Dr. Dobb's ARCHITECTURE & DESIGN WORLD
JULY 21-24, 2008 ■ CHICAGO, IL
HYATT REGENCY MCCORMICK PLACE
WWW.DRDOBBSARCHWORLD.COM
DESIGN, MODEL AND BUILD

Principles of Ruby Application Design

Dean Wampler
Senior Mentor and Consultant
Object Mentor, Inc.
Chicago, IL
dean@objectmentor.com

Get the latest version of this talk:

`aspectprogramming.com/papers`

Building an

application architecture

requires a good

foundation...

11

Principles of OOD

#1

Single Responsibility Principle

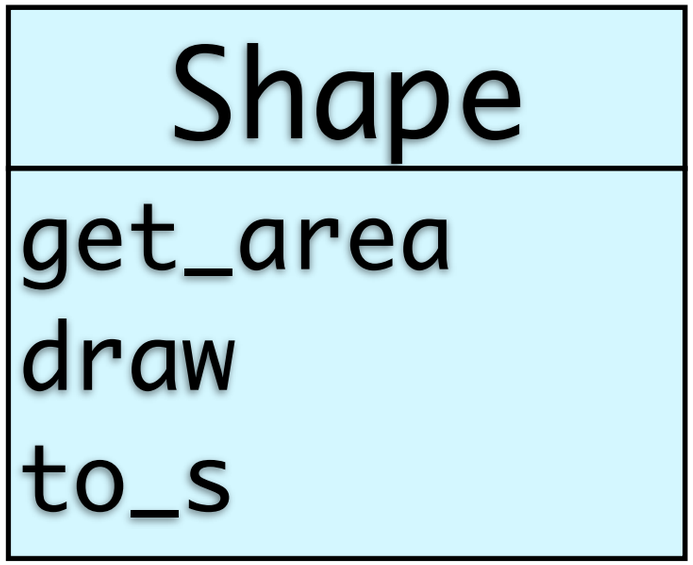
A class should have only one reason to change.

#2

Open-Closed Principle

A module should be open for extension, but closed for modification.

An Example...



Gratuitous UML...

If this were Java, we would need a base class or interface...

??

```
module Shapes
  class Shape
    def draw
      raise NotImplemented
    end

    ...
  end
end
```

Yuck..

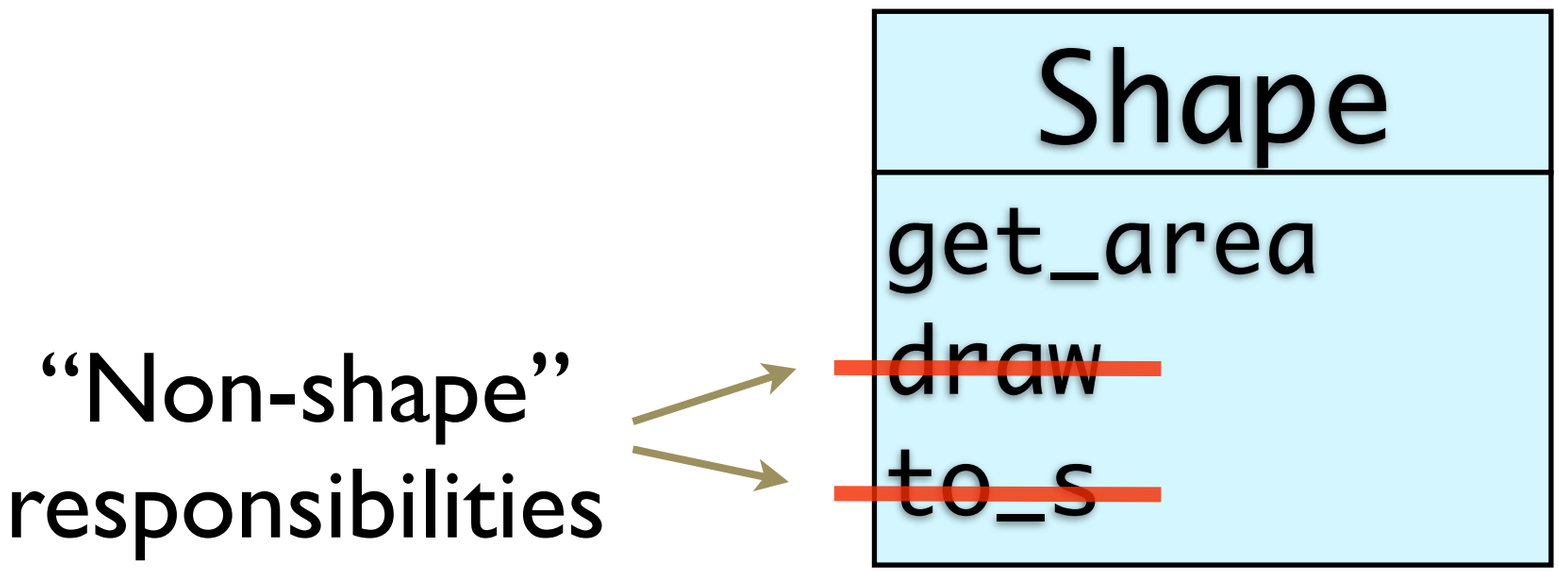
```
module Shapes
  class Polygon
    def get_vertex index
      …
    end
    def draw; …; end
    def to_s; …; end
  end
end
```

