Menu +

SANDI METZ

The Wrong Abstraction

I originally wrote the following for my Chainline Newsletter, but I continue to get tweets about this idea, so I'm re-publishing the article here on my blog. This version has been lightly edited.

I've been thinking about the consequences of the "wrong abstraction." My RailsConf 2014 "all the little things" talk included a section where I asserted:

> *duplication is far cheaper than the wrong abstraction*

And in the summary, I went on to advise:

> *prefer duplication over the wrong abstraction*

This small section of a much bigger talk invoked a surprisingly strong reaction. A few folks suggested that I had lost my mind, but many more expressed sentiments along the lines of:

> *This, a million times this! "@BonzoESC: "Duplication is far cheaper than the wrong abstraction" @sandimetz @rbonales pic.twitter.com/3qMIowaqWb"*
>
> *— 41 shades of blue (@pims) March 7, 2014*

The strength of the reaction made me realize just how widespread and intractable the "wrong abstraction" problem is. I started asking questions and came to see the following pattern:

1. Programmer A sees duplication.
2. Programmer A extracts duplication and gives it a name.

   This creates a new abstraction. It could be a new method, or perhaps even a new class.
3. Programmer A replaces the duplication with the new abstraction.

   Ah, the code is perfect. Programmer A trots happily away.
4. Time passes.
5. A new requirement appears for which the current abstraction is *almost* perfect.
6. Programmer B gets tasked to implement this requirement.

   Programmer B feels honor-bound to retain the existing abstraction, but since isn't exactly the same for every case, they alter the code to take a parameter, and then add logic to conditionally do the right thing based on the value of that parameter.

   What was once a universal abstraction now behaves differently for different cases.
7. Another new requirement arrives.
   Programmer X.

Another additional parameter.
Another new conditional.
Loop until code becomes incomprehensible.

8. You appear in the story about here, and your life takes a dramatic turn for the worse.

Existing code exerts a powerful influence. Its very presence argues that it is both correct and necessary. We know that code represents effort expended, and we are very motivated to preserve the value of this effort. And, unfortunately, the sad truth is that the more complicated and incomprehensible the code, i.e. the deeper the investment in creating it, the more we feel pressure to retain it (the "sunk cost fallacy"). It's as if our unconscious tell us "Goodness, that's so confusing, it must have taken *ages* to get right. Surely it's really, really important. It would be a sin to let all that effort go to waste."

When you appear in this story in step 8 above, this pressure may compel you to proceed forward, that is, to implement the new requirement by changing the existing code. Attempting to do so, however, is brutal. The code no longer represents a single, common abstraction, but has instead become a condition-laden procedure which interleaves a number of vaguely associated ideas. It is hard to understand and easy to break.

If you find yourself in this situation, resist being driven by sunk costs. When dealing with the wrong abstraction, *the fastest way forward is back*. Do the following:

1. Re-introduce duplication by inlining the abstracted code back into every caller.
2. Within each caller, use the parameters being passed to determine the subset of the inlined code that this specific caller executes.
3. Delete the bits that aren't needed for this particular caller.

This removes both the abstraction *and* the conditionals, and reduces each caller to only the code it needs. When you rewind decisions in this way, it's common to find that although each caller ostensibly invoked a shared abstraction, the code they were running was fairly unique. Once you completely remove the old abstraction you can start anew, re-isolating duplication and re-extracting abstractions.

I've seen problems where folks were trying valiantly to move forward with the wrong abstraction, but having very little success. Adding new features was incredibly hard, and each success further complicated the code, which made

adding the next feature even harder. When they altered their point of view from "I must preserve our investment in this code" to "This code made sense for a while, but perhaps we've learned all we can from it," and gave themselves permission to re-think their abstractions in light of current requirements, everything got easier. Once they inlined the code, the path forward became obvious, and adding new features become faster and easier.

The moral of this story? Don't get trapped by the sunk cost fallacy. If you find yourself passing parameters and adding conditional paths through shared code, the abstraction is incorrect. It may have been right to begin with, but that day has passed. Once an abstraction is proved wrong the best strategy is to re-introduce duplication and let it show you what's right. Although it occasionally makes sense to accumulate a few conditionals to gain insight into what's going on, you'll suffer less pain if you abandon the wrong abstraction sooner rather than later.

When the abstraction is wrong, the fastest way forward is back. This is not retreat, it's advance in a better direction. Do it. You'll improve your own life, and the lives of all who follow.

