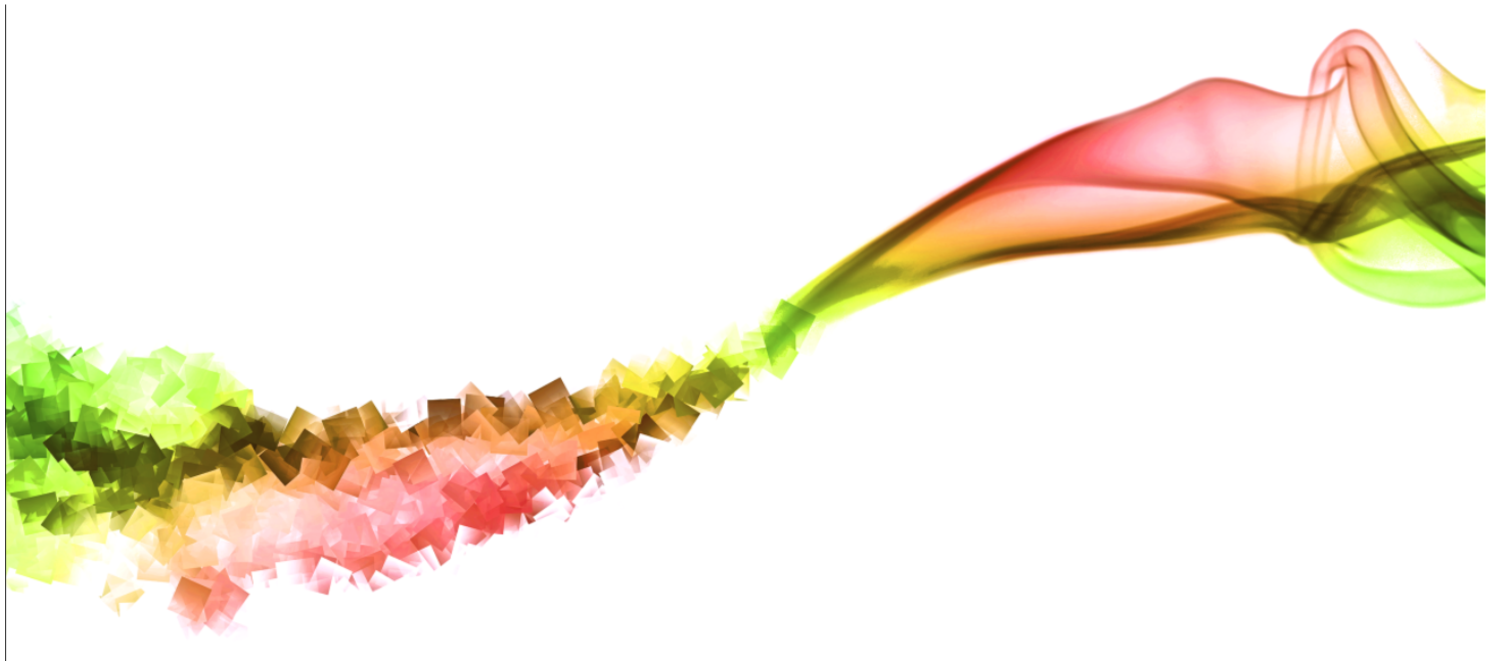
Testing Software: What is TDD?

Eric Elliott Follow
Jan 14, 2020 · 9 min read

A Practical Introduction with Q&A



Test Driven Development is a software development process that follows these steps:

1. Write a failing test for one requirement.
2. Implement just enough code to make the test pass.
3. Refactor with confidence (if it's needed).

TDD & the Scientific Method

TDD is like the scientific method, but for software. The scientific method is how we learn things about the world. It works like this:

1. Make an observation.
2. Ask a question about the observation.
3. Form a hypothesis or testable explanation of the observation.
4. Make a **prediction** based on the hypothesis.
5. Test the prediction.
6. Iterate: use the results to make new hypotheses or predictions.

Let's break that down into simpler elements:

1. Question
2. **Prediction**
3. Experiment
4. Subject

With TDD, instead of learning things about the world, we're creating a new world that should conform to our specifications. Instead of starting with an observation about the real world, we begin with a requirement. TDD flips the process upside down. Instead of learning about the real world, we're adjusting the scientific method to create a new one.

Scientific Method	**TDD**
Question	Requirement
Prediction	*Expected Output*
Experiment	Test Assertion
Subject	Implementation Code

tdd-scientific-method.md hosted with ❤ by **GitHub** **view raw**

The key factor in both the scientific method and TDD is **prediction**. Using prediction, we eliminate the possibility of hindsight bias. After the result is already known, people often believe that they could have predicted the result: That they knew it all along. In code, hindsight bias can have a more insidious effect: your code could produce the wrong output, but you might believe it is the correct output.

