TDD Top Tips

Test Driven Development (TDD) is one of those very rare software engineering practices that can make a real difference to the quality of your code.

As well as minimising bugs in production, when done well, TDD will give you clear feedback on whether your code does what you mean it to; and, whether your design is any good.

Writing tests first is a skill that needs to be learned and practised, and there are some common pitfalls and hurdles to be understood and overcome.

There's no simple checklist or 'TDD by numbers', but here are some of my Top Tips to help you as you practise and learn to adopt TDD.

When to Use TDD?

Start Any New Project with TDD

This is the easiest time to begin!

Establish a Continuous Integration system to run tests automatically and get fast feedback.

Write All New Code with TDD

Even if working to improve, or adding to, an existing/legacy system (see below).

Adopt the approach that new code added to the legacy code will be done with TDD.

Start When you are Clear what the Outcome is that you are Looking For

Don't wait until you know the solution. Start when you understand the problem. Design from the outside in by adopting the discipline to…

Always Start with a Failing Test

Predict the outcome of the test before you run it – How will your test fail?

Test to Evaluate Behaviour, NOT Implementation

We want to keep our tests focused on the behaviour – the desirable outcome, that we seek from our system, rather than detail of implementation.

When writing your tests, imagine different implementations… Is the test still valid?

Design Your Code from the Outside-In. We want to only access our code through public, normal interfaces. In fact, since we write the test before the code, we take the opportunity to design the public interface to our code from this external perspective. This is a very good thing, since it encourages us to design our code to be easy to use. After all we don't want to make our own life more difficult by making it hard to write our test.

Test First To Improve Design

Begin new pieces of work by first creating a test that expresses some need in our software. Next, write some code to make the test pass.

This act of only adding to the code when we have a test that asks a question of it, forces us to focus first on the external view that our code exposes.

At the point when we write the test, we don't yet have any code, so this forces us to focus on the design of the code from the perspective of a consumer of that code.

Since we want our test to be easy to write, we are under a small, subtle pressure to write code that is easy to use. This applied pressure to design code that is easier to use, has a significant impact on the quality of the code that we produce; enhancing the properties that we value as markers of high-quality in code.

Three Mindsets of TDD

TDD is best described as a simple process – "Red, Green, Refactor". This simple organising principle is the driving force behind TDD.

Red – Write a test, run it and see it fail.
When writing the test, the goal is to clearly express the outcome that your code needs to achieve. Do this without worrying about the implementation detail – focus on designing the outside view of your code: how people will see and use it.

Green – You are now in an unstable state. Write simple code to make the test pass.
Your aim is to get the code back to a safe, passing state as quickly and simply as you can. Make the minimum changes to your code to make the test pass.
Do not try and fix, or improve, everything at once: Small Steps, one at a time.

Refactor – Rework code and test, to make them clearer, more expressive, more general and better designed overall.
Focus on the design of the implementation detail. Your tests are passing. Now you can safely make small changes, using tests to reassure you that everything still works, while making your code more general, easier to read and overall better.

In TDD we refactor to make our code simpler, more readable, more concise, more general and easier to maintain, and obviously, more testable!
Tools can help in this. Look for good refactoring tools that support your language, such as: Jetbrains, Eclipse, Vs Code, Xcode, etc
Adopt the discipline of always leaving the codebase in a better state after every commit.

Refactoring for Legacy Systems

Refactoring means – Behaviour-Preserving Change, if you refactor and it changes what your code does, it isn't refactoring!

A Strategy for Refactoring

- Add tests only for code that you are changing. Use Approval Tests or Acceptance Tests to defend the code that you want to change.
- When making changes, prefer design choices that enhance modularity and cohesion.
- Reduce coupling through separation of concerns and abstraction.
- Do Not 'retrofit' TDD-style Unit Tests to legacy code.
- Treat boundaries between modules and services, components and sub-systems, with caution. They should change more slowly than other parts of the code. Defend these boundaries with more loosely-coupled design strategies and with contract testing.

  Use the "Ports & Adapters" design strategy to insulate these parts of the system.

The Shape of Your Tests

Tests should be simple, expressive, and focused on a single outcome. This starts to say something about what good tests look like.

The aim is to use the properties of good tests to keep each test simple and focused.

