symflower

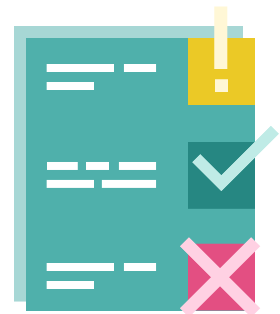
TDD Made Easy: **An introduction to Test-Driven Development**

This white paper is based on interviews with software engineers who practice TDD

Introduction to TDD

Test-Driven Development (TDD) is an incremental and iterative approach to software development that is gaining popularity due to its perceived benefits. Teams jumping on the TDD bandwagon claim that the methodology enables them to focus on software quality and code clarity while ensuring high test coverage in an accelerating development environment.

This guide will provide you with fundamental information about Test-Driven Development, and helps you understand how TDD relates to traditional development. Based on our interviews with a number of software engineering experts about their experiences with TDD, we'll also be covering the pros and cons of this approach, and will provide some best practices for teams looking to adopt this methodology in their projects.



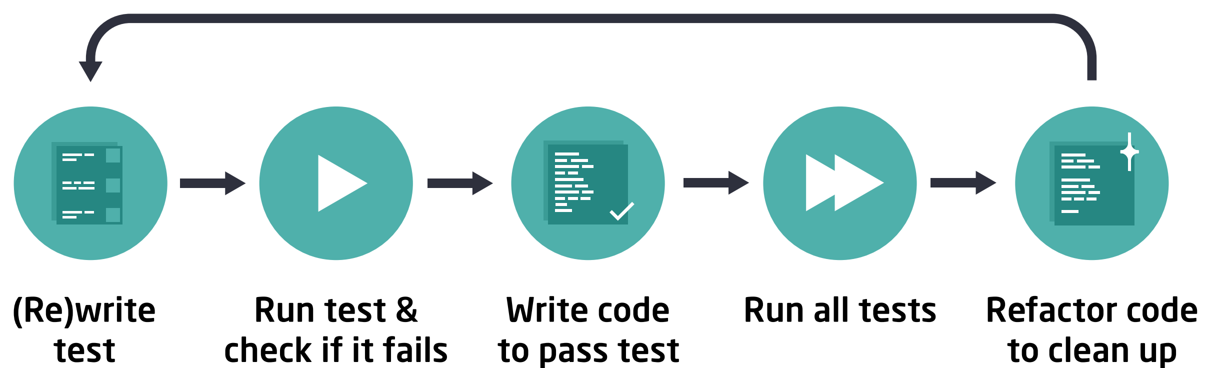
If you're just getting started with Test-Driven Development, this fundamental guide is a good place to begin your journey. If you're already familiar with TDD, you may find our best practices helpful.

What is Test-Driven Development?

TDD has its roots in the test-first programming approach of extreme programming, and has been around since the early 2000s. Its popularity is picking up, but TDD is far from being universally practiced: a 2020 survey found that 41% of respondents claimed to have fully adopted Test-Driven Development, but only 8% of them actually wrote tests before code at least 80% of the time.

The underlying concept of TDD is fairly simple, and is best thought of as a specification technique. The methodology essentially converts requirements into test cases, which in turn will guide the implementation of the production code. In a TDD environment, test cases are used for both specifying and validating what the application should do.

Each test case covers one behavior of the software at a time. Code is written to pass that test case specifically (and nothing else), resulting in a small amount of code being written at a time. Code is then repeatedly added and adapted until it passes all the test cases, making sure that the application delivers all the required functionality. This way, the complexity of code grows in increments, encouraging simple design.



Traditional Development vs TDD

Put simply, the concept of TDD turns the way software applications have traditionally been developed upside down. Whether it is implemented in a Waterfall or an Agile environment, the tried-and-tested classic approach to building software roughly follows this process:

1. **Write requirement specifications** that describe what the software should do
2. **Write code** to tackle those requirements, making sure all the required functionality is covered
3. **Test the application** to make sure it works as intended

