Chapter 3
Define Tests Up-Front

With the advent of agile methods, Test-Driven Development (TDD) has been gaining momentum. A mantra of agile is that stories are ***completed****, not merely written, every iteration. This means they have to go through testing to be considered "done, done, done." Many teams have experienced the productivity gains and value of TDD. Many teams have, unfortunately, shied away from it as well. We believe that the value and reason that TDD works are not fully appreciated. This chapter begins by defining testing and then discusses both why TDD works, and why it isn't really testing up-front.*

A Trim Tab: Testing and Testability

As mentioned in the preface, this book represents the set of trim tabs1 we, at Net Objectives, consider to be most useful for enhancing the productivity of software developers. We consider the issue of testability (the focus of this chapter) to be, perhaps, the greatest of these. Hence, we could say this chapter is about the trim tabs of trim tabs.

What Is Testing?

Merriam Webster's dictionary defines test as *"the procedure of submitting a statement to such conditions or operations as will lead to its proof or disproof or to its acceptance or rejection."* This is testing as an action. However, a test can also be a noun, something that is *"a basis of evaluation."* We're sure you recollect a time someone put a "test" on your desk and then you had to take it. The test you were given in this case specified what you needed to know in order to get a good grade. The action of taking the test is something different altogether.

In the same way, tests in software are about what the software needs to do in order to be considered successfully implemented. This is why we can write tests before we have code to test. We are specifying what the software needs to do. We would suggest this insight leads to the observation that test-first is really analysis first using tests. In other words, we use the tests to determine the behavior we want of the functionality we are testing. This is a form of analysis.

But it is more than that – it is also a type of design using tests to accomplish the design. That is, simultaneously with the analysis, we are figuring out how to implement the interfaces of the functionality. We are splitting the classes up into their methods. We are, in essence, doing design.

[1] If you've not read the preface please do so now – even if you understand what a trim tab is.

Testability and Code Quality

Why is this useful? We suggest it's because testability is highly correlated to the code qualities we want to manifest, in particular, loose coupling, strong cohesion, and no redundancy. We can recollect times that at the start of testing our code we have remarked:

> "I can't test this code, it does too many things that are so intertwined" (weak cohesion)
>
> "I can't test this code without access to dozens of other things" (excessive coupling)
>
> "I can't test this code; it's been copied all over the place and my tests will have to be duplicated over and over again" (redundancy)
>
> "I can't test this code, there are too many ways for external objects to change its internal state"

We've often summed it up by saying – "gee, I wish they had thought of how this code was going to be tested while they were writing it!"

I'm kind of slow sometimes because it took me quite some time to realize (comment by Alan):

I should consider how my code is going to be tested before writing it!

The reason is clear – testability is related to loose coupling, strong cohesion, no redundancy and proper encapsulation. Another way to say this is that the tighter your coupling, the weaker your cohesion, the more your redundancy and the weaker your encapsulation, the harder it will be to test your code. Therefore, making your code easier to test will result in looser coupling, strong cohesion, less redundancy and better encapsulation.

This leads to a new principle:

Considering how to test your code before you write it is a kind of design.

Since testability results in so many good code qualities and since it is done before you write your code, it is a very highly leveraged action. That is, a little work goes a long way – it's a great trim tab.

Case Study - Testability

Let's look at a case study. Let's say we have a piece of hardware whose status we need to detect (i.e., see if it is functioning properly). After detecting the status, we need to send an encrypted message via TCP/IP to some monitoring device. If we were to ask you how to design the system, one of four choices probably comes to mind:

1. One class, one method that does it all (get status, encrypt, and transmit).
2. One class, 4 methods
    a. Control method that calls the other three methods that follow
    b. `getStatus()`
    c. `encrypt()`
    d. `transmit()`
3. Three classes, one or two methods in each
    a. `Hardware.getStatus()`
    b. `Encrypt.encrypt()`
    c. `Transmit.transmit()`

d. `Hardware.getEncryptSendStatus()` calls the prior three methods

4. Four classes (a variant on the prior case where we have the control program in a separate class.

We can get into several "religious" conversations here as to which one is better. Note that the first one has poor cohesion and the likelihood of coupling the methods together. If we start with that and need to modify the code (e.g., use another encrypter) it may be difficult to do so.

Let's look at what happens if we design our code here with the simple mandate of making it as testable as possible. In this case, most people will select cases 2-4. Actually, most will pick 3 with a handful picking 2 or 4. Now, from a testability point of view, we would say 3 or 4 is the easiest. Why? What easier way of testing encryption is there than if it is in its own class? The same is true for the hardware and the transmitter. Some will say that as long as each function is in its own method and each method uses variables only local to the method then it would be easy enough to extract the method out when needed. It's kind of hard to win an argument for separate classes here if the person you are arguing with doesn't like lots of classes (or has misunderstood XP's mandate of fewest number of methods and classes). They can always say – look the code is easy enough to see – why add the extra classes?

