14

Testing Disciplines

Our profession has come a long way in the last three decades. In 1997, no one had heard of testing disciplines such as TDD[1] or TCR.[2] For the vast majority of us, unit tests were short bits of throwaway code that we wrote to make sure our programs "worked." We would painstakingly write our classes and methods, and then we might concoct some ad hoc code to test them. Typically, this would involve some kind of simple driver program that would allow us to manually interact with the program we had written.

1. Test-driven development.

2. Test && commit || revert.

I remember writing a C++ program for an embedded real-time system back in the mid-'90s. The program was a simple timer with the following

signature:

```
void Timer::ScheduleCommand(Command* theCommand, int
```

The idea was simple; the `execute` method of the `Command` [3] would be executed in a new thread after the specified number of milliseconds. The problem was how to test it.

---

3. See the Command pattern in [GOF95].

I cobbled together a simple driver program that listened to the keyboard. Every time a character key was pressed, it would schedule a command that would print the same character five seconds later. Then, I tapped out a rhythmic melody on the keyboard and waited for that melody to replay on the screen five seconds later.

"I … want-a-girl … just … like-the-girl-who-marr … ied … dear … old … dad."

I actually sang that melody while typing the "." key, and then I sang it again as the dots appeared on the screen.

That was my test! Once I saw it work and demonstrated it to my colleagues, I threw the test code away.

As I said, our profession has come a long way.

Nowadays, I write tests that make sure every nook and cranny of the code works as I expect it to. I isolate my code from the operating system rather than creating hard dependencies upon it. If I were to write the above example today, I would use test doubles to "mock" out the OS timing functions so that I had absolute control over the time. I would schedule commands that set boolean flags, and then I would use my tests to step the time forward, watching those flags and ensuring that they went from false to true just as I changed the time to the right value.

Nowadays, once I get a set of tests to pass, I clean those tests to make sure that they are well designed and are strongly decoupled from the production code. I also ensure that they are convenient to run for anyone else who needs to work with the code. I keep the code and tests together in the same source package.

In the early days of Extreme Programming and Agile, we only talked about one testing discipline: TDD. When I wrote the first edition of this book, TDD was the only testing discipline that I presented. In the intervening years, it has become clear that there are other testing disciplines that are as effective, or nearly as effective, as TDD. It has also become clear that there are many programmers who prefer those other disciplines for one reason or another.

And so while I will remain an adherent to the discipline of TDD, I feel that it is appropriate to talk about two other disciplines that are compatible with writing clean code. And, although I am only discussing three such disciplines, there may well be others.

Discipline 1: Test-Driven Development (TDD)

This discipline was proposed by Kent Beck in the mid- to late ’90s and became strongly associated with Extreme Programming and Agile development.

By now, everyone knows that TDD asks us to write unit tests first, before we write production code. But that rule is just the tip of the iceberg.

TDD is not just writing tests first. TDD is an intricate discipline, not a simple rule of thumb. We practitioners do not write a whole batch of tests up front and then, one by one, make them pass. The discipline is far more intricate and intimate than that.

The Three Laws of TDD

What follows are the three laws of TDD. These laws are the basic foundation of the discipline. Following them is very hard, especially at first. Following them also requires some skill and knowledge that are hard to come by. If you try to follow these laws without that skill and knowledge, you will almost certainly become frustrated and abandon the

discipline. We will address that skill and knowledge in subsequent chapters. For the moment, be warned. Following these laws without proper preparation can be very difficult.

The First Law

Write no production code until you have first written a test that fails due to the lack of that production code.

If you are a programmer of any years' experience, this law may seem foolish. You might wonder: What test am I supposed to write if there's no code to test? This question comes from the common expectation that tests are written after code. But, if you think about it, you'll realize that if you can write the production code, you can also write the code that tests the production code. It may seem out of order, but there's no lack of information preventing you from writing the test first.

The Second Law

Write no more of a test than is sufficient to fail, or fail to compile. Resolve the failure by writing some production code.

Again, if you are an experienced programmer, then you likely realize that the very first line of the test will fail to compile because that first line will be written to interact with code that does not yet exist. And that means, of course, that you will often be unable to write more than one line of a test before having to switch over to writing production code.

The Third Law

Write no more production code than will resolve the currently failing test. Once the test passes, write more test code.

