Mocks are Bad, Layers are Bad

It's time we admitted something: use of complex mocks is a code smell, and must be eliminated from a healthy code base.

Sometimes mocks are necessary, but I will argue that we need to structure our code to minimise their use, and to make them simple when they are needed. Let's start with a feeling.

Some days the tests just feel bad.

We've all written the unit tests I'm talking about - they are painful to write because you have to construct layers of mocks to satisfy your module's dependencies, and when you've written them you notice a strange sense of unease: you don't get that safe feeling tests usually give you.

It's almost like writing those tests was a waste of time.

Because let's face it, all you did was check that your code calls the methods you think it should call. If you ever change the details of the implementation, the tests will need to change to match.

At this moment, your colleague (you know, the one who "doesn't see the point of tests") leans over your shoulder and whispers: "you're just writing the code twice."

That sense of unease indicates a smell: we must expunge it.

How can we do things differently?

How do we get rid of mocks?

We want to avoid complex mocks, especially those that embody implementation details of the code under test.

Of course, we could just stop writing unit tests: then we can have that sense of unease all the time. Alternatively, we could adopt "Classical TDD" [Fowler], where tests cover several layers of functionality at once, rather than being restricted to a single unit. This makes our tests quite effective, since they cover the interactions between layers, but can make it much harder to debug when something fails.

What I want to argue, though is that we should do something different: avoid layers.

Reject layering

I hope to convince you that thinking of our software as a series of layers is damaging.

I'm going to start with an example. Imagine you are asked to implement a very simple markup rendering engine that accepts a subset of HTML (involving only text in paragraphs) and renders the result as an image. A layered approach might lead us to write an HTML parsing layer, consumed by a font rendering layer, consumed by a flow layout layer.

[If this design seems crazy to you, you're way ahead of me. Consider: is the complexity of our day-to-day work hiding decisions that are actually as crazy as this one?]

Let's write down some classes and interfaces. We'll make our code look a bit like Java, since the Java community can be quite keen on layers:

```
class ParsedHtml {
    List<Paragraph> paragraphs()
}

class HtmlParser {
    // Arguming naming in honour of [Hilton]
    HtmlParser( String data )
    ParsedHtml parse()
}

interface IFontCalculator {
    void setFontSize()
    void setFontFamily()
    List<Bitmap> render()
}

class ComicSansFontCalculator {
    ComicSansFontCalculator( HtmlParser htmlParser )
    void setFontSize()
    void setFontFamily()
    List<Bitmap> render()
}

class LayoutManager {
    LayoutManager( IFontCalculator fontCalculator,
                   int pageWidth )
    Bitmap layOut()
}
```

Each layer of this code works at its own layer of abstraction, and does a single well-focussed job. Each layer consumes objects of the layer below. We know that there will be many different fonts to choose from, so we ensure the details of font rendering are abstracted behind an interface.

When we come to consider tests for this code, we will find we need to introduce some more seams [Feathers], where we can insert mocks: now `ComicSansFontCalculator` will take an `IHtmlParser` in its constructor, so we can test it without needing to instantiate the real parser.

With this in place, we have a layered architecture similar to what many of us work with day-to-day. Notice that each time we want to test any layer, we need to write mocks for the layer below. Each layer of the system is dependent on the details of they layer below it. When a system becomes much more complex than our example, even if each layer is well-defined and kept to its own level of abstraction, the coupling between layers can become extremely complex and wide-ranging.

Coupling between parts of our code that ought to be separate is bad. When we define "layers" we have good motivations: we aim to simplify our code, making each layer deal with a single set of concepts (or "layer of abstraction"). However, often when we define layers we are really specifying a complex coupling between two separate areas of code: although the internals of a layer may be simple, the interfaces *between* layers are wide and complex, with many moving parts. Layers, like sheets of paper, are wide and flat. When sheets of paper are stacked on top of each other, adjacent sheets touch each other in lots of places.

Enterprise forwarding

We will start with the easy part, by arguing against a common layering technique: what I will call "enterprise forwarding". In our example, it might look like this:

```
class HtmlParser {
    HtmlParser( String data )
    ParsedHtml parse()
}

<layerDefinition>
    <object class="HtmlParser">
        <constructor args="String" argNames="data"/>
        <method name="parse" valueType="ParsedHtml"
                args="" argNames=""/>
    </object>
<layerDefinition>
```

And so on and so on for each object in the system. By repeating ourselves many times in different languages, we eventually achieve that pinnacle of layering: a multi-tiered system (see figure enterprise-forwarding).

But let's move on: everyone agrees that layer upon layer of identical method-declaration code alternating between Java and XML is horribly, horribly wrong.