News: 99 Bottles of OOP in JS, PHP, and Ruby!

The 2nd Edition of 99 Bottles of OOP has been released!

The 2nd Edition contains 3 new chapters and is about 50% longer than the 1st. Also, because 99 Bottles of OOP is about object-oriented design in general rather than any specific language, this time around we created separate books that are technically identical, but use different programming languages for the examples.

99 Bottles of OOP is currently available in Ruby, JavaScript, and PHP versions, and beer and milk beverages. It's delivered in epub, kepub, mobi and pdf formats. This results in six different books and (3x2x4) 24 possible downloads; all unique, yet still the same. One purchase gives you rights to download any or all.

Posted on January 20, 2016 by Sandi Metz.

Newer / Older

ALSO ON **SANDI METZ**

The Wrong Abstraction — Sandi Metz

5 months ago

I've been thinking about the consequences of the "wrong abstraction." My ...

The Wrong Abstraction — Sandi Metz

2 months ago

I've been thinking about the consequences of the "wrong abstraction." My ...

The Wrong Abstraction — Sandi Metz

4 months ago

I've been thinking about the consequences of the "wrong abstraction." My ...

Th — S

2 m

I've cor abs

Rohan Jayasekera

5 years ago

Indeed. BTW for maximum clarity I'd suggest that step 1 explicitly say "Delete the abstraction". It's implied, but it may be easy to forget to do it – in which case a new programmer may see the abstraction and think they're supposed to use it.

Barry Davies

6 years ago

Seems like a key part of this is whether it truly was duplication to begin with. A question that keeps coming up for me when I run into this situation is "do I really want these two places to move in lock-step, or do they just cosmetically look duplicated?"

Robert Moskal

4 years ago

We can open this up again. I remember reading this two years ago, and having then been working with a legacy code base, agreeing totally.

I do question the refactoring approach. Instead of reintroducing the duplicated code (looking at the parameters and conditionals to decide which parts), I would try to refactor the abstraction as a composition. Under condition a do:

```
a(g(x))
```

under condition b, do:

```
b(g(x))
```

A bad abstraction as described in the above can definitely be resolved via a composition. And as one finds other use cases, it's easy to extend the model:

```
c(b(g(x)))
```

In general, I say if a problem can be expressed as a composition, it should be. So much of the code I see every day hides one inside of it. Devs should be taught to look for them.

7 0 Reply • Share ›

Juan Labrada

5 years ago

I would notice that the wrong abstraction began not by programmer A who first removed duplication but with programmer B or C who misunderstood to be "honor-bound" and instead of continuing refactoring and improving the code quality just added the conditional. This process also shows off a lack of code review practices that would have identified this problem. So good by programmer A.

9 1 Reply • Share ›

Ben Randles-Dunkley

6 years ago

I agree with this entirely. If some code has become (or was designed to be) over-complex, time-consuming to amend, overly confusing or obfuscated, it's probably time to look into replacing it with something better. The investment in time will be rewarded many times over in time saved later and employee morale.

It doesn't have to be just for abstractions of duplicated code. Maybe you've got a data structure spread over three classes, with broken encapsulation. Maybe a complex and fragile persistence system could be replaced with something simpler and more robust. We all come across sections of code that are over-engineered, hurriedly slapped together for a deadline or built for a set of conditions that may not be relevant any more. We shouldn't be afraid to re-think, re-engineer, refactor. (There may be muttering as you do so. This is normal.)

When it comes to duplication, sometimes it's a good idea to leave it in in the first place. (Gasp!) Except in trivial cases, a single duplication doesn't give much information about the shape of a general solution. Once you're doing the same thing three times you might have a bit of a better idea, but always be prepared to back out and change the abstraction when more requirements come in...

3 0 Reply • Share ›

B

Brian

5 years ago

I don't see this as a failure of abstraction. It's caused by some specific other failures.

1. Failure of people to understand data modeling, and the difference between interface and implementation. "A new requirement appears for which the current abstraction is almost perfect." Well, is it or isn't it? Just because the code is the same doesn't mean it should use the same abstraction. The data model is wrong, not the abstraction.

2. Failure of people to think of the entire system, when making a change. The rookie mistake is to think that adding one more conditional is always harmless, but clearly at some point you've got 3 tons of straw on that poor camel's back. Adding just one more case may have been the correct fix last week, but not this week.

