The Addison-Wesley Signature Series

A Mike Cohn Signature Book

Developer Testing

Building Quality into Software

Alexander Tarlinder

Forewords by Jeff Langr *and* Lisa Crispin

Developer Testing

```
type FederalId = FederalId of string
type StateId = StateId of string
type ListA =
   | PassportOnly of FederalId
type ListB =
   | DriversLicenseOnly of StateId
type ListC =
   | SocialSecurityCardOnly of FederalId
type Identification =
   | PrimaryId of ListA A>
   | TwoValidForms of ListB * ListC
type Employee =
   {
   identification: Identification;
   }
let fedIdNumber = FederalId "C00001549"
let passport = PassportOnly fedIdNumber
let primaryId = PrimaryId passport
let employee = { identification = primaryId }
```

Now we don't need to test for each case because it's literally impossible in our system to be an employee and not have the necessary ID.[e] Instead, we incrementally build valid data, always guaranteed to have a "legal" state. Essentially we have implicit preconditions living in our types. How? It's hiding right in plain sight: we actually have logical operators operating on our types! In F#, sum types are represented with | and product types are represented with *. These correspond to the logical operators $\vee$ (or) and $\wedge$ (and), respectively.

The more you work in this way, the easier it becomes to see how to encode valuable business logic out of preconditions in functions, or out of functions entirely, and into our data types. Does this remove the need for unit tests entirely? In my experience, no. But the more logic you move into your type system, the less you will need to test.

[a] https://docs.eiffel.com/book/method/et-design-contract-tmassertions-and-exceptions.

[b] Even functions have types because in a functional language you have higher order functions that operate on other functions!

[c] Formally, not all operations are "closed under" a type.

[d] https://blogs.janestreet.com/effective-ml-revisited/.

[e] You may ask: What if we need to represent people who aren't yet employees? (And thus don't have ID.) You'd create a new type!

Domain-to-Range Ratio

Speaking of data types and their ranges naturally takes us to this chapter's last piece of theory. How would we test a function, *f*, that supposedly says whether a number is odd or even and returns 0 if it's odd and 1 if it's even?

Given that we accept that 0 is an even number, the first test that comes to mind is calling the function with 0 and comparing the result to 0. Next, we'd probably call it with a 1, and expect 1 in return. Then what? Is *f(10) = 0* a good test? Or maybe *f(9999) = 1*? That depends.

Let's leave the world of software and go for a more mathematical definition. *f* maps the set of natural numbers to [0, 1]. This means that we no longer have to concern ourselves with things like *f("hello world")*. The *range* of *f* is the set consisting of 0 and 1, whereas its *domain* is the set of natural numbers. Given these definitions, the *domain-to-range ratio* (DRR) can be introduced. It's the quotient of the number of possible inputs over the number of different outputs. In a more mathematical language, we can state this as the cardinality of the function's domain over the cardinality of its range (Woodward & Al-Khanjari 2000):

$$DRR = \frac{|D|}{|R|}$$

Why is that interesting? Let's reduce the size of our problem and replace the infinite set of natural numbers with the set of numbers from 1 to 6. Thus, the size of the domain is 6, which makes the Domain-to-Range Ratio equal to 6/2. The measure tells us something about the information loss that occurs when multiple values in the input map to the same output. In the example, three input values map to the same output value, three to another. It would be tempting to create only two test cases for this scenario; after all, there are two reasonable equivalence classes here—odd and even numbers.

Now, suppose that *f* looks like this:

f(1) = 1
f(2) = 0
f(3) = 1
f(4) = 1
f(5) = 1
f(6) = 0

It's almost a function that determines whether a number is even or odd, but it has an exception built in. If there's no test for f(4), we're in for a surprise. This is an example of how bugs can creep into areas that suffer from information loss. The problem is amplified if the input domain (and consequently the DRR) grows. Without getting too formal, we can say that the DRR is a measure of risk; the higher it is, the more unsafe it'll be to have very few tests.

The previous example illustrates how a trivial function with obvious equivalence partitions can include surprises that may remain unfound, unless the DRR isn't considered. Naturally this doesn't mean that we should throw equivalence partitioning out the window. Rather, it means that we should be careful both in situations involving discontinuous large input domains that cannot be easily partitioned and in situations where there's information loss (as indicated by the DRR). It's also yet another reason for keeping data type sizes close to the range of the variable that they hold and to introduce abstractions that uphold invariants and keep the size of the domain down.

Summary

Several constructs and behaviors in code affect testability. *Direct input/output* is observable through a program element's public interface. This makes testing easier, because the tests need only be concerned about passing in interesting arguments and checking the results, as opposed to looking at state changes and interactions with other program elements.

Conversely, *indirect input/output* cannot be observed through the public interface of a program element and requires tests to somehow intercept the values coming in to and going out from the tested object. This usually moves tests away from state-based testing to interaction-based testing.

The more complex state a program element allows, the more complex the tests need to become. Therefore, keeping state both minimal and isolated leads to simpler tests and less error-prone code.

Temporal coupling arises if one method requires another method to be invoked first. Typical examples are initializer methods. Temporal coupling is actually state in disguise and should therefore be avoided if possible.

The *Domain-to-Range Ratio* is a measure of information loss in functions that map large input domains to small output domains, which may hide bugs. It's yet another tool when determining what abstractions to use and how many tests there should be.

