Automated Developer 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.

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. We have a constant balance between fear and confidence. Anytime we become fearful, we write more tests. Once we feel we are on solid ground again. 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]**.

With this confidence, we can now confidently refactor knowing that if all of the tests pass that our software behaves as expected. Since we are able to refactor safely, our technical debt can be kept under strict control. We can release builds to production environments without going through extensive regression and change management boards. We can move more quickly, safely, and effectively.

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]**

# 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.”

**[Puppet2017]**

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. 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.

# Quick 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), or scheduled code reviews, since those will slow the process down too much. 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 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]**

**Test Driven Development**

Test Driven Development is a technique that involves writing tests prior to writing the associated implementation. There are various rules that are followed to various levels, such as:

* Write new code only if an automated test has failed
* Eliminate duplication

The primary mantra of TDD is Red, Green, Refactor. This involves:

* Writing a failing test that will not compile. (Red)
* Writing enough code to make your test compile, but not pass.
* Writing enough code to make your test pass. (Green)
* Refactoring your code to be cleaner and eliminate duplication (Refactor)

# Benefits of TDD

While TDD, does share all of the benefits of Automated Developer Testing, it comes with its own share of unique benefits. Other benefits that Automated Developer Testing provides are enhanced by TDD to a greater degree.

# Decomposing Problems

Imagine programming as turning a bucket of water from a well. When a bucket is small, a free spinning crank is fine. When a bucket is big and full of water, you’re going to get tired before the bucket is all the way up. You need a ratchet mechanism to enable you to rest between bouts of cranking. The heavier the bucket, the closer the teeth on the ratchet need to be. The tests in test driven development are the teeth on the ratchet. **[Beck2014]**

TDD is a tool to manage difficult tasks, one small step at a time.

# TDD Keeps you honest

Writing tests first ensures that you will not skip out on this critical step later. It allows you to get into a flow where tests come naturally, and the implementation follows suit.

# Improved Code Quality Faster

Since you write tests firsts with TDD, you discover coupling between classes quickly. This becomes less of an issue as developers become more experienced and learn to design effective code without the guardrails of TDD, but when starting out, tightly coupled classes is a major concern.

# Evaluating Assumptions Early

TDD forces you to evaluate your assumptions about how the system should work immediately. You state your test cases up front, and then design a solution that will satisfy that criteria.

When writing tests gets challenging, it is often a sign that our software is becoming rigid. That being said this is not always the case, and it should be carefully evaluated. Changing software for the sole purpose of allowing a component to be tested (when the change otherwise is not a valuable change) is often considered a code smell.

# Developer Satisfaction

TDD is a stress reliever. It feels good to make predictable, measured progress. It feels good to know that each step of the way all of the previous functionality is still working as expected. It is extremely satisfying to write a new test, and after implementing the required functionality have it flash green. It turns software development into a type a minigame that I’m playing. All of these things make a life as a software developer more enjoyable.

Automated Developer Testing Tradeoffs

Like I said in the beginning of this document, there is no such thing as a free lunch. Automated testing is not free, and the costs and benefits should be weighed carefully. While it is fairly universally accepted that all software projects should have some form of automated testing, we must always consider how much and of what type. As with all things, we have diminishing returns as we approach an asymptote.

# Developers Must Be Trained in Automated Testing

There will be a learning curve. Like any skill that relies on craft rather than rote processes, a programmer learning to write unit tests will have to go through phases of learning and development, of trial and error, reflection, experimentation, and integration. This takes time, energy, and funding away from other activities. It will cause an initial slow-down in development as people grow accustomed to the practice.

That said, this is a one-time cost. The cost of bringing someone up to speed on unit testing is relatively low if good unit testing practices are already in place on a team, and unit testing skills are portable from one project to the next. Hence, the learning curve is steepest for teams that don't have any unit testing practices at all.

Unit testing, like any other tool, language, or process, can be applied poorly—especially when one first begins, and even more so if one has no good examples to follow, nor mentors for guidance. Unit tests which are brittle, large, slow, perpetually broken (and subsequently ignored), or flaky set bad examples which can get replicated through an entire test suite like a virus. Poorly-written tests can actually be worse than no tests at all, leaving the impression that testing is a waste of time. Builds remain broken and ignored, flooding the testing signal with the noise of constant failures. Developers uninterested in working with the testing environment become willing to live with the fear of making slow and painful changes. The end result is a drag on productivity, an increased risk of defects, and a team convinced that testing is for other people.

