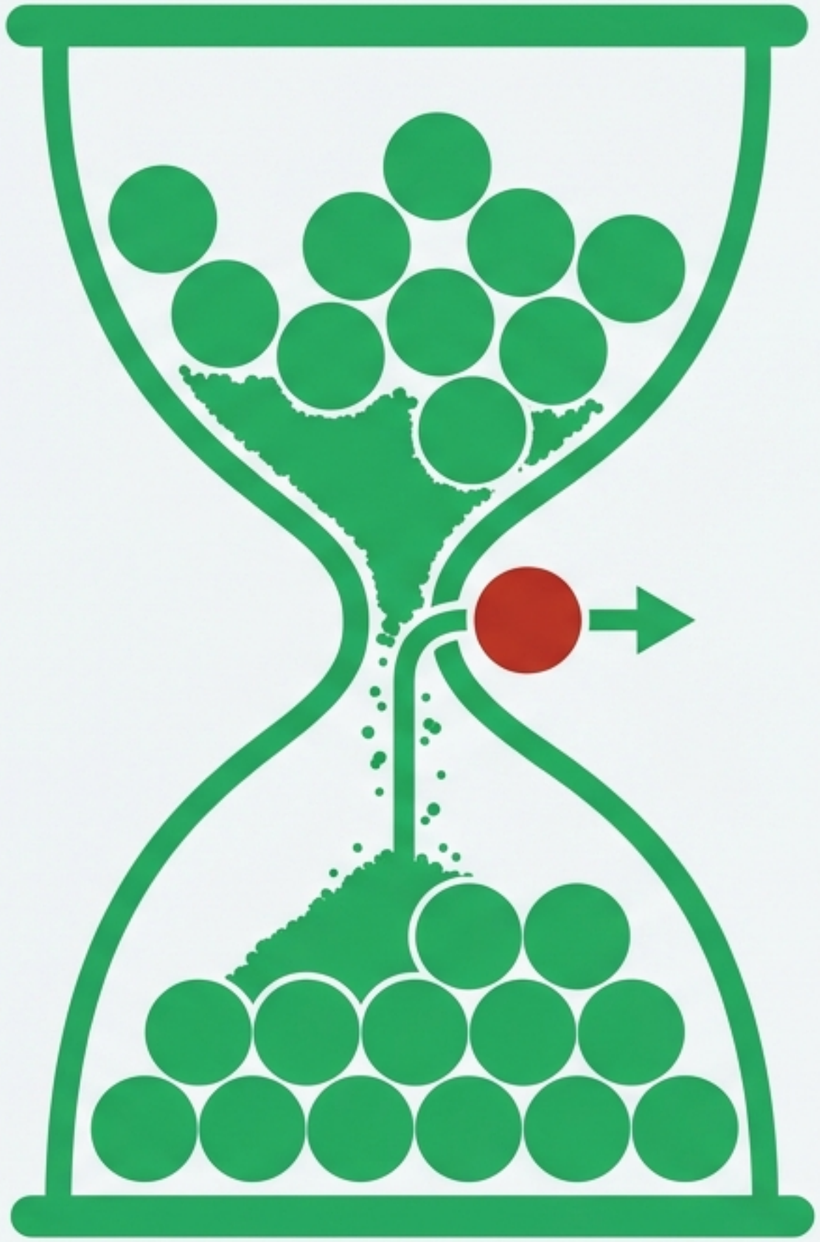
Pass-Driven Development

Turning TDD Upside Down for Robust, Evolving Software



A STANDALONE GUIDE BASED ON THE WORK OF DINIS CRUZ

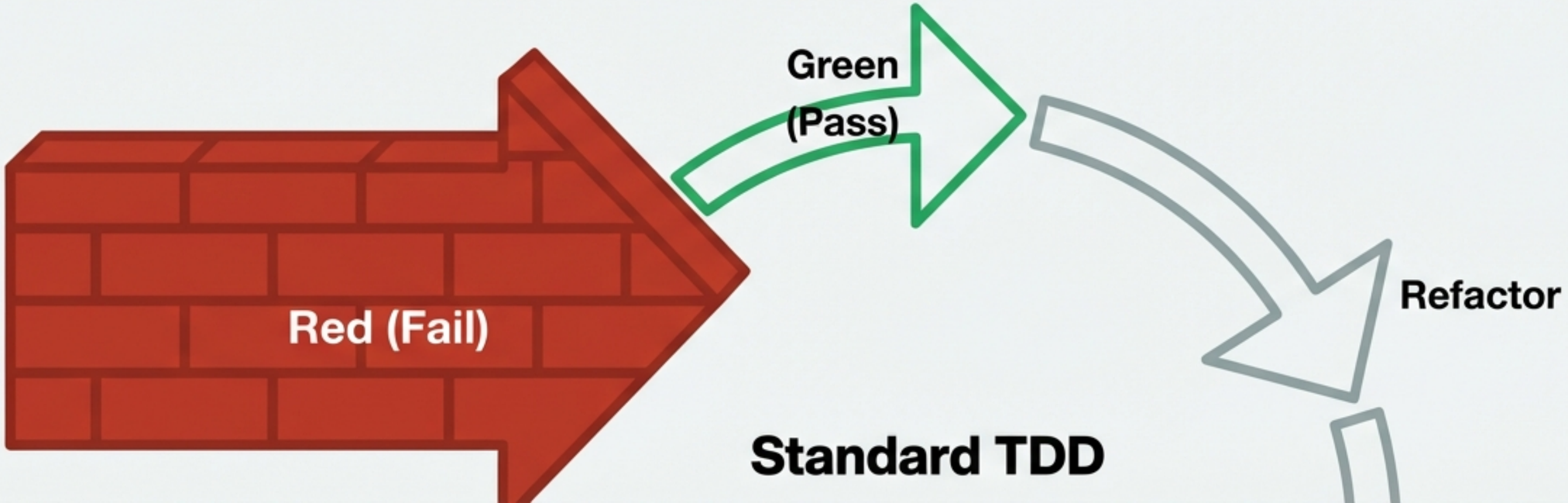
The Ideal of TDD vs. The Reality of Discovery

Test-Driven Development (TDD) is built on a sound principle: write a failing test (Red), make it pass (Green), then Refactor. It assumes you know the destination before you start the engine.

The Conflict

In practice, development is rarely linear. When exploring a new feature, the final design often emerges during coding. Upfront tests can become obsolete constraints.

