Code reuse in Ruby – It's complicated

Gregory Brown (practicingruby.com)

February 12, 2013

Abstract

Ruby provides at least seven common ways of reusing code, all of them with their own strengths and weaknesses. However, the main thing that separates these various techniques is whether they are a form of *implementation sharing* or a form of *behavior sharing.* This article explains what distinguishes those two categories, the kinds of complexities that can arise from each of them, and some practical recommendations that you can apply to reusing code within your own projects.

1. Introduction

As a deeply object-oriented programming language, Ruby permits code reuse in more ways than most languages. When put in the right hands, Ruby's flexibility is extremely powerful, allowing us to model our systems any way we want. But with great power comes great responsibility.

If not used carefully, Ruby's code reuse mechanisms can quickly erode most (if not all) of the benefits that object-oriented design is meant to offer us. The larger our systems get, the easier it is for things to spiral out of control. But the truth is that most codebases don't start out as an unmaintainable mess, nor do they suddenly turn bad overnight. Instead, they erode gradually – one poor decision at a time.

In this article, we'll discuss the various pitfalls of Ruby's code reuse methods. My hope is that by studying these issues, you'll be more aware of the risks involved in certain modeling decisions, and that will help you better understand the compromises you must make while designing your projects.

2. Common methods of code reuse

Ruby's code sharing methods can be divided into two groups: those that provide direct access to the internals of the shared component (*implementation sharing*), and those that do not (*behavior sharing*). While each approach has its own set of costs and benefits, a lot can be said about the complexity of a technique by knowing which reuse category it belongs to.

2.1 Implementation sharing techniques

The following techniques reuse code in ways that provide direct access to internals:

- **inheriting** from a superclass
- **including** a module into a class
- **extending** an individual object using a module
- **patching** a class or individual object directly
- **evaluating** code in the context of a class or individual object

2.2 Behavior sharing techniques

The following techniques rely on message passing between distinct objects for code sharing, limiting direct access to internals:

- **decorating** an object using a dynamic proxy
- **composing** objects using simple aggregation

2.3 Reference examples

Our goal is only to discuss the complexities of implementation sharing and behavior sharing in general, so you don't need to be familiar with all seven methods of code reuse listed above in order to understand the rest of this article.

However, if you want some additional clarification about what each of these terms mean, you can check out this set of code-reuse examples.

3. Complexities of implementation sharing

An entire book can be written about the complexities involved in sharing functionality without proper encapsulation between components. However, since we don't have room for that level of detail in this article, I've attempted to group the common issues together into three broad areas of concern to look out for: shared instance variables, shared method definitions, and combinatorial effects.

3.1 Shared instance variables

Each object has a single set of instance variables, even if it has a very complex ancestry chain. For example, the following code references an instance variable that was defined by its superclass:

```
require "ostruct"

class PrettyStruct < OpenStruct
  def inspect
    @table.map { |k,v| "#{k} = #{v.inspect}" }.join("\n")
  end
end

struct = PrettyStruct.new(:a => 3, :b => 4, :c => 5)
p struct

# a = 3
# b = 4
# c = 5
```

When two or more shared components reference the same instance variable, it may be intentional or unintentional. It goes without saying that unintentional variable name collisions can lead to defects that are hard to debug, but intentional shared access (such as in the snippet above) has more subtle issues to consider.

Whenever we directly access a variable rather than using a public accessor, we may be skipping validations, transformations, caching features, or concurrency-related features that are meant to keep the underlying data consistent and synchronized. Is a simple read-only reference such as the one we've done here really that risky? The truth is, there's no way to know without reading the `OpenStruct` source code.

Unfortunately, the only way to know for sure what instance variables will be defined, accessed, and modified at runtime for *any* Ruby object is to read the source of every single class and module that is in its ancestry chain, both at the individual object and class definition level. Because new variables can spring into existence any time a method is called, this kind of static analyis is not practical for most non-trivial programs.