This may not be a problem at this point. However, when methods like this are lumped together, we often takes advantage of the fact that even private members are public to the classes' members. Hence, refactoring out the methods may or may not be easy. We also point out that you must instantiate the entire object – which may or may not be simple. For example, you may have to provide a valid TCP/IP connection to instantiate it. However, with the understanding that lumping all the methods together is only acceptable until one of the methods starts to vary (and then we'll have to pull it out) we wouldn't argue too long here. You should readily observe that considering the testability of the code here creates a better design.

By the way, it is worth noting that we are not suggesting that we have only one method per class. We are suggesting you have highly cohesive classes. In this example, each entity type didn't have a lot of functionality, so that's how things worked out.

Setting Ourselves Up For Change

You should note that the results we'll get with this approach will be code that is highly modularized, cohesive and loosely coupled. This means we should be prepared for changes in the future. We'll take this a step further in the "Refactoring to the Open-Closed Principle" later in the book that'll show how design from test specifications such as we've done here sets the stage for extending our designs in an efficient manner.

Programmer as Frog

This brings up an interesting point. Why are programmers like frogs? In our classes we've gotten some pretty interesting answers to this question. They include we like finding bugs, we've got warts, and you've got to kiss a log of programmers before you find your prince charming. But we're not looking for any of these answers. ☺

It turns out, what we are referring to is actually an urban legend (we looked it up; we didn't try it, so don't report us to the ASPCA). It used to be believed that if you put a frog in hot water it'd immediately jump out. However, if you put it in a pot of room temperature water and placed the pot on the stove and heated it up, the frog would stay in the pot until it boiled to death. Now we know we would hop out of a pot that was on a stove (probably even before it got too hot!). So why are we like frogs? Because as a developer we don't notice the slow degradation of our code.

We've all seen this in something we call "switch creep." The first switch we have isn't bad. The second one isn't bad either. But somewhere around the 57th switch, the water temperature is pretty up there!

This leads to a habit developers should get into – don't degrade your code! Or, at least if you must, do it intentionally! That is, do it only when you know you are doing it. This takes discipline as well as team buy-in. But if you can do that, it also enables you to avoid slowly degrading your code.

We heard Ward Cunningham once say "spend as much time as you need to build the best quality code you can, but don't add any functionality that you don't need right now."[2]

A Reflection on Up-Front Testing

As we were saying, up front testing really isn't testing at all. It is really up front design through the analysis of our tests. Can we take this testing even further? When XP came out and suggested doing unit tests, many of us realized that if we combined a series of unit tests together we could get the equivalent of automated acceptance testing.

However, there was a better way. With the invention of FIT (Framework for Integrated Testing) defining acceptance tests became a separate process from combining unit tests together. We now had an easy method to have non-programmers define and (virtually) implement acceptance tests. These two testing practices were still often practiced separately.

Eventually we came across Rick Mugridge's FIT For Developing Software (the first 50 pages of which are a must read for all developers in our mind regardless of your testing practices). He eloquently states how defining acceptance tests up front improves the quality of the conversation between QAs, BAs, and devs. Given that these people will (or at least should) reflect on acceptance test definitions at some point, this implies we should pick the most beneficial time to do this – and that'd be *up-front* (for a full exploration of this, please look at Appendix C, *The Role of Quality Assurance in Lean-Agile Software Developmen*t and *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration*).

This creates value (better conversation, better understanding, clearer scope) without creating extra work. This leads to the insight that we should not drive our acceptance tests from our unit tests, but rather do it the other way around since we need to be creating our acceptance tests first. In other words, our unit tests need to be created within the context of manifesting the behavior our acceptance tests dictate.

As a disciple of Christopher Alexander,[3] this makes sense in another way. Alexander's work proposes that designing from context (the big picture) creates better designs. However, we believe this is a broader principle than just a design principle. In general, we suggest keeping in mind why you are doing anything (the context) improves whatever it is you are doing (whether it is design or something else).

A big part of agile software development is discovering what the customer wants or needs. In doing this, one writes stories. Following Alexander's ideas here would mean that we should start with the big picture of the behavior we want, then go into the details. Again, this means driving unit tests within the context of the desired behavior. Hence, TDD is not even design first as much as it is analysis and then design. So perhaps we have Test Driven Analysis and Design and then Development (TDADD?).

