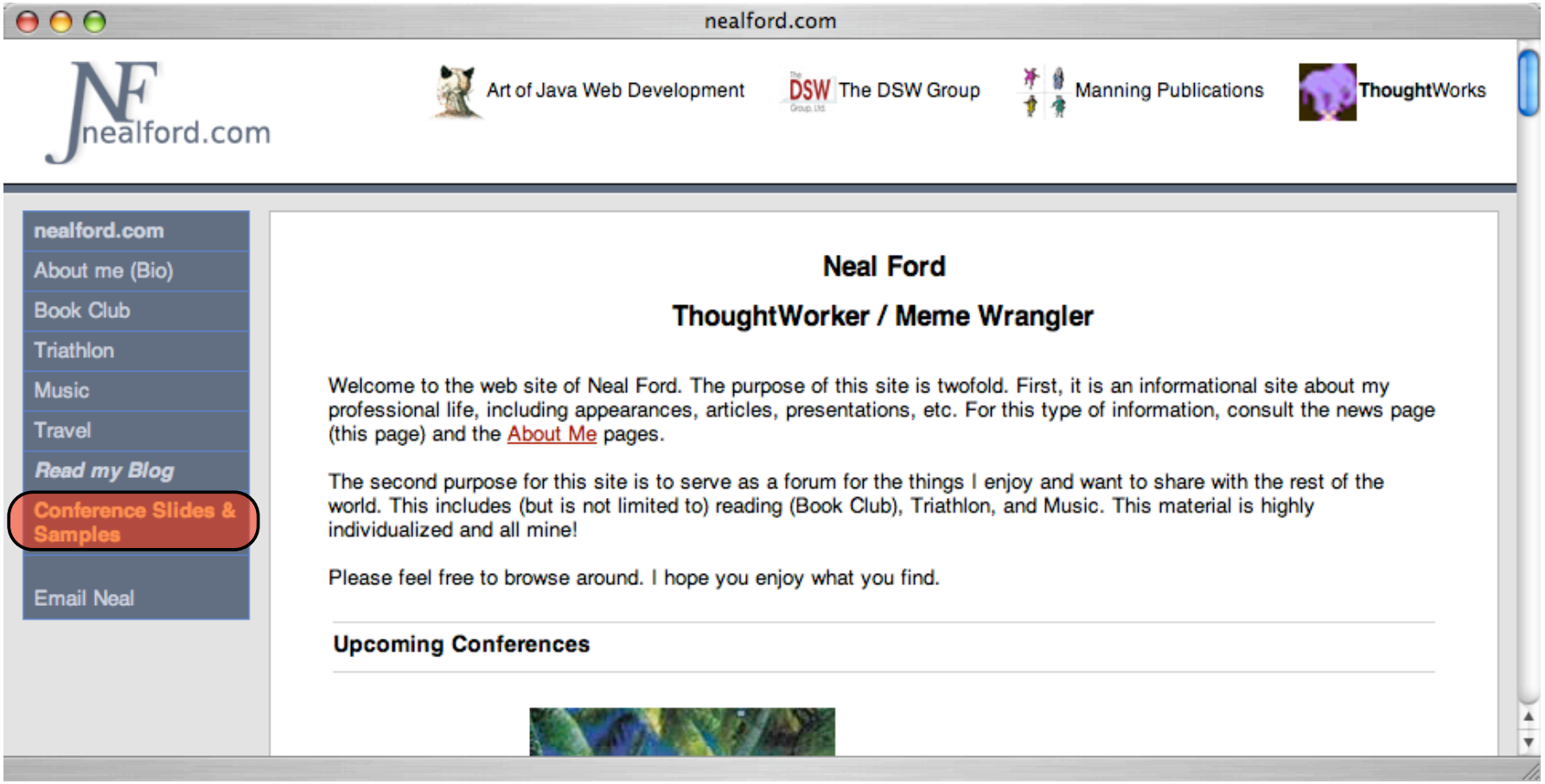
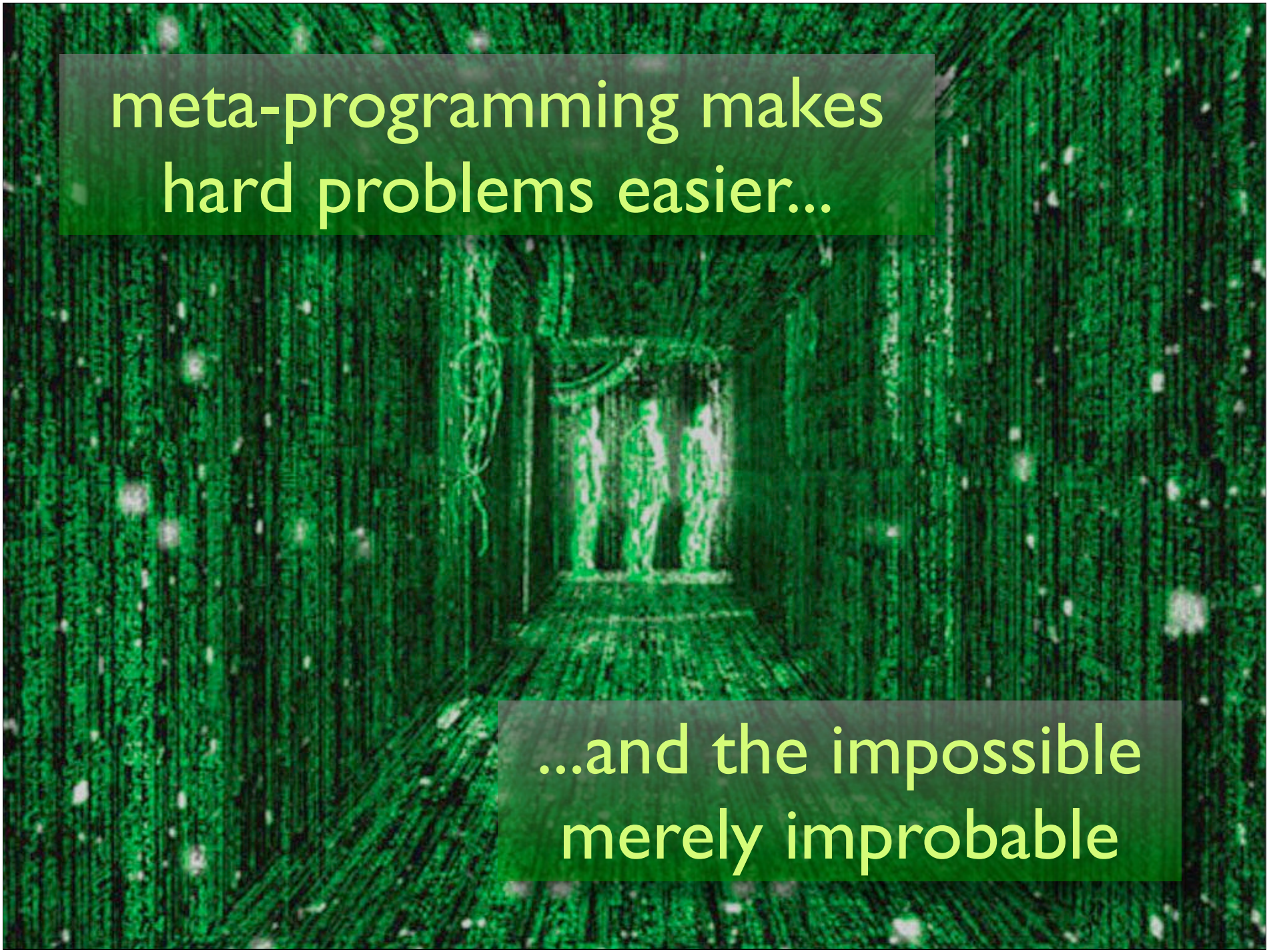
meta-programming ruby for fun & profit

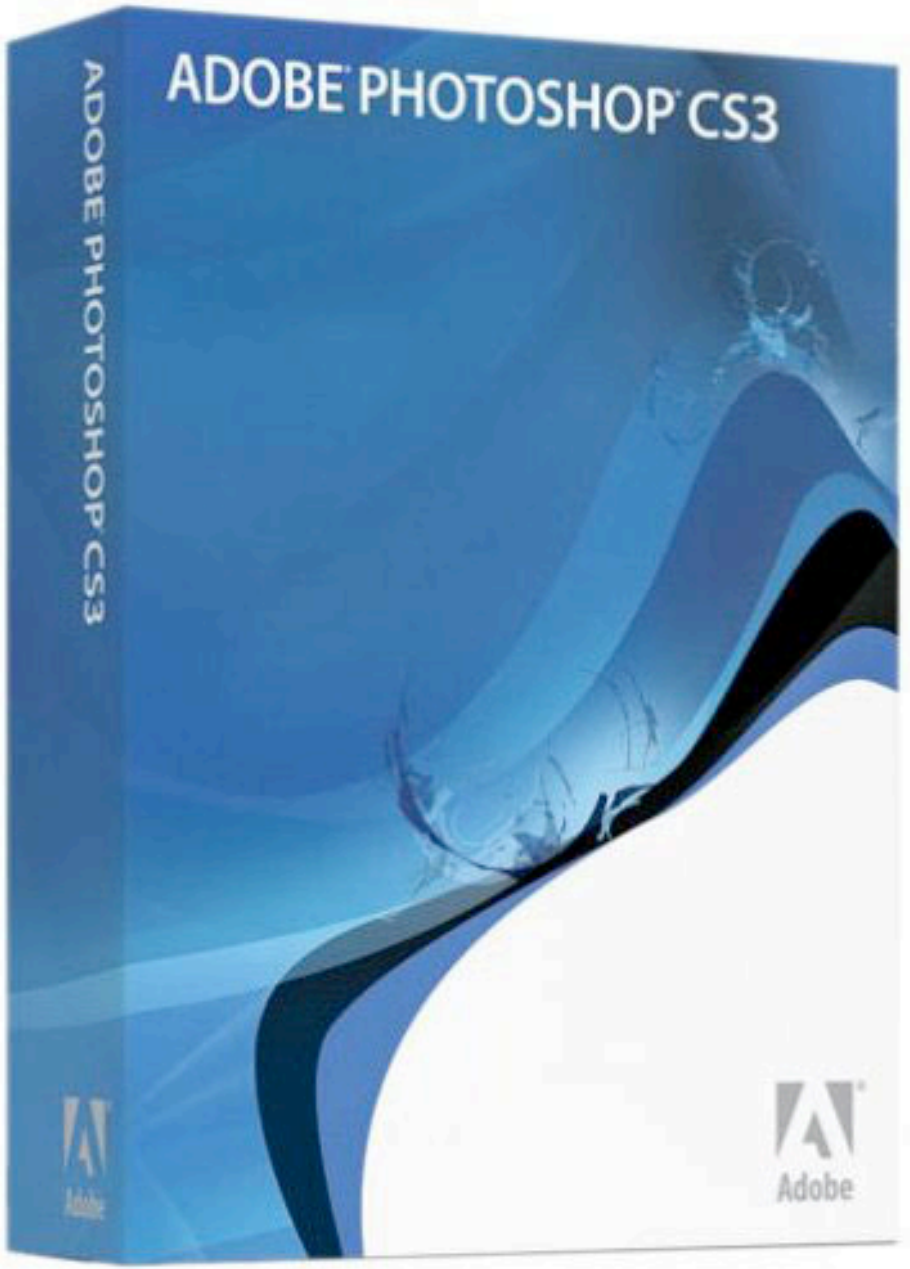
NEAL FORD software architect / meme wrangler