To remedy this lack of knowledge and experience, motivated developers can band together to improve one another's unit testing skills and increase the amount of test coverage of the code base over time. In this section, I will describe how the Google Web Server Team built up its test coverage and achieved a high degree of overall productivity; in later sections I will explain how Google as a whole was able to adopt a unit testing culture, and how lessons from that experience may apply to individual teams. However, self-training will take time and energy, and the big-picture payoff may not be immediately apparent, so it requires patience, honest effort, and commitment to see all the way through. Over time, though, as the code base grows and more developers join the team, the value becomes increasingly clear. A two-person team might manage without unit testing, but a twenty-person team will have a harder time, as feature and communication complexity is compounded.

If developers are not motivated to research available materials and improve their skills, or just don't have any idea how to begin, this may imply the need to invest in internal training programs or to contract outside help to provide training. This can lead to a bit of price-shock if resources are tight, deadlines are looming, and the future benefits do not seem clear. The time required to learn the necessary skills should be no greater than that required to train developers in any other skill or technology; but if developers resist, the process can become more drawn-out, painful, and expensive. **[Bland2014]**

# Tests Must be Maintained

Tests should be treated as first class citizens. They are very important and must be maintained properly. If you developers cannot trust the tests in terms of reliability or if the tests take too long to run, the developers will no longer use the tests. If your team has doubts about what the tests say about the reliability of the code, you lose many of the advantages that automated testing provides.

Sometimes, tests themselves can become a maintenance burden; it may seem like they paint a project into a corner, restricting progress rather than maximizing it. This is a particular danger to new teams that lack experience with unit testing and don't understand its value. Mock objects are prone to misuse by inexperienced practitioners, leading to brittle tests of dubious value. With experience, this scenario becomes less likely. You eventually learn to step back, reevaluate the goal of the code and the test, and rewrite one, the other, or both. In the meanwhile, it may become necessary at times to replace an overly-restrictive test rather than to spend the effort salvaging it.

Speaking of new projects or teams or companies or domains, as ideal as it may be to follow [Agile](https://www.martinfowler.com/agile.html) practices to the letter and practice pure [Test Driven Development](https://www.martinfowler.com/bliki/TestDrivenDevelopment.html) (TDD) at all times, sometimes a developer or a team needs to explore, to play, before getting serious about defining expectations and behavior. (Some argue that always following all Agile practices to the letter is a demonstration that you don't understand Agile.) While it’s always nice to get testing experience as early as possible on a project, sometimes you just need to write throw-away, prototype code; in that case thorough unit testing is probably overkill. This may be especially true of startups trying to launch a product as fast as possible.

On the other hand, be aware of the saying: "There is nothing more permanent than throw-away code." The trade-off is that the more features are implemented without accompanying tests, the more [Technical Debt](https://www.martinfowler.com/bliki/TechnicalDebt.html) a team builds up that must be repaid later. Unit testing can be difficult if you don't design for testability from the start—using [dependency injection](https://www.martinfowler.com/articles/injection.html), writing well-defined classes that focus on one thing, and so forth. It is up to the team to gauge the acceptable limits of such debt, and at which point it must be paid to avoid an even more expensive rewrite once maintenance and new feature development grow too cumbersome.

**[Bland2014]**

# Tests Can Result in a False Sense of Security

There's no guarantee that unit tests themselves will be bug-free. Consider this example (in C++-like pseudocode based on the [Google Test](https://code.google.com/p/googletest/) framework):

TEST\_F(FooTest, IfAPresentFilterB) {

setup input and add "A:" , "B:"

run call

EXPECT\_TRUE(PresentInOutput("A:"))

EXPECT\_FALSE(PresentInOutput("B"))

}

The second expectation in this test should check for "B:", with a colon, not just "B". If the code under test accidentally filters for "B" without a colon, the test will pass when it should fail.

It's arguable that the test makes things worse in this case, providing a false sense of security. However, the bug could exist even if the test hadn't been written; given the existence of the buggy test, fixing the code and the test is tantamount to providing a regression test for the bug. Fixing the test and learning from the mistake provides value; blaming the test and deleting it is a step backwards. As one possible measure to avoid buggy tests in the future, the team responsible for such a bug could endeavor to take a closer look at the test code submitted as part of future code reviews, to provide it with the same priority and care as "production" code.

