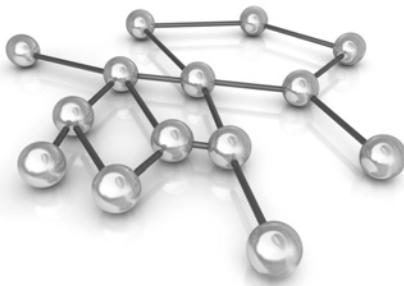


“design patterns” in ruby



NF

NEAL FORD software architect / meme wrangler

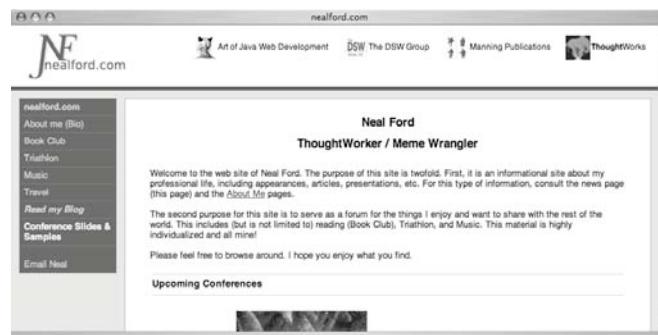
ThoughtWorks

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

housekeeping

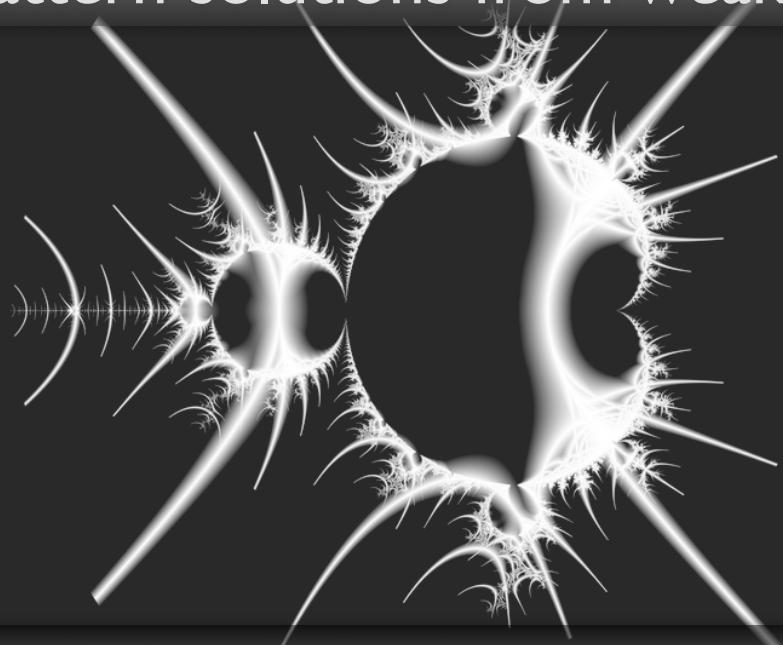
ask questions anytime

download slides from
nealford.com →



download samples from github.com/nealford

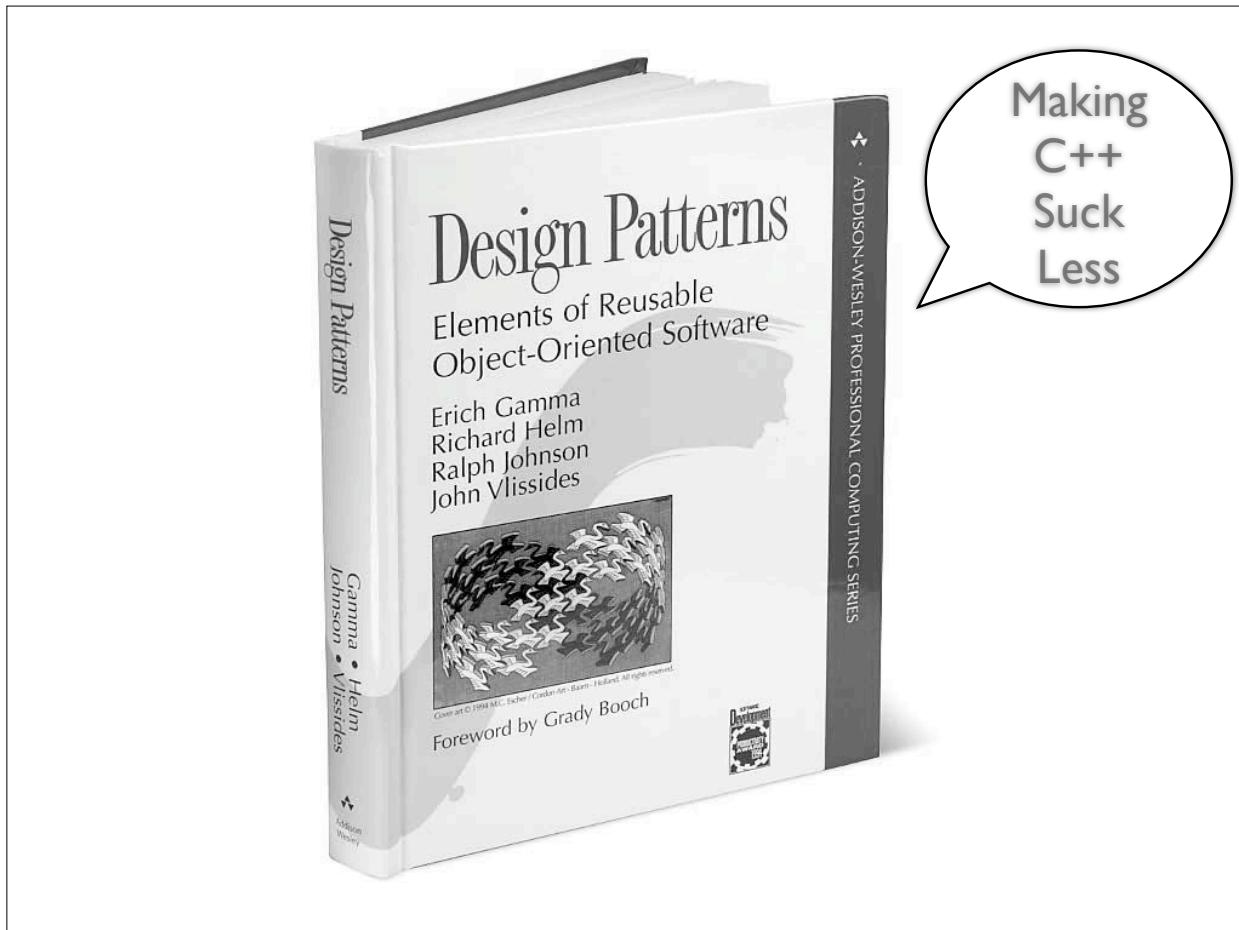
pattern solutions from weaker languages



have more elegant solutions



blended whisky
is made from
several
single malts



patterns define common
problems

dynamic languages give you
better tools to solve those
problems

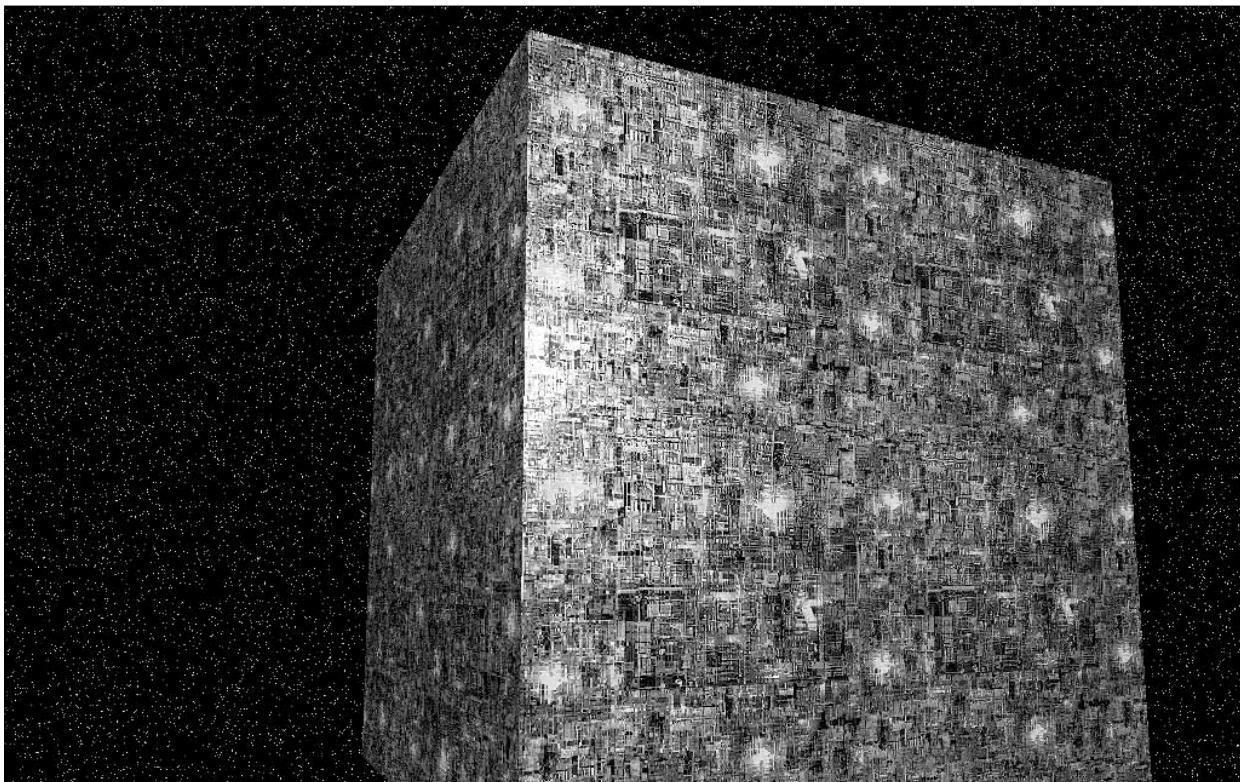


“Go To Statement Considered Harmful”