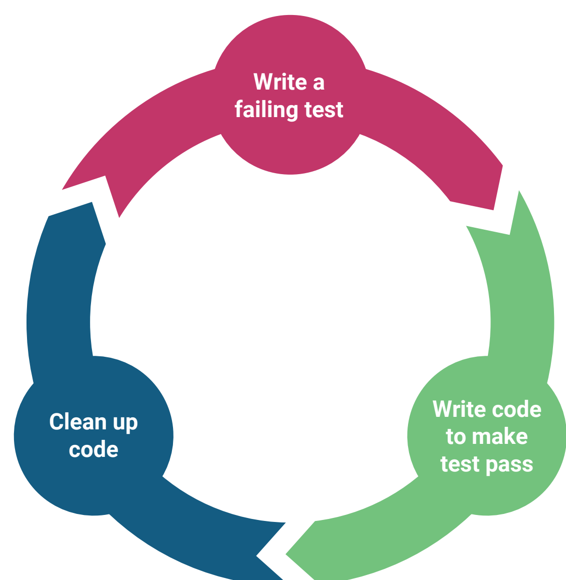
With Test-Driven Development, that whole process works exactly the other way around.
A general TDD workflow consists of the following stages of the Red/Green/Refactor cycle:

1. **Write a test** with the right acceptance criteria to validate the behavior that you want to have in your application.
2. **Run all tests.** Your new test should fail for the expected reasons.
3. **Write the simplest code** that enables the implementation to pass your new test.
4. **All tests** (your new test as well as all previous test cases) should now **pass successfully**.
5. **Refactor as** needed to clean your code of duplications, running the tests after each refactor.
6. **Repeat the process,** adding tests and then code to pass that test.

Starting out, the **Red phase** is, of course, red because your new test will fail. This is where you'll define what your implementation should do.

In the **Green phase,** you'll write just enough code to pass that first test. In simple terms: don't make it pretty, just make it work.

The **Refactor phase** is where you'll get around to improving your code and removing any duplications. Your unit test will support the refactoring, as your implementation will still have to pass that same tests at the end. Your implementation may change, but the tests stay the same, and will only change in case the underlying business requirement changes.



That makes it pretty easy to understand the key difference between TDD and traditional development. While testing traditionally may have been the last phase of development before an increment of code was shipped to production, TDD makes it the very first step, and continues to test throughout the process of the software being built in increments.

The importance of testing first is underlined by the three key laws of Test-Driven Development, as defined by software engineering guru Robert C. Martin (aka “Uncle Bob”), founder of the Agile Manifesto and the first chairman of the Agile Alliance:

> 1. You can’t write any production code until you have first written a failing unit test.
> 2. You can’t write more of a unit test than is sufficient to fail, and not compiling is failing.
> 3. You can’t write more production code than is sufficient to pass the currently failing unit test.
>
> — *Robert C. Martin, The Three Laws of TDD*

Those rules highlight how TDD changes the way we perceive failing test cases. In a TDD environment, a successful test is one that fails, as it has successfully uncovered a problem to be solved. While that’s not dissimilar to traditional development, it’s harder to feel happy about a failing test there, as it means that you have to go back and fix the code you already wrote. With TDD, a failing test is what drives development forward. But feeling better over failing tests isn’t the point – TDD also offers more tangible benefits for software developers.

symflower

The benefits of Test-Driven Development

Improved code quality	Better solution design	Faster development
Faster feedback	Code with confidence	Better communication & collaboration
Simplified documentation	Better APIs	Greater testability
Easier maintenance	Increased code reuse	

Since TDD inverts the way software is traditionally developed and tested, implementing it involves a bit of a learning curve (more on this in a subsequent section). The reason teams still dedicate time and effort to adopt it is because of all the documented benefits of TDD. Below, we're listing some of the key benefits – some anecdotal evidence from online resources, some from our personal interviews with TDD practitioners:

Improved code quality

As one of our interviewees, Evelyn Haslinger, Founder & COO of Symflower and a TDD practitioner put it, *"Having no tests is like working in the dark: you basically hope what you are doing is correct"*. By writing tests before writing code, you can be sure that your code meets the desired specifications, works as intended, and is adequately covered with test cases. Using TDD tends to lead to code that is easily testable and that, by definition, has high test coverage from the get-go.

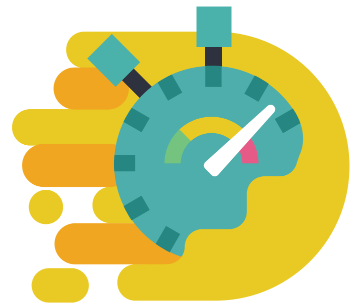
"TDD is the best way to write high-quality, maintainable code"
– Vincent Finn, Software Engineering Consultant

Better solution design

The process of writing tests can help you to architect your code in a more modular and flexible way, which can make it easier to extend and maintain.