Writing a failing test for complex logic often fails for the wrong reasons— setup issues or missing prerequisites— before asserting the actual behaviour.



The Cost of ‘Failing First’

Breaking the Build

TDD inherently creates a temporary broken state. In complex features, this “Red” state can persist, preventing commits to the main branch and violating the core DevOps tenet of a healthy CI pipeline.

Fragile Beginnings

Initial tests often cover specific, imagined outcomes. If the implementration path changes, the test acts as a false negative, confusing the debugging process.

Perception of Tax

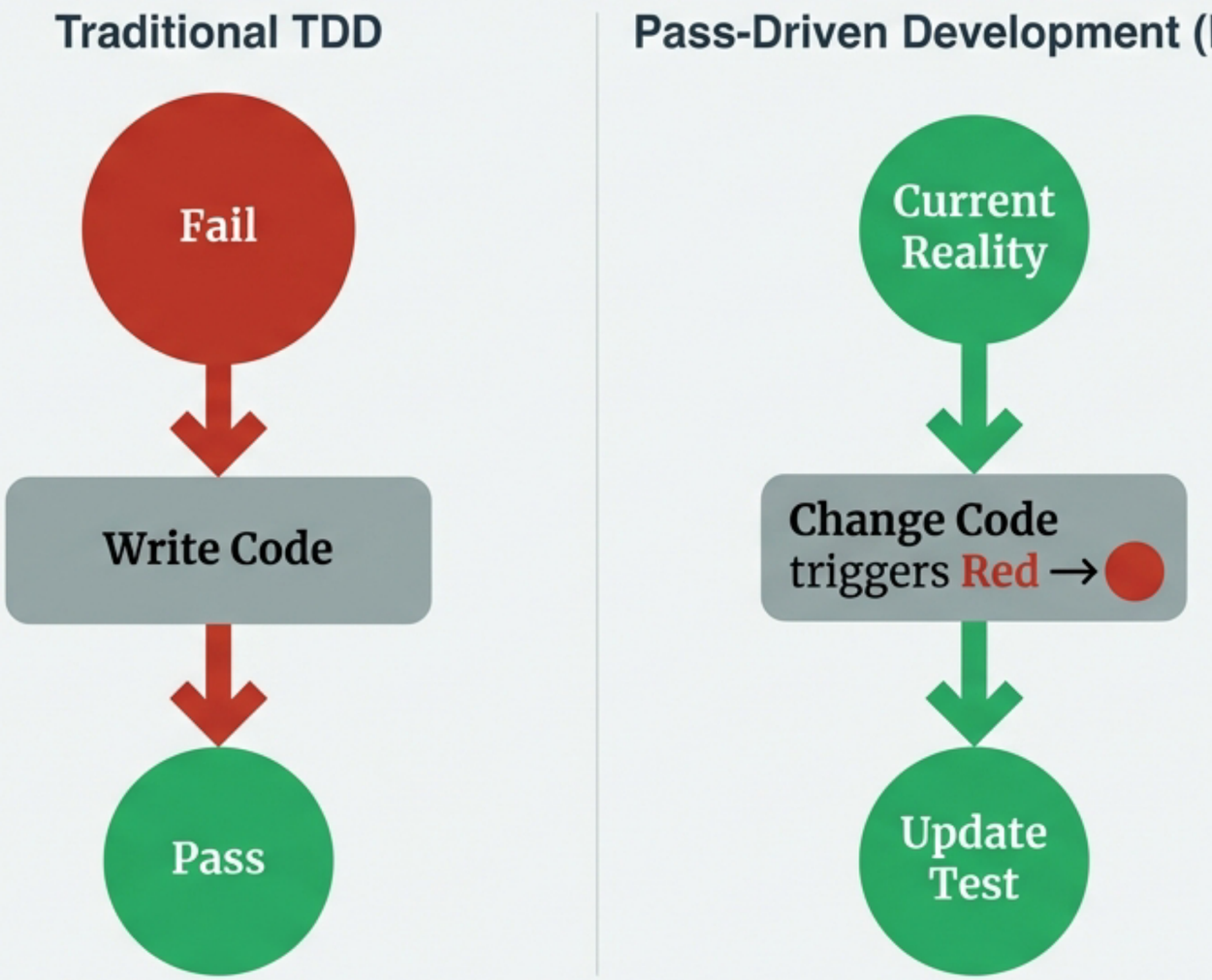
Without immediate green lights, TDD can feel like overhead—a tax paid in speed for uncertain future benefits.

