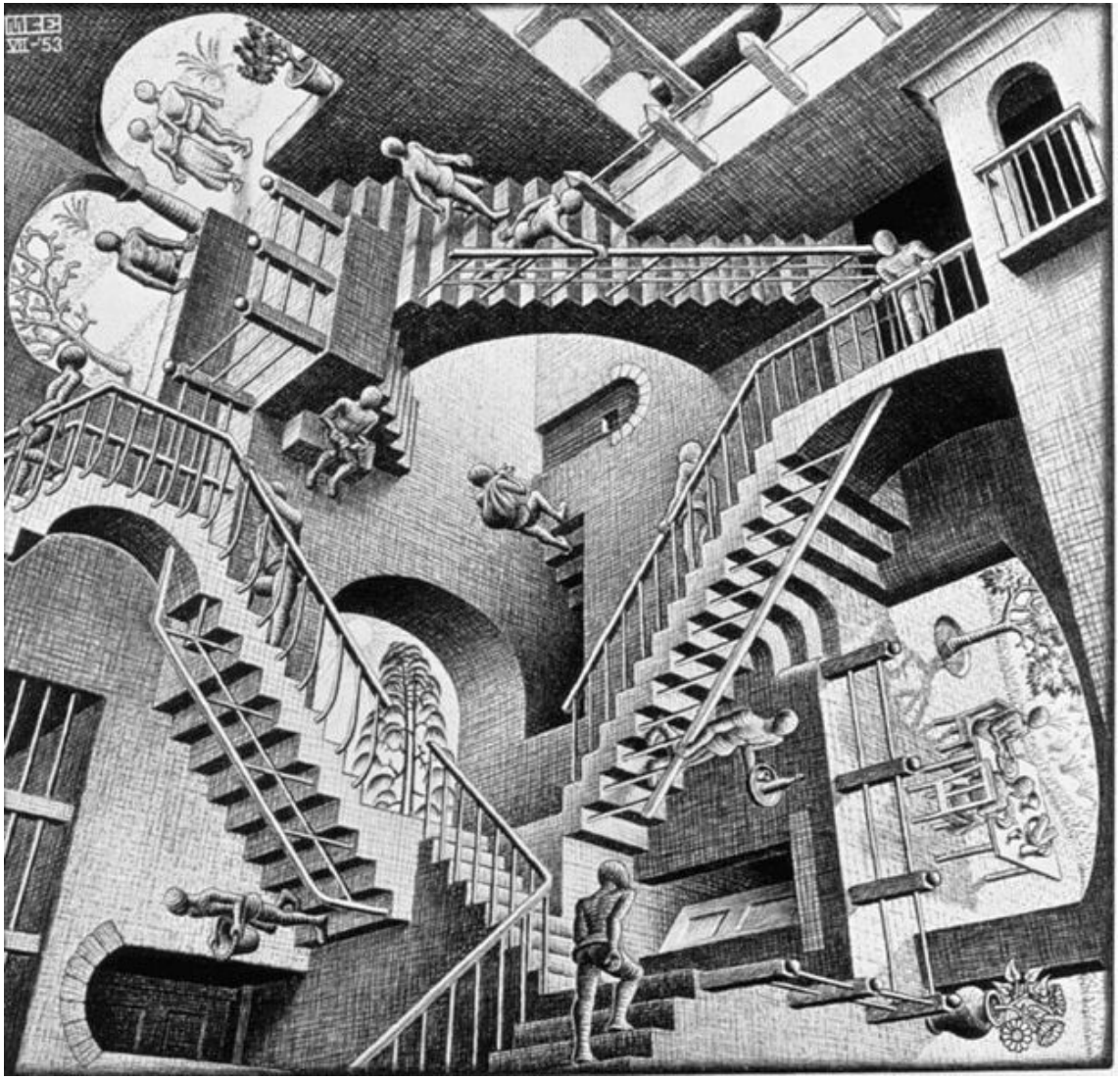
Metaprogramming Ruby

Jarosław Rzeszótko

Gadu-Gadu SA

sztywny@gmail.com // www.stifflog.com

“All Cretans are liars” - Epimenides, Cretan



What is metaprogramming?

meta - *change, transformation, alternation; beyond, transcend, more comprehensive; at a higher state of development; having undergone metamorphosis;*

– **The American Heritage Dictionary of the English Language**

Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves) as their data or that do part of the work during compile time that is otherwise done at run time. In many cases, this allows programmers to get more done in the same amount of time as they would take to write all the code manually.

– **Wikipedia**

What /really/ is metaprogramming?

