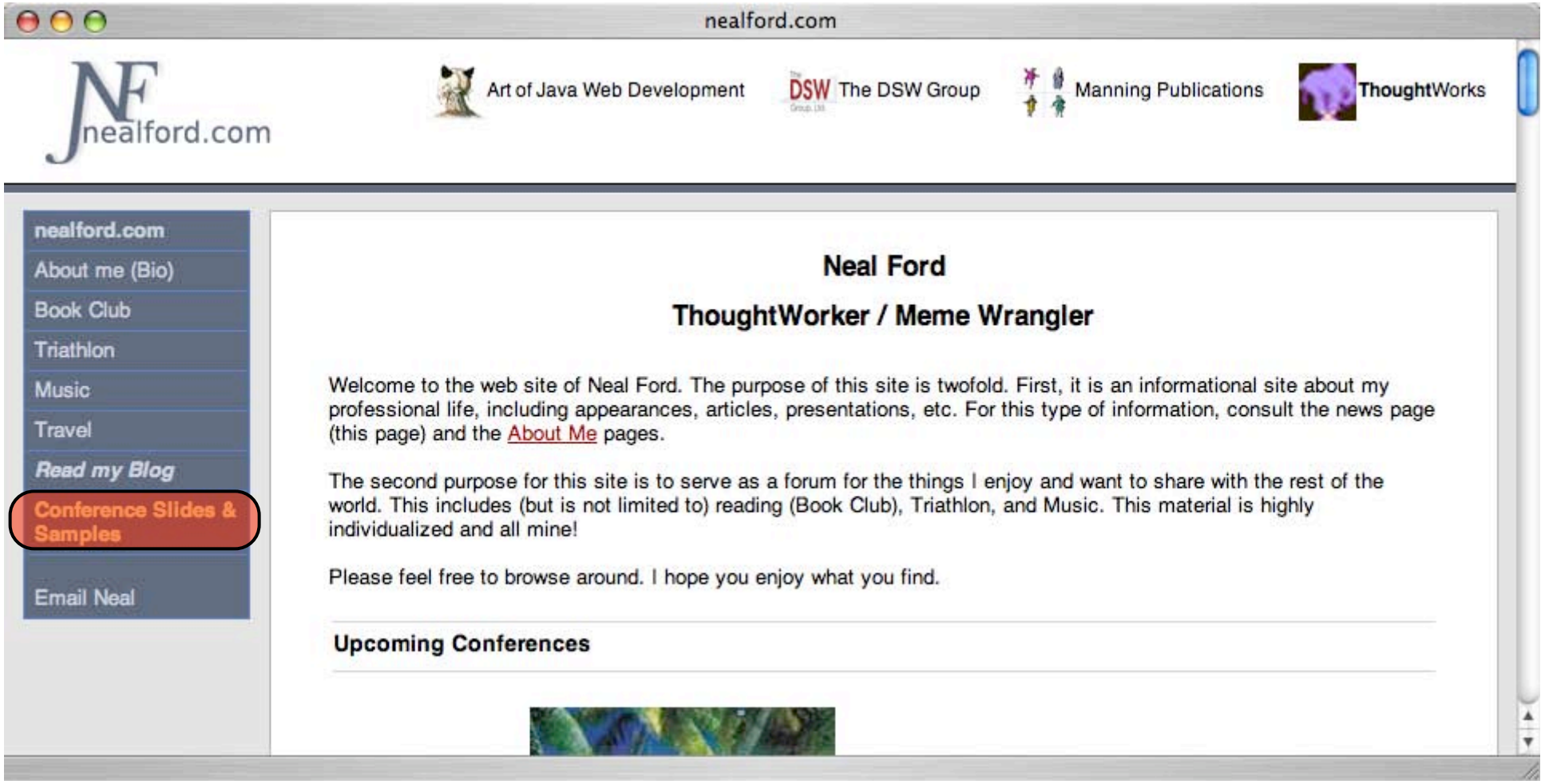
“design patterns” in dynamic languages

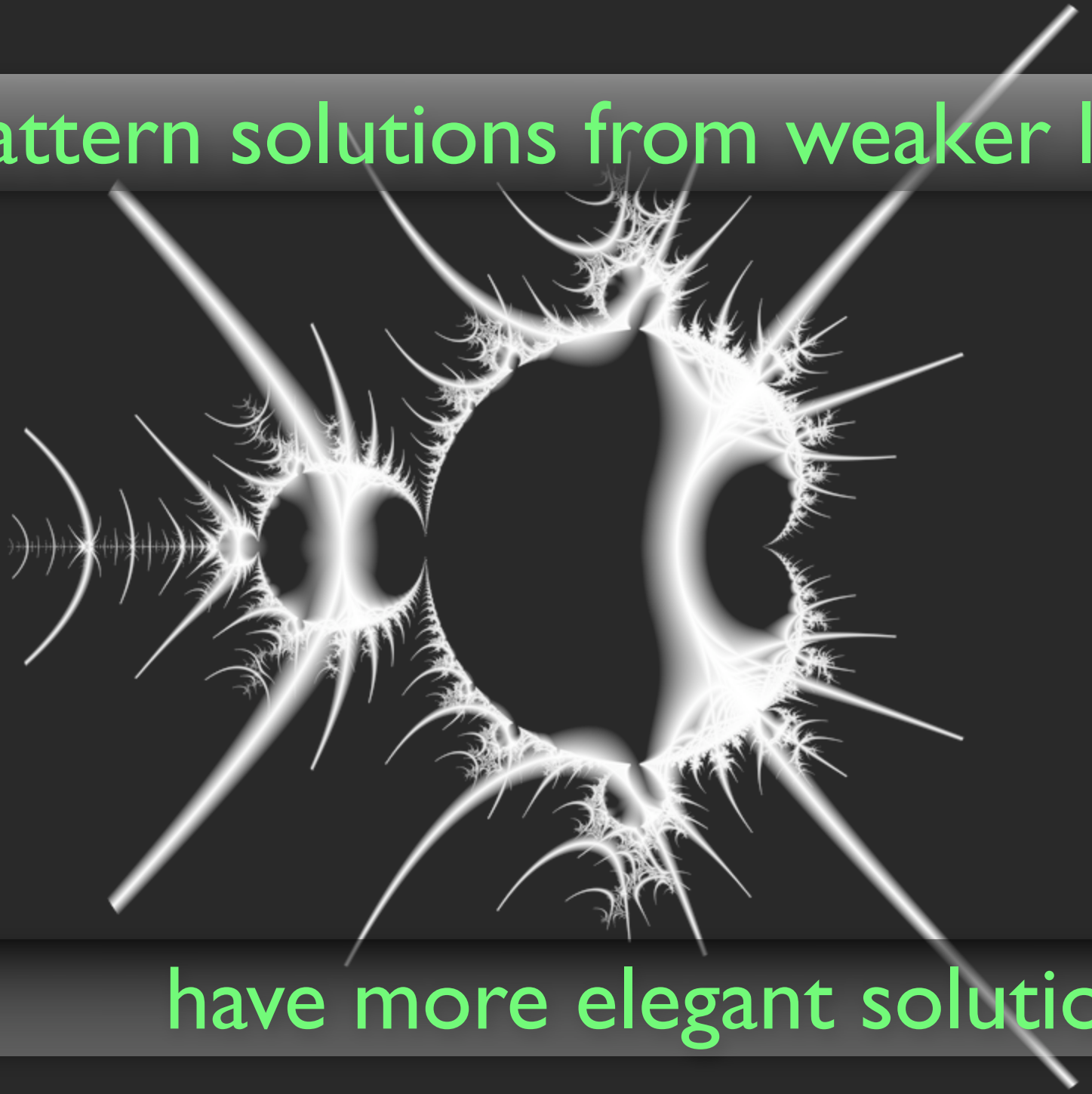
NEAL FORD software architect / meme wrangler

ThoughtWorks

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

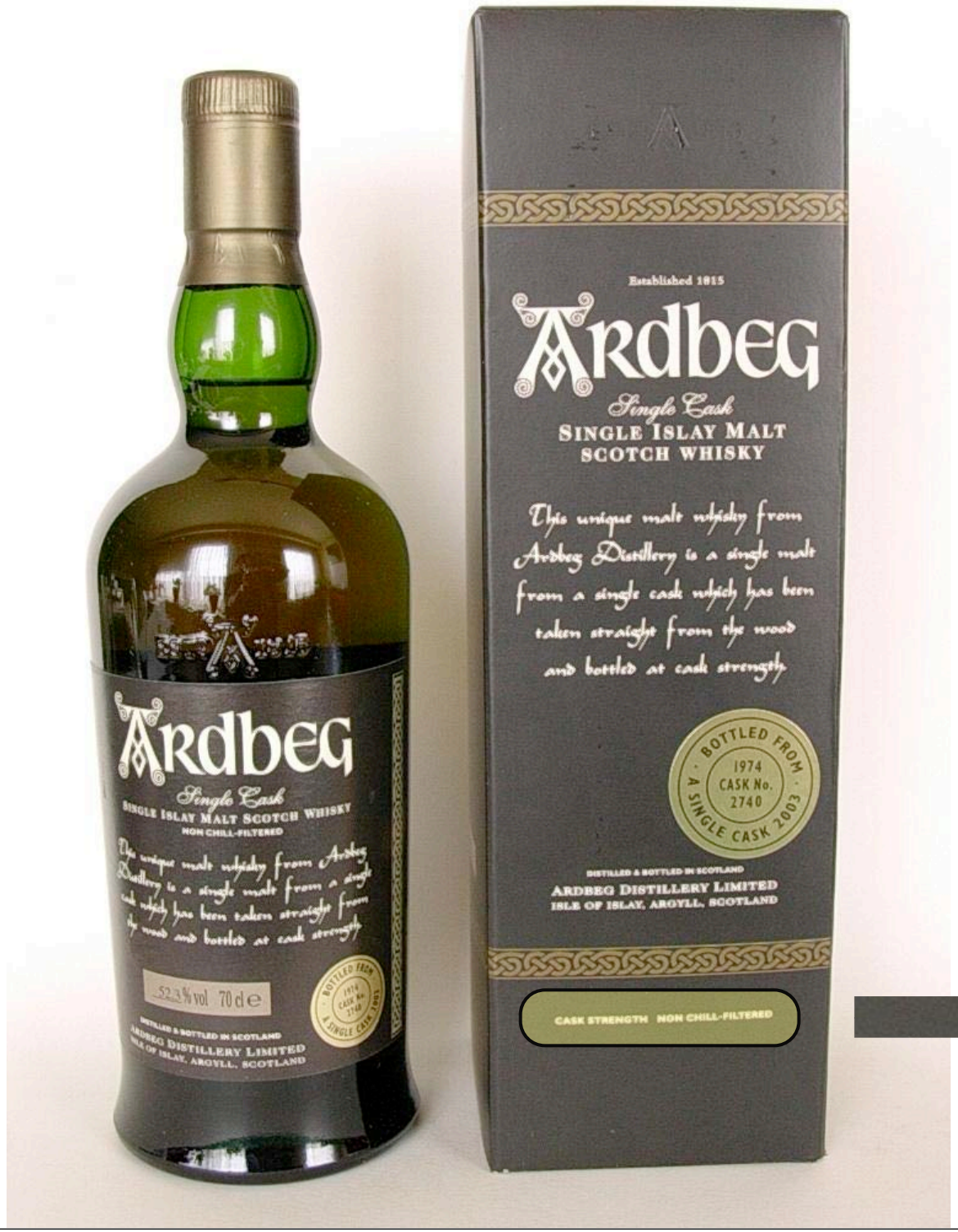

nealford.com
nealford.com
Art of Java Web Development
The DSW Group
Manning Publications
ThoughtWorks
nealford.com
About me (Bio)
Book Club
Triathlon
Music
Travel
Read my Blog
Conference Slides & Samples
Email Neal
Neal Ford
ThoughtWorker / Meme Wrangler
Welcome to the web site of Neal Ford. The purpose 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