ThoughtWorks

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
memeagora.blogspot.com

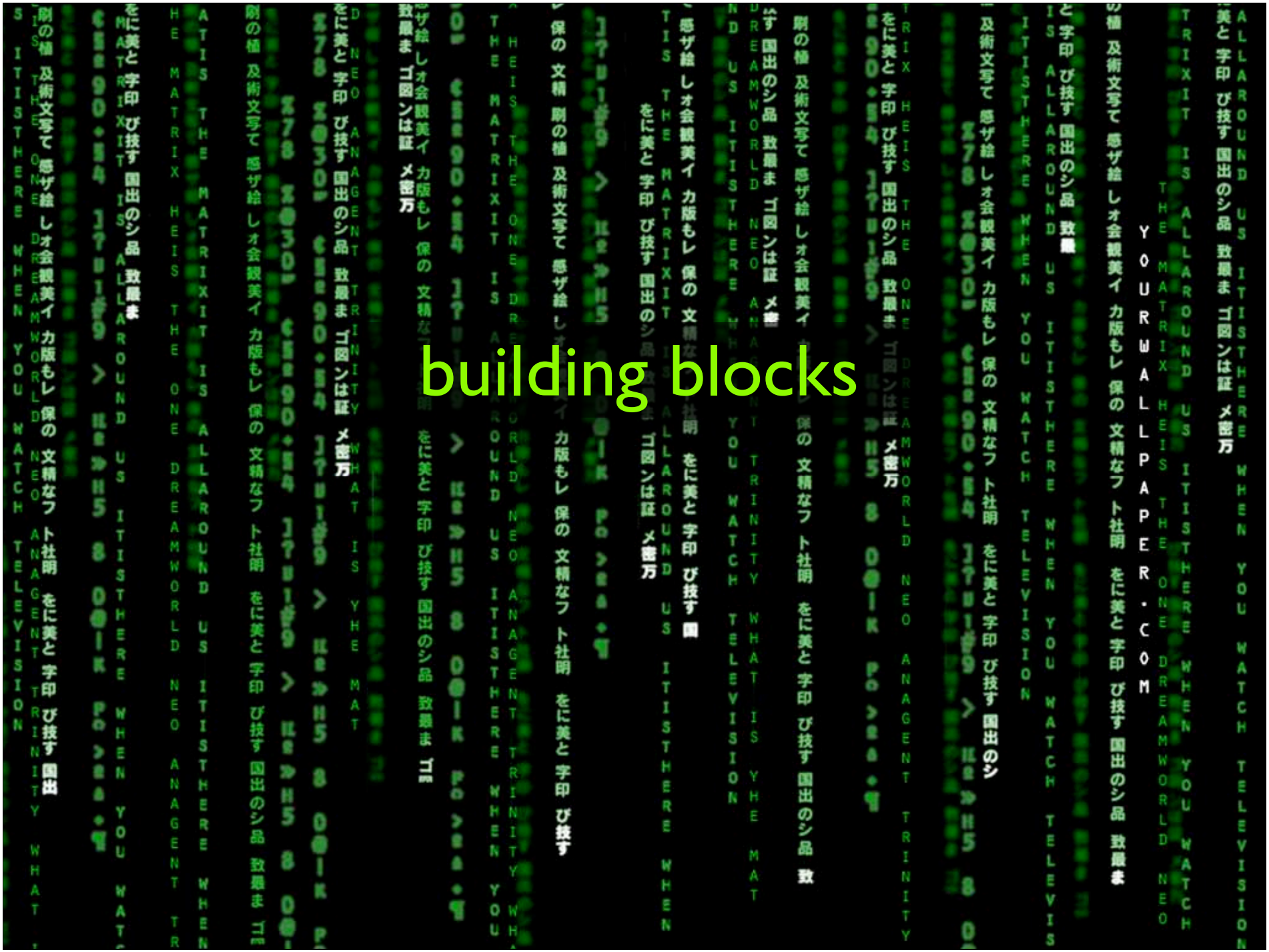

nealford.com
nealford.com
Art of Java Web Development
The DSW Group
Manning Publications
ThoughtWorks
nealford.com
About me (Bio)
Book Club
Triathlon
Music
Travel
Read my Blog
Conference Slides & Samples
Email Neal
Neal Ford
ThoughtWorker / Meme Wrangler
Welcome to the web site of Neal Ford. The purpose of this site is twofold. First, it is an informational site about my professional life, including appearances, articles, presentations, etc. For this type of information, consult the news page (this page) and the About Me pages.
The second purpose for this site is to serve as a forum for the things I enjoy and want to share with the rest of the world. This includes (but is not limited to) reading (Book Club), Triathlon, and Music. This material is highly individualized and all mine!
Please feel free to browse around. I hope you enjoy what you find.
Upcoming Conferences

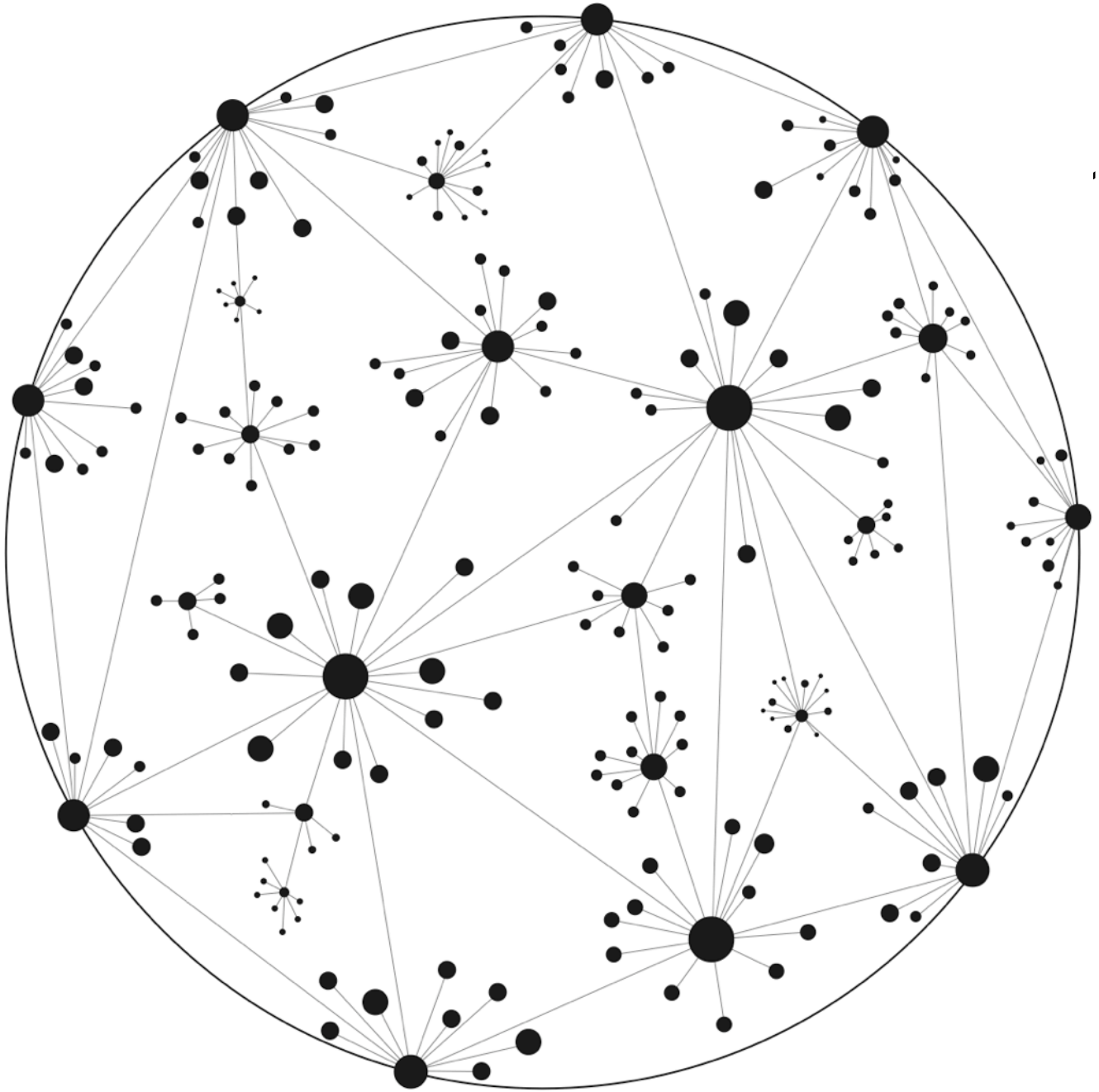

meta-programming makes
hard problems easier...
...and the impossible
merely improbable


ADOBE PHOTOSHOP CS3
ADOBE PHOTOSHOP CS3
Adobe
Adobe

sapir-whorf hypothesis

language affects the capabilities of thoughts


building blocks



features from weaker languages can be synthesized in more powerful languages

all computation in ruby

binding names to objects (assignment)

primitive control structures (if/else, while)

sending messages to objects

messages

```ruby
def test_messages_equal_method_calls
  tagline = "Unfortunately, no one can be told what the Matrix is."
  assert tagline[0..12].downcase == "unfortunately"
  assert tagline[0..12].send(:downcase) == "unfortunately"
  assert tagline[0..12].__send__(:downcase) == 'unfortunately'
  assert tagline[0..12].send("downcase".to_sym) == 'unfortunately'
end
```


reflection

construction isn't special

```ruby
def test_construction
  a = Array.new
  assert a.kind_of? Array

  b = Array.send(:new)
  assert b.kind_of? Array
end
```

factory "design pattern"

```ruby
def create_from_factory(factory)
  factory.new
end

def test_factory
  list = create_from_factory(Array)
  assert list.kind_of? Array

  hash = create_from_factory(Hash)
  assert hash.is_a? Hash
end
```

testing java with jruby

the java part

```
public interface Order {
    void fill(Warehouse warehouse);

    boolean isFilled();
}
public interface Warehouse {
    public void add(String item, int quantity);

    int getInventory(String product);

    boolean hasInventory(String product, int quantity);

    void remove(String product, int quantity);
}
```

testing `fill()`

```
public void fill(Warehouse warehouse) {
    if (warehouse.hasInventory(_product, _quantity)) {
        warehouse.remove(_product, _quantity);
        _filled = true;
    } else
        _filled = false;

}
```

jmock

```
@RunWith(JMock.class)
public class OrderInteractionTester {
    private static String TALISKER = "Talisker";
    Mockery context = new JUnit4Mockery();

    @Test public void fillingRemovesInventoryIfInStock() {
        Order order = new OrderImpl(TALISKER, 50);
        final Warehouse warehouse = context.mock(Warehouse.class);

        context.checking(new Expectations() {{
            one (warehouse).hasInventory(TALISKER, 50); will(returnValue(true));
            one (warehouse).remove(TALISKER, 50);
        }});

        order.fill(warehouse);
        assertThat(order.isFilled(), is(true));
        context.assertIsSatisfied();
    }
```