Faster development

With TDD, you'll focus on writing only the code that is needed to pass one test at a time. As Vincent Finn (Software Engineering Consultant) put it: *"TDD helps you to not build more than you need to"*. TDD can help you to avoid writing unnecessary code, focus on simplicity, and can speed up the whole process of development.



Faster feedback

By running your tests frequently as you write code, you'll get faster feedback on whether your code is working as expected. *"You can be constantly delivering something and working on the feedback you get"*, said **Hélio Rocha**, Senior Solutions Engineer at an identity verification unicorn. This can help you to catch and fix issues early on, which can save time in the long run.

Code with confidence

TDD helps ensure that your application satisfies all requirements. By having a comprehensive test suite, you can have greater confidence in the quality and reliability of your code. Once your tests no longer fail, you know that you have implemented everything that was needed to tackle that requirement, and that you have implemented all that in a way that functions correctly. *"If we didn't have TDD, our code could be more flaky, we wouldn't have the code confidence that we like to have."* said **Benjamim Alves**, former Senior Software Engineer.

Better communication & collaboration

By writing tests that specify the desired behavior of your code, you can more clearly communicate your intentions to other developers working on the same codebase, leading to a shared understanding. As Gabriel Matos Boubee, Frontend Developer and Student put it: *"Communication between groups improved immensely. Everyone is requested to go with the flow of the tests. This stops anyone from going with unplanned code or something that deviates from the tests."*

Simplified documentation

Somewhat related to the previous point, in a TDD environment, unit tests serve as documentation. There may be no need to produce code documentation (comments alongside code) at all, and code is just easier to navigate. Quoting Benjamim Alves (Senior Software Engineer): *"When I enter a new code base, I don't know the business logic, I don't know what those functions produce. The first thing to do is to read the tests. I can instantly see what that method does, what is meant for that function."*

Better APIs

Since the developer writing the code is actually the first user of a newly added API, they have a better chance of identifying bad design choices early on. Therefore, using TDD tends to lead to APIs that are more convenient to use.

Greater testability

By writing tests before writing code, you will naturally design your code in a way that makes it easier to test. This can be particularly helpful for code that is difficult to test, such as code that integrates with external systems, or code that involves some complex logic.

Easier maintenance

Code written with TDD tends to be clean, readable, and simple to navigate & manage. TDD can make it easier to maintain and refactor code over time because the tests can be used to verify that any changes to the code do not break existing functionality. *"The more you are covered by test cases the more safe it is to make changes"* said Sameh Muhammed, Software Engineer. It's also worth noting that the tests themselves need not be changed when refactoring. As Benjamim Alves, Senior Software Engineer: *"If you're changing tests, you need to check if the business requirement has changed. If not, the test shouldn't be changed."*

Increased code reuse

Writing tests before writing code lets you design code in a way that makes it easier to reuse. This can help avoid duplication, and reusing code across different projects contributes to enhanced efficiency in the long run.

Test-Driven Development Best Practices

Because it's such a departure from the code-first approach, adopting Test-Driven Development can be a bit of a challenge. While TDD does promise all those appealing benefits, old habits die hard. If you are used to working without TDD, it takes some discipline to enforce Test-Driven Development practices in a team.

Benjamim Alves, former Senior Software Engineer said: *"As humans, we always try to go right to code, write the solution, and put it up and running"*. With Test-Driven Development, you need to suppress that urge and flip your thinking to writing tests first – this transition can take some time. Following these best practices can help shorten your route to value with TDD and ensure that you stay efficient with Test-Driven Development:

Best practice #1

Keep tests small, rinse and repeat

This one should go without saying, but let's start with the basics: it is crucial to write tests that are focused on a single specific aspect of the code's behavior. Run these small tests very frequently. This way, you can ensure that

- any changes to your code don't break the existing functionality
- failures are easy to identify
- and that your test suite is maintainable.

