Automated Testing and TDD

# About TDD

### What is test driven development?

* Predictable way to develop
* Gives you a chance to learn all the lessons that the code has to teach you. If you only slap together the first thing you think of, then you never have time to think of the second, and better, thing.
* Improves the lives of the users of your software
* It lets your teammates count on you, and you on them
* It feels good to write it.

### Rules of TDD:

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

We must design organically, with running code providing feedback between decisions

We must write our own tests, because we can’t wait 20 times per day for someone else to write a test

Our development environment must provide rapid response to small changes

Our designs must consist of many highly cohesive, loosely coupled components

### Red, Green, Refactor

* If the number of nasty surprises can be reduced enough, then project managers can estimate accurately enough to involve real customers in daily development
* If the topic of technical conversations can be made clear enough, then software engineers can work in minute-by-minute collaboration instead of daily or weekly collaboration

# **Benefits**

### 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.

### 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.

### TDD and Courage

What does fear do to a software person?

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

TDD 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 and take smaller steps to build confidence. Once we feel we are on solid ground again, we can take larger steps with our tests.

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

### TDD and Defects

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]**

Test Driven Development doesn’t hit on all of these points, but it does hit on the top 2 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 TDD, we can all depend on one another regarding the structure, correctness, and quality of our software.

**TDD and**

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

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.

### TDD and Defect 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.

### TDD and 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.

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.

# **Trade Offs**

TDD has become dogmatic and moralistic

### Tests Must be Maintained

### Encourages Testing Extremes

* Costs Rise as you approach an Asymptote
* Cost/Value Analysis not being applied
* If you say “we need 100% test coverage”, and also say that we make tradeoffs when it comes to criticality, clearly you do not

### Test Induced Design Damage

* Lots of Indirection
* Lots of Mocking
* Who cares about swappability when things will never be swapped
* Faith based TDD – “I will eventually get to the right design that is easily testable but also works”

### Overtesting

* Too many tests increase maintenance burden on the system

### Refactoring Can Be Skipped

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

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

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 Tests

## Unit Tests

* State Based Verification
* Behavior Based Verification

## Integration Tests

## Contract Tests

## End To End Tests

## UI Tests

## Acceptance Tests

## Exploratory Testing

# References

**[Beck2015]** Beck, Kent, and Cynthia Andres. *Extreme Programming Explained: Second Edition, Embrace Change*. Addison-Wesley, 2015

**[Beck2014]** Beck, Kent. *Test-Driven Development by Example*. Addison-Wesley, 2014.

**[Google2016]** “Re:Work.” *Google*, Google, rework.withgoogle.com/print/guides/5721312655835136/.