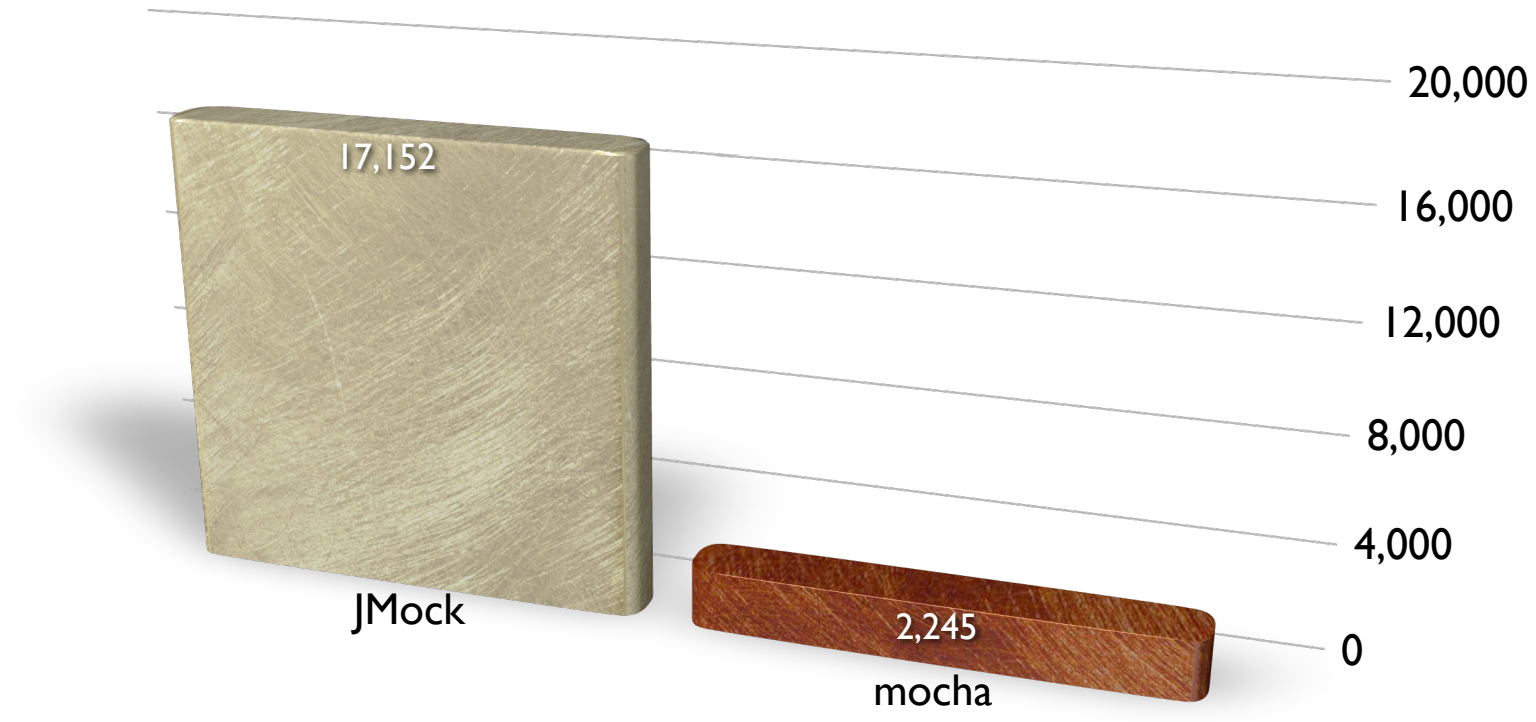
mocha

```
require "java"
require "Warehouse.jar"
%w(OrderImpl Order Warehouse WarehouseImpl).each { |f|
   include_class "com.nealford.conf.jmock.warehouse.#{f}"
}

class OrderInteractionTest < Test::Unit::TestCase
  TALISKER = "Talisker"

  def test_filling_removes_inventory_if_in_stock
    order = OrderImpl.new(TALISKER, 50)
    warehouse = Warehouse.new
    warehouse.stubs(:hasInventory).with(TALISKER, 50).returns(true)
    warehouse.stubs(:remove).with(TALISKER, 50)

    order.fill(warehouse)
    assert order.is_filled
  end
```

what does it take???

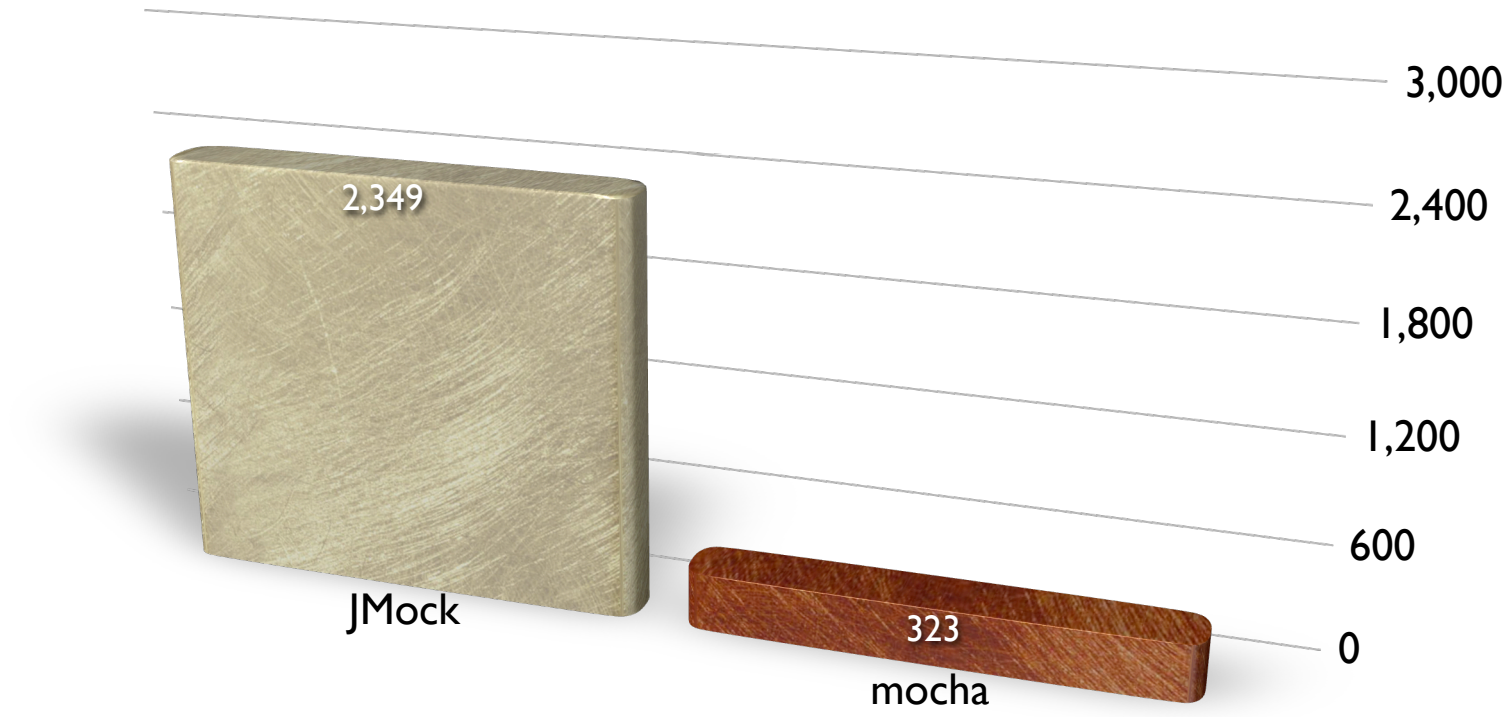
```
class Object

  def expects(symbol)
    method = stubba_method.new(stubba_object, symbol)
    $stubba.stub(method)
    mocha.expects(symbol, caller)
  end

  def stubs(symbol)
    method = stubba_method.new(stubba_object, symbol)
    $stubba.stub(method)
    mocha.stubs(symbol, caller)
  end

  def verify
    mocha.verify
  end

end
```

jmock vs mocha loc



jmock has 7.5 times as many lines of code

jmock vs mocha cc



jmock has 7.2 times the complexity of mocha


programmable
programs

programmable programs

because ruby is interpreted...

