Ruby Monitor-Functions - Or Meta-Meta-Programming in Ruby

Table of Contents

Ruby Monitor-Functions

Or Meta-Meta-Programming in Ruby

Tue Aug 14 13:09:14 UTC 2007

Erik Veenstra <rubyscript2exe@erikveen.dds.nl>

1. Rationale

I had a discussion with a friend. A Java guy. He wants the arguments of a method call being checked. *"I want the first one to be an Integer. And the second one is a String. Period."* No discussion. I explained our duck-typing paradigm. He's not convinced. He thinks Java. So, he gets Java.

Lets check the types of the arguments of a method call!

(Well, this article is not about type checking at all. It's about how to implement such a type checker. Or, more general, it's about monitoring-functions.)

I wanted to do this with a nice and clean implementation, with the real magic pushed down to a place I would never come again (*write once, read never*). I wanted something like this (focus on line 8):

```
1   class Foo
2     def bar(x, y, z)
3       # x should be Numeric
4       # y should be a String
5       # z should respond to :to_s
6     end
7
8     check_types :bar, Numeric, String, :to_s
9   end
```

(Focus on line 8, once again. Make it three times. It's all about line 8.)

That was good enough for him. *"But you can't do this. You simply can't. That's magic."* I laughed at him, turned around and did it...

That's what this story is all about. To be more accurate: It's about *how* it's done, about *monitor-functions* and *method-wrapping*, not about *type-checking*.

2. Wrapping an Instance Method

First, here's a piece of code which doesn't do anything, except that it seems to wrap the original method with a block (focus on line 14):

```
 1  class Module
 2    def just_wrap(method_name)
 3      wrap_method(method_name) do |org_method, args, block|
 4        org_method.call(*args, &block)
 5      end
 6    end
 7  end
 8
 9  class Foo
10    def bar(x, y, z)
11      p [x, y, z]
12    end
13
14    just_wrap :bar
15  end
16
17  Foo.new.bar("a", "b", "c") # ===> ["a", "b", "c"]
```

You can find the implementation of `Module#wrap_method` in Implementation. This article is all about that very one method. It's the big trick. You don't need to understand its implementation. Knowing how to use it is good enough.

Line 3 retrieves the original method and yields the given block with this method, as well as with its arguments and its block. Not `*args`, not `&block`. Just `args` and `block`. Blocks don't get blocks, you know. (Although it's introduced in Ruby 1.9.)

Within the given block (near line 4), we can do whatever we want to. That's where the real stuff goes.

But, someday, we have to call the original method with the original parameters and the original block. That's what we do on line 4.

That's about it. That's the whole story. There's nothing more to say.

Except for an example or two...

Here's a simple example. It *upcases* every argument. It must be silly to *upcase* every argument like this, but we'll do it anyway. Introducing line 4:

```
 1  class Module
 2    def big_arguments(method_name)
 3      wrap_method(method_name) do |org_method, args, block|
 4        args = args.collect{|x| x.to_s.upcase}
 5
 6        org_method.call(*args, &block)
 7      end
 8    end
 9  end
10
11  class Foo
```

```
12     def bar(x, y, z)
13       [x, y, z]
14     end
15
16     big_arguments :bar
17   end
18
19   Foo.new.bar("a", "b", "c") # ===> ["A", "B", "C"]
```

Here's another example. Lines 4, 5, 6 and 7. They inform you about nil things.

```
 1   class Module
 2     def find_nil(method_name)
 3       wrap_method(method_name) do |org_method, args, block|
 4         if args.include?(nil)
 5           $stderr.puts "Found a nil when called \
 6                         from #{caller[1..-1].inspect}."
 7         end
 8
 9         org_method.call(*args, &block)
10       end
11     end
12   end
13
14   class Foo
15     def bar(x, y, z)
16     end
17
18     find_nil :bar
19   end
20
21   Foo.new.bar("a", "b", "c") # ===>
22   Foo.new.bar("a", "b", nil) # ===> Found a nil when
23                                     called from from
24                                     ["test.rb:22"].
25   Foo.new.bar("a", "b", "c") # ===>
```

I call `check_types`, `just_wrap`, `big_arguments` and `find_nil`: ***monitor-functions***. I don't know exactly how this term got into my head, but it does sound good: *monitor-functions*. It's definitely better than *wrap-method-functions*. (You can build *non-monitor-functions* as well. But that's really stupid: *monitor-and-non-monitor-functions*.)

Did I mention that it is possible to double-wrap a method with two or more monitor-functions? The order in which they are executed is, of course, bottom-up:

```
 1   class Foo
 2     def bar(x, y, z)
 3       # x should be Numeric
 4       # y should be a String
 5       # z should respond to :to_s and :gsub
 6     end
 7
 8     check_types :bar, Numeric, String, [:to_s, :gsub]
 9     log_args    :bar
10   end
```

Meanwhile, I played with a couple of monitor-functions: debugging, logging, synchronization, statistics, benchmarking, roles (like on WebSphere), #typechecking type checking, even value checking and range checking. Ideas? It's easy to create them. Try it. Let me know.

