The Essence of TDD

Design Discovery Through Tests

Felix Asher

Preface

Test-Driven Development is often introduced as a simple loop:

Write a test.
Make it pass.
Refactor.

Many teams follow this loop faithfully.

They write tests first.
They keep coverage high.
Their CI stays green.

And yet, something feels wrong.

Design discussions stall.
Refactors feel risky despite extensive tests.
New features accumulate complexity instead of clarity.

This book starts from that discomfort.

It is not an attempt to redefine TDD as a better testing technique,
nor to propose a new framework or methodology.

Instead, it explores a different question:

What if TDD is not about producing tests at all?
What if it is about discovering what must hold for a system to exist?

Chapter 1: The Premise Shift

Most misunderstandings of TDD come from a subtle premise:

> Tests exist to prove that code works.

This premise is not false — but it is incomplete.

Code can work while the system remains conceptually unstable.
Functions can behave correctly while invariants quietly erode.
Components can pass tests while assumptions remain unexamined.

In real systems, failures rarely come from a single function behaving incorrectly.
They come from mismatched expectations between parts of the system.

- When is something considered “done”?
- What must never happen?
- Which assumptions are safe, and which are still provisional?

Traditional interpretations of TDD rarely surface these questions explicitly.

They optimize for local correctness,
but leave global meaning implicit.

The premise shift of this book is simple:

The primary value of TDD is not verification,
but clarification.

Chapter 2: A Definition of Real TDD

If TDD is not primarily about testing, what is it?

In this book, TDD is treated as:

> A process for discovering, validating, and retiring design assumptions.

Under this definition:

- A failing test does not merely indicate missing implementation.
- It signals that an assumption has not yet been made safe.
- A passing test does not merely indicate correctness.
- It marks one assumption as “no longer questionable.”

This reframes the familiar cycle:

Red

Not “the test fails,” but
“the system does not yet hold together.”

Green

Not “the implementation works,” but
“one assumption has been stabilized.”

Refactor

Not cleanup, but
an observation phase — looking for the next assumption likely to break.

Seen this way, TDD is not linear progress.
It is gradual convergence.

Chapter 3: Why Vertical Slice First

Most teams practicing TDD start with small units.

A class.
A function.
A module.

This feels safe.
It feels disciplined.

But unit-level correctness does not tell you whether a system is viable.

Horizontal slices answer questions like:

- Does this function return the correct value?
- Does this component handle its inputs correctly?

They do not answer:

- Does this behavior still make sense when the system is complete?
- Are assumptions preserved across boundaries?
- What breaks when time, concurrency, or partial failure are introduced?

A vertical slice cuts through the system end-to-end:

- Interface
- Domain logic
- Persistence
- Integration boundaries

It is the smallest slice that can reveal whether an assumption holds in reality.

Vertical-slice-first testing is not about integration for its own sake.
It is about confronting the system with the conditions under which it must survive.

Only after that confrontation does it make sense to refine internals.

Chapter 8: The Danger of “TDD Cosplay” (Excerpt)

Many teams practice what looks like TDD.

Tests are written first.
The red–green–refactor cycle is followed.
Coverage metrics improve steadily.

And yet, nothing important seems to be settling.

This phenomenon will be referred to here as TDD Cosplay.

TDD Cosplay is not caused by laziness or ignorance.
It is usually caused by focusing on the *form* of TDD
while losing sight of its *function*.

Common symptoms include:

- Tests increase, but design discussions repeat.
- CI stays green, but refactors feel dangerous.
- Failures occur in production that “should have been impossible.”

In these situations, tests are doing work — but not the right work.

They verify behavior that was already assumed correct,
instead of challenging assumptions that were never examined.

The result is a comforting illusion of safety.

Yet nothing fundamental has been settled.

This sample ends where most teams stop learning.