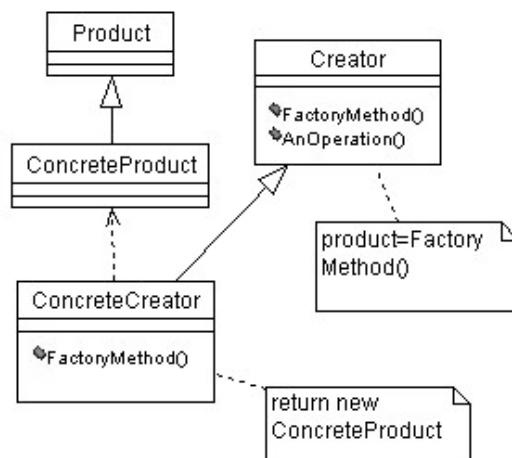
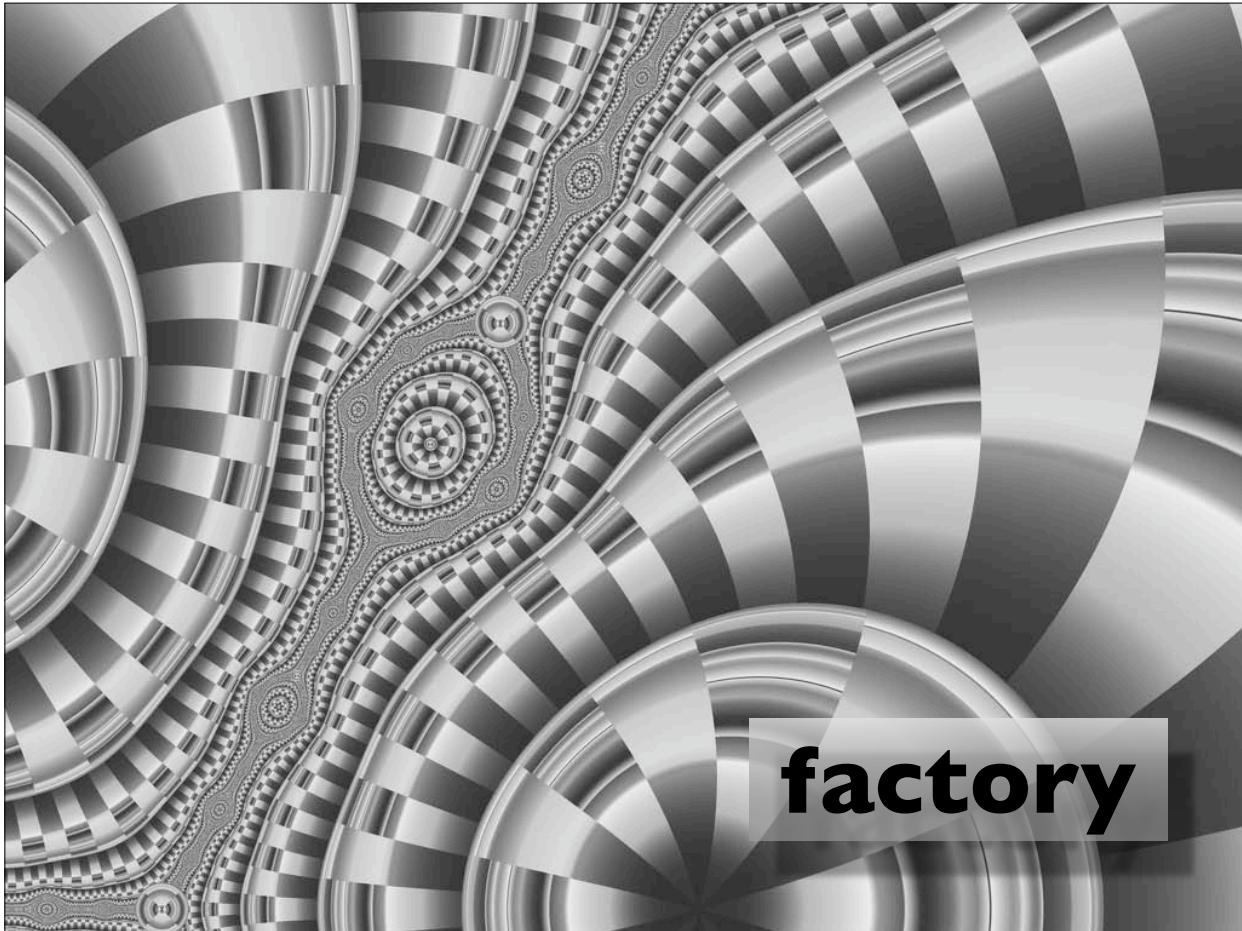
Edsger W. Dijkstra

Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.
Copyright © 1968, Association for Computing Machinery, Inc.

the gof patterns redux



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



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

construction isn't special

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

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

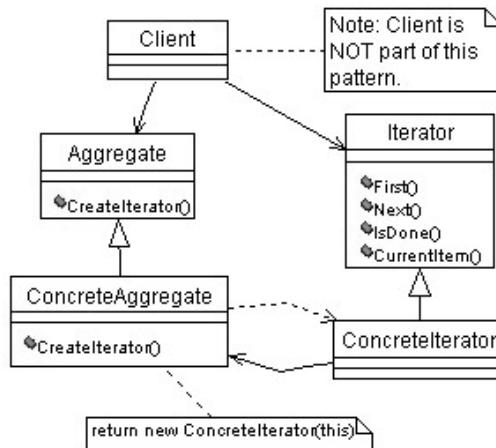
factory “design pattern”

```
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
```

iterator



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

```

def print_all_in_container
  output = ""
  for i in 0..container.length - 1 do
    output += "#{container[i]} "
  end
  output
end

def setup
  @it = IteratorDemo.new
  @numbers = [1, 2, 3, 4]
  @words = %w(first second third fourth)
end

def test_simple_iterator
  assert_equal("1 2 3 4 ",
    @it.print_all_in(@numbers))
  assert_equal("first second third fourth ",
    @it.print_all_in(@words))
end

```

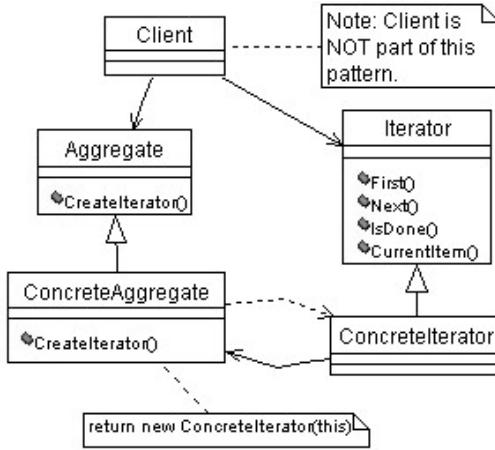
```

def print_all_with_internal_container
  output = ""
  container.each do |n|
    output += "#{n} "
  end
  output
end

def test_internal_iterator
  assert_equal("1 2 3 4 ",
    @it.print_all_with_internal(@numbers))
  assert_equal("first second third fourth ",
    @it.print_all_with_internal(@words))
end

```

ceremony



VS.

essence

```
numbers.each do |n|
  puts n
end
```

internal vs. external iterators

`.each` is an *internal* iterator

ruby 1.8 has no default syntax for external
iterators

ruby 1.9 adds *enumerators* on collections

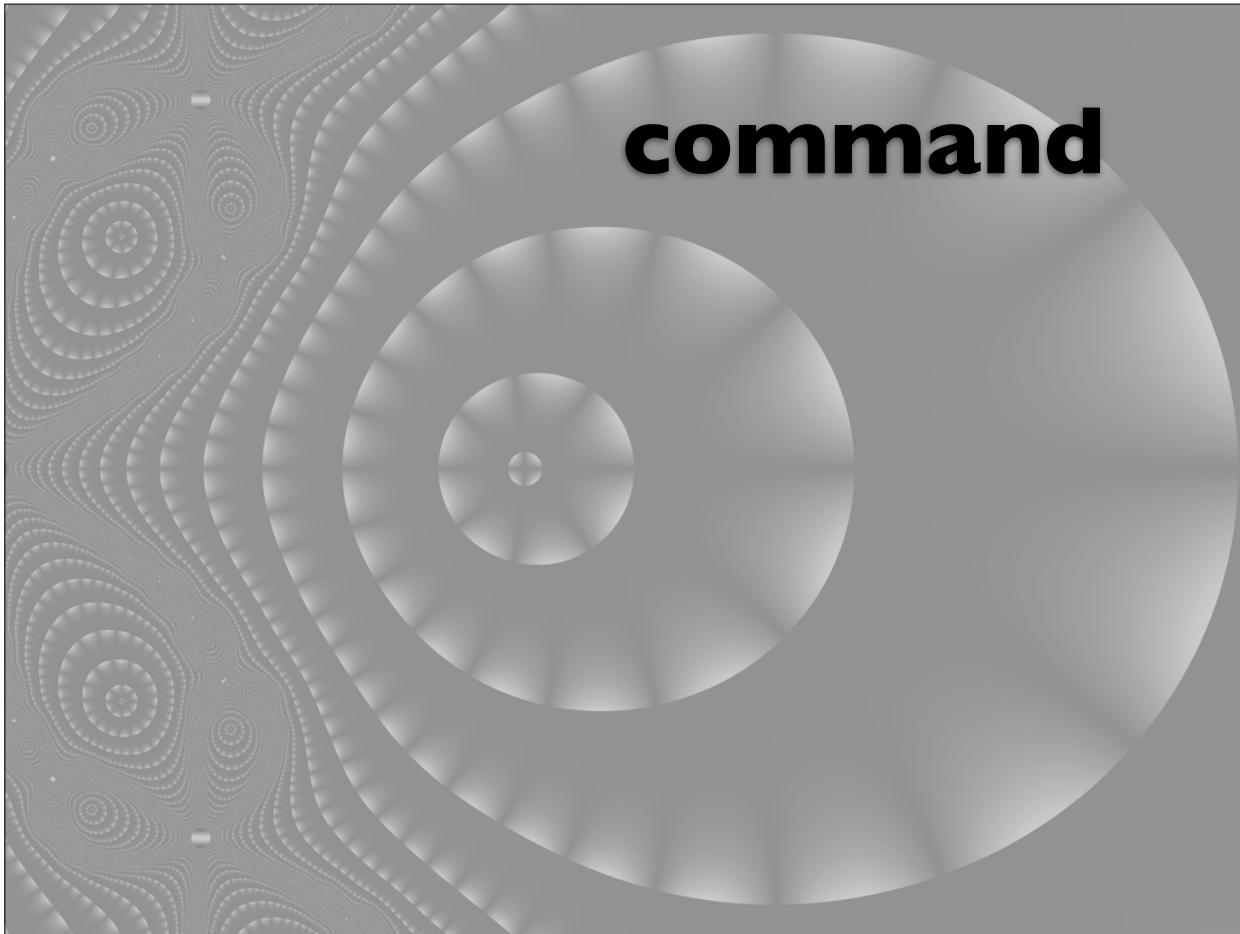
external iterator

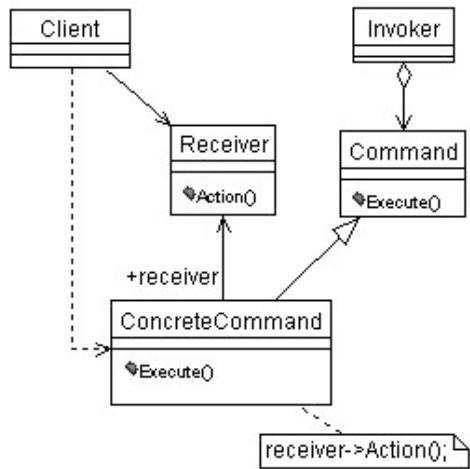
```
iterator = 9.downto(1)
loop do
  print iterator.next
end
puts "...blastoff!"
```



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.

