

dean@deanwampler.com

@deanwampler

The Haystack, Oregon

How Functional Programming Changes Developer Practices

polyglotprogramming.com/talks

Agile 2011, August 11, 2011

1



Thursday, August 11, 2011

Adapted from my longer tutorial at github.com/deanwampler/Presentations/BetterProgrammingThroughFP.
All photos © 2010 Dean Wampler, unless other noted. Most of my photos are here: <http://www.flickr.com/photos/deanwampler/>. Most are from the Oregon coast. Some are from the San Francisco area. A few are from other places I've visited over the years.

(The Haystack, Cannon Beach, Oregon)

Who am I?



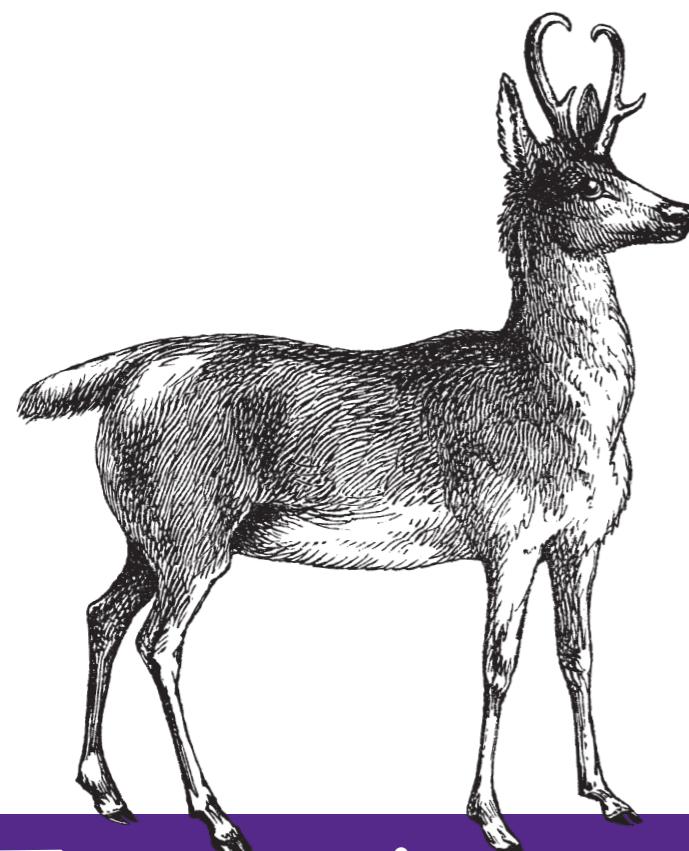
The image shows the header section of the Think Big Analytics website. On the left, the company logo "THINK BIG ANALYTICS" is displayed, with "THINK BIG" in large blue letters and "ANALYTICS" in red letters below it. To the right of the logo is a call-to-action text: "DEVELOP A STRATEGY, ARCHITECT YOUR TECHNOLOGY AND DEPLOY BIG DATA SOLUTIONS WITH OUR PROVEN APPROACH". Below this text is a link "Download White Paper: Large Scale Analytics in the Enterprise" with a small grey arrow icon. At the bottom of the header, there is a navigation menu with links: HOME, SERVICES, CUSTOMERS, ABOUT THINK BIG, CAREERS, and CONTACT US.

“Big Data”
Consulting
thinkbiganalytics.com

2

Thursday, August 11, 2011

Think Big Analytics does “big data” consulting, primarily using the Hadoop tools.