3. Failure of programming languages and frameworks to support the necessary abstractions. In a language like Java, for example, it's common for the easiest way to be "just add another conditional". In a language that supports multiple dispatch, though, you can easily add more situations that are similar to existing cases, without writing any conditionals at all, and it stays easy to maintain because you're not writing a dispatcher by hand along with your business logic.

6 2 Reply • Share ›

S

SherekhudaHazratali.com

5 years ago

After all the hemming and hawing about this, I'd really like to see a real case study on it. It's never been an issue for me. I've had functions where I added parameters later on for new use cases to fit the function to them, yes, but I never felt like I was suffering for it. Maybe DRY isn't so bad after all.

2 0 Reply • Share ›

Paul Slusarz

6 years ago

I have also seen this pattern, and used the same general inline technique to get out of the mess. One additional method I use is to examine runtime behavior to trim down unneeded branches. By the time I get to see the code the combinations of inputs are in the 100s, and side effects in dozens. Fortunately, runtime behavior is a lot simpler. I think that's because the code serves business processes which would have long collapsed on their own weight if they were too complicated.

2 0 Reply • Share ›

D

Dave Smith

6 years ago

It seems to me that if parameters are added to abstractions to enable optional features, yes it is the wrong abstraction, but more specifically, the abstraction lacks cohesion. Good abstractions do only one thing. There is nothing wrong with using a bunch of small cohesive abstractions in your implementation to reduce code duplication.

For example the set of functional list abstractions like foldl, map, and filter all generalize the operations over a list. The foldl function traverses a list from left to right and applies an operation , accumulating an output. Sometimes this doesn't work for my case, and I need to traverse the list from right to left. The library developers didn't add a parameter to foldl; they created a new abstraction called foldr.

In the OO world, adhering to the SOLID principals should keep your abstractions cohesive

and coherent. A good library will almost always have a public interface consisting of a layer of small cohesive interfaces.

I think may people get frustrated with abstraction because they see so many "bad" abstractions, and so assume getting the right abstraction is hard.

3 1 Reply • Share ›

B

BillyHam

5 years ago

If you play out the duplication scenario as you did the DRY scenario, you will see the problem with repetition. Programmer 2 alters the duplicated code, but also something changes that is common to dupe1 and dupe2, but they only change dupe2. Repeat. Now your code, instead of being complicated, is riddled with bugs because people keep forgetting to change dupe1, dupe2, dupe3, etc.

4 3 Reply • Share ›

Mike Post

5 years ago

I believe you're talking about the challenge of legacy code, rather than the challenge of abstraction.

Abstraction is the wrong target, it is the effect rather than the cause of the problem here. People won't understand this though, and the internet will catch fire with an "all abstraction is bad" movement. I'm retiring.

2 1 Reply • Share ›

Rasheed Abdul-Aziz

5 years ago

This is not a newly realized problem, but DRY has made it a monster to combat, and this article is the best armor I've found. In my eyes, this is your magnum opus Sandi.

2 1 Reply • Share ›

Chris Tanner

5 years ago

You've identified the problem but come up with a bad solution. If the abstracted class/function can't be modified without side effects, it's doing too much and should be broken up into smaller functions which can be called and extended as needed.

1 0 Reply • Share ›

Aleksandr Strutynskyy

6 years ago

ah classes, you got to love'em... code duplication by necessity, if only there was a way to get away from them, say some way to compose ideal object from small functional pieces... wouldn't that be great? yeah. Anyways back to topic, I totally agree with the "prefer

duplication over the wrong abstraction".

1 0 Reply • Share ›

Jeroen De Dauw

5 years ago

This is a nice write-up. I wrote about the same topic with more abstract analysis in my post The Fallacy of DRY:
https://www.entropywins.wtf...

2 2 Reply • Share ›

doublejosh

3 years ago

Imagine an IDE plugin called UNABSTRACT that created the duplicated classes!

0 0 Reply • Share ›

Bruno Guimarães Carneiro

3 years ago

The rule is: THERE ARE NOT CONDITIONS IN ABSTRACTIONS.

If you find yourself adding a condition in one, refactor it.

0 0 Reply • Share ›

Subscribe

Sign up to receive news and updates.

First Name

Last Name

Email Address

SIGN UP

Home
Work With Me
Products
Speaking
Courses
Blog
About
Contact