Building Interfaces and Abstract Classes in Ruby

7 Feb, 2011 · Read in about 6 min · (1140 Words)

General java ruby Tutorials

So back in the dark ages of my career, pre-2006, I spent a long time coding Java. Yeah, I know, please don't judge. Anyway, In Java, for those of you who are unaware were two constructs that I occasionally wish I had in Ruby, those areInterfaces and Abstract Classes. The difference between these two constructs is subtle, but important.

In Java an Interface is a basically a blueprint of methods that the class who implements the Interface needs to implement. For example:

```
interface Bicycle {

  void changeGear(int newValue);

  void speedUp(int increment);

  void applyBrakes(int decrement);
}

public class ACMEBicycle implements Bicycle {

   public void changeGear(int newValue) {
     // do some work here
   }

   public void speedUp(int increment) {
     // do some work here
   }

   public void applyBrakes(int decrement) {
     // do some work here
   }

}
```

Here we have a `Bicycle` Interface that says there are three methods that need to be implemented. It is then the responsibility of the `ACMEBicycle` class to implement those methods. Now, an Abstract Class in Java is similar to an Interface in that it too is a blueprint of methods that the extending class may or may not need to implement. There in lies one of the differences between the two. Let's take a look at the same example, but this time we want to implement the same behavior of all of our extending classes for the `applyBrakes` method:

```
abstract class Bicycle {

  abstract public void changeGear(int newValue);

  abstract public void speedUp(int increment);

  public void applyBrakes(int decrement) {
    // do some work here
  }

}

public class ACMEBicycle extends Bicycle {

   public void applyBrakes(int decrement) {
     // do some work here
   }

}
```

An Abstract Class is a great way to provide a mix of fully implemented methods as well as providing subclasses with a mixture of methods that need to be implemented by the extending class.

The really powerful part of all of this is two fold. First, the Java compiler will happily yell at you and fail if it finds that you haven't implemented some of the methods that you were told you had to. Second, you can easily see the methods that you need to document right there, you can even copy/paste their definitions right into your class so you can start to fill them out.

So, how does this bring us over to Ruby? Great question. I'd like to take a few moments and explore a few ways we can get some of this power in Ruby.

Unfortunately, or fortunately depending on how you look at it (I see it as a mixed blessing), there is no compiler in Ruby, so we don't really have a good way of having the system yell at us if we don't implement the methods we were supposed to. But, there is still plenty we can do to help those who are implementing our classes both know what they need to implement and to find out what they haven't implemented when their program is executing.

Here is one implementation on we can gain a bit of that functionality back in Ruby:

```
module AbstractInterface

  class InterfaceNotImplementedError < NoMethodError
  end

  def self.included(klass)
    klass.send(:include, AbstractInterface::Methods)
    klass.send(:extend, AbstractInterface::Methods)
  end
```

```
  module Methods

    def api_not_implemented(klass)
      caller.first.match(/in \`(.+)\'/)
      method_name = $1
      raise AbstractInterface::InterfaceNotImplementedError.new("#{klass.class.name} needs to im
plement '#{method_name}' for interface #{self.name}!")
    end

  end

end

class Bicycle
  include AbstractInterface

  # Some documentation on the change_gear method
  def change_gear(new_value)
    Bicycle.api_not_implemented(self)
  end

  # Some documentation on the speed_up method
  def speed_up(increment)
    Bicycle.api_not_implemented(self)
  end

  # Some documentation on the apply_brakes method
  def apply_brakes(decrement)
    # do some work here
  end

end

class AcmeBicycle < Bicycle
end

bike = AcmeBicycle.new
bike.change_gear(1) # AbstractInterface::InterfaceNotImplementedError: AcmeBicycle needs to impl
ement 'change_gear' for interface Bicycle!
```

What we've done here is to inject a Module into our `Bicycle` class to give it a nice error it can raise and a little bit of help building a nice error message for the user. Then in our `Bicycle` class we define all the methods we want and in the ones we need the end user to define we can call the `api_not_implemented` method and it will raise the `AbstractInterface::InterfaceNotImplementedError` error for us.

We could simplify this a bit by having a nice little helper macro that we can use to build these methods, like this:

```
module AbstractInterface

  class InterfaceNotImplementedError < NoMethodError
```

```
  end

  def self.included(klass)
    klass.send(:include, AbstractInterface::Methods)
    klass.send(:extend, AbstractInterface::Methods)
    klass.send(:extend, AbstractInterface::ClassMethods)
  end

  module Methods

    def api_not_implemented(klass, method_name = nil)
      if method_name.nil?
        caller.first.match(/in \`(.+)\'/)
        method_name = $1
      end
      raise AbstractInterface::InterfaceNotImplementedError.new("#{klass.class.name} needs to im
plement '#{method_name}' for interface #{self.name}!")
    end

  end

  module ClassMethods

    def needs_implementation(name, *args)
      self.class_eval do
        define_method(name) do |*args|
          Bicycle.api_not_implemented(self, name)
        end
      end
    end

  end

end

class Bicycle
  include AbstractInterface

  needs_implementation :change_gear, :new_value
  needs_implementation :speed_up, :increment

  # Some documentation on the apply_brakes method
  def apply_brakes(decrement)
    # do some work here
  end

end

class AcmeBicycle < Bicycle
end

bike = AcmeBicycle.new
bike.change_gear(1) # AbstractInterface::InterfaceNotImplementedError: AcmeBicycle needs to impl
ement 'change_gear' for interface Bicycle!
```

That approach certainly makes our code look a bit cleaner, I'm not denying that, however it has one really big flaw, at least for me anyway, it doesn't give us a good to place to hang our documentation hat. In the previous approach we had actual methods that we could then document and that documentation would then show up in RDoc when it's outputted. With the latter approach, however, we can document the hell out of the `needs_implementation` calls we have in the `Bicycle` class, but they won't ever show up in the documentation. That means that users of our library have to crack open the actual code itself to see what it they are expected to implement.

Another approach we could've taken, which I bother to demonstrate here as I don't think it offers a better approach is to have the `needs_implementation` method collect up the names of those methods and use `method_missing` to report that the method needs to be implemented. I mention it here only for completeness, but it definitely is not the best solution to this problem.

Finally, I would like to note that, as far as I can see, there is no way in Ruby to create a callback hook for when a class has been defined. If there was in fact such a hook we could use to it immediately notify the end user that they have forgotten to implement certain methods. Perhaps in Ruby 2.0??? That's just pure hope on my part.

That's it. I hope you enjoyed our brief (*cough*) look through implementing Interface and Abstract Classes in Ruby. I hope you've enjoyed it.

- PS, yes, I'm aware I didn't talk about multiple vs. single inheritance in either Java or Ruby, nor did I talk about the fact that in Ruby you can't really have Abstract Classes. I thought that was all a bit much for an already rather lengthy post as it was. Perhaps another day. :)