Functional Programming

for Java Developers

O'REILLY®

Dean Wampler

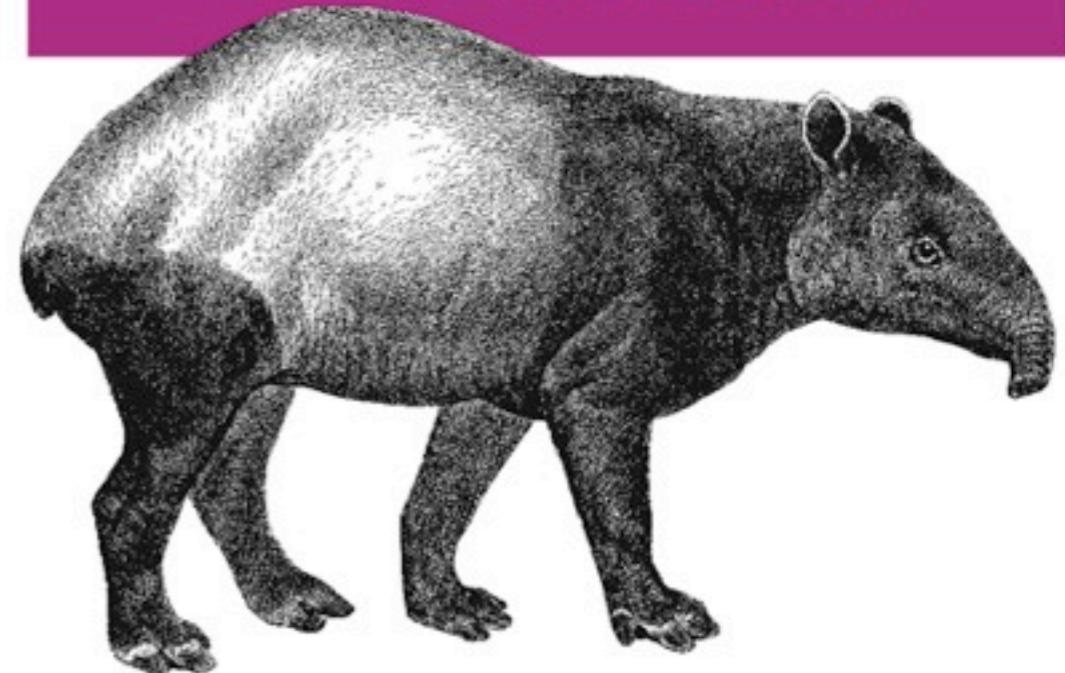
[polyglotprogramming.com/
fpjava](http://polyglotprogramming.com/fpjava)

3

Scalability = Functional Programming + Objects

Programming

Scala



O'REILLY®

Dean Wampler & Alex Payne

programmingscala.com

Thursday, August 11, 2011

I got interested in FP about 5 years ago when everyone was talking about it. I decided it was time to learn myself and I expected to pick up some good ideas, but otherwise remain primarily an “object-oriented developer”. Actually, it caused me to rethink my views and now I tend to use FP more than OOP. This tutorial explains why.

- The problems of our time.
- What is Functional Programming?
- Better reusability.
- Better concurrency.
- Better objects.

The problems of our time.

Thursday, August 11, 2011

What problems motivate the need for change, for which Functional Programming is well suited?

(Nehalem State Park, Oregon)

Concurrency



San Francisco Bay

Thursday, August 11, 2011

Concurrency is the reason people started discussing FP, which had been primarily an academic area of interest. FP has useful principles that make concurrency more robust and easier to write.

(San Francisco Bay)



We're Drowning in Data.

Google

twitter

facebook

YouTube

...

Thursday, August 11, 2011

Not just these big companies, but many organizations have lots of data they want to analyze and exploit.

(San Francisco)



We need better modularity.

Thursday, August 11, 2011

I will argue that objects haven't been the modularity success story we expected 20 years ago, especially in terms of reuse. I'm referring to having standards that actually enable widespread interoperability, like electronics, for example. I'll argue that object abstractions are too high-level and too open-ended to work well.

(Mud near Death Hollow in Utah.)

We need better agility.



Thursday, August 11, 2011

Schedules keep getting shorter. The Internet weeded out a lot of process waste, like Big Documents Up Front, UML design, etc. From that emerged XP and other forms of Agile. But schedules and turnaround times continue to get shorter.

(Ascending the steel cable ladder up the back side of Half Dome, Yosemite National Park)

We need a return to simplicity.

Thursday, August 11, 2011

Every now and then, we need to stop, look at what we're doing, and remove the cruft we've accumulated. If you're a Java programmer, recall how efforts like the Spring Framework forced a rethinking of J2EE. I claim that a lot of the code we write, specifically lots of object middleware, is cruft. Functional programming isn't **simple**, but in my view it reflects a refocusing on core principles and minimally-sufficient design.

(Maligne Lake, Near Jasper National Park, Jasper, Alberta)

What is Functional Programming?



Nehalem State Park, Oregon

11

Thursday, August 11, 2011

This is rich field, so I can't cover everything. I'll mention the things that I believe are most useful to know for beginners and those curious about FP.

(Nehalem State Park, Oregon)

*Functional
Programming
is inspired by
Mathematics.*

12

Thursday, August 11, 2011

FP follows the “rules” for the behavior of functions, variables, and values in mathematics. Everything else falls out from there...

