higher levels of automation and high performing teams. Refer to the Devops reports listed in the references section of this document for a full definition of High Performing teams and the exact metrics. DORA and Puppet publish a report every year, and I highly recommend them.

The below excerpts from an article written by Mike Bland, a former Googler, speaks toward this with the Google Web Server team as an example.

Mike speaks about the Google team which is full of many of the best developers in the world. Google did not embrace automated testing until the mid 2000s. So why didn’t Google stay that way?

The Google Web Server Story

Despite the risks and the costs, it's important to realize that the benefits of unit testing go beyond merely minimizing the chances of releasing catastrophic bugs.

When I joined Google in 2005, it was already very successful and many "long timers" believed it was because we were doing everything right. As a result, at that time and for some years afterwards, there was a lot of resistance to change. However, as the user base and potential for catastrophe exploded, and as success and the growth that came with it caught up to Google, it became clear that more “rock stars” producing “rock star” code was going to produce nothing but a bunch of noise and confusion in the long-term. An influx of new Google developers eventually helped accelerate the cultural shift towards unit testing adoption, both because these new developers were open to the idea, and because testing eventually proved effective in helping these new folks get up to speed and avoid making mistakes.

As a concrete example, let's take what is possibly the most popular page on the Internet: Google's home page. The Google Web Server (GWS) team's unit testing story became well-known throughout the company. The GWS Team had gotten into a position in the mid 2000's where it was difficult to make changes to the web server, a C++ application serving Google's home page and many other Google web pages. Despite this difficulty, integrating new features was integral to the success of Google as a business. The barrier that was stopping people from making changes as rapidly as possible was the same that slows change on most mature codebases: a quite reasonable fear that changes will introduce bugs.

[Fear is the mind-killer.](http://en.wikipedia.org/wiki/Bene_Gesserit#Litany_against_fear) It stops new team members from changing things because they don't understand the system, and it stops experienced people changing things because they understand it all too well.

The Google Web Server Team took a hard line: No code was accepted without an accompanying unit test.

Determined to overcome this fear, the GWS Team introduced a testing culture. They took a hard line: No code was accepted, no code review was approved without an accompanying unit test. This often frustrated contributors from other teams trying to launch their features, but the GWS Team stuck to its guns.

Over time, unit test coverage and development momentum went up, while defect, production rollback, and emergency release counts went down. New team members found themselves becoming productive far more quickly because the tests allowed them to gain a deeper perspective on a system one unit at a time, and to begin contributing changes with the confidence that the existing tests would likely detect any unexpected side-effects. Any tests they caused to fail in the course of their early efforts accelerated their grasp of the system. Experienced members of the team, who had grown cautious of making changes and accepting changes from contributors, were able to make and accept changes quickly for the same reason and no longer had to rely primarily upon large and expensive system or manual tests with feedback cycles on the order of hours or days. Adding more new developers actually allowed the team to move faster and do more, avoiding the scenario described by [Brooks's Law](http://en.wikipedia.org/wiki/Brooks%27_law) in which "adding manpower to a late software project makes it later".

Furthermore, the mitigation of fear led to the expansion of their joy in programming, as they could see tangible progress being made towards exciting new milestones without being held back by chronic outbreaks of high-priority bugs. The impact on productivity of high morale, based on the ability to remain in a state of creative flow, cannot be overstated. While I was at Google, the GWS Team exhibited the ideal testing culture, integrating an enormous number of complex changes from outside contributors while making their own constant improvements.

Thanks to the GWS example inspiring the efforts of the Testing Grouplet (a team of developers volunteering to promote unit testing adoption, described in a later section of this article), many teams at Google were able to transition to a unit testing culture and benefit from reduced fear and increased productivity. It did take time to overcome inertia, indifference, the friction of outdated tools, and resistance, since at first unit testing felt like a cost and some people worried that the time spent writing that second representation of behavior could be spent writing new code (that would get them promoted). Eventually, as people experienced what it meant to cast aside the fear of change, they came to see this side-effect as easily outweighing those lines of code, in terms of its impact on their happiness, on their team's happiness, and on the bottom-line of productive output. **[Bland2014]**

Benefit 2: Defect Reduction

By having a comprehensive set of tests that validate the behavior of a system, we can drastically reduce the number of defects.

“Defects destroy the trust required for effective software development. The customers need to be able to trust the software, The managers need to be able to trust the reports of progress. The programmers need to be able to trust each other. Defects destroy that trust. Without trust, people spend much of their time defending themselves against responsibility that someone else may have made a mistake” **[Beck2015]**

It is no secret that defects and regression are one of the biggest enemies a large software project has to face. As the size of the project continues to grow, it becomes impossible to contain the entire application within a single developers brain. The developer can no longer confidently say that when they have made a change that they have not introduced incorrect behavior into the system. Many software projects over their lifespan continue to accrue more and more defects until the entire development team ceases to work on new functionality, and is spending their entire work day correcting defects. This is no exaggeration, and is very common among large software projects. Defects also cost more the later they are found in the software development lifecycle. It is in everyone’s best interest to catch defects early, and to prevent them altogether if possible.