pattern solutions from weaker languages

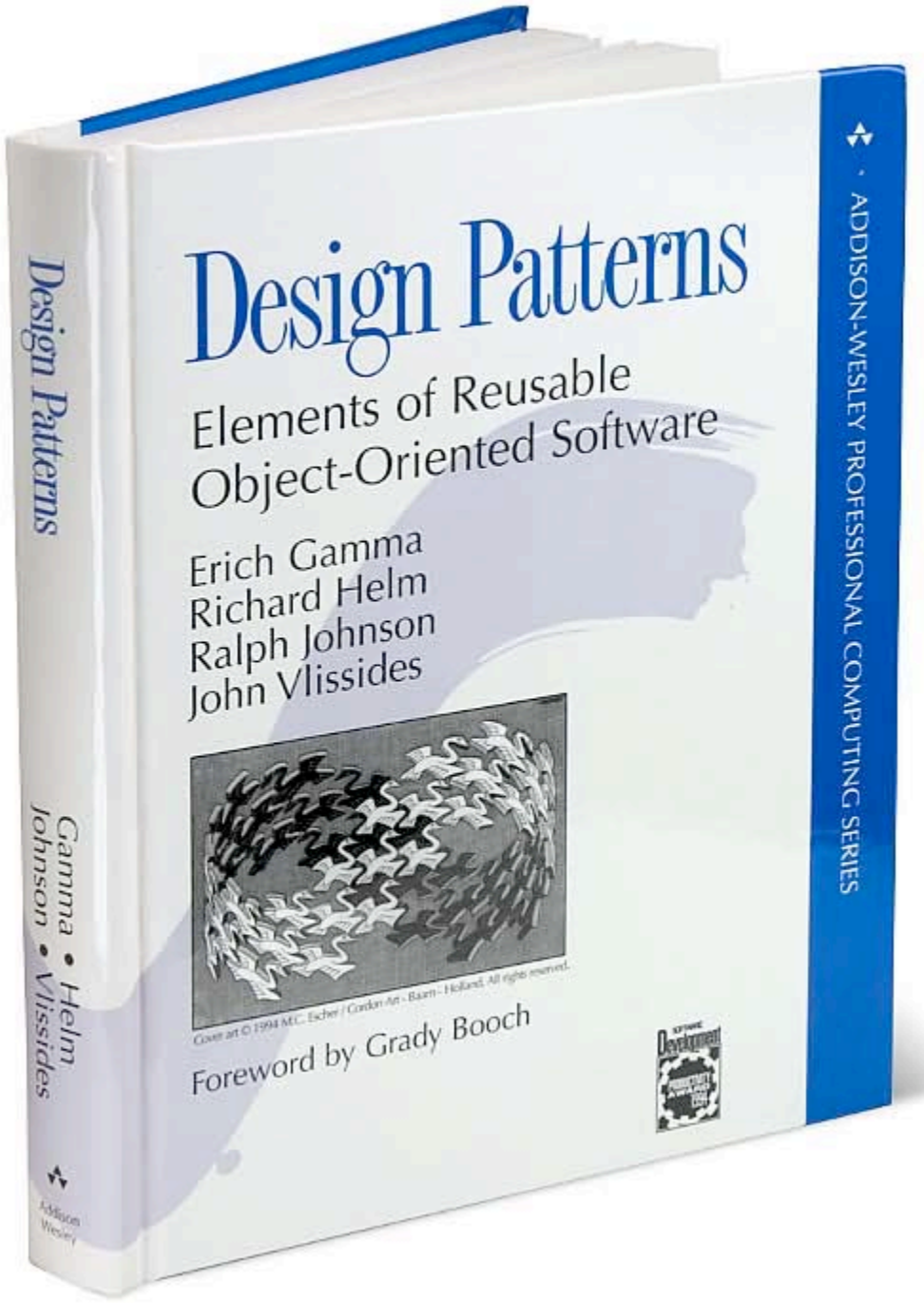


have more elegant solutions

blended whisky is made from several single malts



cask strength



Making C++ Suck Less

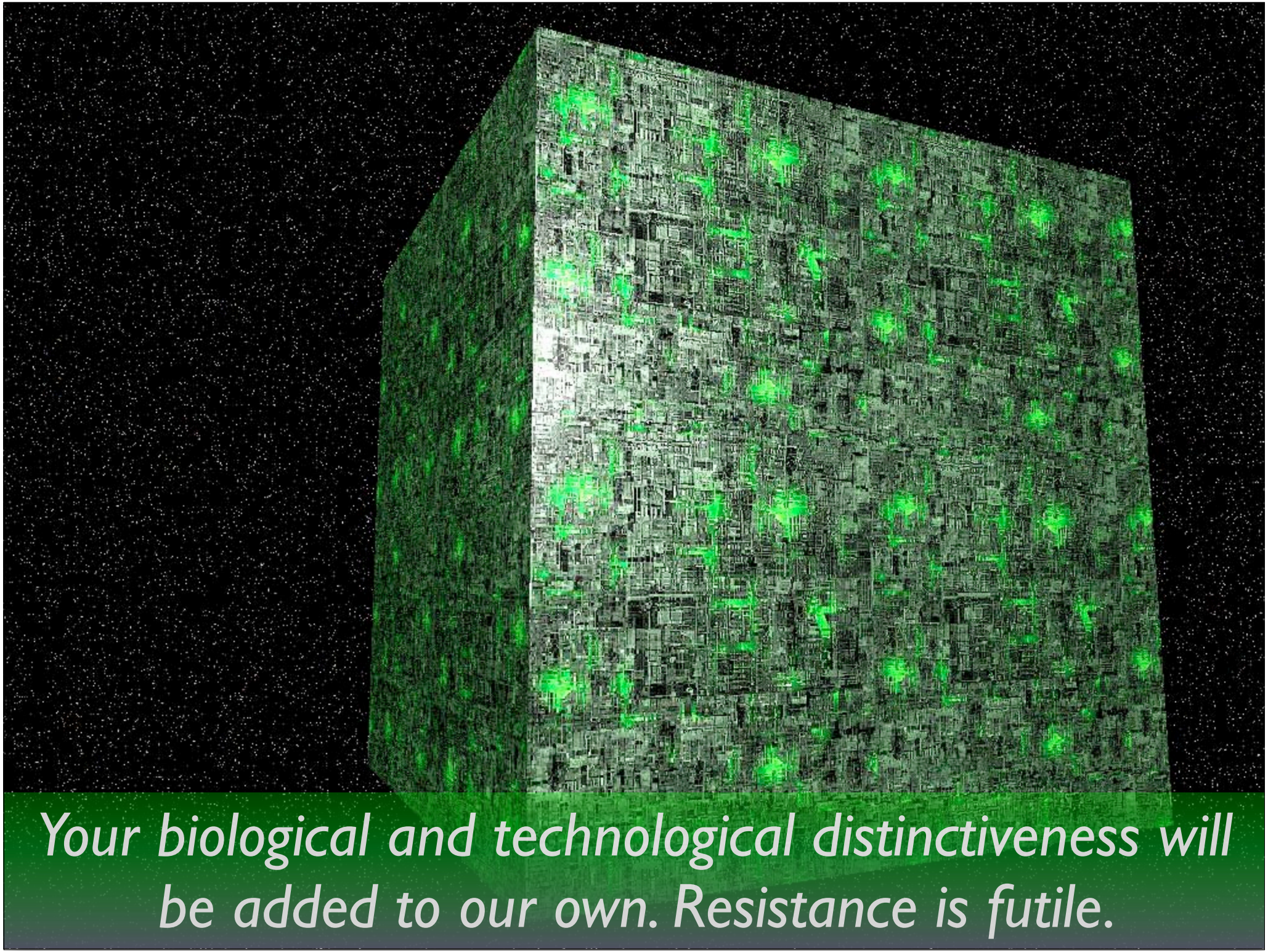
patterns define common problems

dynamic languages give you better tools to solve those problems

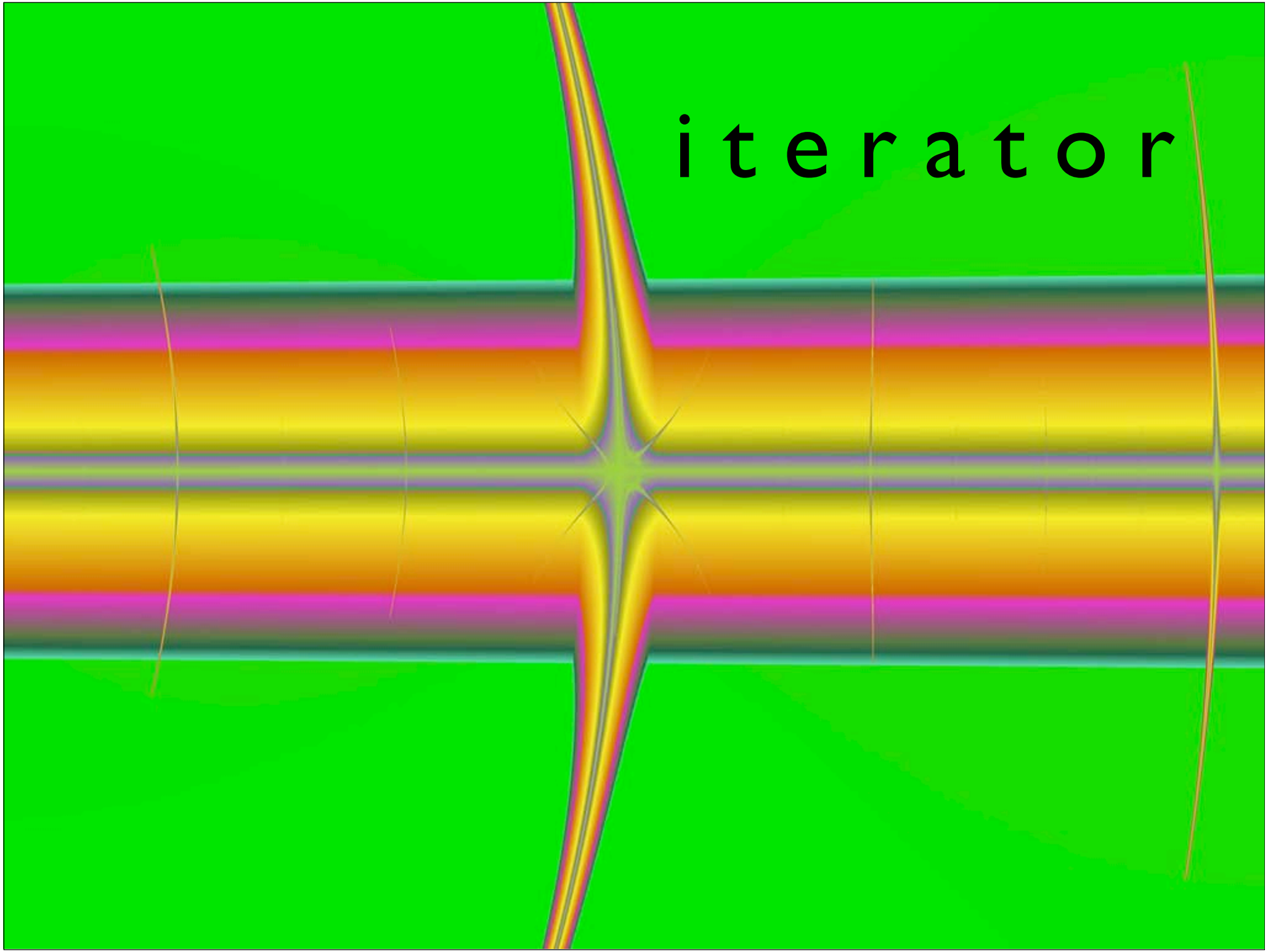
"Go To Statement Considered Harmful"

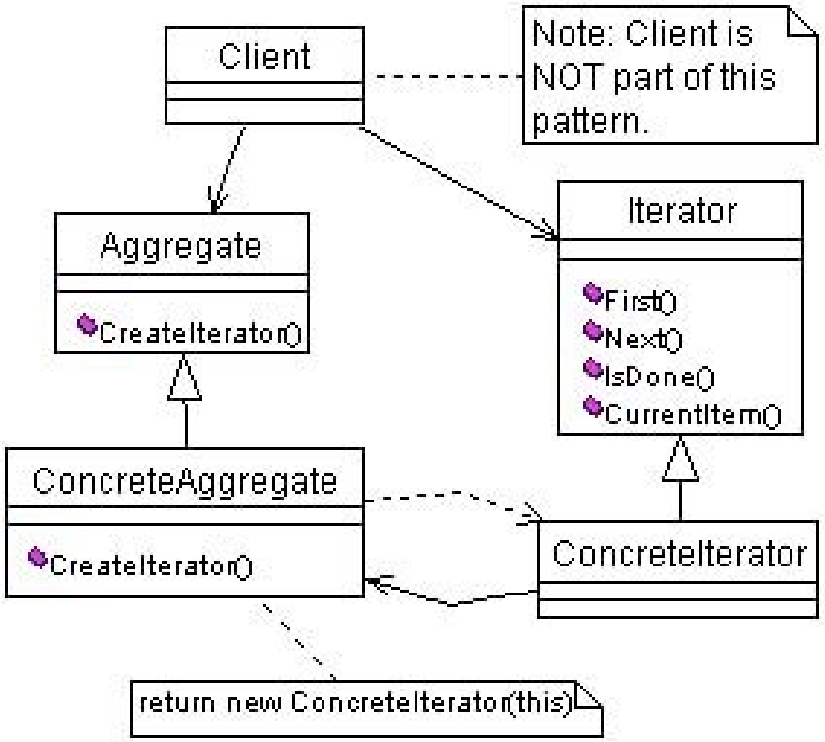
Edsger W. Dijkstra

Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.


Your biological and technological distinctiveness will
be added to our own. Resistance is futile.

the gof patterns redux

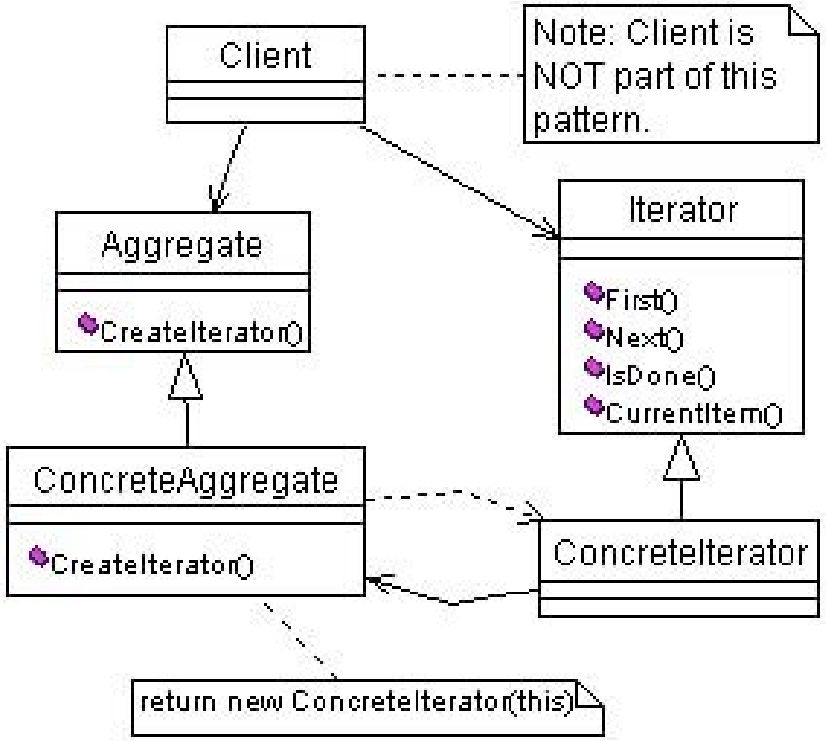

iterator



Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

```groovy
def printAll(container) {
    for (item in container) { println item }
}

def numbers = [1,2,3,4]
def months = [Mar:31, Apr:30, May:31]
def colors = [java.awt.Color.BLACK, java.awt.Color.WHITE]
printAll numbers
printAll months
printAll colors


numbers.each { n ->
  println n
}
```

ceremony



VS.

essence

```
def printAll(container) {
    for (item in container) { println item }
}
```

internal vs. external iterators

`.each` in groovy & ruby are *internal* iterators

groovy provides external iterators via `iterator()`

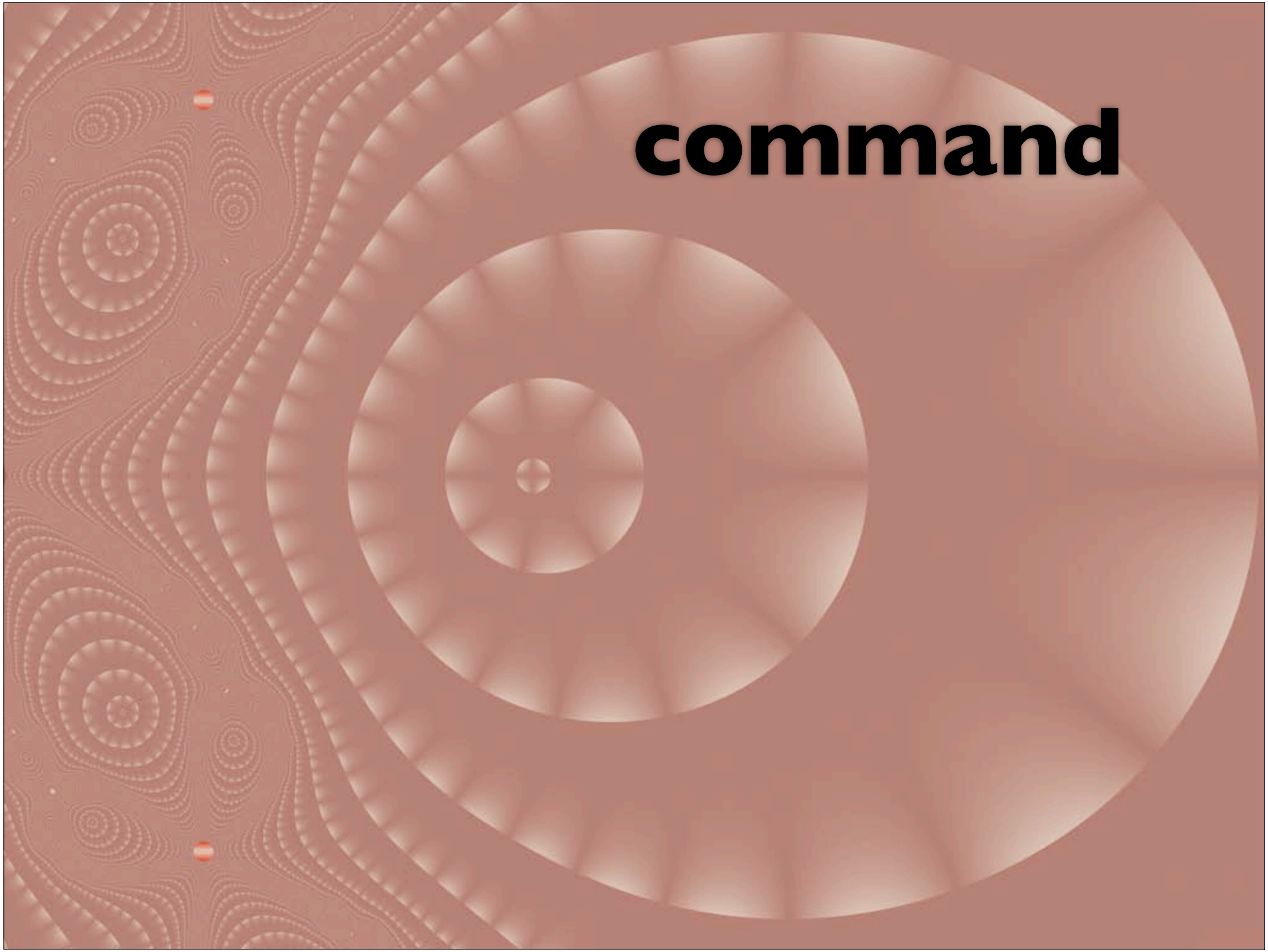
ruby 1.8 has no default syntax for external iterators

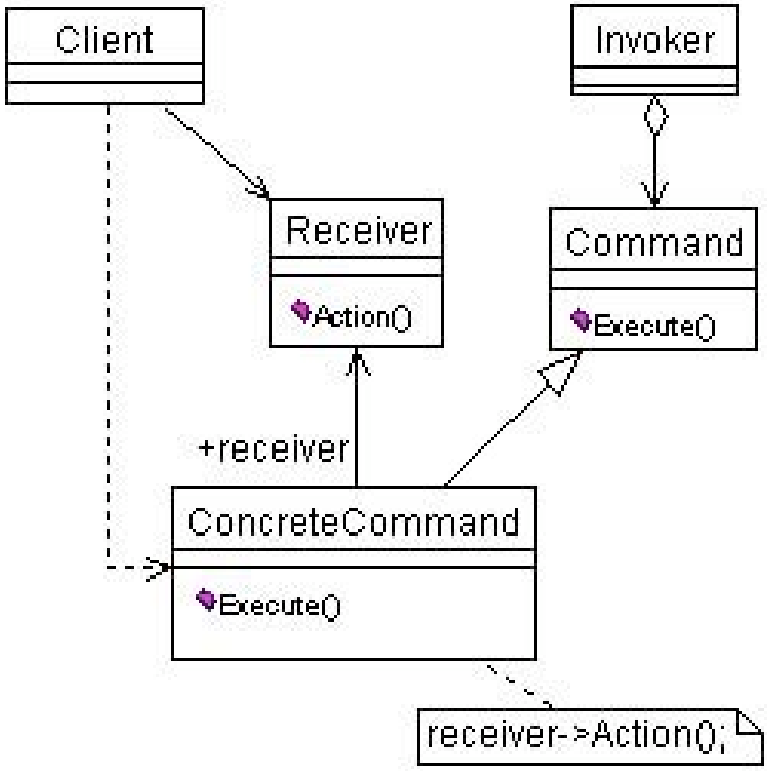
ruby 1.9 adds *enumerators* on collections

external iterator

```
iterator = 9.downto(1)
loop do
  print iterator.next
end
puts "...blastoff!"
```

1.9


command

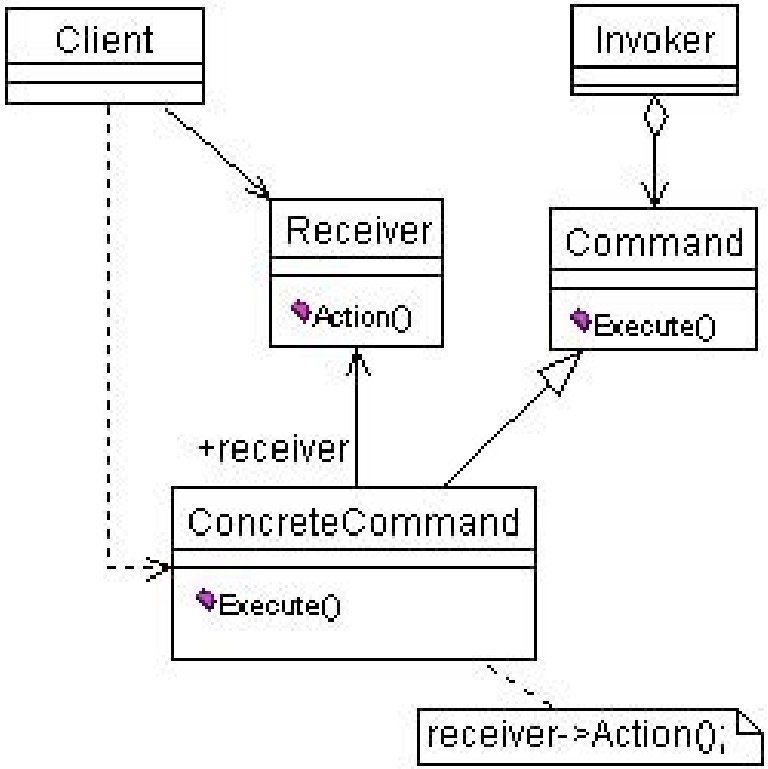


Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

commands == closures


```groovy
def count = 0
def commands = []

1.upto(10) { i ->
    commands.add { count++ }
}

println "count is initially ${count}"
commands.each { cmd ->
    cmd()
}
println "did all commands, count is ${count}"
```

```
count is initially 0
did all commands, count is 10
```