And now the cycle is complete. It should be obvious to you that these three laws lock you into a cycle that is just a few seconds long. It looks like this:

- You write a line of test code, but it doesn't compile (of course).
- You write a line of production code that makes the test compile.
- You write another line of test code that doesn't compile.

- You write another line or two of production code that makes the test compile.
- You write another line or two of test code that compiles, but fails an assertion.
- You write another line or two of production code that passes the assertion.

You saw this cycle being played out in the previous chapter. Watching that cycle is the best way to understand the tight feedback loop. I wrote a book sometime back titled *Clean Craftsmanship*. It has many examples of the TDD cycle. There are also some videos of those examples that you can see here: informit.com/cleancode2.

Discipline 2: Test && Commit || Revert (TCR)

This discipline was proposed[4] by Kent Beck in 2018. TCR does not insist that tests be written first. Rather, it insists that code that passes tests should be immediately committed, and code that fails tests should be immediately reverted. This is generally enforced with a script that runs the tests whenever source files are saved and automatically executes the commit or revert based on the test results.

---

4. https://medium.com/@kentbeck_7670/test-commit-revert-870bbd756864

The immediate reversion on failure forces the programmers to move very carefully, and in very small steps. The programmer can write as much test and production code as they like, but they are liable to lose what they wrote if they have not written tests that pass. Thus, they will write tests and code together in very short cycles. Those cycles are as small as those in TDD.

Personally, I find TCR to be stressful, while I find TDD to be relaxing. But others may find TCR to be better for their mindset because they are not forced into writing tests first.

Discipline 3: Small Bundles

This discipline was proposed to me by John Ousterhout while I was in the process of writing this second edition. It forces neither the ordering of the

tests and code, nor the intensely small cycle of TDD and TCR. The cycle time is longer, probably by an order of magnitude. Minutes as opposed to seconds.

The idea is simple. Programmers write a small bundle of code and tests to achieve a certain result. They might write the code first. They might write the tests first. They might intersperse the writing of the code and tests. But they will conclude with that small bundle having high test coverage and all tests passing.

If you are used to TDD or TCR, you might feel that the bundling discipline is a watered-down compromise. I disagree. It is perhaps a softer ritual; but it can be just as effective. Any discipline can be abused, and any discipline can be honored. If programmers honor this discipline by keeping bundles small and testing every branch and condition, then they will produce a test suite that they can trust with their lives. Therefore, it is entirely compatible with writing clean code.

Design

You might object that disciplines like these are too tactical because they make no room for strategic design. However, testing disciplines are not intended to supplant strategic thinking. I, for example, am a big believer in hammock-driven development.[5] Thinking up front is vital in all software endeavors—so long as that up-front thinking is brought back to reality on a frequent basis. Months of strategic planning, without any coding, is suicide. Coding without any strategic planning is equally suicidal.

---

5. https://www.youtube.com/watch?v=f84n5oFoZBc

After an appropriate amount of strategic design, programmers must become tactical. After all, the lines of code have to get written, and writing them is tactical. The testing disciplines described above are tactical disciplines—they augment the tactic of writing code.

Moreover, these disciplines test the design of the code by imposing the viewpoint of users—after all, the tests are a kind of user.

This imposition of viewpoint has led many practitioners to consider these disciplines to be design disciplines as well as coding disciplines. Personally, I find that TDD assists in verifying my lowest-level designs, but it does not help much in higher-level design and architecture. I have seen some pretty bad designs implemented with TDD. Some are in this very book.

Discipline

Disciplines like these should be taken seriously; but they are not universally applicable and should not be followed blindly. For example, I am a pilot; I follow the discipline of using checklists instead of committing many complex procedures to memory. I take that discipline very seriously. There are, however, certain procedures that I must commit to memory because time is of the essence, and the time it would take to pull out and follow a checklist would be unsafe. Pilots call these procedures *memory items*, and the learning and refreshing of memory items is yet another discipline.

Tedious, Boring, and Slow

Any programmer who is unfamiliar with these disciplines, and who has been a programmer for more than a few years, may find these descriptions horrifying. They may sound tedious, boring, and slow. You might infer that you'd never be able to think through a problem. With TDD and TCR, it even seems likely that you'd never be able to write a little `while` loop or `if` statement without interrupting yourself. And all three will certainly break your concentration and interrupt your flow. In short, you could conclude that these disciplines are stupid.

But consider.