At the extreme end of the spectrum, you have objects that inherit from `ActiveRecord::Base`; they exist at the tail end of an ancestry chain that provides several instance variables and hundreds of methods through dozens of modules, and that's assuming that you haven't installed any third-party plugins. If you aren't convinced by the trivial example I've shown in this article, spend some time with the Rails source code and you'll surely get the point.

3.2 Shared method definitions

Even when reusing an ancestor's public API, it can be challenging to avoid strange inconsistencies. Bob Martin provided a classic example of this problem in an article on the Liskov Substitution Principle. Consider a `Rectangle` class with a `Square` subclass, as shown below:

```ruby
class Rectangle
  def initialize(width, height)
    self.width   = width
    self.height  = height
  end

  attr_accessor :width, :height

  def area
    width * height
  end
end

class Square < Rectangle
  def initialize(size)
    super(size, size)
  end
end
```

On the surface, this implementation looks simple, and seems to work as expected:

```ruby
square = Square.new(5)

p square.area                    #=> 25
p [square.width, square.height] #=> [5, 5]
```

But there is also the potential for bad behavior here, because the `Square` class also inherits `Rectangle#width=` and `Rectangle#height=`, which can lead to inconsistent data in the `Square` object:

```ruby
square.width = 10

p [square.width, square.height] #=> [10, 5] -- not a square!
```

One way to resolve this issue would be to override `Rectangle#width=` and `Rectangle#height=` so that the two values are synchronized:

```
class Square < Rectangle
  def initialize(size)
    super(size, size)
  end

  def width=(size)
    @width   = size
    @height  = size
  end

  def height=(size)
    @width   = size
    @height  = size
  end
end

square = Square.new(5)

square.width = 10
p [square.width, square.height] #=> [10, 10]
p square.area                   #=> 100
```

This change enables the kind of behavior you might expect from a `Square`, and if you are simply reusing code to keep things DRY, that might be good enough. However, there may still be some subtle issues in code which assumes that a rectangle's height can vary independently of its width, such as in this test code:

```
def test_area
  rect.width  = 5
  rect.height = 10

  assert 50, rect.area
end
```

Arguably, this test is written poorly if it is meant to be used as a shared example for all descendents of the `Rectangle` object. The problem is that at a first glance, the flaw is not at all obvious. And that essentially is the core challenge in inheritance-based modeling: ancestors must guess about the kinds of ways that they will be extended, and descendents need to guess about whether their extensions will break upstream features. With some practice and careful design thought this is possible, but it certainly is not *easy* to reason about.

3.3 Combinatorial effects

Shared method definitions and shared instance variables are at the root of what makes implementation sharing complex, but that complexity is compounded by the fact that ancestry chains can grow arbitrarily long. Module mixins in particular tend to cause this problem, because they are typically viewed by Ruby programmers as a tool for implementing orthogonal *plugins*, but are functionally more similar to *multiple inheritance*.

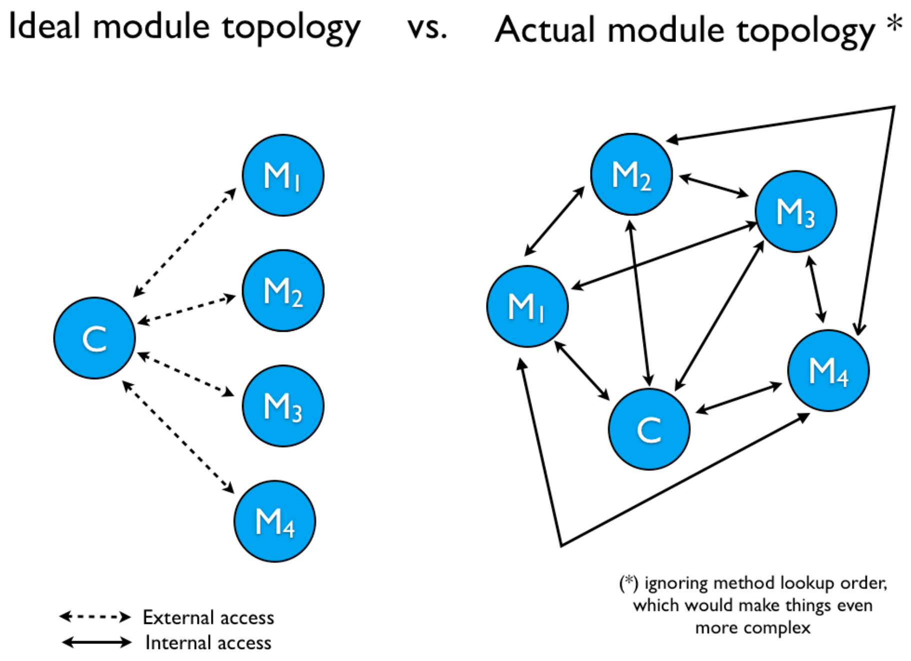
Consider an arbitrary class C, with four modules mixed into it: M_1, M_2, M_3, and M_4. Typically, each of these modules will provide some features to C and perhaps require that C implement a few methods to enable those features. Since each of these modules is meant to be used standalone, they aren't directly aware of one another, nor do they depend on each other's features.