But are we really testing first? We don't think so. Let's look at another activity – making and using plans. We can talk about planning in three ways:

- Planning (the action of making the plan)

[2] Said at an eXtreme Programming Seattle User Group meeting circa 2006.

[3] Christopher Alexander. *The Timeless Way of Buildin*g. New York: Oxford University Press, 1979

- Plan (a description of steps we intend to perform)
- Following the plan (doing these steps)

Now in testing, we have words for steps 2 and 3 above, but not for step one. For example, the equivalent of a plan is a test specification. The equivalent of following the plan is running the tests. But what is the equivalent of planning? We would say it is creating the test specification. In other words, we have:

Plans	**Tests**
Planning	Creating the test specification
Plan	Test specification
Following the plan	Running the test

So, what we actually do is up-front Test-Specification (OK, so we're being picky – but a test specification is different from running the test). Test-specifications, of course, are another way of stating what the system needs to do – that is, it is analysis. We guess you could say we do TSDADD (Test-Specification Driven Analysis and Design and Development). No wonder they shortened it to TDD! Funny, however, that it wasn't given an intention revealing name!

The challenge now with the term TDD is that it has a lot of latitude on what it means. Are we starting from the functional level (unit tests - TSDD)? Or the behavior of the system level (acceptance tests - TSDADD)? To avoid confusion in the rest of this article, we are going to refer to what we've been calling TSDADD (tongue-in-cheek), Acceptance Test-Driven Development (or ATDD, which is described in detail in Chapter 7. [4] We will refer to the class writing unit tests first UTDD. As stated earlier, starting from the behavioral level (the big picture) creates better conversations and context. Let's look at the advantages of doing so:

- Better design
- Improving the clarity of scope – avoiding excess work
- Reducing complexity

Better Design

Let's get back to Alexander's hypothesis. Alexander states that designing from context provides insights into what the functionality you are creating needs to do. Designing from the whole enables you to see how the smaller pieces should fit together. This is a fundamental design principle which works in other industries besides his (building construction). Although software development is not at all like building buildings, design patterns are somewhat based on this concept (although we believe there is even more confusion about what design patterns are – see Can Patterns Be Harmful?). One aspect of this is the Dependency Inversion Principle. We should expect then, that doing ATDD should result in better designs than the classic UTDD where we specified the pieces first and put them together.

Improving Clarity of Scope – Avoiding Excess Work

By providing our acceptance tests up front, we help the developer understand what is in the scope of the requirement. This prevents developers from overbuilding the system. This, of course, helps avoid excess work.

[4] ATDD is very similar to Dan North's BDD but has some differences that we will not review here.

You might be concerned that this minimalistic design will not prepare us for future changes. But just the opposite is true. By having an automated test suite, high quality code, and an understanding of quality design (design patterns) you set yourself up to be able to add design changes later, when you both know they are needed and you know how to implement them. This is much more efficient than designing ahead of time.

Another way acceptance tests can help us avoid work is they can give us an indication of how far down we have to go in specifying unit tests. UTDD somewhat requires every function to be tested because the unit tests are your safety net. But in ATDD, the acceptance tests are your safety net – the unit tests are there to help you design your functions and to find errors faster (both good things – but not always necessary).

Reducing Complexity

Overbuilding systems is one of the greatest causes of waste in software development. A function that isn't used doesn't just waste the time it took to write it, it makes the system more complex. This makes adding other functionality or fixing existing errors much more time-consuming. Not to mention that the system is likely to have more errors in it. By clarifying the scope of work in a clearer way, we can help avoid this.

Other Advantages

If you are building software that must meet government testing specifications (e.g., health care instrumentation) it is easier to prove the software is doing what it needs to if you have a full set of automated acceptance tests. If you just have a set of unit tests (even a complete set) you will still have to demonstrate that your unit tests demonstrate acceptance test criteria. If you are going to have to specify these acceptance tests anyway, might as well do them first for all the reasons we've been mentioning.

No Excuses

By the way, there may be certain arguments for not writing and maintaining a set of automated tests. We don't think we would agree with them, or at least, in virtually all cases we are fairly sure we wouldn't. However, there is little argument that can be made for not at least specifying the tests first. This is because you are going to have to specify them at some time. You might as well do it at the time it provides the greatest value.

Summary

There are two kinds of Test-Driven Development. The classic style of writing unit tests first (often called TDD, but what we are now calling UTDD) and the more productive method of writing acceptance tests first and then your unit tests (which we are calling ATDD). Both are really about specifying your tests first--then writing the tests, then writing your code, then running the tests. This creates many advantages. By creating unit tests from the context of your acceptance tests, you will get a better definition of your scope, avoid doing extra work, reduce complexity and achieve better designs.