Metaprogramming is writing **psychodelic code**.

If you're like me it's also:

- All the chapters you've skipped or skimmed over in your Ruby book
- All the uber-cool code snippets you've always wanted to write but never knew how
- All the _why's programs you never understood
- All the miraculous Rails methods that work "somehow"

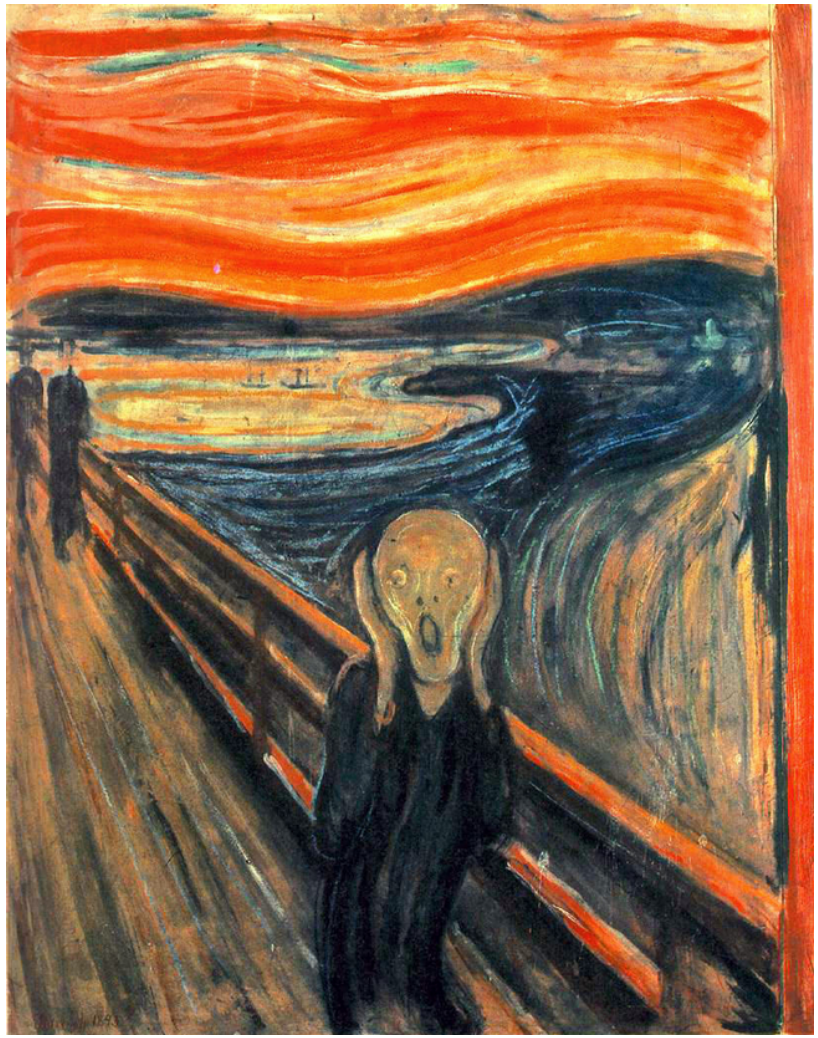
But Ruby is easy, right?

```
class Article < ActiveRecord::Base
  belongs_to :author
  has_many   :comments
end
```

```
fixtures :articles
describe Article do
  it "should have many comments" do
    articles(:test_article).should have(3).comments
  end
end
```

Rrrright...

```
def define_expected_method(sym)
  if target_responds_to?(sym) && !@proxied_methods.include?(sym)
    if metaclass.instance_methods.include?(sym.to_s)
      metaclass.__send__(:alias_method, munge(sym), sym)
    end
    @proxied_methods << sym
  end

  metaclass_eval(<<-EOF, __FILE__, __LINE__)
    def #{sym}(*args, &block)
      __mock_proxy.message_received :#{sym}, *args, &block
    end
  EOF
end
```



How Ruby achieves metaprogramming capabilities?