In practice, buggy unit tests tend to be the exception. If practicing pure Test-Driven Development, a failing test should be written before the code that makes it pass; this could help to prevent such bugs. If not practicing pure TDD, temporarily adding an error into the code under test to make sure the test will fail can also help. In either case, writing multiple test cases that check that the code doesn't do what it shouldn't do (instead of just checking the happy path where all inputs are valid) may reveal bugs in other test cases. Still, the possibility remains that unit tests themselves may contain bugs, especially if care isn't taken to ensure that they fail when they're supposed to. **[Bland2014]**

# Test Induced Design Damage

Sometimes software designs are not easily testable. Abstractions may be introduced for the sake of testing, and only for the sake of testing. These abstractions are test induced design damage. Generally, modifying code for the sole purpose of testing is considered a code smell. That being said, there are times where having the test can be very valuable. You must evaluate how valuable the test you are writing is and contrast that value with the potential damage being done to your code base. In many cases, there will be no damage at all. However, in cases that require mocking, indirection and abstraction for the sake of testing, carefully consider the damage that is being done to the cohesion and readability of the code for their sake.

Arguments are often made that even if an abstraction is only created for the use of tests, it does keep options open in the future for alternate implementations. This is often a defense given for creating an abstraction over the data access components. It is also correctly pointed out that these implementations rarely change. You are unlikely to swap from SQL Server to Oracle in a project. While “swapability” can be valuable, it is important to be honest with ourselves about the potential of a swap to actually happen.

In the following video series, Martin Fowler, Kent Beck, and David Heinemeier Hansson (DHH) discuss some of the drawbacks of automated testing and TDD:

<https://martinfowler.com/articles/is-tdd-dead/>

TDD Trade Offs

### TDD Encourages Testing Extremes

TDD has been criticized for encouraging testing extremes that are not necessarily the correct approach in all cases. If I am creating software that will be utilized by a pacemaker, and will responsible for keeping people alive, my unit testing coverage and reliability of my software will be held to a much higher standard than that of a CRUD application. Achieving higher levels of reliability requires an exponential increase in costs. 99% reliability is several times more expensive than 95% reliability, and 99.99999% reliability is several orders of magnitude more expensive again.

TDD can take a hard line that 100% test coverage is a goal. It is often stated with TDD that you should not write any production code without it being mandated by a failing test. These are extremes, and are not necessarily valuable. In fact, these extremes can be very damaging. We should make cost/benefit decisions, and never resort to a default answer based on a dogmatic rule.

<https://dhh.dk/2014/tdd-is-dead-long-live-testing.html>

<https://martinfowler.com/articles/is-tdd-dead/>

### Refactoring Can Be Skipped

TDD is driven by the Red, Green, Refactor loop, but the dopamine spike driven by the green bar arrives during the Green phase. At times, the refactor step is skipped resulting in suboptimal code. This is a point brought up by DHH in the “Is TDD Dead” series. DHH holds the position that developers chasing the dopamine spike of TDD is the cause, while Martin and Kent hold that this is a sign of a lack of experience.

### TDD is not the way everyone’s brain works

For some people, it is very natural to write a test first and then the create an implementation that satisfies the given condition. For others, they think about this in the inverse. It is difficult to think about what the test should be without getting into the implementation. While TDD can be a valuable tool for the tool belt, when it is mandated it can be damaging developers who think differently.

How to write Automated Developer Tests

Finally, we are at a point where we can start talking about writing tests! The following sections will pull heavily from The Art of Unit Testing by Roy Osherove. I highly recommend the book.

Unit Tests

Unit tests are the fundamental building blocks of automated testing. In the following examples, I will be writing tests after code. It is up to you whether or not you would like to use TDD during the project. Some (Such as Kent Beck) recommend starting with TDD right away. Others, such as Roy Osherove, say that you should focus on learning one concept at a time. I will leave the choice up to you.

# Definitions

For the definitions listed in this section, I will mostly be referring to the definitions used by Roy Osherove in his book. These terms definitely have some flexibility to them, and the scope may change from one context to another, but these terms and definitions listed here will serve as a foundation for the rest of this course.

