Why unit tests and how to make them work for you

Here's a link to a video of me chatting about this topic

If you're not into videos, here's wordy version of it.

Software

The promise of software is that it can change. This is why it is called *soft* ware, it is malleable compared to hardware. A great engineering team should be an amazing asset to a company, writing systems that can evolve with a business to keep delivering value.

So why are we so bad at it? How many projects do you hear about that outright fail? Or become "legacy" and have to be entirely re-written (and the re-writes often fail too!)

How does a software system "fail" anyway? Can't it just be changed until it's correct? That's what we're promised!

A lot of people are choosing Go to build systems because it has made a number of choices which one hopes will make it more legacy-proof.

- Compared to my previous life of Scala where I described how it has enough rope to hang yourself, Go has only 25 keywords and *a lot* of systems can be built from the standard library and a few other small libraries. The hope is that with Go you can write code and come back to it in 6 months time and it'll still make sense.
- The tooling in respect to testing, benchmarking, linting & shipping is first class compared to most alternatives.
- The standard library is brilliant.
- Very fast compilation speed for tight feedback loops
- The Go backward compatibility promise. It looks like Go will get generics and other features in the future but the designers have promised that even Go code you wrote 5 years ago will still build. I literally spent weeks upgrading a project from Scala 2.8 to 2.10.

Even with all these great properties we can still make terrible systems, so we should look to the past and understand lessons in software engineering that apply no matter how shiny (or not) your

language is.

In 1974 a clever software engineer called Manny Lehman wrote Lehman's laws of software evolution.

> The laws describe a balance between forces driving new developments on one hand, and forces that slow down progress on the other hand.

These forces seem like important things to understand if we have any hope of not being in an endless cycle of shipping systems that turn into legacy and then get re-written over and over again.

The Law of Continuous Change

> Any software system used in the real-world must change or become less and less useful in the environment

It feels obvious that a system *has* to change or it becomes less useful but how often is this ignored?

Many teams are incentivised to deliver a project on a particular date and then moved on to the next project. If the software is "lucky" there is at least some kind of hand-off to another set of individuals to maintain it, but they didn't write it of course.

People often concern themselves with trying to pick a framework which will help them "deliver quickly" but not focusing on the longevity of the system in terms of how it needs to evolve.

Even if you're an incredible software engineer, you will still fall victim to not knowing the future needs of your system. As the business changes some of the brilliant code you wrote is now no longer relevant.

Lehman was on a roll in the 70s because he gave us another law to chew on.

The Law of Increasing Complexity

> As a system evolves, its complexity increases unless work is done to reduce it

What he's saying here is we can't have software teams as blind feature factories, piling more and more features on to software in the hope it will survive in the long run.

We **have** to keep managing the complexity of the system as the knowledge of our domain changes.

Refactoring

There are *many* facets of software engineering that keeps software malleable, such as:

- Developer empowerment
- Generally "good" code. Sensible separation of concerns, etc etc
- Communication skills
- Architecture
- Observability
- Deployability
- Automated tests
- Feedback loops

I am going to focus on refactoring. It's a phrase that gets thrown around a lot "we need to refactor this" - said to a developer on their first day of programming without a second thought.

Where does the phrase come from? How is refactoring just different from writing code?

I know that I and many others have *thought* we were doing refactoring but we were mistaken

Martin Fowler describes how people are getting it wrong

> However the term "refactoring" is often used when it's not appropriate. If somebody talks about a system being broken for a couple of days while they are refactoring, you can be pretty sure they are not refactoring.

So what is it?

Factorisation

When learning maths at school you probably learned about factorisation. Here's a very simple example

Calculate `1/2 + 1/4`

To do this you *factorise* the denominators, turning the expression into

`2/4 + 1/4` which you can then turn into `3/4`.

We can take some important lessons from this. When we *factorise the expression* we have **not changed the meaning of the expression**. Both of them equal `3/4` but we have made it easier for us to work with; by changing `1/2` to `2/4` it fits into our "domain" easier.

When you refactor your code, you are trying to find ways of making your code easier to understand and "fit" into your current understanding of what the system needs to do. Crucially **you should not be changing behaviour**.

An example in Go

Here is a function which greets `name` in a particular `language`

```
func Hello(name, language string) string {

  if language == "es" {
     return "Hola, " + name
  }

  if language == "fr" {
     return "Bonjour, " + name
  }

  // imagine dozens more languages

  return "Hello, " + name
}
```

Having dozens of `if` statements doesn't feel good and we have a duplication of concatenating a language specific greeting with `,` and the `name.` So I'll refactor the code.

```
func Hello(name, language string) string {
        return fmt.Sprintf(
                "%s, %s",
                greeting(language),
                name,
        )
}
```

```
8
9  var greetings = map[string]string {
10   "es": "Hola",
11   "fr": "Bonjour",
12   //etc..
13 }
14
15 func greeting(language string) string {
16   greeting, exists := greetings[language]
17
18   if exists {
19      return greeting
20   }
21
22   return "Hello"
23 }
```

The nature of this refactor isn't actually important, what's important is I haven't changed behaviour.

When refactoring you can do whatever you like, add interfaces, new types, functions, methods etc. The only rule is you don't change behaviour

When refactoring code you must not be changing behaviour

This is very important. If you are changing behaviour at the same time you are doing *two* things at once. As software engineers we learn to break systems up into different files/packages/functions/etc because we know trying to understand a big blob of stuff is hard.

We don't want to have to be thinking about lots of things at once because that's when we make mistakes. I've witnessed so many refactoring endeavours fail because the developers are biting off more than they can chew.

When I was doing factorisations in maths classes with pen and paper I would have to manually check that I hadn't changed the meaning of the expressions in my head. How do we know we aren't changing behaviour when refactoring when working with code, especially on a system that is non-trivial?

