polyglotprogramming.com

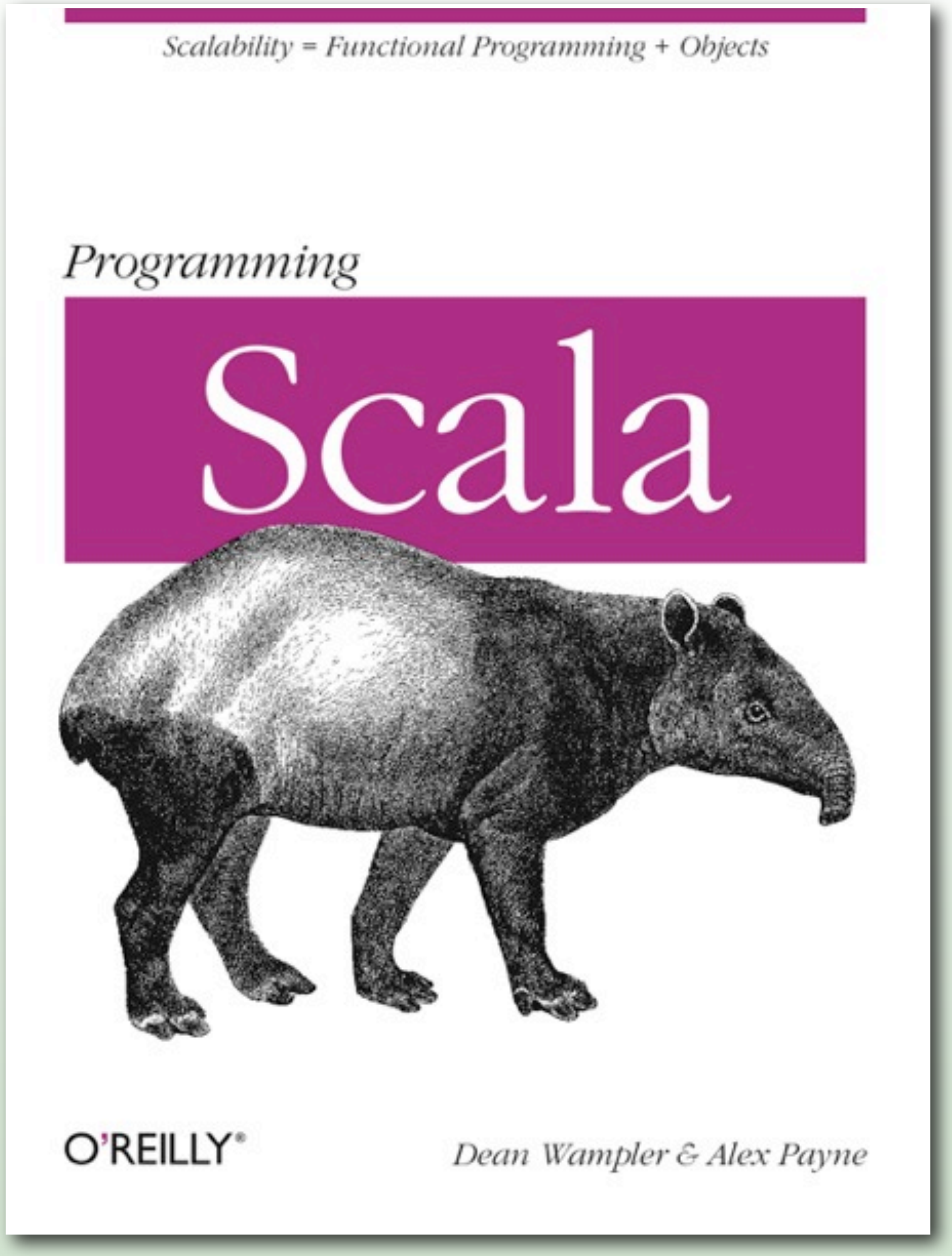
Polyglot and Poly-paradigm Programming

Dean Wampler
dean@deanwampler.com
@deanwampler

April 2, 2011

Co-author, *Programming Scala*

programmingscala.com


Scalability = Functional Programming + Objects

Programming Scala

O'REILLY

Dean Wampler & Alex Payne

IEEE Software

Building the Community of Leading Software Practitioners

www.computer.org/software

Guest Editor,

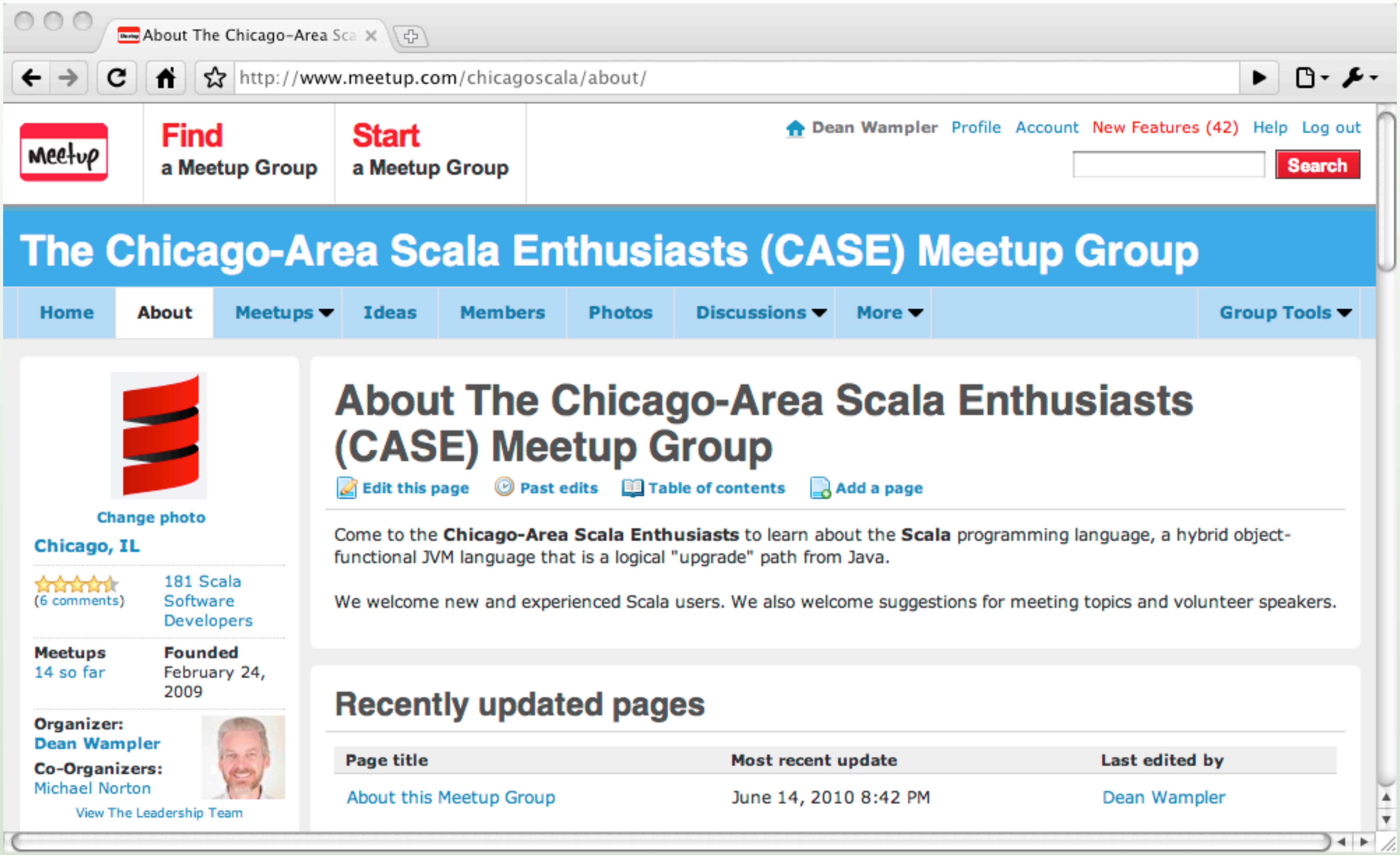
IEEE Software

Special Issue on

Multi-paradigm Programming

Sept/Oct 2010

computer.org/software



About The Chicago-Area Sca

http://www.meetup.com/chicagoscala/about/

The Chicago-Area Scala Enthusiasts (CASE) Meetup Group

Chicago, IL

(6 comments) 181 Scala Software Developers

Meetups 14 so far

Founded February 24, 2009

Organizer: Dean Wampler

Co-Organizers: Michael Norton

About The Chicago-Area Scala Enthusiasts (CASE) Meetup Group

Come to the **Chicago-Area Scala Enthusiasts** to learn about the **Scala** programming language, a hybrid object-functional JVM language that is a logical "upgrade" path from Java.