What is Functional Programming?

*Immutable
Values*

13

Thursday, August 11, 2011

First, values in FP are immutable, but variables that point to different values, aren't.

Immutable Values

$$\begin{aligned}y &= \sin(x) \\1 &= \sin(\pi/2)\end{aligned}$$

x and y are *variables*.

Once you assign a *value* to x,
you fix the *value assigned to y*.

Immutable Values

$$y = \sin(x)$$

You can start over with new *values* assigned to the same *variables*.

But you never modify the *values*, themselves.

Immutable Values

$$\pi += 1$$

What would that mean?

Immutable Values

If a value is *immutable*,
synchronizing access is no longer necessary!

Concurrency becomes far easier.

What is Functional Programming?

Side-effect
free
functions

18

Thursday, August 11, 2011

Math functions don't have side effects. They don't change object or global state. All work is returned and assigned to y.

Functions

$$y = \sin(x)$$

$\sin(x)$ does not *change state* anywhere!

Referential Transparency

$$1 = \sin(\pi/2)$$

We can replace $\sin(\pi/2)$ with 1.

We can replace 1 with $\sin(\pi/2)$!

Functions and values are interchangeable

20

Thursday, August 11, 2011

A crucial implication of functions without side effects is that functions and values are interchangeable. A mundane benefit is that it's easy for an implementation to cache previous work for a given input value, for efficiency. But there are more profound benefits.

Functions

$$y = \sin(x)$$

$\sin(x)$ can be used *anywhere*.
I don't have to worry about the
context where it's used

What is Functional Programming?

*First-class
functions*

First Class Functions

```
i = 1  
l = List.new(i, ...)  
f = lambda do |x|  
  puts "Hello, #{x}!"  
end
```

First Class: values that can be assigned to variables, pass to and from functions.

Lambda is a common name for *functions*.

23

Thursday, August 11, 2011

A “thing” is first class in a language if you can use it as a value, which means you can assign it to variables, pass it as an argument to a function and return it from a function. In Ruby, objects, even classes are first class. Methods are not. Lambdas are ruby’s way of defining anonymous functions (A second mechanism, Procs, is similar).

The term “lambda” comes from Lambda Calculus, a mathematical formalism developed in the ‘30s that explored how functions should work. The lambda symbol was used to represent anonymous functions.

We'll see the power
of *First-class functions*

in a moment...

We'll see how first-class functions let us build
modular, composable, and reusable tools.

Better Reusability

Lists

Better Reusability

26

Thursday, August 11, 2011

I want to make the case that functional concepts lead to better modularity than objects.
Let's look at one of the functional data structures, List, which we've already looked at a bit, but we need to explore further.

List

```
class List
attr_reader :head, :tail
def initialize(head, tail)
  @head = head
  @tail = tail
end
...
end
```

Head is the first *element*.

Tail is itself a List.

27

Thursday, August 11, 2011

So, don't use attr_accessor or attr_writer in Ruby.

If you don't like dynamic typing, at least appreciate the compact, clean syntax.

Ruby

```
list = List.new(1,  
               List.new(2,  
                         List.new(3, EMPTY)))
```

We need a special *tail* to terminate a List.

28

Thursday, August 11, 2011

Creating a list (we'll see less verbose syntax later)

How should we terminate this list?? What should the special tail EMPTY be?? We'll come back to that.

List (cont.)

```
class List  
...  
def to_s  
  "(#{head},#{tail})"  
end  
...  
end
```

class List

A separate *object* to represent *empty*.

```
...
EMPTY = List.new(nil,nil)
def EMPTY.head
  raise "EMPTY list has no head!!"
end
def EMPTY.tail
  EMPTY
end
def EMPTY.to_s
  "()"
end
end
```

30

Thursday, August 11, 2011

We declare a *constant* named EMPTY, of type List. We use nil for the head and tail, but they will never be referenced, because we redefine the head method for this “singleton” object to raise an exception, while tail simply returns EMPTY itself! We also define to_s to return “()”.

By overriding the methods on the instance, we’ve effectively given it a unique type.

(There’s a more short-hand syntax for redefining these methods, but for simplicity, I’ll just use the syntax shown.)