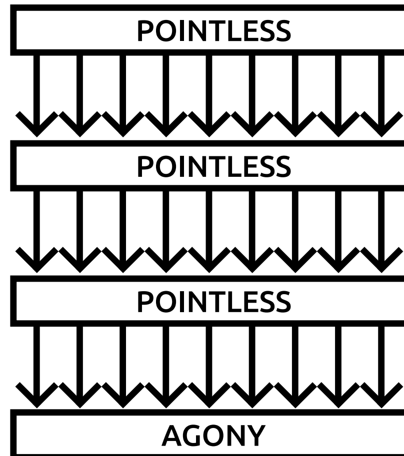

Figure 1: enterprise-forwarding

The only thing that could make things worse would be if the code did absolutely nothing at all until it was hooked up by some opaque, undebuggable blob of magic XML. But we would never do that.

Onion Skins

Enterprise forwarding aside, it is generally agreed that a particular piece of code should be written at a single layer of abstraction [Henney-how-to-write-a-method], and we see that certain areas of our code operate at the same level as each other, so we may find ourselves defining layers like onion skins, each building on the one beneath. If we test the outer layers together with the inner ones we may be Classical TDDers (figure classical-tdd) and if we write mocks to go under each layer we may be Mockist TDDers[Fowler] (figure mockist-tdd).

Whether classical or mockist in style, the onion skin approach leads us towards having wide and complex interfaces between parts ("layers") of our program. In our example, `IFontCalculator` classes depend on `HtmlParser`, and `LayoutManager` depends on `IFontCalculator`.

If we choose to isolate each layer during unit testing, we must write the kinds of complex mocks described in the introduction. Finding ways to keep our mocks simple enough that we can be confident they are not simply a second copy of

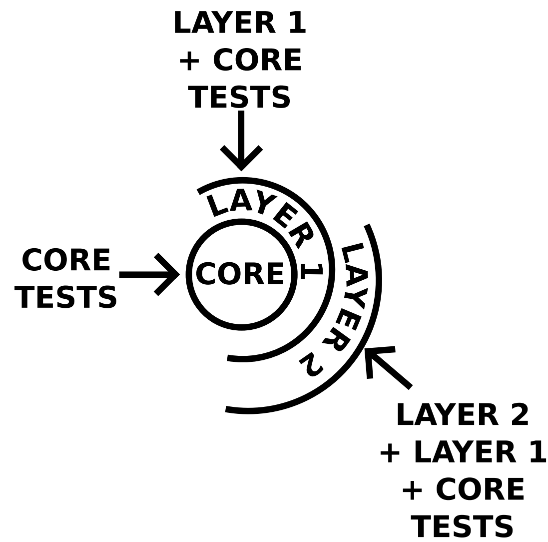

Figure 2: classical-tdd

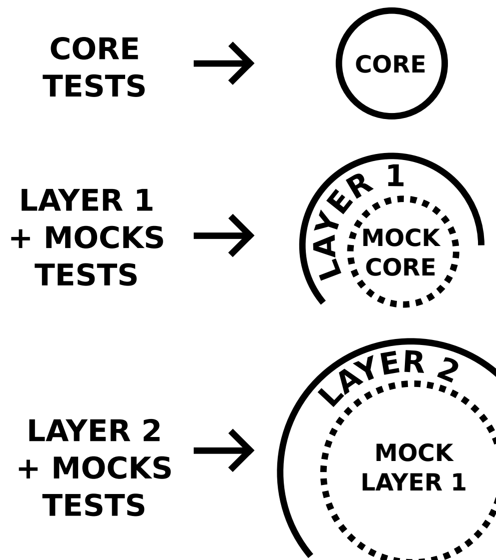

Figure 3: mockist-tdd

the code under test becomes increasingly difficult.

Instead of onion skins, we should strive to write small, genuinely self-contained units of code, that interact with other parts via simple, narrow interfaces. We will see some techniques and examples to help us with this later.

Some other things that are bad

As a side note, it's worth saying that we are touching on some explanations for why many people are beginning to view anything described as a "framework" with caution.

A framework is itself a layer (or series of layers) that surrounds your code, requiring you to plug your code into predefined slots. Often this means your code can't be used outside the framework (possibly even in tests), and can't work in a straightforward way, as it would if it were written as an independent module.

Similarly, a complex inheritance hierarchy is precisely an example of the kind of layering that can cause problems: the well-motivated desire to keep coherent units of code together has accidentally pushed us towards complex and subtle interactions between these units, so that while they look simple (because each source code file is small), they are actually highly coupled with many other units higher and lower in the inheritance chain.

