Why people misuse inheritance

18 Mar, 2023

Jimmy Koppel wrote a thread about inheritance:

> *"The Flaws of Inheritance" by @CodeAesthetic1 is beautiful, as always.*
>
> *Problem though, is that none of the things discussed in the video have anything to do with inheritance*
>
> *Time for a 🧵 on the most mind-bending construction in mainstream programming languages https://t.co/oSpZJgyBTN*
>
> *— Jimmy Koppel (@jimmykoppel) March 8, 2023*

In the thread, he recalls the adage "prefer composition over inheritance". This is a well-known principle of good OOP code, and yet inheritance is commonly used where composition would serve better; the question that comes to my mind is, "Why?" I think I have at least a partial answer, but let me meander a bit before getting to it.

The thread gives an example use case of a map that counts explicit insertions, which is the example I'll use here. If you inherit and override the `put()` method, the behavior you get may be wrong, or may be right in one version and wrong in the next. On the other hand, you didn't have to write a lot of code.

(One post, an oldie but a goodie, that significantly influenced my thinking on this matter, suggests that you should never override a method that was not designed to be overridden, and talks about how building this into the language slightly improves ergonomics when you *do* override.)

Suppose I did the "right" thing and used composition (in this case, also delegation, and also the decorator pattern, for people who think in GoF design patterns). I would have to implement the map interface, calling down to a map implementation that stores the actual data. To implement the `Map` interface in Java I would need to implement 25 methods! Most of them would be boilerplate, just passing the arguments to the equivalent method on the delegate. Other languages are not better (the Haskell `Data.Map` module has more than 100 functions).

This leads me to my answer: it often feels like less work to implement and less work to maintain an inheritance-based approach than a composition/delegation-based one. Of course, the inheritance-based one is often either outright wrong or a subtle bug waiting to happen, but that doesn't make it feel any better to write over 20 methods of boilerplate.

So what's the alternative? I wish I could say "metaprogramming", but that's been around since the '60s and still hasn't caught on as a common tool in most programmers' toolkits, so it can't be the real answer. But it's *so* compelling: the pattern that you want is common and simple to describe, so why can't you just write a program to do it? The next most plausible idea is "libraries/frameworks that do the metaprogramming for you"; these certainly exist, but to my knowledge they don't reach critical mass for widespread adoption. A partial solution is for libraries to offer a generic delegates for interfaces, which will enable parsimonious and correct use of inheritance, but this puts additional burden on every library implementation.

The actual answer that I want, as unlikely as it is to come to be, is "language support" (or at least standardization on a specific library that solves the problem using the language's metaprogramming facilities, which isn't that different). I expect that in a language that has such blessed, boilerplate-free delegation tools, inheritance will be less overused.

In the mean time, what should you do?

- Remember that subtly wrong behavior and bugs-waiting-to-happen are also maintenance burdens, and they are worse than boilerplate.
- Learn your language's metaprogramming facilities and see if there is enough value in using them to reduce the boilerplate.
- If you maintain a large codebase, look for and standardize use of a library that solves the problem in your language (or possibly write one).
- If you're developing a language or standard library, consider adding a feature that reduces the boilerplate in common delegation scenarios.
- If you're doing programming language research, this is probably not news to you; you might enjoy reading this tangentially related post about a related problem in Haskell or this OCaml paper that inspired it.