Automated Testing and TDD

# About 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. These tests will return a result, which will be a success or failure based on one or many conditions.

# Benefits of Automated Testing

### 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

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 valuable, and be carefully evaluated from a value, cost, and risk perspective.

We can now confidently refactor legacy code knowing that if all of the tests pass that our software behaves as expected.

### 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”

<https://www.computer.org/csdl/magazine/so/2007/03/s3024/13rRUygT7kK>

<https://www.testingexcellence.com/pros-cons-test-driven-development/>

<https://medium.com/crowdbotics/tdd-roi-is-test-driven-development-worth-the-money-d535c8d5a5f>

<https://devblogs.microsoft.com/premier-developer/pragmatic-tdd/>

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

In 2016, Google published the findings of its Aristotle Project, an investigation into what makes an effective team. The project defined what a team was, what effectiveness was, and then sought out to collect data and measure effectiveness.

The researchers identified the following dynamics, listed in order of importance:

1. Psychological Safety
   1. Team members feel safe to take risks and be vulnerable in front of each other
2. Dependability
   1. On dependable teams, [members reliably complete quality work on time](http://amj.aom.org/content/53/3/535.short) (vs the opposite - [shirking responsibilities](http://www.jstor.org/stable/pdf/258490.pdf?acceptTC=true))Structure and Clarity
3. Structure & Clarity
   1. [An individual’s understanding of job expectations, the process for fulfilling these expectations](http://www.jstor.org/stable/1556372), and the consequences of one’s performance are important for team effectiveness. Goals can be set at the individual or group level, and must be specific, challenging, and attainable. [Google often uses Objectives and Key Results (OKRs)](https://rework.withgoogle.com/guides/set-goals-with-okrs/steps/introduction/) to help set and communicate short and long term goals.
4. Meaning
   1. Finding a sense of purpose in either the work itself or the output is important for team effectiveness. The meaning of work is personal and can vary: financial security, supporting family, helping the team succeed, or self-expression for each individual, for example.
5. Impact
   1. The results of one’s work, [the subjective judgement that your work is making a difference, is important for teams](http://www.ncbi.nlm.nih.gov/pubmed/18211139). Seeing that one’s work is contributing to the organization’s goals can help reveal impact.

**[Google2016]**

Automated testing doesn’t hit on all of these points, but it does hit on the top two in a large way. By writing valuable tests that validate the behavior of our software, we have a net of psychological safety. By having a team that exercises this discipline, we can all depend on one another regarding the structure, correctness, and quality of our software.

### Improved Work Practices

Having fewer defects really empowers a team to take new and interesting approaches. We can now:

* If the defect density can be reduced enough, then quality assurance can shift from reactive work to proactive work
* If the defect density can be reduced enough, we can have shippable software with new functionality every day, leading to new business relationships with customers
* 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.

### 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.

### Clean, Decoupled Code

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.

# About TDD

### What is 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

### Red, Green, Refactor

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**

### TDD and Challenging 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.”

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

### Decoupled Code

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.

### TDD and Evaluating Assumptions

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.

### TDD is Satisfying

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 Testing Trade Offs**

### 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.

### 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.

# **TDD Trade Offs**

TDD has become dogmatic and moralistic

### 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/>

The above link contains about 2 ½ hours of discussion between David Heinemeier Hansson, Martin Fowler, and Kent Beck on TDD. I highly recommend giving it a watch. It challenges several assumptions about TDD and has some great value based discussions about TDD and automated testing.

### 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.

### 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

### TDD can result in

As humans it is tempting to try and quantify things. Quantifying the quality of a design is really difficult. We may tend to focus on things that are easily measurable but less important than things that are more important but not as easily measured.

Make certain to constantly evaluate trade offs

### TDD is not how everyone’s brain works

The ability to write tests for anything is a valuable skill to have. It is the four wheel drive of programming. If I get into a really sticky spot, I can always ratchet up the tests.

At the end of the day, the principles and practices and decisions are up to the team

# **Types of Automated Tests**

## Unit Tests

* A small, fast test designed to cover a single unit of work, and make a single assertion about how that unit of work behaves. These tests should make up the bulk of your testing suite. These tests should run in memory, and not require any dependencies to function.

## Integration Tests

* A test verifies if “independently developed units of software work correctly when they are connected to each other”.[1](#IntegrationTest)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>

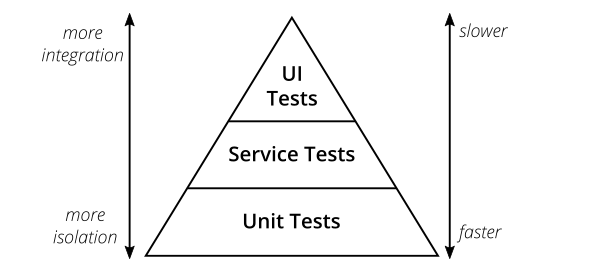
## End to End Tests

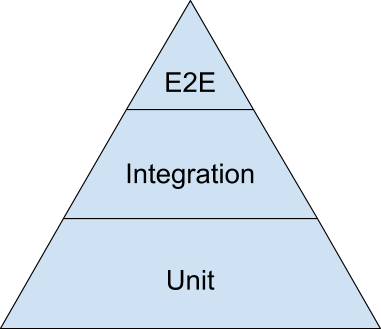
* A test that covers large sections of the system, form the start point to the end point. You ensure that your entire application and its dependent systems integrate correctly. They closely mimic how a real user would use the application.

## UI Tests

* UI testing involves verifying that UI elements perform correct behaviors. Since UIs change frequently, these tests can be expensive to maintain. A test suite will usually have less UI tests than integration tests.

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.

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

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

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

# Types of Unit Tests

Unit tests are generally divided into two categories. State verification and behavior verification.

# Mocks and Stubs

Mocking and stubbing are tools that are used to remove dependencies from tests.

**Mock:** An implementation of an interface or base class that has some predefined behavior

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

# References

1. Beck, Kent, and Cynthia Andres. *Extreme Programming Explained: Second Edition, Embrace Change*. Addison-Wesley, 2015
2. Beck, Kent. *Test-Driven Development by Example*. Addison-Wesley, 2014.
3. “Re:Work.” *Google*, Google, rework.withgoogle.com/print/guides/5721312655835136/.