Forget about the implementation of `Module#wrap_method`. It's just sitting there, waiting to be used to implement a monitor-function. It's easy to implement a monitor-function. And it's very, very easy to use it. Those where my goals.

Oh, by the way, if such a monitor-function is kind of meta-programming (it's a buzz-word, I know, but it is, isn't it?), how would you call `wrap_method`? *Meta-meta-programming*?

3. Wrapping a Module Method or Class Method

Now, we have this `Module#wrap_method` for wrapping instance methods. But what about wrapping module methods, like `Module#wrap_module_method`?

(`Module#wrap_module_method` is deprecated. Use `aModule.metaclass.wrap_method` instead.)

We are going to take this dangerous type checking thing to the next level. Just as an example. (I'm not promoting this technique; it's just an example!)

Imagine, we want the type definition to be positioned before the method instead of after the method. Just for better readability:

```
1   class Foo
2     def_types Numeric, String, [:to_s, :gsub]
3     def bar(x, y, z)
4       # x should be Numeric
5       # y should be a String
6       # z should respond to :to_s and :gsub
7       # Very long method...
8     end
9   end
```

... instead of:

```
1   class Foo
2     def bar(x, y, z)
3       # x should be Numeric
4       # y should be a String
5       # z should respond to :to_s and :gsub
6       # Very long method...
7     end
8
9     check_types :bar, Numeric, String, [:to_s, :gsub]
10  end
```

We could do this by storing the types in def_types and overwriting Foo::method_added. But what about the old functionality in Foo::method_added? You could alias to another method and than use this alias. That's the common way to work around this problem. (I've never liked it...) But, again, we can use `wrap_method` to add the new functionality to the original method. A module is an instance itself after all, isn't it? Introducing `metaclass.wrap_method`:

```
1   class Module
2     def def_types(*types)
3       metaclass.wrap_method(:method_added) do |org_method, args, block|
4         if types
5           method_name     = args[0]
6           t               = types
7           types           = nil   # Avoid looping
8
9           check_types(method_name, *t)
10        end
11
12        org_method.call(*args, &block)    if org_method
13      end
```

```
14     end
15   end
```

But, be careful, there's a hidden loop. `def_types` wraps `method_added`. From now on, every time `method_added` is called, it calls `check_types` (line 9, but only if we removed `if types` on line 4), which calls `wrap_method`, which adds methods and thus triggers `method_added`. And so on. This `types = nil` (line 7) avoids this looping. It also applies the wrapping to *only* the next method defined in `Foo` (only to `Foo#bar1` and not to `Foo#bar2`). You only have to add this mechanism when wrapping `Module#method_added`, not for other module methods.

Do you see that this `metaclass.wrap_method` looks like `wrap_method`? They should look the same. They are brother and sister.

Once again, it should be possible to wrap the wrapper:

```
 1   class Foo
 2     def_types Numeric, String, [:to_s, :gsub]
 3     def_stat "/tmp/stats.log"
 4
 5     def bar1(x, y, z)
 6     end
 7
 8     def bar2(x, y, z) # bar2 is neither logged, nor checked.
 9     end
10   end
```

4. Module#pre_condition and Module#post_condition

Since doing checks on arguments before executing an instance method is very common, I've made `Module#pre_condition`. You give it the names of the instance methods you want to wrap (as symbols or as strings) and you give it a block. This block is executed *in the original context* and receives `|obj, method_name, args, block|`.

There's also a `Module#post_condition`.

Example:

```
 1  class Foo
 2    def bar(*args, &block)
 3      p [:in_method, args, block]
 4
 5      yield(*args)
 6    end
 7
 8    pre_condition(:bar) do |obj, method_name, args, block|
 9      p [:in_pre_condition, args]
10    end
11
12    post_condition(:bar) do |obj, method_name, args, block|
13      p [:in_post_condition, args]
14    end
15  end
16
17  foo     = Foo.new
18
19  foo.bar(1, 2, 3, 4) do |*args|
20    p [:in_block, args]
21  end
```

Which results in:

```
[:in_pre_condition, [1, 2, 3, 4]]
[:in_method, [1, 2, 3, 4], #<Proc:0xb7d7f868@.....:19>]
[:in_block, [1, 2, 3, 4]]
[:in_post_condition, [1, 2, 3, 4]]
```

Use `metaclass` as receiver of `Module#pre_condition` or `Module#post_condition` to wrap a module method instead of an instance method:

```
 1  require "ostruct"
 2
 3  class Foo < Struct.new(:aa, :bbb)
 4    metaclass.post_condition(:new) do |obj, method_name, args, block|
 5      line = caller.select{|s| s.include?(__FILE__)}.shift.scan(/\d+/)[-1]
 6
 7      puts "An object of class #{self} has been created."
 8      puts "The call looked like: #{self}.new(#{args.inspect})."
 9      puts "It's done on line: #{line}."
10      puts
```

```
11    end
12  end
13
14  Foo.new(11, 111)
15  Foo.new(22, 222)
16  Foo.new(33, 333)
```