No `get_area` *yet...*

```
module Shapes
  class Rectangle < Polygon
    def get_area; …; end
  end
  class Square < Polygon
    def get_area; …; end
  end
end
```

Square *is a* Rectangle?

A Refactoring…



```
class Polygon
  def getVertex index
    …
  end
  def draw; …; end
  def to_s; …; end
end
```

reusable module

"abstract" method to make command string

```
module Shapes
  module Drawing
    def draw
      command = make_command
      render command
    end
    def render command; …; end
  end
end
```

The Template Method Pattern

```ruby
# shapes_drawing.rb
module Shapes
  class Polygon
    include Drawing
    def make_command; …; end
  end
  class Circle
    include Drawing
    def make_command; …; end
  end
end
```

Just reopen the classes.

use the module

```ruby
# shapes_to_s.rb
module Shapes
  class Polygon
    def to_s; …; end
  end

  …
end
```

Reopen the classes.

Exploits Duck Typing

Reopening Types to Separate Responsibilities

```
module Shapes
  class Polygon
    include Drawing
    ...
```

Separate Files Supports OCP

```
# shapes.rb
# shapes_to_s.rb
# shapes_drawing.rb
```

#2

Open-Closed Principle II

A module should be open for extension, but closed for ***source and contract*** *modification.*

But, is it bad to spread responsibilities across files?

Modules as Traits

Traits: *Composable Units of Behavior*

Analogous to Mixins

http://www.iam.unibe.ch/~scg/Research/Traits/

Traits

1. Primitive units of behavior
2. Classes composed of traits
3. Not inheritance based
4. Better for SRP & OCP

Ruby Modules work in a similar way.

What if a trait is needed temporarily?

What is a Person?

```
class Person
 attr_accessor :name, :address, \
  :salary, :pension, :insurance,\
  :medical_history, ???
  ...
end
```

Sometimes,

Person is an Employee.

At other times,

Person is a Patient, ...

We could make Person have everything...

- *Leads to bloat*
- *Breaks SRP and maybe OCP*
- *Cumbersome to maintain*
+ *Domain model: One Person*

We could have separate Person variants

- *Leads to boilerplate mapping between types*
- *Or deep hierarchies*
+ *Better for SRP & OCP*

I would rather

include modules (traits)

on demand,

and then remove them.

Unfortunately, you can't remove modules in Ruby...

We'll come back to this...

Another Example:

Observer Design Pattern

Observe Drawing

```
module Shapes
  module Drawing
    include Subject
      alias_method :draw1, :draw
      def draw
        draw1; notify
      end
end; end
```

Reopen

Subject part of pattern

"Wrap" the method

```
module Subject
  def notify
    @observers.each do |o|
      o.receive_update self
    end
  end
  def register observer
    @observers ||= []
    @observers << observer
end; end
```

"Implicit" Observer Abstraction

No Observer module is needed, but @observers must respond to receive_update.