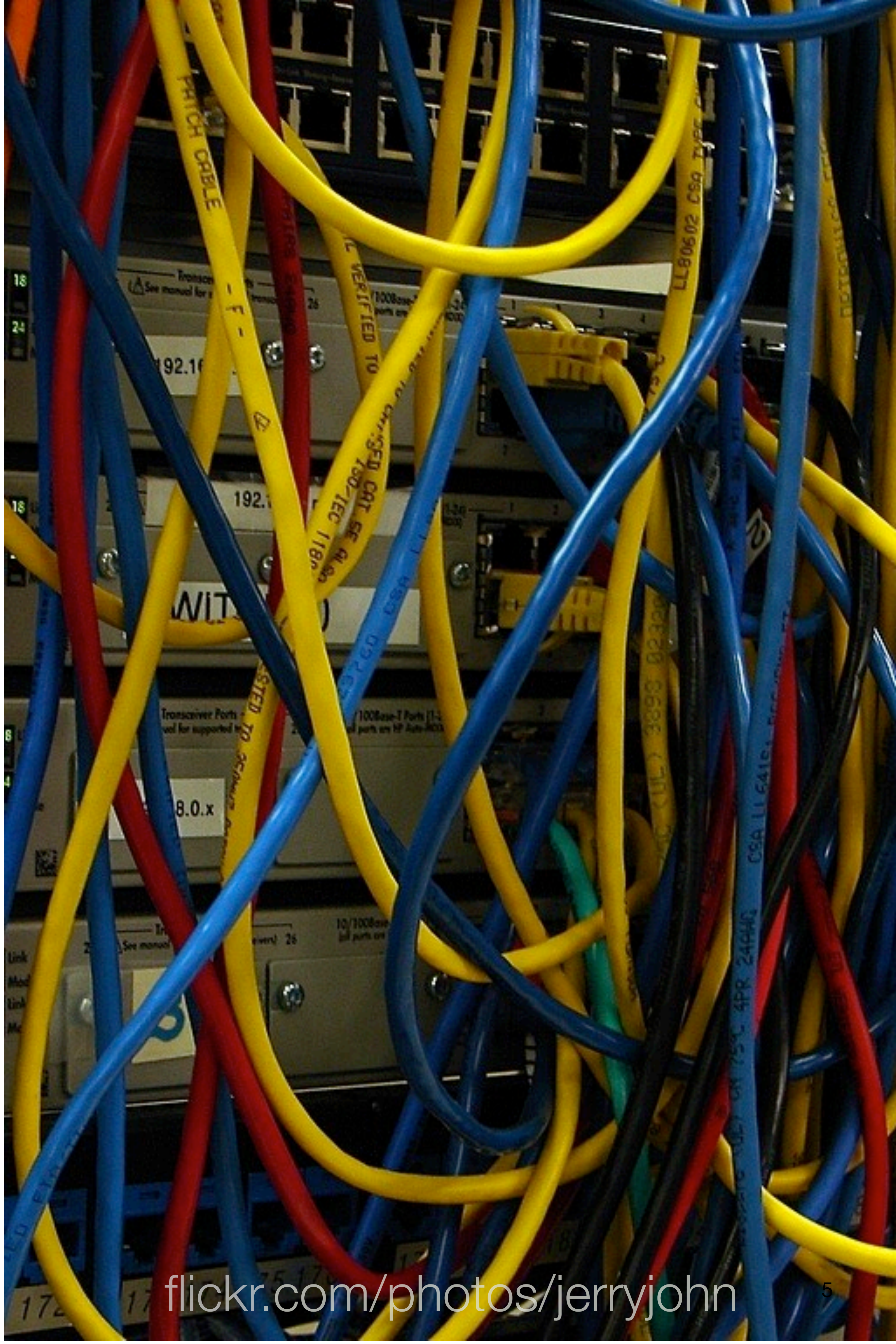
We welcome new and experienced Scala users. We also welcome suggestions for meeting topics and volunteer speakers.

Recently updated pages

Page title	Most recent update	Last edited by
About this Meetup Group	June 14, 2010 8:42 PM	Dean Wampler

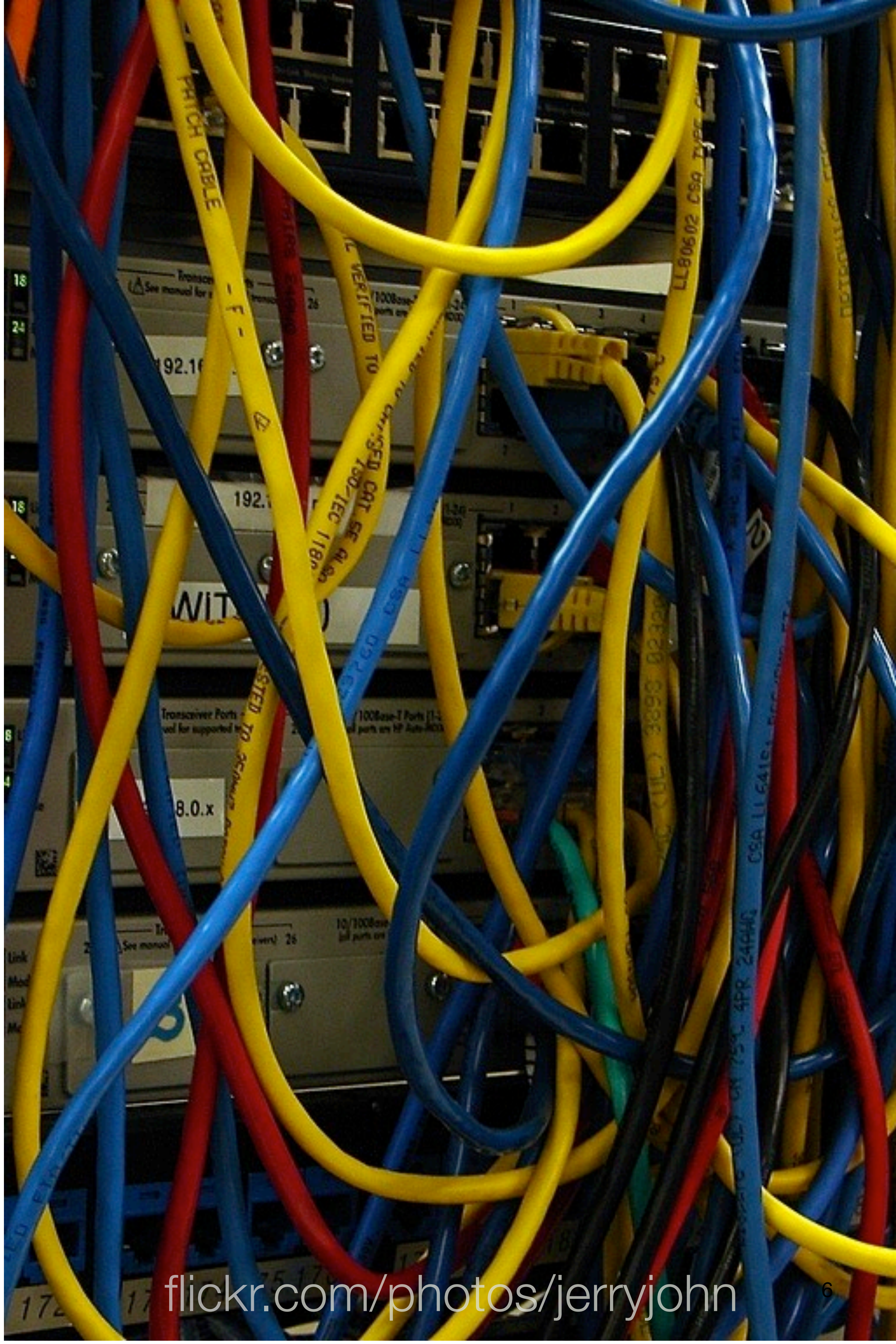
Today's applications:

- Are *networked*,
- Have graphical and "service" *interfaces*,



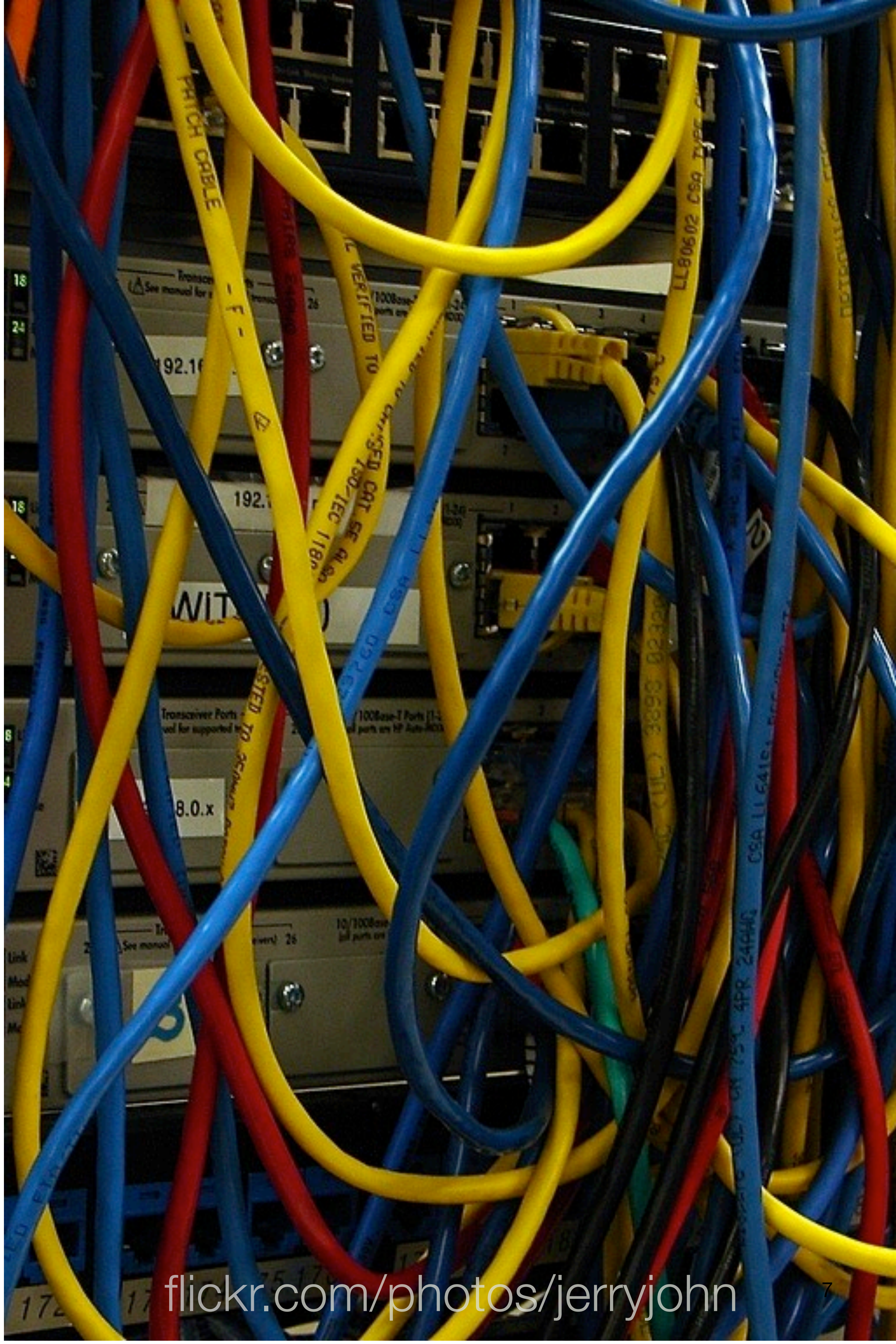
Today's applications:

- *Persist* data,
- Must be *resilient* and *secure*,
- Must *scale*,


flickr.com/photos/jerryjohn

Today's applications:

- … and must do all that by *Friday*.



Polyglot or *Multilingual*: many languages

Poly-paradigm or *Multiparadigm*:

many modularity paradigms

Thesis:

modern problems are poorly served by *“Monocultures”*

monocultures => “monoglot” and “mono-paradigm” programming.

Mono-paradigm:

Object-Oriented Programming:

right for all requirements?

Monolingual

Is one *language* best for all *domains*?

twitter.com/photos/watchsmart

domains: e.g., the problem domain for the app (usually an object model), the security model, the network/web topology, the relational or other data model, ...

Symptoms of *Monocultures*

- *Why is there so much XML in my Java?*
- *Why do I have similar code for persistence, transactions, security, etc. scattered all over my code base?*

Symptoms of *Monocultures*

- *How can I scale my application to internet scales?*
- *Why is my application so hard to extend?*
- *Why can’t I respond quickly when requirements change?*

Pervasive Symptom:

```
switch (elementItem)
{
    case "header1:SearchBox" :
    {
        __doPostBack('header1:goSearch','');
        break;
    }
    case "Text1":
    {
        window.event.returnValue=false;
        window.event.cancel = true;
        document.forms[0].elements[n+1].focus();
        break;
    } ...
```

Too much code!

Let’s examine some *common problems* with *PPP solutions*:

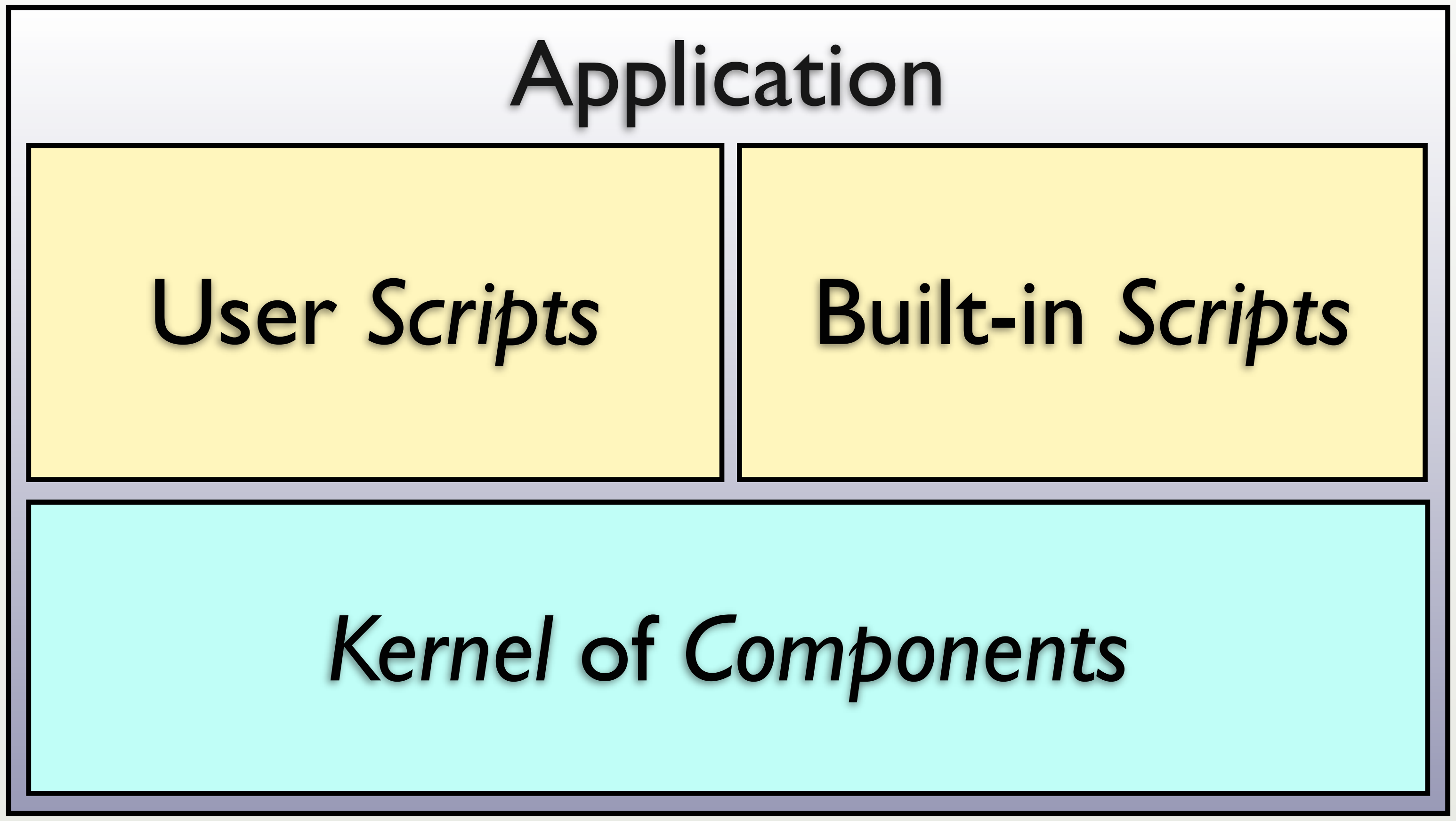
Change is *slow* and *painful*.

