

# Jay Fields' Thoughts

experiences in software development

---

Thursday, July 24, 2008

---

## Ruby: Underuse of Modules

---

When I began seriously using Ruby I noticed two things that I didn't like about the language.

- There's no method to get the metaclass.
- Some people [create modules as a hint](#) that they reopened the class.

It's a few years later and I've noticed a few interesting things.

- I can't remember the last time I actually wanted a method to get the metaclass.
- If you use a module to add behavior your behavior becomes part of the ancestor tree, which is significantly more helpful than putting your behavior directly on the class.

### No metaclass method necessary

In April 2005, Why gave us [Seeing Metaclasses Clearly](#). I'm not sure the article actually helped me see metaclasses clearly, but I know I pasted that first block of code into several of my first Ruby projects. I [used metaclasses in every way possible](#), several of which were probably inappropriate, and saw exactly what was possible, desirable, and painful.

I thought I had a good understanding of the proper uses for metaclasses, and then Ali Aghareza brought me a fresh point of view: defining methods on a metaclass is just mean.

We were on the phone talking about who knows what, and he brought up a blog entry I'd written where I [dynamically defined delegation methods on the metaclass](#) based on a constructor argument. He pointed out that doing so limited your ability to change the method behavior in the future. I created some simple examples and found out he was right, which lead to my blog entry on why you should [Extend modules instead of defining methods on a metaclass](#).

Ever since that conversation and the subsequent blog entry, I've been using modules instead of accessing the metaclass directly. "Just in case someone wants to redefine behavior" isn't really a good enough reason for me if the level of effort increases, but in this case I found the code to be easier to follow when I used modules. In programming, there are few win-win situations, but Ali definitely showed me one on this occasion.

If you interact with a metaclass directly, do a quick spike where you introduce a module instead. I think you'll be happy with the resulting code.

### Include modules instead of reopening classes

In January of 2007 I wrote a blog entry titled [Class Reopening Hints](#). I didn't write it because I thought it was very valuable, I wrote it so developers afraid of open classes could get some sleep at night. Those guys think we are going to bring the end of the world with our crazy open classes, and I wanted to let them know we'd at least thought about the situation.

I'm really not kidding, I thought the entry was a puff piece, but it made some people feel better about Ruby so I put it together. I never followed the *Use modules instead of adding behavior directly* advice though, and I don't think many other Rubyists did either. It was extra effort, and I didn't see the benefit. In over two and an half years working with Ruby I've never once had a problem finding where behavior was defined. With that kind of experience, I couldn't justify the extra effort of defining a module -- until the day I wanted to change the [behavior of Object#expects \(defined by Mocha\)](#). I was able to work around the fact that Mocha defines the `expects` method directly on `Object`, but the solution was anything but pretty.

It turns out, using modules instead of adding behavior directly to a reopened class has one large benefit: I can easily define new behavior on a class by including a new module. If you only need new behavior, then defining a new method on the class would be fine. But, if you want to preserve the original behavior, having it as an ancestor is much better.

Take the following example. This example assumes that a method `hello` has been defined on `Object`. Your task is to change the `hello` method to include the original behavior and add a name.

```
# original hello definition
class Object
  def hello
    "hello"
  end
end

# your version with additional behavior
class Object
  alias old_hello hello
  def hello(name)
    "#{old_hello} #{name}"
  end
end

hello("Ali") # => "hello Ali"
```

That code isn't terrible. In fact, there are a [few different ways to redefine methods](#) and access the original behavior, but none of them look as nice as the following example.

```
# original hello definition
class Object
  module Hello
    def hello
      "hello"
    end
  end
end
```

```

end
include Hello
end

# your version with additional behavior
class Object
  module HelloName
    def hello(name)
      "#{super()} #{name}"
    end
  end
  include HelloName
end

hello("Ali") # => "hello Ali"

```

When you have an ancestor, the behavior is only a `super` call away.

*note:* Yes, I've also reopened the class to include the module, but when I talk about reopening the class I'm talking about defining the behavior directly on the reopened class. I could have also included the module by using `Object.send :include, HelloName`. Do whichever you like, it's not pertinent to this discussion.


### Prefer Modules to metaclasses and reopened classes

The previous example illustrates why you should prefer modules, it gives simple access to your method behavior to anyone who wishes to alter but reuse the original behavior. This fact applies to both classes that include modules and instances that extend modules.

So why didn't Matz give us first class access to the metaclass? Who cares. He probably knew extending modules was a better solution, but even if he didn't -- it is. He didn't give you a method to access the metaclass, and whether he knew it or not, you don't need it.

Labels: [extend](#), [include](#), [metaclass](#), [module](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

# posted by Jay Fields : 1:57 AM   
[ general focus: [Ruby](#) - [Ruby on Rails](#) - [Ruby Rake](#) ]

## Comments:

That post summarizes my thoughts too! Great read.

# posted by  Jacek Becela : 5:47 AM

Can you show another example, then, of how one might implement the "magic" of Dwemthy's Array (<http://poignantguide.net/dwemthy/>) just using modules? I can never remember how to do this sort of thing, and if modules can make it conceptually simpler it would be most useful. I've attempted this, without success.