What if I want to observe all calls to get_area in all Shapes?

How do I observe all Shape types?

metaprogramming

Assume we have a Shape class or module!

```
descendents_of(Shape).each do |s|
  s.class_eval do
    alias_method :area1, :get_area
    def get_area
      area1
      notify
    end
  end
end
```

```
def descendents_of t
  Module.constants.map do |const|
    Module.class_eval const
  end.find_all do |t2|
    t2.respond_to?(:ancestors) and
    t2.ancestors.include?(t)
  end
end
```

Note: doesn't handle nested types

Simpler alternative:

Aquarium

aquarium.rubyforge.org

```
require “aquarium”
module GetAreaObserver
 include Aquarium::DSL
  after :calls_to => :get_area,
  :on_type_and_descendents_of => Shape \
    do |context, object|
      object.notify
  end
end
```

You can also use

Aquarium to dynamically include the Subject module in the Shape types.

```
require “aquarium”
include Aquarium::Finders
TypeFinder.find(
  :on_type_and_descendents_of =>
    Shape).each do |type|
      type.send :include, Subject
end
```

Aquarium
simplifies many
metaprogramming
techniques.

Aquarium

supports

Aspect-Oriented

Programming.

Back to Interfaces...

How do we document the "Observer" abstraction?

How do we document the metaprogramming stuff?

Automated tests are even more important for dynamically-typed languages like Ruby.

Automated tests tell you what the code is really doing.

#3

Liskov Substitution Principle

Subtypes must be substitutable for their base types.

Square *is a* Rectangle?

What do the RSpec examples *(unit tests) say?*

```
describe Shapes, “#draw” do
  it “should work for any Shape” do
    shapes = [Square.new,
              Rectangle.new]
    shapes.each do |s|
      lambda { s.draw }.
        should_not raise(…)
    end
  end
end
```

Works fine…

Square *is a* Rectangle?

Apparently, it is!
(at least, in this context…)

Mutability

What if…

```
module Shapes
  class Rectangle
    attr_accessor :width, :height
  end

  class Square < Rectangle; end
end
```

```
describe Rectangle, “height” do
  it “should be indep. of width” do
    [Rectangle.new(2,2),
     Square.new(2,2)].each do |s|
      old_width = s.width
      s.height = 2 * s.height
      s.width.should == old_width
    end
  end
end
```

Does this pass??

Square *is a* Rectangle?

Not in the context of mutability

LSP *is*

context dependent!

LSP implies a contract involving:

1. *Parent type*
2. *Child types*
3. *Included modules*
4. *Clients*

Mutability vs. Immutability: Functional Programming

Functional Programming

Application of functions vs. imperatively changing state

Characteristics

1. *No side-effects*
2. *Declarative*
3. *Composition of functions*
4. . . .

Side-effect free:

programs are easier to

scale and develop.

Check out Erlang

Declarative Programs

1. *Less code*
2. *More behind-the-scenes implementation options.*

Rails: ActiveRecord

```
class Picture < ActiveRecord::Base
  belongs_to :portfolio
  has_many   :shapes

  ...
end
```

We declare what we want,
not how to do it.

Composition of functions

improves modularity and reduces code bloat.