Best practice #2

Test with purpose

It's important to understand that TDD is not a silver-bullet solution to all your testing needs. Prioritize application functionality and test on various levels (i.e. unit, integration, and system testing) to adequately cover each behavior and aspect of the solution. As Hélio Rocha, Senior Solutions Engineer at an identity verification unicorn put it: *"Don't limit yourself to unit tests only. You should always have integration and regression tests as well. That's one of the most important aspects of CI/CD."*

Symflower blog recommendation:

A crash course into the different types of testing

Use tests wisely. Make your tests well-structured, short, and simple. When in doubt, fall back on the FIRST principle which suggests that your unit tests should be:

Fast: If running tests takes too long, developers may decide to run their test suites less frequently. It's crucial to ensure that tests execute fast, allowing developers to run them as needed.

Independent: Avoid dependencies between tests. Tests shouldn't be dependent on external factors nor should they depend on each other.

Repeatable: Your tests should be deterministic, i.e. no matter how many times tests are run (and in what environment or in what order they are executed), they should return the same output.

Self-validating: Information on the outcome of each test should be readily available, no manual checking should be needed to find out if a test has passed or failed.

Thorough: You'll want to make sure that your unit test suite covers the entirety of the specified behavior (including both happy paths and corner cases, illegal arguments, security issues, and basically every realistically imaginable use case and scenario).

symflower

Best practice #3

Write minimal assertions first

When writing tests, you'll want to keep the assertions in each test to a minimum to avoid adding complexity. Ideally, each unit test should assert a single logical outcome – only add more assertions to a test if those assertions focus on the same logical condition or piece of functionality.

Best practice #4

Get your naming conventions right

For any team, naming conventions are important for organizing tests – in a TDD environment, they are especially crucial because of the volume of tests you're going to produce. Make sure the names you use for your test methods are descriptive, refer unambiguously to the appropriate implementation classes, and are easy to understand. Using the right names can help other developers working on the code to decipher why certain tests have failed.

Best practice #5

Keep implementation and testing code organized

This may be natural to most, but it's worth mentioning just in case: have separate source directories (e.g. src/test/java and src/main/java) to avoid mixing up tests with production code. That said, it's a good idea to use the same package structure for test classes that you are using for your implementation. When tests are located in the same package as the code they test, that provides clarity and a transparent structure that is easy to navigate for all contributors.

Symflower blog recommendation:

How to name test files and where to store them

Read blog post

Best practice #6

Use mocks in your unit tests

External dependencies can slow down development. Use mocks where appropriate to accelerate unit test execution. Mocks help you focus on small units of functionality, and save you the time of setting up external dependencies. Mocking is a key practice for unit tests and therefore, for TDD – but that doesn't mean you can skip other higher-level forms of testing (see next section).

Best practice #7

Don't skimp on higher-level testing

While in the case of newly written code, you'll most likely be starting out with TDD on the level of unit tests, the concept can (and should) actually be applied on higher levels of testing, too. Even if you religiously practiced TDD on the unit test level, that doesn't mean you've done sufficient testing to ensure your software works as intended. Gradually work your way up through integration and system testing with TDD until your final test suite is actually at the acceptance level.

Best practice #8

Minimize your time spent in Red

Follow the advice of **Kent Beck**: *"move slowly very very frequently"*. By keeping the amount of code you're writing at a time to a minimum, you can ensure that you're only taking tiny steps – but take those steps very frequently.

You may even want to go as far as Kent Beck advises: maximize the time spent in red in 60 seconds. In other words, any implementation you deliver to pass a failing test shouldn't take longer than 1 minute. While the 60-second timebox is certainly not suitable for all development environments, it makes sense to set time limits for yourself to make sure you stick to simplicity and focus on small bits of software behavior at a time.

Best practice #9

Use test coverage tools

Ideally, in a TDD environment, any and all implementations already have at least one test case. In practice, however, that's not always the case (for instance, if you're working with legacy code). Therefore, it's a good idea to use test coverage tools to identify areas of your code that are not being adequately tested. This will help make sure you tackle whatever goals you have set for high test coverage.

Best practice #10

Double and triple-check your tests

When writing tests manually in a TDD environment, there's some chance you'll miss corner cases. As we heard from Dan Putman, Customer Engineering Manager: *"I can have 100% coverage without having a null check"*. Taking advantage of pair programming practices, going back to check your tests, or using smart tools to generate unit tests that complement your own test suite are all good practices.