This can have striking results.

“A good example of this can be seen in the transformation at HP LaserJet.4 The firmware division was on the critical path for hardware releases; by undertaking a continuous improvement initiative and investing in automation — including significant investments in automated testing — HP LaserJet was able to increase time spent on developing new features by **700 percent**.”  **[Puppet2017]**

This holds true for high performers across the industry:

“When we compared high performers to low performers, we found that high performers are doing significantly less manual work, and so have automated:

* 33 percent more of their configuration management.
* 27 percent more of their testing.
* 30 percent more of their deployments.
* 27 percent more of their change approval processes.”

Automated testing has a significant impact in defect reduction and reducing regression. Having fewer defects opens the doors to many interesting and more efficient work practices. Since we are no longer reacting to a constant stream of defects, our teams can now be proactive, and pursue some of the following:

Proactive Quality Assurance

If the defect density can be reduced enough, then quality assurance can shift from reactive work to proactive work. QA teams can begin to help plan what the correct behavior of the software is. This level of involvement helps reduce the amount of time a developer might spend developing a piece of software that will later fail QA. It also increases the effectiveness and reduces the time of the QA spends on each item. The QA engineer is already familiar with the use cases since they helped plan them, and as a result can make more effective decisions more quickly when testing.

Faster Release Cycles

In a fast moving industry where services are deployed with new functionality multiple times a day, quality checks have to exist to ensure that faulty components are not being deployed. These quality checks cannot appear in the form of mandatory gate keepers (Certification and Delivery). Which can slow the process down too much, with negligible benefits. Tests fill this void.

Defect Root Cause Analysis

When defects are reduced to a low number, they can also be evaluated thoroughly. Why did the defect occur? Was it a misinterpreted requirement? Was there a complex implementation and an edge case was mixed? Understanding why a defect exists can be used to drive continuous improvement so that the team does not introduce similar defects in the future. When there are too many defects this is not possible, as the team is struggling just to resolve all of the defects and survive.

The following article by Martin Fowler explores the idea of self testing code, and its many benefits:

<https://www.martinfowler.com/bliki/SelfTestingCode.html>

Other Minor Benefits

The below benefits are primarily taken from Mike Bland’s write up in the following blog post: <https://www.martinfowler.com/articles/testing-culture.html>

I highly recommend giving the entire post a read.

Improved Code Quality

Writing tests for code that is tightly coupled, is very difficult, and can be entirely impossible if you strive for testing only one unit at a time. TDD naturally supports software design principles such as the Open / Closed Principle, Dependency Inversion Principle, and Single Responsibility Principle. You can even build a dynamic sweet of unit tests to run against all classes that implement a given interface and evaluate if it violates the Liskov Substitution Principle.

Far from being an exercise in academic purity, code quality matters. Bad code provides bugs with plenty of shadows in which to hide; good code increases the chances that they will be found and squashed sooner rather than later. When the author of a piece of code writes a test for that code, the author effectively becomes the first user. Just as [eating your own dogfood](http://en.wikipedia.org/wiki/Eating_your_own_dog_food) is good software development practice at the overall product level, having to write code that uses your own code can lead to improved designs that are more readable, maintainable, and debuggable.

Think of what problems you're trying to solve with the code you're writing; then think of the code you'd like to write, as a client, to make use of the solution. That ideal client code can be expressed as unit test cases that use the interface of the code you're developing.

When code-level design is approached this way, all of the smaller pieces that make up the larger system become not just more reliable, but easier to understand. This makes everyone more productive, as the mental effort required to comprehend what a specific piece of code does is minimized. **[Bland2014]**

More Productivity

Unit testing is not in the same class as integration testing, or system testing, or any kind of adversarial "black-box" testing that tries to exercise a system based solely on its interface contract. These types of tests can be automated in the same style as unit tests, perhaps even using the same tools and frameworks, and that's a good thing. However, unit tests codify the intent of a specific low-level unit of code. They are focused, and they are fast. When an automated test breaks during development, the responsible code change is rapidly identified and addressed.

This [rapid feedback cycle](http://en.wikipedia.org/wiki/OODA_loop) generates a sense of [flow](http://en.wikipedia.org/wiki/Flow_(psychology)) during development, which is the ideal state of focus and motivation needed to solve complex problems. Contrast that with the opposite phenomenon, using the familiar operating systems metaphor of [context switching](http://en.wikipedia.org/wiki/Context_switch). Context switching requires that the present state of operations be saved somehow, and that a new state of operations be swapped in before initiating the new activity; then there's the time and effort involved in switching back. Plus, there's the issue of how much state must be managed per operation. Without unit tests, we have to use more of our brains to remember weird corner cases and strange side-effects, giving us less time and energy to do the thing we're better at than the computer: advancing solutions to new problems rather than juggling the weight of all the problems that have already been solved.