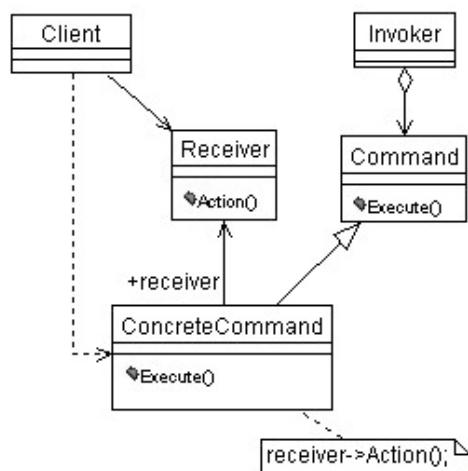
commands == closures

```

def test_counting
  count = 0
  commands = []
(1..10).each do |i|
  commands << proc { count += 1 }
end

assert_equal(0, count)
commands.each { |c| c.call }
assert_equal(10, count)
end

```



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

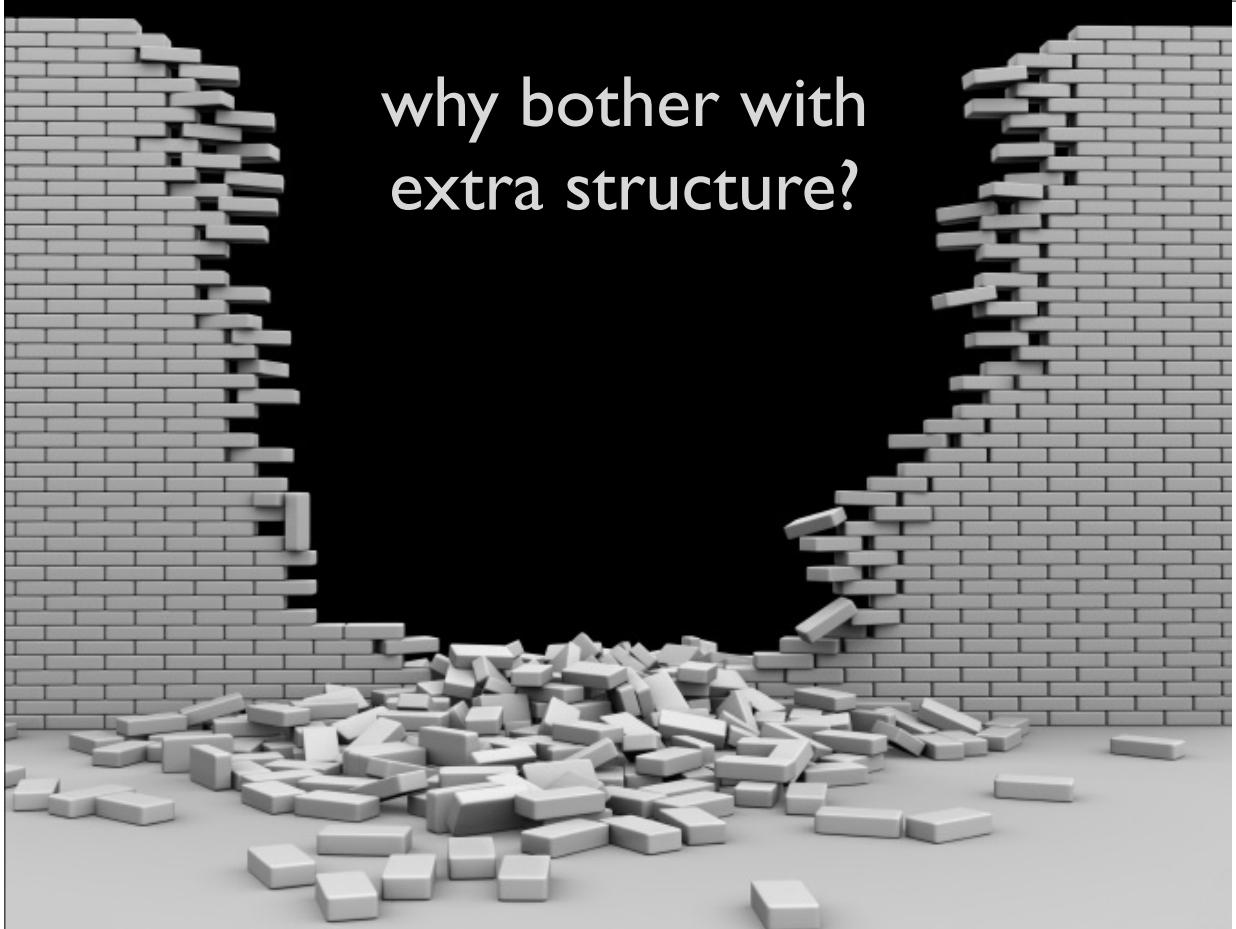
```
class Command
    def initialize(doit, undoit)
        @do = doit
        @undo = undoit
    end

    def do_command
        @do.call
    end

    def undo_command
        @undo.call
    end
end
```

```
def test_command_class
    count = 0
    commands = []
(1..10).each do |i|
    commands << Command.new(
        proc { count += 1 }, proc { count -= 1 } )
end

assert_equal(0, count)
commands.each { |cmd| cmd.do_command }
assert_equal(10, count)
commands.each { |cmd| cmd.undo_command}
assert_equal(0, count)
end
```



why bother with
extra structure?

*Execution in
the Kingdom
of Nouns*

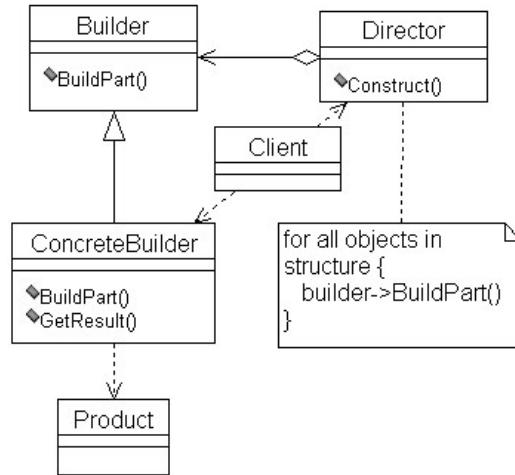
steve
yegge



command design pattern is
built into languages
with closures

add structure as needed

builder



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

“traditional” builder

```

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

```

```

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

```

```
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
```

```
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

```
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
```

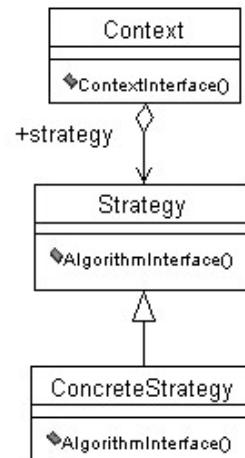
in computer_builder:

```
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
```

```
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
```



strategy



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

“traditional” strategy

```
module Strategy
  def execute; end
end

class CalcByMult
  include Strategy

  def execute(n, m)
    n * m
  end
end

class CalcByAdds
  include Strategy

  def execute(n, m)
    result = 0
    n.times do
      result += m
    end
    result
  end
end
```

```
class TestStrategyMult < Test::Unit::TestCase
  def setup
    @data = [
      [3, 4, 12],
      [5, -5, -25]
    ]
  end

  def test_as_strategies
    [CalcByMult.new, CalcByAdds.new].each do |s|
      @data.each do |l|
        assert_equal(l[2], s.execute(l[0], l[1]))
      end
    end
  end
end
```

better strategy for strategy

what if `execute` was `call` instead?

ruby already has a mechanism for passing around executable code

combination of the command & strategy patterns

```
class MultStrategies
  attr_reader :strategies

  def initialize()
    @strategies = [
      proc { |n, m| n * m}
    ]
  end

  def add_strategy(&block)
    @strategies << block
  end
end
```

```
def test_adding_strategy
  s = MultStrategies.new
  s.add_strategy do |n, m|
    result = 0
    n.times { result += m }
    result
  end
  s.strategies.each do |s|
    @data.each do |l|
      assert_equal(l[2], s.call(l[0], l[1]))
    end
  end
end
```

another strategy

dynamically applying the proper module based
on file types for image processing

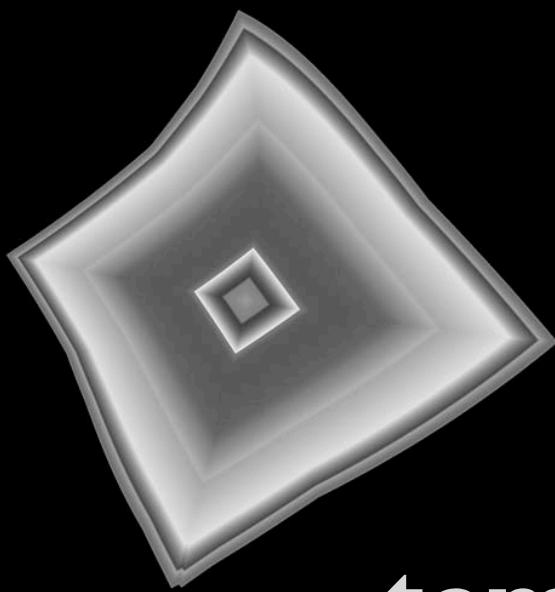
```
open(image_file) do |in|
  case identify_image_format(in)
  when "JPG"
    in.extend(JPGMethods) # readframe, readsof
  when "TIFF"
    in.extend(TIFFMethods) # readlong, readdir
  end
end
```

blatantly stolen from glenn vanderburg,
and all he got was this lousy acknowledgement

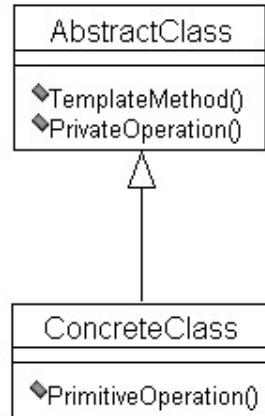


**mixins are a great way
to impose strategies
because they can
contain behavior**

ummmmm, sprinkles



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.

```

class Customer
  attr_accessor :plan

  def initialize
    @plan = []
  end

  def check_credit
    raise "Must override this method"
  end

  def check_inventory
    raise "Must override this method"
  end

  def ship
    raise "Must override this method"
  end

  def process
    check_credit
    check_inventory
    ship
  end
end

```

```

class UsCustomer < Customer
  def check_credit
    plan << "checking US customer credit"
  end

  def check_inventory
    plan << "checking US warehouses"
  end

  def ship
    plan << "Shipping to US address"
  end
end

class EuropeanCustomer < Customer
  def check_credit
    plan << "checking European customer credit"
  end

  def check_inventory
    plan << "checking European warehouses"
  end

  def ship
    plan << "Shipping to European address"
  end
end

```

```

class TestTemplate < Test::Unit::TestCase

  def test_template_methods_throw_exceptions_when_not_defined
    c = Customer.new
    assert_raise(RuntimeError) { c.check_credit }
    assert_raise(RuntimeError) { c.check_inventory }
    assert_raise(RuntimeError) { c.ship }
  end

  def test_customer
    c = UsCustomer.new
    c.process
    assert_equal c.plan[0], "checking US customer credit"
    assert_equal c.plan[1], "checking US warehouses"
    assert_equal c.plan[2], "Shipping to US address"
    e = EuropeanCustomer.new
    e.process
    assert_equal e.plan[0], "checking European customer credit"
    assert_equal e.plan[1], "checking European warehouses"
    assert_equal e.plan[2], "Shipping to European address"
  end
end

```

```

class UsCustomer < Customer
  def initialize
    super
    @check_credit = lambda { plan << "checking US customer credit" }
    @check_inventory = lambda { plan << "checking US warehouses" }
    @ship = lambda { plan << "Shipping to US address" }
  end
end

def test_customer_derivative
  c = UsCustomer.new
  c.process
  assert_equal c.plan[0], "checking US customer credit"
  assert_equal c.plan[1], "checking US warehouses"
  assert_equal c.plan[2], "Shipping to US address"
end

```

why abstract methods?