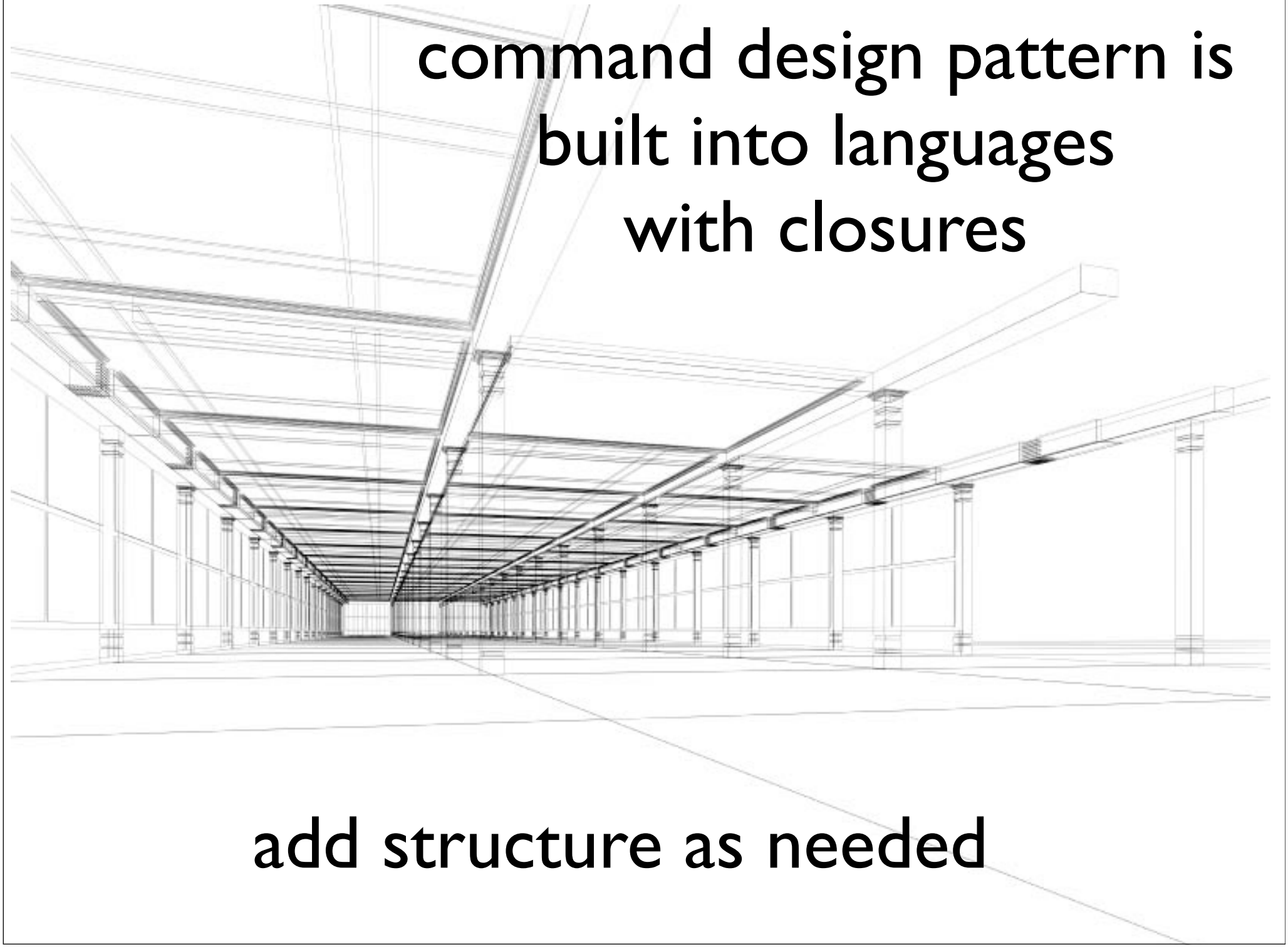


Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and ***support undoable operations***.

```
class Command {
    def cmd, uncmd

    Command(doCommand, undoCommand) {
        cmd = doCommand
        uncmd = undoCommand
    }

    def doCommand() {
        cmd()
    }

    def undoCommand() {
        uncmd()
    }
}
```


```groovy
def count = 0
def commands = []
1.upto(10) { i ->
    commands.add(new Command({count++}, {count--}))
}

println "count is initially ${count}"
commands.each { c -> c.doCommand() }
commands.reverseEach { c -> c.undoCommand() }
println "undid all commands, count is ${count}"
commands.each { c -> c.doCommand() }
println "redid all command, count is ${count}"
```

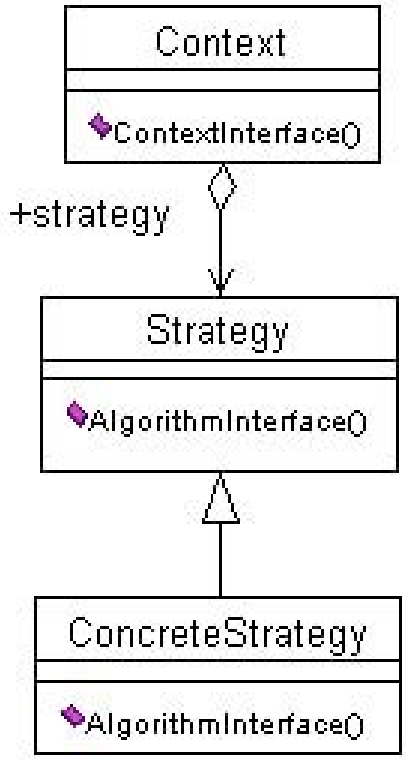
```
count is initially 0
undid all commands, count is 0
redid all command, count is 10
```



command design pattern is
built into languages
with closures

add structure as needed

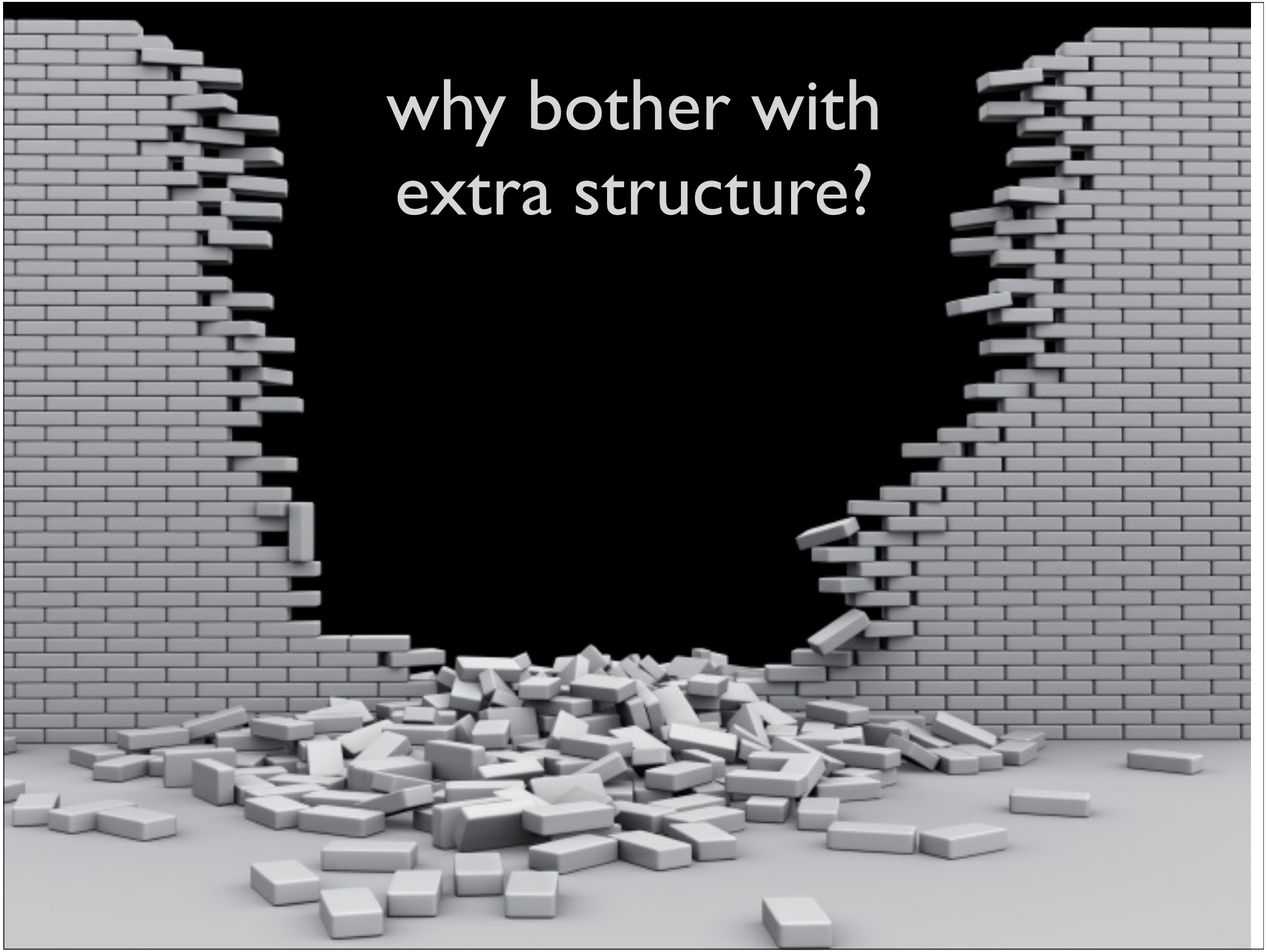

strategy



Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

```groovy
interface Calc {
    def product(n, m)
}

class CalcByMult implements Calc {
    def product(n, m) { n * m }
}

class CalcByManyAdds implements Calc {
    def product(n, m) {
        def result = 0
        n.times {
            result += m
        }
        result
    }
}
```

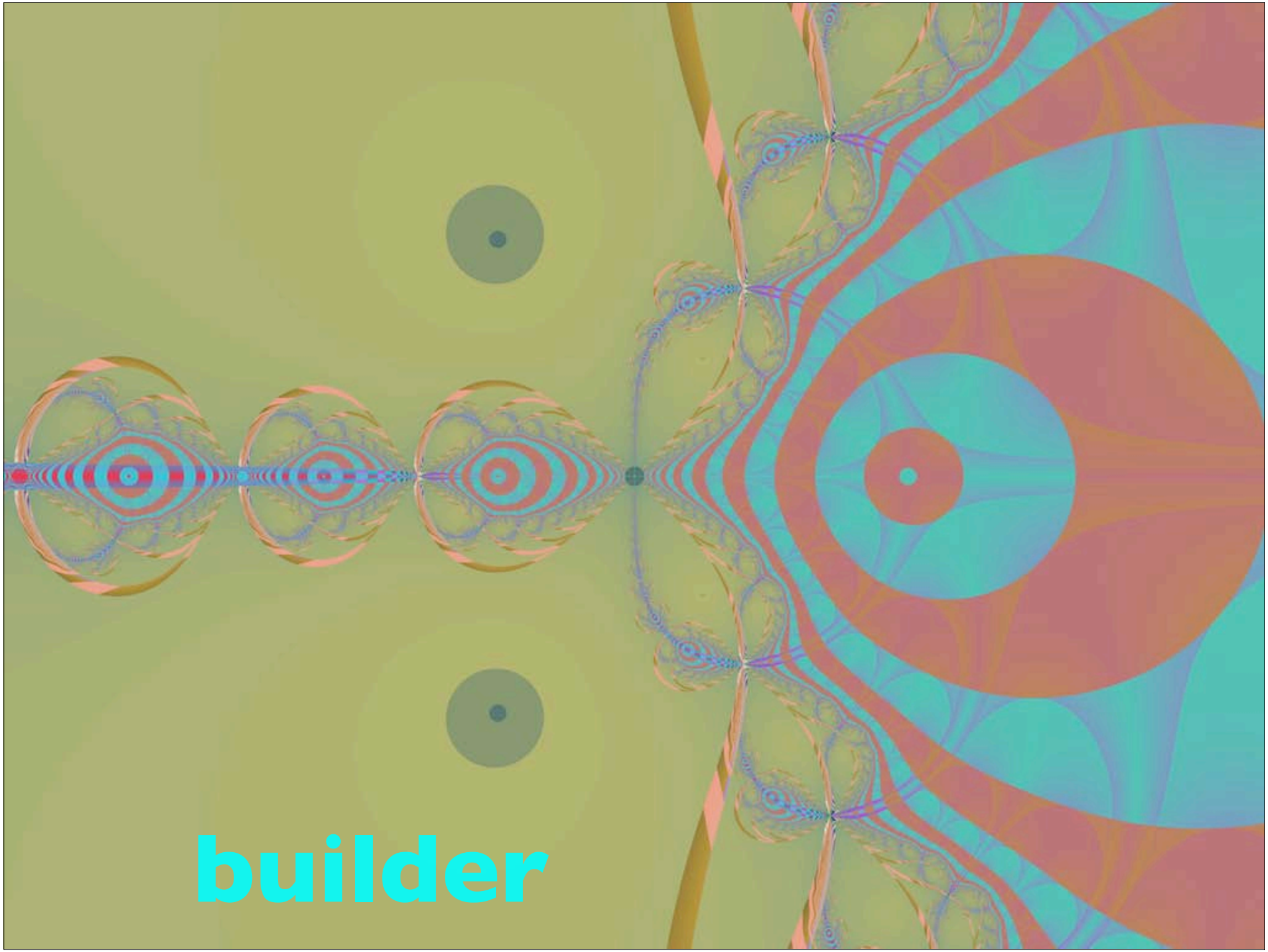
```groovy
def sampleData = [
    [3, 4, 12],
    [5, -5, -25]
]

Calc[] multiplicationStrategies = [
    new CalcByMult(),
    new CalcByManyAdds()
]

sampleData.each{ data ->
    multiplicationStrategies.each{ calc ->
        assert data[2] == calc.product(data[0], data[1])
    }
}
```

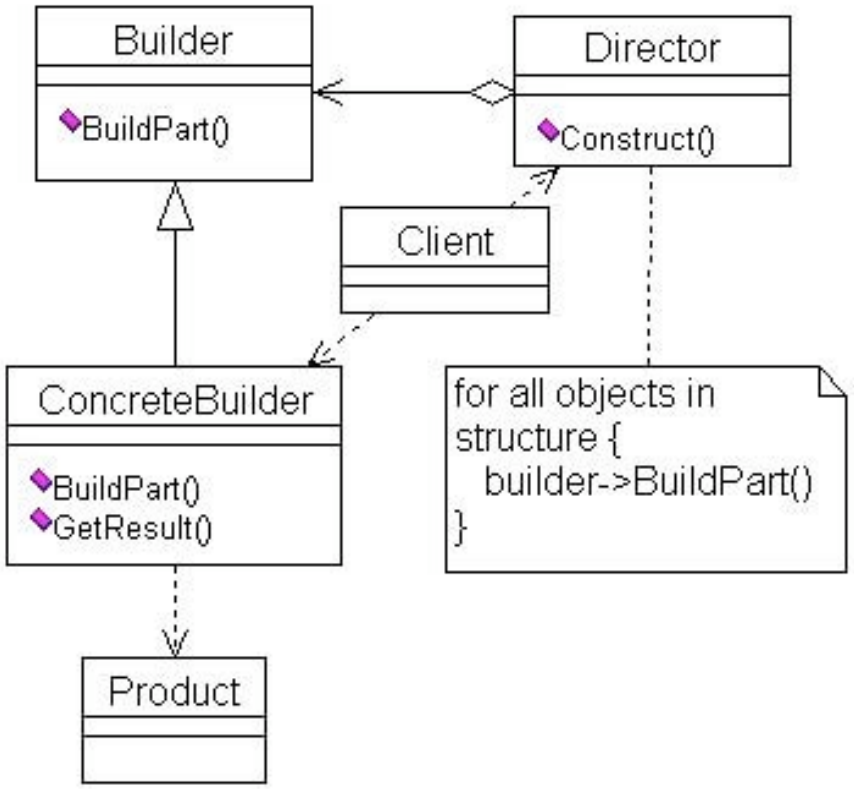

why bother with
extra structure?

Execution in the Kingdom of Nouns

steve
yegge


```groovy
def multiplicationStrategies = [
    { n, m -> n * m },
    { n, m -> def result = 0
      n.times{ result += m }
      result
    }
]

def sampleData = [
    [3, 4, 12],
    [5, -5, -25]
]

sampleData.each{ data ->
    multiplicationStrategies.each{ calc ->
        assert data[2] == calc(data[0], data[1])
    }
}
```


builder



Separate the construction of a complex object from its representation so that the same construction process can create different representations.