Which results in:

```
An object of class Foo has been created.
The call looked like: Foo.new([11, 111]).
It's done on line: 14.

An object of class Foo has been created.
The call looked like: Foo.new([22, 222]).
It's done on line: 15.

An object of class Foo has been created.
The call looked like: Foo.new([33, 333]).
It's done on line: 16.
```

Use `obj.instance_eval` to execute part the block in the context of the object, instead of in the original context:

```
 1  require "ostruct"
 2
 3  class Foo < Struct.new(:aa, :bbb)
 4    post_condition(:initialize) do |obj, method_name, args, block|
 5      line = caller.select{|s| s.include?(__FILE__)}.shift.scan(/\d+/)[-1]
 6
 7      obj.instance_eval do
 8        puts "An object of class #{self.class} has been created."
 9        puts "The arguments are: @aa=#{aa} and @bbb=#{bbb}."
10        puts "The object has id: #{__id__}."
11        puts "It's done on line: #{line}."
12        puts
13      end
14    end
15  end
16
17  Foo.new(11, 111)
18  Foo.new(22, 222)
19  Foo.new(33, 333)
```

Which results in:

```
An object of class Foo has been created.
The arguments are: @aa=11 and @bbb=111.
The object has id: -605321068.
It's done on line: 17.

An object of class Foo has been created.
The arguments are: @aa=22 and @bbb=222.
The object has id: -605321448.
It's done on line: 18.

An object of class Foo has been created.
The arguments are: @aa=33 and @bbb=333.
```

```
The object has id: -605321898.
It's done on line: 19.
```

It is possible to wrap a non-existing method. And since the block is, well, uh, a block, and not a method definition, it captures the local variables.

```
 1  class Foo
 2    x     = 7
 3
 4    post_condition(:bar) do
 5      puts "Variable x is captured in the closure and has value #{x}."
 6    end
 7
 8    x      = 8
 9  end
10
11  Foo.new.bar
```

Which results in:

```
Variable x is captured in the closure and has value 8.
```

(Although they're named `*_condition`, they're not checking anything. They should be named `*_action`. But `pre_action` is harder to remember than `pre_condition`. So I stick to the latter.)

5. Implementation

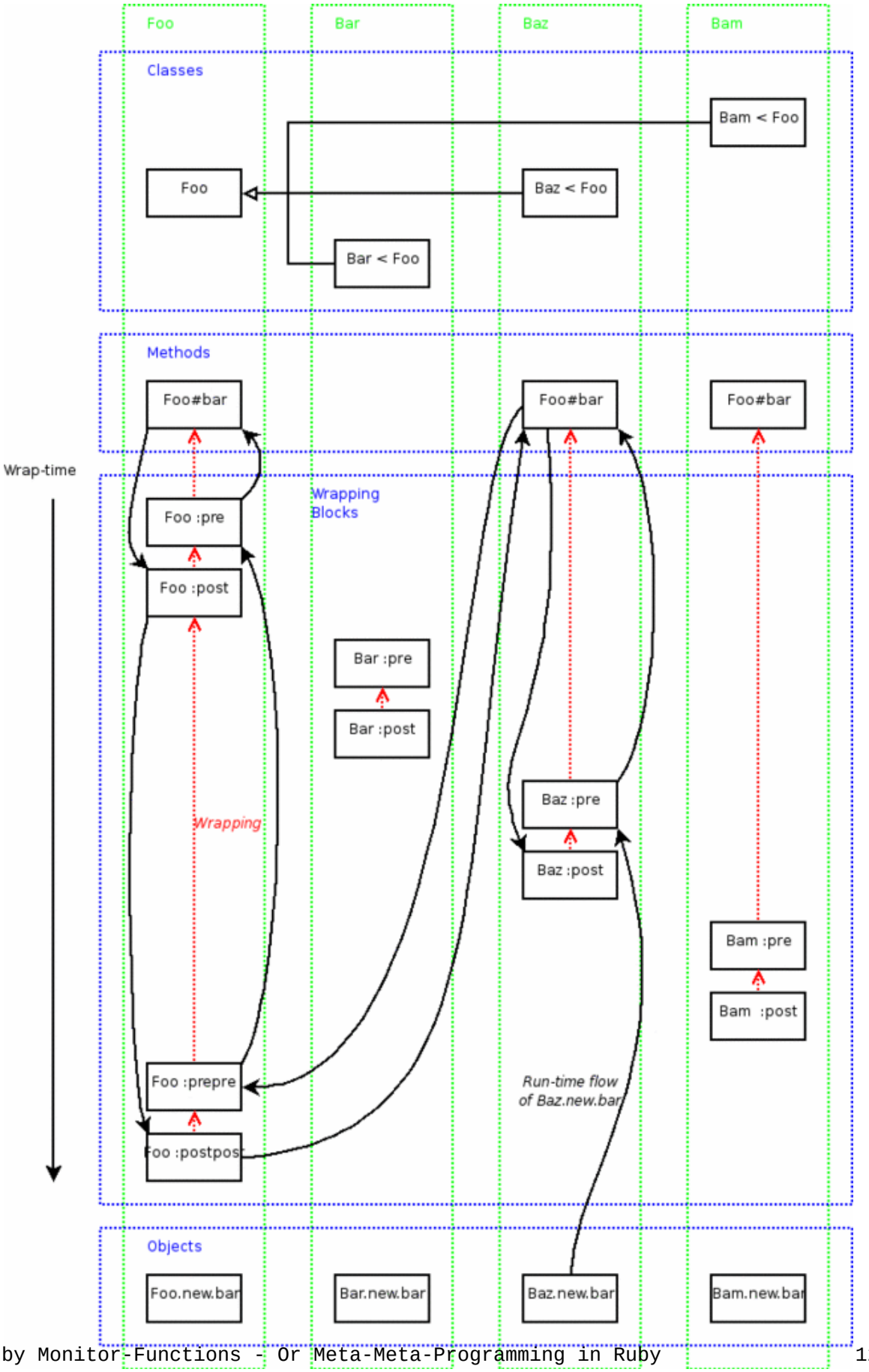
5.1. Order of Execution of Recursively Wrapped Methods