NOTE: It would be reasonable for EMPTY.tail to throw an exception like head throws.

```
class List
...
def to_s
  "(#{head},#{tail})"
end

...
def EMPTY.to_s; "()"; end
...
end
```

List.to_s is recursive, but
EMPTY.to_s will terminate the
recursion with *no conditional test!*

31

Thursday, August 11, 2011

No conditional test is required in *to_s* to terminate the recursion. It is not an infinite recursion, though, because all lists end with *EMPTY*, which will terminate the recursion.
We've replaced a conditional test with structure, which is actually a classic OO refactoring.

List.to_s

```
puts List.new(1,  
             List.new(2,  
                     List.new(3, EMPTY))  
           )  
  
=> "(1, (2, (3, ())))"
```

Lists are represented
by two types:

List and EMPTY.

A scenic view of a beach at low tide. The sand is wet and reflects the sky. In the background, there are green hills or mountains under a cloudy sky.

filter, map, fold

Better Reusability

34

Thursday, August 11, 2011

Let's look at the 3 fundamental operations on data structures and understand their power.

Filter, map, fold

filter	Return a new collection with some elements removed.
map	Return a new collection with each element transformed.
fold	Compute a new result by accumulating each element.

All take a *function* argument.

35

In Ruby...

filter

find_all

map

map

fold

inject

Add map to List

f takes one arg, each item,
and returns a new value for
the new list.

```
def map(&f)
  t = tail.map(&f)
  List.new(f.call(head), t)
end
def EMPTY.map(&f); self; end
```

f.call(head) converts
head into something new.

37

Thursday, August 11, 2011

Add map first, because it's the easiest. Note that we will show the implementations for both List and EMPTY together, to compare and contrast and to make the behavior of the recursion clear.

Example of map

```
list = ... # 1,2,3,4
lm = list.map { |x| x*x}
puts "list: #{list}"
puts "lm:   #{lm}"
# => list: (1,(2,(3,(4,())))))
# => lm:   (1,(4,(9,(16,()))))
```

Add filter to List

f takes one arg, each item,
and returns true or false.

```
def filter(&f)
  t = tail.filter(&f)
  f.call(head) ?
    List.new(head, t) : t
end
def EMPTY.filter(&f); self; end
```

f.call(head) returns
true or false (keep or discard)

39

Thursday, August 11, 2011

f.call(head) returns true if we keep the element or false if we discard it. If true, we return a new list with head and whatever t is. Otherwise, we just return t.

Example of filter

```
list = ... # 1,2,3,4
lf = list.filter { |x| x%2==1}
puts "list: #{list}"
puts "lf:   #{lf}"
# => list: (1,(2,(3,(4,())))))
# => lf:   (1,(3,())))
```

There are *two* folds:
foldl (left) and
foldr (right).

Add foldl to List

accum is the
accumulator.

f takes two args, accum
and each item, and
returns a new accum.

```
def foldl(accum, &f)
  tail.foldl(
    f.call(accum, head), &f)
end
def EMPTY.foldl(accum,&f)
  accum
end
```

tail.foldl(...) is called *after*
calling f.call(...)

Add foldr to List

f takes two args, each item and accum, and returns a new accum.

```
def foldr(accum, &f)
  f.call(head,
    tail.foldr(accum, &f))
end
def EMPTY.foldr(accum,&f)
  accum
end
```

tail.foldr(...) is called before calling f.call(head,...)

43

Thursday, August 11, 2011

Foldr calls tail.foldr before calling f.call(head,accum). Note that it “groups” the accum with the last element (because head isn’t handled until the whole recursion finishes!), so it works down to the end of the list first, then builds the accumulator on the way back up.

Note that the arguments to f are reversed compared to foldl. We’ll see why this is useful in a moment.

Example of foldl

```
ll = list.foldl(0) { |s,x| s+x}
lls= list.foldl("0") { |s,x|
  "(#{s}"+#{"x})"
}
puts "ll: #{ll}"
puts "lls: #{lls}"
# => ll: 10
# => lls: (((((0+1)+2)+3)+4)
```

Example of foldr

```
lr = list.foldr(0) { |x,s| x+s}
lrs= list.foldr("0") { |x,s|
  "(#{x})"+#{s})"
}
puts "lr: #{lr}"
puts "lrs: #{lrs}"
# => lr: 10
# => lrs: (1+(2+(3+(4+0))))
```

Compare foldl, foldr

```
foldl: (((((0+1)+2)+3)+4) == 10  
foldr: (1+(2+(3+(4+0)))) == 10
```

The *sums* are the same,
but the *strings* are *not!*

Addition is *commutative* and *associative*.

