**Mixins and Compositions**

**Composition**

*Composition*, as [defined by Wikipedia](http://en.wikipedia.org/wiki/Composite_reuse_principle) is:

*a technique by which classes may achieve polymorphic behavior and code reuse by containing other classes that implement the desired functionality instead of through inheritance.*

This definition is focused on where the implementation of a particular public interface lives. With inheritance, we present a public interface whose implementation is provided by the superclass. Here, it’s done via “some other class”. In a statically typed language like Java, composition is a pain - you have to define all the methods and proxy their calls to the composed class. For a dynamic language like Ruby, classes like Delegator or a well-crafted method\_missing can make this a snap.

But we’re talking about code-reuse here. As it applies to code re-use, the term “composition” colloquially means “call methods on a private object”. In other words, if I want to re-use some shared logic, I put that logic in a class, create an object of that class, and call its methods.

This has the advantage of being easy to implement and easy to understand - it’s probably what an inexperienced person would do if they didn’t know about inheritance. Using composition in this way doesn’t affect our types - our class’ public interface remains unchanged - and doesn’t require fitting our model into some complex hierarchy.

This technique has a few downsides:

* method calls are on some object, making them more verbose
* if we create said objects as needed, we make isolated testing clumsy, difficult, or even impossible
* if we instead use [inversion of control](http://en.wikipedia.org/wiki/Inversion_of_control) and let someone else give us the objects we need, we now have to have some “container” to “wire up” all of these dependencies.

The last two issues are particularly dicey. In your average Java app, using externally configured dependency injection is the defacto standard, so you get used to writing classes based on inversion of control. In Ruby or Rails, this is not the way things work, and adding this “object container” just feels wrong. The “container” adds complexity that we’d like to avoid, even if it affords us easier isolated testing.

Is there a way to avoid the issues and restrictions around inheritance, but without the baggage of composition? There is: mixins.

**Mixins**

On Wikipedia, a [mixin](http://en.wikipedia.org/wiki/Mixins) is defined as:

*a class that provides a certain functionality to be inherited or just reused by a subclass, while not meant for instantiation*

This is a bit vague, but the idea is that we can “mix in” methods from one class into another without creating a rigid, single “is a” relationship, but without *also* having to use complex delegation to a composed object.

Ruby uses modules for mixins, and Scala uses traits. Neither are technically classes, but they *are*types.

The way in which code is re-used from a mixin is identical to the way it’s done via inheritance - the methods magically appear as if part of the class. This goes for both private/protected members *and*public members. You can also mixin as many modules/traits as you like - there’s no practical limit.

This has a lot of positive effects:

* A single place for shared logic
* Said logic is available expediently - no need to call methods on another object
* No need to manage instances of other objects or worry about inversion of control
* Isolated testing is much simpler than with inheritance

This sounds like a pretty awesome solution to the problems presented by both inheritance and composition. So, what’s the problem?

The “abuse cases” of mixins are particularly annoying. Case #1 involves including “too many” mixins. Since a mixin can contain public members, namely methods, a class with a lot of mixins will have a very large public footprint. The resulting objects begin to move away from a “type” toward a “god object”. The rules of coupling and cohesion start to come into play, and you can end up with a system where changes to a module that’s frequently included can have disastrous effects. Making this mess with inheritance or composition is *much* more difficult.

Of course, no technique should be dismissed because those with bad design taste abuse it. We find a good rule of thumb is in how these mixins are named. Mixins names should adjectives, not nouns or verbs. Although DHH refers to this pattern as “concerns”, you’ll notice that none of his mixins are named “Concerns”. He doesn’t have a PersonConcerns module, but instead has modules for various features to be added. If you can’t name your mixins as adjectives, you might be doing something wrong, and if your mixin has the word “Concerns” in it - you’ve definitely screwed up.

Abuse case #2 is a misguided attempt at code organization. To feel better about creating bloated god objects, developers will extract groups of possibly-related functionality into a mixin and then mix it into the main class. This is hiding the problem of a bloated class that does too much. Preventing this is easy - if your mixin isn’t being used by more than one class then it should not have been extracted (the naming rule helps, too - if you can’t name something you’re less likely to extract it)

There’s an interesting footnote of sorts to this technique, and that is DCI - Data, Context, and Integration.