One more minute...

In normal Ruby, when a method of an object is called, it's searched for in the class of that object. If it's not found, Ruby tries to find it in the superclass, and so on.

When wrapping methods, it's basically the same.

At wrap-time, you can only wrap methods in the specified class, not methods in the superclass. At run-time, this chain of blocks is called in reversed order. At the end (beginning?) of the chain, the original method is called. If there's no original method, the superclass is searched for its method (or chain).


Foo
Bar
Baz
Bam
Classes
Bam < Foo
Foo
Baz < Foo
Bar < Foo
Methods
Foo#bar
Foo#bar
Foo#bar
Wrap-time
Wrapping Blocks
Foo :pre
Foo :post
Bar :pre
Bar :post
Baz :pre
Baz :post
Bam :pre
Bam :post
Wrapping
Foo :prepre
Foo :postpost
Run-time flow of Baz.new.bar
Objects
Foo.new.bar
Bar.new.bar
Baz.new.bar
Bam.new.bar

5. Implementation

Imagine this code:

```
 1  class Foo
 2    def bar
 3      p [Foo, :bar]
 4    end
 5
 6    pre_condition(:bar)   {p [Foo, :pre]}
 7    post_condition(:bar)  {p [Foo, :post]}
 8  end
 9
10  class Bar < Foo
11    pre_condition(:bar)   {p [Bar, :pre]}
12    post_condition(:bar)  {p [Bar, :post]}
13  end
14
15  class Baz < Foo
16    def bar
17      p [Baz, :bar, :before]
18      super
19      p [Baz, :bar, :after]
20    end
21
22    pre_condition(:bar)   {p [Baz, :pre]}
23    post_condition(:bar)  {p [Baz, :post]}
24  end
25
26  class Bam < Foo
27    def bar
28      p [Bam, :bar, :before]
29      # no super
30      p [Bam, :bar, :after]
31    end
32
33    pre_condition(:bar)   {p [Bam, :pre]}
34    post_condition(:bar)  {p [Bam, :post]}
35  end
36
37  class Foo
38    pre_condition(:bar)   {p [Foo, :prepre]}
39    post_condition(:bar)  {p [Foo, :postpost]}
40  end
41
42  puts
43  Foo.new.bar
44  puts
45  Bar.new.bar
46  puts
47  Baz.new.bar
48  puts
51  Bam.new.bar
```

Which results in:

```
[Foo, :prepre]
[Foo, :pre]
[#<Foo:0xb7d618b8>, :bar]
[Foo, :post]
[Foo, :postpost]
```

```
[Bar, :pre]
[Foo, :prepre]
[Foo, :pre]
[#<Bar:0xb7d61250>, :bar]
[Foo, :post]
[Foo, :postpost]
[Bar, :post]

[Baz, :pre]
[#<Baz:0xb7d60878>, :bar, :before]
[Foo, :prepre]
[Foo, :pre]
[#<Baz:0xb7d60878>, :bar]
[Foo, :post]
[Foo, :postpost]
[#<Baz:0xb7d60878>, :bar, :after]
[Baz, :post]

[Bam, :pre]
[#<Bam:0xb7d5ec18>, :bar, :before]
[#<Bam:0xb7d5ec18>, :bar, :after]
[Bam, :post]
```

5.2. The Real Stuff

Finally!

