orphan:

Problem 1: Initializers are complicated

By formalizing Objective-C's initialization conventions, we've ended up with a tower of complexity where users find it easier to do the wrong thing and then follow the compiler fix-its. I [Jordan R] still feel like the individual rules aren't so complicated:

> Designated initializers chain.
> Designated initializers are inherited if (a) there are no manual initializers, and (b) all properties have initial values.
>
> Convenience initializers delegate.
> Convenience initializers are inherited if all of the superclass's designated initializers are present.
>
> If you want to call an initializer on a dynamic type, it must be marked required.
> Protocols are one way to do this, so initializers that satisfy protocol requirements must be required.
> If your superclass has a required initializer, you must provide it somehow.

but

> "When even Andy Matuschak and Rob Rix can't understand your model, you have a problem." - Joe Groff

Problem 2: Convenience initializers are missing use cases

With all our rules, we actually rule out some important use cases, like this one on NSDocument:

> The `init` method of NSDocument is the *designated initializer,* and it is invoked by the other initializers `initWithType:error:` and `initWithContentsOfURL:ofType:error:`. If you perform initializations that must be done when creating new documents but not when opening existing documents, override `initWithType:error:`. If you have any initializations that apply only to documents that are opened, override `initWithContentsOfURL:ofType:error:`. If you have general initializations, override `init`. In all three cases, be sure to invoke the superclass implementation as the first action.
>
> —Document-Based App Programming Guide for Mac

Because we don't allow overriding convenience initializers with other convenience initializers, there's nowhere to perform post-customization of NSDocuments for each particular case.

Problem 3: Factory Initializers

Finally, we try to standardize on initializers for object creation in Swift, even going as far as to import Objective-C factory methods as initializers...but there are some patterns that cannot be written in Swift, such as this one:

```
class AnyGenerator<Element> : GeneratorType {
  init<
    WrappedGenerator: GeneratorType
  where
    WrappedGenerator.Element == Element
  >(wrapped: WrappedGenerator) -> AnyGenerator {
    return AnyGeneratorImpl(wrapped)
  }
  // other generator stuff
}

class AnyGeneratorImpl<WrappedGenerator: GeneratorType> :
    AnyGenerator<WrappedGenerator.Element> {
  var wrapped: WrappedGenerator
  init(wrapped: WrappedGenerator) {
    self.wrapped = wrapped
  }
  // other generator stuff
}
```

We ended up making `AnyGenerator` a struct that wraps `AnyGeneratorImpl` to get around this, but it's not a nice solution.

Solutions?

We've had a number of ideas for improving the state of the world, including

- Allow designated initializers to delegate to other designated initializers (using static dispatch). This makes convenience initializers a niche feature.
- Add the concept of factory initializers, which don't promise to return `Self`. These are either never inherited or must always be overridden in a subclass.
- Allow convenience initializers to chain to superclass convenience initializers. This isn't strictly safe, but it permits the

NSDocument idiom.

None of these solve all the initializer problems listed above on their own, and we'd want to be careful not to *increase* complexity in this space.