Try subtraction

```
foldl: (((0-1)-2)-3)-4) == -10  
foldr: 1-(2-(3-(4-0))) == -2
```

Substitute - for +.
Subtraction is *neither commutative nor associative.*

Modularity

Better
Reusability

48

Thursday, August 11, 2011

Let's look at one of the functional data structures, List, which we've already looked at a bit, but we need to explore further.

filter, map and fold as *modules*...

49

Thursday, August 11, 2011

So, we looked at these. What's the big deal?? They are excellent examples of why functional programming is the right approach for building truly modular systems...

A Good Module:

interface	Single responsibility, clear abstraction, hides internals
composable	Easily combines with other modules to build up behavior
reusable	Can be reused in many contexts

50

Thursday, August 11, 2011

Here are some of the qualities you expect of a good “module”. It exposes an interface that focuses on one “task”. The use of the abstraction is clear, with well defined states and transitions, and it’s easy to understand how to use it. The implementation is encapsulated.

You can compose this module with others to create more complex behaviors.

The composition implies reusability! Recall that it’s hard to reuse anything with side effects. Mutable state is also problematic if the module is shared.

Group email addresses

```
addrs = List.make(  
  "Dean@GMAIL.COM",  
  "bob@yahoo.com",  
  "tom@Spammer.COM",  
  "pete@YAHOO.COM",  
  "bill@gmail.com")
```

Exercise: implement
`List.make`

Let's *convert* to lower case, *filter* out spammers, and *group* the users by address...

Group email addresses

```
grouped = addrs.map { |x|
  x.downcase
}.filter { |x|
  x !~ /spammer.com$/
}.foldl({}) { |grps,x|
  name, addr = x.split('@')
  l = grps[addr] || List::EMPTY
  grps[addr] = List.new(name,l)
  grps
}
```

Group email addresses

```
...  
grouped.each { |key,value|  
  puts "#{key}: #{value}"  
}
```

```
=> yahoo.com: (pete,(bob,()))  
=> gmail.com: (bill,(dean,()))
```

We calculated this grouping
in *10 lines of code!!*

If we had
GroupedEmailAddresses
objects,
how much more code
would be *required*?

How much more
development time
would be *required?*

filter, map, and fold are ideal *modules*.

Each has a *clear abstraction*,
composes with others,
and is *reusable*.

filter, **map**, and **fold**
are *combinators*.



Better Objects

58

Immutable Values



Better Objects

59

Immutable values
are better for
concurrency and they
minimize obscure
bugs because of
side effects.

60

Thursday, August 11, 2011

If you must do multithreaded programming, it's far easier if your values are immutable, because there is nothing that requires synchronized access. Also, obscure bugs from "non-local" side effects are avoided.

Immutability tools

- *final* or *constant* variables.
- No field “setter” methods.
- Methods have no side effects.
 - Methods return new objects.
- (Persistent data structures.)

TDD

Better Objects

62

Test Driven Development (including refactoring)

is still useful in FP,
but there are changes.

First, you tend to use
more experimentation
in your *REPL*
and *less test first.*

64

Thursday, August 11, 2011

It's somewhat like working out a math problem. You experiment in your Read Eval Print Loop (interactive interpreter), working out how an algorithm should go. Then you commit it to code and write tests afterwards to cover all cases and provide the automated regression suite. The test-driven design process seems to fit less well, but other people may disagree!

Testing Money

```
class Money
  PRECISION = 0.00001
  attr_reader value
  def initialize value
    @value = round(value)
  end

  def round value
    # return rounded to ? digits
  end

  ...
end
```

65

Thursday, August 11, 2011

Money is a good domain class to implement as a “functional” type, because it has well-defined semantics and supports several algebraic operations!
The round method rounds the value to the desired PRECISION. I picked 5 decimal places, even though we normally only show at most a tenth of a penny...

Testing Money

```
...
def add other
  v = other.instance_of?(Money) ?
    other.value : other
  Money.new(value + v)
end
...
end
```

Imaginary RSpec

```
describe "Money addition" do
  money_gen = Generator.new do
    Money(-100.0) to Money(100.0)
  end
...
```

Define a “generator” that generates a random sample of instances between the ranges shown.

Imaginary RSpec

```
describe "Money addition" do
  money_gen = Generator.new do
    Money(-100.0) to Money(100.0)
  end
  property "is commutative" do
    money_gen.make_pairs do |m1,m2|
      m1.add(m2).should_be_close(
        m2.add(m1), Money::PRECISION)
    end
  end
end
```

verify that addition is
commutative!