Problem #1

Symptoms

- *Features* take *too long* to implement.
- We can't *react* fast enough to *change*.
- Uses want to *customize* the system *themselves*.

Solution



(C Components) + (Lisp scripts) = Emacs

Components + *Scripts*
=
Applications

see John Ousterhout, IEEE Computer, March ’98

Pronunciation: OH-stir-howt

Kernel Components

- *Statically-typed language*:
  - C, C++, Java, C#, ...
- *Compiled* for speed, efficiency.
- Access *OS services*, 3rd-party *libraries*.
- Lower developer *productivity*.

Scripts

- *Dynamically-typed language*:
  - Ruby, Lisp, JavaScript, Lua, ...
- *Interpreted* for agility.
  - *Performance* less important.
- Glue together components.
- Raise developer *productivity*.

In practice, the divide between components and scripts is not so distinct.

To be clear about typing,

- *Static typing*

  → *compile time* checking.

- *Dynamic typing*

  → *run time* checking.

In practice,
the *boundaries* between
components and scripts
are not so *distinct*...

Ola Bini's *Three Layers*

- *Stable* layer
  - JVM + generic libraries
- *Dynamic* layer
  - e.g., JRuby, application libs.
- *Domain* layer
  - *Internal* and *External* DSLs.

x

Other Examples:

- *UNIX/Linux* + *shells*.
  - Also *find*, *make*, *grep*, ...
    - Have their own *DSLs*.

C++/Lua Examples:

- *Adobe Lightroom*
  - 40-50% written in Lua.
- *Game Engines*

Lightroom: Lua API used for 3rd-party plugins.
Lots of games combine C++ and Lua, too.

Embedded Systems:

- *Tektronix Oscilloscopes*: C + Smalltalk.
- *NRAO Telescopes*: C + Python.
- *Google Android*: Linux + libraries (C) + Java.

```
<view-state id="displayResults" view="/searchResults.jsp">
    <render-actions>
        <bean-action bean="phonebook" method="search">
            <method-arguments>
                <argument expression="searchCriteria"/>
            </method-arguments>
            <method-result name="results" scope="flash"/>
        </bean-action>
    </render-actions>
    <transition on="select" to="browseDetails"/>
    <transition on="newSearch" to="enterCriteria"/>
  </view-state>
</flow>
```

SpringSource just acquired *G2One* (*Groovy* and *Grails*). Will they switch to *Groovy* for configuration?

x

Hopefully, SpringSource will de-emphasize XML and emphasize Groovy for configuration “wiring”.

Property-based Programming

- Excellent for *malleable* objects.
- See Steve Yegge's blog
  - http://steve-yegge.blogspot.com/2008/10/universal-design-pattern.html
  - JavaScript, Lua, Self, ...

x

"Malleable" objects are those whose properties and behaviors may not be so clear cut. They may need to change over the life of the object.

Other Examples: Multilingual VM's

- On the *JVM*:
  - JRuby, Groovy, Jython, Scala.
  - Ruby on Rails on JRuby.

Another realization of C+S=A is to put several languages on the same VM, rather than using the OS as the component layer.

Other Examples: Multilingual VM's

- *Dynamic Language Runtime* (DLR).
  - Ruby, Python, ... on the *.NET CLR*.

Another realization of C+S=A is to put several languages on the same VM, rather than using the OS as the component layer.

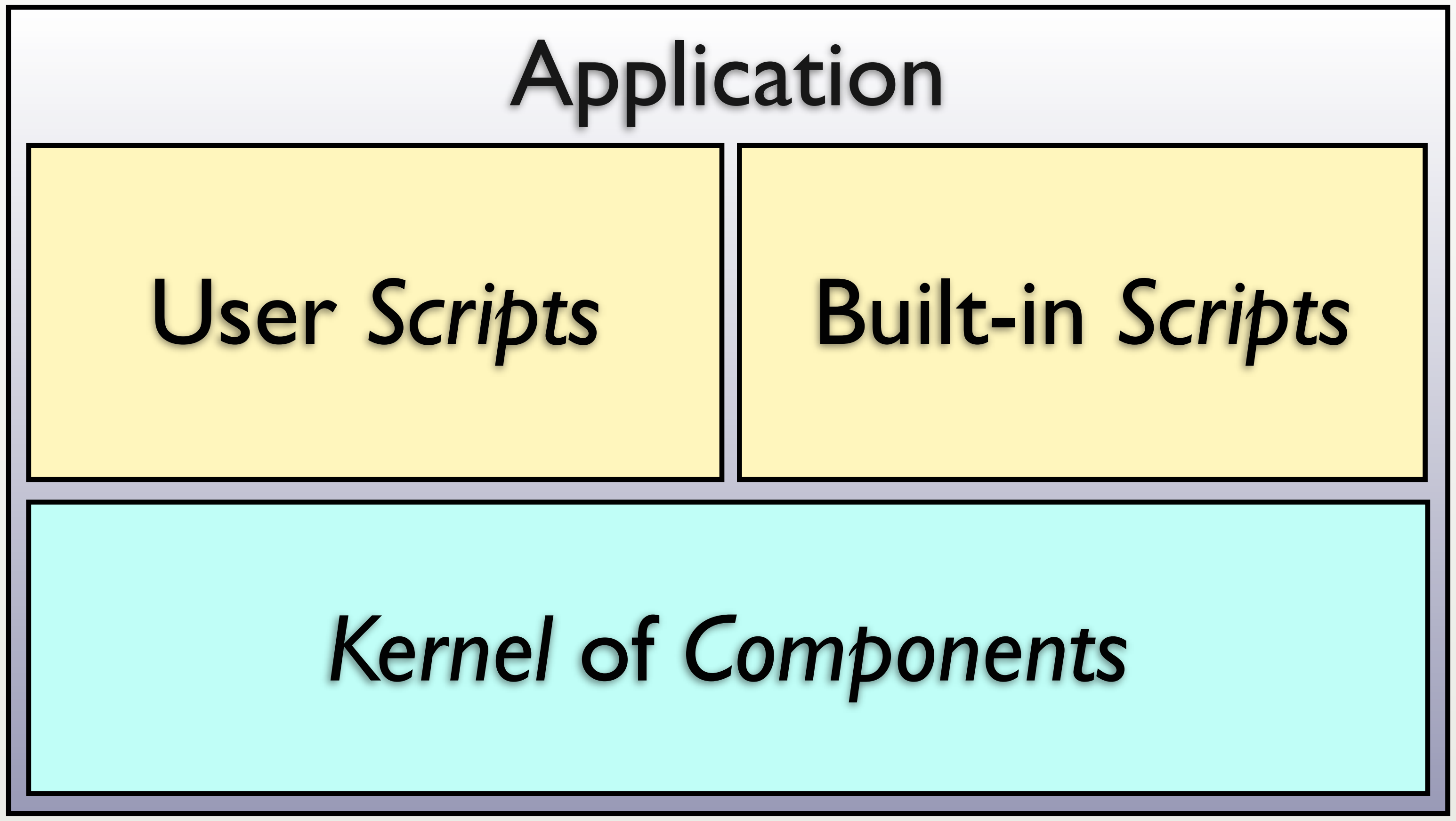
XML in Java

```
<view-state id="displayResults" view="/searchResults.jsp">
    <render-actions>
        <bean-action bean="phonebook" method="search">
            <method-arguments>
                <argument expression="searchCriteria"/>
            </method-arguments>
            <method-result name="results" scope="flash"/>
        </bean-action>
    </render-actions>
    <transition on="select" to="browseDetails"/>
    <transition on="newSearch" to="enterCriteria"/>
  </view-state>
</flow>
```