Debugging

Imagine a room full of programmers following one of these disciplines. Walk up to any programmer you like, at any time you like. Everything they are working on executed and passed their tests sometime in the last few minutes. And it doesn't matter who you pick or when you pick them: Everything worked a few minutes ago.

What would your life be like if everything worked a few minutes ago—and this was always true? How much debugging would you do?

It's hard to believe you would debug much if everything worked so recently. The most probable debugging technique would be to revert back to the last working version and just try again.

Those of us who follow these disciplines have little use for debuggers. Oh, I'll fire one up from time to time. I mean, this is still software, and it's still hard. Weird things still happen. But the time I spend in the debugger is vanishingly small. So small, in fact, that I don't remember the hot keys. Every time I try to use the debugger I have to reacquaint myself with how to operate it.

Following one these disciplines (TDD) has reduced my debugging time by a very significant factor. I contend that any of these disciplines will do the same for every adherent who masters them.

Documentation

Have you ever integrated a third-party package? Perhaps you downloaded a zip file from a website. Upon unzipping, you find the artifacts you need to integrate, but also a PDF manual with instructions to help with the integration. At the end of that manual, there is an ugly appendix with all the code examples. And, of course, that's the first place you go.

You go to the code examples because you hope that the example code is current and that it will tell you more truth than the words within the manual. If you are very lucky, you can copy and paste that code from the manual into your project and fiddle it into working.

When you follow a testing discipline, you write a series of tiny little tests. Each one of those tests describes how a part of the system works. If you want to know how to create an object, there are tests that create that object every way it can be created. If you need to know how to call a certain API, there are tests that call that API every way it can be called. These tests are the code examples for the whole system.

What's more, each one of those tests is isolated from all the others. The tests themselves are not a system. Each is understandable in its own right.

The tests produced are a document that describes the low-level design of the system. And that document is written in a language that you intimately understand. It is utterly unambiguous, it is so formal that it executes, and it cannot get out of sync with the application that it describes. The tests are an almost perfect form of low-level documentation.

That's not to say that they are high-level documentation—they are not. But at the lowest level of the system, the tests are about as good a document as can be created.

Reliability

Imagine a programmer who does not follow one of these disciplines. He has just finished writing a module and has run it through a battery of manual tests. It seems to work just fine. But then the process proctor reminds him that he needs to write unit tests. He feels that this is redundant make-work since he's already tested the module, so his effort is less than stellar.

He writes a few tests here and there, and they pass just fine. But then he encounters the module that's hard to test. It's hard to test because it was not designed to be testable. It is coupled to things that should not be executed in the context of a test.

He could redesign that module to make it testable; but that would take a lot of time that he doesn't think he has. And besides, his manual tests proved to him that everything already works. So he walks away without writing that test.

Now, if you've been a programmer for more than a few years, you may recognize this scenario. Many of us have written tests after the fact; and many of us have walked away from a difficult test because we didn't want to change a "working" design.

Unfortunately, this leaves a hole in the test suite. And if you have left holes in the test suite, you know everyone else has left some holes as well. And

so everyone knows that the test suite is full of holes.

What happens when you run such a test suite and see it pass? Everyone smiles knowingly and no other action is taken. When you know that the test suite is full of holes, there is no safe decision you can make when it passes. All you can do is smile, knowing all too well that the passing of the test suite does not mean that the system works properly.

When you follow a testing discipline, things are very different. When the tests pass, you know with a high degree of confidence that the system works as it should. And that allows you to make critical decisions.

For example, you could decide to *deploy* the system. I've worked on systems like this. When the tests passed, we deployed because we had very high confidence in our test suite.

Even if deployment is not an option, the passing of the tests could allow you to move the system into a new stage. For example, you could move it from DEV to QA.

Design

Something else happens when you follow these disciplines. You cannot write the module that's hard to test. By writing the test along with the code, you are automatically designing every module to be easy to test.

A module that is easy to test is decoupled. It is the decoupling that makes the module easy to test. So the end result is a less-coupled design. The mere act of writing the tests along with the code improves the overall design of the system.

Reprise

So, by following these disciplines, you will in all likelihood:

- Significantly reduce your debugging time.
- Create a stream of nearly perfect low-level documentation.
- Create a test suite that you trust so much that you can deploy when it passes.
- Create a system with a less coupled design.

And while these four things are good—very good, in fact—they are not the reason that we follow a testing discipline.