If you want this code to be available as a gem, please shout. If enough people want it, I might build one.

About the license: If you want to copy this code to your application, go ahead. Just don't sell this code as a standalone solution. That's not fair.

```
require "thread"

class Module

        # Meta-Meta-Programming

        # With this, we can create monitoring functions.

        # It might not be clearly readable,
        # but it's written only once.
        # Write once, read never.
        # Forget about the internals.
        # Just use it.
        # It should be part of Ruby itself, anyway... :)

        # This wrap_method is low-level stuff.
        # If you just want to add code to a method, scroll
        # down to pre_condition and post_condition.
        # They're much easier to use.

  def wrap_method(*method_names, &block1)
    raise ArgumentError, "method_name is missing"       if method_names.empty?
    raise ArgumentError, "block is missing"             unless block1
```

```
    Thread.exclusive do
      method_names.flatten.each do |method_name|
        count =
        Module.module_eval do
          @_wm_count_ ||= 0
          @_wm_count_ +=1
        end

        module_eval <<-EOF

            # Get the method which is to be wrapped.

          method = instance_method(:"#{method_name}")   rescue nil

            # But it shouldn't be defined in a super class...

          if method.to_s != "#<UnboundMethod: " + self.to_s + "##{method_name}>"
            method      = nil
          end

          if method.nil? and ($VERBOSE or $DEBUG)
            $stderr.puts \
              "Wrapping a non-existing method ["+self.to_s+"##{method_name}]."
          end

            # Store the method-to-be-wrapped and the wrapping block.

          define_method(:"_wm_previous_#{method_name}_#{count}_") do
            [method, block1]
          end

            # Avoid this stupid "warning: method redefined".

          unless :#{method_name} == :initialize
            undef_method(:"#{method_name}")     rescue nil
          end

            # Define __class__ and __kind_of__.

          define_method(:__class__) \
                    {Object.instance_method(:class).bind(self).call}

          define_method(:__kind_of__) \
                    {|s| Object.instance_method(:"kind_of?").bind(self).call(s)}

            # Define the new method.

          def #{method_name}(*args2, &block2)
            if self.__kind_of__(Module)
              context   = metaclass
            else
              context   = self.__class__
            end

              # Retrieve the previously stored method-to-be-wrapped (old),
              # as well as the wrapping block (new).
              # Note: An UnboundMethod of self.superclass.metaclass can't be
              # bound to self.metaclass, so we "walk up" the class hierarchy.
```

```
          previous    = context.instance_method(
                              :"_wm_previous_#{method_name}_#{count}_")

          begin
            previous  = previous.bind(zelf ||= self)
          rescue TypeError => e
            retry     if zelf = zelf.superclass
          end

          old, new    = previous.call

            # If there's no method-to-be-wrapped in the current class, we
            # should look for it in the superclass.

          old ||=
            context.superclass.instance_method(:"#{method_name}") rescue nil

            # Since old is an unbound method, we should bind it.
            # Note: An UnboundMethod of self.superclass.metaclass can't be
            # bound to self.metaclass, so we "walk up" the class hierarchy.

          begin
            old &&= old.bind(zelf ||= self)
          rescue TypeError => e
            retry     if zelf = zelf.superclass
          end

            # Finally...

          new.call(old, args2, block2, self)
        end
      EOF
    end
  end
end

def wrap_module_method(*method_names, &block1)        # Deprecated
  if $VERBOSE or $DEBUG
    $stderr.puts "Module#wrap_module_method is deprecated."
    $stderr.puts "Use aModule.metaclass.wrap_method instead."
  end

  metaclass.wrap_method(*method_names, &block1)
end

      # Since adding code at the beginning or at the
      # end of an instance method is very common, we
      # simplify this by providing the next methods.
      # Althoug they're named *_condition, they're
      # not checking anything. They should be named
      # *_action. But pre_action is harder to remember
      # than pre_condition. So I stick to the latter.

def pre_condition(*method_names, &block1)
  pre_and_post_condition(true, false, *method_names, &block1)
end

def post_condition(*method_names, &block1)
  pre_and_post_condition(false, true, *method_names, &block1)
end
```

```
  def pre_and_post_condition(pre, post, *method_names, &block1)
    method_names.flatten.each do |method_name|
      wrap_method(method_name) do |org_method, args2, block2, obj2|
        block1.call(obj2, method_name, args2, block2)   if pre

        res     = org_method.call(*args2, &block2)      if org_method

        block1.call(obj2, method_name, args2, block2)   if post

        res
      end
    end
  end

end

class Object

  def metaclass
    class << self
      self
    end
  end

end
```