### Unit of Work

A unit of work is the sum o factions that take place between the invocation of a public method in the system and a single noticeable end result by a test of that system. A noticeable end result can be observed without looking at the internal state of the system and only through the public APIs and behavior. An end result is any of the following:

* The invoked public method returns a value
* There is a change in the state or behavior of the system
* There is a callout to a third party system over which the test has no control and that third party system does not return any value, or any return value from the system is ignored

**[Osherove2015]**

### System Under Test (SUT)

When you are testing, the thing you are testing is referred to as the System Under Test. This can also be referred to as the Class Under Test or Code Under Test.

### Unit Test

A unit test is an automated piece of code that invokes a unit of work and checks one specific end result of that unit of work being tested, and then checks some assumptions about a single end result of that unit. If the assumptions on the end result turn out to be wrong, the unit test has failed. A unit test’s scope can span as little as a method or as much as multiple classes. These tests should make up the bulk of your testing suite. These tests should run in memory, and not require any dependencies to function.

The term unit test is often misused to define manual testing with a unit of work or integration testing. I recommend sticking to the stricter definition above. These extended definitions only serve to cause the original term to lose its meaning.

### Integration Test

A test verifies if “independently developed units of software work correctly when they are connected to each other”. It is a slower test that covers a larger surface area. The size of this surface area can vary greatly. Since these tests are generally more involved and more difficult to create, there will generally be less Integration tests than Unit tests.

<https://martinfowler.com/bliki/IntegrationTest.html>

### What makes a good Unit Test?

A good unit test should be:

* Automated and repeatable
* Easy to implement
* Relevant Tomorrow
* Anyone should be able to run it at the push of a button
* Runs quickly
* Has consistent results
* Has full control of the unit under test
* Fully isolated (runs independently of other tests)
* When it fails, it should be easy to detect what was expected and determine how to pinpoint the problem

# Test Harnesses

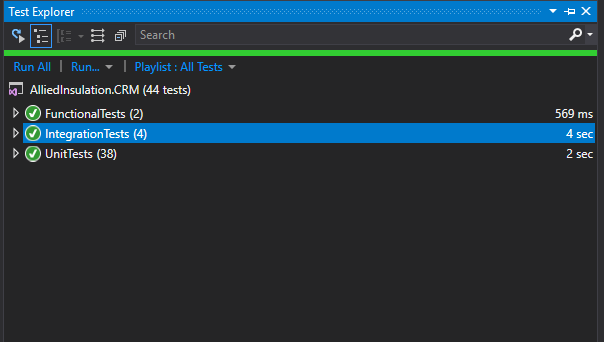
It is possible to create your own testing framework and testing harness using a command line project. You can register all your test classes in the console application and run all of your tests, and display the results in the console window. Teams used to use assembly scanning to find all classes that end in “Test” and automatically add them to the list of tests run by the harness. It is a lot of work, and thankfully in the vast majority of cases no longer necessary.

To describe automated testing with no framework, I’ll borrow a section from Roy’s book:

Doing tests and regression testing completely manually, repeating all the same actions again and again like a monkey, is error prone and time consuming, and people seem to hate that as much as anything can be hated in software development. These problems are alleviated by tooling. Unit testing frameworks help developers write tests quickly with a set of known APIs, execute those tests automatically, and review the results of those tests easily. And they never forget! **[Osherove2015]**

# Testing Frameworks

Thankfully, setting up a test environment today is easier than ever. Visual studio can automatically detect test classes and test methods and display them in the Test Explorer:



Other IDEs have similar functionality. .NET and NPM have the ability to run and display test results from the command line with:

* Npm test
* Dotnet test

Whatever IDE and whatever testing framework you are using, automated testing is ubiquitous enough that it is highly likely you will be able to get tests up and running quickly and easily.

Across all languages, there are over one hundred Test Frameworks to choose from. The examples in this course will be written in c#, so I will reference the three .NET frameworks:

* NUnit
* xUnit.net
* Microsoft Test

For the examples in this course, I will utilize xUnit. Given that this is an introductory course, we will not be running into any functional differences between the frameworks at this time. Take your time trying each framework out and see which you like best. Once you are on a team, the team will adopt one framework and use that across the board, so it cannot hurt to be familiar with all three. They are also simple enough that once you know one you can easily pick up the others.