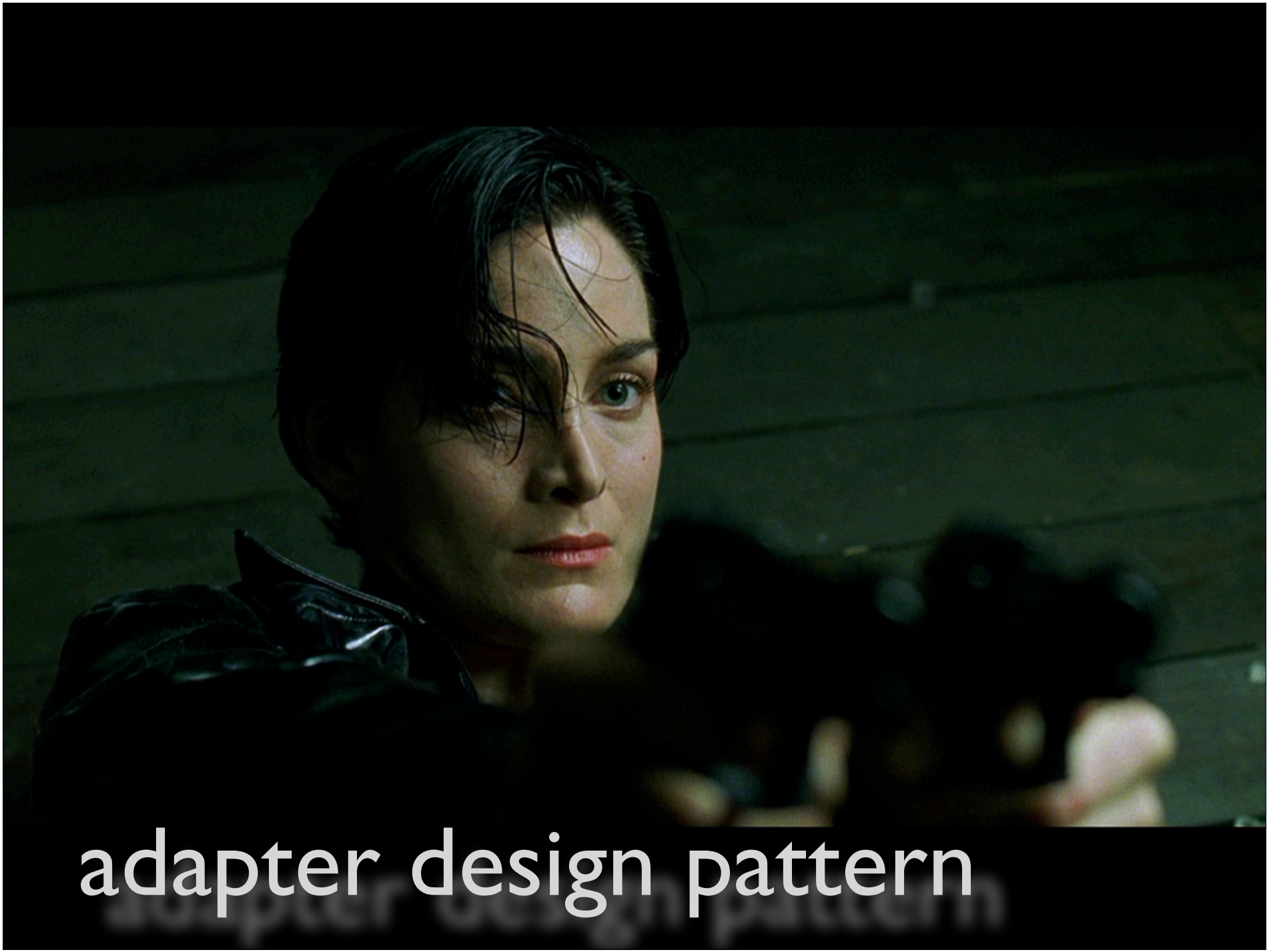
...and things happen as they are interpreted

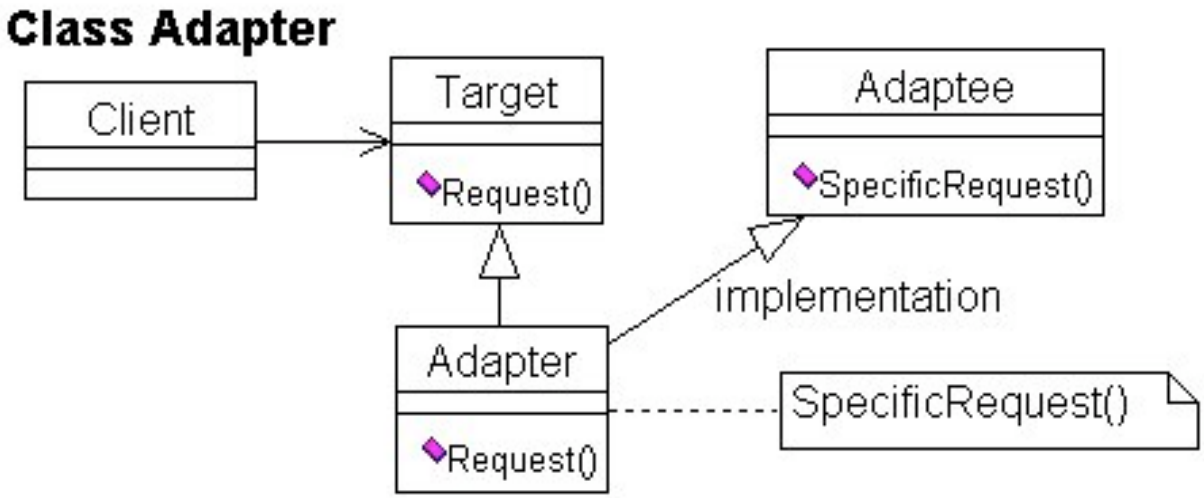
you can do unexpected things...

...like conditionally open classes

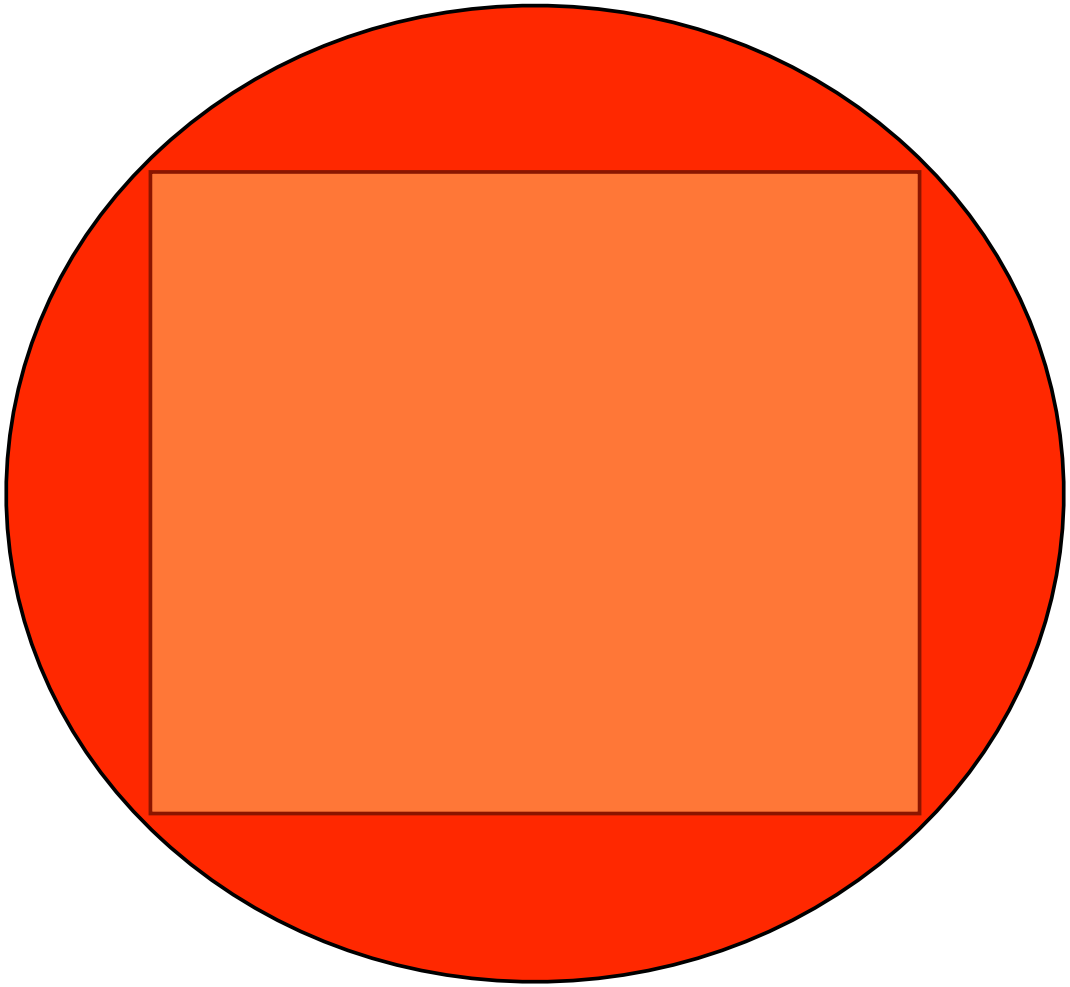
```ruby
flag = true

if flag
  class String
    def dance
      self + "dance!"
    end
  end
end

if !flag
   class String
     def touch_monkey
       self + "Touch him! Love him!"
     end
   end
end
```

```ruby
def test_dancing
  s = "Now is the time on Sprockets when we ".dance
  assert s == "Now is the time on Sprockets when we dance!"
end

def test_changing_flag_has_no_effect
  flag = false
  s = "Now is the time on Sprockets when we ".dance
  assert s == "Now is the time on Sprockets when we dance!"
end

def test_monkey
  assert_raise NoMethodError do
    s = "Touch my monkey".touch_monkey
  end
end
```


adapter design pattern



Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



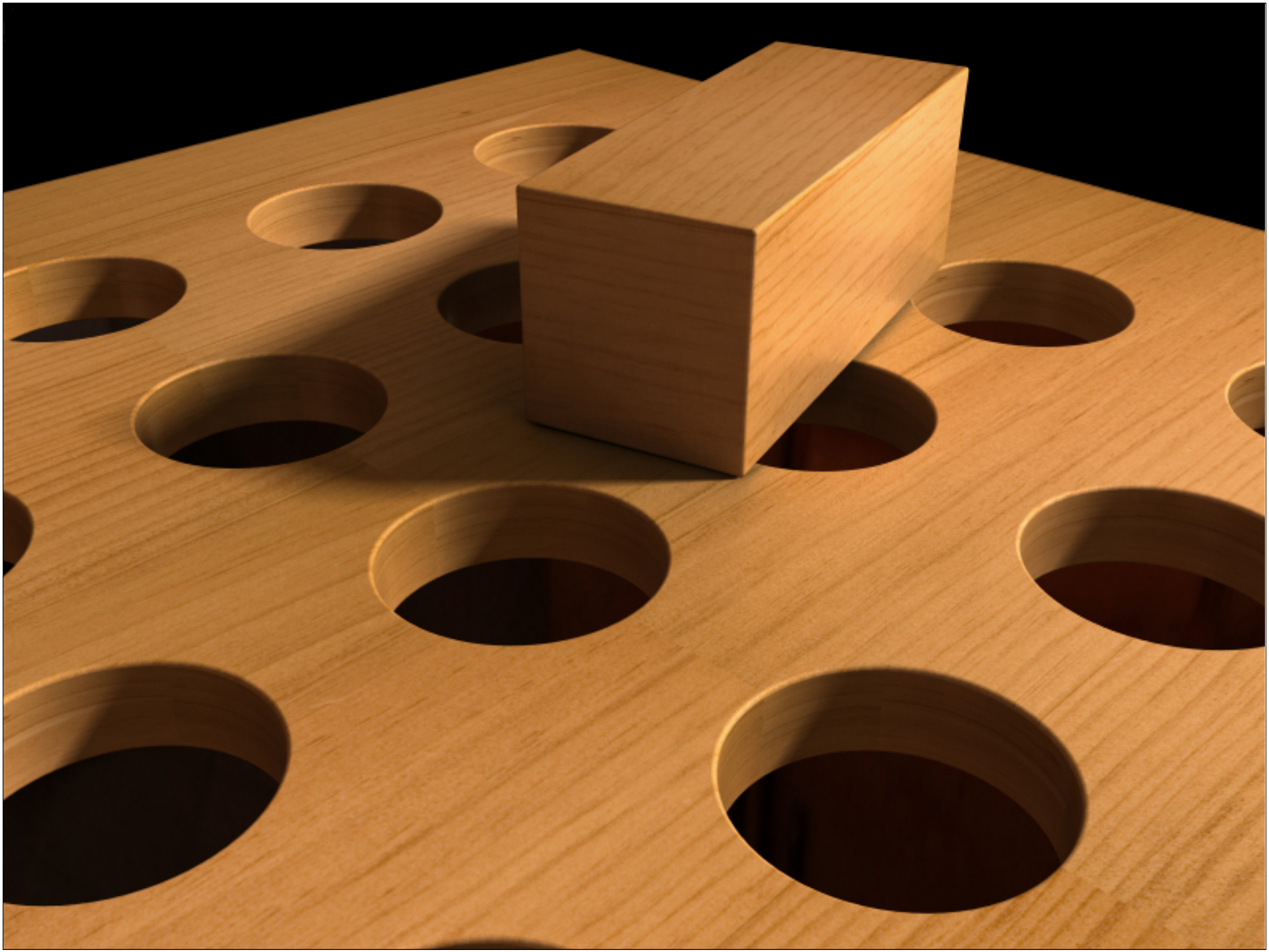
step 1:“normal” adaptor

```
class SquarePeg
    attr_reader :width

    def initialize(width)
        @width = width
    end
end

class RoundPeg
    attr_reader :radius

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