The Angel and the Devil

In Chapter 1, "Clean Code," we discussed the cost of making a mess. I described the way that the ever-increasing mess degrades productivity. I introduced the Boy Scout Rule and told you that keeping the code clean was the only way to prevent that loss.

So how do you keep the code clean?

Imagine that you are sitting at your computer, scrolling through some code you need to work on. As you scroll, the blood drains from your face and you realize that this code is an ungodly mess that is going to be hell to work with.

Just then, a little angel appears on your right shoulder and whispers in a still small voice: "You could clean it."

But then, in a gout of flame, a devil appears on your left shoulder and screams: "NO! Don't touch it! If you touch it, you will break it; and if you break it, it will become yours FOREVER. Muahahahahahaha!"

And so you ignore the angel and back away from the code. You leave it in its current state. You leave it for someone else to deal with. You leave it to fester and to rot and to pull down the productivity of the team.

This is a fear reaction. You are afraid to clean the code.

I want you to think very carefully about how wildly unprofessional and irresponsible it is to have allowed the code to get so far out of your control that you are afraid of it. You react to it, rather than it reacting to you. You defer to its demands, rather than forcing it to conform to yours. You, the creator, have lost control of your creation and have become a minion of its whims.

And because of that fear, the only thing that can happen to the code in that system is that it *must* rot. No one dares do the one thing that might reverse

that rot—cleaning. And so down it will go, with every month getting worse and worse, and dragging the productivity of the team down with it.

And if you think that's an exaggeration, talk to some of the older programmers you know. They'll tell you.

Muzzling the Devil

Let's revisit that scenario again. But this time let's say that the team has been practicing a good testing discipline and has built up a very good test suite.

There you are, looking at your screen, scrolling through the code. You see that bad code and the blood drains from your face as you stare into the depths of its horror. And, as expected, the little angel whispers in that still small voice: "Clean it!"

And the devil does not appear. The devil has no power. Because you have the test suite.

So you make a small change, perhaps just the name of a variable. And you run the tests. And the tests pass and you know you haven't broken anything. So you make another small change, perhaps splitting a longer function in two. And again, the tests pass. So you move one of those new functions to a different class, and the tests fail! So you quickly revert and see the tests pass. Then, you realize your mistake and you move the function again, but this time correctly, and the tests pass.

If you have a test suite, and if you trust that test suite, and if that test suite runs quickly, then you *will* clean the code. You will, because cleaning the code is virtually risk free. And it won't be just you cleaning the code—everyone on the team will clean the code. Everyone on the team will follow the Boy Scout Rule and continuously improve the quality of the code, keeping productivity high.

Complications and Loopholes

If that last discussion sounded too good to be true, it is. Oh, it can be true for a very large fraction of a system; but the devil still hides in the details.

There are situations where writing tests is either impossible or at least impractical.

For example, it is impractical, if not downright impossible, to write tests for code that communicates outside the hardware boundary of the machine. For example, unless you somehow use a camera that's staring at the screen, it is impossible to test the final output of a graphical user interface. The same holds true for testing the movement of the mouse, or the data going in and out over a socket. Automated tests for these things are not quite impossible; but in most cases, writing them is deeply impractical.

Therefore, seasoned developers will design their systems such that those untestable stretches of code are very thin. They will move as much intelligence as possible away from the hardware boundaries and ensure that the code that does finally touch those boundaries is as anemic as possible.

This technique is called *The Humble Object Pattern*.[6] I talk in much more detail about these kinds of situations in my book *Clean Craftsmanship*.

---

6. http://xunitpatterns.com/Humble%20Object.html

Another complication can occur when using third-party frameworks. If a framework was not written to be testable, then the code that uses that framework may also be impractical to test. Again, good designers will anticipate this and attempt to push as much intelligence away from the framework as possible and build an isolation layer between the framework and the code that needs to be tested. Care should be taken when committing to frameworks, because they have the potential to prevent very large swathes of code from being properly tested.