```

class Customer
  attr_accessor :plan, :check_credit, :check_inventory, :ship

  def initialize
    @plan = []
  end

  def process
    @check_credit.call
    @check_inventory.call
    @ship.call
  end
end

```

```
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
```

```
class UsCustomerOptional < CustomerWithOptionalTemplates
  def initialize
    super
    @check_credit = lambda { plan << "checking US customer credit" }
    @ship = lambda { plan << "Shipping to US address" }
  end
end

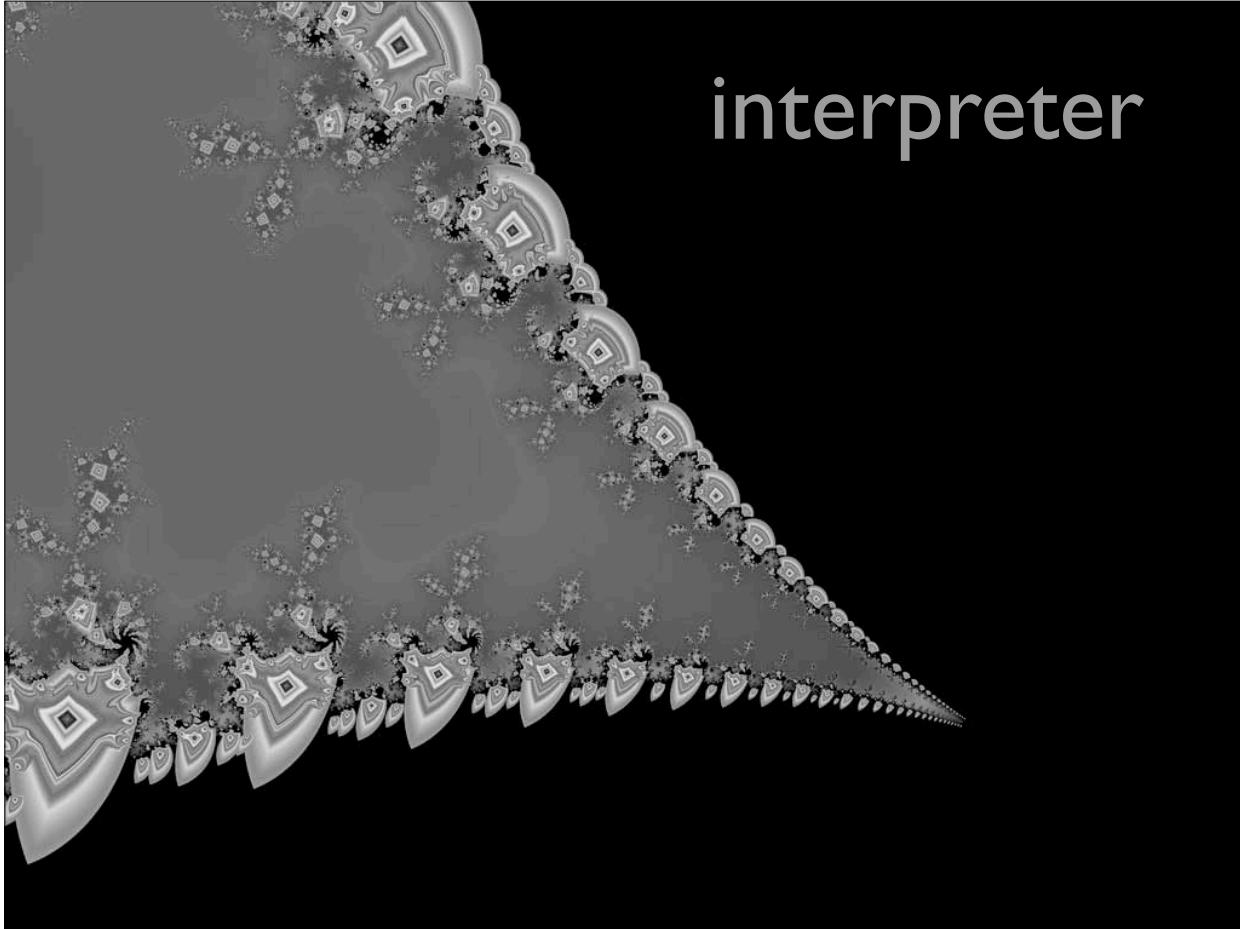
def test_customer_with_optional_templates
  c = UsCustomerOptional.new
  c.process
  assert_equal c.plan[0], "checking US customer credit"
  assert_equal c.plan[1], "Shipping to US address"
end
```

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

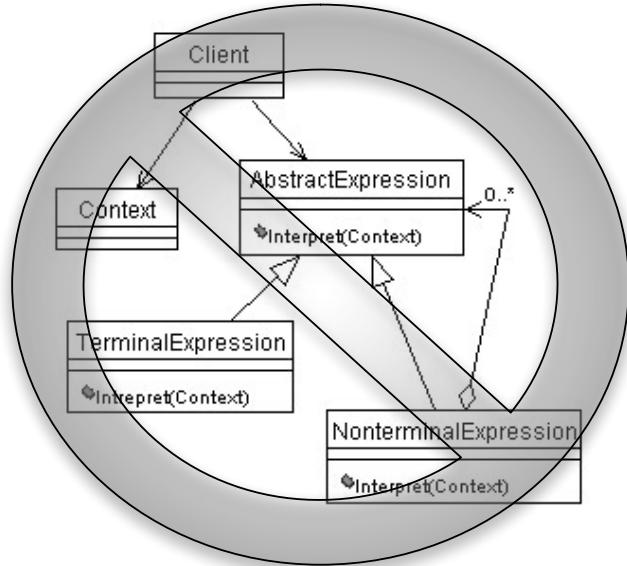
  def initialize
    @plan = []
  end

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

  def process_succinct
    [@check_credit, @check_inventory, @ship].each do |p|
      p.call unless p.nil?
    end
  end
end
```



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

```

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

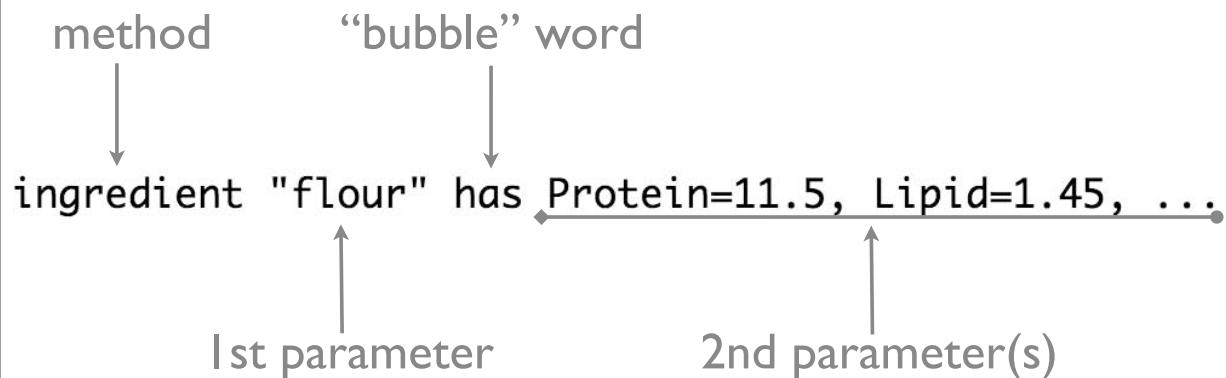
  def initialize(name, protein=0, lipid=0, sugars=0, calcium=0, sodium=0)
    @name = name
    @protein, @lipid, @sugars = protein, lipid, sugars
    @calcium, @sodium = calcium, sodium
  end

  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?



```

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

```

```

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

```

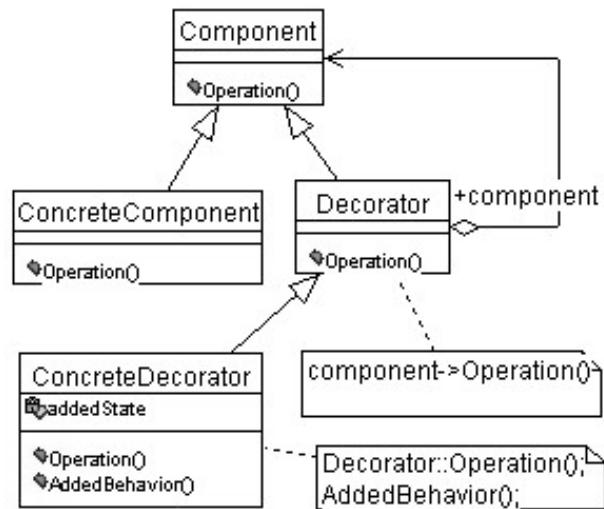
```
def polish_text(definition_line)
  polished_text = definition_line.clone
  polished_text.gsub!(/=/, '=>')
  polished_text.sub!(/and /, '')
  polished_text.sub!(/has /, ',')
  polished_text
end
```

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

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

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

internal dsl's == embedded interpreter



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

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

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

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

```
class Milk
  include Decorator

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

```
class Whip
  include Decorator

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

```
class Sprinkles
  include Decorator

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

```
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
```

```
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
```

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



```
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
```