‘TDD assumes the developer knows the final desired behaviour upfront... which is not always the case in exploratory development.”

Inverting the Cycle

[redacted]

Pass-Driven Development (PDD) is a workflow where tests are written to pass from the outset, mirroring the system's current behaviour—even if that behaviour is incomplete or buggy.

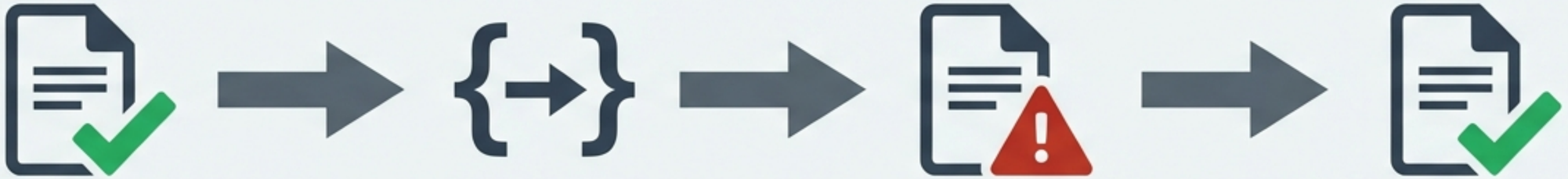


Key Insight: Tests are not a future target; they are a present mirror.

Tests as Living Documentation

In PDD, a failing test is not a prerequisite; it is a **progress indicator.**

The Evolution Cycle



Test describes reality (Green) → Code improves → Test fails: mismatch (Red) → Test updated to new reality (Green)

This makes the test suite a dynamic specification. Once behaviour is merged—whether a feature or a bug—it is a “fact” of the system. The test suite preserves this history.

The Mechanism: The PDD Feedback Loop

In PDD, ‘Red’ is not a prerequisite; it is a signal of progress.

ANCHOR	MOVE	ALIGN
Write a passing test for what the code does right now. If the feature is missing, assert it returns void or a placeholder.	Change the production code (add feature/fix bug). The test now fails because behaviour has diverged from the anchor.	Update the test to match the new correct behaviour. The test suite is now in sync with the new reality.
Green (Pass) ✓	Red (Fail) ✗	Green (Pass) ✓

Crucial Distinction: You do not write a test expecting a future behaviour.
You change the code, causing the failure, and then **update the test.**

Scenario A: The Bug-First Workflow

The **Counter-Intuitive Step:** Write a test that passes because it reproduces the bug.

Recipe

1. Identify the bug (e.g., function returns null).
2. Write a **'Passing Bug Test'**. Assert that the result **IS null**.
3. Commit. CI remains ***Green***. Bug is documented.
4. Fix code. Test now fails (Red).
5. Update test to assert correct object.