# Naming Conventions

There are a few different naming conventions out there, but for this course we will adopt the following:

[UnitOfWorkName]\_[ScenarioUnderTest]\_[ExpectedResult]

# A First Unit Test

Let’s write a simple unit test to cover the Times method of the Dollar class. The expected behavior is quite simple. We expect that a call to the Times method will return a new Dollar instance, whose amount will be equal to the amount of the current Dollar instance multiplied by the multiplier parameter.

public class Dollar

{

public decimal Amount { get; set; }

public Dollar(decimal amount)

{

Amount = amount;

}

public Dollar Times(decimal multiplier)

{

return new Dollar(Amount \* multiplier);

}

}

We can write the following test to confirm this behavior:

[Fact]

public void Times\_GivenMultiplier\_ReturnsNewDollarWithAmountTimesMultiplier()

{

Dollar five = new Dollar(5);

Dollar product = five.Times(2);

Assert.Equal(10, product.Amount);

}

### Arrange, Act, and Assert

This test is divided into three sections:

* Arrange
  + Prepare all the necessary pieces that your test will require.
* Act
  + Perform the operation that will have a noticeable result in the system
* Assert
  + Make an assertion based on the noticeable result that has been created in the Act phase.

Almost all tests will have this structure, and it is very common to add spacing between each of these three sections so that they are clearly visible.

### The Fact Attribute

We can also introduce our first xUnit attribute, the Fact. All this attribute does is denote that the method it decorates is a test. Other testing frameworks will do something very similar (For example, NUnit decorates methods with [Test]).

### The Assert Statement

Last but not least, we can see an assertion being made. There are many different types of assertions that exist within unit testing frameworks. In this case:

Assert.Equal(10, product.Amount);

We are asserting that the expected value (10) is equal to the actual value (product.Amount). If they are indeed equal, the test will pass.

### Always Make Your Tests Fail

It is easy to make assumptions about how your code should behave, and these assumptions can come back to bite you if you do not test them. In then example above, my test passes. However, I should change the test at least once to verify that the test fails under failing conditions. For example, I might change the system under test to intentionally return an incorrect result to see if it is caught by my test:

public class Dollar

{

public decimal Amount { get; set; }

public Dollar(decimal amount)

{

Amount = amount;

}

public Dollar Times(decimal multiplier)

{

return new Dollar(Amount); //This is wrong!

}

}

Once I have verified that my test fails, I can change the SUT back to the correct state and run the test again.

This test is what is known as a value test. We have a value that is returned from a method, and we are performing an assertion based upon that value.

State Based Tests

State based tests involve testing the state of the system after an operation has been performed. In this case a value is not returned via a method (Perhaps the method is void), or we are not concerned with testing the value returned.

Perhaps in our Dollar class we also want to track if a new Dollar instance has been created using the Times method.

public class Dollar

{

public decimal Amount { get; set; }

public bool PriceAdjusted { get; set; } = false;

public Dollar(decimal amount)

{

Amount = amount;

}

public Dollar Times(decimal multiplier)

{

PriceAdjusted = true;

return new Dollar(Amount \* multiplier);

}

}

Despite the fact that we do not want to test the return value, we can still create a test that asserts against the state of a class.

[Fact]

public void Times\_GivenMultiplier\_SetsPriceAdjustedToTrue()

{

Dollar five = new Dollar(5);

Dollar product = five.Times(2);

Assert.True(five.PriceAdjusted);

}

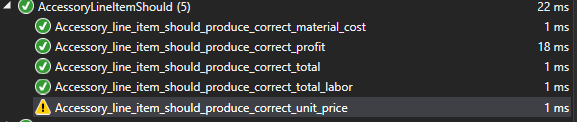
# Ignoring Tests

There may be times where you need to temporarily ignore tests. Almost all unit testing frameworks come with a method of ignoring tests, and this is vastly preferable to commenting the tests out. Commented out tests have a tendency to be forgotten.

In XUnit, this is accomplished by adding a parameter to the Fact attribute:

[Fact(Skip = "Temporarily Disabled")]

These tests will be displayed with a warning icon, so that you will remember to come back and address them later:



# Stubs

So far the code in these examples have not relied on any external dependencies, but what if we do? How can we ensure that we test our target without worrying about unrelated failures? For example, what if a method makes a call to a third party API, and that call is unrelated to what we are testing. If the API is down, do I not get to test any of the other parts of the method? Of course not. We should instead use a stub, so that we are not forced to interact with the third party API for our test.