- Short answer:
  - through a Smalltalk-style object model and Lisp-style closures
- Long answer:
  - Everything is an object
  - Everything is modifiable at runtime
  - Deep reflection capabilities
  - Classes and objects are always “open”
  - Closures capture the context they were taken it

“Some may say Ruby is a bad rip-off of Lisp or Smalltalk, and I admit that. But it is nicer to ordinary people” - **Matz**

Ruby's object model #1

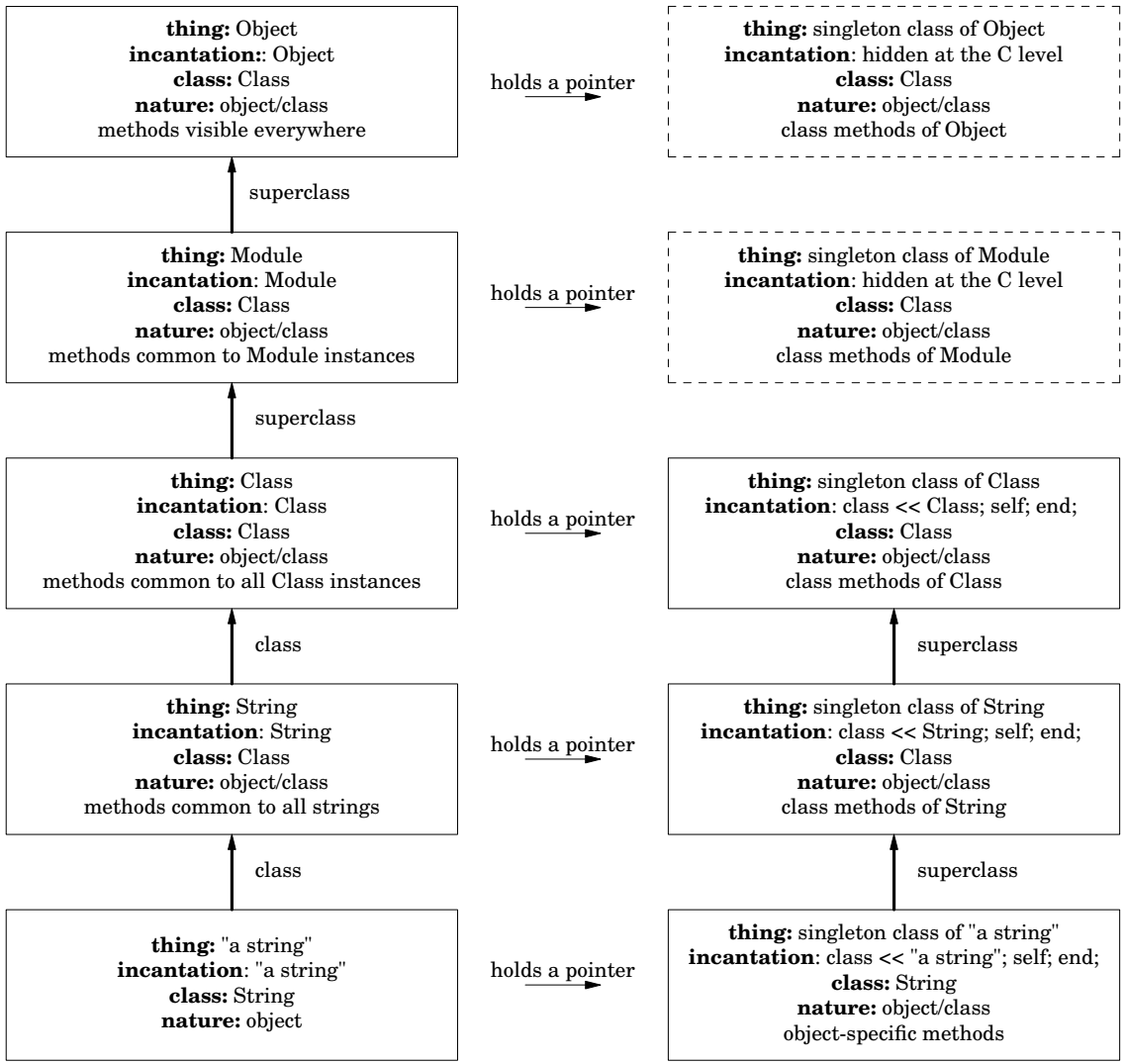
A Ruby koan:

```
>> Class.is_a? Object
=> true
>> Object.is_a? Class
=> true
```

And there are other not so obvious things:

```
class SomeClass
  def self.whats_this?; class << self; self; end;  end
end

SomeClass.class       # => Class
SomeClass.whats_this? # => #<Class:SomeClass>
```

Ruby's object model #2



Ruby's object model #3

Just as in Escher's pictures, there are some illusions going on that may fool you. When studying the topics we're discussing, you may encounter many "strange loops" - as in the "koan" we've shown or when checking superclasses of classes / singleton classes. The C code may show you how simple tricks can cause this confusion:

```
static VALUE rb_class_superclass(VALUE klass)
{
    VALUE super = RCLASS(klass)->super;
    if (!super) {
        rb_raise(rb_eTypeError, "uninitialized class");
    }
    if (FL_TEST(klass, FL_SINGLETON)) {
        super = RBASIC(klass)->klass;
    }
    while (TYPE(super) == T_ICLASS) {
        super = RCLASS(super)->super;
    }
    if (!super) {
        return Qnil;
    }
    return super;
}
```