Why not *replace* XML with *JavaScript*, *Groovy* or *JRuby*??

De facto “scripting language” in Java.
Not an optimal choice:
- All data.
- No behavior (to speak of...).
- Verbose.

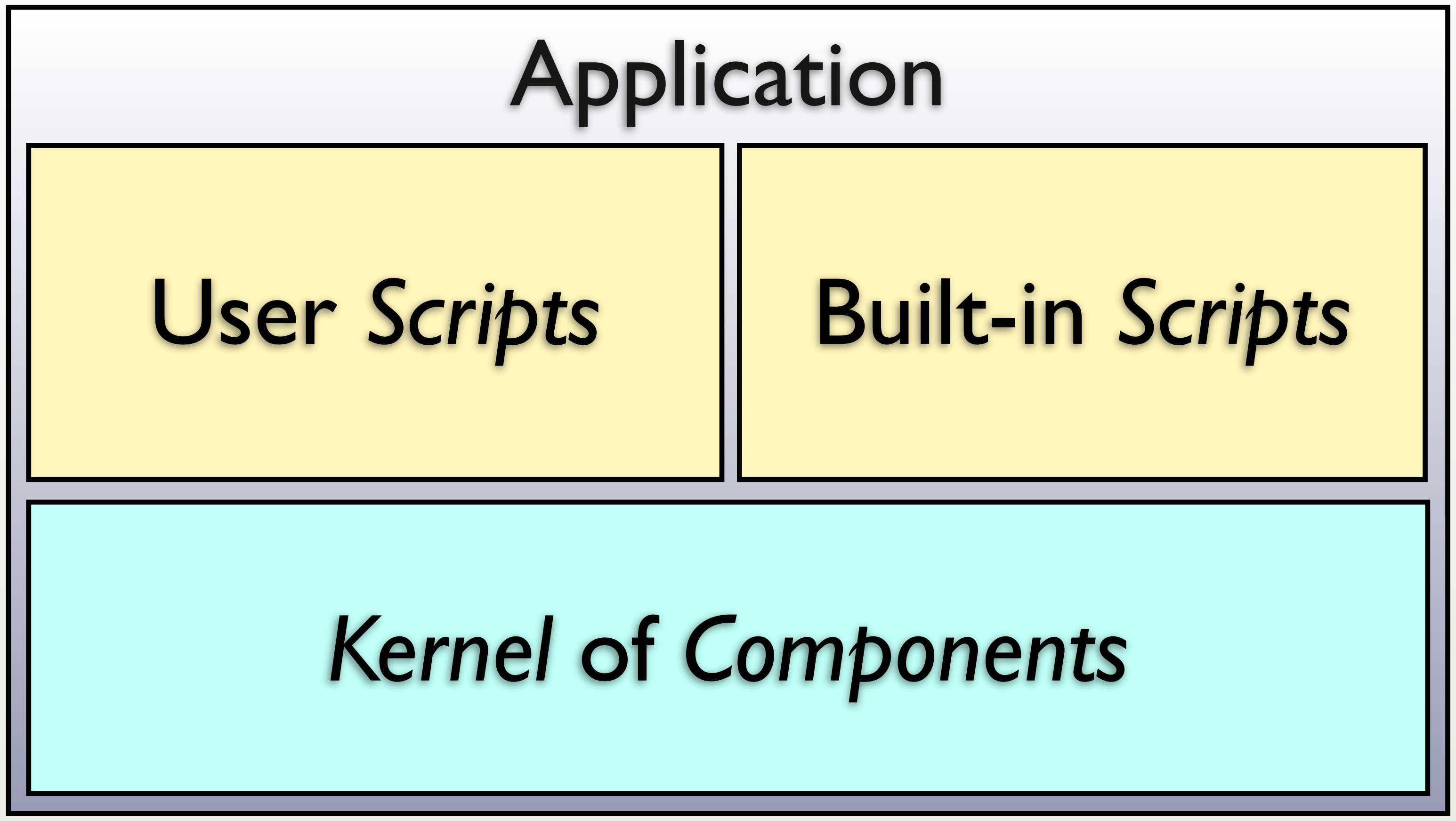
Benefits



- Optimize *performance* where it matters.
- Optimize *productivity*, *extensibility*, *agility* and *end-user customization* everywhere else.

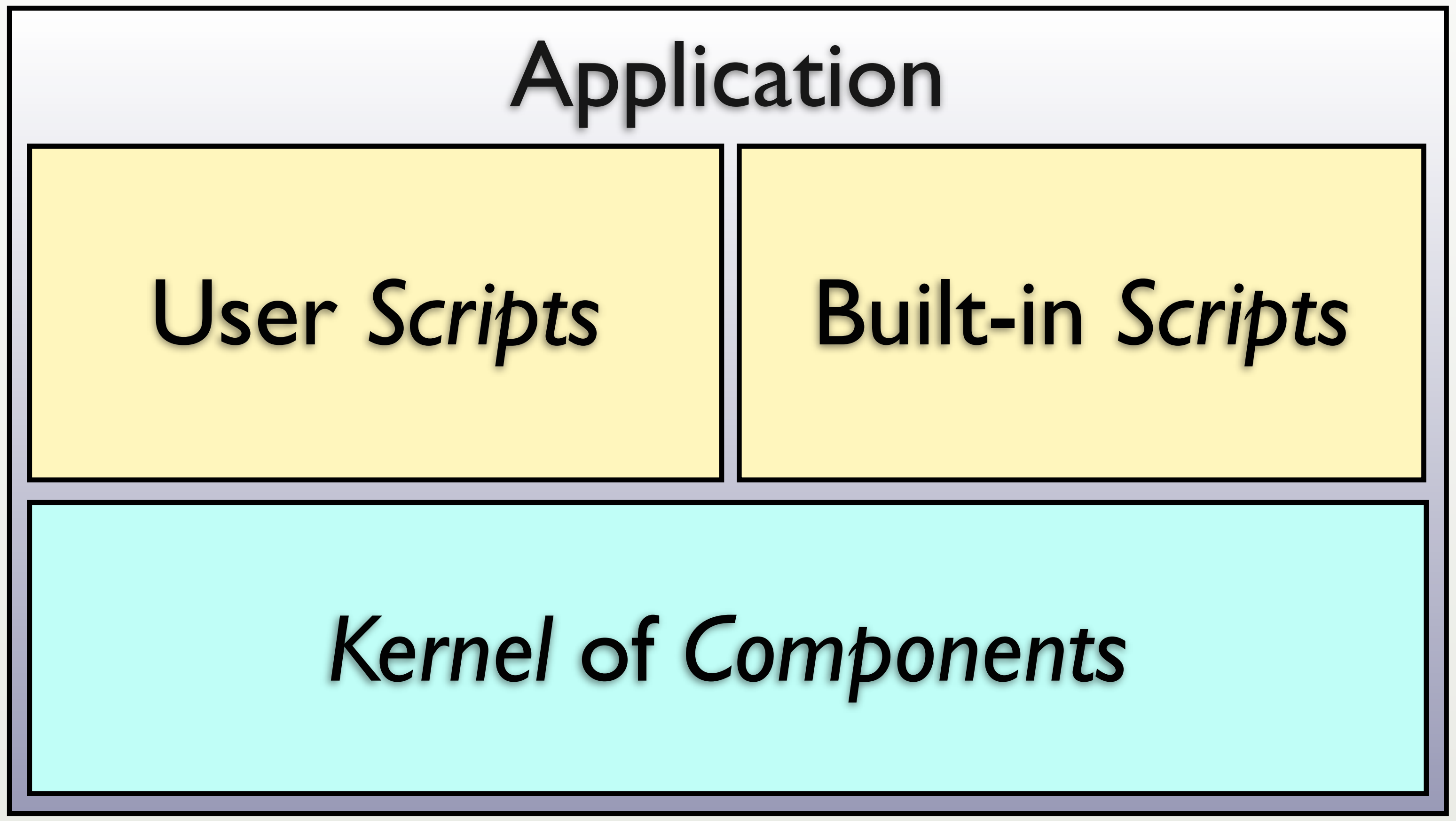
This is an underutilized architecture.