Code Evidence

test_bug_reproduction.js

```
it('should return null when input is invalid', () => {
  const result = system.process(badInput);
  expect(result).toBe(null); // Documenting the bug
});
```

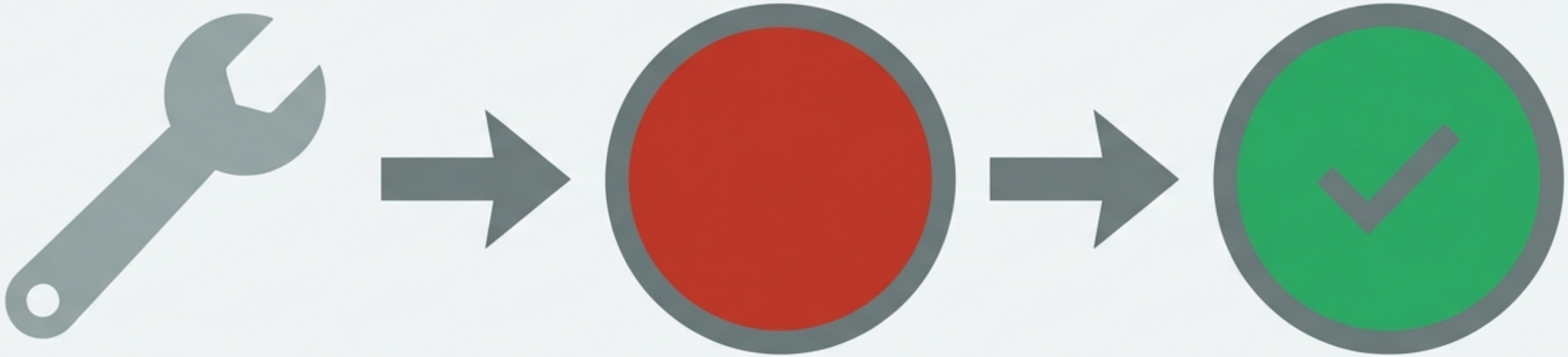
✅ PASS

Terminal Output after Fix

```
Expected: null
Received: { status: 'error', message: 'Invalid Input' }
Status: FAILED
```

❌ FAIL

Closing the Loop on Bugs



Fix Code

Code Black

Test Fails (Signal)

Success! The bug is gone,
so the bug-test fails.

Update Test

Assert correct behaviour.
Regression test established.

You have fixed the bug and created a permanent regression test in one tight, safe loop.

Scenario B: Evolving New Features

Documenting the journey from ‘Not Implemented’ to ‘Done’.

Initial State

Write test calling new function. Assert it returns ‘Not Implemented’.

Green

Increment

Implement basic logic. Test fails because behaviour changed from stub.

Red (Update Assertion)

Refine

Add edge cases. Code matches new reality.

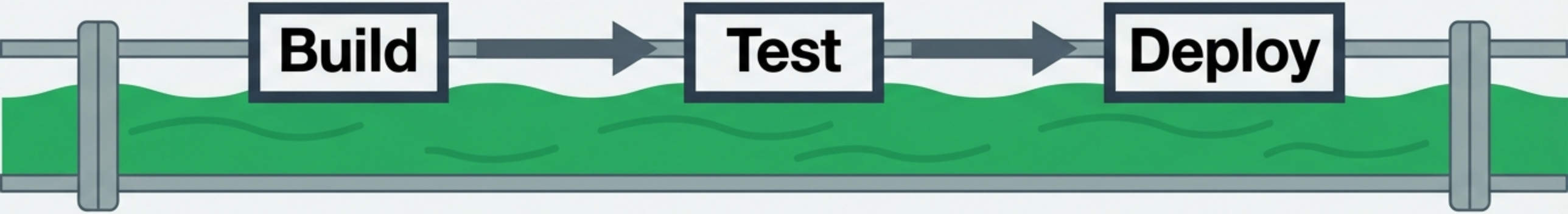
Green

Key Takeaway

You are never ‘guessing’ the final state. At any commit, the test suite accurately describes the code, even if the code is just a placeholder.