Ruby's object model #4

In fact, classes are just instances of Class bound to a constant. Their special classyness is hidden deep in the C code of Ruby which adds to the overall confusion when trying to grok the tiny details of Ruby's OO. Anyway, those two definitions are equivalent (well, through the use of blocks the scoping may be a bit different):

```
class Foo
  def bar
    puts "ehhlo worldh"
  end
end


Foo = Class.new
Foo.class_eval do
  def bar
    puts "ehhlo worldh"
  end
end
```

Try to understand the basic, but don't worry about all this too much. It's just that the things used in the C code to make Ruby work don't map conceptually very well to Ruby's high-level abstractions. It really doesn't matter most of the time, through.

The tricks - introspection #1

```
# "Convention over configuration", kind of
class TestSuite
  def run
    self.methods.grep(/^test/).each { |m| send m }
  end

  def test_one_thing
    puts "I'm testing one thing"
  end

  def test_another_thing
    puts "I'm testing another thing"
  end

  def test_something
    puts "I'm testing something thing"
  end
end
```

The tricks - introspection #2

Module#included:

```
module Hook1
  def self.included(in_what)
    puts in_what.inspect
  end
end
class Test1; include Hook1; end
# ruby test1.rb
"Test1"
```

Class#inherited:

```
class Hook2
  def self.inherited(by_what)
    by_what.inspect # => "Test2"
  end
end
class Test2 < Hook2; end
# ruby test2.rb
"Test2"
```

Module#method_added:

```
class Parent
  def self.method_added(method_name)
    puts method_name
  end
end

class Test2 < Parent
  def hello; end;
  def world; end;
end
# ruby test2.rb
hello
world
```

The tricks - accessing things

A class is just a constant, so we can fetch it easily:

```
class Things
  attr_reader :things

  def self.thing(name, kind)
    thing = Kernel.const_get(kind.to_s.capitalize)
    @things ||= []
    @things << thing.new
  end

  # thing :i, :fixnum - wrong!
  thing :s, :string
  thing :o, :object
  thing :h, :hash
end

Things.things # => ["", #<Object:0x77d1ac78>, {}]
```

The tricks - evaluating things #1

Both instance_eval and module_eval (aka. class_eval) change the meaning of "self" inside of the supplied block or string, changing it to the object on which the method was called. To get the difference between those two, it's best to look at the C source of Ruby:

```
static VALUE specific_eval(int argc, VALUE* argv, VALUE klass, VALUE self)
...

VALUE rb_obj_instance_eval(int argc, VALUE* argv, VALUE self)
{
    VALUE klass;
    if (SPECIAL_CONST_P(self))
        klass = Qnil;
    else
        klass = rb_singleton_class(self);
    return specific_eval(argc, argv, klass, self);
}

VALUE
rb_mod_module_eval(int argc, VALUE* argv, VALUE mod)
{
    return specific_eval(argc, argv, mod, mod);
}
```

The tricks - evaluating things #2

As you see, instance_eval changes the context of language constructs like “def” to the singleton class where class_eval uses the “normal” class. Because of that (and having in mind the places instance / class methods reside in), instance_eval lets you create class methods, and class_eval instance methods. Some other methods change the context too ie. define_method.

```
class AClass; end

AClass.class_eval do
  def instance_meth; "I'm an instance method"; end
end

AClass.instance_eval do
  def class_meth;    "I'm a class method";     end
end

AClass.new.instance_meth # => "I'm an instance method"
AClass.class_meth        # => "I'm a class method"
```

The tricks - defining things #1

Have you ever thought how methods like attr_accessor work?

```
class Base
  def self.my_attr(a)
    define_method(a)       {       instance_variable_get("@#{a.to_s}") }
    define_method("#{a}=") { |val| instance_variable_set("@#{a.to_s}", val) }
  end
end

class Child < Base
  my_attr :x
  def initialize
    @x = 5

    @x          # => 5
    self.x      # => 5
    self.x = 8
    @x          # => 8
    self.x      # => 8
  end
end
```

Define_method lets you create methods using variables instead of literals for their names.