Experience Report:

I recently wrote a significant application that used the Java Swing framework. Trying to test any of the code that touched Swing was impractical. I had to separate the application into a tested component and an untested (Swing) component. That untested component grew fairly large

because the Swing API forces the construction of large, interconnected data structures that contain callbacks. This left my application with ~70% test coverage. I could easily refactor the tested component; but touching the Swing component was always very risky. I won't use Swing again.
Still another complication occurs when you write a function, but you don't know the correct results of that function. For example, it is hard to test whether a font looks "right," or whether a field is positioned "correctly," or whether the format of a report looks "good." In all of these cases, the results are subjective. The only way to ascertain whether the code is correct is to look at the results and make a judgment. There's no way to write a test for that. So there's no way to write the code and tests together. It's also doubtful that writing a test long after the fact is very helpful, since all it can manage to do is cement a subjective decision.

These three cases are the only situations that I have encountered in which tests are impractical. There are, however, likely to be others. For example, how do you test the results of an AI?

In the end, when faced with areas of the system that are impractical to test, try to make those areas as small and simple as possible, and protect the rest of the system from them.

Costs and Repercussions

If we follow these disciplines, we will write dozens of tests every day, hundreds of tests every month, and thousands of tests every year. Those tests will cover virtually all of our production code. The sheer bulk of those tests, which can rival the size of the production code itself, can present a daunting management problem.

Keeping Tests Clean

Many years ago I was asked to coach a team who had explicitly decided that their test code should not be maintained to the same standards of quality as their production code. They gave one another license to break all the rules of clean code and good design in their unit tests. "Quick and dirty" were the watchwords. Their variables did not have to be well named; their test functions did not need to be short and descriptive. Their test code did not need to be well designed and thoughtfully partitioned. So long as the

test code worked, and so long as it covered the production code, it was good enough.

Some of you reading this might sympathize with that decision. Perhaps, long in the past, you wrote tests of the kind that I wrote for that `Timer` class mentioned at the beginning of this chapter. It's a huge step from writing that kind of throwaway test to writing a suite of well-designed, well-partitioned, automated unit tests. So, like the team I was coaching, you might decide that having dirty tests is better than having no tests.

What that team did not realize was that having dirty tests is equivalent to, if not worse than, having no tests. The problem is that tests must change as the production code evolves. The dirtier the tests, the harder they are to change. The more tangled the test code, the more likely it is that you will spend more time cramming new tests into the suite than it takes to write the new production code. As you modify the production code, old tests start to fail, and the mess in the test code makes it hard to get those tests to pass again. So the tests become viewed as an ever-increasing liability.

From release to release the cost of maintaining this team's test suite rose. Eventually, it became the single biggest complaint among the developers. When managers asked why their estimates were getting so large, the developers blamed the tests. In the end, they were forced to discard the test suite entirely.

But, without a test suite, they lost the ability to make sure that changes to their codebase worked as expected. Without a test suite, they could not ensure that changes to one part of their system did not break other parts of their system. So their defect rate began to rise. As the number of unintended defects rose, the devil's voice grew louder, and they started to fear making changes. They stopped cleaning their production code because they feared the changes would do more harm than good. Their production code began to rot. In the end, they were left with no tests, tangled and bug-riddled production code, frustrated customers, and the feeling that their testing effort *had* failed them.

In a way, they were right. Their testing effort had failed them. But it was their decision to allow the tests to be messy that was the seed of that failure. Had they kept their tests clean, their testing effort would not have failed. I

can say this with some certainty because I have participated in, and coached, many teams who have been successful with *clean* unit tests.

The moral of the story is simple: Test code is just as important as production code. It is not a second-class citizen. It requires thought, design, and care. It must be kept as clean as production code.

Tests Enable the -ilities

If you don't keep your tests clean, you will lose them. And without them, you lose the very thing that keeps your production code flexible. Yes, you read that correctly. It is tests that keep our code flexible, maintainable, and reusable. The reason is simple. If you have tests, you do not fear improving the code! Without tests, every change is a possible bug. No matter how flexible your architecture is, no matter how nicely partitioned your design is—without tests, you will be reluctant to make changes due to the fear that you will introduce undetected bugs. Without tests, you fear to clean the code.

But with tests, that fear virtually disappears. The higher your test coverage, the lower your fear. You can make changes with near impunity to code that has a less-than-stellar architecture and a tangled and opaque design. Indeed, you can improve that architecture and design without fear!

So, having an automated suite of tests that cover the production code is the key to keeping your design and architecture as clean as possible. Tests enable all the -ilities, because tests enable change.

So, if your tests are dirty, then your ability to change your code is hampered, and you begin to lose the ability to improve the structure of that code. The dirtier your tests are, the dirtier your code becomes. Eventually, you lose the tests, and your code rots.