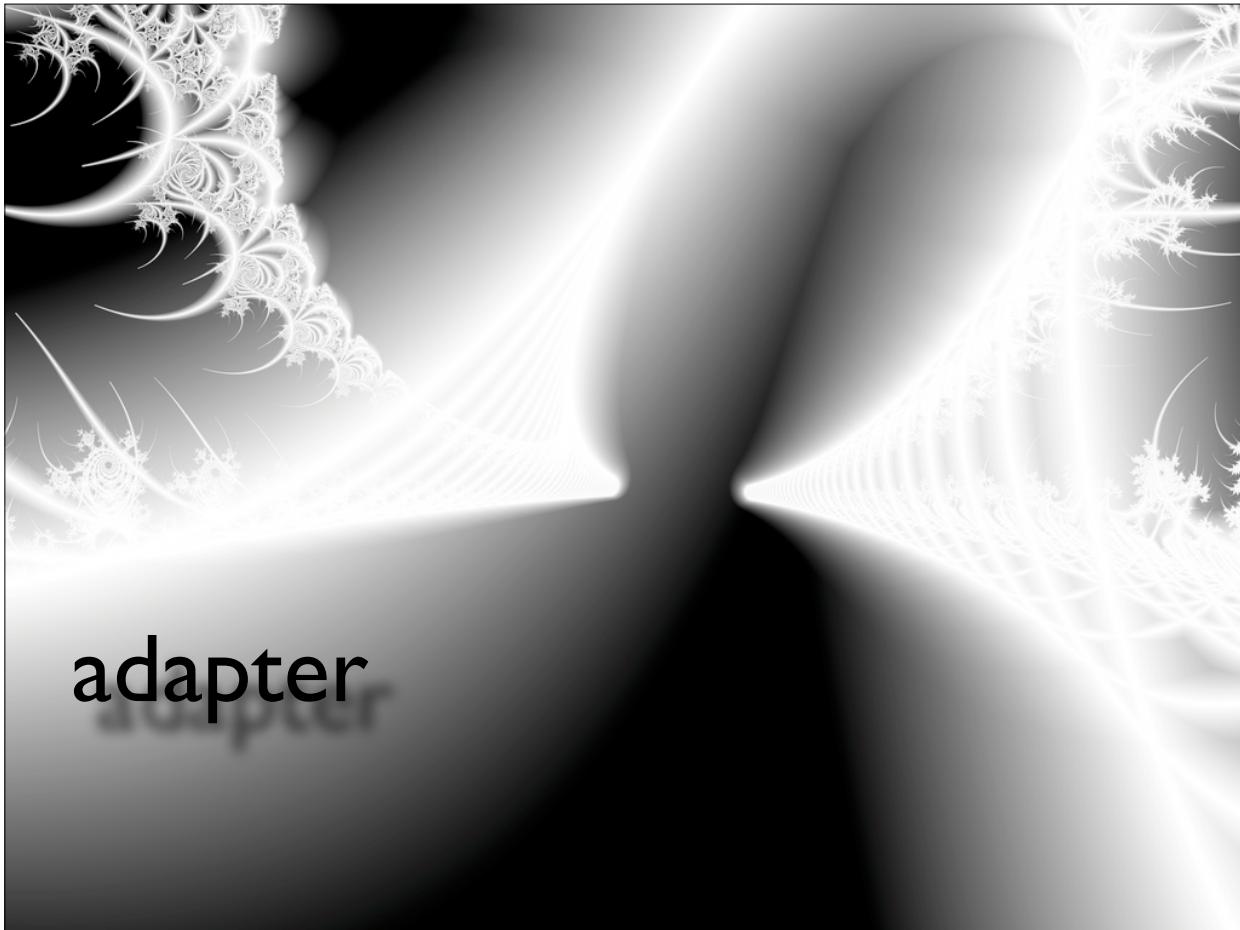
```
class Recorder
  def initialize
    @messages = []
  end

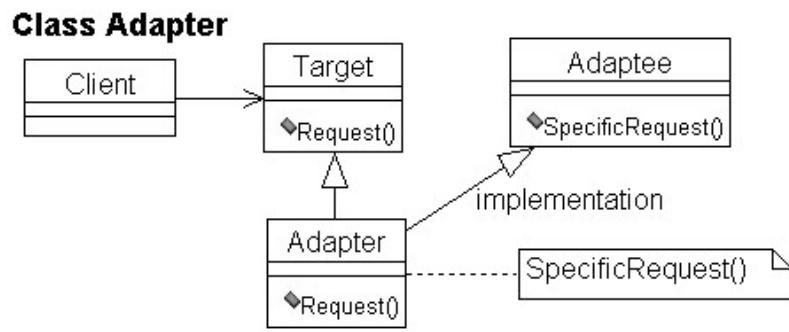
  def method_missing(method, *args, &block)
    @messages << [method, args, block]
  end

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

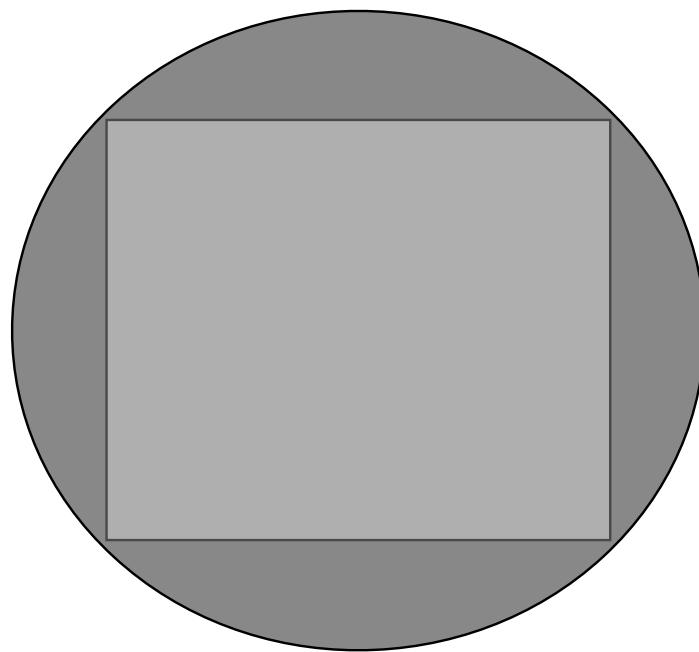
```
def test_recorder
  r = Recorder.new
  r.sub!(/Java/) { "Ruby" }
  r.upcase!
  r[11, 5] = "Universe"
  r << "!"

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





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
```

```
class RoundHole
    attr_reader :radius

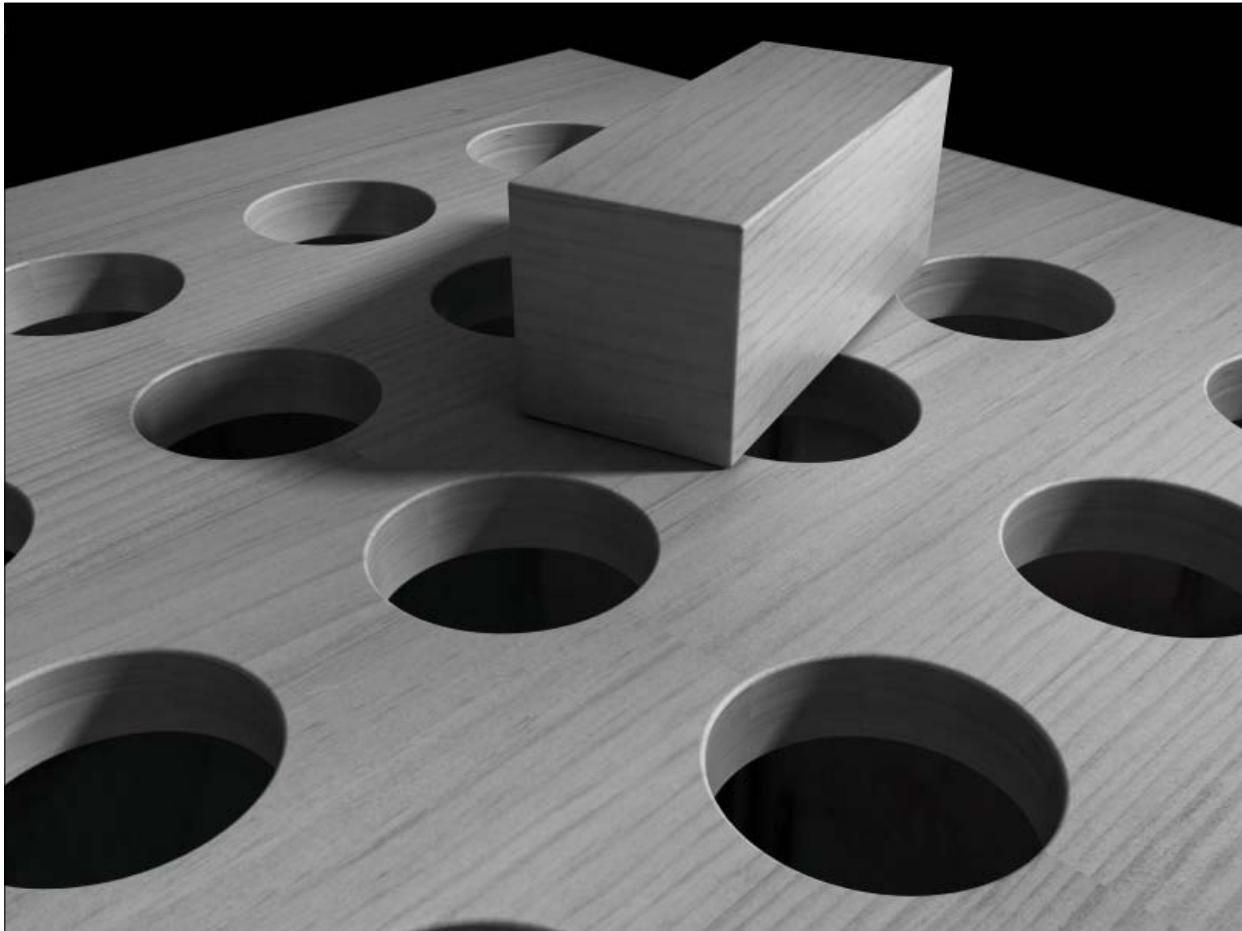
    def initialize(r)
        @radius = r
    end

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

```
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?



```
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

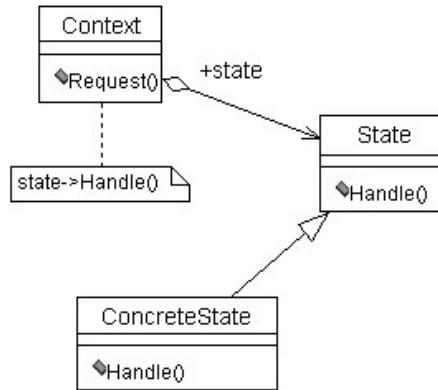
```
class Class
  def def_interface(interface, *syms)
    @_interface_ ||= {}
    a = (@_interface_[interface] ||= [])
    syms.each do |s|
      a << s unless a.include? s
      alias_method "__#{s}__#{interface}__".intern, s
      remove_method s
    end
  end
end
```

```
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**





Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

mixology?

a gem developed during thoughtworks project
work by:

Pat Farley, anonymous z, Dan Manges, Clint Bishop

ruby allows you to mix-in behavior

mixology allows you to easily unmix behavior

state pattern on steroids

```

class Door

  def initialize(open = false)
    if open
      extend Open
    else
      extend Closed
    end

    def closed?
      kind_of? Closed
    end

    def opened?
      kind_of? Open
    end

  end

  module Closed
    def knock
      puts "knock, knock"
    end

    def open
      extend Open
    end
  end

  module Open
    def knock
      raise "just come on in"
    end

    def close
      extend Closed
    end
  end
end

```

```

class DoorTest < Test::Unit::TestCase
  def test_an_open_door_is_opened_and_not_closed
    door = Door.new :open
    assert door.opened?
    assert !door.closed?
  end

  def test_a_closed_door_is_closed_and_not_opened
    door = Door.new
    assert door.closed?
    assert !door.opened?
  end

  def test_closing_an_open_door_makes_the_door_closed_but_not_opened
    door = Door.new :open
    door.close
    assert door.closed?
    assert !door.opened?
  end

  def test_opening_a_closed_door_makes_the_door_opened_but_not_closed
    door = Door.new
    door.open
    assert door.opened?
    assert !door.closed?
  end
end

```

```

Loaded suite door
Started
..FF
Finished in 0.006623 seconds.

1) Failure:
test_closing_an_open_door_makes_the_door_closed_but_not_opened(DoorTest)
[door.rb:67]:
is not true.

2) Failure:
test_opening_a_closed_door_makes_the_door_opened_but_not_closed(DoorTest)
[door.rb:74]:
is not true.