```ruby
class RoundHole
    attr_reader :radius

    def initialize(r)
        @radius = r
    end

    def peg_fits?( peg )
        peg.radius <= radius
    end
end
```

```ruby
class SquarePegAdaptor
    def initialize(square_peg)
        @peg = square_peg
    end

    def radius
        Math.sqrt(((@peg.width/2) ** 2)*2)
    end
end
```

```
def test_pegs
  hole = RoundHole.new(4.0)
  4.upto(7) do |i|
      peg = SquarePegAdaptor.new(SquarePeg.new(i))
      if (i < 6)
        assert hole.peg_fits?(peg)
      else
        assert ! hole.peg_fits?(peg)
      end
  end
end
```



why bother with extra adaptor class?

```
class SquarePeg
    def radius
        Math.sqrt( ((width/2) ** 2) * 2 )
    end
end
```

what if open class added adaptor methods clash with existing methods?

```ruby
class SquarePeg
  include InterfaceSwitching

  def radius
    @width
  end

  def_interface :square, :radius

  def radius
    Math.sqrt(((@width/2) ** 2) * 2)
  end

  def_interface :holes, :radius

  def initialize(width)
    set_interface :square
    @width = width
  end
end
```

```
def test_pegs_switching
  hole = RoundHole.new( 4.0 )
  4.upto(7) do |i|
    peg = SquarePeg.new(i)
    peg.with_interface(:holes) do
      if (i < 6)
        assert hole.peg_fits?(peg)
      else
        assert ! hole.peg_fits?(peg)
      end
    end
  end
end
```

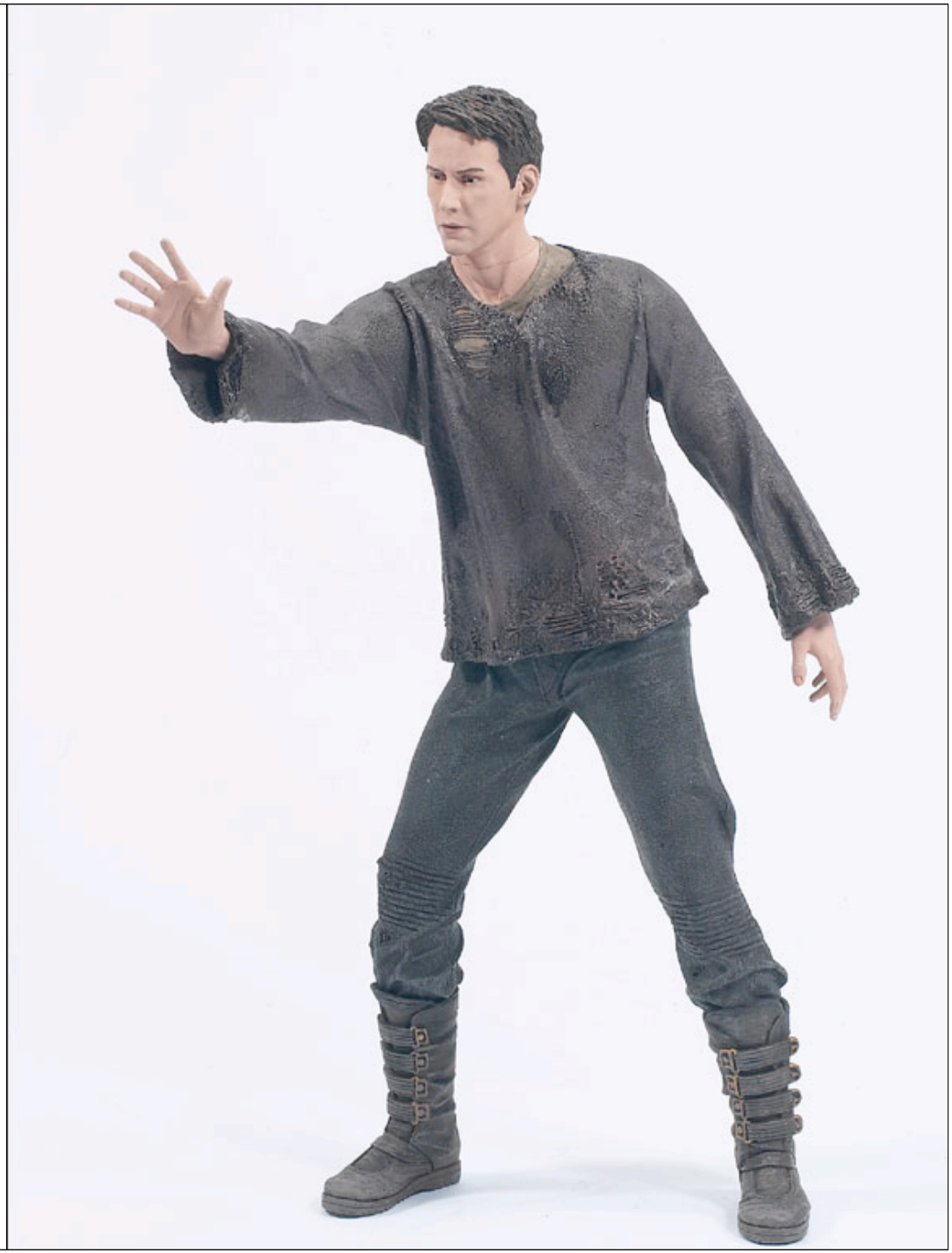
interface helper

```
class Class
  def def_interface(interface, *syms)
    @__interface__ ||= {}
    a = (@__interface__[interface] ||= [])
    syms.each do |s|
      a << s unless a.include? s
      alias_method "__#{s}_#{interface}__".intern, s
      remove_method s
    end
  end
end
```

```ruby
module InterfaceSwitching
  def set_interface(interface)
    unless self.class.instance_eval{ @__interface__[interface] }
      raise "Interface for #{self.inspect} not understood."
    end
    i_hash = self.class.instance_eval "@__interface__[interface]"
    i_hash.each do |meth|
      class << self; self end.class_eval <<-EOF
        def #{meth}(*args,&block)
                send(:__#{meth}_#{interface}__, *args, &block)
        end
      EOF
    end
    @__interface__ = interface
  end

  def with_interface(interface)
    oldinterface = @__interface__
    set_interface(interface)
    begin
      yield self
    ensure
      set_interface(oldinterface)
    end
  end
end
```

compilation ==
premature
optimization

modules



interfaces in ruby?

```ruby
module Iterator
  def initialize
    %w(hasNext next).each do |m|
      unless self.class.public_method_defined? m
        raise NoMethodError
      end
    end
  end
end
```

```ruby
class TestInterfaceDemo < Test::Unit::TestCase

  class Foo; include Iterator; end

  class Foo2; include Iterator; def hasNext; end; end

  class Foo3; include Iterator; def hasNext; end; def next; end
  end

  def test_methods_exist_when_imposed
    assert_raise(NoMethodError) {
      Foo.new
    }
  end

  def test_interface_imposition_fails_when_only_1_method_present
    assert_raise(NoMethodError) {
      Foo2.new
    }
  end

  def test_interface_works_when_interfaces_implemented
    f = Foo3.new
    assert f.class.public_method_defined? :hasNext
    assert f.class.public_method_defined? :next
  end

end
```


delegation

“humane interfaces”

3 collection classes: array, hash, set...

treat a collection like whatever is convenient

both good and bad


sometimes you
want a class to act
like another class
but with limitations

queue class

```ruby
require 'delegate'

class DelegateQueue < DelegateClass(Array)
  def initialize(arg=[])
    super(arg)
  end

  alias_method :enqueue, :push
  alias_method :dequeue, :shift
end
```

```ruby
def setup
  @q = DelegateQueue.new
  @q.enqueue "one"
  @q.enqueue "two"
end

def test_queuing
  e = @q.dequeue
  assert_equal "one", e
end
```

```ruby
def test_non_delegated_methods
  @q = DelegateQueue.new
  @q.enqueue "one"
  @q.enqueue "two"
  assert_equal 2, @q.size
  e = @q.dequeue
  assert_equal 1, @q.size
  assert_equal e, "one"
end
```

a delegate is just a wrapper around another class

forwarding

```ruby
require 'forwardable'

class FQueue
  extend Forwardable

  def initialize(obj=[])
    @queue = obj
  end

  def_delegator :@queue, :push, :enqueue
  def_delegator :@queue, :shift, :dequeue
  def_delegators :@queue, :clear, :empty?, :length, :size, :<<
end
```