Disadvantages



- More *complexity* with 2+ languages.
- *Interface* between the layers.
- *Splitting behavior* between layers.

The complexity includes idioms, tools, and developer expertise for more than 1 language.



An *underutilized* architecture!

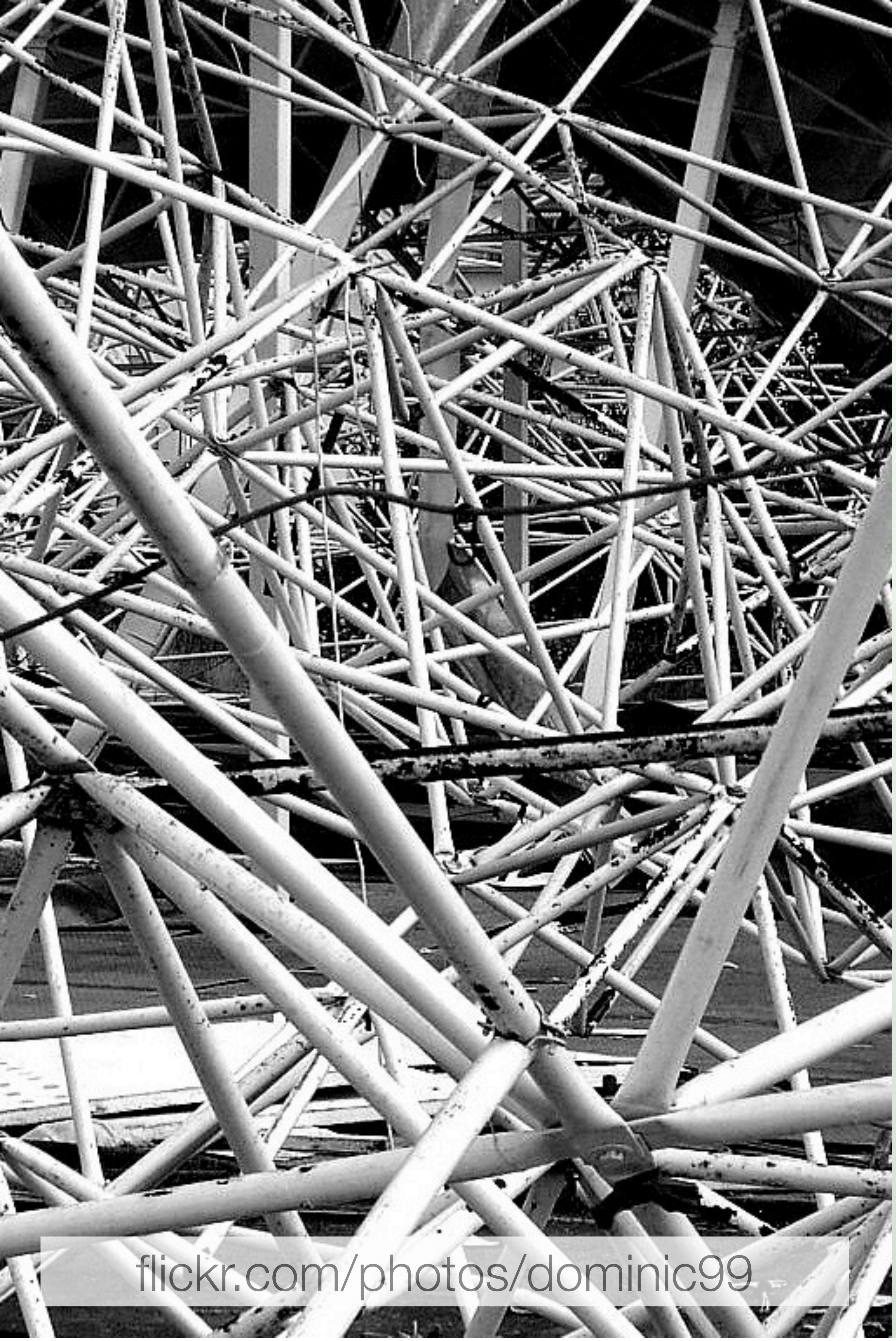
Parting Thought...

Why don't *Eclipse*, *IntelliJ*, etc.
have built-in *scripting engines*?

Parting Thought...

Cell phone makers ~~are~~ were drowning in *C++*.

(One reason the *IPhone* and *Android* are interesting.)



I don't *know* what my *code* is *doing*.

Problem #2

The *intent*
of our *code*
is *lost*
in the *noise*.

Symptoms

- New team members have a long *learning curve*.
- The system *breaks* when we *change* it.
- Translating *requirements* to *code* is *error prone*.

Solution #1

Write
less code!

You're welcome.

Less Code

- Means *problems* are *smaller*:
    - Maintenance
    - Duplication
    - Testing
    - Performance
    - *etc.*

How to Write Less Code

- Root out *duplication*.
- Use *economical* designs.
  - *Functional* vs. *Object-Oriented*?
- Use *economical* languages.

Solution #2

Separate *implementation details* from *business logic*.

Domain Specific Languages

Make the code read like “structured” domain prose.

Example DSL

```
internal {
  case extension
    when 100...200
      callee = User.find_by_extension extension
      unless callee.busy? then dial callee
      else
        voicemail extension

    when 111 then join 111

    when 888
      play weather_report('Dallas, Texas')

    when 999
      play %w(a-connect-charge-of 22
         cents-per-minute will-apply)
      sleep 2.seconds
      play 'just-kidding-not-upset'
      check_voicemail
  end
}
```

Adhearsion

=

Ruby DSL

+

Asterisk

+

Jabber/XMPP

+

...

PBX = Private Branch Exchange, the telephony exchange that serves a business or other office, etc.

DSL Advantages

- Code *looks* like domain prose:
  - Is easier to understand by *everyone*,
  - Is easier to *align* with the *requirements*,
  - Is more *succinct*.