Harmony with Continuous Integration

The “Always Green” Standard



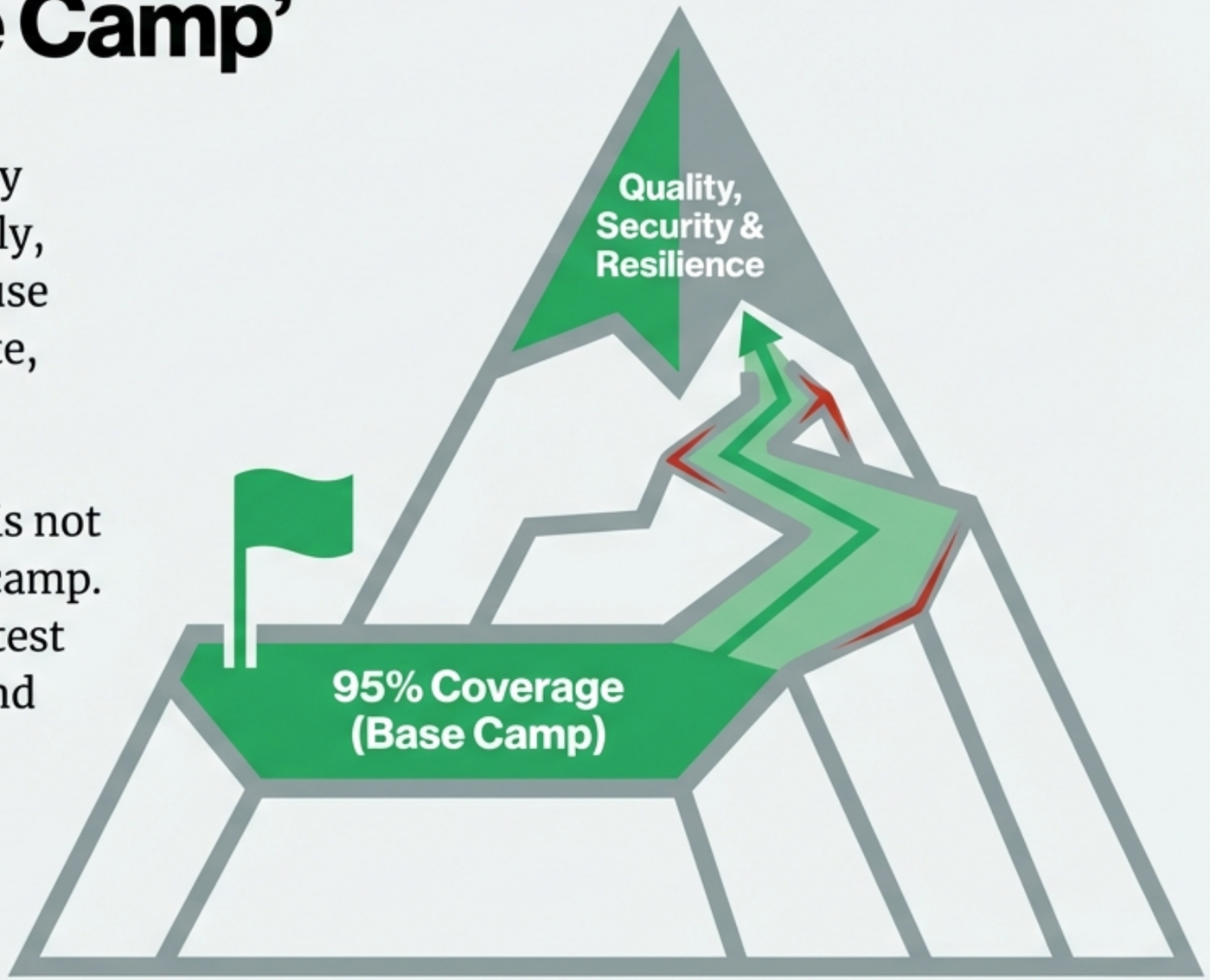
- **Stability:** Failures in CI become genuine anomalies to be investigated, not ‘work in progress’ noise.
- **Deployability:** The software is always in a deployable state. Even incomplete features are covered by tests that assert their current (incomplete) state.

Traditional TDD often requires skipping tests during long feature builds. PDD keeps everything active and passing.

Coverage as 'Base Camp'