5.3. Unit Tests

Work in progress...

```
require "ev/metameta"   # This is where I store my meta-meta-programming stuff.
require "test/unit"

class TestMetaMeta1 < Test::Unit::TestCase
  TESTRESULT    = []

  class Foo
    def bar
      TESTRESULT << [Foo, :bar]

      :foo_bar
    end

    pre_condition(:bar)   {TESTRESULT << [Foo, :pre]}
    post_condition(:bar)  {TESTRESULT << [Foo, :post]}
  end

  class Bar < Foo
    pre_condition(:bar)   {TESTRESULT << [Bar, :pre]}
    post_condition(:bar)  {TESTRESULT << [Bar, :post]}
  end

  class Baz < Foo
    def bar
      TESTRESULT << [Baz, :bar, :before]
      super
      TESTRESULT << [Baz, :bar, :after]
```

```
    :baz_bar
  end

  pre_condition(:bar)   {TESTRESULT << [Baz, :pre]}
  post_condition(:bar)  {TESTRESULT << [Baz, :post]}
end

class Bam < Foo
  def bar
    TESTRESULT << [Bam, :bar, :before]
    # no super
    TESTRESULT << [Bam, :bar, :after]

    :bam_bar
  end

  pre_condition(:bar)   {TESTRESULT << [Bam, :pre]}
  post_condition(:bar)  {TESTRESULT << [Bam, :post]}
end

class Foo
  pre_condition(:bar)   {TESTRESULT << [Foo, :prepre]}
  post_condition(:bar)  {TESTRESULT << [Foo, :postpost]}
end

def setup
  TESTRESULT.clear
end

def test_foo
  res = Foo.new.bar

  assert_equal(:foo_bar, res)
  assert_equal([      [Foo, :prepre],
                      [Foo, :pre],
                      [Foo, :bar],
                      [Foo, :post],
                      [Foo, :postpost]],
                      TESTRESULT)
end

def test_bar
  res = Bar.new.bar

  assert_equal(:foo_bar, res)
  assert_equal([      [Bar, :pre],
                      [Foo, :prepre],
                      [Foo, :pre],
                      [Foo, :bar],
                      [Foo, :post],
                      [Foo, :postpost],
                      [Bar, :post]],
                      TESTRESULT)
end

def test_baz
  res = Baz.new.bar

  assert_equal(:baz_bar, res)
  assert_equal([      [Baz, :pre],
```

```
                        [Baz, :bar, :before],
                        [Foo, :prepre],
                        [Foo, :pre],
                        [Foo, :bar],
                        [Foo, :post],
                        [Foo, :postpost],
                        [Baz, :bar, :after],
                        [Baz, :post]],
                        TESTRESULT)
  end

  def test_bam
    res = Bam.new.bar

    assert_equal(:bam_bar, res)
    assert_equal([      [Bam, :pre],
                        [Bam, :bar, :before],
                        [Bam, :bar, :after],
                        [Bam, :post]],
                        TESTRESULT)
  end
end

class TestMetaMeta2 < Test::Unit::TestCase
  TESTRESULT    = []

  def setup
    TESTRESULT.clear
  end

  class Foo
    def self.xyz
      TESTRESULT << :xyz2

      :foo_xyz
    end

    metaclass.pre_condition(:xyz) do |o, m, a, b|
      TESTRESULT << :xyz1
    end
  end

  class Bar < Foo
  end

  class Bam < Foo
    metaclass.post_condition(:xyz) do |o, m, a, b|
      TESTRESULT << :xyz3
    end
  end

  def test_xyz_bar
    res = Bar.xyz

    assert_equal(:foo_xyz, res)
    assert_equal([:xyz1, :xyz2], TESTRESULT)
  end

  def test_xyz_bam
    res = Bam.xyz
```

```
    assert_equal(nil, res)
    assert_equal([:xyz3], TESTRESULT)
  end
end
```

6. Real-World Examples

6.1. Type-Checking

In this example, we implement a type-checking mechanism with `wrap_method`.

```
require "ev/metameta"   # This is where I store my meta-meta-programming stuff.

class Module

        # Type checking.
        # Or duck-type checking.

        # Example:
        # class Foo
        #   def_types String, Numeric, [:to_s, :gsub]
        #   def :bar(x, y, x)
        #     # x should be Numeric
        #     # y should be a String
        #     # z should respond to :to_s and :gsub
        #   end
        # end

  def def_types(*types)
    metaclass.pre_condition(:method_added) do |obj, method_name, args, block|
      if types
        method_name     = args[0]
        t               = types
        types           = nil   # Avoid looping

        check_types(method_name, *t)
      end
    end
  end

        # Example:
        # class Foo
        #   def :bar(x, y, x)
        #     # x should be Numeric
        #     # y should be a String
        #     # z should respond to :to_s and :gsub
        #   end
        #   check_types :bar, String, Numeric, [:to_s, :gsub]
        # end

  def check_types(method_names, *types)
    [method_names].flatten.each do |method_name|
      pre_condition(method_name) do |obj, method_name, args, block|
        args.each_with_index do |arg, n|
          [types[n]].flatten.each do |typ|
            if typ.kind_of?(Module)
              unless arg.kind_of?(typ)
                raise ArgumentError, "wrong argument type " +
                                     "(#{arg.class} instead of #{typ}, " +
                                     "argument #{n+1})"
              end
            elsif typ.kind_of?(Symbol)
```

```
              unless arg.respond_to?(typ)
                raise ArgumentError, "#{arg} doesn't respond to :#{typ} " +
                                     "(argument #{n+1})"
              end
            else
              raise ArgumentError, "wrong type in types " +
                                   "(#{typ}, argument #{n+1})"
            end
          end
        end
      end
    end
  end

end
```