Techniques for removing layering, or "some things that are good"

If we agree that layers should be avoided, we must find techniques and structures that allow us to escape them, without sacrificing testability or coherence of our code.

The **Selfish Object**[Henney-selfish-object] pattern is a powerful tool in our quest. Whereas layers lead us to wrap each implementation class in an interface that exactly reflects it, Selfish Object encourages us to build interfaces from the point of view of the classes using them, making them tightly focussed on the job the object is being used for in that context, rather than the full functionality of the underlying class (see figure selfish-object).

For example, our `LayoutManager` class may have no interest in changing font sizes, so it may be able to use a reduced `IFontCalculator` interface. While we're at it, maybe we could rename it to `BlockRenderer` since the `LayoutManager` has no interest in whether the `Bitmap`s being dealt with originated from letters or anything else:

```
interface BlockRenderer {
```

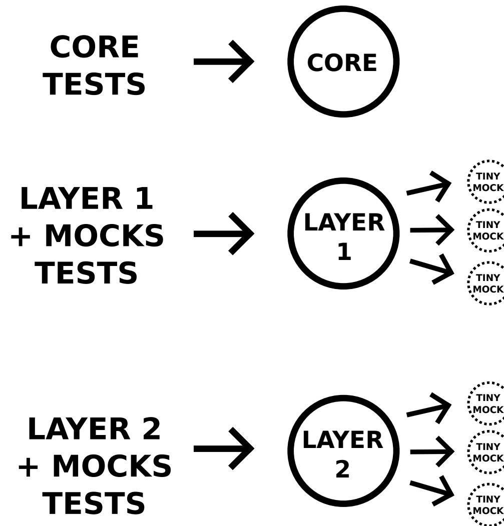

Figure 4: selfish-object

```
    List<Bitmap> render()
}
```

If other classes deal with font calculators, they may well have other needs. In those cases, separate interfaces could be provided, also implemented by e.g. `ComicSansFontCalculator`.

A powerful technique for avoiding complex mocks and layering is **Refactor to Functional** (see e.g. [Balaam]), which involves restructuring code so that the core logic exists in free functions with no state. These functions operate on simple, easily-constructed value-typed objects, meaning that unit tests no longer have complex set-up costs (see figure functional).

For example, our `HtmlParser` class may not require any internal state, meaning we can refactor it to look like this:

```
class HtmlParser {
    static ParsedHtml parse( String data )
}
```

Some Java developers may look at you oddly if you suggest a static method, having been burned in the past by global static state in their enterprise code. If

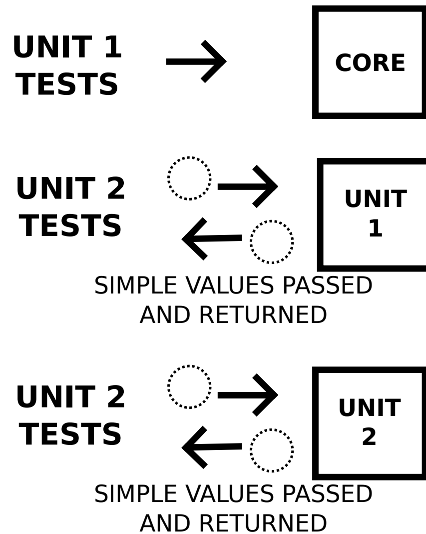

Figure 5: functional

this happens, it is important to emphasise the difference between mutable static state (which is another name for a global variable), and a stateless static method (which is another name for a “pure” function). The former is to be avoided at all costs, and the latter is one of the simplest, most predictable and most testable structures in programming.

Taking these ideas further, we can simplify the interactions between our classes, and reduce the need for complex mocks by ensuring we pass only simple types as parameters and return values of methods. If we are able to stick to types provided by the programming language we are working in (such as strings, numbers, lists, structs or tuples) then we completely eliminate dependencies between different areas of our program.

Obviously, this technique, which we will call **Talk in Fundamentals**, can be taken too far. Classes that are conceptually close, and within the same level of abstraction, should pass classes and interfaces between them that allow rich communication, and do not require us to deconstruct and reconstruct objects that could simply have been passed untouched. Particularly, when code at one level is being used to build a “language” that is then “spoken” by code at a higher level (see e.g. [SICP]), Talk In Fundamentals is certainly not appropriate, since the the building block classes actually are the fundamental types being used by the higher level.