The tricks - defining things #2

Have you thought how Rails adds those dynamic finders to your classes?

```
class Person
  def self.enhance_me(description)
    singleton_class = class << self; self; end;
    singleton_class.instance_eval do
      define_method(:introduce_yourself) do
        "I am a #{self.inspect}, #{description}"
      end
    end
  end
end

class Programmer < Person;  enhance_me "I write programs"; end
class Painter    < Person;  enhance_me "I paint pictures"; end

Programmer.introduce_yourself # => "I am a Programmer, I write programs"
Painter.introduce_yourself    # => "I am a Painter, I paint pictures"
```

Through we define a method in the parent class, using it alters the child class. Also, as singleton class inheritance order follows that of normal class inheritance, the added methods will be available also in child classes of Painter or Programmer.

The tricks - defining things #3

You can't really add instance variables when doing things from inside class (think has_many), so you have to do some kind of lazy initialization:

```
class Base
  def self.create_variable(name, &block)
    define_method(name) do
      var_name = "@#{name.to_s}".intern
      instance_variable_get(var_name) if instance_variable_defined?(var_name)
      instance_variable_set(var_name, instance_eval(&block))
    end
  end
end

class Slacker < Base
  create_variable(:foo) { @x + @y }
  def initialize
    @x = 4
    @y = 7
    self.foo # => 11
  end
end
```

The tricks - wrapping things

Poor man's Aspect-Oriented Programming:

```
module Aspects
  def precondition(on_method, &condition)
    alias_method("___#{on_method}", on_method)
    define_method(on_method) do |*args|
      return send("___#{on_method}", *args) if condition.call(args)
      "Failed precondition, method '#{on_method}' won't be called"
    end
  end
end

class Calculator
  class << self; include Aspects; end

  def add(x1, x2); x1+x2; end
  def sub(x1, x2); x1-x2; end
  def mul(x1, x2); x1*x2; end
  def div(x1, x2); x1/x2; end

  precondition(:add) { |x1, x2| x1.is_a?(Fixnum) && x2.is_a?(Fixnum) }
  precondition(:sub) { |x1, x2| x1.is_a?(Fixnum) && x2.is_a?(Fixnum) }
  precondition(:mul) { |x1, x2| x1.is_a?(Fixnum) && x2.is_a?(Fixnum) }
  precondition(:div) { |x1, x2| x1.is_a?(Fixnum) && x2.is_a?(Fixnum) && x2 != 0 }
end

Calculator.new.add(3, 7)   # => 10
Calculator.new.div(5, 0)   # => "Failed precondition, method 'div' won't be called"
Calculator.new.add("x", 5) # => "Failed precondition, method 'add' won't be called"
```

Notice how we use alias_method to add additional behaviour to an existing method.

The tricks - DSLs / Higher Order Messanging #1

We want to be able to write cool things like this:

```
people = People.new([
  Person.new("Zed Shaw",                  true),
  Person.new("Guido Van Rossum",          false),
  Person.new("Jay Fields",                true),
  Person.new("Bill Gates",                false),
  Person.new("David Heinemeier Hansson", true)
])

people.who.program_ruby.are.cool

# ruby dsl.rb
Zed Shaw is definietly cool!
Jay Fields is definietly cool!
David Heinemeier Hansson is definietly cool!
```

And we want them to be extensible.

The tricks - DSLs / Higher Order Messanging #2

The Person class itself should be simple, we don't want the "meta" parts to clutter the logic too much:

```
class Person
  def programs_ruby; @programs_ruby; end
  def make_cool;     puts "#@name is definietly cool!"; end

  def initialize(name, programs_ruby)
    @name          = name
    @programs_ruby = programs_ruby
  end
end
```

The tricks - DSLs / Higher Order Messanging #3

The People class gathering several Persons is simple too, it only returns the appropiate proxy class for the given prefix like “who” or “are”:

```
class People
  attr_accessor :collection

  def who; SelectProxy.new(self); end
  def are; EachProxy.new(self); end

  def initialize(collection)
    @collection = collection
  end
end
```

When you call the “who” method on “people” (as in “people.who”), you send a message “who” to the “people” receiver in Smalltalk terminology. Thinking in terms of messages and receivers can alter the way you do OO.

The tricks - DSLs / Higher Order Messanging #4

The magic happens in the proxy classes - consider the SelectProxy we use for handling "who":

```
class SelectProxy
  def initialize(people); @people = people; end
  def method_missing(id, *args)
    @people.collection =
      @people.collection.select do |x|
        x.send("#{id.to_s.gsub("_", "s_")}", *args)
      end
    @people
  end
end
```

Method_missing lets you capture any message sent to an object that wasn't previously captured by any explicit method definition. The "higer order" part comes from the fact that the SelectProxy doesn't interpret the received messages on its own. Instead, it forwards the received message to every object in the collection of interest, retaining only the objects for which it returned true.