“traditional” builder

```ruby
class Computer
  attr_accessor :display, :motherboard, :drives

  def initialize(display=:crt, motherboard=Motherboard.new, drives=[])
    @motherboard = motherboard
    @drives = drives
    @display = display
  end

end

class CPU
  # CPU stuff
end

class BasicCPU < CPU
  # not very fast CPU stuff
end

class TurboCPU < CPU
  # very fast CPU stuff
end
```

```ruby
class Motherboard
  attr_accessor :cpu, :memory_size

  def initialize(cpu=BasicCPU.new, memory_size=1000)
    @cpu = cpu
    @memory_size = memory_size
  end
end

class Drive
  attr_reader :type, :size, :writable

  def initialize(type, size, writable)
    @type = type
    @size = size
    @writable = writable
  end
end
```

```ruby
class ComputerBuilder
  attr_reader :computer

  def initialize
    @computer = Computer.new
  end

  def turbo(has_turbo_cpu=true)
    @computer.motherboard.cpu = TurboCPU.new
  end

  def display=(display)
    @computer.display = display
  end

  def memory_size=(size_in_mb)
    @computer.motherboard.memory_size = size_in_mb
  end

  def add_cd(writer=false)
    @computer.drives << Drive.new(:cd, 760, writer)
  end
```

```ruby
def test_builder
  b = ComputerBuilder.new
  b.turbo
  b.add_cd(true)
  b.add_dvd
  b.add_hard_disk(100000)
  computer = b.computer
  assert computer.motherboard.cpu.is_a? TurboCPU
  assert computer.drives.size == 3
end
```

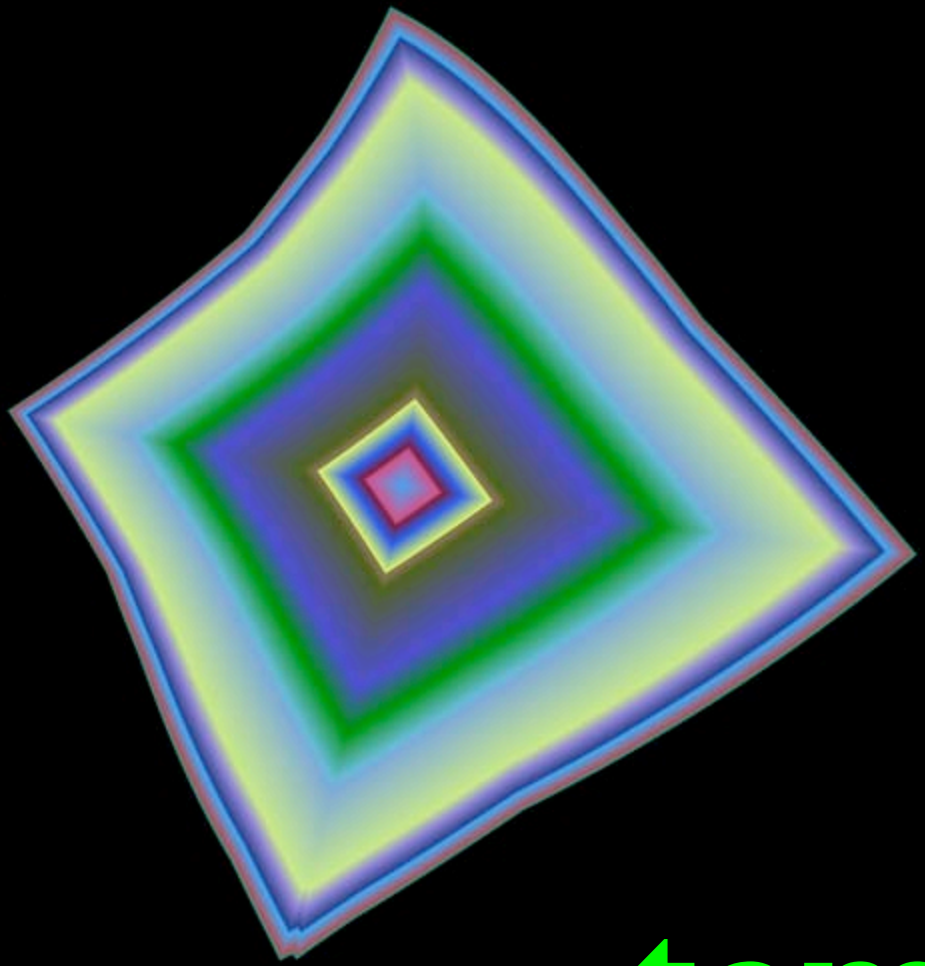
add dynamic behavior: ad hoc combinations

```ruby
def test_synthetic_method
  b = ComputerBuilder.new
  b.add_turbo_and_dvd_and_harddisk
  assert b.computer.motherboard.cpu.is_a? TurboCPU
  assert b.computer.drives.size == 2
end
```

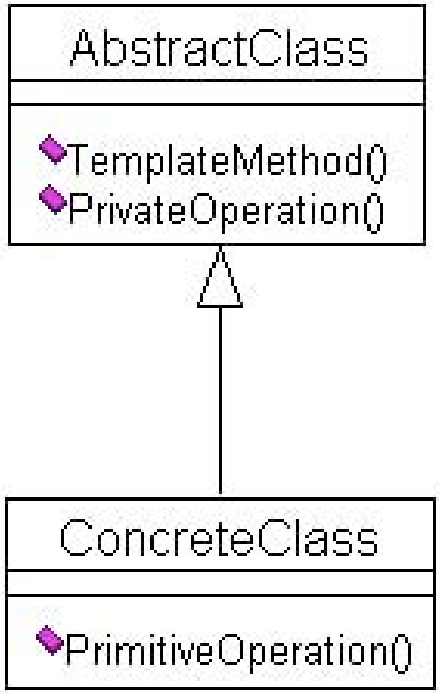
in computer_builder:

```ruby
def method_missing(name, *args)
  words = name.to_s.split("_")
  return super(name, *args) unless words.shift == 'add'
  words.each do |word|
    next if word == 'and'
    add_cd if word == 'cd'
    add_dvd if word == 'dvd'
    add_hard_disk(100000) if word == 'harddisk'
    turbo if word == 'turbo'
  end
end
```

```ruby
def test_synthetic_method
  b = ComputerBuilder.new
  b.add_turbo_and_dvd_and_harddisk
  assert b.computer.motherboard.cpu.is_a? TurboCPU
  assert b.computer.drives.size == 2
end
```



template



Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

```groovy
abstract class Customer {
    def plan

    def Customer() {
        plan = []
    }

    def abstract checkCredit()
    def abstract checkInventory()
    def abstract ship()

    def process() {
        checkCredit()
        checkInventory()
        ship()
    }
}
```

```groovy
class UsCustomer extends Customer {
    def checkCredit() {
        plan.add "checking US customer credit"
    }

    def checkInventory() {
        plan.add "checking US warehouses"
    }

    def ship() {
        plan.add "Shipping to US address"
    }
}
```

```
class EuropeanCustomer extends Customer {
    def checkCredit() {
        plan.add "checking European customer credit"
    }

    def checkInventory() {
        plan.add "checking European warehouses"
    }

    def ship() {
        plan.add "Shipping to European address"
    }
}
```

```groovy
@Test void customers() {
    def c = new UsCustomer()
    c.process()
    assertThat "checking US customer credit", is(c.plan[0])
    assertThat "checking US warehouses", is(c.plan[1])
    assertThat "Shipping to US address", is(c.plan[2])
    def e = new EuropeanCustomer()
    e.process()
    assertThat "checking European customer credit", is(e.plan[0])
    assertThat "checking European warehouses", is(e.plan[1])
    assertThat "Shipping to European address", is(e.plan[2])
}
```

what's dynamic?

so far, this is the "traditional" template-method

why not blocks as placeholders

```
class CustomerBlocks {
    def plan, checkCredit, checkInventory, ship

    def CustomerBlocks() {
        plan = []
    }

    def process() {
        checkCredit()
        checkInventory()
        ship()
    }
}
```

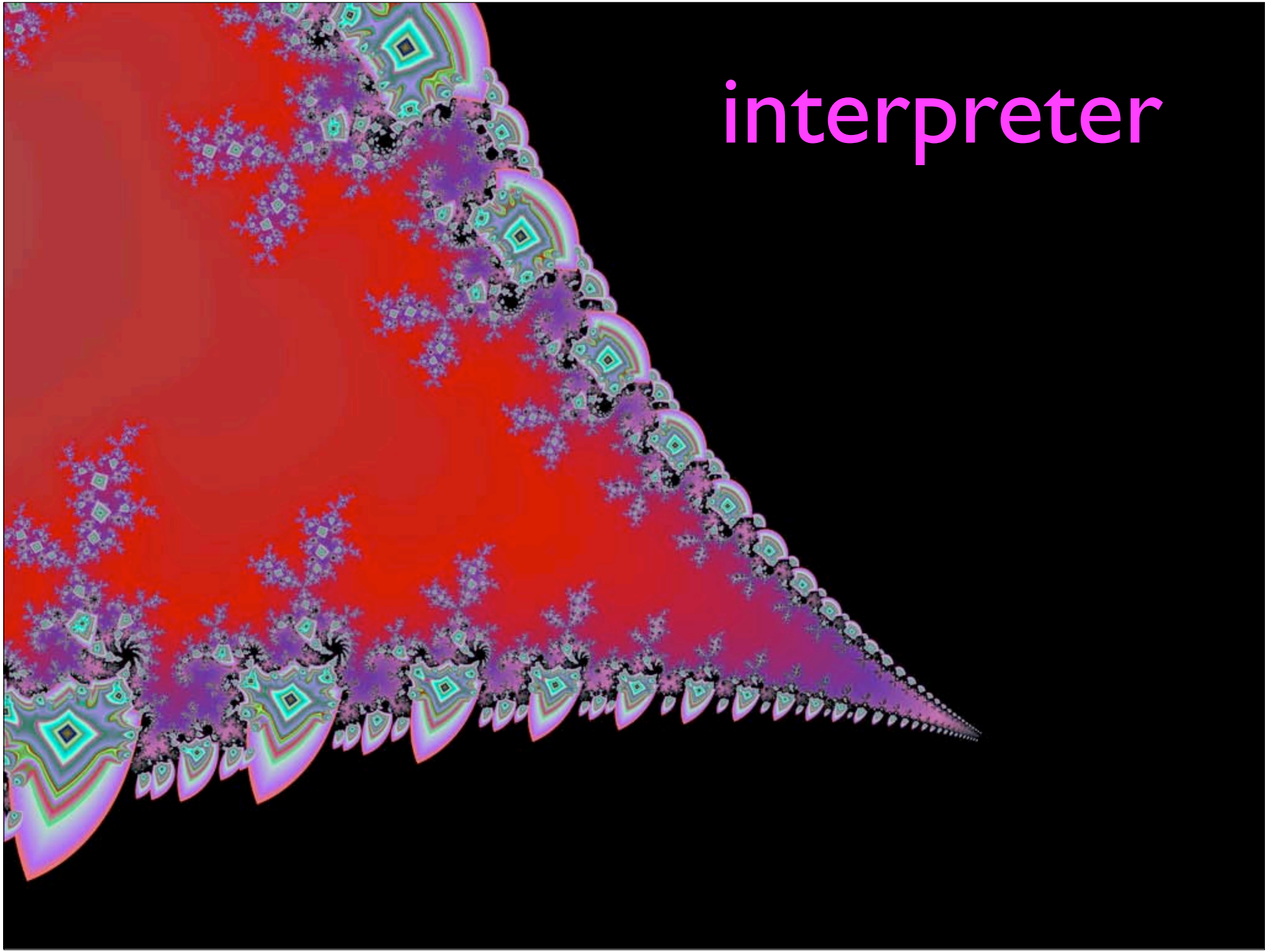
```
class UsCustomerBlocks extends CustomerBlocks{
    def UsCustomerBlocks() {
        checkCredit = { plan.add "checking US customer credit" }
        checkInventory = { plan.add "checking US warehouses" }
        ship = { plan.add "Shipping to US address" }
    }
}
```

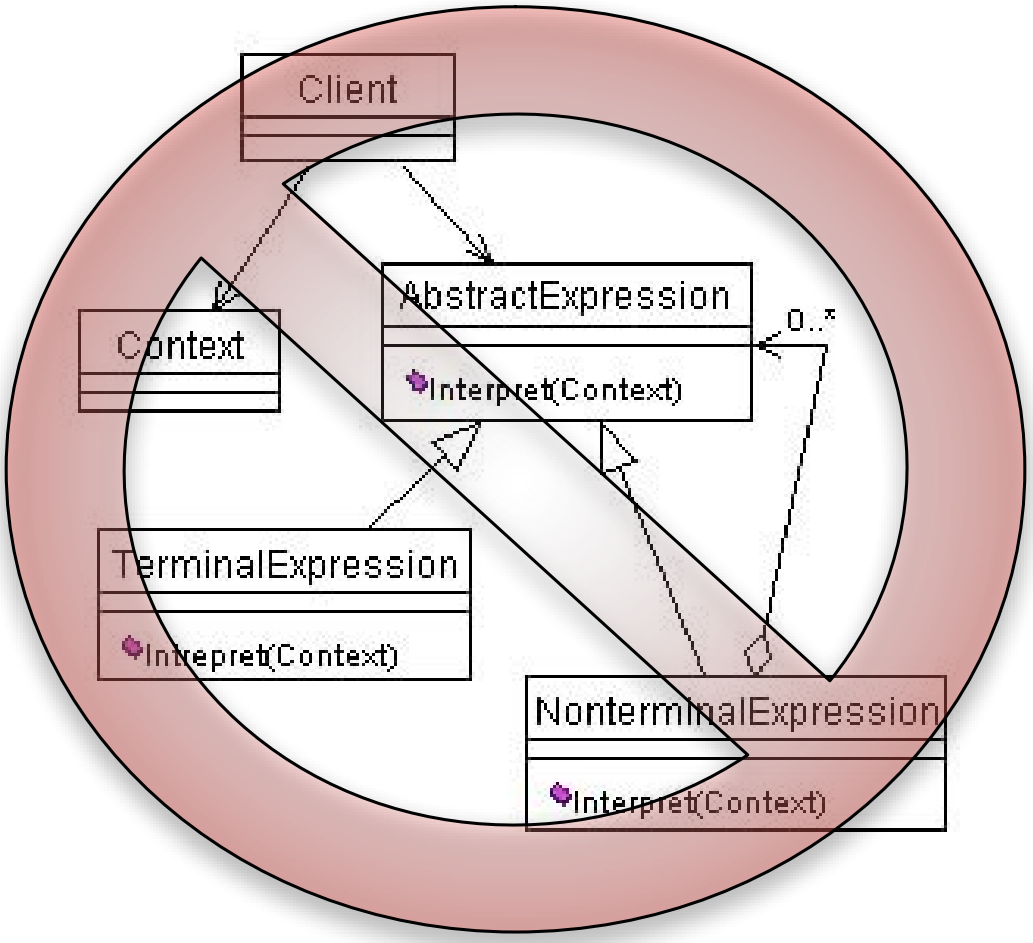
what about protection?

```
class CustomerBlocksWithProtection {
    def plan, checkCredit, checkInventory, ship

    def CustomerBlocksWithProtection() {
        plan = []
    }

    def process() {
        checkCredit?.call()
        checkInventory?.call()
        ship?.call()
    }
}
```


```groovy
class UsBlocksWithProtection extends CustomerBlocksWithProtection {
    def UsBlocksWithProtection() {
        checkCredit = { plan.add "checking US customer credit" }
        ship = { plan.add "Shipping to US address" }
    }
}
```

```groovy
class ProtectionBlocksTest {
    @Test void customers() {
        def c = new UsBlocksWithProtection()
        c.process()
        assertThat "checking US customer credit", is(c.plan[0])
        assertThat "Shipping to US address", is(c.plan[1])
    }
}
```


interpreter



Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

building domain specific languages atop groovy


Grandy
BAKERY

the goal

```
@Test void test_add_multiple_ingredients() {
    def recipe = new Recipe("Smoky Flour")
    recipe.add 1.pound.of("Flour")
    recipe.add 2.grams.of("Nutmeg")
    assertThat 2, is(recipe.ingredients.size)
    assertThat "Flour", is(recipe.ingredients.get(0).name)
    assertThat "Nutmeg", is(recipe.ingredients.get(1).name)
}
```

expando property

```
Integer.metaClass.getGram = {->
    delegate.intValue()
}
Integer.metaClass.getGrams = { -> delegate.gram }


@Test void test_gram_as_property() {
    assertThat 1, is(1.gram)
}


@Test void test_grams() {
    assertThat 2, is(2.grams)
}
```

open classes for recipes

```
Integer.metaClass.getGram = {->
    delegate.intValue()
}
Integer.metaClass.getGrams = { -> delegate.gram }
```

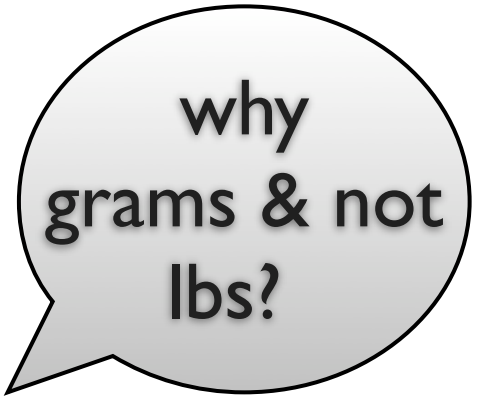

```
Integer.metaClass.getPound = {->
    delegate * 453.59237
}
Integer.metaClass.getPounds = {-> delegate.pound }
Integer.metaClass.getLb = { -> delegate.pound }
Integer.metaClass.getLbs = { -> delegate.pound }
```

recipe redux

```
@Test void test_add_multiple_ingredients() {
    def recipe = new Recipe("Smoky Flour")
    recipe.add 1.pound.of("Flour")
    recipe.add 2.grams.of("Nutmeg")
    assertThat 2, is(recipe.ingredients.size)
    assertThat "Flour", is(recipe.ingredients.get(0).name)
    assertThat "Nutmeg", is(recipe.ingredients.get(1).name)
}
```

of

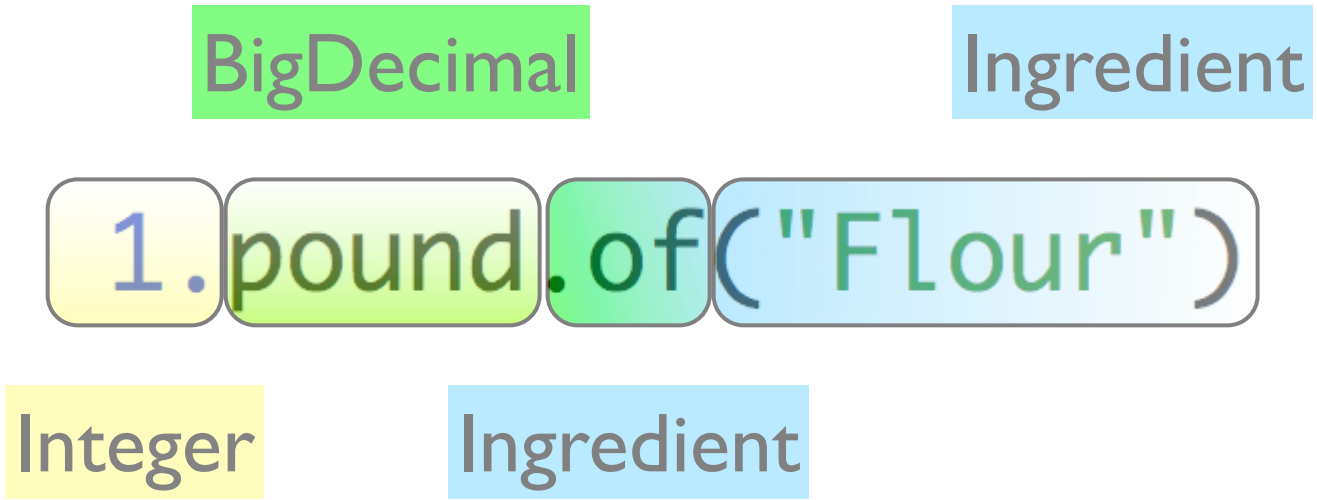
```
Integer.metaClass.of = { name ->
    def ingredient = new Ingredient(name)
    ingredient.quantity = delegate.intValue()
    ingredient
}

@Test void test_of() {
    i = 2.grams.of("Flour")
    assertTrue i instanceof Ingredient
    def i = 2.lbs.of("Flour")
    assertTrue i instanceof Ingredient
}
```



# of redux

```
Integer.metaClass.of = { name ->
    def ingredient = new Ingredient(name)
    ingredient.quantity = delegate.intValue()
    ingredient
}

BigDecimal.metaClass.of = { name ->
    def ingredient = new Ingredient(name)
    ingredient.quantity = delegate.intValue()
    ingredient
}
```

who returns what?



