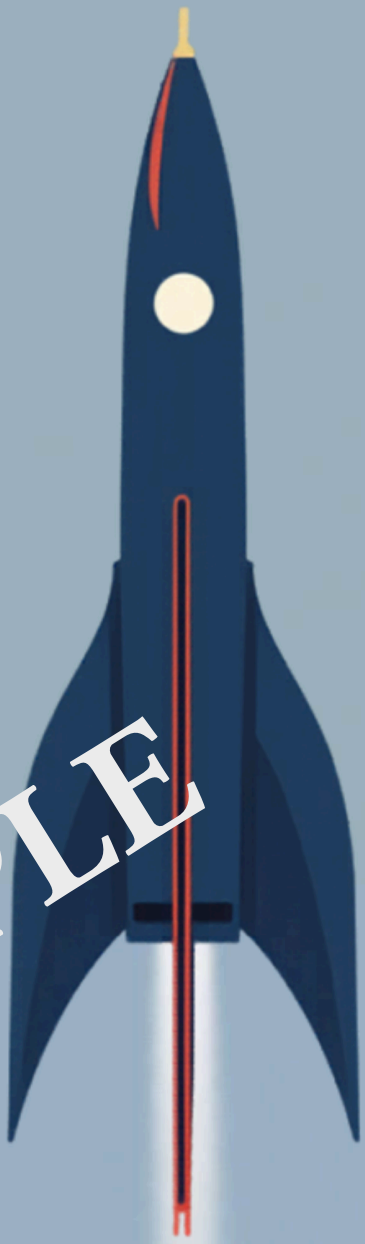
Lean TDD

TDD Without the Waste



SAMPLE

Brian Okken

Contents

This sample includes Chapter 1 in full. The complete book also covers:

Chapter 1

Why Lean TDD?

TDD is awesome. That's a good place to start.

I learned about Test Driven Development early in my career as a software engineer. I'd like to say that I saw an immediate productivity boost. But the benefits were more gradual. Learning to effectively include automated testing into the software development process definitely did have massive positive effects, but the transformation wasn't quick, and it required incorporating other practices to get the full benefit.

The general idea of TDD immediately made sense to me. I had by then been on a handful of development teams and had been through several software projects. These were all in the domain of RF and communication test instruments and systems. All of the projects followed some form of the traditional SDLC, Software Development Life Cycle, with most of the QA happening at the end of the cycle. But testing at the end causes problems.

Problems like not knowing how long the test and fix phase is going to be, having schedule slips, having to quickly rewrite a part of the system, and just generally a stressful not-fun time at the end of each development cycle.

When QA starts testing, they find issues. That's expected. Then bug reports are filed. The reports are triaged and prioritized and assigned. The reports need to be evaluated to decide if the problem is in the test code or the production code. Any fixes change the code, so we have to retest. And the fix might break something else, and that gets reported. Rinse, repeat.

I remember thinking as a developer that a lot of that process and pain could be eliminated if we had the tests available during development. And it does improve things.

A big speedup happens when the development team can run the failing tests when diagnosing and fixing the issue. Even better if they can run the whole suite. That keeps the whack-a-mole of fixes causing other problems within the development team and saves handoff time.

Even more speedup happens when the development team is involved in defining and writing the tests, and that brings us to Test Driven Development.

TDD seems reasonable. We can start writing tests at the same time as the production code. And whenever we get a test to pass, we make sure we don't muck it up during future feature development.

That's kinda what I thought TDD was when I first heard of it. But that's not what most people learn when they learn TDD. Most versions of TDD are not about those "make sure the system works" tests and are more focused on individual developer "did my code do what I thought it should do" tests. This ends up causing a duplication of effort between developer tests and system validation tests.

There are a couple of exceptions. BDD, Behavior Driven Development, and ATDD, Acceptance Test Driven Development, focus more on the system verification tests. However, proponents of ATDD and BDD still frequently recommend more traditional forms of TDD for the developer focused tests. So we still have the duplication of effort.

Lean TDD aims to remove this duplication of test effort. And while we're at it, we'll try to remove any other wasted effort we find.

To build Lean TDD, I've pulled ideas from Test Driven Development by Example[1], Lean Software Development[2], The Pragmatic Programmer[3], and tons of other great books and articles to create a software philosophy and practice of continuous improvement, adaptable processes, clear communication, early feedback, and, at the center of it all, incorporating automated tests into the entire software development workflow.

Lean TDD is a very effective, efficient form of TDD that translates well to all kinds of software development. It works with large teams down to solo developers.

[1] *Test Driven Development by Example*, Kent Beck, Addison-Wesley, 2002.

[2] *Lean Software Development: An Agile Toolkit*, Mary Poppendieck and Tom Poppendieck, Addison-Wesley, 2003.

[3] *The Pragmatic Programmer: From Journeyman to Master*, Andrew Hunt and David Thomas, Addison-Wesley, 1999.

This book is about Lean TDD, the flavor of TDD that has worked best for me. But I think the reason it makes sense to me is also due to all of the study I've done in other practices. The ideas from many flavors of TDD and the lessons from Lean in finding and reducing waste all come together to form Lean TDD. I want to give you a crash course in that background study. That way, even if you don't follow my logic and come to my same conclusion that Lean TDD is the obvious winner, you'll have enough of a head start on other material to come up with what's best for you.