The tricks - DSLs / Higher Order Messanging #5

Putting it all together, we have:

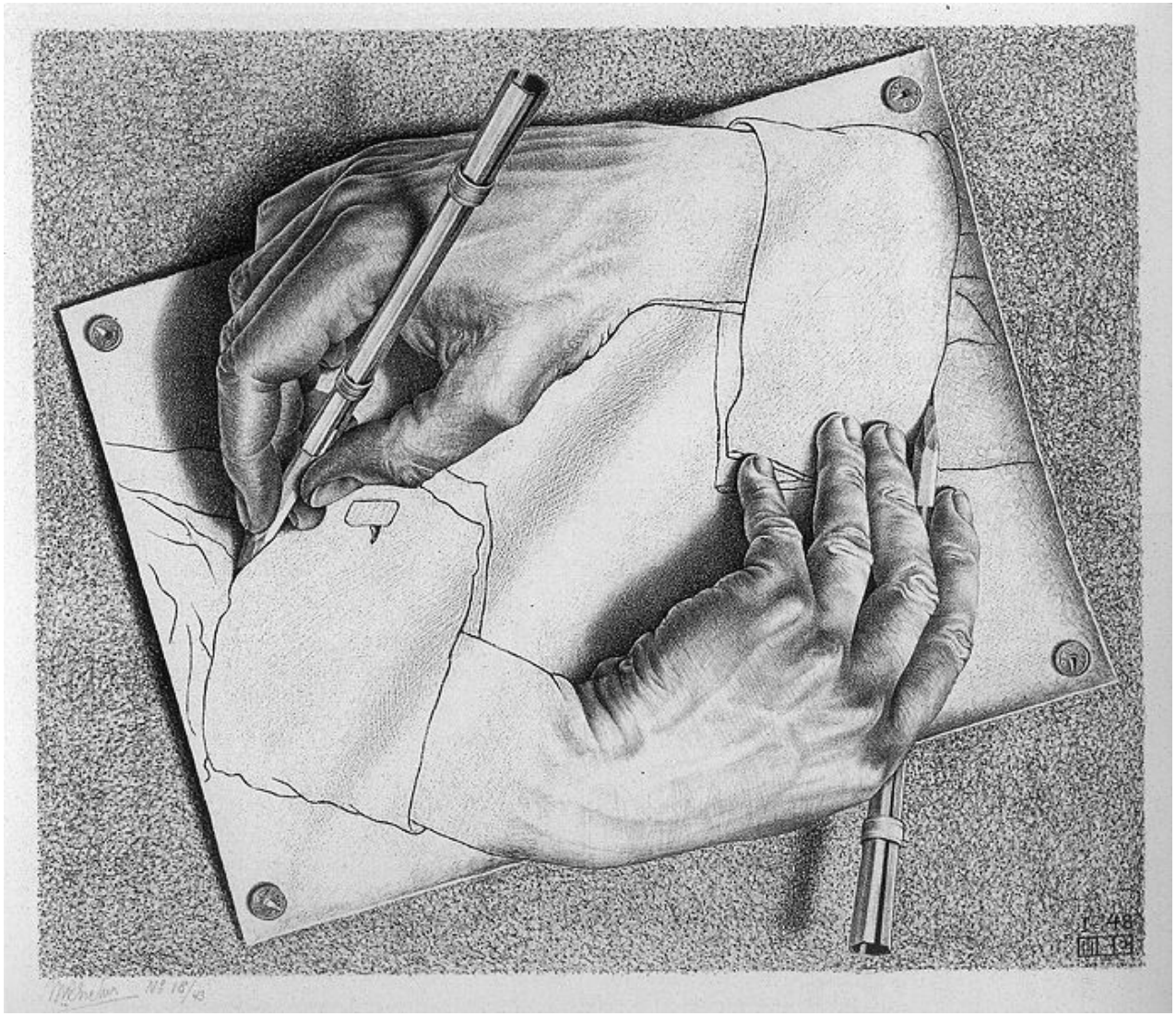
```
class SelectProxy
  def initialize(people); @people = people; end
  def method_missing(id, *args)
    @people.collection =
      @people.collection.select do |x|
        x.send("#{id.to_s.gsub("_", "s_")}", *args)
      end
    @people
  end
end

class EachProxy
  def initialize(people); @people = people; end
  def method_missing(id, *args)
    @people.collection.each do |x|
      x.send("make_#{id}", *args)
    end
  end
end

class People
  attr_accessor :collection

  def who; SelectProxy.new(self); end
  def are; EachProxy.new(self); end

  def initialize(collection)
    @collection = collection
  end
end
```

```
class Person
  def programs_ruby; @programs_ruby; end
  def make_cool;     puts "#@name is definietly cool!"; end

  def initialize(name, programs_ruby)
    @name          = name
    @programs_ruby = programs_ruby
  end
end

people = People.new([
  Person.new("Zed Shaw",                  true),
  Person.new("Guido Van Rossum",          false),
  Person.new("Jay Fields",                true),
  Person.new("Bill Gates",                false),
  Person.new("David Heinemeier Hansson", true)
])

people.who.program_ruby.are.cool

# ruby dsl.rb
Zed Shaw is definietly cool!
Jay Fields is definietly cool!
David Heinemeier Hansson is definietly cool!
```