In this scenario, each module might need to make calls to C's public API and vice-versa, but there would be no need for the modules to be able to call each other's public methods directly. Furthermore, in an ideal situation, C and its mixed-in modules would communicate entirely via public method calls, allowing each to have their own private methods and internal state. If these constraints were enforced at the language level, it'd be possible to model mixins as a simple, horizontal lookup path that would be trivial to reason about.

From our perspective as Ruby users, the scenario described above might cover 90% of what we use modules for on a day to day basis. But because modules are actually a much more powerful and generalized construct, we cannot expect that simplistic mental model to be a good fit for how they actually work. In reality, every module we mix into a class has direct access to the variables and methods defined by every other mixed in module in that class, resulting in a combinatorial explosion of possible interactions.

The following graph attempts to illustrate the difference between our typical way of thinking about (and using) modules, and how they actually work:

(see next page)



What you see above is just one small slice of the total method lookup path, but it illustrates the general problem that repeats itself along the whole chain: every ancestor can access the internals of every other, and the number of possibilities expands greatly with each new component added to the chain.

In practice, when concerns really are orthogonal, most of the combinatorial effects between components can safely be ignored as long as you apply some informal reasoning. But as an object gets larger, it becomes more likely that some pair of ancestors will accidentally develop conflicting definitions of what a method or variable is meant to be used for, and those issues can be very difficult to debug. Furthermore, each new ancestor also makes it harder to add new functionality to an object without accidentally breaking existing features.

This issue can be mitigated by the use of mixins at the individual object level, which can allow different bits of reusable functionality to be used in isolation of one another by only mixing in one module at at a time. However, this technique only works around the issue – it does not eliminate it entirely.

4. Complexities of behavior sharing

Behavior sharing techniques do not suffer from any of the issues we've discussed so far, and that alone makes them worth considering as a better default approach. However, they do have their own share of problems, so you need to be aware of what the tradeoffs are when deciding how to model your systems.

4.1 Indirect access

When access to an object's internals is truly necessary, it isn't practical to use composition based techniques. For example, consider the following mixin-based code which implements a memoization routine for caching method return values:

```ruby
module Cached
  def cache(*method_names)
    method_names.each do |m|
      original = instance_method(m)
      results  = {}

      define_method(m) do |*a|
        results[a] ||= original.bind(self).call(*a)
      end
    end
  end
end

## EXAMPLE USAGE:

class Numbers
  extend Cached

  def fib(n)
    raise ArgumentError if n < 0
    return n if n < 2

    fib(n - 1) + fib(n - 2)
  end

  cache :fib
end

n = Numbers.new

(0..100).each { |e| p [e, n.fib(e)] }
```