Automate the creation of unit test templates with Symflower.
This IDE plugin writes your boilerplate code to accelerate TDD.

get.symflower.com

Challenges and drawbacks of TDD

Despite what all the positive experience reports suggest about TDD, some practitioners agree that Test-Driven Development is not for every project, and there are some caveats. Before making the decision to try TDD, you'll need to carefully consider some of the difficulties related to adopting and using this approach.

TDD can be time-consuming at first

The general experience is that for teams new to TDD, learning and getting used to the methodology can take a bit of time initially. *"You have to practice at the unit level before you can think about applying TDD to your projects"* said Mark Bradley, Consulting Software Engineer and host of the Testing All The Things screencast. The majority of the developers we talked to agreed that the benefits of TDD still justify its use in most projects. But they also warned not to underestimate the learning curve and move slow with adopting TDD: *"Don't try to practice and learn TDD in your production code base where it's messy and complicated"* said Mark.

Be cautious when working with legacy code

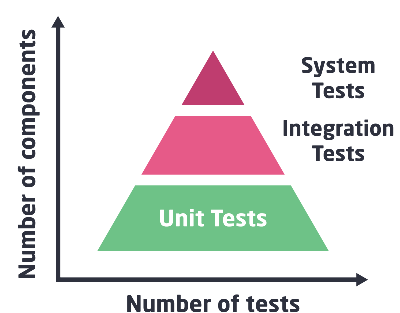
Similarly, it can be challenging to apply TDD to legacy code, as the code may not have been designed with testing in mind. If you are new to TDD, it's easier to start writing from scratch rather than going into existing code. If you do have to work with legacy code where testability is a problem, it makes sense to start by adding tests to the outside of the application. Having system tests makes it safer to start making changes to the code base. Once you have some test coverage, you can start refactoring code to add testability: *"Test the core functionality from the outside, and then work your way in. You want to have that safety net of non-unit tests to modify things"* as Mark Bradley noted.

Difficult process enforcement

Since it essentially flips developers' understanding of software development upside down, TDD requires quite a bit of discipline to implement. It can be quite easy to slip back into the habit of writing code first. In order to adopt TDD, all team members have to fully opt in and apply its principles, which can be difficult to enforce in a team. Pair programming helps, as does setting clear guidelines and having the right processes in place. Regularly practicing katas also helps learn the ropes in a Test-Driven Development environment.

Streamlining TDD with smart unit test templates

Based on the concept of the testing pyramid, most of your testing will be carried out at the unit test level. Working in a TDD environment, unit tests will make up the bulk of your test cases, meaning that you'll face most of the time and effort costs of writing test cases first at the level of unit tests.



To minimize the impact that writing unit tests has on your development efficiency, you'll need to find a way to create all-encompassing unit test suites as fast as possible. That's exactly what you can do with Symflower, a smart IDE plugin that automatically creates unit test templates for Java and Go code. Use these as boilerplate code and just fill in the right values to significantly reduce the time and effort costs of unit testing.

symflower

Summary

The wide range of benefits that Test-Driven Development promises has more and more teams jumping on the TDD bandwagon. Faster development, better code quality, more efficient collaboration, and reduced documentation effort all contribute to the value of the Test-Driven Development approach. So does the easier maintenance and streamlined reuse of code written with TDD.

But as we saw, TDD is not for everyone, and you'll need to carefully consider the characteristics of this approach before you decide to implement it. We hope that this fundamental guide has provided you with a thorough understanding of the concept of TDD, making it easier to make a decision on whether you'd benefit from applying Test-Driven Development in your projects.

Sources

https://www.guru99.com/test-driven-development.html
https://www.diffblue.com/DevOps/research_papers/2020-devops-and-testing-report/
https://www.ibm.com/garage/method/practices/code/practice_test_driven_development/
http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd
https://methodpoet.com/test-driven-development-best-practices/
https://fortegrp.com/test-driven-development-benefits/
https://www.agiledata.org/essays/tdd.html
https://medium.com/@tasdikrahman/f-i-r-s-t-principles-of-testing-1a497acda8d6
https://opensource.com/article/19/10/test-driven-development-best-practices
https://scand.com/company/blog/test-driven-development-best-practices/
https://stanislaw.github.io/2016/01/25/notes-on-test-driven-development-by-example-by-kent-beck.html
https://phoenixnap.com/blog/tdd-vs-b