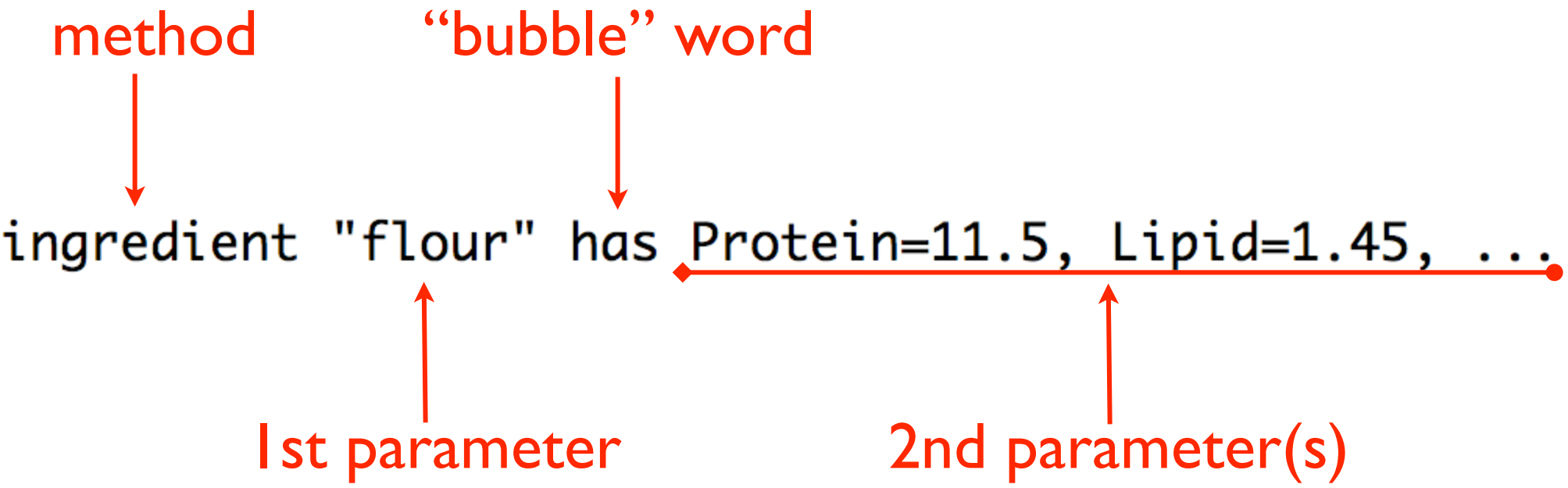
type transmogrification

transform types as needed as part of a fluent interface call

embedded interpreter

```
ingredient "flour" has Protein=11.5, Lipid=1.45, Sugars=1.12, Calcium=20, Sodium=0
ingredient "nutmeg" has Protein=5.84, Lipid=36.31, Sugars=28.49, Calcium=184, Sodium=16
ingredient "milk" has Protein=3.22, Lipid=3.25, Sugars=5.26, Calcium=113, Sodium=40


class NutritionProfile
  attr_accessor :name, :protein, :lipid, :sugars, :calcium, :sodium

  def initialize(name, protein=0, lipid=0, sugars=0, calcium=0, sodium=0)
    @name = name
    @protein, @lipid, @sugars =  protein, lipid, sugars
    @calcium, @sodium = calcium, sodium
  end

  def self.create_from_hash(name, h)
    new(name, h['protein'], h['lipid'], h['sugars'], h['calcium'], h['sodium'])
  end

  def to_s()
    "\tProtein: " +   @protein.to_s       +
    "\n\tLipid: " +   @lipid.to_s         +
    "\n\tSugars: " +  @sugars.to_s        +
    "\n\tCalcium: " + @calcium.to_s       +
    "\n\tSodium: " +  @sodium.to_s
  end

end
```

what is this?


method
"bubble" word
ingredient "flour" has Protein=11.5, Lipid=1.45, ...
1st parameter
2nd parameter(s)

```ruby
class NutritionProfileDefinition
  class << self
    def const_missing(sym)
      sym.to_s.downcase
    end
  end

  def ingredient(name, ingredients)
    NutritionProfile.create_from_hash name, ingredients
  end

  def process_definition(definition)
    t = polish_text(definition)
    instance_eval polish_text(definition)
  end

  def polish_text(definition_line)
    polished_text = definition_line.clone
    polished_text.gsub!(/=/, '=>')
    polished_text.sub!(/and /, '')
    polished_text.sub!(/has /, ',')
    polished_text
  end

end
```

```ruby
def test_polish_text
  test_text = "ingredient \"flour\" has Protein=11.5, Lipid=1.45, Sugars=1.12, Calcium=20, and Sodium=0"
  expected = 'ingredient "flour" ,Protein=>11.5, Lipid=>1.45, Sugars=>1.12, Calcium=>20, Sodium=>0'
  assert_equal expected, NutritionProfileDefinition.new.polish_text(test_text)
end
```

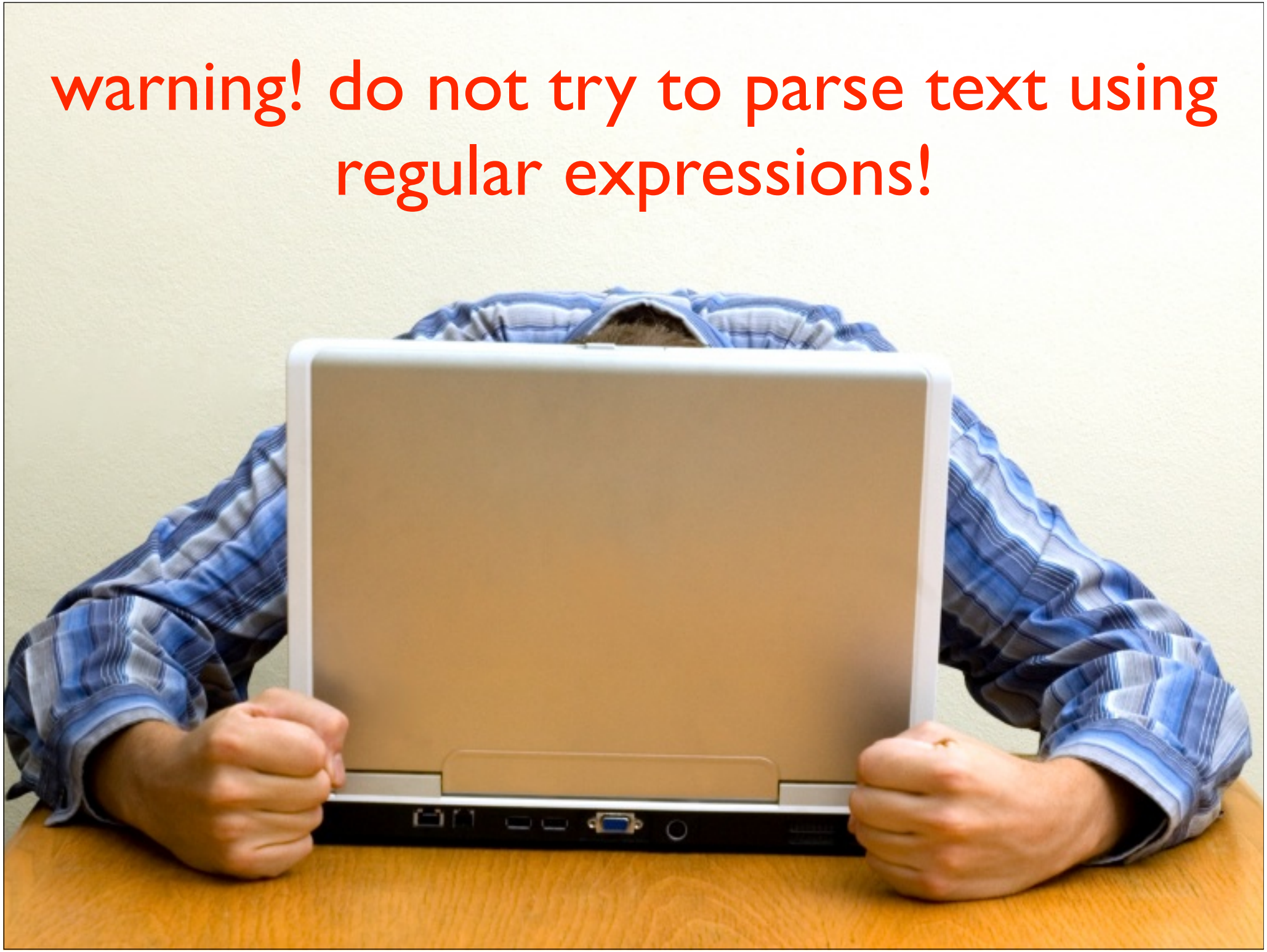
```ruby
def polish_text(definition_line)
  polished_text = definition_line.clone
  polished_text.gsub!(/=/, '=>')
  polished_text.sub!(/and /, '')
  polished_text.sub!(/has /, ',')
  polished_text
end
```

```
def process_definition(definition)
  instance_eval polish_text(definition)
end
```

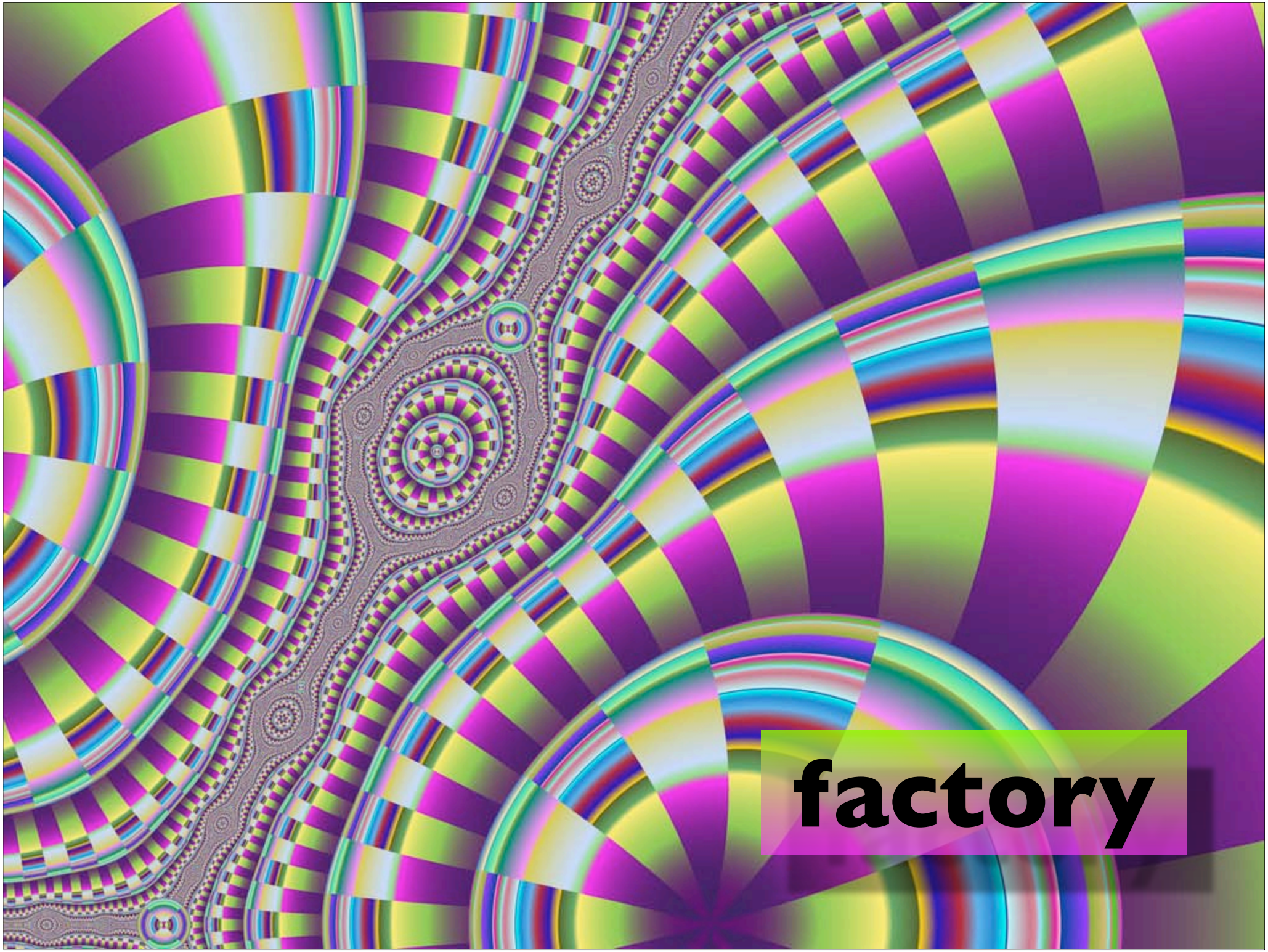
```
'ingredient "flour" ,Protein=>11.5, Lipid=>1.45,
```

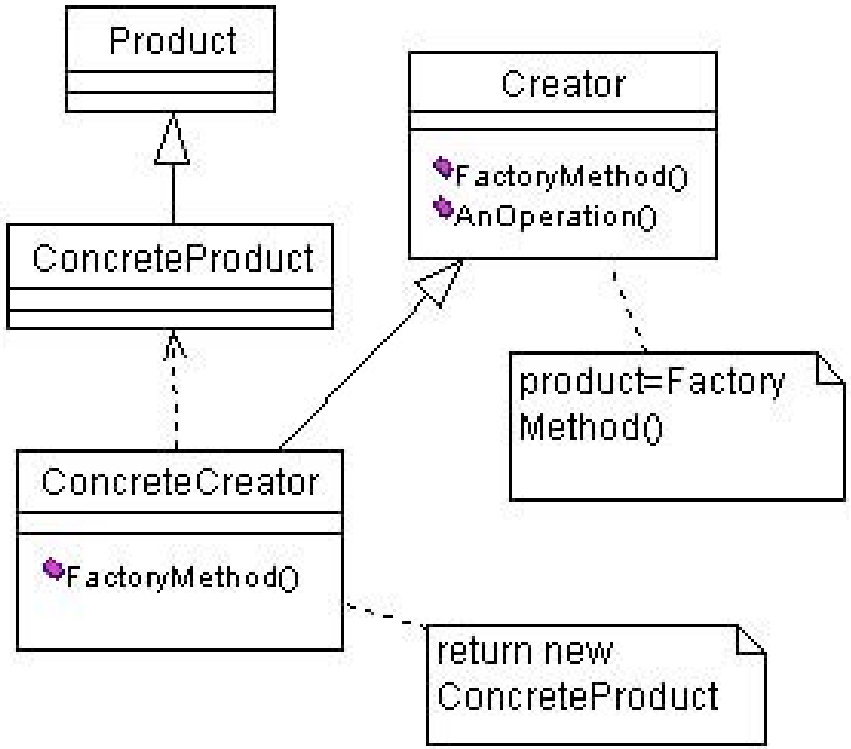
```ruby
def ingredient(name, ingredients)
  NutritionProfile.create_from_hash name, ingredients
end
```

warning! do not try to parse text using regular expressions!