A naive attempt to refactor the `Cached` module into a `ComposedCache` class might end up looking something like this:

```ruby
class ComposedCache
  def initialize(target)
    @target = target
  end
  def cache(*method_names)
    method_names.each do |m|
      results = {}

      define_singleton_method(m) do |*a|
        results[a] ||= @target.send(m, *a)
      end
    end
  end
end

n = ComposedCache.new(Numbers.new)
n.cache(:fib)

(0..100).each { |e| p [e, n.fib(e)] }
```

Unfortunately, this code has a critical flaw in it that makes it unsuitable for general use: It caches calls made through the `ComposedCache` proxy, but it does not cache internal calls made within the objects it wraps. In practice, this makes it absolutely useless for optimizing the performance of recursive functions such as the `fib()` method we're working with here.

There is no way around this problem without modifying the wrapped object. In order to stick with composition-based modeling and still get proper caching behavior, here's what we'd need to do:

```
class ComposedCache
  def initialize(target)
    @target  = target
  end

  def cache(*method_names)
    method_names.each do |m|
      original = @target.method(m)
      results  = {}

      @target.define_singleton_method(m) do |*a|
        results[a] ||= original.call(*a)
      end

      define_singleton_method(m) { |*a| @target.send(m, *a) }
    end
  end
end
```

```ruby
n = ComposedCache.new(Numbers.new)
n.cache(:fib)

(0..100).each { |e| p [e, n.fib(e)] }
```

Such a design *would* prevent a new ancestor from being introduced into the `Numbers` object's lookup path, and it would externalize the code that actually understands how to handle the caching. However, because `ComposedCache` still directly modifies the behavior of the `Numbers` objects it wraps, it loses the benefit of encapsulation that typically comes along with composition based modeling.

We also end up with an interface that feels awkward: defining what methods ought to be cached via an instance method call does not feel nearly as natural as using a class-level macro, and might be cumbersome to integrate within a real project. There are ways to improve this interface, but that would require us to jump through a few more hoops, increasing the complexity of the implementation.

Because the `ComposedCache` expects all cached methods to be explicitly declared and it does not support automatic delegation to the underlying object, it might be cumbersome to work with – it would either need to be modified to forward all uncached method calls to the object it wraps (losing the benefits of a narrow surface), or the caller would need to keep both a reference to the original object and the composed cache object around (which is very awkward and confusing!).

Good composition-based modeling produces code that is simpler than the sum of its parts, as a direct result of strong encapsulation and well-defined interactions between collaborators. Unfortunately, our implementation of the `ComposedCache` class has none of those benefits, and so it serves as a useful (if pathological) example of the downsides of composition-based modeling.

4.2 Self-schizophrenia

When sharing behavior via decorators, it can sometimes be tricky to remember what `self` refers to. This can happen both on the proxy side (a reference to `self` accidentally refers to the proxy rather than the target), and within the target object (a reference to `self` accidentally exposes the target rather than the proxy). This common mistake can lead to subtle bugs that are tricky to detect.

A clear example of this problem can be found in the Celluloid concurrency framework. Pay attention to the lines marked #1 and #2 in the following code:

(see next page)

```ruby
require "celluloid"

class Alert
  include Celluloid

  def initialize(message, delay)
    @message = message
    @delay   = delay
    @display = Display.new
  end

  attr_reader :message

  def run
    loop do
      sleep @delay

      @display.async.render(Actor.current)  # 1
    end
  end
end

class Display
  include Celluloid

  def render(actor)
    puts actor.message
  end
end

alert = Alert.new("Foo", 5)
alert.async.run # 2

sleep
```

In the line marked #1, the `Actor.current` method is called, rather than referring to `self`. This is a direct effect of Celluloid relying on a proxy mechanism for handling its fault tolerance and concurrency functionality.

When `alert.async.run` is called on the line marked #2, `Alert#run` is not executed directly, but instead gets scheduled to be run indirectly by a proxy object. However, once the method is actually executed, `self` refers to the `Alert` object, not the proxy object that enables it to be used in a concurrent, thread-safe way. Celluloid ensures that the `Actor.current` method will return a reference to that proxy object, and this is how you can safely pass a reference to an object that you're using Celluloid with.