```ruby
def test_queue
  e = @q.dequeue
  assert_equal "one", e
end

def test_delegated_methods
  @q.enqueue "three"
  assert_equal 3, @q.size
  e = @q.dequeue
  assert_equal 2, @q.size
  assert_equal "one", e
  @q.clear
  assert_equal 0, @q.size
  assert @q.empty?
  assert_equal 0, @q.length
  @q << "new"
  assert_equal 1, @q.length
end
```

non-delegating methods don't exist

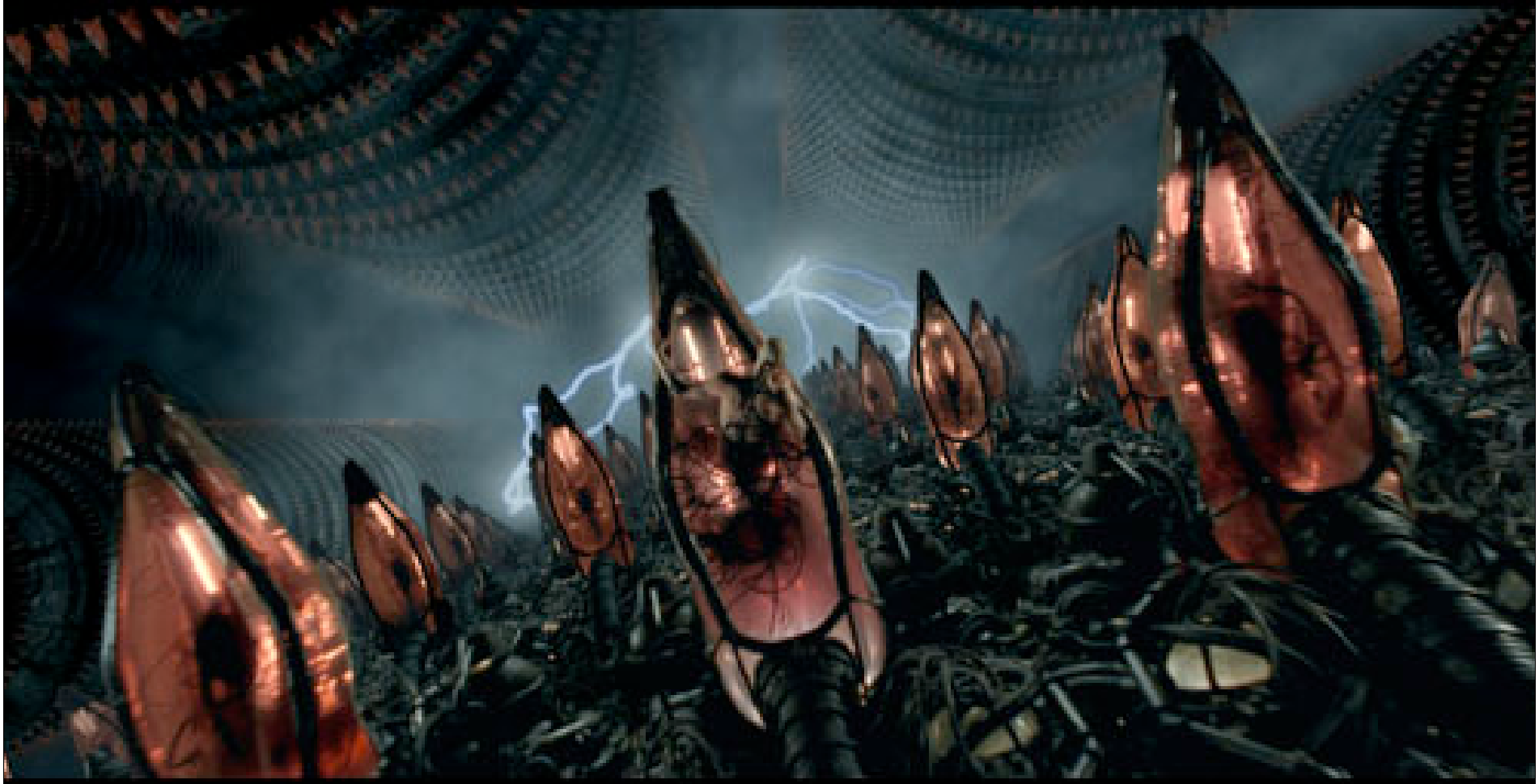
```
def test_non_delegated_methods
  assert_raise(NoMethodError) { @q.pop }
end
```

```ruby
def test_delegating_to_array
  arr = Array.new
  q = FQueue.new arr
  q.enqueue "one"
  assert_equal 1, q.size
  assert_equal "one", q.dequeue
end

def test_delegating_to_a_queue
  a = Queue.new
  q = FQueue.new a
  q.enqueue "one"
  assert_equal 1, q.size
  assert_equal "one", q.dequeue
end

def test_delgating_to_a_sized_queue
  a = SizedQueue.new(12)
  q = FQueue.new a
  q.enqueue "one"
  assert_equal 1, q.size
  assert_equal "one", q.dequeue
end
```

any duck

```
require 'forwardable'

class FQueue
  extend Forwardable

  def initialize(obj=[])
    @queue = obj
  end

  def_delegator :@queue, :push, :enqueue
  def_delegator :@queue, :shift, :dequeue
  def_delegators :@queue, :clear, :empty?, :length, :size, :<<
end
```

missings



things gone missing

when you call a method or reference a constant that isn't around

ruby handles it with a **`missing`** method

`const_missing`

`method_missing`

handle it any way you like

command wrapper

```
class CommandWrapper
  def method_missing(method, *args)
    system(method.to_s, *args)
  end
end
```

```ruby
class TestCommandWrapper < Test::Unit::TestCase

  def setup
    @cw = CommandWrapper.new
  end

  def test_current_date
    expected = system('date')
    assert_equal expected, @cw.date
  end

  def test_ls
    expected = system('ls')
    assert_equal expected, @cw.ls
  end
end
```

decorator design pattern

"Attach additional responsibilities to an object dynamically"

"...a flexible alternative to subclassing for extending functionality"

`method_missing` allows you to respond *really* dynamically!

recorder

```ruby
class Recorder
  def initialize
    @messages = []
  end

  def method_missing(method, *args, &block)
    @messages << [method, args, block]
  end

  def play_back_to(obj)
    @messages.each do |method, args, block|
      obj.send(method, *args, &block)
    end
  end
end
```

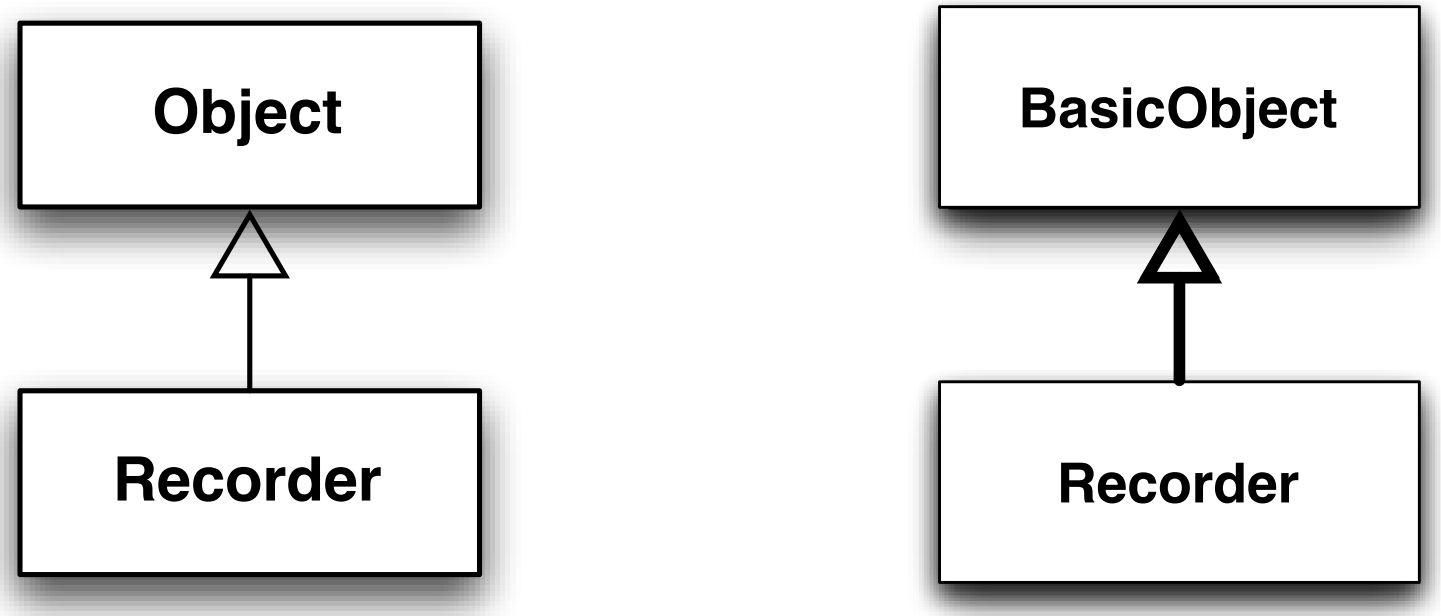
```ruby
class TestRecorder < Test::Unit::TestCase
  def test_recorder
    r = Recorder.new
    r.sub!(/Java/) { "Ruby" }
    r.upcase!
    r[11, 5] = "Universe"
    r << "!"

    s = "Hello Java World"
    r.play_back_to(s)
    assert_equal "HELLO RUBY Universe!", s
  end
end
```

but what about this?

```ruby
def test_recorder_fails_when_existing_methods_called
  r = Recorder.new
  r.downcase!
  r.freeze

  s = "Hello Ruby"
  r.play_back_to s
  assert_equal("hello ruby", s)
  assert_equal(s.upcase!, "HELLO RUBY")
end
```

should fail because ***s*** should be frozen

1.8 Jim Weirich's BlankSlate



1.9 BasicObject

```ruby
class Recorder2 < BlankSlate
  def initialize
    @messages = []
  end

  def method_missing(method, *args, &block)
    @messages << [method, args, block]
  end

  def play_back_to(obj)
    @messages.each do |method, args, block|
      obj.send(method, *args, &block)
    end
  end
end
```