internal dsl's == embedded interpreter


factory



Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

```
import java.util.*;

class CollectionFactory {
    def List getCollection(description) {
        if (description == "Array-like")
            return new ArrayList()
        else if (description == "Stack-like")
            return new Stack()
    }
}
```

```groovy
@Test void get_an_array() {
    def l = f.getCollection("Array-like")
    assertTrue l instanceof java.util.ArrayList
    l.add("foo")
    assertThat l.get(0), is("foo")
    l.removeAll(l)
    assertThat l.size(), is(0)
}

@Test void get_a_stack() {
    def s = f.getCollection("Stack-like")
    assertTrue s instanceof java.util.Stack
    s.add("foo")
    assertThat s.get(0), is("foo")
    s.removeAll(s)
    assertThat s.size(), is(0)
    s.push("bar")
    assertThat s.size(), is(1)
    def r = s.pop()
    assertThat r, is("bar")
    assertThat s.size(), is(0)
}
```


```groovy
@Test
void test_search() {
    List l = f.getCollection("Stack-like")
    assertTrue l instanceof java.util.Stack
    l.push("foo")
    assertThat l.size(), is(1)
    def r = l.search("foo")
}
```

search exists on `Stack`

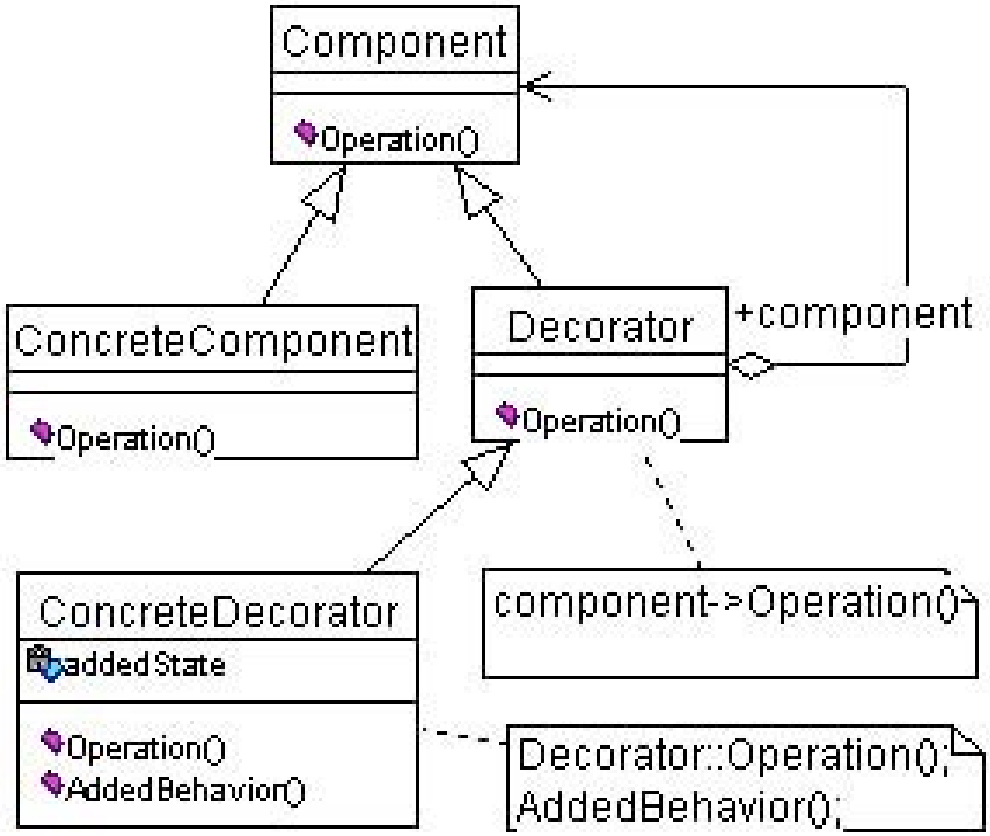
```groovy
@Test(expected=groovy.lang.MissingMethodException.class)
void verify_that_typing_does_not_help() {
    List l = f.getCollection("Array-like")
    assertTrue l instanceof java.util.ArrayList
    l.add("foo")
    assertThat l.size(), is(1)
    def r = l.search("foo")
}
```

but not on `ArrayList`

or `List`!


strong
type? nice try,
java guy


decorator



Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

“traditional” decorator

```groovy
class Logger {
    def log(String message) {
        println message
    }
}
```

```groovy
class TimeStampingLogger extends Logger {
    private Logger logger
    TimeStampingLogger(logger) {
        this.logger = logger
    }
    def log(String message) {
        def now = Calendar.instance
        logger.log("$now.time: $message")
    }
}

class UpperLogger extends Logger {
    private Logger logger
    UpperLogger(logger) {
        this.logger = logger
    }
    def log(String message) {
        logger.log(message.toUpperCase())
    }
}
```


```
def logger = new UpperLogger(
    new TimeStampingLogger(
        new Logger()))

logger.log("Groovy Rocks")
```

```
Tue May 22 07:13:50 EST 2007: GROOVY ROCKS
```

step 1: make it more dynamic

```groovy
class GenericLowerDecorator {
    private delegate
    GenericLowerDecorator(delegate) {
        this.delegate = delegate
    }

    def invokeMethod(String name, args) {
        def newargs = args.collect{ arg ->
            if (arg instanceof String) return arg.toLowerCase()
            else return arg
        }
        delegate.invokeMethod(name, newargs)
    }
}
```


```
logger = new GenericLowerDecorator(
    new TimeStampingLogger(
        new Logger()))

logger.log('IMPORTANT Message')
```

```
Tue May 22 07:27:18 EST 2007: important message
```

step 2: decorate in place

```
GroovySystem.metaClassRegistry.metaClassCreationHandle =
        new ExpandoMetaClassCreationHandle()

def logger = new Logger()
logger.metaClass.log = {
    String m -> println 'message: ' + m.toUpperCase()
}
logger.log('x')
```

decoration via smart dispatch

```ruby
module Decorator
  def initialize(decorated)
    @decorated = decorated
  end

  def method_missing(method, *args)
    args.empty? ?
      @decorated.send(method) :
      @decorated.send(method, args)
  end
end
```

```ruby
class Coffee
  def cost
    2
  end
end
```

```ruby
class Milk
  include Decorator

  def cost
    @decorated.cost + 0.4
  end
end
```

```ruby
class Whip
  include Decorator

  def cost
    @decorated.cost + 0.2
  end
end
```

```ruby
class Sprinkles
  include Decorator

  def cost
    @decorated.cost + 0.3
  end
end
```

```ruby
def test_that_decoration_works
  assert_equal 2.2,
    Whip.new(Coffee.new).cost
  assert_equal 2.9,
    Sprinkles.new(Whip.new(Milk.new(Coffee.new))).cost
end
```

```ruby
module Whipped
  def cost
    super + 0.2
  end
end

module Sprinkles
  def cost
    super + 0.3
  end
end

class Coffee
  def self.with(*args)
    args.inject(self.new) {|memo, val| memo.extend val}
  end

  def cost
    2
  end
end
```

```ruby
def test_decorator_works_via_inject
  x = Coffee.with Sprinkles, Whipped
  assert_equal 2.5, x.cost
end
```

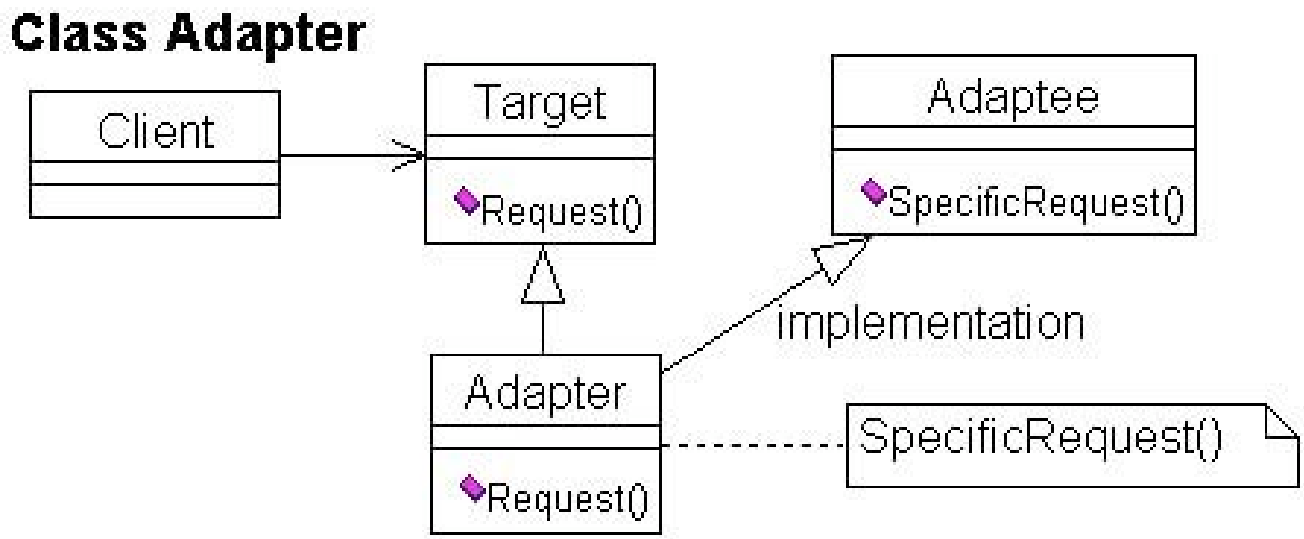

the ultimate decorator:
the recorder pattern

```ruby
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
```

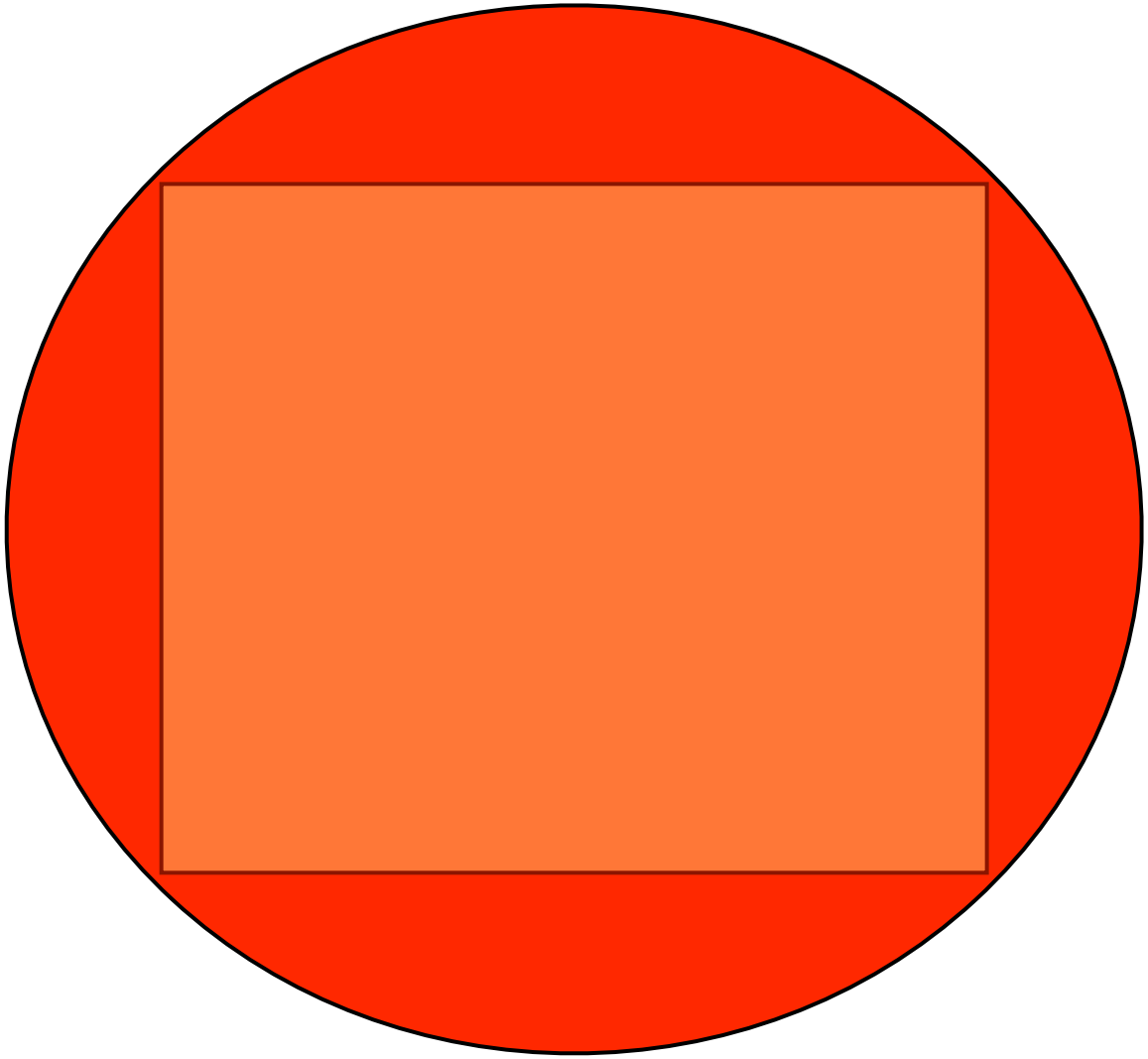
```ruby
class Recorder
  def initialize
    @messages = []
  end

  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
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
```


adapter



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



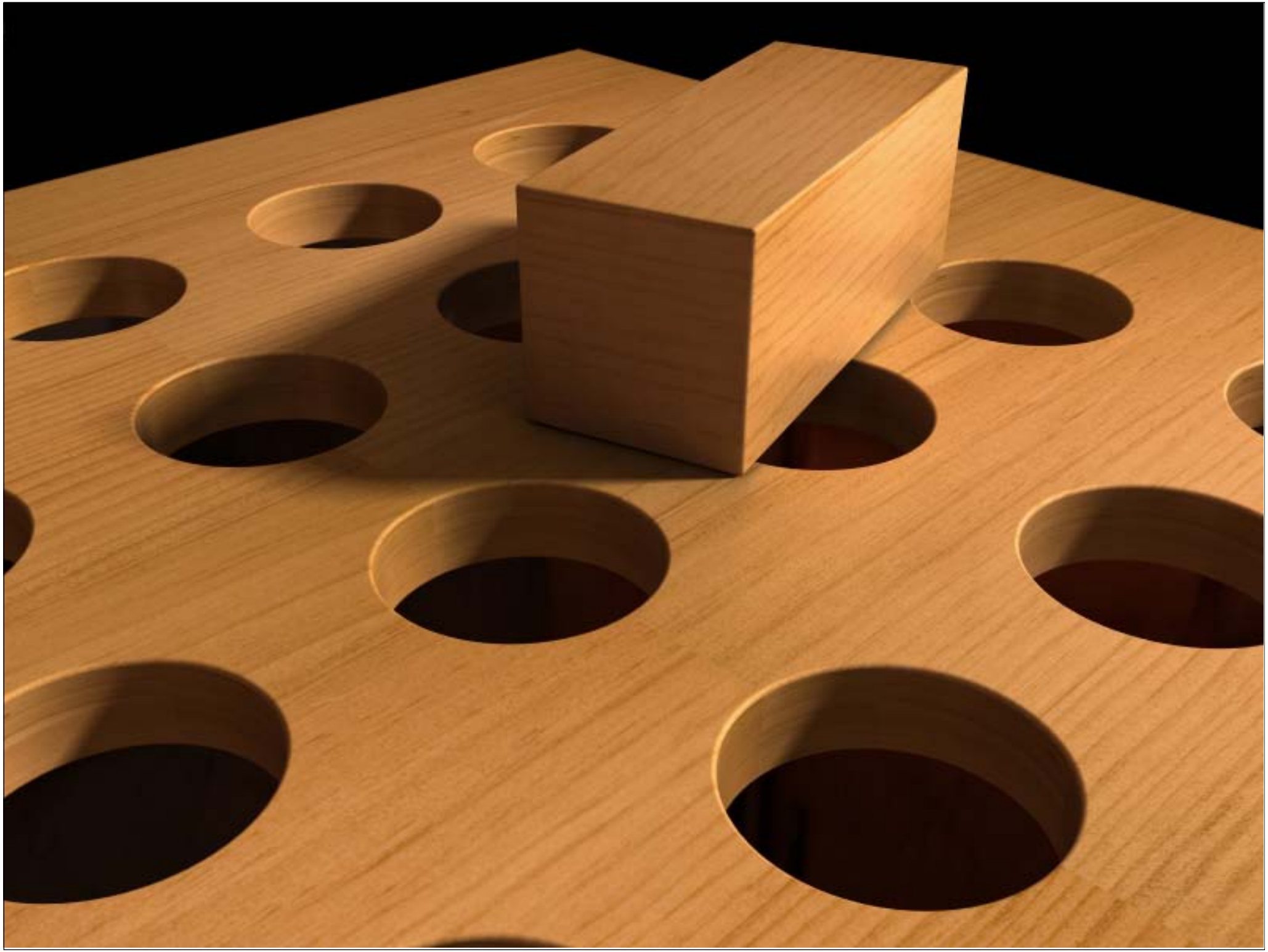
step 1: "normal" adaptor

```
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?

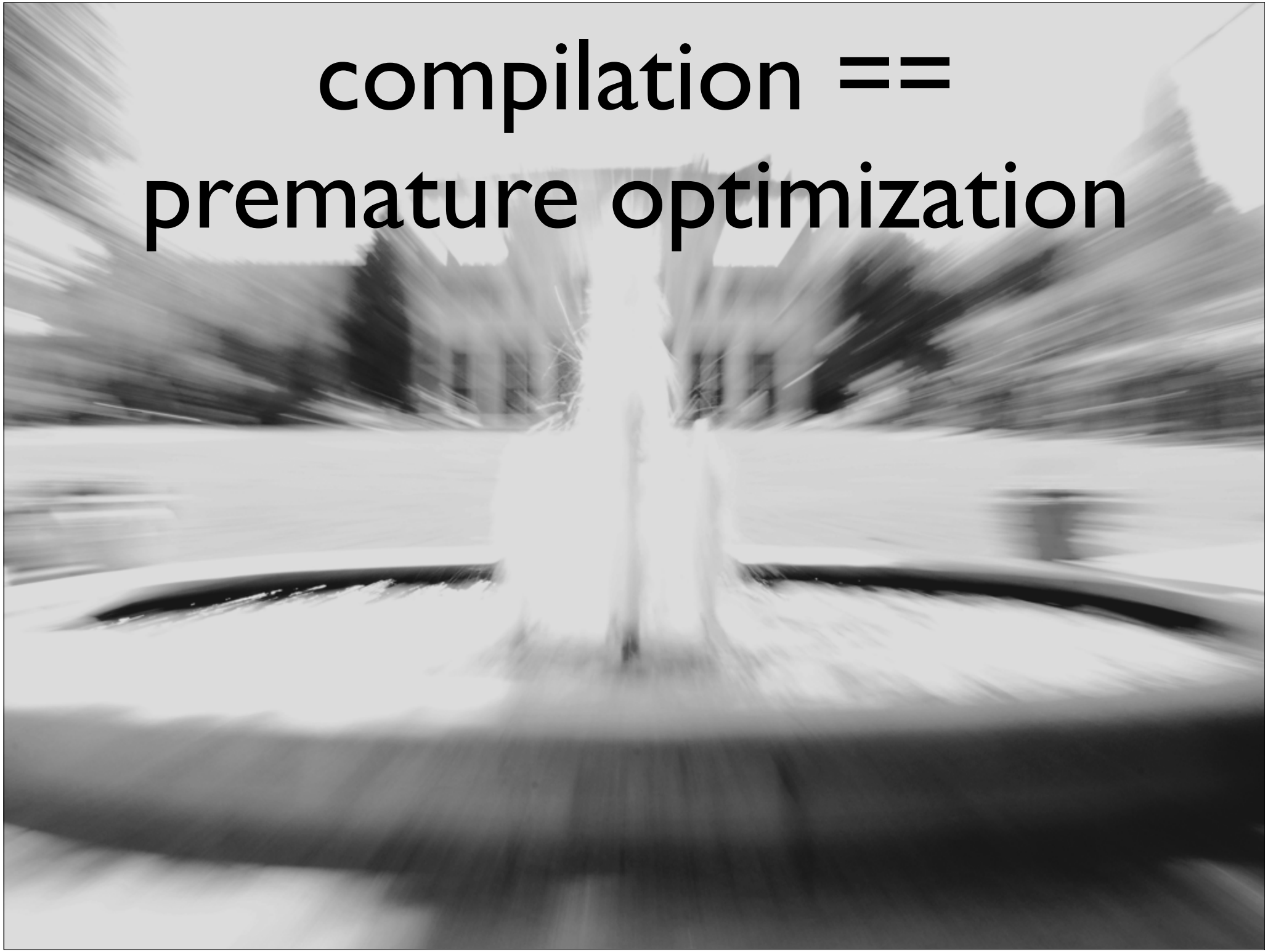
```ruby
class SquarePeg
  include InterfaceSwitching