If this design technique sounds awkward, it's because it is. However, there isn't really a better composition-based workaround: this kind of complexity arises from the indirect access problem that we discussed in the previous section, and is worsened by the automatic delegation that is meant to make two distinct objects appear as if they were one single coherent entity.

When faced with the self-schizophrenia issue, it's important to consider how much benefit is gained by encapsulating implementation details. In the case of Celluloid, the benefit of not mixing complicated concurrency mechanics into ordinary objects is probably well worth it, but in other cases it may make sense to use an implementation sharing approach instead.

NOTE: The self-schizophrenia problem also can occur when using the ***eval*** *implementation sharing approach. However, since it not a general problem for that category, I've categorized it as more of a behavior sharing problem.*

4.3 Lack of established design practices

Although it is not a technical issue, one of the main barriers to making effective use of behavior sharing in Ruby is that most developers are simply not comfortable with using aggregation as a primary modeling technique. Ruby has lots of tools that make this style of programming easier, but they tend to take a back seat to module mixins and eval-based domain-specific interfaces.

Decorators and simple composition are definitely gaining in popularity due to the encapsulation and understandability benefits that they offer, but in many cases they are used as direct replacements for inheritance-based modeling. This leads to somewhat high-ceremony and awkward interfaces that aren't necessarily convenient or comfortable to use.

In other words, we haven't yet established idioms or practices that truly allow composition based modeling to shine: most of our libraries and frameworks still heavily rely on implementation sharing techniques, and until that changes, our applications will tend to follow in their footsteps.

This is an issue that will hopefully be resolved in time, but for now I think it's only fair to include a lack of familiarity with behavior sharing methods as something that makes code that uses them more complicated to reason about.

5. Notes and recommendations

Implementation sharing is very powerful, and that makes it a good deal more complex than behavior sharing by default. To decide which style of code reuse is better to use in a given situation, it makes sense to ask yourself whether your code actually needs direct access to the internals of its ancestors.

In the rare cases where direct access is needed, it makes sense to use as weak of a form of implementation sharing as possible. Techniques which limit global effects are most desireable, e.g. individual object mixins, eval-based domain specific interfaces, and adding methods directly to individual objects. But if you find that the setup for these techniques ends up introducing needless complexity, including a module into a class or inheriting from a base class is still an option. No matter what technique you choose, it's best to not directly rely on instance variables or private methods from ancestors, just to play it safe.

However, you might find that most of the problems you currently solve with implementation sharing methods could fairly easily be solved with a behavior sharing approach. If a little extra work is likely to save you maintenance effort in the future, and it makes the code easier to reason about, it makes sense to reach for simple composition based modeling by default. Using a dynamic decorator can also offer a reasonable middle ground when you are trying to build an object that can serve as a drop-in replacement for some other component.

If you try to go the behavior sharing route and find it's too complicated or that it has obvious drawbacks (such as in the caching example we discussed in this article), you can always go back to implementation sharing techniques. However, since most of the issues with behavior sharing tend to happen along the edge cases, and the issues with implementation sharing are baked into its core, it does make sense to try to avoid the latter where possible.

Much more research into this problem is needed. If you'd like to discuss it with me, don't hesitate to drop a message on the conversation thread over at practicingruby.com, or email me at **gregory@practicingruby.com**.

6. Further reading

There are three papers I'd recommend if you want to study these issues further:

- *Disciplined inheritance, M. Sakkinen 1989*
- *A behavioral notion of subtyping, Liskov / Wing 1994*
- *Out of the tar pit, Moseley 2006*

The first two papers deal squarely with the issues of implementation sharing vs. behavior sharing in code reuse, and the third provides a more general inquiry into what makes our programs difficult to reason about. All three are more formal than this article, but also much more in-depth.

For a Ruby-centric summary of the first two papers, see Issue 3.7 and Issue 3.8 of Practicing Ruby. However, please note that these articles only reveal a small portion of the insight to be gained from the papers listed above.