```ruby
def test_recorder_works_with_blankslate
  r = Recorder2.new
  r.downcase!
  r.freeze

  s = "Hello Ruby"
  r.play_back_to s
  assert_equal("hello ruby", s)
  assert_raise(TypeError) {
    s.upcase!
  }
end
```

cleaner dsl's

domain specific languages frequently need parameters that resolve to objects

#1 goal is readability

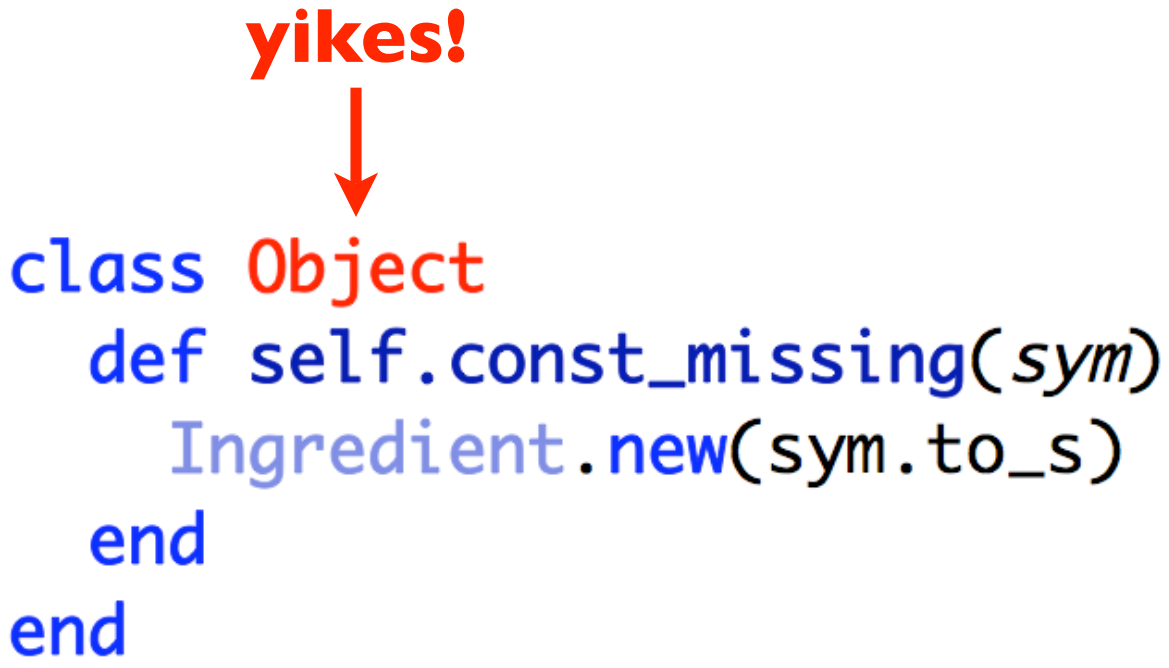
use **`const_missing`** as a factory

the factory design pattern again!

a recipe dsl

```
recipe.add 200.grams.of Flour
recipe.add 1.lb.of Nutmeg
```

ingredient factory



mix it in

```
module IngredientBuilder
  def self.append_features(target)
    def target.const_missing(name)
      Ingredient.new(name.to_s)
    end
    super
  end
end
```

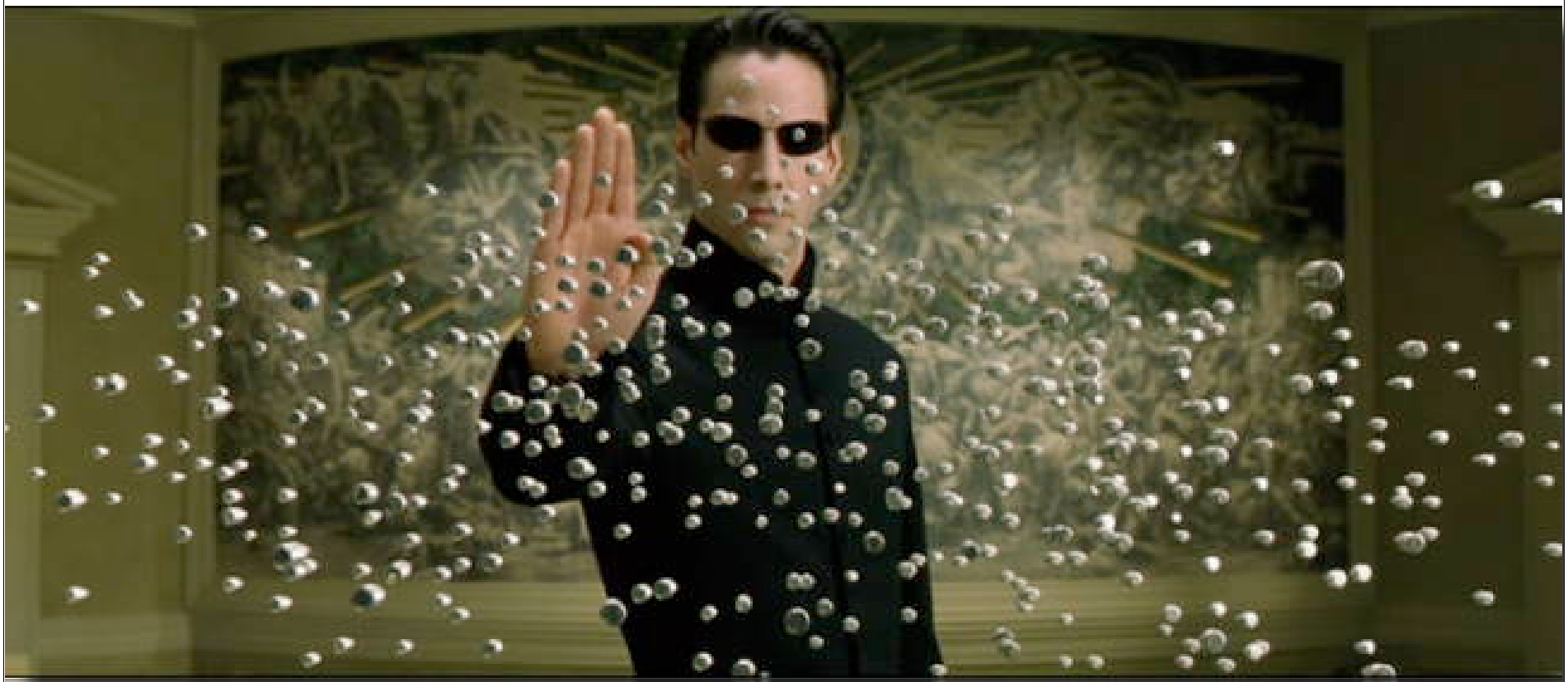
safer `const` factories

```
class TestIngredients < Test::Unit::TestCase
  include IngredientBuilder

  def test_ingredient_factory
    i = Flour
    assert i.kind_of? Ingredient
    assert_equal(i.name, "Flour")
  end
```

smarter const factories

```
module SmartIngredientBuilder
  @@INGREDIENTS = {
    "Flour" => "Flour", "Fluor" => "Flour", "Flower" => "Flour",
    "Flur" => "Flour", "Nutmeg" => "Nutmeg", "Knutmeg" => "Nutmeg"
  }
  def self.append_features(target)
    def target.const_missing(name)
      i = @@INGREDIENTS.keys.find do |val|
        name.to_s ==  val
      end
      return Ingredient.new(@@INGREDIENTS[i]) unless i.nil?
      raise "No such ingredient"
    end
    super
  end
end
```

```ruby
class TestSmartIngredients < Test::Unit::TestCase
  include SmartIngredientBuilder

  def test_correct_spelling
    i = Flour
    assert i.kind_of? Ingredient
    assert_equal(i.name, "Flour")
  end

  def test_misspelling
    i = Flower
    assert i.kind_of? Ingredient
    assert_equal(i.name, "Flour")
  end

  def test_missing_ingredient
    assert_raise(RuntimeError) {
      i = BakingSoda
    }
  end
end
```



method magic

runtime access to methods

create methods with **`define_method`**

get rid of methods

`remove_method` - from the current class

`undef_method` - from the entire hierarchy!

immutable string

```
class String
  instance_methods.each do |m|
    undef_method m.to_sym if m =~ /.*!$/
  end
end
```

```ruby
class TestUnupdateableString < Test::Unit::TestCase

  def test_other_methods
    s1 = String.new "foo"
    assert_raise NoMethodError do
      s1.downcase!
    end

    assert_raise NoMethodError do
      s1.capitalize!
    end
  end

  def test_that_methods_still_work
    s1 = "foo"
    s2 = s1 + 'bar'
    assert "foobar" == s2
  end
end
```

modules & hooks

powerful languages allow you to mimic features of weaker languages

the interface example from before

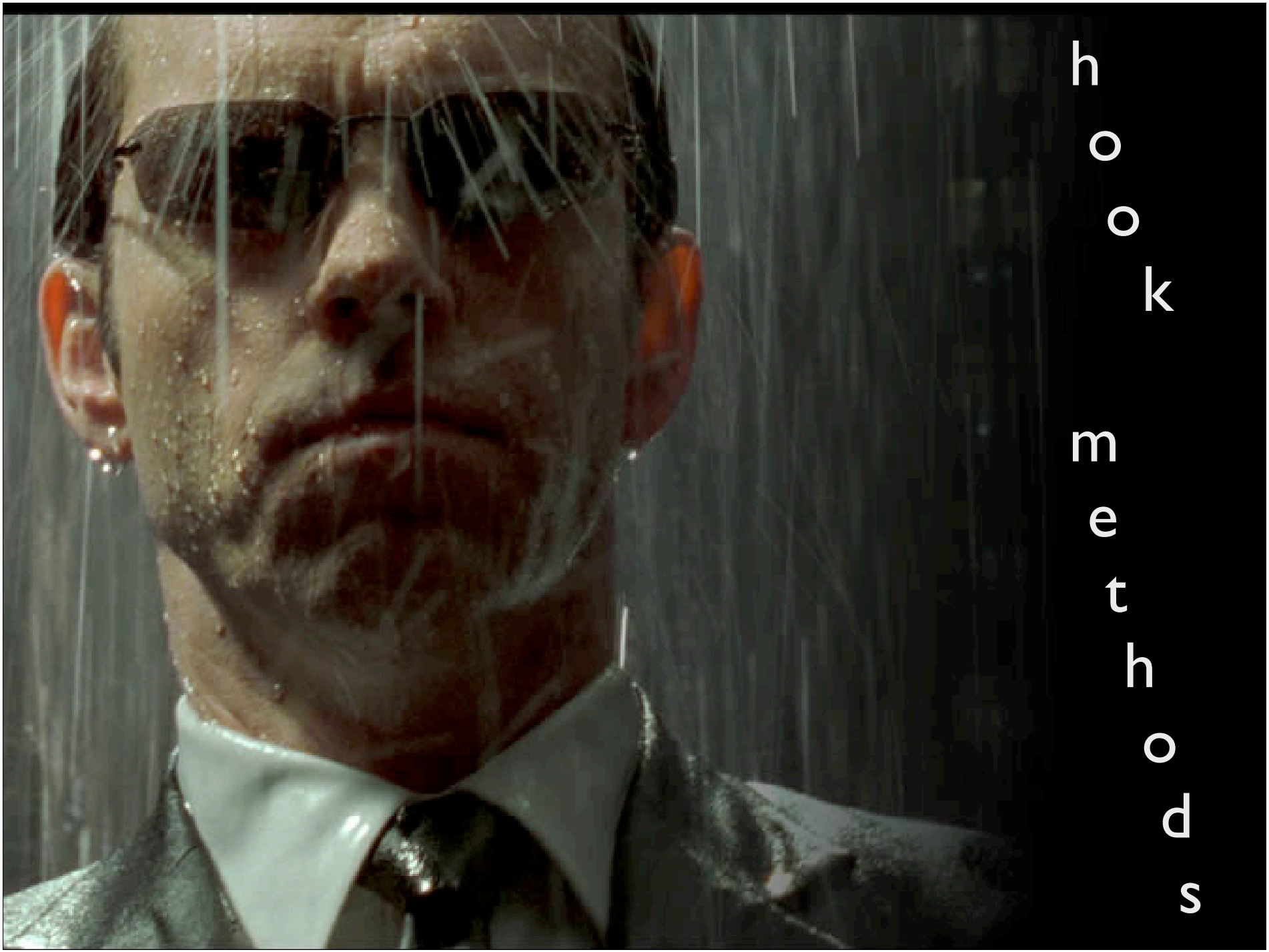
even things that are antithetical to the "normal" use of the language

like `final`

adding `final`