  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
```

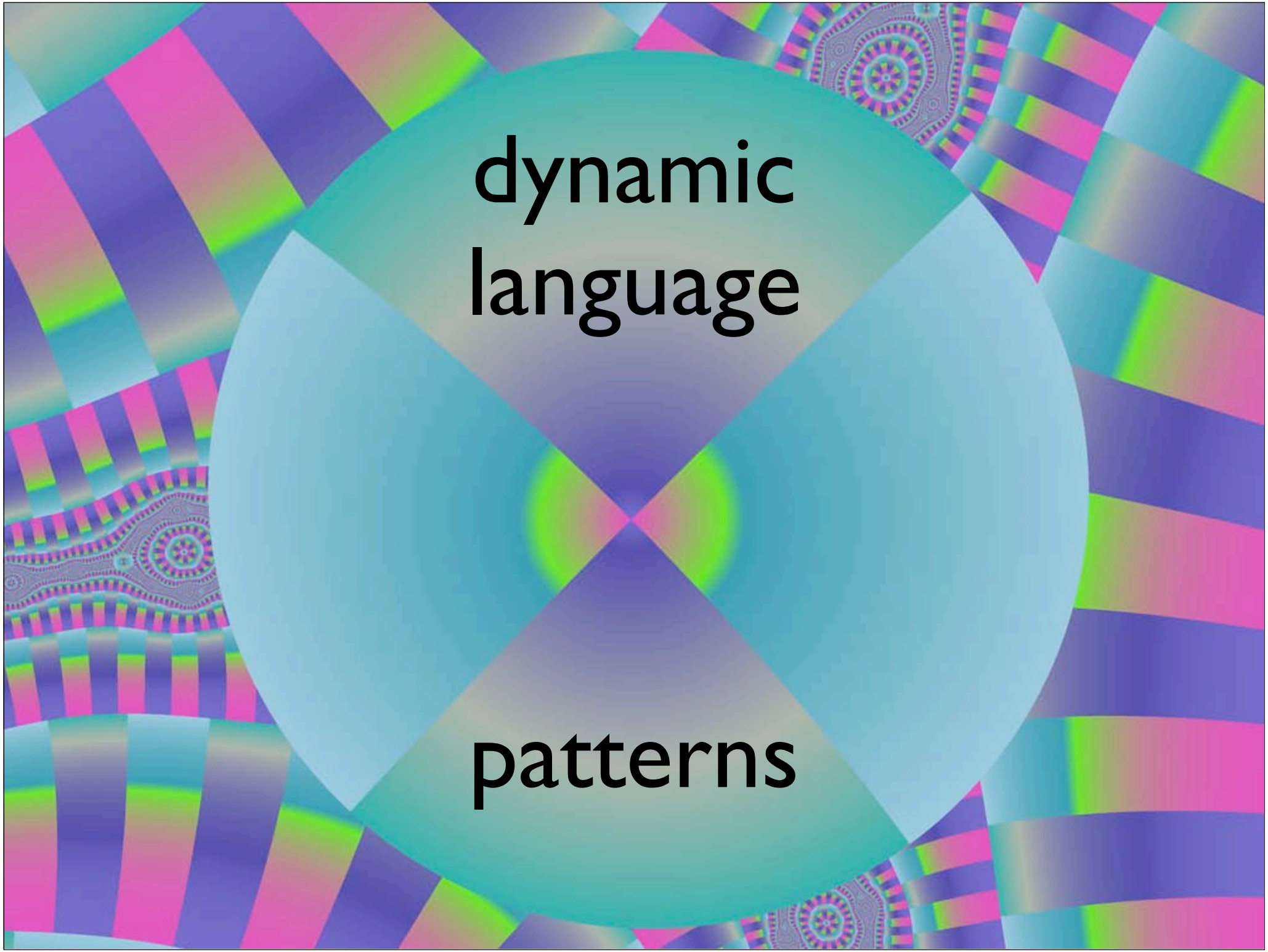
```
def test_pegs_switching
  hole = RoundHole.new( 4.0 )
  4.upto(7) do |i|
    peg = SquarePeg.new(i)
    peg.with_interface(:holes) do
      if (i < 6)
        assert hole.peg_fits?(peg)
      else
        assert ! hole.peg_fits?(peg)
      end
    end
  end
end
```

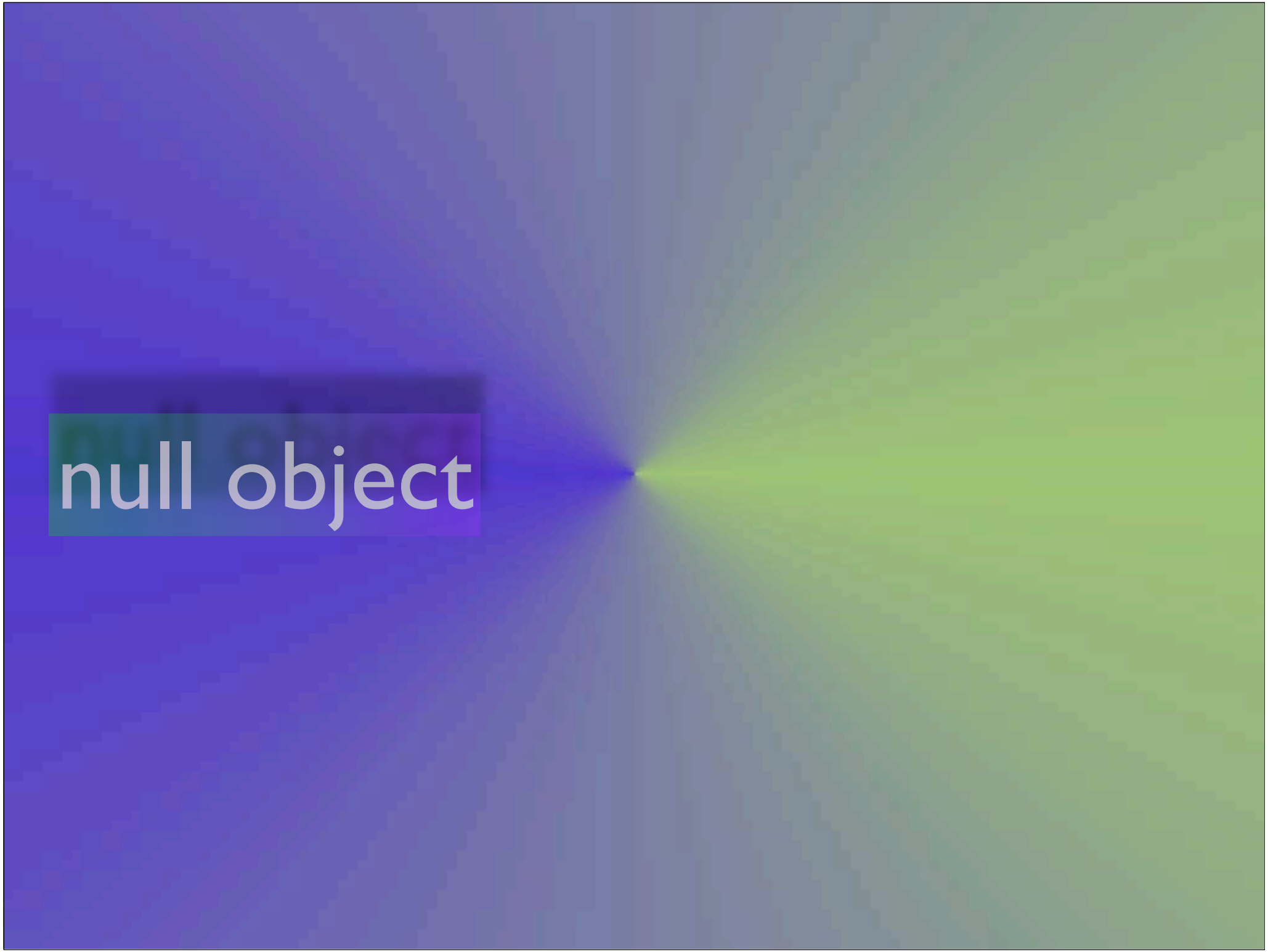
interface helper

```
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


dynamic
language
patterns


null object

The null object pattern uses a special object representing null, instead of using an actual null. The result of using the null object should semantically be equivalent to doing nothing.


```groovy
class Job {
    def salary
}

class Person {
    def name
    def Job job
}

def people = [
    new Person(name:'Tom', job:new Job(salary:1000)),
    new Person(name:'Dick', job:new Job(salary:1200)),
]

def biggestSalary = people.collect{ p -> p.job.salary }.max()
println biggestSalary
```

```
people << new Person(name:'Harry')
```


```
def biggestSalary = people.collect{ p -> p.job.salary }.max()
println biggestSalary
```

java.lang.NullPointerException

```
class NullJob extends Job { def salary = 0 }

people << new Person(name:'Harry', job:new NullJob())
biggestSalary = people.collect{ p -> p.job.salary }.max()
println biggestSalary
```

```
people << new Person(name:'Harry')
biggestSalary = people.collect{
    p -> p.job?.salary
}.max()
println biggestSalary
```

p?.job?.salary

this pattern doesn't exist in ruby

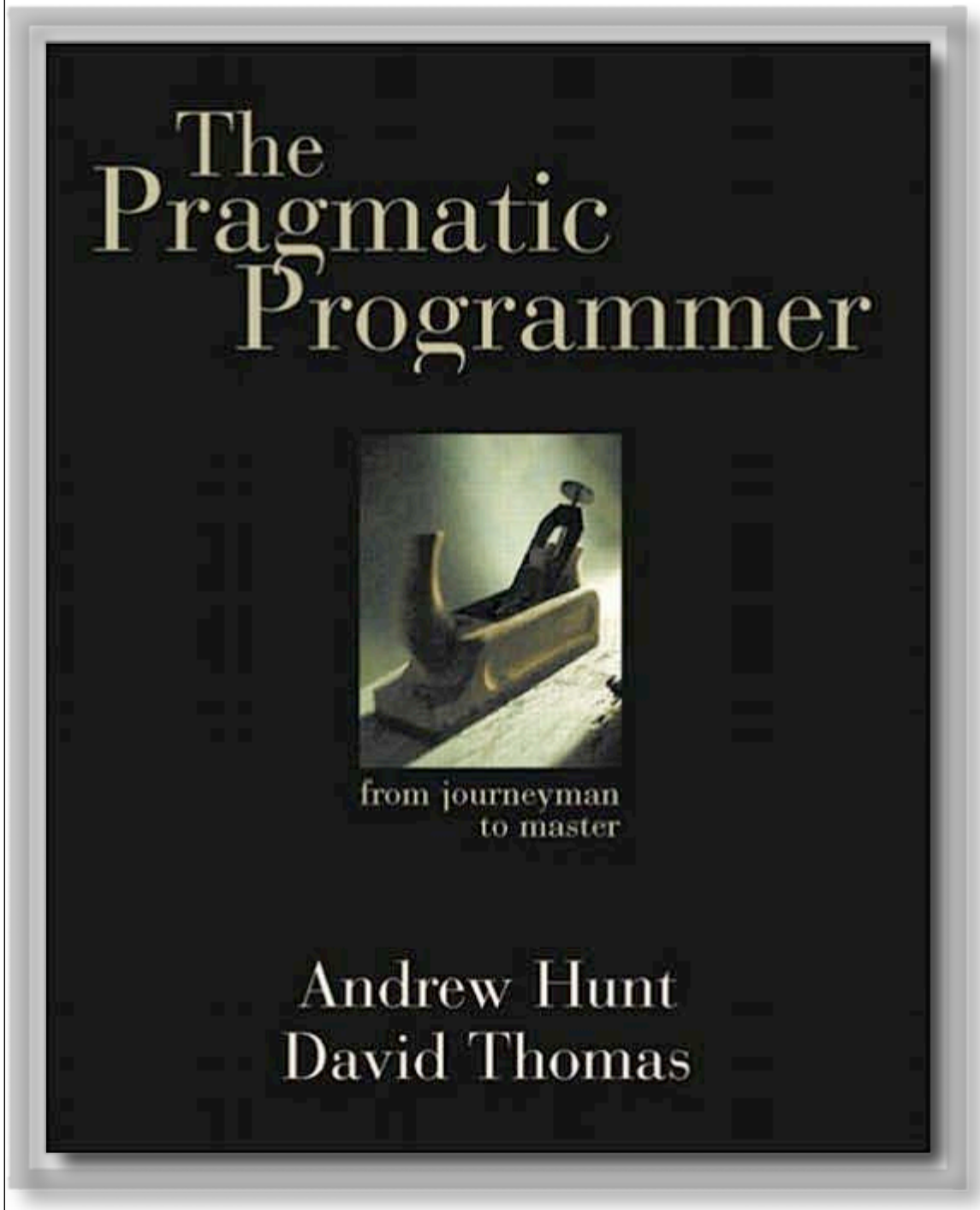
NilClass is already defined

```ruby
class CustomerWithOptionalTemplates
  attr_accessor :plan, :check_credit, :check_inventory, :ship

  def initialize
    @plan = []
  end

  def process
    @check_credit.call unless @check_credit.nil?
    @check_inventory.call unless @check_inventory.nil?
    @ship.call unless @ship.nil?
  end

end
```

```ruby
class NilClass
  def blank?
    true
  end
end
```

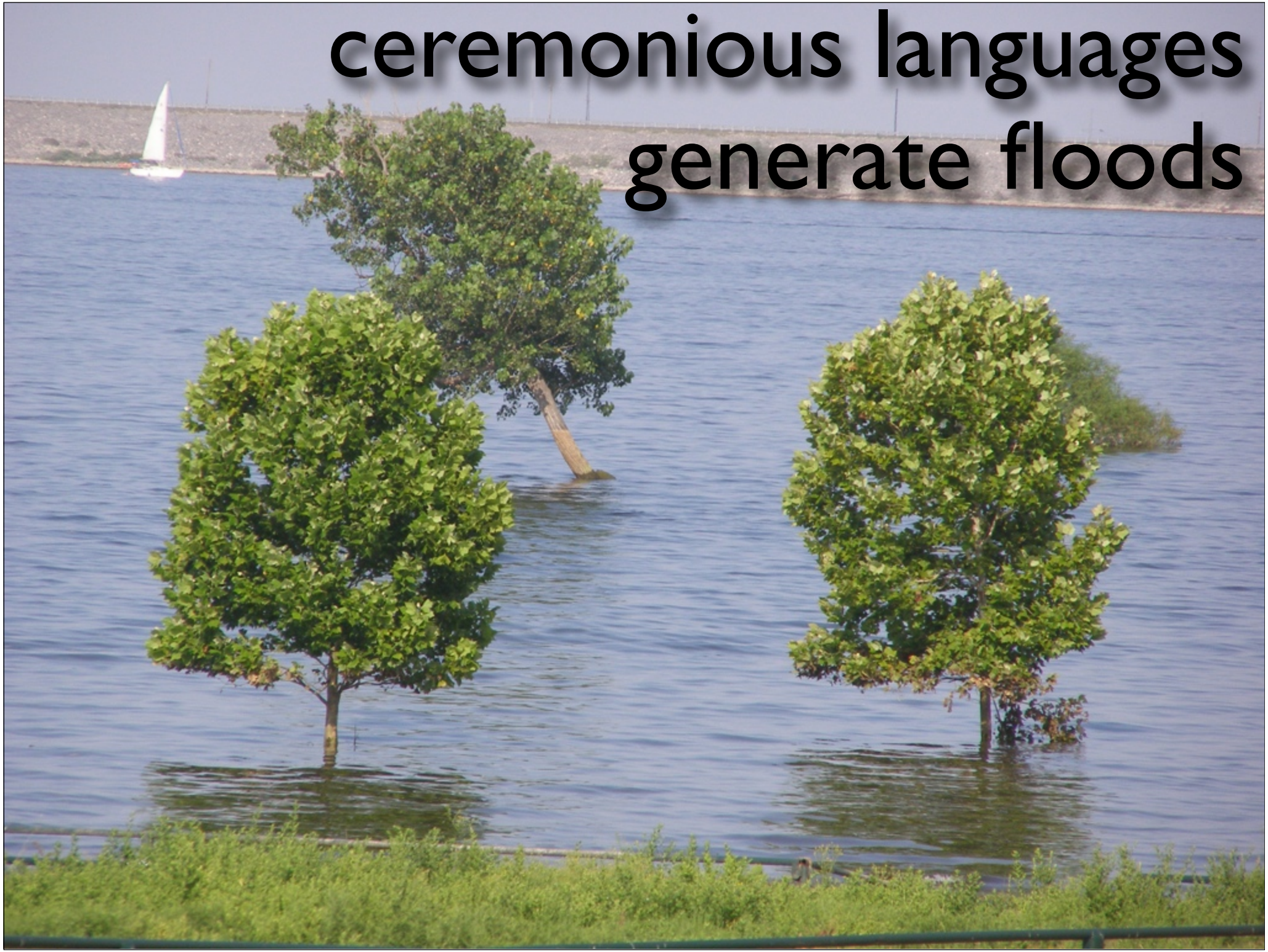
```ruby
class NilClass
  WHINERS = [ ::ActiveRecord::Base, ::Array ]

  @@method_class_map = Hash.new

  WHINERS.each do |klass|
    methods = klass.public_instance_methods - public_instance_methods
    methods.each do |method|
      @@method_class_map[method.to_sym] = klass
    end
  end

  def id
    raise RuntimeError, "Called id for nil..."
  end

  private
    def method_missing(method, *args, &block)
      raise_nil_warning_for @@method_class_map[method], method, caller
    end

    def raise_nil_warning_for(klass = nil, selector = nil, with_caller = nil)
      message = "You have a nil object when you didn't expect it!"
      message << "\nYou might have expected an instance of #{klass}." if klass
      message << "\nThe error occurred while evaluating nil.#{selector}" if selector

      raise NoMethodError, message, with_caller || caller
    end
end
```