Blocks

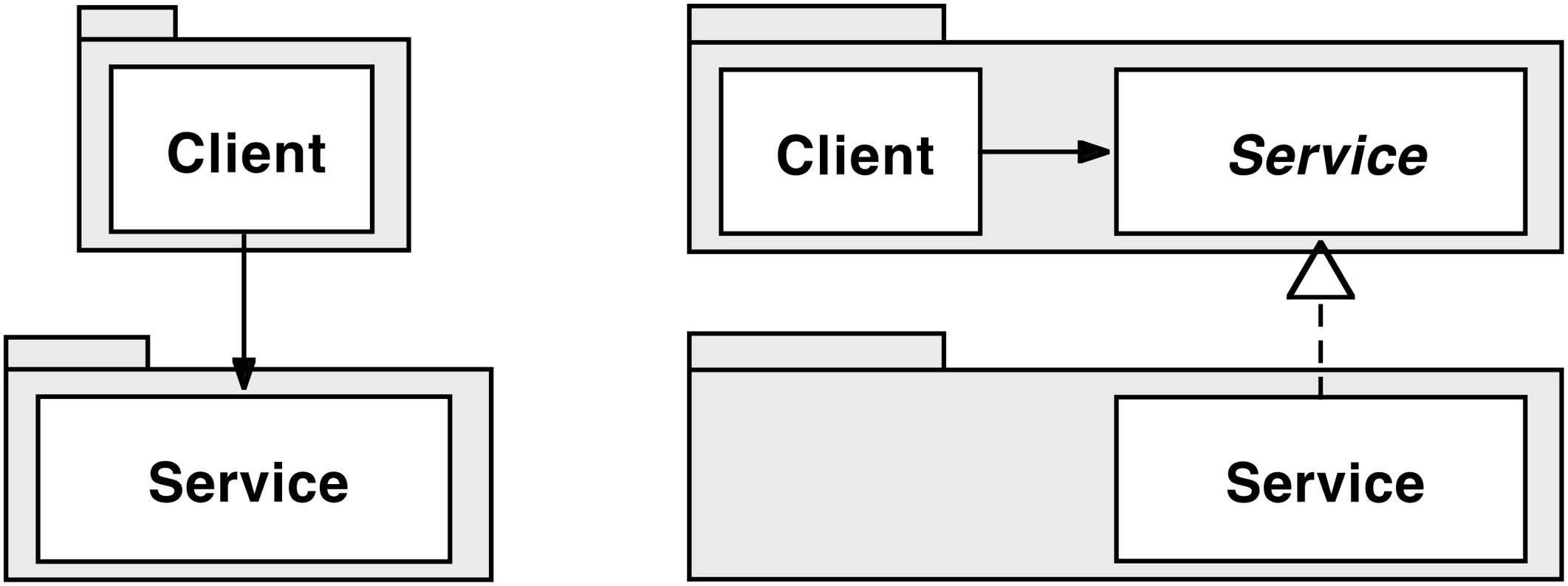
```
class Picture < …

  …
  def draw_all shapes
    shapes.each do |shape|
      shape.draw
    end
  end
end
```

Common idioms, like iteration over a collection, make composition easy.

#4

Dependency Inversion Principle

Abstractions should not depend upon details. Details should depend upon abstractions.


Client
Service
Client
Service
Service

In Java, a Service interface would be needed.

In Ruby, we use

Duck Typing...

```
class Client
  def initialize service
    @service = service
  end
  def do_something *args
    @service.do_service args
  end
end
```

It just works,
so long as @service
responds to do_service!

But sometimes we can

turn DIP around...

What if the Client has a collection and the Service is invoked on each item?

```
class Service
  def do_service client
    client.each do |item|
      service_item item
    end
  end
  private
  def service_item item; …; end
end
```

