001 - Laying out scope/spec structure

Federico M. Iachetti

June 1, 2015

I'm going to start by laying down the basic structure for running our tests.

Our target will be to get a nested structure for our tests into place (similar to the RSpec describe/it structure).

Each one of our tests will share some context with it's children, and each nested spec will be able to add to the context without leaking up the structure.

To achieve this, we will design classes that share context using two strategies:

- Namespacing: each nested class will be literally nested inside the namespace of the wrapping class.
- Hierarchy: each nested class will be a subclass of the wrapping class.

First, we will create the 'Example' class, which will be the base for all specs:

```
module Matest
  class Example
  end
end
```

Let's write our first test. We want to have a 'scope' method on the Example object, which accepts a block and executes it.

```
module Matest
  describe Example do
    describe "#scope" do
      it "executes the block" do
```

```
        scp = Example.new.scope { "THE SCOPE CODE" }
        scp.should == "THE SCOPE CODE"
      end
    end
  end
end
```

We can make this test pass by accepting a &block argument and sending the '#call' method.

```
def scope(&block)
  block.call
end
```

Next we want to lay down a Skipping mechanism. We will write a spec that asserts that a 'Skip' will be returned if the block wasn't given

```
it "skips if no block was given" do
  scp = Example.new.scope
  scp.should == Skip
end
```

To make this pass we first need to return 'Skip' if no block was given:

```
def scope(&block)
  if block_given?
    block.call
  else
    Skip
  end
end
```

This shows a new error "NameError: uninitialized constant Matest::Example::Skip". Which we can fix by defining a 'Skip' class inside the 'Matest' module

```
module Matest
  class Skip; end
end
```

And now we're green.

You may have noticed I used 'scp' as a variable name for scope. This is because in the near future we will introduce 'scope' as a global method and we don't want to get confused.

Now we want to introduce a second method, one that specifies the actual test. We will call this method 'spec'. We'll add aliases for both this methods in the future, but for now, 'spec' will do.

To start, we'll define 'spec' the same way as 'scope' (in fact we will just copy, paste and change names):

```
module Matest
  describe Example do
    describe "#spec" do
      it "executes the block" do
        scp = Example.new.spec { "THE SPEC CODE" }
        scp.should == "THE SPEC CODE"
      end

      it "skips if no block was given" do
        scp = Example.new.spec
        scp.should == Skip
      end
    end
  end
end
```

And the same with the actual code:

```
module Matest
  class Example
    def spec(&block)
      if block_given?
        block.call
      else
        Skip
      end
    end
  end
end
```

And our specs are passing.

Now that we have both methods created, let's do something a little more interesting. Let's try to call the 'spec' method from inside the 'scope' method. This is the kind of nesting we want on our test library.

```
describe "nesting" do
  it "allows to call #spec from inside #scope" do
    scp = Example.new.scope do
      spec
    end
  end
end
```

This raises an error: "undefined local variable or method 'spec' for ..."

This means that '#spec' is not available on the current scope, which, at this point, is an instance of the 'Example' class. To solve this problem, instead of calling the passed block, we need to 'instance_eval' it on the instanced object (see 'instance_eval' video).

```
def scope(&block)
  if block_given?
    instance_eval(&block)
  else
    Skip
  end
end
```

We could make a second test calling '#scope' instead of '#spec', but it won't give us any further information, and it will be one more test to mantain in the future.

To end this episode, lets do a small refactoring. One of my rules for this project is "no 'if's" or, at least, put them in a place that doesn't get in the way. We currently have two instances of conditionals, both asserting if the block was given. To remove them we will use my 'callable' gem (see video about callable).

Lets do it one by one.

The first conditional is:

```
if block_given?
  instance_eval(&block)
else
  Skip
end
```

We can eliminate it by calling the ‘Callable‘ global method with a default value, like this

```
instance_eval(&Callable(block, default: Skip))
```

This keeps the tests passing.

We can get rid of the second conditional

```
if block_given?
  block.call
else
  Skip
end
```

in a similar way

```
Callable(block, default: Skip).call
```

And with this, we are done.