In other words, you can be more productive since you can iterate on code much quicker: You don't need to start up some heavyweight server if you can just run a unit test instead. So if it takes a few tries to get some code right, those few tries might take minutes (or longer) if you have to start up a server again and again, compared to seconds if you just need to rerun the unit tests each time. **[Bland2014]**

Executable Documentation

Well-written unit tests can provide two types of documentation: the test names act as a sort of specification of the code's behavior; and the tests themselves act as code samples for each behavior case. Even better than typical Application Programming Interface (API) documentation, well-maintained unit tests are by definition an up-to-date representation of actual behavior. The author of a unit test effectively communicates to other developers how a piece of code should be used, and what to expect from it. These "other developers" may be brand new to the team, or may not yet be hired (or even born). Such documentation helps developers understand unfamiliar code, even entire systems, without interrupting anyone else to the degree that they might without unit tests.

Poorly-written unit tests lack this quality, usually because less thought is given to test code than "production" code. The solution: Set the same quality bar for test code as production code. If you don't, your tests will become hard to maintain and slow down the team. **[Bland2014]**

Accelerated Understanding

Think of it like this: Every time a test fails, that is an opportunity to deepen your understanding of the system. If you're new to a team, breaking many tests as you begin to make changes to the system can help you become productive far more quickly, as each of these events align your awareness of the system more closely with reality. If you've been on the team for a long time, existing tests will answer many questions that new contributors may have, saving your time and focus. They will also remind you of all the nuances of the code you might have written in the past, and haven't had to think about for some time, should you have to dive back into it. In other words, you benefit your future self when adding a well-crafted suite of tests to your code, minimizing the time needed to context-switch back into that prior state of mind.

Think of the opposite, as was the case in the pre-unit testing days of GWS: When you're on a project that doesn't have ample unit testing coverage, you're afraid to do anything since you don't know what you might break. **[Bland2014]**

### Faster Bug Hunting

Imagine a bug is found in integration or system testing, or after a new release is pushed to a datacenter, or perhaps by a user some time after that. The developers responsible for the buggy code have already moved on to other tasks, and are likely under deadline pressure to deliver. If the bug is severe enough, at least one of those developers will have to stop to address it, slowing the progress of the new development work underway.

If the buggy code is well-covered by a suite of automated tests, especially small unit tests, this interruption may not take much time on the part of the developer assigned to fix the bug. The existing tests serve as documentation of the intent of the affected code. The developer adds a new test to reproduce the bug, verifying that the defect is well-understood before attempting to fix it. This new test verifies the fix for the bug, and the existing tests provide a high degree of confidence that the fix is free of unintentional side-effects. The new test becomes a permanent part of the test suite to guard against regression, the fix is released, and development on the new release continues. The interruption is finished.

Contrast that against the situation where the buggy code isn't well-covered by unit tests. The developer must take time to understand the affected code and far more care to pinpoint the error and ensure its fix is free of side-effects. Verification of the fix may not come for days or even longer, depending on the nature of whatever pre-release testing happens to be in place, if any. The interruption is prolonged, and drains more development and testing time from the new release.

Or, even worse: The team may decide to leave the bug in-place from fear of breaking something else. That certainly doesn't inspire user trust, much less developer confidence and productivity. **[Bland 2014]**

* 1. *Benefits of Test Driven Development*
  2. *Tradeoffs of Automated Developer Testing*

*Outline of Day 2 Subject*

* *Url to Day 2 training article*
  1. *Tradeoffs of Test Driven Development*

*Outline of Day 2 Subject*

* *Url to Day 2 training article*
  1. *High Level Overview of Automated Testing*

*Outline of Day 2 Subject*

* *Url to Day 2 training article*
  1. *Unit Testing*

*Outline of Day 2 Subject*

* *Url to Day 2 training article*
  1. *Fakes, Stubs, and Mocks*

*Outline of Day 2 Subject*

* *Url to Day 2 training article*
  1. *Isolation Frameworks*

*Outline of Day 2 Subject*

* *Url to Day 2 training article*

# Member’s Responsibilities

A member will be required to demonstrate knowledge of all items identified in the Course Outline section of this document above and create a .Net Framework solution for the Project requires above. The intent is that the created solution does not necessarily have to contain or demonstrate all of the elements detailed in the Course Outline section. However, at demonstration time, the member must definitively demonstrate detailed knowledge of how to implement each skill.

# Reviewer’s Responsibilities

The reviewer is responsible for certifying the member as having completed this training. The reviewer must perform the following steps in the process:

* Review completed assignment with member
* Assess knowledge of all core skills identified in the Course Outline section
* Provide notification of course completion to the Member Relations Coordinator

# Schedule Time Line

The following timeline is recommended for the core competency process:

|  |  |
| --- | --- |
| Task | Timeline |
| Complete review of all items within the Course Outline section | *20 hours* |
| Complete development of assignment | *20-40 hours* |