Hindsight bias is a common phenomenon with snapshot testing, a form of tests that alert you when something changes, rather than when something is wrong. If you've just made a change in code, and a snapshot test alerts you that there was a change, you're already expecting the alert, and you're biased to believe that the outcome is correct, which makes you vulnerable to accepting incorrect results.

But the effect is not limited to snapshot testing. Often, while adding tests after the implementation is complete, developers are inclined to accept the output that the program generates without carefully checking the results with manual calculations.

TDD forces us to make a predictive calculation of the expected output because the code which will produce the actual output does not exist yet. There's no opportunity for hindsight bias to cloud our judgment.

With this in mind, we can enhance our TDD process:

1. Translate a requirement into a falsifiable test.
2. **Predict** the expected output based on the requirement.
3. Run the test to get the actual output. It should *fail* the first time. If it passes, go to step 1.
4. Write the implementation. The test should *pass* this time. If it fails, repeat this step.
5. Move on to the next requirement and repeat from step 1.

TDD Benefits

- **Eliminates fear of change.** If a code change introduces a bug, developers are alerted to it quickly, and TDD's tight feedback loop will quickly notify them when it's fixed.
- **A safety net** which makes continuous deployment safer. Test failures halt the deployment process, allowing you to fix bugs before customers ever have the chance

to see them.

- **40% — 80% fewer bugs**.
- **Better code coverage** than writing tests after the fact. Because we create code to make a specific test pass, code coverage will be close to 100%.
- **Faster developer feedback loop.** Without TDD, developers must manually test each change to ensure that it works. With TDD, unit tests can run on-change automatically, providing faster feedback during development and debugging sessions.
- **Interface design aid:** Developers often think about the software implementation before thinking about the developer experience of using the software component. TDD flips this around, forcing developers to design the API before working on the implementation.
- **KISS and YAGNI** — "Keep it Simple, Stupid", and "You Ain't Gonna Need It" are two overlapping software design principles. KISS means just like what it sounds like it means. Keep things simple. YAGNI means don't build features and abstractions unless those features serve a specific *existing requirement* (not a future requirement). TDD helps with that process by forcing you to work in small iterations, tackling one requirement at a time on an as-needed basis.

TDD Costs

Users without TDD experience may find they move 15% — 30% slower, but with 1–2 year's practice, TDD's realtime feedback process can enhance productivity.

Tips

- **Prediction comes before implementation.** (Snapshots and test-after do not qualify as TDD).
- **Use small iterations.** Test and build one requirement at a time.
- **Refactor with confidence,** but only when it's needed.
- **Learning TDD is the process of learning modular app architecture.** It usually requires 1–2 years of practice to get good at it. Be patient. TDD taught me most of

what I know about modular application architecture.

Questions and Answers

> *Is 100% code coverage reasonable?*

It is unreasonable to aim for 100% code coverage using unit tests alone, because a lot of code exists to integrate your app with other services, such as the network, the disk, the user's display, and so on. Unit tests work best with deterministic, pure functions: functions which, given the same inputs, always return the same outputs, without any changes to any externally visible state.

That leaves a portion of your code uncovered by unit tests. That's where functional tests and other types of integration tests come in. Functional and integration tests ensure that your units of code behave in integration with each other.

The closer you get to 100% coverage with unit tests, the harder it is to improve unit test coverage. At some point, you'll find yourself jumping through hoops, which makes your code more bug-prone and harder to maintain.

Selecting the correct type of test for the job at hand can help you avoid the temptation of writing unnecessary mocks. It can also keep your code simple by helping you avoid hoops like dependency injection when they aren't required. You should not complicate your code for the sake of unit tests.

Rule of thumb: TDD should make your code simpler, not more complicated.

In summary: 100% coverage is good, but you should achieve it with a combination of unit, integration, and functional tests. Use the right kind of test for the job at hand.

> *Should you use TDD for visual styling?*

No. Instead, visual regression and snapshot tools can help you test visual styling. Visual designs tend to change frequently and are often more subjective than objective. It's challenging to come up with a scientific prediction for how something should look and how that look can be achieved with markup and styles. Visual regression tools can alert you if the UI changed, but a designer should approve or reject the change, not a computer. Keep in mind that snapshot style regression tests do *not* qualify as TDD. Visual snapshots can enhance your process, but can not replace TDD.