Those who choose not to write tests will typically be reliant on manual testing. For anything other than a small project this will be a tremendous time-sink and does not scale in the long run.

In order to safely refactor you need unit tests because they provide

- Confidence you can reshape code without worrying about changing behaviour

- Documentation for humans as to how the system should behave
- Much faster and more reliable feedback than manual testing

An example in Go

A unit test for our `Hello` function could look like this

```
1 func TestHello(t *testing.T) {
2   got := Hello(“Chris”, es)
3   want := "Hola, Chris"
4
5   if got != want {
6      t.Errorf("got %q want %q", got, want)
7   }
8 }
```

At the command line I can run `go test` and get immediate feedback as to whether my refactoring efforts have altered behaviour. In practice it's best to learn the magic button to run your tests within your editor/IDE.

You want to get in to a state where you are doing

- Small refactor
- Run tests
- Repeat

All within a very tight feedback loop so you don't go down rabbit holes and make mistakes.

Having a project where all your key behaviours are unit tested and give you feedback well under a second is a very empowering safety net to do bold refactoring when you need to. This helps us manage the incoming force of complexity that Lehman describes.

If unit tests are so great, why is there sometimes resistance to writing them?

On the one hand you have people (like me) saying that unit tests are important for the long term

health of your system because they ensure you can keep refactoring with confidence.

On the other you have people describing experiences of unit tests actually *hindering* refactoring.

Ask yourself, how often do you have to change your tests when refactoring? Over the years I have been on many projects with very good test coverage and yet the engineers are reluctant to refactor because of the perceived effort of changing tests.

This is the opposite of what we are promised!

Why is this happening?

Imagine you were asked to develop a square and we thought the best way to accomplish that would be stick two triangles together.

Two right-angled triangles to form a square

We write our unit tests around our square to make sure the sides are equal and then we write some tests around our triangles. We want to make sure our triangles render correctly so we assert that the angles sum up to 180 degrees, perhaps check we make 2 of them, etc etc. Test coverage is really important and writing these tests is pretty easy so why not?

A few weeks later The Law of Continuous Change strikes our system and a new developer makes some changes. She now believes it would be better if squares were formed with 2 rectangles instead of 2 triangles.

Two rectangles to form a square

She tries to do this refactor and gets mixed signals from a number of failing tests. Has she actually broken important behaviours here? She now has to dig through these triangle tests and try and understand what's going on.

It's not actually important that the square was formed out of triangles but **our tests have falsely elevated the importance of our implementation details**.

Favour testing behaviour rather than implementation detail

When I hear people complaining about unit tests it is often because the tests are at the wrong abstraction level. They're testing implementation details, overly spying on collaborators and mocking too much.

I believe it stems from a misunderstanding of what unit tests are and chasing vanity metrics (test coverage).

If I am saying just test behaviour, should we not just only write system/black-box tests? These kind of tests do have lots of value in terms of verifying key user journeys but they are typically expensive to write and slow to run. For that reason they're not too helpful for *refactoring* because the feedback loop is slow. In addition black box tests don't tend to help you very much with root causes compared to unit tests.

So what *is* the right abstraction level?

Writing effective unit tests is a design problem

Forgetting about tests for a moment, it is desirable to have within your system self-contained, decoupled "units" centered around key concepts in your domain.

I like to imagine these units as simple Lego bricks which have coherent APIs that I can combine with other bricks to make bigger systems. Underneath these APIs there could be dozens of things (types, functions et al) collaborating to make them work how they need to.

For instance if you were writing a bank in Go, you might have an "account" package. It will present an API that does not leak implementation detail and is easy to integrate with.

If you have these units that follow these properties you can write unit tests against their public APIs. *By definition* these tests can only be testing useful behaviour. Underneath these units I am free to refactor the implementation as much as I need to and the tests for the most part should not get in the way.

Are these unit tests?

YES. Unit tests are against "units" like I described. They were *never* about only being against a single class/function/whatever.

Bringing these concepts together

We've covered

- Refactoring
- Unit tests
- Unit design

What we can start to see is that these facets of software design reinforce each other.

Refactoring

- Gives us signals about our unit tests. If we have to do manual checks, we need more tests. If tests are wrongly failing then our tests are at the wrong abstraction level (or have no value and should be deleted).
- Helps us handle the complexities within and between our units.

Unit tests

- Give a safety net to refactor.
- Verify and document the behaviour of our units.

(Well designed) units

- Easy to write *meaningful* unit tests.
- Easy to refactor.

Is there a process to help us arrive at a point where we can constantly refactor our code to manage complexity and keep our systems malleable?

Why Test Driven Development (TDD)

Some people might take Lehman's quotes about how software has to change and overthink elaborate designs, wasting lots of time upfront trying to create the "perfect" extensible system and end up getting it wrong and going nowhere.

This is the bad old days of software where an analyst team would spend 6 months writing a requirements document and an architect team would spend another 6 months coming up with a design and a few years later the whole project fails.

I say bad old days but this still happens!

Agile teaches us that we need to work iteratively, starting small and evolving the software so that we get fast feedback on the design of our software and how it works with real users; TDD enforces this approach.

TDD addresses the laws that Lehman talks about and other lessons hard learned through history by encouraging a methodology of constantly refactoring and delivering iteratively.

Small steps

- Write a small test for a small amount of desired behaviour
- Check the test fails with a clear error (red)
- Write the minimal amount of code to make the test pass (green)
- Refactor
- Repeat

As you become proficient, this way of working will become natural and fast.

You'll come to expect this feedback loop to not take very long and feel uneasy if you're in a state where the system isn't "green" because it indicates you may be down a rabbit hole.

You'll always be driving small & useful functionality comfortably backed by the feedback from your tests.

Wrapping up