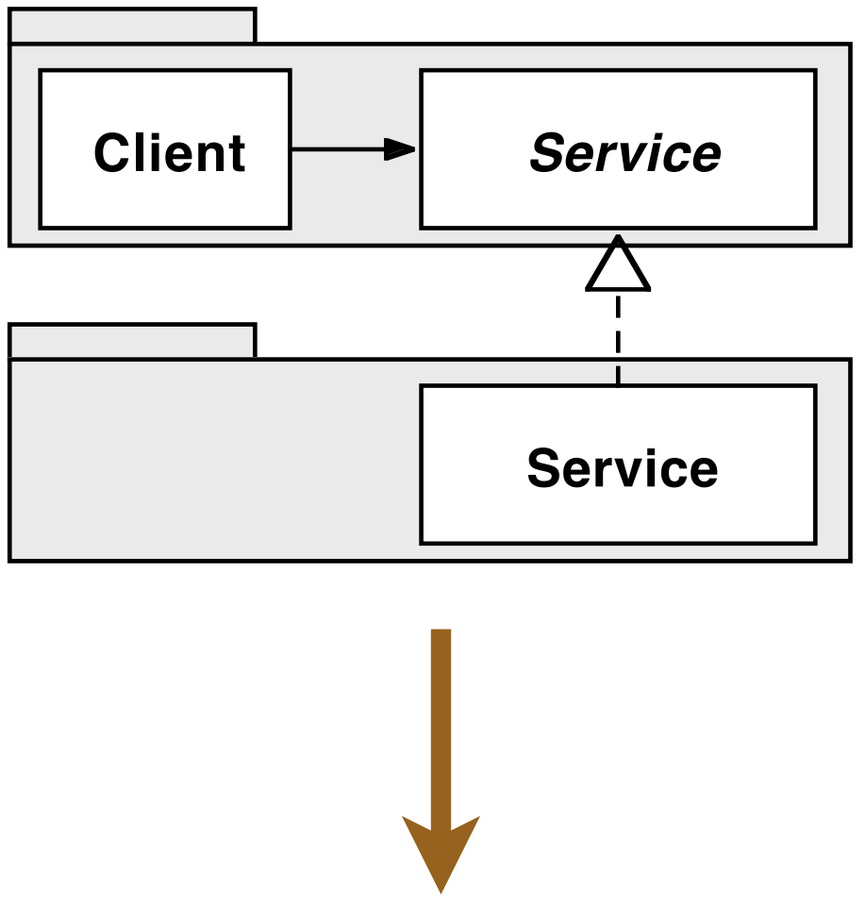
Pass the Client to the Service, which iterates over the Client.