However, many interactions between classes can be expressed using fundamental

types without any loss of expressiveness. For example, `LayoutManager` need not consume a `BlockRenderer` (or `IFontCalculator`), but simply a list of blocks to be rendered. If we also use Refactor to Functional as well, we might have something like this:

```
class LayoutManager {
    static Bitmap layOut( List<Bitmap> blocks )
}
```

Leading to the counter-intuitive conclusion that we can reduce coupling between two classes by *removing* an interface. `BlockRenderer` (or `IFontCalculator`) is not needed any more.

The further apart conceptually two communicating classes are, the more compelling is the case for using only simple types in their communication.

These techniques break our code into smaller independent units. We build libraries instead of frameworks, and functions instead of classes. We choose composition instead of inheritance, and we simplify the means of communication between distant pieces of code so that there is no dependency at all between them.

The Unix philosophy (of course)

If all of this sounds familiar, that's because it is mostly a re-expression of the **Unix Philosophy** pattern [Unix] (we'll follow it by the word "pattern" either to annoy the hackers, or to make it sound official to the enterprise programmers). In Unix, code is decomposed into small, stateless functions called "programs" which interact through very simple fundamental types called "streams".

Modern programming languages allow very flexible and type-safe streaming approaches using iterators, meaning that we can write our code to be agnostic not just of what other parts of the code are doing but also when they are doing it. We can consume our input as it arrives, and stream it to other consumers in the same way, potentially enabling different parts of the system to work concurrently (see figure streams).

If we apply everything we've learnt so far, we can refactor our example again, (leaving out class names since they are now noise):

```
static Iterator<Paragraph> parseHtml( InputStream text )

static Iterator<Bitmap> comicSans(
    Iterator<Paragraph> html )

static Bitmap layOut(
    Iterator<Bitmap> blocks, int pageWidth )
```

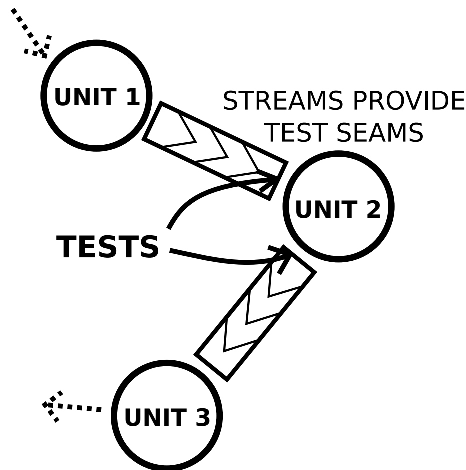

Figure 6: streams

Some aspects of what we've ended up with may be distasteful (for example, representing parsed HTML as `Iterable<Paragraph>` makes me feel a little uneasy), but there is no doubt that we have ended up with three wholly independent units of code that are easily tested and re-used, have no interdependencies, and may in principle execute in parallel.

Once we start taking the Unix philosophy seriously, the key consideration is how to make the units of code we write composable. To do this we need to find a shape for our primitives that can be composed via a common operation. In Unix the primitives are programs and composition is via text streams. We have some other examples of composable structures (see everything ever written in Lisp, and e.g. [Freeman]), but it seems we still have much to learn about how to achieve composability in mainstream programming languages.

Conclusion

We can avoid mocks by avoiding layers and building independent, composable units in our programs, as in the Unix Philosophy.

While the Unix philosophy is always appealing, it is sometimes hard to see how to apply it outside of Unix. The techniques discussed above may help to break our code into independent blocks, rather than interdependent layers.

So remember: if unit testing is becoming painful, don't mock - decompose.

References

[Balaam]: Andy Balaam "Avoid Mocks by Refactoring to Functional" (http://www.artificialworlds.net/blog/2014/04/11/avoid-mocks-by-refactoring-to-functional/)

[Feathers]: Michael Feathers "Working Effectively with Legacy Code" (http://www.informit.com/store/working-effectively-with-legacy-code-9780131177055)

[Fowler]: Martin Fowler "Mocks Aren't Stubs" (http://martinfowler.com/articles/mocksArentStubs.html)

[Freeman]: Steve Freeman and Nat Pryce "Building SOLID Foundations" (http://www.infoq.com/presentations/design-principles-code-structures)

[Henney-how-to-write-a-method]: Kevlin Henney "How to Write a Method" (https://vimeo.com/74316116)

[Henney-selfish-object]: Kevlin Henney, "The Selfish Object" (http://accu.org/content/conf2008/Henney-The%20Selfish%20Object.pdf)

[Hilton]: Peter Hilton "How to name things" (http://hilton.org.uk/presentations/naming)

[SICP]: Abelson, Sussman and Sussman "Structure and Interpretation of Computer Programs" (http://mitpress.mit.edu/sicp/)

[Unix]: Wikipedia "The Unix Philosophy" (https://en.wikipedia.org/wiki/Unix_philosophy)