Why TDD Still Gets Resistance

It's been over 25 years now since Kent Beck published his first book on TDD, but lots of developers still don't run any of the tests. And even less actually write tests. Why not?

First off, TDD is not just one practice. We use one name, but the different ways people interpret TDD are wildly different.

Another problem is that the tutorials often focus on a single class or function, but don't go on to describe how to apply TDD to a whole project.

Most tutorials seem like they're for single developer projects, and most corporate projects are not just one developer. How do we do TDD with a team? Is the development team writing the tests? Or is there a QA team? All of these questions influence how well TDD seems to fit for different projects.

There's also this idea that crops up that using TDD will slow you down. I have heard more than one supposed pro-TDD person say something like "yeah, it takes more time and it's more work up front, but it's worth the effort for better quality." That kind of a statement doesn't engender confidence or excitement. Really? Takes more time and is more work up front? Most developers stopped listening after "it takes more time and more work". Most people are trying to get more done in less time, so any process that is billed as "slower, but probably worth it" is just not gonna cut it.

TDD Is Not Slower

But TDD is NOT slower. If it is, maybe you're doing it wrong.

A developer recently told me they didn't have time to write tests. That if they started writing tests they'd have to add that to the schedule and deliver later. But when someone writes some code, they don't usually just hope it

works and check it in. Without automated tests, you're left with manual testing.

Here's a not-atypical development workflow without tests:

- write the code you think you need to write
- try it out
- maybe poke some buttons, walk through a part of a user workflow
- or if there's no UI yet, write some code to drive the thing you're working on
- get the code to a point where you think it's ok
- check it in
- throw away your little driver code
- wait for someone else to write an automated test for the feature
- get bug reports days or weeks later
- if you can run the test QA is running, cool, but probably not
- try to reproduce the bug manually
- attempt a code fix
- push your fix and notify QA
- hope for the best

There's no way that's faster than just spending a few minutes writing some tests during development. That manual stuff and throwing stuff back and forth to QA is what I don't have time for.

If I'm going to write a bit of code to drive my code under test, why not just make it a test function? And when done with the whole process, check in the test code and production code together. There's no way that's slower. And you don't need to write a complete test all at once. Incrementally writing the test code and production code together is the real sweet spot. And that incremental building of test and production is one of the core ideas of TDD.

Think First

There is a mind shift you need to go through to get the most benefit. The mind shift helps even with a single developer, but really gives you time back if your entire team goes through the mind shift.

What's the shift? The shift is thinking about "how will I know if this is working". It's not extra time. You always have to figure that out. If you do that thinking beforehand, you can use it to help you code faster and arrive at a clean design faster.

Thinking about the question of "how will I know if this is working" and writing that down as an automated test before you start coding is "test first", and it turns out to be that crucial difference that gets you to the goal of coding faster with better quality and less stress.

Tests Are Reusable Tools

When you write a test first, it's a tool you can use. When you write a test after, it's a chore.

It's the chore of "test after" that drives many development teams to want an external QA team to test their code. Even if a QA team or test engineer is involved, having developers in the process is where the huge gains come from.

The trick is to get development teams involved without slowing down development. That's where Lean TDD comes in.

Over the years, TDD has been taught in ways that actually do make it slower, with practices like rigid micro-steps, excessive mocking, testing every function in isolation, testing at low levels when a higher level test will work fine, and duplicating effort across system tests, component tests, and unit test. Those practices introduce waste and slow you down. Lean thinking, with its focus on identifying value and eliminating waste, gives us the tools to figure out which parts of the various TDD interpretations to keep and which to discard.

The Lean Part

In this book we're going to look at various interpretations of TDD, look for the value, and look for waste, using the lens of Lean Software Development. That's where the "Lean" in "Lean TDD" comes from.

The concepts in the book will work in lots of work environments:

- if you are new to coding or an experienced developer
- if you are on a team or a solo developer
- if the development team is doing all the testing, or if you have both developers and test engineers
- if you are using AI coding agents to help with production code, test code, both, or neither

That last point matters more than ever. AI can write code, and even write tests, faster than any of us. But someone still has to decide what "correct" means, and someone still has to trust the result enough to ship it. Lean TDD gives you that framework, whether the code in front of you came from a teammate, from your own fingers, or from an AI agent.

If I convince you to try Lean TDD, awesome, give this book to a fellow developer so they too can code faster and have more fun. And if I don't convince you, it's not a total loss. Along the way you'll learn about different approaches to TDD and pick up some Lean thinking, and that'll help your career regardless.

Roadmap for This Book

Here's the plan. We'll start with some history of how we got here, then cover TDD, then Lean and its concept of waste. From there we'll look at the many TDD interpretations that have emerged, evaluate them, and build up to Lean TDD, what I think is the most pragmatic approach. We'll also talk about working with AI, and applying Lean TDD in teams.

By the end, you won't just have opinions about TDD. You'll have a practical, grounded approach you can defend to a skeptical teammate, apply on your very next feature, and keep adapting as your team and your tools, AI included, keep changing.

Keep Reading

If Chapter 1 resonated with you, the full book goes deeper into pragmatic TDD, Lean thinking, team workflows, and working effectively with AI.

Get the full book:

Amazon

leantdd.com

This sample includes Chapter 1: Why Lean TDD?