4 tests, 8 assertions, 2 failures, 0 errors

```

state transitions fail because the old
mixin is still mixed in

```

class Door
  def initialize(open = false)
    @open = open
    if open
      mixin Open
    else
      mixin Closed
    end

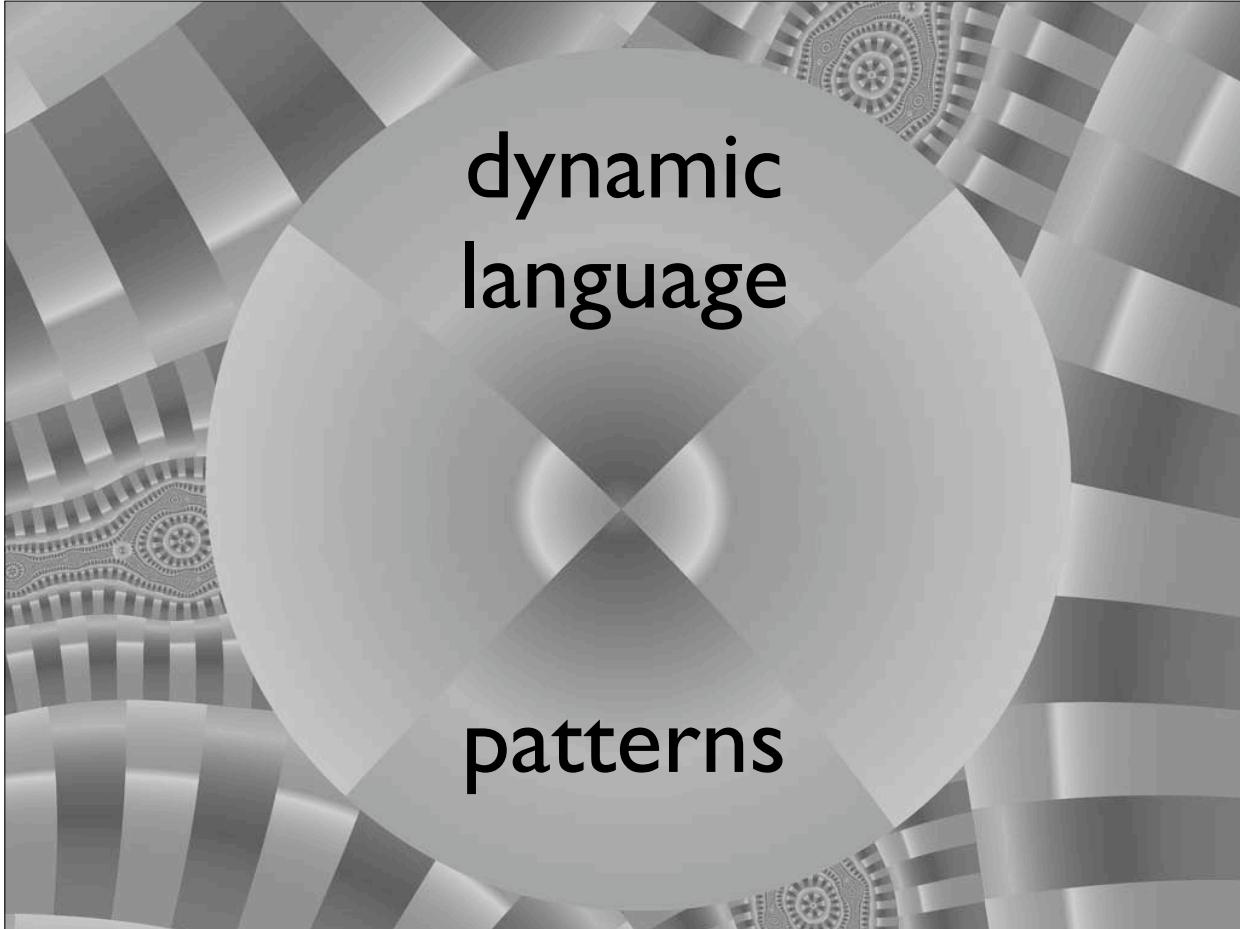
    def closed?
      kind_of? Closed
    end

    def opened?
      kind_of? Open
    end
  end

  module Closed
    def open
      unmix Closed
      mixin Open
    end
  end

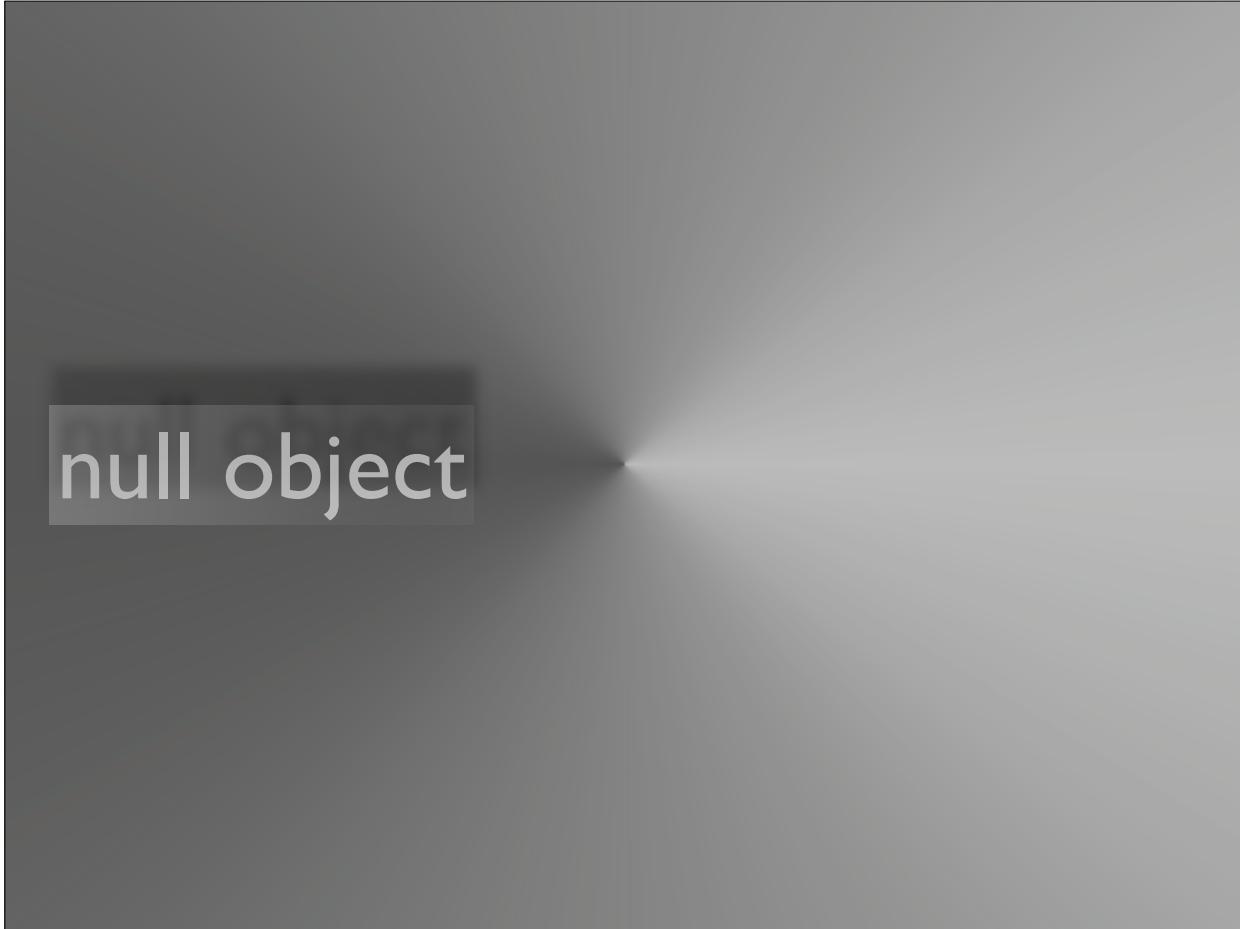
  module Open
    def close
      unmix Open
      mixin Closed
    end
  end
end

```

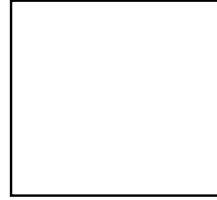


dynamic
language

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.

this pattern doesn't
exist in ruby

NilClass is already defined

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

def initialize
  @plan = []
end

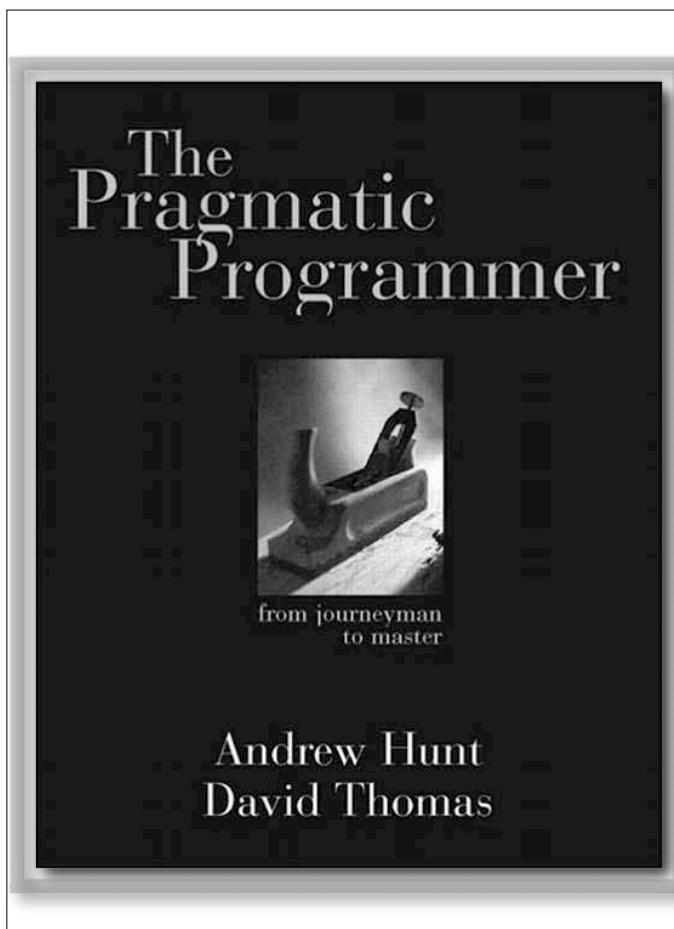
def process
  @check_credit.call unless @check_credit.nil?
  @check_inventory.call unless @check_inventory.nil?
  @ship.call unless @ship.nil?
end

end
```