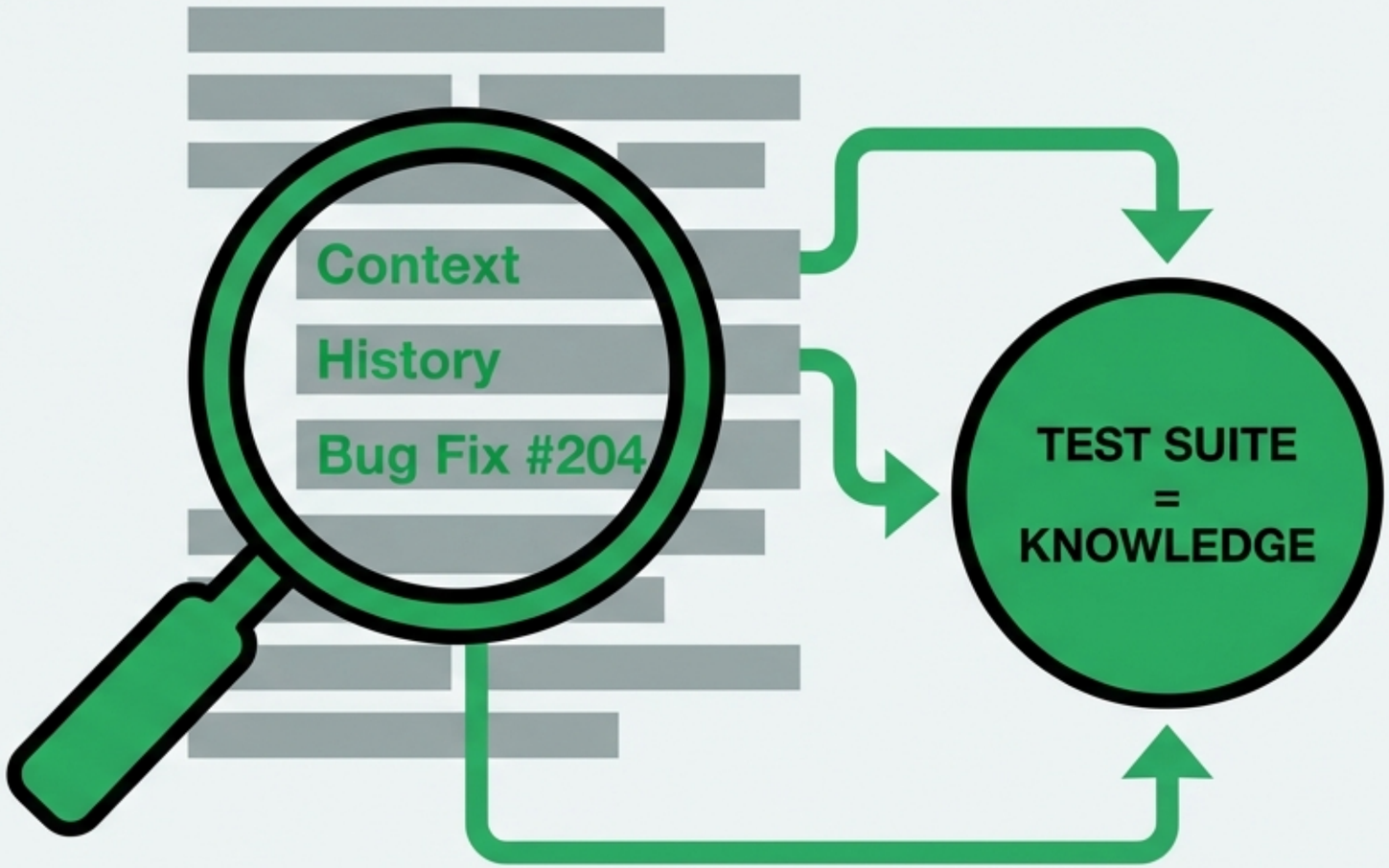
The Metric: PDD practitioners routinely achieve 95%+ code coverage organically, without management mandates. Because every code change triggers a test update, "untested code" rarely exists.

The Mindset: Reaching high coverage is not the destination—it is merely the base camp. From this secure base, developers can test multiple data variations, edge cases, and security scenarios.



Knowledge Preservation in Code

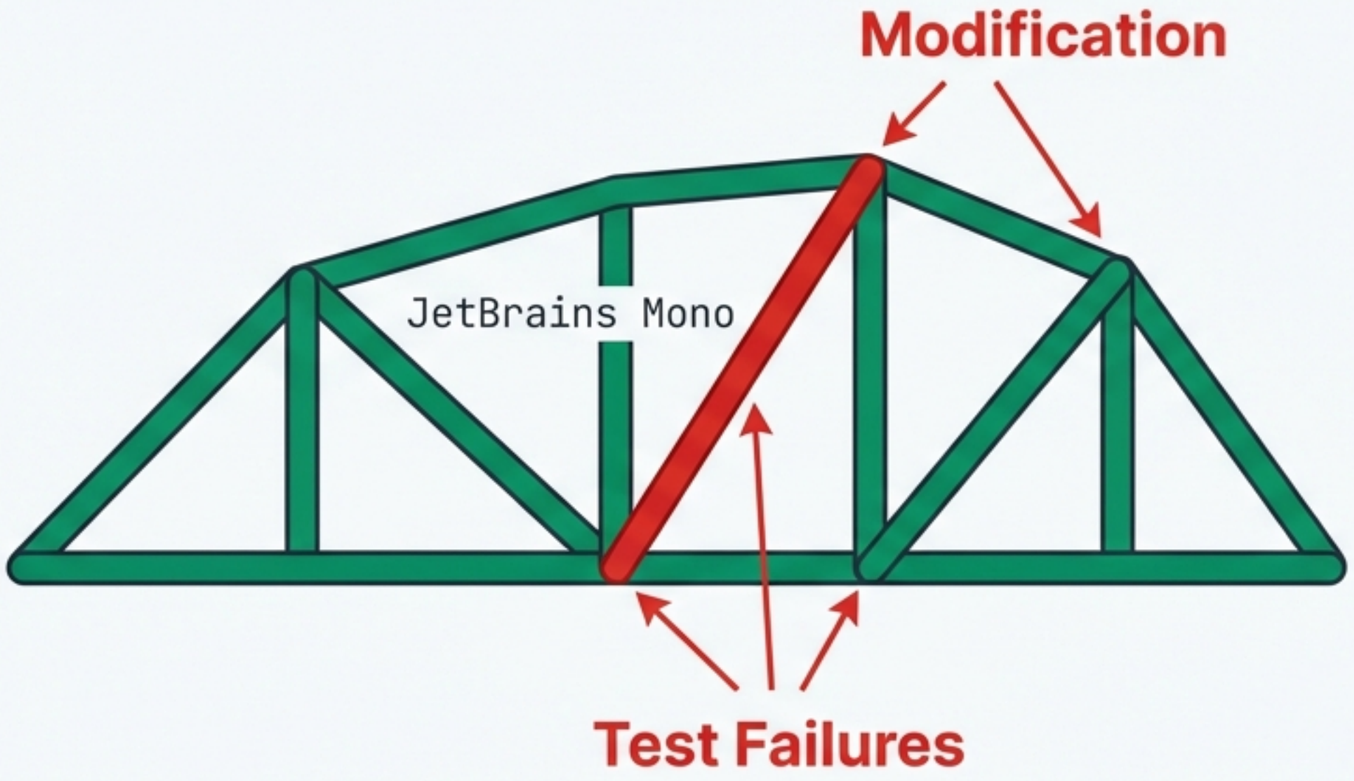
The PDD Solution: **The test suite captures the history of the developer's understanding.**
- When a bug is fixed, the diff in the test file explains exactly what the edge case was.
- New team members can read the tests to understand system behaviour, quirks, and handled bugs.