This makes their intent clearer, and it means that they are usually less tightly-coupled to the system under test. This, in turn, frees us to more easily change our code without invalidating the test. Which is an important property of good testing.

Listen to Your Tests

TDD can help you to understand if your design is good, if you listen to the messages that your tests and design are sending to you as you proceed.

If a test is hard to write, this is a sign that the code, and or your design, may be too complex.

Focus on just the current step, and make it easy. Decompose code into smaller steps.

Pay attention to the complexity of your tests. If your tests are big or complex, it usually means that either you didn't practise TDD, and wrote the test after the code, or that your design is bad. Don't try and fix this in the test, fix the design problem instead.

Changing Your Design

Although TDD will help you to a better design for your software, over time, your understanding of the problem will inevitably change. You will face new problems and may think of better ways to solve old ones. So you will want to be able to re-visit your design choices.

If you have used TDD to develop modular code, with loose coupling and good separation of concerns, this will be easier, but it can still be a trial!

At this point, break the restructuring of tests and code into small steps, using refactoring techniques. Steer the code and tests in small steps toward your new model. Work to keep your tests passing for as long as possible while doing this.

This will still be a bit of a pain, changing your mind always comes at a cost. If you aim to work to be perfect first time, either your problem is trivial or you are in for disappointment. At this point remember all the times when TDD made life easier, and your code better, because now you need to pay for some of those advantages.

Treat the ability to safely change your code, at any point in its life, as a good measure of its quality.

Testing at the Edges

Code that touches the edges of our system (i.e user interfaces, and storage, remote comms, in fact any form of I/O really) is more difficult to test.

The points where our code interacts with hardware or people are more complex. So minimise this code! Aim to maximise your ability to test everything else easily and thoroughly.

You can do this by improving the abstraction and separation of concerns in your design, particularly at the points where the code touches these edges. This has the advantage of making your code more testable, but also making it more flexible in general.

Use abstraction to hide the details of these I/O interactions. Aim to make the interface to this "edge-code" as simple as makes sense. For example:

```
storeAccount(Account account)
```

means the code calling the store has abstracted the problem of storage, and so doesn't know how things work, while:

```
sqlCommand("UPDATE account_table SET (id = 1, name = my-account, …)) WHERE …)
```

Has leaked the details of storage into logic that doesn't need to care.

Pick simple generic cases for edge tests. DO NOT try to test all behaviours through those edges. Once you can "store an account" who cares if you are storing it because an address has changed, or because a phone number was added?

Unit test pieces and use integration tests to validate interactions.

Practise

TDD improves the skills of the developer and the quality of the code they write. It is worth investing the time and effort to learn these techniques.

Set expectations as a team. Agree together to practise, and encourage each other to do so.

To reinforce your learning, regularly practise katas to develop your skills - when starting out we recommend that you practise katas for at least 30 minutes every day for 2 weeks. Be diligent, and disciplined, in applying the skills you learned.

Start with a failing test, only write code that is demanded by a test, use Red, Green, Refactor.

Further Learning, Reading & Viewing

Cyber Dojo - To practise Coding 'Katas' and TDD exercises in a variety of languages.
https://cyber-dojo.org/creator/home

TDD & BDD: Design Through Testing - a comprehensive self-paced study training course, with practical exercises and demos by Dave Farley.
https://courses.cd.training/courses/tdd-bdd-design-through-testing

Books*:

"Test Driven Development: By Example" (The Addison-Wesley Signature Series), Kent Beck.
https://amzn.to/2NcqgGh

"Growing Object Oriented Software Guided by Tests", by Nat Price & Steve Freeman.
https://amzn.to/2Lt3jho

"Refactoring: Improving the Design of Existing Code" (Addison-Wesley Signature Series) by Martin Fowler.
https://amzn.to/30ntgaK

"Working Effectively with Legacy Code", by Michael Feathers.
https://amzn.to/3hP0F4z

YouTube Videos:

Subscribe to https://www.youtube.com/c/ContinuousDelivery and find helpful TDD videos via this playlist:
https://www.youtube.com/playlist?list=PLwLLcwQlnXByqD3a13UPeT4SMhc3rdZ8q

*We use Amazon Affiliate links for books. If you use these links to buy a book, Continuous Delivery Ltd. will get a small fee for the recommendation with NO increase in cost to you.