68

Thursday, August 11, 2011

In our fictitious RSpec extensions, we verify the property that addition is commutative. We ask the “money_gen” to create some random set of pairs, passed to the block, and we verify that $m1+m2 == m2+m1$ within the allowed precision.

Test Driven Development
becomes
property verification.

Refactoring?

```
grouped = addrs.map { |x|
  x.downcase
}.filter { |x|
  x !~ /spammer.com$/
}.foldl({}) { |grps,x|
  name, addr = x.split('@')
  l = grps[addr] || List::EMPTY
  grps[addr] = List.new(name,l)
  grps
}
```

How might you
refactor this code?

Recall

```
grouped = addrs.map { |x|
  x.downcase
}.filter { |x| ← Extract Function?
  x !~ /spammer.com$/
}.foldl({}) { |grps,x|
  name, addr = x.split('@')
  l = grps[addr] || List::EMPTY
  grps[addr] = List.new(name,l)
  grps
}
```

oo

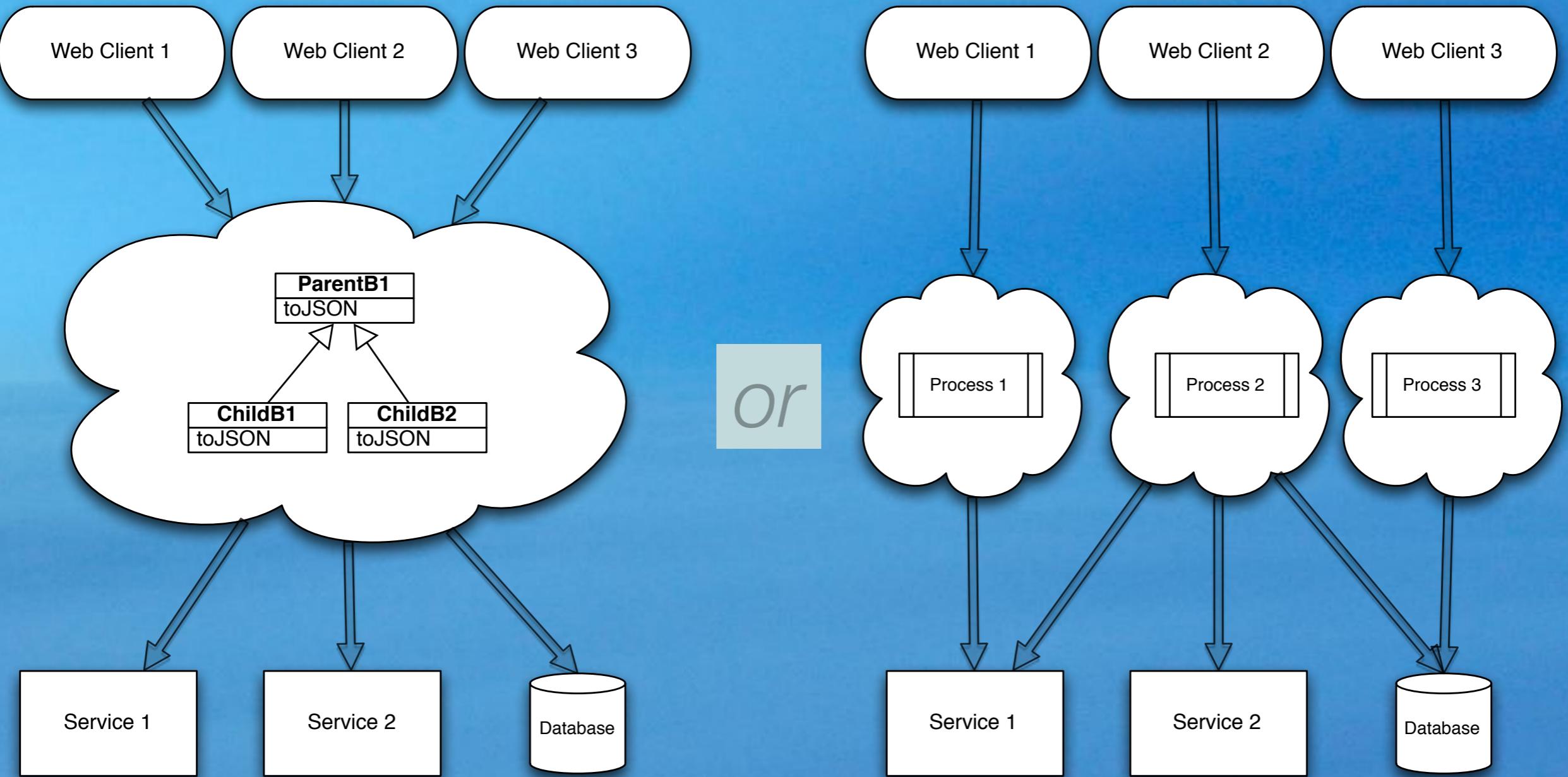
Middleware

Better Objects

72

In a *highly-concurrent*
world, do we really
want a *middle*?

Which Scales Better?

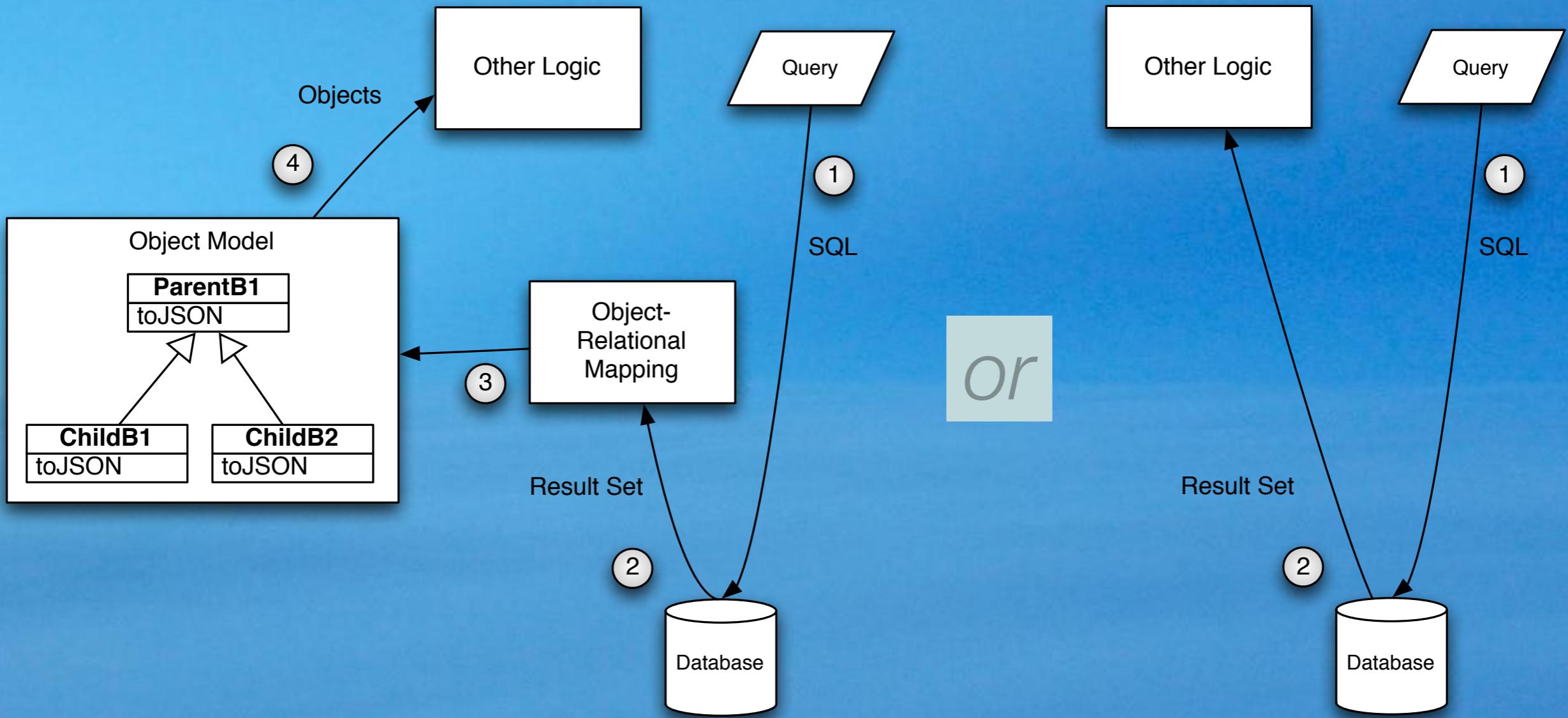


74

Thursday, August 11, 2011

If we funnel everything through a faithfully-reproduced domain object model, our services will be bigger, harder to decompose into smaller pieces, and less scalable. *Modeling* our domain to understand it is one thing, but implementing it in code needs to be rethought. The compelling power of combinators and functional data structures are about as efficient and composable as possible. It's easier to compose focused, stateless services that way and scale horizontally.