"This isn't just testing; it's knowledge preservation in code form." — Dinis Cruz

The Safety Net

Fearless Refactoring



The Silence Red Flag

If you change code and NO test fails, that is dangerous. It means the code is uncovered.

The Confidence Loop

In PDD, changing code SHOULD break a test. This confirms the safety net is working.

Result: Enables aggressive refactoring because the “Anchor” (current behaviour) is fully locked down.

The Toolkit & Technique

Requirements for Instant Feedback

Real-Time Execution

Tools like Wallaby.js or NCrunch are essential. Feedback must be instantaneous.

Test Data Factories

Invest heavily in helper methods to seed databases and create valid objects. Avoid boilerplate fatigue. fatigue.

Minimise Mocking

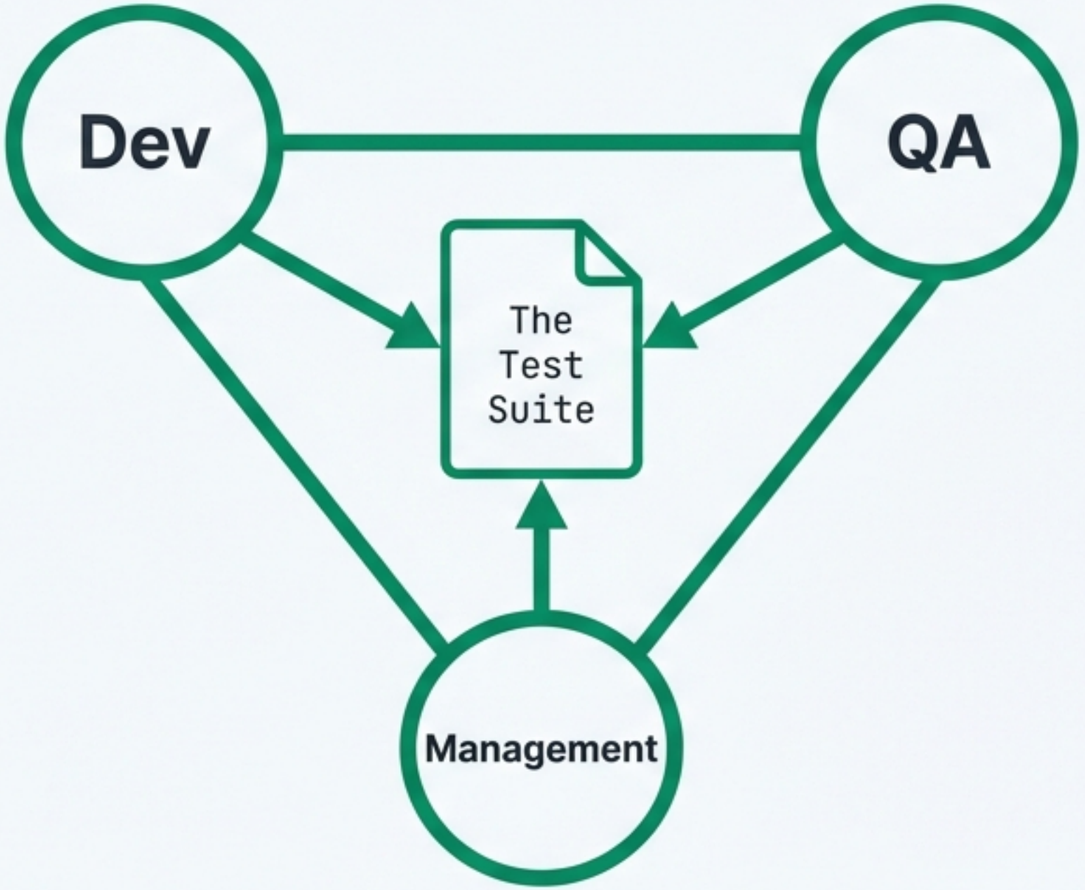
Prefer real objects and state. Mock only external boundaries to test actual interactions.