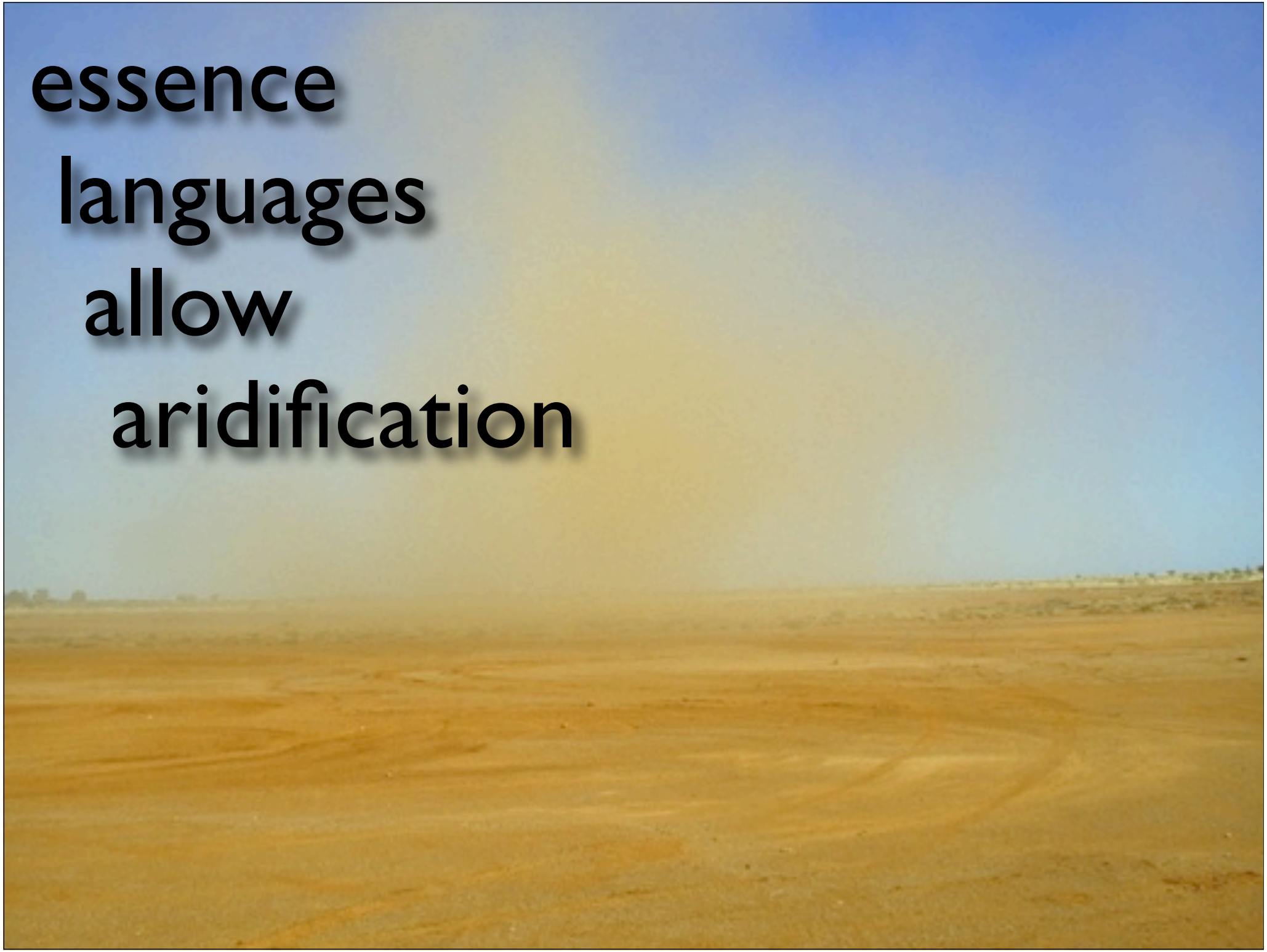

aridifier



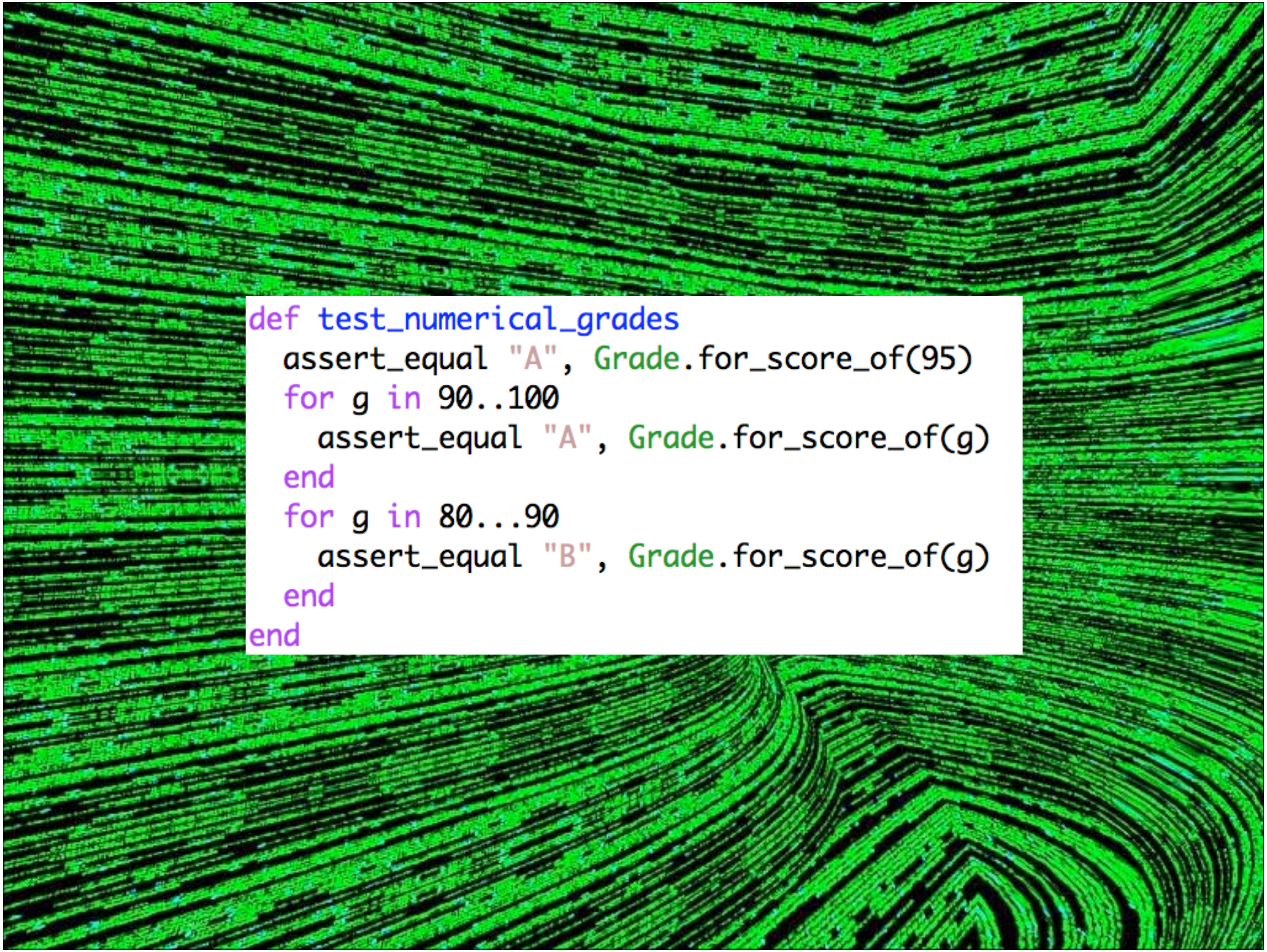
d r y

don't repeat yourself


ceremonious languages
generate floods


essence
languages
allow
aridification

```ruby
class Grade
  class << self
    def for_score_of(grade)
      case grade
        when 90..100: 'A'
        when 80..90 : 'B'
        when 70..80 : 'C'
        when 60..70 : 'D'
        when Integer: 'F'
        when /[A-D]/, /[F]/ : grade
        else raise "Not a grade: #{grade}"
      end
    end
  end
end
```


def test_numerical_grades
assert_equal "A", Grade.for_score_of(95)
for g in 90..100
assert_equal "A", Grade.for_score_of(g)
end
for g in 80...90
assert_equal "B", Grade.for_score_of(g)
end
end

```ruby
TestGrades.class_eval do
  grade_range = {
    'A' => 90..100,
    'B' => 80...90,
    'C' => 70...80,
    'D' => 60...70,
    'F' => 0...60}

  grade_range.each do |k, v|
    method_name = ("test_" + k + "_letter_grade").to_sym
    define_method method_name do
      for g in v
        assert_equal k, Grade.for_score_of(g)
      end
    end
  end
end
```

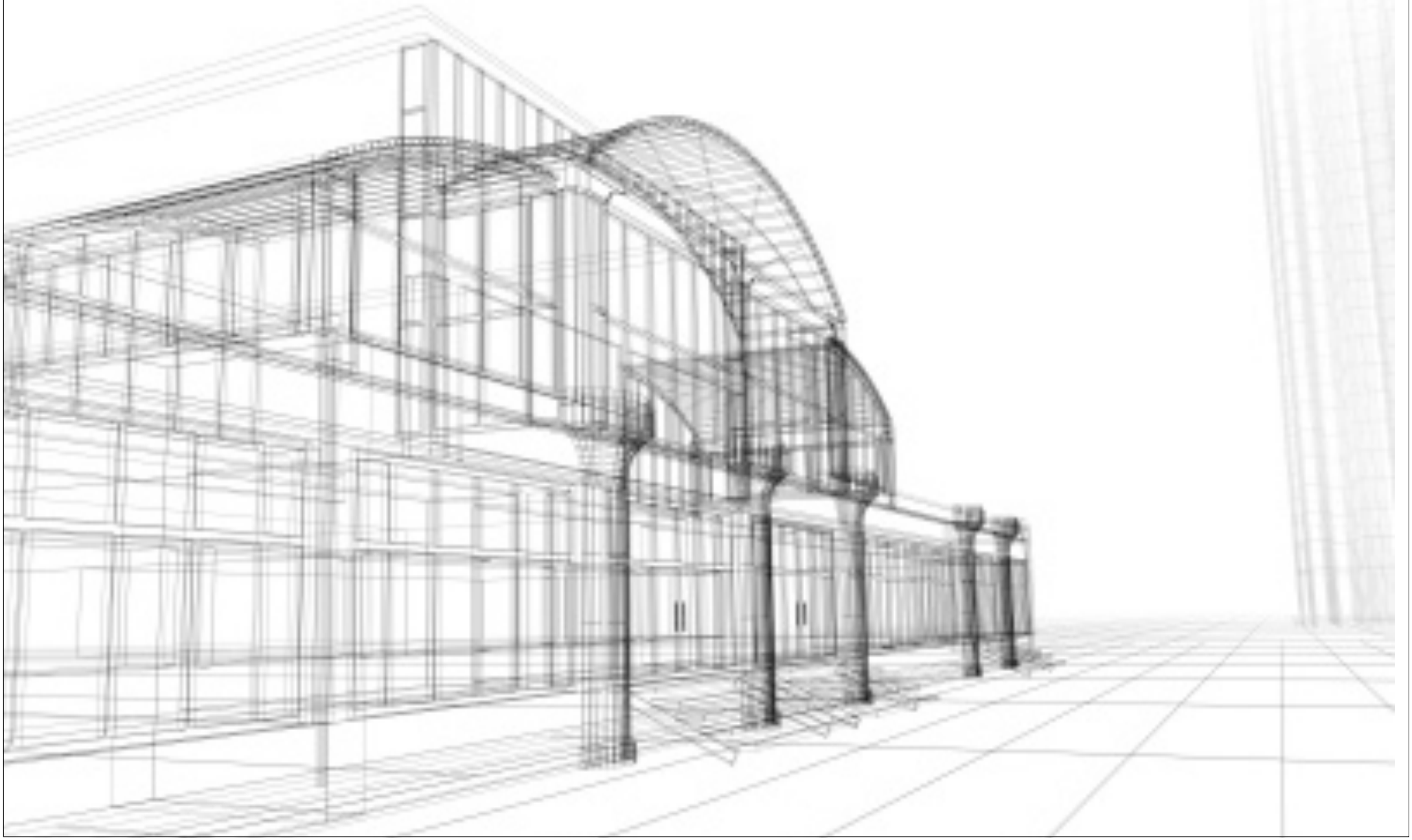
moist tests?

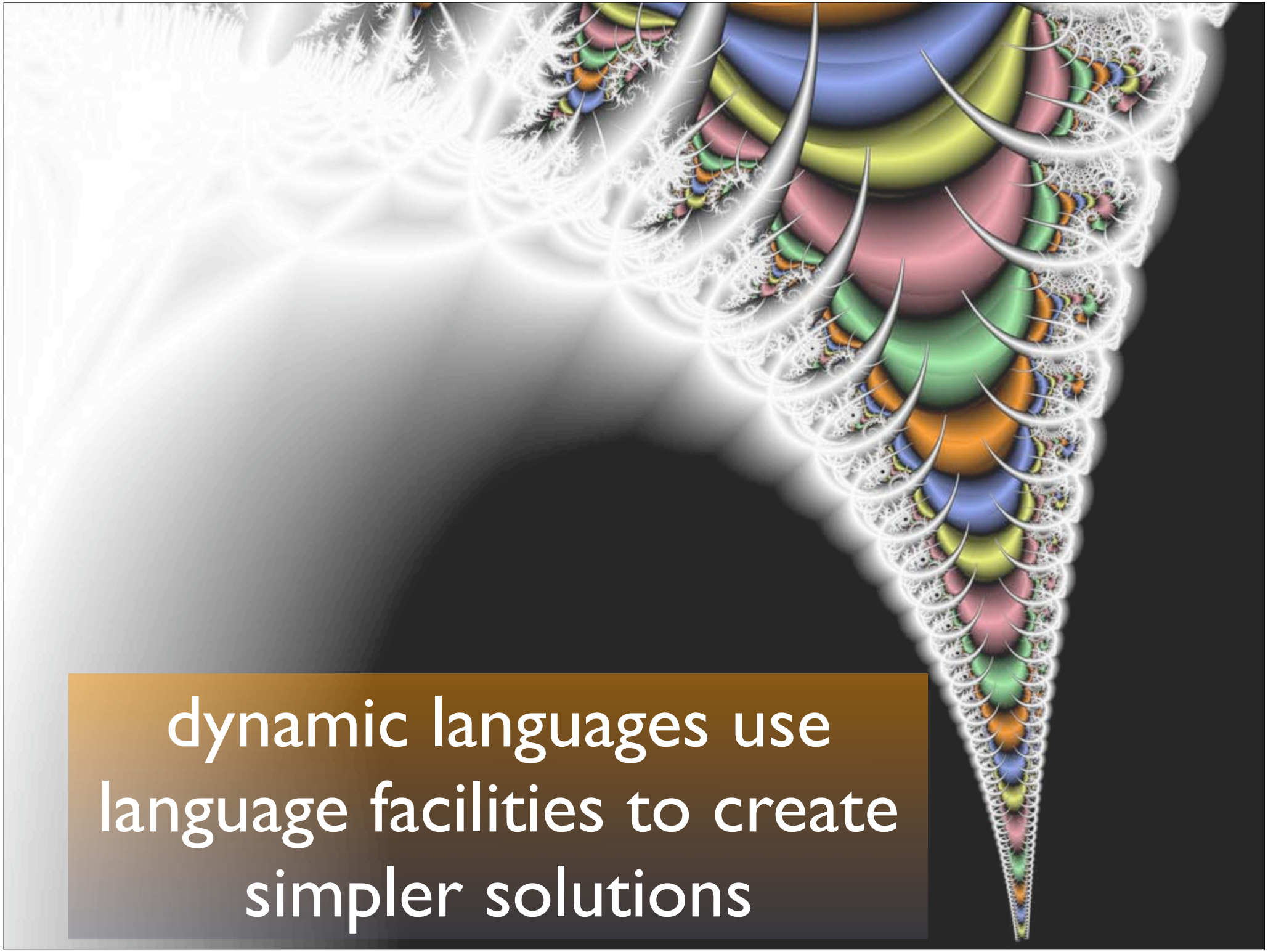
```ruby
def test_delegating_to_array
  arr = Array.new
  q = FQueue.new arr
  q.enqueue "one"
  assert_equal 1, q.size
  assert_equal "one", q.dequeue
end

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
```

drier tests

```
TestForwardQueue.class_eval do
    [Array, Queue, SizedQueue].each do |c|
      method_name = ("test_queue_delegated_to_" + c.to_s).to_sym
      define_method method_name,lambda {
        if c == SizedQueue
          a = c.new 12
        else
          a = c.new
        end
        q = FQueue.new a
        q.enqueue "one"
        assert_equal 1, q.size
        assert_equal "one", q.dequeue
      }
    end
end
```

“traditional” design patterns rely heavily on *structure* to solve problems




dynamic languages use
language facilities to create
simpler solutions


understand patterns
for what they are...
descriptions of
common problems


implement solutions that
take advantage of your tools

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

Execution in the Kingdom of Nouns Steve Yegge
http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html

Design Patterns in Groovy on groovy.codehaus.org
http://groovy.codehaus.org/Design+Patterns+with+Groovy

Design Patterns in Ruby
Russ Olsen