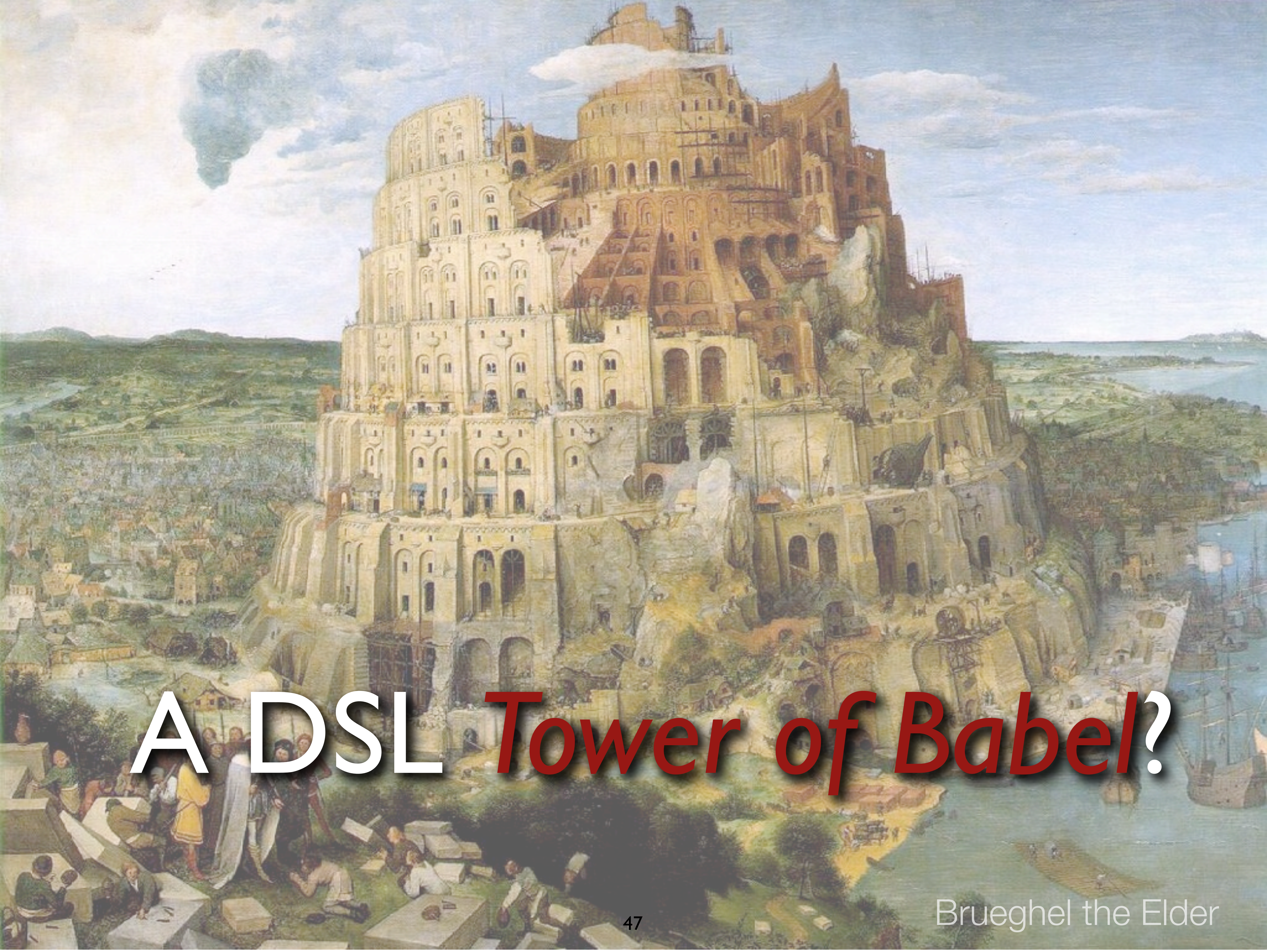
DSL Disadvantages

Many people are *poor* API designers.

DSLs are *harder* to *design*.

DSL Disadvantages

DSLs can be hard to implement, test, and debug.



A DSL *Tower of Babel*?

Not too many of this examples yet, but one comes to mind: mocking (for testing) frameworks in Ruby, BDD tools in several languages.

Parting Thought...

Perfection is achieved,
not when there is nothing left to add,
but when there is nothing left to remove.

-- Antoine de Saint-Exupery

He wrote “The Little Prince”, among other books. He was an aviator who disappeared over the Mediterranean in 1944, flying for Free French Forces.

Parting Thought #2...

Everything should be made as simple as possible, but not simpler.

-- Albert Einstein

Corollary:

Entia non sunt multiplicanda praeter necessitatem.

-- Occam's Razor

a.k.a. "Law of Parsimony" or "Law of Succinctness". Paraphrased translation.

Corollary:

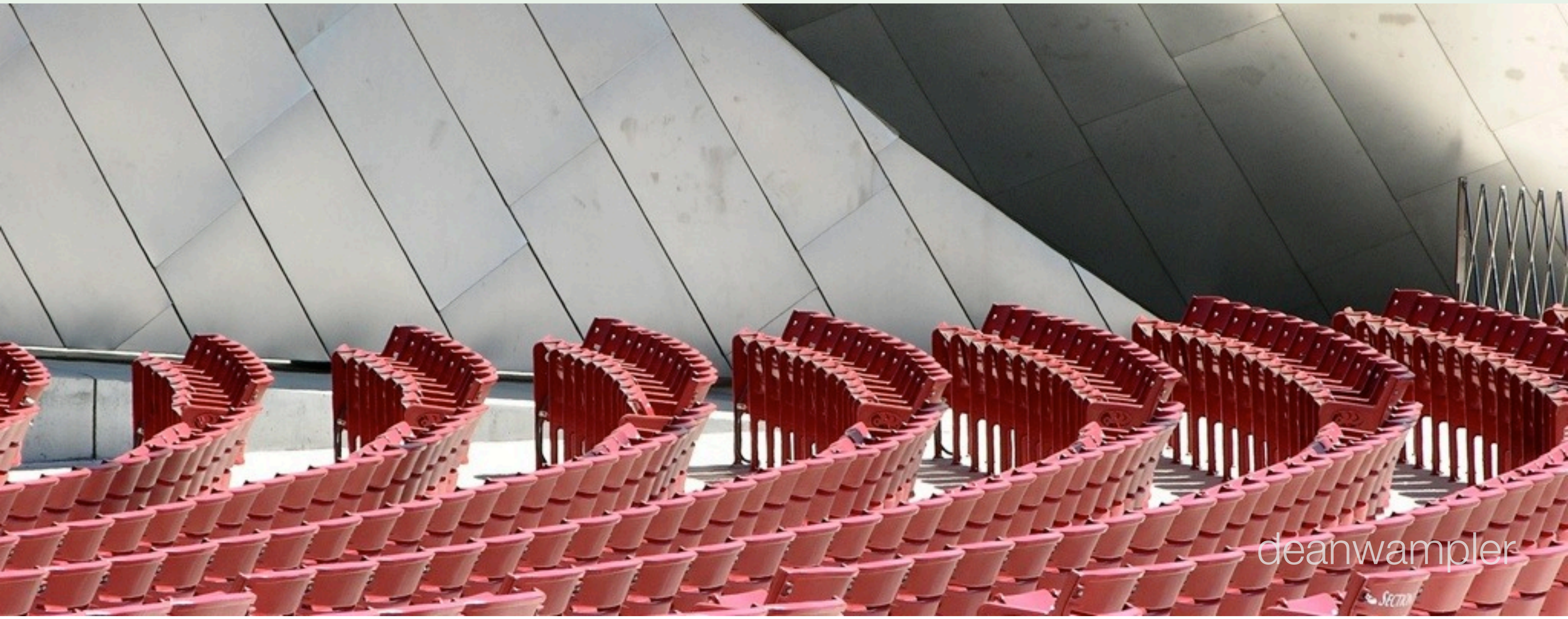
All other things being equal,
the simplest solution is the best.

-- Occam's Razor

a.k.a. "Law of Parsimony" or "Law of Succinctness". Paraphrased translation.

We have *code duplication* everywhere.

Problem #3



Symptoms

- *Persistence logic* is embedded in *every* "domain" class.
- Error handling and logging is *inconsistent*.

Cross-Cutting Concerns.

Solution

Aspect-Oriented Programming

Removing Duplication

- In order, use:
  - *Object* or *functional* decomposition.
  - *DSLs*.
  - *Aspects*.

Make sure your object and functional decomposition is right first, then use DSLs appropriately. Finally, use aspects.