> *Should you use TDD for accessibility concerns?*

Yes! Concerns such as *"are we using the correct semantic markup?"* and *"does the component have the correct ARIA hints?"* can be easily unit tested.

> *Sometimes, new requirements overlap with existing requirements that already have a passing test. How should I handle that?*

Whether or not the existing code already covers a test case, you still want to explicitly add the new test case because tests help document the requirements of the API. When this happens, you can comment out the code that makes the test pass, write the new test, watch it fail, and uncomment the code to watch the test pass again. It's essential to see every test fail because if you've never seen a test fail, you don't know if the test works.

> *Sometimes, function output is hard or impossible to predict. Using the TDD process, how do I know how to predict the expected value?*

It should be uncommon to encounter situations where you really can't predict the outcome of a function call.

It's often a code smell indicating that the function is not deterministic. For instance, the function uses the current system time or generates a random number.

Create optional parameter overrides for function-generated data, such as the date or a random number.

```
import cuid from 'cuid';

const createTodo = ({ description, date = Date.now(), id = cuid() })
=> ({
  type: 'createTodo',
  payload: { description, date }
});
```

If you don't pass in the `date` or `id`, it will generate them at call time. In your unit tests, you can pass them in to achieve deterministic output.

Sometimes, you're just unfamiliar with how an API will behave. Perhaps it's not well documented, or it might take too long to find the documentation. In those cases, I'll often start with an intentionally wrong expectation, view the actual output, and then copy and paste the output into the test's expectation.

To test the test, I'll comment out the implementation code and ensure that the test fails without it, then revert the comments and watch the test pass again. Just keep in mind, this style of testing is weaker than prediction-driven tests because you're skipping the critical prediction step in the scientific process, and introducing the possibility of hindsight bias.

> *Should I capture the TDD process in my git history?*

No. Good test coverage is assurance enough that the TDD process is working. Commits like "unit test for feature x" will only clutter your git history and take you out of the flow of the work you're doing in the code.

As you're building features, your mind should be on the code you're writing, not on process for the sake of process. Just like you should keep your code simple, you should also keep your process simple.

For example, you can use a watch script that automatically runs your tests on every file save. You don't have to perform any extra steps to see if your code works. Write the implementation code and check the test output in your development console, the whole while staying in the flow of building code. You don't need to stop writing code, manually run the tests, and then task-switch back to writing code again. Don't distract yourself from your real work.

> *When we refactor our code, a lot of tests will start to fail. Should we refactor the tests, or write new ones?*

This question exposes a common misuse of the word "refactor." Refactor doesn't mean "any code change." Instead, "refactor" means that we're altering the internal structure of the unit under test without changing its external behavior. Use black-box testing, which means that for each unit of code (regardless of granularity), you only test the surface area of that unit's API.

Don't write tests that know about the internal implementation details. For example, if you're testing a map method for a new data structure, don't check the data structure itself, because if you do, and you later change the data structure, your test will fail, even if the external behavior has not changed.

Tests should not break when you refactor the unit under test. Unit tests should only break if the unit's behavior requirements change. For instance, you start out returning an array, and later discover you need to return a keyed object. At that stage, you have a choice to make. Would it require more work to refactor the existing tests, or to rewrite them?

Sometimes requirements change so deeply that the existing test code is not very usable. In those cases, you should still carefully look over the requirements and decide which ones apply to the new behavior, and which ones don't. Make notes of all the applicable requirements you need to test, and then scrap the tests you don't need.

Next Steps

TDD Day is a full day (5 hour) team onboarding webcast recording that will teach your team TDD essentials and provide a ton of practical advice driven by Q&A from teams like yours — the same quality training used by teams like Disney Interactive, Spotify, Netflix, PayPal, and Amazon.

The TDD Day videos are available to members of EricElliottJS.com, along with a ton of content on topics like functional programming, React, Redux, and more. The Shotgun series lets you watch me build real production apps, so you can see the TDD process in action.

Sign up today.

Eric Elliott *is the author of the books, "Composing Software" and "Programming*

JavaScript Applications". As co-founder of EricElliottJS.com and DevAnywhere.io, he teaches developers essential software development skills. He builds and advises development teams for crypto projects, and has contributed to software experiences for ***Adobe Systems, Zumba Fitness, The Wall Street Journal, ESPN, BBC,*** *and top recording artists including* ***Usher, Frank Ocean, Metallica,*** *and many more.*

He enjoys a remote lifestyle with the most beautiful woman in the world.

Thanks to JS_Cheerleader.

JavaScript Tdd Technology Software Development Programming