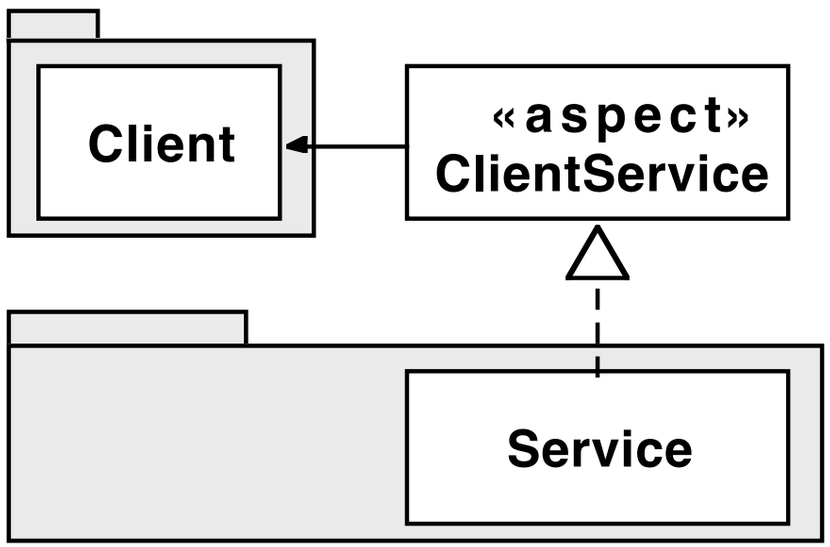
Similar to the Visitor pattern

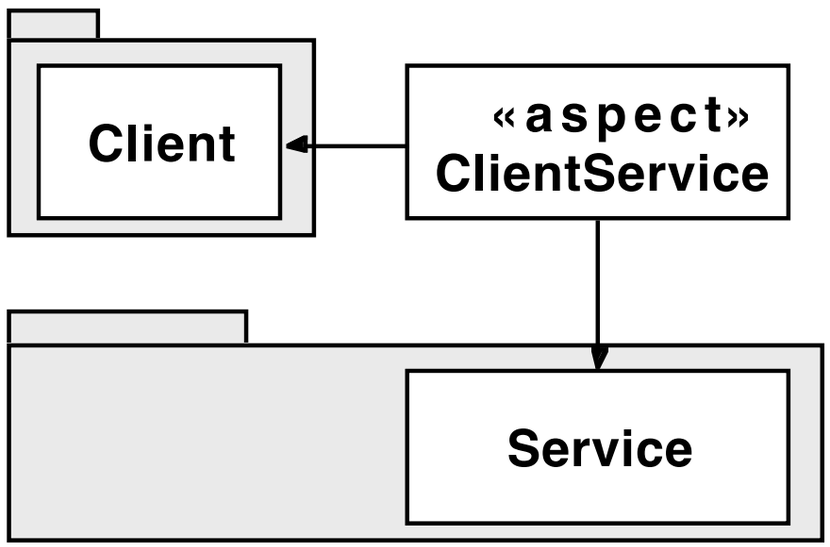
Sometimes an

Aspect is best...

```
require “aquarium”
module PersistenceService
 include Aquarium::DSL
 after :changing => :state_var,
 :on_types_and_descendents_of => Shape \
   do |context, object|
     persist_change_to object
 end
 def persist_change_to object; …; end
end
```


Client
Service
Service


Client
«aspect»
ClientService
Service


Client
«aspect»
ClientService
Service

What about Ruby on Rails?

#1

Rails Models are tightly coupled to ActiveRecord.

This makes

testing harder

and code

less adaptable.

Introduce Abstraction?

```
class Picture  <    Domain
  belongs_to :portfolio
  has_many    :shapes
  # picture attributes
  ...
end
```

Instead of ActiveRecord::Base

Now must define the attributes here

Persist with DataMapper.

```
persist Domain,
    :using => :DataMapper
```

Generate "boilerplate"
for rendering.

```
present Domain,
    :except_for_types => […],
    :using => :WebPack
```

#2

Rails

Domain-Specific Languages

simplify development.

ActiveRecord DSL Example

```
class Picture < ActiveRecord::Base
  belongs_to :portfolio
  has_many   :shapes

  ...
end
```

We declare what we want,
not how to do it.

RSpec DSL Example

```
describe Rectangle, “height” do
  it “should be indep. of width” do
    [Rectangle.new(2,2),
     Square.new(2,2)].each do |s|
      old_width = s.width
      s.height = 2 * s.height
      s.width.should == old_width
      ...
```

Aquarium DSL Example

```
after :calls_to => :get_area,
    :on_types_and_descendents => Shape \
      do |context, object|
        object.notify
end
```

DSL's are *declarative*,
like *Functional*
Programming.

DSL's map

requirements more

closely to code.

DSL's promote Domain Models.

DSL's reduce the amount of code you write.

DSL's promote components + scripts = applications.

See *Polyglot* talk at
aspectprogramming.com/papers

… but *DSL implementations* can be *challenging*:

ActiveRecord Again

```
pictures = Picture.find_by_name(
   “My Circle”)
```

```
pictures = Picture.find_by_portfolio(
   “App Design”)
pictures = Picture.find_by_date(
   “July 24, 2008”)
```

How it's implemented

```
def method_missing meth_id, *args
 if match =
   /^find_(all_by|by)_([_a-zA-Z]\w*)$/.
       match(meth_id.to_s)
   finder = determine_finder match
   return finder.result unless errors?
 end
 super
end
```

So, use

DSL's wisely.

Conclusions

Ruby simplifies many design principles and patterns.

Automated tests *are*
even more important
in *dynamic languages*
like *Ruby*.

Automated tests needed for:

1. *Quality*
2. *Documentation*
   - *Lack of interfaces*
   - *Metaprogramming*

Apply Functional Programming

1. *No side-effects*
2. *Declarative*
3. *Composition of functions*
4. . . .

Use Ruby DSL's

1. *Map domain to code*
2. *Functional idioms*
3. *Components + Scripts = Applications*

This foundation produces less code to develop and maintain.

Less code makes applications more agile and scalable.

Thank You!

- dean@objectmentor.com
- http://objectmentor.com
  - Watch for our forthcoming *Clean Code* book!
- http://aquarium.rubyforge.org
- http://aspectprogramming.com/papers