### What is a Stub?

“A stub is a controllable replacement for an existing dependency in the system. By using a stub, you can test your code without dealing with the dependency directly.” **[Osherove2015]**

### Stub injection

We can inject a stub the same way we would inject any other dependency using the dependency inversion principle, but first let’s take a look at a new example:

### Example

public class Dollar

{

public decimal Amount { get; set; }

public Dollar(decimal amount)

{

Amount = amount;

}

public Dollar Times(decimal multiplier)

{

var newAmount = new InflationService().AdjustForInflation(Amount);

return new Dollar(newAmount \* multiplier);

}

}

Perhaps for a new requirements we want to make certain that our dollar values are always adjusted based on some inflation amount. Perhaps we are dealing with transactions from several decades ago, and we want to represent them using an adjusted value. We have to communicate to some 3rd party API in order to get the inflation information. However, we are currently not interested in testing this 3rd part api. We want to simply test the return value of Times method.

The new keyword within the scope of methods is problematic for testing. There is not way to remove the dependency on the ExchangeRateService as the code currently exists. Instead, we should inject the dependency using an interface:

public interface IInflationService

{

decimal AdjustForInflation(decimal currentAmount);

}

public Dollar Times(decimal multiplier, IInflationService inflationService)

{

var newAmount = inflationService.AdjustForInflation(Amount);

return new Dollar(newAmount \* multiplier);

}

With this interface in place, we can create a handmade stub, that we can then use for testing:

public class InflationServiceStub : IInflationService

{

public int ValueToReturn { get; set; } = 1;

public decimal AdjustForInflation(decimal currentAmount)

{

return ValueToReturn;

}

}

Our resulting test looks like this:

[Fact]

public void Times\_GivenMultiplier\_ReturnsNewDollarWithInflationAdjustedAmountTimesMultiplier()

{

InflationServiceStub stub = new InflationServiceStub();

stub.ValueToReturn = 10;

Dollar five = new Dollar(5);

Dollar product = five.Times(2, stub);

Assert.Equal(20, product.Amount);

}

We can now safely test the class without worrying that the third party API will cause our test to fail unexpectedly. You’ll notice that the name of our method has become quite lengthy at this point. This is fine. Tests should be as descriptive as possible. Readability of tests is an incredibly important aspect.

Interaction Testing / Behavior Verification

What if we needed to test that our application attempted to call a third party service given a condition? If this call to the third party service does not return a value or change the state of the system in a noticeable way, we have to use an interaction test.

“Interaction testing is testing how an objects sends messages (calls methods) to other objects. You use interaction testing when calling another object is the end result of a specific unit of work.” **[Osherove2015]**

# Mocks

With interaction tests, since not noticeable change occurs within the system, we have to take a different approach when making assertions. We have to use mocks in order to find out if the call to the other class was made.

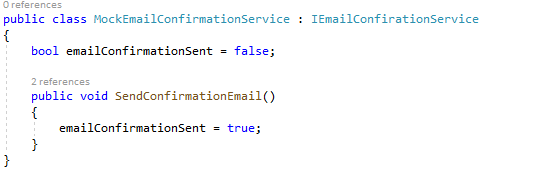
“A mock object is a fake object in the system that decides whether the system under test has passed or failed. It does so by verifying whether the object under test called the fake object as expected. There’s usually no more than on mock per test.” **[Osherove2015]**

Let’s take a look at an example:

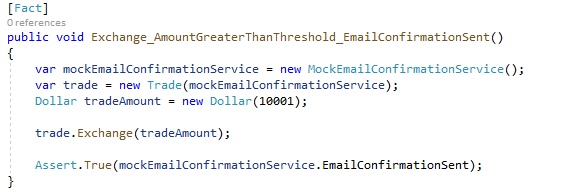
# 

In the above example, an email is sent to an email confirmation service if a trade is made exceeding a certain threshold. We need to create a mock that will allow us to determine if the emailConfirmationService was called.

Creating this Mock is very simple:



From here we can create the corresponding test, and wala, we have confirmed that the method was called:



# Interaction Test Caveats

Interaction tests, by nature, are harder to maintain than value and state based tests. You are reaching into the internals of another class and verifying that something internal to this class has happened. This is somewhat encapsulation breaking. You should always use this type of test as a last resort, when you cannot use a value or state based test to perform the assertion.