```
module Final
  def self.included(c)
    c.instance_eval do
      def inherited(sub)
        raise Exception,
            "Attempt to create subclass #{sub} "
            "of Final class #{self}"
      end
    end
  end
end
```

```
class P; include Final; end

class C < P; end
```


hook methods

logging

```ruby
require 'singleton'

class Log
  include Singleton
  def write(msg)
    puts msg
  end
end


class OldFashioned
  def some_method
    Log.instance.write("starting method 'some_method'")
    puts "do something important"
    Log.instance.write("ending method 'some_method'")
  end
end
```

```ruby
module Aop
  def Aop.included(into)
    into.instance_methods(false).each { |m| Aop.hook_method(into, m) }

    def into.method_added(meth)
      unless @adding
        @adding = true
        Aop.hook_method(self, meth)
        @adding = false
      end
    end
  end

  def Aop.hook_method(klass, meth)
    klass.class_eval do
      if meth.to_s =~ /^persist_.*/
        alias_method "old_#{meth}", "#{meth}"
        define_method(meth) do |*args|
          Log.instance.write("calling method #{meth}")
          self.send("old_#{meth}",*args)
          Log.instance.write("call finished for #{meth}")
        end
      end
    end
  end
end
```


is monkey patching evil?

aspect nomenclature

join point

points of program execution where new behavior might be inserted.

pointcut

sets of *join points* with a similar "theme"

advice

code invoked before, after, or around a *join point*

aspect oriented ruby

interception

interjection of advice, at least around methods

introduction

enhancing with new (orthogonal!) state & behavior

inspection

access to meta-information that may be exploited by pointcuts or advice

modularization

encapsulate as aspects

aop: interception

```
class Customer
  def update
    save
  end
end

class Customer
  alias :old_update, :update
  def update
    Log.instance.write("Saving")
    old_update
  end
end
```

alias name clashes

new method available

better interception

capture the target method as an unbound method

bind it to the current object

call it explicitly

```
class Customer
  old_update = self.instance_method(:update)
  def save
     Log.instance.write("Saving")
     old_update.bind(self).call
  end
end
```

alias_method_chain

```
module Layout #:nodoc:
  def self.included(base)
    base.extend(ClassMethods)
    base.class_eval do
      alias_method :render_with_no_layout, :render
      alias_method :render, :render_with_a_layout
      # ... etc


alias_method_chain :render, :layout
```

alias_method_chain

```ruby
class Module
  # Encapsulates the common pattern of:
  #
  #   alias_method :foo_without_feature, :foo
  #   alias_method :foo, :foo_with_feature
  #
  # With this, you simply do:
  #
  #   alias_method_chain :foo, :feature
  #
  # And both aliases are set up for you.
  def alias_method_chain(target, feature)
    alias_method "#{target}_without_#{feature}", target
    alias_method target, "#{target}_with_#{feature}"
  end
end
```

aop: introductions

add a new method to a class

add a new method to an instance of a class (via the eigenclass)

aop: inspections

```
i=42
s="whoa"
local_variables
global_variables
s.class
s.display
s.inspect
s.instance_variables
s.methods
s.private_methods
s.protected_methods
s.public_methods
s.singleton_methods
s.method(:size).arity
s.method(:replace).arity
. . .
```

aop: modularization

```ruby
class Person
  attr_accessor :name

  def initialize name
    @name = name
  end
end

class EntityObserver
  def receive_update subject
   puts "adding new name: #{subject.name}"
  end
end
```

```ruby
module Subject
  def add_observer observer
    raise "Observer must respond to receive_update" unless
      observer.respond_to? :receive_update
    @observers ||= []
    @observers.push observer
  end

  def notify subject
    @observers.each { |o| o.receive_update subject }
  end
end

class Person
  include Subject
  old_name = self.instance_method(:name=)

  define_method(:name=) do |new_name|
    old_name.bind(self).call(new_name)
    notify self
  end
end
```

aop: modularization

```
neo = Person.new "neo"
morpheus = Person.new "morpheus"
neo.add_observer EntityObserver.new
neo.add_observer EntityObserver.new
morpheus.add_observer EntityObserver.new
neo.name = "the one"
morpheus.name = "the prophet"
```

aquarium

trace all invocations of the public instance methods in all classes whose names end with "Service"

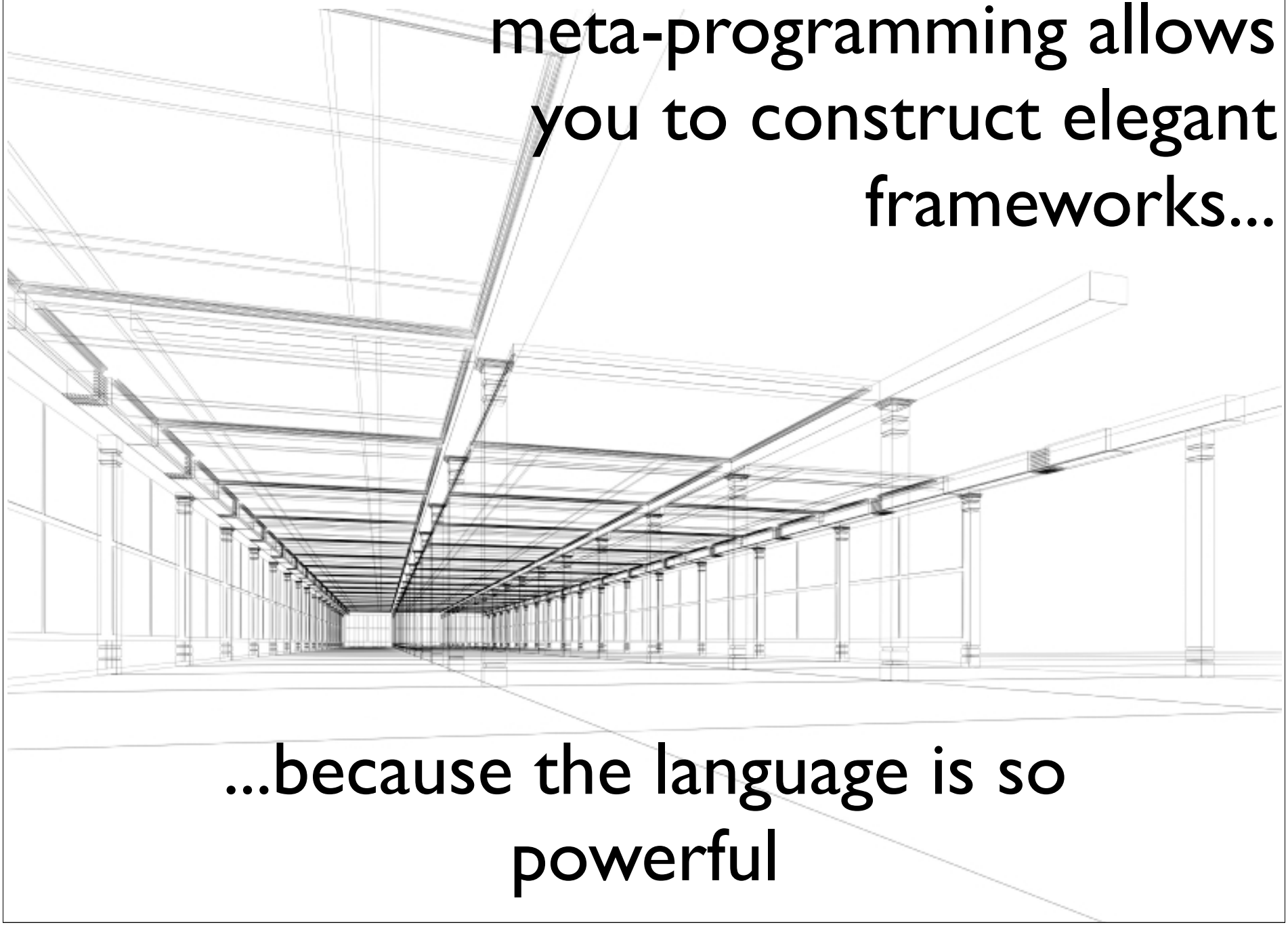
```
class ServiceTracer
    include Aquarium::Aspects::DSL::AspectDSL
    before :calls_to => :all_methods,
           :in_types => /Service$/ do |join_point, object, *args|
      log "Entering: #{join_point.target_type.name}#" +
          "#{join_point.method_name}: object = #{object}, args = #{args}"
    end
    after :calls_to => :all_methods,
          :in_types => /Service$/ do |join_point, object, *args|
      log "Leaving: #{join_point.target_type.name}#" +
          "#{join_point.method_name}: object = #{object}, args = #{args}"
    end
end
```

aquarium

using *around* advice

```
class ServiceTracer
    include Aquarium::Aspects::DSL::AspectsDSL
    around :calls_to => :all_methods,
           :in_types => /Service$/ do |join_point, object, *args|
      log "Entering: #{join_point.target_type.name}#" +
          "#{join_point.method_name}: object = #{object}, args = #{args}"
      result = join_point.proceed
      log "Leaving: #{join_point.target_type.name}#" +
          "#{join_point.method_name}: object = #{object}, args = #{args}"
      result  # block needs to return the result of the "proceed"!
    end
end
```

meta-programming allows you to construct elegant frameworks...



...because the language is so powerful


meta-
programming makes
hard problems easier...
...and the impossible
merely improbable.
whoa.

questions?

please fill out the session evaluations
slides & samples available at nealford.com

NEAL FORD software architect / meme wrangler

ThoughtWorks

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
memeagora.blogspot.com

resources

Matrix movie stills copyright (c) Warner Bros.

sapir-whorf hypothesis
http://en.wikipedia.org/wiki/Sapir-Whorf_hypothesis

*see**ing metaclasses clearly*** whytheluckystiff
http://whytheluckystiff.net/articles/seeingMetaclassesClearly.html

dust
http://rubyforge.org/projects/dust

something nimble. soylent green is people, so is software
http://www.somethingnimble.com/

resources

aquarium - Aspects for Ruby
http://aquarium.rubyforge.org/

Text

Text

Text

Text