The tricks - DSLs / Higher Order Messanging #6

As a sidenote: in the real world, both of the Proxy classes should have all of the non-crucial methods undefined and we would use "__send__" instead of send, so that we don't get any name clashes. See Jim Weirichs talk on advanced Ruby class design for more details.

Really going meta



ParseTree + Ruby2Ruby #1

Imagine a class for Logo-style programming, where you're steering a turtle, that holds a pencil (skipping the graphics details):

```
class Picture
  def left(distance)
    print "left: #{distance} "
  end
  def right(distance)
    print "right: #{distance} "
  end
  def fwd(distance)
    print "forward: #{distance} "
  end
  def bwd(distance)
    print "backward: #{distance} "
  end
  def pencil_down
    print "starting drawing "
  end
  def pencil_up
    print "stopping drawing "
  end
end
```

ParseTree + Ruby2Ruby #2

This hypothetic class has one flaw - we have to write programs like this:

```
p = Picture.new
p.left(30)
p.pencil_down
p.fwd(100)
p.left(90)
p.fwd(100)
p.pencil_up
```

Where we would like to do it simply like that:

```
p = Picture.new.left(30).pencil_down.fwd(100).left(90).fwd(100).pencil_up
```

For this to work, all the methods should return “self” as the last statement.

ParseTree + Ruby2Ruby #3

There are a couple of ways to modify the program, but imagine we have 100 or more existing methods and notice a simple search replace won't make it here (ie. "end" can also mean the end of the block or class), and creating sort of a macro for defining methods isn't such a good idea neither. We will use our two secret weapons here - **ParseTree** and **Ruby2Ruby**.

ParseTree + Ruby2Ruby #4

That's a really nice parse tree that you have:

```
%w{pp ruby2ruby}.each { |l| require l }

# I'm code, but I'm data too
class Picture
  def left(distance)
    print "left: #{distance} "
  end
  def right(distance)
    print "right: #{distance} "
  end
  def fwd(distance)
    print "forward: #{distance} "
  end
  def bwd(distance)
    print "backward: #{distance} "
  end
  def pencil_down
    print "starting drawing "
  end
  def pencil_up
    print "stopping drawing "
  end
end

pp ParseTree.translate(Picture)
```

```
# [:class,
#  :Picture,
#  [:const, :Object],
#  [:defn,
#   :bwd,
#   [:scope,
#    [:block,
#     [:args, :distance],
#     [:fcall,
#      :print,
#       [:array, [:dstr, "backward: ", [:evstr, [:lvar, :distance]], [:str, " "]]]]]]],
#  [:defn,
#   :fwd,
#   [:scope,
#    [:block,
#     [:args, :distance],
#     [:fcall,
#      :print,
#       [:array, [:dstr, "forward: ", [:evstr, [:lvar, :distance]], [:str, " "]]]]]],
#  ...
#  [:defn,
#   :right,
#   [:scope,
#    [:block,
#     [:args, :distance],
#     [:fcall,
#      :print,
#      [:array,
#       [:dstr, "right: ", [:evstr, [:lvar, :distance]], [:str, " "]]]]]]]
```

ParseTree + Ruby2Ruby #5

The parse tree enables you analyze the semantics of the program - instead of seeing raw text, you have all the logical pieces of your program split-out and tagged with their type/meaning. The power of Lisp lies in the fact that there is not much distincton between the source code and the “meaningful” tree - so modyfing the code tree isn’t some kind of a special operation, it’s an explict part of the language. In Ruby, you have to convert the tree back to code...