Test-as-Code

Tests require the same care, refactoring, and review as production code.

Cultural & Team Impact

Testing as a Collaborative Language



Onboarding: Helps juniors contribute safely; helps seniors retain context.

QA:

Reviews tests to understand changes rather than just manual regression.

Management:

Reviews test logic to verify business requirements without parsing implementation.

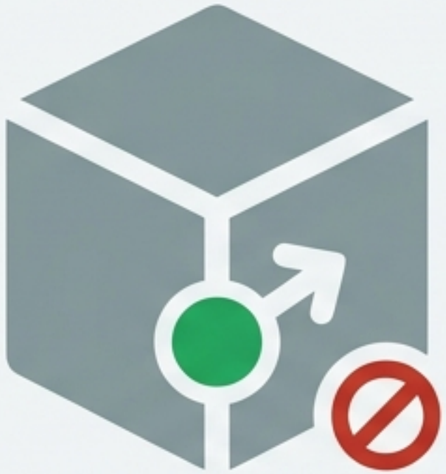
Dev:

Mindset shift—"Never make a change without a test that covers it."

Best Practices for Adoption

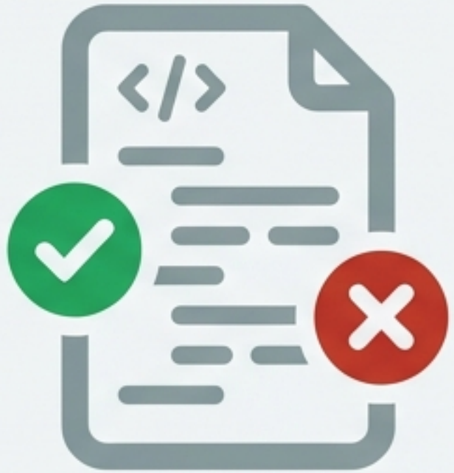
Instant Feedback

Tooling is critical. Use real-time execution monitors (e.g., Wallaby.js, NCrunch) to see Red/Green status immediately upon saving.



Minimise Mocking

PDD prefers testing real interactions. Use in-memory databases or lightweight containers rather than extensive mocks.



Test-as-Code

Treat test code with the same respect as production code. Review it. Refactor it. Maintain it.

Summary: Why Make the Switch?

	Traditional TDD	Pass-Driven Development
Testing Approach	Speculative (Future)	Verifiable (Reality)
CI Status	Short-lived Instability	Always Green
Refactoring	Fear of Breakage	Fearless Improvement
Perception	Tax / Overhead	Development Partner

Result: Near-zero critical bugs in production.

The PDD Mindset

- **Green from the Start: Never accept a broken build as part of the process.**
- **Bug-First Testing: Embrace defects. Capture them before fixing them.**
- **Fearless Refactoring: When the test suite mirrors reality with high coverage, you can aggressively evolve the codebase.**

PDD creates a safety net that encourages, rather than constrains, the evolution of code.

Adoption Strategy

Getting Started Tomorrow

1 Start with the Next Bug

Do not rewrite old tests. When the next bug arrives, write a **'Passing Bug Test'** that asserts the bug exists. Fix it, then update the test.

2 Invest in Infrastructure

Build a library of **Test Data Factories** to make writing these tests frictionless.

3 Mentorship

Pair program on the first PDD feature to demonstrate the '**Green-Red-Green**' **rhythm**.

4 Gradual Rollout

Apply PDD only to new features. Over time, the '**Base Camp**' of coverage will naturally expand.

Conclusion

Stop testing what you hope will happen. Start testing what IS happening.

PDD is about psychological safety and truth in code. It turns the test suite into a partner that evolves with your understanding of the system.

Embrace the **'Pragmatic Inversion'.**

Sources
Based on the work of Dinis Cruz and concepts from Martin Fowler.