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



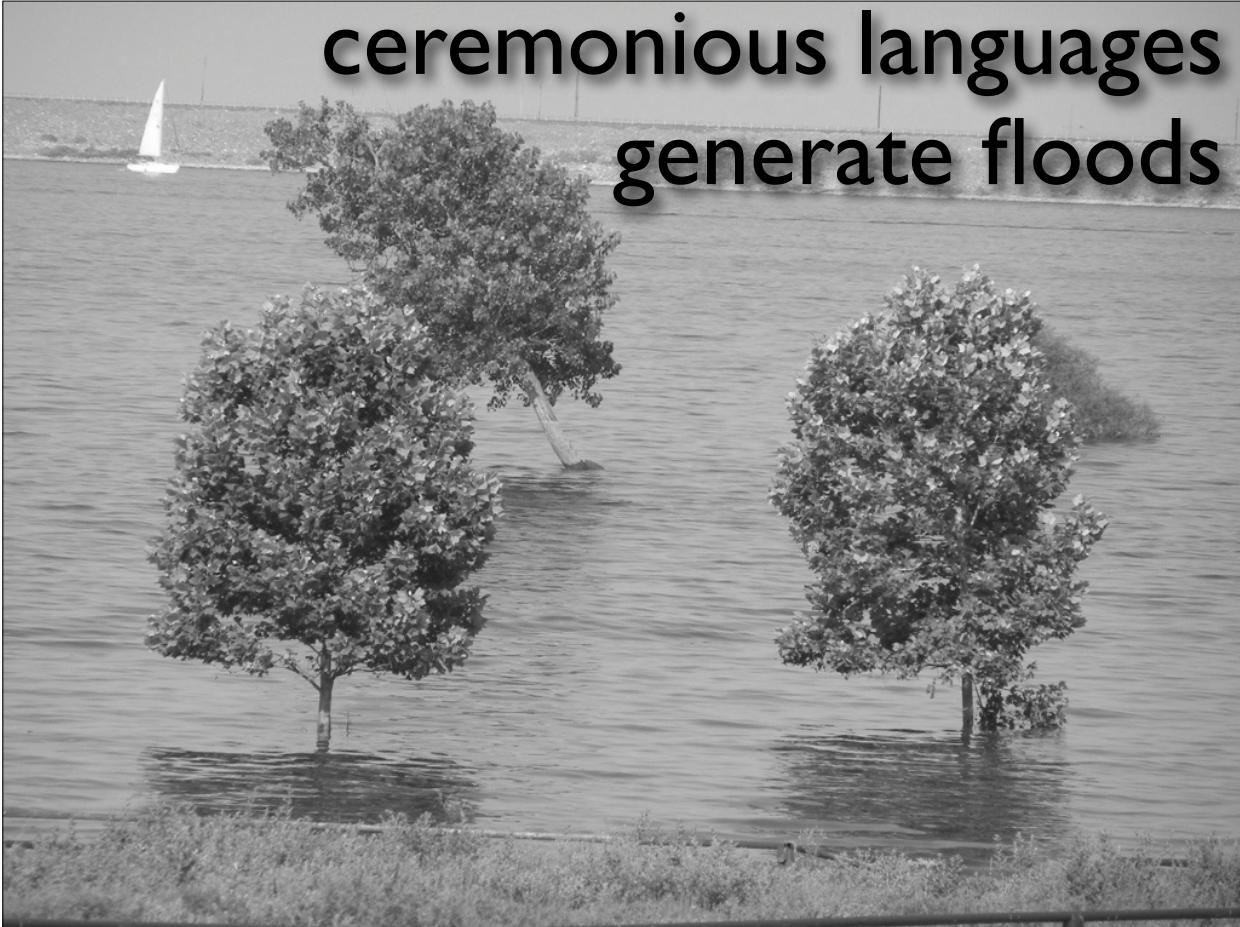
aridifier



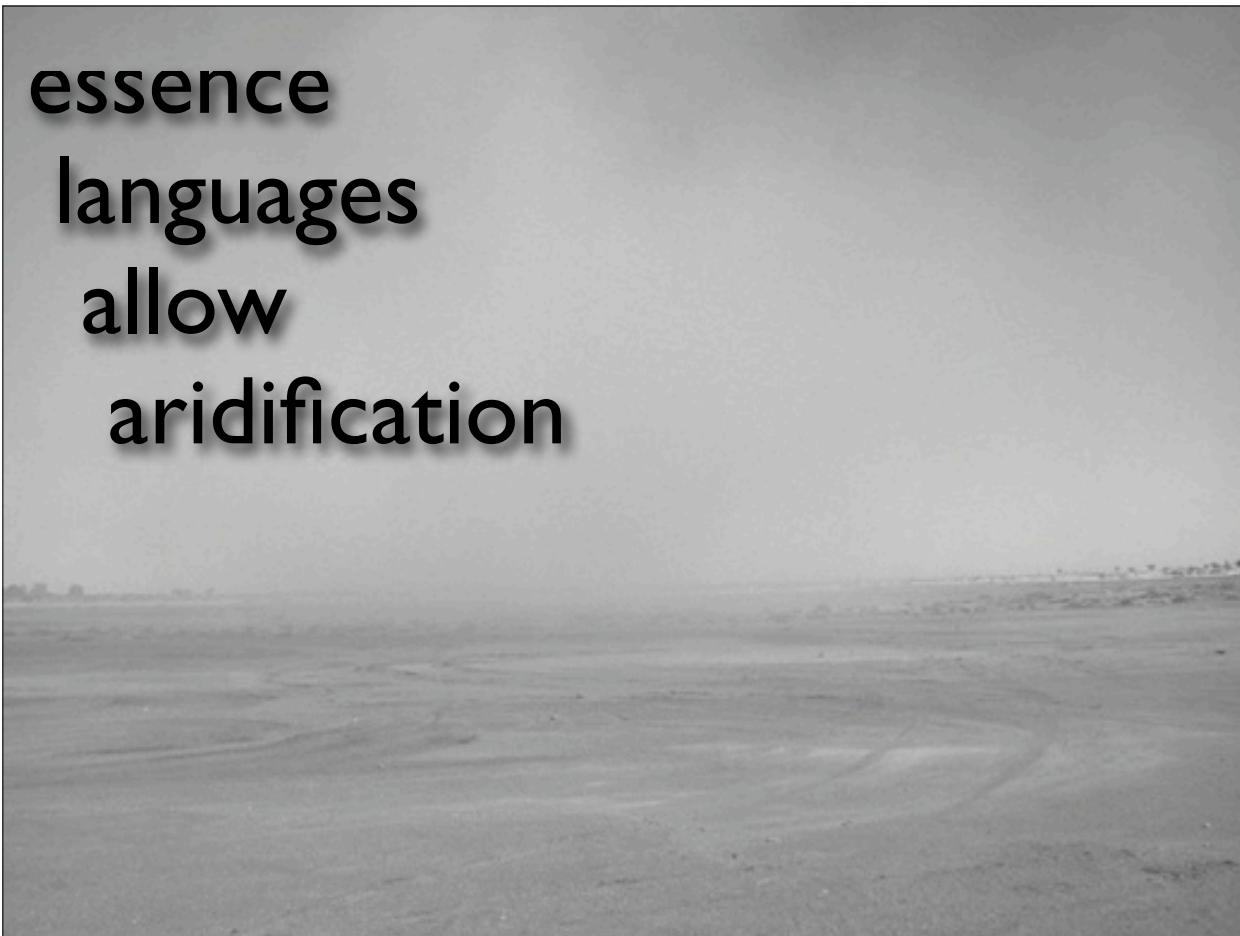
d r y

don't
repeat
yourself

**ceremonious languages
generate floods**



**essence
languages
allow
aridification**



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

```

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?

```

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

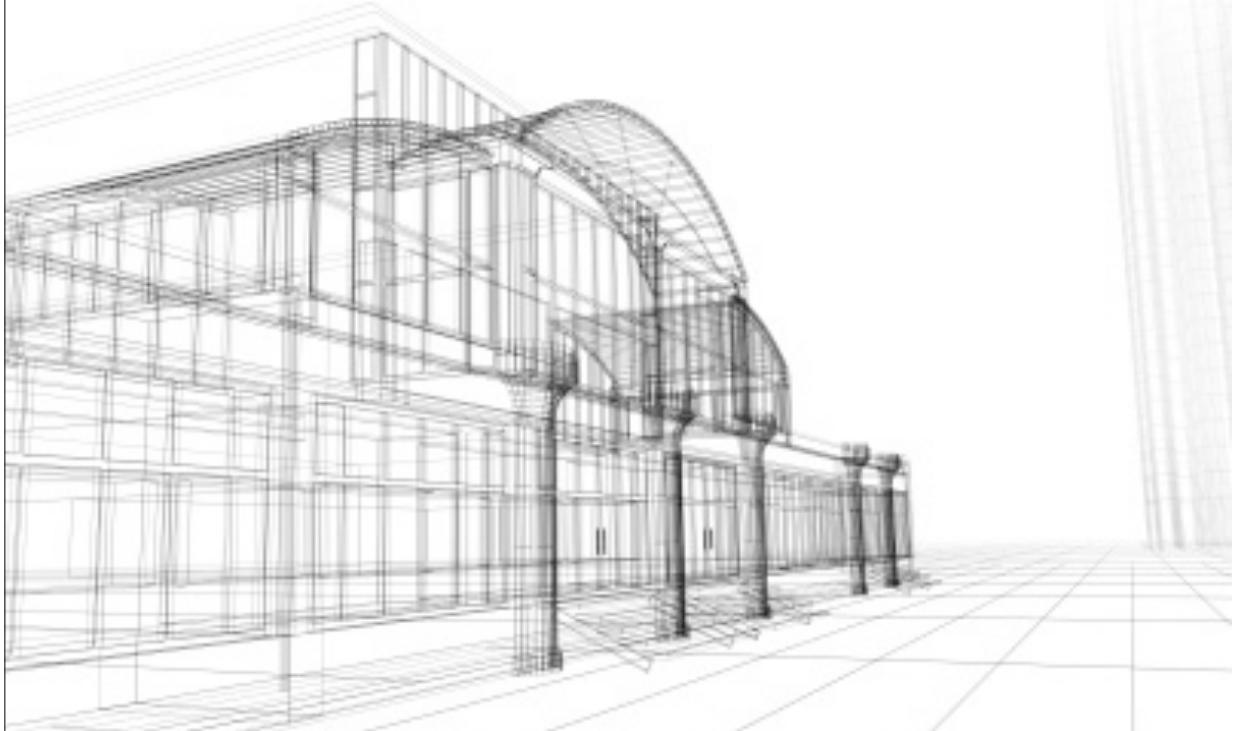
def test_delegating_to_a_sized_queue
  a = SizedQueue.new(12)
  q = FQueue.new a
  q.enqueue "one"
  assert_equal 1, q.size
  assert_equal "one", q.dequeue
end