And this is a little test script:

```
require "ev/types"
require "test/unit"

class TestTypes < Test::Unit::TestCase
  class Thing
    def_types Numeric, String, [:gsub, :to_s]

    def go(x, y, z)
      # x should be Numeric
      # y should be a String
      # z should respond to :gsub and :to_s

      throw "BOOM"      if x == 0

      [x, y, z]
    end
  end

  def test_ok
    assert_equal([7, "8", "9"], Thing.new.go(7, "8", "9"))
  end

  def test_first_parm_not_numeric
    assert_raise(ArgumentError) {Thing.new.go("7", "8" "9")}
  end

  def test_second_parm_not_string
    assert_raise(ArgumentError) {Thing.new.go(7, 8, "9")}
  end

  def test_third_parm_not_responding
    assert_raise(ArgumentError) {Thing.new.go(7, "8", 9)}
  end

  def test_exception_comes_through
    assert_raise(NameError)     {Thing.new.go(0, "8", "9")}
  end
end
```

6.2. Value-Checking

This example checks an argument of a call for certain values. It accepts arrays of values or ranges (everything that responds to `:include?`). `nil` means: no check.

```
require "ev/metameta"   # This is where I store my meta-meta-programming stuff.

class Module

        # Example:
        # class Foo
        #   def :bar(x, y, z)
        #     # z should be :abc or :xyz
        #   end
        #   check_values :bar, nil, nil, [:abc, :xyz]
        # end

  def check_values(method_names, *values)
    [method_names].flatten.each do |method_name|
      pre_condition(method_name) do |obj, method_name, args, block|
        args.each_with_index do |arg, n|
          unless values[n].nil? or values[n].include?(args[n])
            argss       = args.collect{|s| s.inspect}.join(", ")
            call        = "%s#%s(%s)" % [self, method_name, argss]

            raise ArgumentError, "value of argmument #{n+1} is invalid [#{call}]"
          end
        end
      end
    end
  end

end
```

And this is a little test script:

```
require "ev/values"
require "test/unit"

class TestValues < Test::Unit::TestCase
  class Thing
    def go(x, y, z)
       # x can have any value
       # y should be in the range 20..29
       # z should be :abc or :xyz

      throw "BOOM"      if x == 0

      [x, y, z]
    end

    check_values  :go,    nil, 20..29, [:abc, :xyz]
                #  ^^^     ^^^  ^^^^^^  ^^^^^^^^^^^^
                # method    x      y          x
  end

  def test_ok
    assert_equal([11, 22, :abc], Thing.new.go(11, 22, :abc))
```

```
  end

  def test_second_parm_not_correct
    assert_raise(ArgumentError) {Thing.new.go(11, 33, :abc)}
  end

  def test_third_parm_not_correct
    assert_raise(ArgumentError) {Thing.new.go(11, 22, :def)}
  end

  def test_exception_comes_through
    assert_raise(NameError)     {Thing.new.go(0, 22, :abc)}
  end
end
```

6.3. Once

In this example, we implement a caching mechanism with `wrap_method`. The given method will be executed only once. The result is cached. The next time the method is called, the result is returned immediately. The drawback is that the object itself should be frozen to enforce some integrity. For the same reason, blocks are prohibited. You can pass `true` to freeze the object on the fly, or pass `false` to skip this check.

```
require "ev/metameta"   # This is where I store my meta-meta-programming stuff.
require "thread"

class Module

  def once(*method_names)
    forced_freeze       = method_names.include?(true)
    ignore_frozen       = method_names.include?(false)
    method_names        = method_names.select{|x| x.kind_of?(Symbol) or
                                                  x.kind_of?(String)}

    Thread.exclusive do
      pre_condition(:freeze) do |obj, method_name, args, block|
        Thread.exclusive do
          unless obj.instance_variable_get("@_once_results_")
            obj.instance_variable_set("@_once_results_", {})
          end
        end
      end
    end

    [method_names].flatten.each do |method_name|
      wrap_method(method_name) do |org_method, args, block, obj|
        obj.freeze      if forced_freeze and not obj.frozen?

        raise "object must be frozen"   unless obj.frozen? or ignore_frozen
        raise "block handling is not (yet) implemented" if block

        results = nil

        if obj.frozen?
          results = obj.instance_variable_get("@_once_results_")
        else
          Thread.exclusive do
            unless results = obj.instance_variable_get("@_once_results_")
```

```
              results = obj.instance_variable_set("@_once_results_", {})
            end
          end
        end

        results[[method_name, args]] ||= org_method.call(*args) if org_method
      end
    end
  end

end
```