Chapter 7

Unit Testing

Unit testing is the professional developers' most efficient strategy for ensuring that they indeed complete their programming tasks, that the code they write works in accordance with their assumptions,[1] and that it can be changed by them and their peers.

A hobby hack written and used by one person doesn't need to have unit tests. One person suffers the consequences of bugs, and if any refactorings take more time than necessary or totally break the project, that's probably fine too. If the project is more about coding for fun than producing something that an actual customer is willing to pay money for and that can be developed and maintained by more than one person for a longer period of time, having no unit tests is a viable strategy.

Why Do It?

Why should you invest time in writing unit tests when working with software professionally? Here are a couple of reasons. Some of them echo arguments made previously in the book, but it doesn't make them less true. Unit tests

- **Enable scaling—**Software development simply doesn't scale without the code being supported by various types of tests, of which unit tests are the base. It's hard to have collective code ownership without unit tests. Having several people or teams working on a codebase that's not covered by tests leads to accidental overwriting of code, regression defects, and us-and-them type of conflicts between teams, at worst, and long release cycles prolonged by days of manual testing, at best.
- **Lead to better design—**Code written so that it can undergo unit testing can't get totally rotten. When developers exercise a unit of work with a test, they'll tend to make it small and to the point, and they'll be mindful of its dependencies. The mere existence of unit tests, or even just the awareness of what it takes to achieve testability at the unit level, will save the code from some of the following:

1. It's tempting to write "works correctly" instead of "works in accordance with their assumptions," but proving that a program is correct is impossible, except for simplistic snippets used in a university course on formal methods.

- Methods with too many parameters
- Monster methods
- Global state (in static classes and singletons)
- Excessive dependencies
- Side effects

Such constructs tend to make developers' lives miserable in the world of untestable legacy code.

- **Enable change—**Adding and removing features to software requires redesign and refactoring, as do smaller changes. Whoever makes the changes to one part of a system has to know what other parts need to be re-executed to verify that it hasn't been broken. This effectively stops developers that are new on the team from making changes to critical areas of the system, because they can't possibly know what to retest or rerun. Not only that, but even more seasoned developers will refrain from changing and refactoring code if they risk breaking it in some unforeseen manner. Automated tests, and among them unit tests, provide the safety net needed to make changes without fear of unexpected breakdowns.
- **Prevent regressions—**In the absence of tests, the only practical way to verify that the software seems to be working is by running it. There are some downsides to this approach. First, running the software over and over again to verify that a certain part of it seems to work (the part that's just been written or modified) is monotonous and boring. Second, as pointed out previously, it's not always obvious what to rerun. Third, time is not unlimited. As the system grows, manual testing will be able to cover a smaller and smaller fraction of its functionality, and doing exhaustive regression testing will be impossible. A suite of unit tests executed by the developer while changing the code, along with a build server running the tests on a continuous basis, will catch regressions in areas covered by tests almost as soon as defects are introduced.
- **Provide a steady pace of work—**Writing unit tests is a way to achieve and maintain a steady pace of work. Code written in tandem with tests tends to lead to fewer surprises or last-minute problems. If everything implemented up to a point is passing unit tests, it most likely works on a functional level at least. Furthermore, if a bug is found in code with unit tests, fixing it is a matter of adding yet another unit test and adjusting the code without the drama and potential delays of last-minute manual regression testing.
- **Free up time for testing—**Unit tests are the simplest, fastest, and cheapest way to perform fundamental checking like verification of boundary values,

input validation, or invocation of the happy path. This allows testing performed manually to uncover things that are far more interesting than, let's say, off-by-one errors. Conversely, teams and organizations that lack unit tests will have to compensate by manual means, which translates into manual checking.

- **Specify behavior and document the code—**Ideally, a unit test is a description of some behavior of the tested code; that is, an example of how the code should work or implement a specific business rule. It's documentation. And what documentation tends to actually get read—a dryly written, autogenerated method description or working code?

Making People Responsible for Code

One strategy for trying to control change and regression is making a person or a team truly responsible for an area of the system. This strategy suffers from some obvious drawbacks, like people going on vacation, quitting their jobs, or just becoming bottlenecks. It will also encourage some individuals to keep information to themselves as a means of work protection.

If you really want to walk down this path, I suggest seeking inspiration in open-source projects, which have many contributors, who supply changes and patches, and few committers, who review these and commit them to the trunk. Because the contributors deliver complete change sets, the committers only have to review them, in contrast to implementing the changes themselves. This makes them less of a bottleneck.

Just a reminder, though: this way of working also comes with unit tests. They simplify the committer's job and prevent regressions.

What Is a Unit Test?

As we've seen in Chapter 3, "The Testing Vocabulary," nailing testing terminology down is hard. Defining the exact meaning of *unit test* is no different. Many details and technicalities can easily be debated. In this book, I've chosen to combine several sources[2] to provide a definition of unit test that, although probably not unchallenged in some circles, should be quite acceptable to the developer community.

A unit test is a piece of code that tests a unit of work—a method, class, or cluster of classes that implement a single logical operation, which is accessible through a public interface.[3] Unit tests have the following properties:

2. The following definition is inspired by Osherove (2009); Langr, Hunt, and Thomas (2015); and Feathers (2004).
3. This is one of these rules that has exceptions, but stop and think before testing encapsulated, nonpublic behavior.