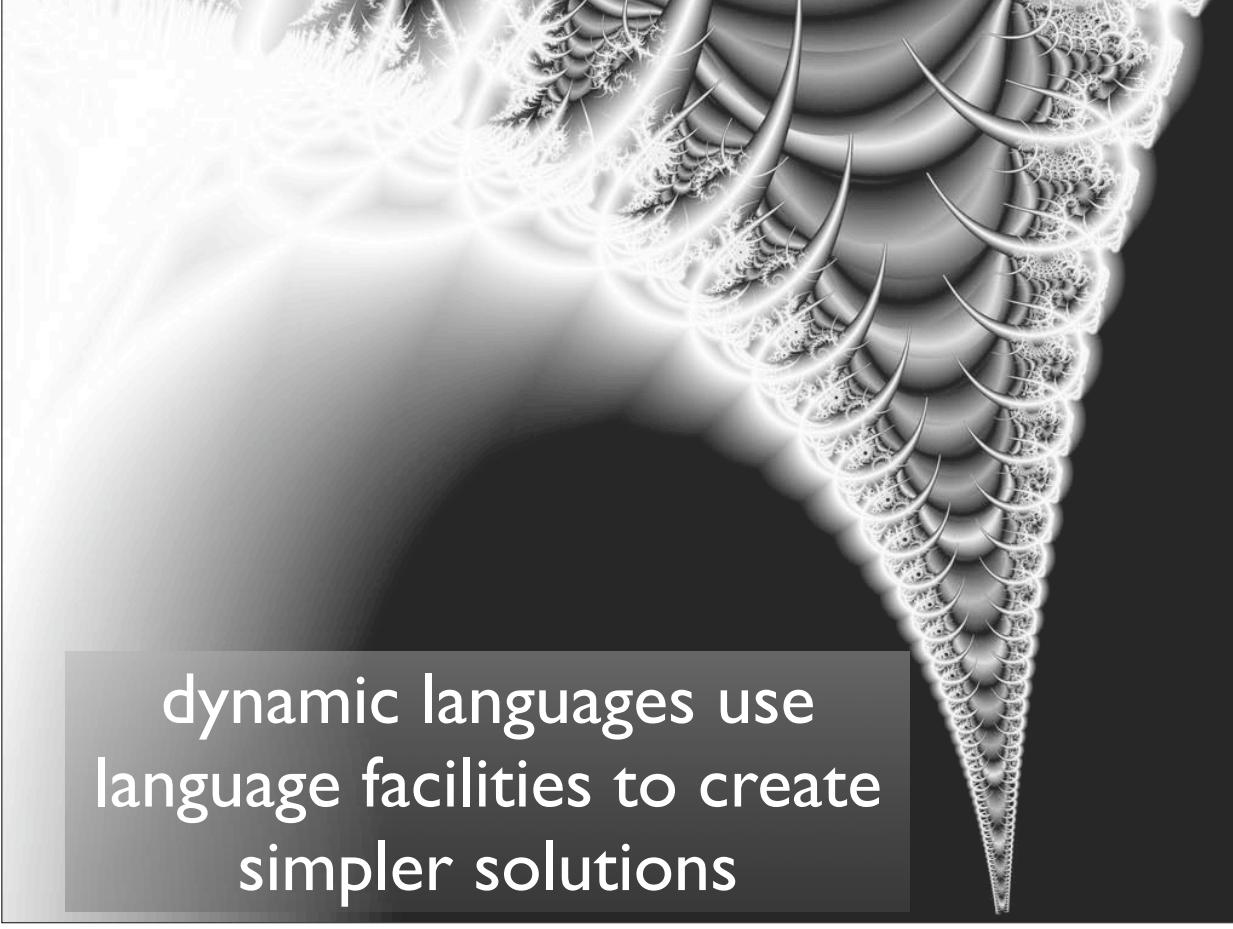
```

drier tests

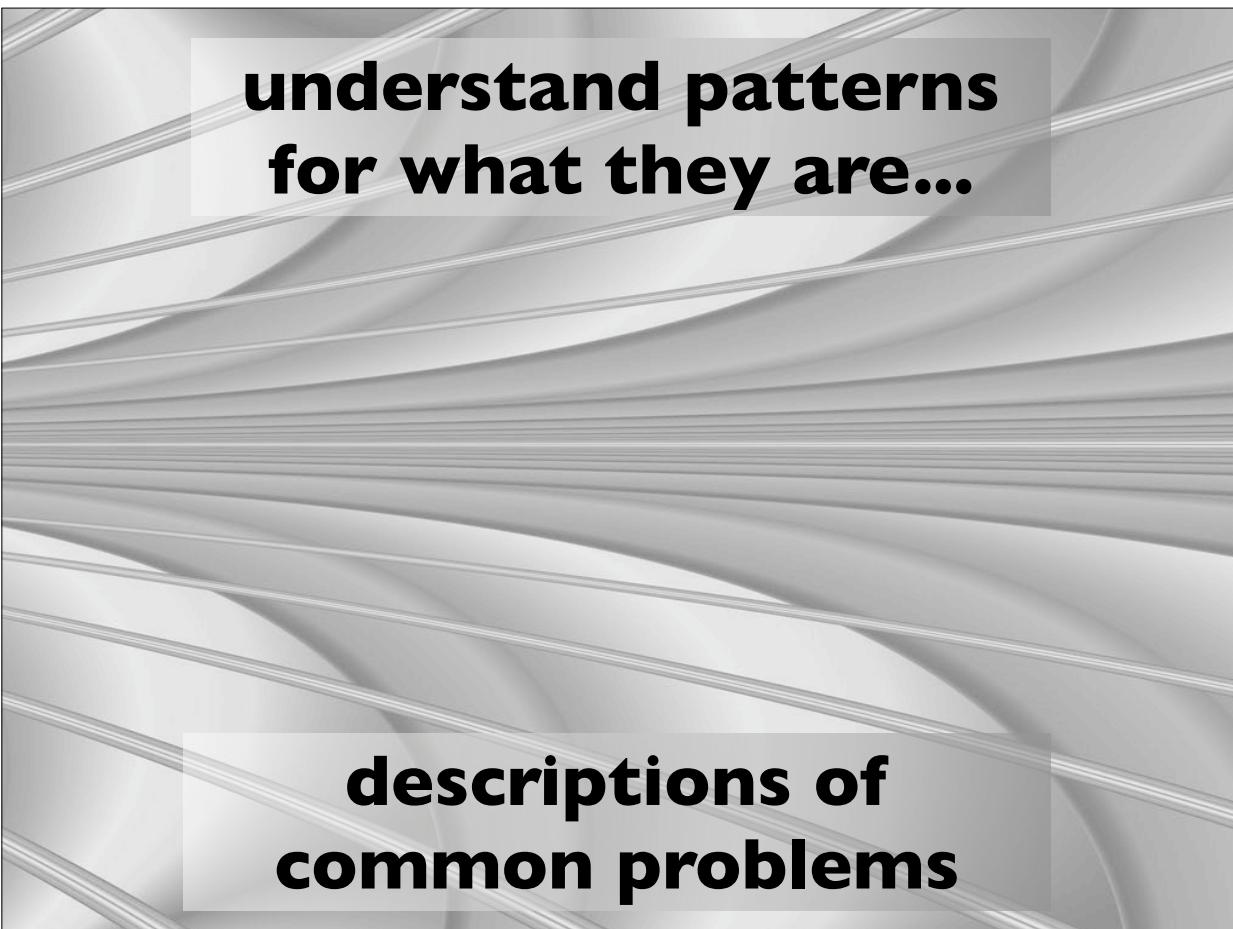
```
TestForwardQueue.class_eval do
  [Array, Queue, SizedQueue].each do |c|
    method_name = "test_queue_delegated_to_" + c.to_s
    define_method method_name do
      a = c == SizedQueue ? c.new(12) : c.new
      q = FQueue.new a
      q.enqueue "one"
      assert_equal 1, q.size
      assert_equal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

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



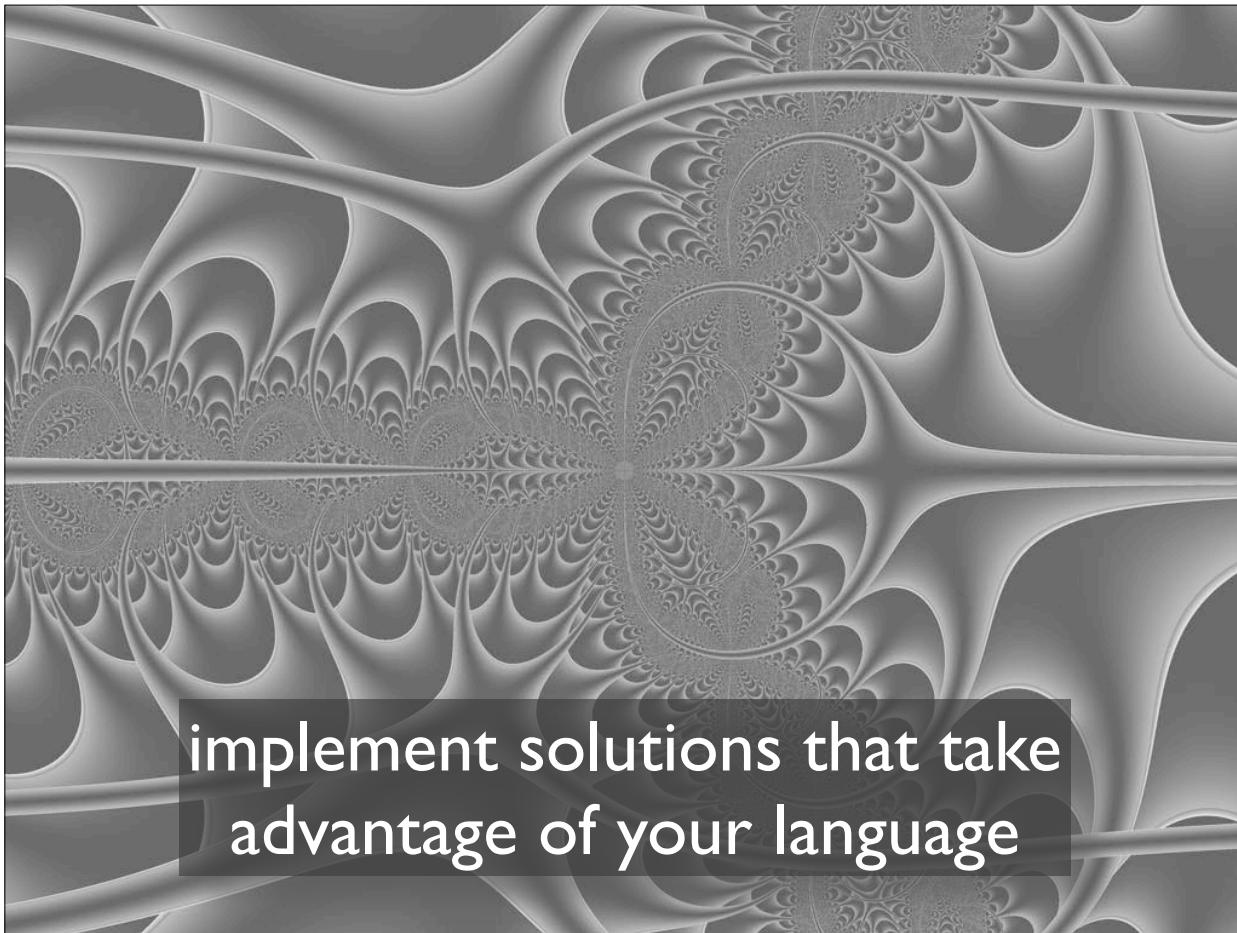


**dynamic languages use
language facilities to create
simpler solutions**



**understand patterns
for what they are...**

**descriptions of
common problems**



implement solutions that take
advantage of your language

ThoughtWorks

? ' S

please fill out the session evaluations
samples at github.com/nealford



This work is licensed under the Creative Commons
Attribution-Share Alike 3.0 License.

<http://creativecommons.org/licenses/by-sa/3.0/us/>

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

N

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

NF