What about ORM?



Question Object-Relational Mapping

75

Thursday, August 11, 2011

What if your business logic just worked with the collections returned from your database driver? It's true that some of these collections, like Java's ResultSet, don't have the powerful combinators we've been discussing, but those "methods" could be added as static service methods in a helper class.

The question to ask is this: does the development and runtime overhead of converting to and from objects justify the benefits?

*Object middleware,
including ORM, isn't
bad. It just has costs
like everything else...*

Recap

Thursday, August 11, 2011
(Nehalem State Park, Oregon)

Concurrency



San Francisco Bay

Thursday, August 11, 2011

Concurrency is the reason people started discussing FP, which had been primarily an academic area of interest. FP has useful principles that make concurrency more robust and easier to write.

(San Francisco Bay)



We're Drowning in Data.

twitter

facebook

YouTube

...

Thursday, August 11, 2011

Not just these big companies, but many organizations have lots of data they want to analyze and exploit.
(San Francisco)

Mud, Death Hallow Trail, Utah



We need better modularity.

Thursday, August 11, 2011

I will argue that objects haven't been the modularity success story we expected 20 years ago, especially in terms of reuse.

(Mud near Death Hollow in Utah.)

We need better agility.



Thursday, August 11, 2011

Schedules keep getting shorter. The Internet weeded out a lot of process waste, lot Big Documents Up Front, UML design, etc. From that emerged XP and other forms of Agile. But schedules and turnaround times continue to get shorter.

(Ascending the steel cable ladder up the back side of Half Dome, Yosemite National Park)

We need a return
to simplicity.

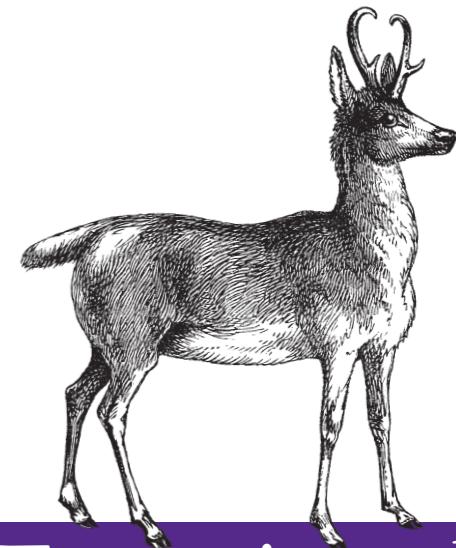
Thursday, August 11, 2011

Every now and then, we need to stop, look at what we're doing, and remove the cruft we've accumulated. I claim that a lot of the code we write, specifically lots of object middleware, is cruft.

(Maligne Lake, Near Jasper National Park, Jasper, Alberta)

Thank You!

- dean@deanwampler.com
- [@deanwampler](https://twitter.com/deanwampler)
- polyglotprogramming.com/talks
- thinkbiganalytics.com



Functional Programming

for Java Developers

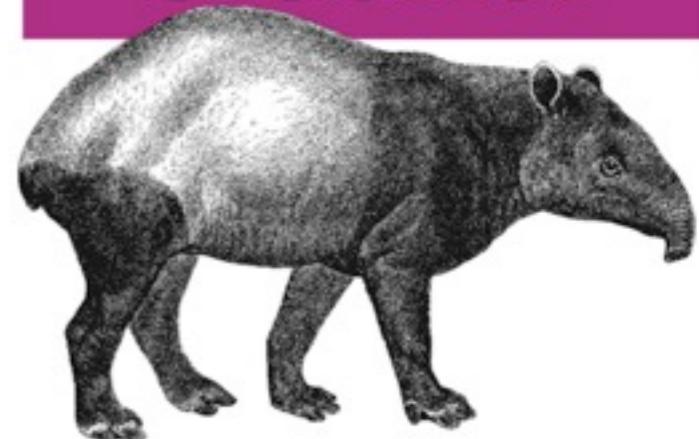
O'REILLY®

Dean Wampler

Scalability = Functional Programming + Objects

Programming

Scala



O'REILLY®

Dean Wampler & Alex Payne