And this is a little test script:

```
require "ev/once_method"
require "test/unit"

class TestOnceMethod < Test::Unit::TestCase
  TESTRESULT    = []

  class Foo
    def initialize
      freeze
    end

    def bar3(n)
      TESTRESULT << [:bar3, n]

      fail "BOOM!"      if n == 0

      3*n
    end

    def bar4(n)
      TESTRESULT << [:bar4, n]

      4*n
    end

    once :bar3
  end

  def setup
    TESTRESULT.clear
  end

  def test_ok
    foo = Foo.new

    assert_equal(18, foo.bar3(6))
    assert_equal(18, foo.bar3(6))
    assert_equal(21, foo.bar3(7))
    assert_equal(21, foo.bar3(7))
    assert_equal(32, foo.bar4(8))
    assert_equal(32, foo.bar4(8))
    assert_equal(36, foo.bar4(9))
    assert_equal(36, foo.bar4(9))

    assert_equal([      [:bar3, 6],
                        [:bar3, 7],
```

```
                        [:bar4, 8],
                        [:bar4, 8],
                        [:bar4, 9],
                        [:bar4, 9]], TESTRESULT)
  end

  def test_block_not_supported
    assert_raise(RuntimeError)  {Foo.new.bar3(1){2}}
  end

  def test_exception_comes_through
    assert_raise(RuntimeError)  {Foo.new.bar3(0)}
  end
end
```

6.4. Singleton

These are implementations of singleton. (Don't use them! Stick to the original!)

```
require "ev/metameta"   # This is where I store my meta-meta-programming stuff.
require "thread"

class Module
  def singleton1
    metaclass.wrap_method(:new) do |org_method, args, block, obj|
      Thread.exclusive do
        @_instance ||= org_method.call(*args, &block)
      end
    end
  end
end
```

Or better:

```
require "ev/metameta"   # This is where I store my meta-meta-programming stuff.
require "thread"

class Module
  def singleton2
    metaclass.wrap_method(:new) do |org_method, args, block, obj|
      Thread.exclusive do
        unless @_instance
          @_instance = org_method.call(*args, &block)

          metaclass.wrap_method(:new) do # Use define_method for pure speed...
            @_instance
          end
        end
      end

      @_instance
    end
  end
end
```

We could even use once.

```
require "ev/once_method"

class Module
  def singleton3
    metaclass.once :new, true
  end
end
```

And this is a little test script:

```
require "ev/singleton_by_wrap_1"
require "ev/singleton_by_wrap_2"
require "ev/singleton_by_wrap_3"
require "test/unit"

class TestSingleton123 < Test::Unit::TestCase
  class Foo1
    singleton1
  end

  class Foo2
    singleton2
  end

  class Foo3
    singleton3
  end

  def test_singleton1
    id1 = Foo1.new.object_id
    id2 = Foo1.new.object_id
    id3 = Foo1.new.object_id

    assert_equal(id1, id2)
    assert_equal(id1, id3)
  end

  def test_singleton2
    id1 = Foo2.new.object_id
    id2 = Foo2.new.object_id
    id3 = Foo2.new.object_id

    assert_equal(id1, id2)
    assert_equal(id1, id3)
  end

  def test_singleton3
    id1 = Foo3.new.object_id
    id2 = Foo3.new.object_id
    id3 = Foo3.new.object_id

    assert_equal(id1, id2)
    assert_equal(id1, id3)
  end
end
```

6.5. Abstract Methods

This enforces subclasses to implement certain methods.

(This really slows down your application. You might want to use it only in debug and test environments.)

```
require "ev/metameta"

class Class
  def abstract_method(*method_names)
    metaclass.pre_condition(:new) do |obj, method_name, args, block|
      method_names.each do |method_name|
        obj.instance_eval do
          unless instance_methods.include?(method_name.to_s)
            raise NotImplementedError,
                  "Class #{self} doesn't implement method #{method_name}."
          end
        end
      end
    end
  end
end
```

And this is a little test script:

```
require "ev/abstract"
require "test/unit"

class TestAbstractMethods < Test::Unit::TestCase
  class Foo
    abstract_method :bam        # if `hostname` == ...
  end

  class Bar < Foo
  end

  class Baz < Foo
    def bam
      :ok
    end
  end

  def test_foo
    assert_raise(NotImplementedError){Foo.new}
  end

  def test_bar
    assert_raise(NotImplementedError){Bar.new}
  end

  def test_baz
    assert_nothing_raised{Baz.new}
  end
end
```

6. Real-World Examples