Also, be very careful using Mocks, and make certain that your mocks are not becoming too complex. If your mocks start becoming very complex, it may be a sign that you need to rethink part of your application design. Try to keep to one mock per test as well. If you find yourself having multiple mocks, you may be testing too many things.

<https://osherove.com/blog/2008/7/12/over-specification-in-tests.html>

I also highly recommend this article by Martin Fowler regarding State Based vs Interaction Testing, and Stubs vs Mocks.

<https://martinfowler.com/articles/mocksArentStubs.html>

# Problems with Handwritten Mocks and Stubs

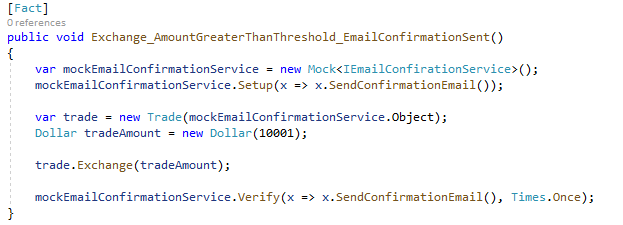
As you may suspect, these stub and mock classes can quickly grow like a plague. They also have to be maintained, and it can be difficult striking a balance between reuse and maintainability.

Handrolling mocks and stubs is also time consuming. We have to create many new classes and take the time to write them out, where to put the file, what to call the class, so on and so forth.

Mock frameworks make this tremendously easier. There are times (rarely for me) where you may still find value in creating handrolled mocks, however the ease of use mocking frameworks can provide is massive.

# Mock Frameworks

Let’s take a second look at the above tests, this time using a popular mocking framework, Moq.



Using the code above, I no longer need any additional classes. The MockEmailConfirmationService is no longer required. Let’s break down exactly what changed.



In the first two lines, we create a new Mock of type IEmailConfirmationService. The Moq framework will dynamically create a Mock class behind the scenes for us. In the next line, we call the SetUp method for SendConfirmationEmail. This is letting the Moq framework know that we are going to make an assertion against the SendConfirmationEmail method.



When we pass in our dependency to the Trade class constructor, we pass in the object property of the Mock. This object is the dynamically generated class that Moq created for us.



Finally, we ask Moq to make an assertion that the method SendConfirmationEmail was called exactly one time.

Mocks are extremely customizable, but I would be careful in making your Mocks too complex. Complex mocks generally make a test harder to read and maintain.

To find out more about Mocks and the moq framework, I recommend checking out the Moq quickstart guide:

<https://github.com/Moq/moq4/wiki/Quickstart>

Or taking this course on pluralsight:

<https://app.pluralsight.com/library/courses/nunit-moq-mocking/table-of-contents>

# Additional Resources

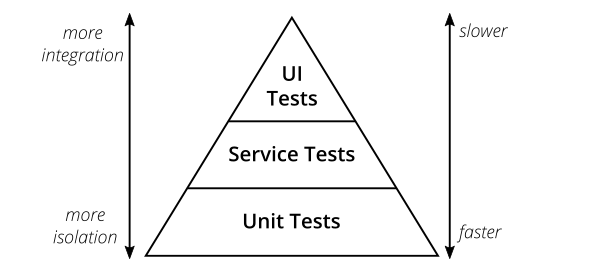
The below youtube video contains a talk by Misko Hevery at the Google Tech Talks regarding unit testing. It is only around 30 minutes, and I recommend giving it a watch.

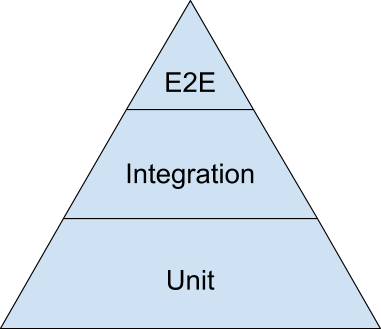
<https://www.youtube.com/watch?v=wEhu57pih5w>

Types of Tests

Now that we have covered the fundamentals of Unit Tests, I want to take a step back and discuss different types of tests, and the role each plays within a test suite.

You will often see these tests displayed as a Test Pyramid, representing the difficulty of writing and maintaining the tests to how many of the tests are contained within a test suite. These pyramids can differ in appearance, but the general idea is to write less tests that take a long time to run and cover large surface areas, or tests which are likely to change frequently (Such as UI tests).