ParseTree + Ruby2Ruby #6

Finally, DIY automatic refactoring:

```
%w{pp ruby2ruby picture}.each { |l| require l }

def enable_chaining(class_tree, result=[])
  if class_tree.first == :block
    result << (class_tree << [:self])
  else
    class_tree.each do |node|
      node.is_a?(Symbol) ? result << node : enable_chaining(node, result)
    end
  end
end

eval(Ruby2Ruby.new.process(enable_chaining(ParseTree.translate(Picture))))
Picture.new.pencil_down.fwd(10).left(90).fwd(10).pencil_up
# ruby r2r.rb
starting drawing forward: 10 left: 90 forward: 10 stopping drawing
```

For more practical example, check out the Heckle project.

Rubinius

Hey, now you can play even with things that were once hardcoded in C!

```
VALUE rb_class_new(VALUE super) {
    Check_Type(super, T_CLASS);
    if (super == rb_cClass) {
        rb_raise(rb_eTypeError,
          "can't make subclass of Class");
    }
    if (FL_TEST(super, FL_SINGLETON)) {
        rb_raise(rb_eTypeError,
          "can't make subclass of virtual class");
    }
    return rb_class_boot(super);
}

VALUE rb_class_boot(VALUE super) {
    NEWOBJ(klass, struct RClass);
    OBJSETUP(klass, rb_cClass, T_CLASS);
    klass->super = super;
    klass->iv_tbl = 0;
    klass->m_tbl = 0;           /* safe GC */
    klass->m_tbl = st_init_numtable();
    OBJ_INFECT(klass, super);
    return (VALUE)klass;
}
[...]
```

vs.

```
class Class
  protected :object_type
  protected :has_ivars
  protected :needs_cleanup

  def initialize(sclass=Object)
    unless sclass.kind_of?(Class)
      raise TypeError, "superclass must be\
        a Class (#{sclass.class} given)"
    end
    @instance_fields = sclass.instance_fields
    @has_ivar = sclass.has_ivars
    @needs_cleanup = sclass.needs_cleanup
    @object_type = sclass.object_type
    @superclass = sclass

    super()
    [...]
  end
  [...]
end
```

The final step - dropping Ruby semantics and going freestyle

TreeTop, Ragel, Racc/Rex, Lex/Yacc

- Create a grammar
- Generate a scanner/parser from it
- Traverse the resulting tree
- Generate Ruby, C, assembly, machine code
- Create whatever you can imagine

And TreeTop is like meta-meta!

```
grammar Arithmetic
  rule additive
    multitive '+' additive / multitive
  end

  rule multitive
    primary '*' multitive / primary
  end

  rule primary
    '(' additive ')' / number
  end

  rule number
    [1-9] [0-9]*
  end
end
```

Summary - everytime you're metaprogramming, God kills a kitten

- If you've got a new hammer, everything looks like a nail
- So don't be "evil" when other people are around
- When you're alone, being "evil" is actually recommended

Resources #1

- **http://mwrc2008.confreaks.com/03bowkett.html**
  A great talk by Giles Bowkett about Ruby, Lisp, Perl, code generation & Ruby2Ruby
- **http://nat.truemesh.com/archives/000535.html**
  A more detailed explanation of higher-order messaging
- **http://whytheluckystiff.net/articles/seeingMetaclassesClearly.html**
  An article by _why explaining metaclasses
- **http://redhanded.hobix.com/inspect/theBestOfMethod_missing.html**
  Another nice one by _why with method_missing use cases
- **http://ola-bini.blogspot.com/2006/09/ruby-metaprogramming-techniques.html**
  A good overview of Ruby metaprogramming techniques
- **http://rubyquiz.com/quiz67.html**
  A great exercise in metaprogramming from the RubyQuiz
- **http://rubyconf2007.confreaks.com/d1t1p2_advanced_ruby_class_design.html**
  Jim Weirichs talk on advanced Ruby class design
- **http://innig.net/software/ruby/closures-in-ruby.rb**
  An explanation of closures in Ruby, with a lot of examples

Resources #2

- **The Ruby Programming Language**
  by David Flanagan,Yukihiro Matsumoto

  The new standard Ruby book.
- **Structure And Interpretation Of Computer Programs**
  by Harold Abelson, Gerald Jay Sussman, Julie Sussman

  Just read it. And again.
- **Programming Language Pragmatics**
  by Michael L. Scott

  If you want to know how programming languages really work.
- **Goedel, Escher, Bach**
  by Douglas Hofstadter

  The canonical book for all things “meta”.
- **On Lisp**
  by Paul Graham

  If you think metaprogramming in Ruby is scary you should try to read it.