There are more types of tests, such as Contract Tests, End to End tests, Spike tests, and Exploratory tests. These tests are not within the scope of this document.

It is important to keep to this type of testing structure. While end the end tests can be valuable, they are often flaky. Anything going wrong along a very large path can cause the test to fail. Many times the failure will not necessarily be a sign that the code is not working correctly. When an end to end test does fail, it can be difficult to ascertain the reason why. You must search the entire area of code covered by the test to find the failure.

<https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html>

This is in stark contrast to unit tests, which only cover a very specific and small section of code. If my unit test fails, I should immediately know the affected area, and why the failure occurred.

All of these types of tests exist on a continuum. There is not a clearly defined point where a unit test becomes an integration test and a integration test becomes an end to end test. That being said there are clear examples of each.

* If a test only interacts with a single class, runs exclusively in memory, and stubs and mocks dependencies, the test is clearly a unit test
* If a test tests the interaction between two classes and how they interact with the file system through an interface, it is an integration test.
* If a test covers an entry point in the UI and goes all the way to the database and back again, it is an end to end test.

Even when the lines get a little blurry between these test types, it is still important to roughly stick to a pyramid type structure. If you test suite becomes slow, or many of your tests are flaky and fail for suspect reasons, it may be a sign that you need to take a look at the composition of your test suite.

You may choose the break these tests up and only run certain tests at certain times depending on various factors. For example, since unit tests are very fast, you may choose to separate them from the integration tests and end to end tests. Developers can run all the unit tests as they develop and get quick results back. You may choose to only run end to end tests when a check in is made. Regardless, it is important that your test suite have some performance considerations. If you unit tests become too slow, your developers will stop using it as a tool.

<https://martinfowler.com/articles/practical-test-pyramid.html>

# Caveats on Test Coverage

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

<http://www.exampler.com/testing-com/writings/coverage.pdf>

<http://www.developertesting.com/archives/month200705/20070504-000425.html>

Test Certified

<https://mike-bland.com/2011/10/18/test-certified.html>

# References

1. **[Beck2015]** Beck, Kent, and Cynthia Andres. *Extreme Programming Explained: Second Edition*, Embrace Change. Addison-Wesley, 2015
2. **[Beck2014]** Beck, Kent. *Test-Driven Development by Example*. Addison-Wesley, 2014.
3. **[Google2016]** “Re:Work.” *Google*, Google, <https://rework.withgoogle.com/print/guides/5721312655835136/>.
4. **[Bland2014]** “Goto Fail, Heartbleed, and Unit Testing Culture” Bland, Mike, <https://www.martinfowler.com/articles/testing-culture.html>.
5. **[Puppet2017]** “2017 State of Devops Report”, <https://puppet.com/resources/whitepaper/2017-state-of-devops-report/thank-you>
6. **[Humble2017]** “Secrets Of High Performing Teams: Science Edition” Humble, Jez, <https://gotocon.com/dl/goto-cph-2016/slides/JezHumble_SecretsOfHighPerformingTeamsScienceEdition.pdf>.
7. **[Osherove2015**] Osherove, Roy, et al. *Art of Unit Testing*. 2015.

# Resources

**Skillport:**

[https://cgi.skillport.com/skillportfe/main.action#summary/COURSES/CDE$130120:\_ss\_cca:sd\_adev\_a07\_it\_enus](https://cgi.skillport.com/skillportfe/main.action%23summary/COURSES/CDE$130120:_ss_cca:sd_adev_a07_it_enus)

[https://cgi.skillport.com/skillportfe/main.action#summary/COURSES/CDE$121695:\_ss\_cca:wd\_dmvc\_a04\_it\_enus](https://cgi.skillport.com/skillportfe/main.action%23summary/COURSES/CDE$121695:_ss_cca:wd_dmvc_a04_it_enus)

**Pluralsight:**

<https://app.pluralsight.com/library/courses/play-by-play-wilson-tdd/table-of-contents>

<https://app.pluralsight.com/library/courses/pragmatic-unit-testing/table-of-contents>

<https://app.pluralsight.com/library/courses/writing-highly-maintainable-unit-tests/table-of-contents>

**Books**

TDD By Example

The Art of Unit Testing