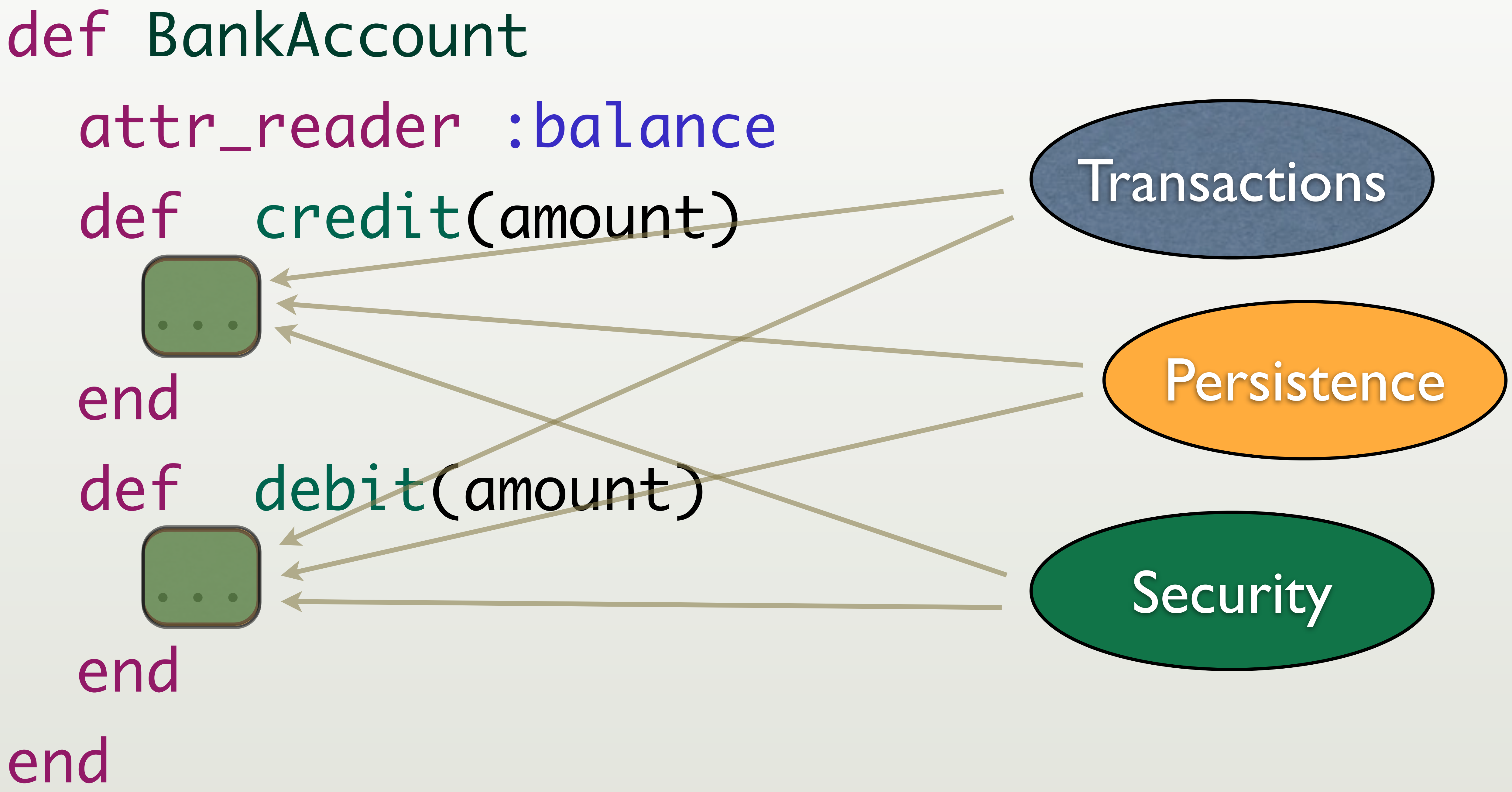
An Example...

Clean Code

```
class BankAccount
  attr_reader :balance

  def  credit(amount)
    @balance += amount
  end
  def  debit(amount)
    @balance -= amount
  end
  …
end
```

But, real applications need:



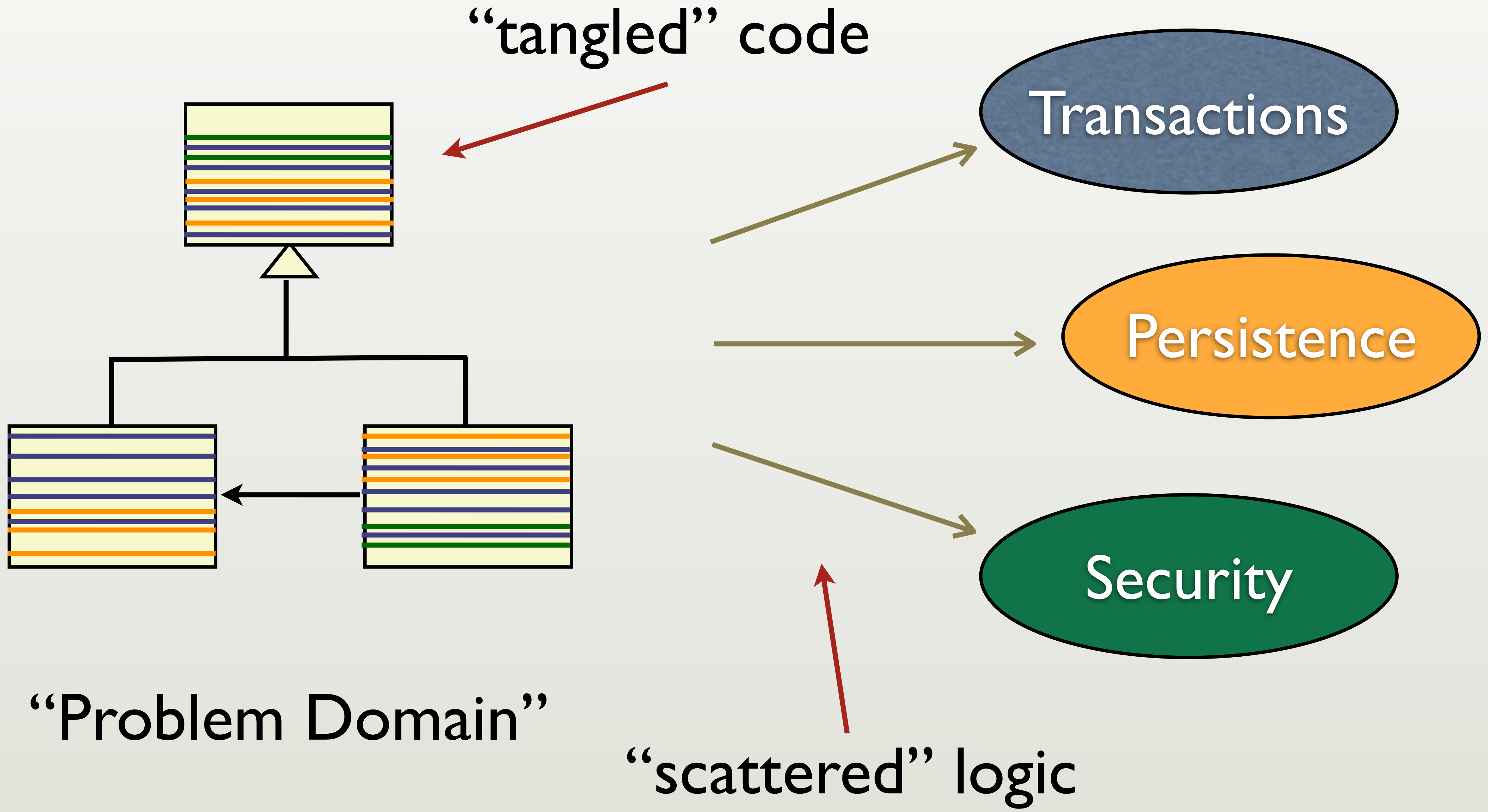
```
def BankAccount
  attr_reader :balance
  def  credit(amount)
    ...
  end
  def  debit(amount)
    ...
  end
end
```

So `credit` becomes…

```
def credit(amount)
  raise "…" if unauthorized()
  save_balance = @balance
  begin
    begin_transaction()
    @balance += amount
    persist_balance(@balance)
  …
```

```
    …
    rescue => error
      log(error)
      @balance = saved_balance
    ensure
      end_transaction()
    end
  end
```

We're mixing *multiple domains*, with fine-grained *intersections*.

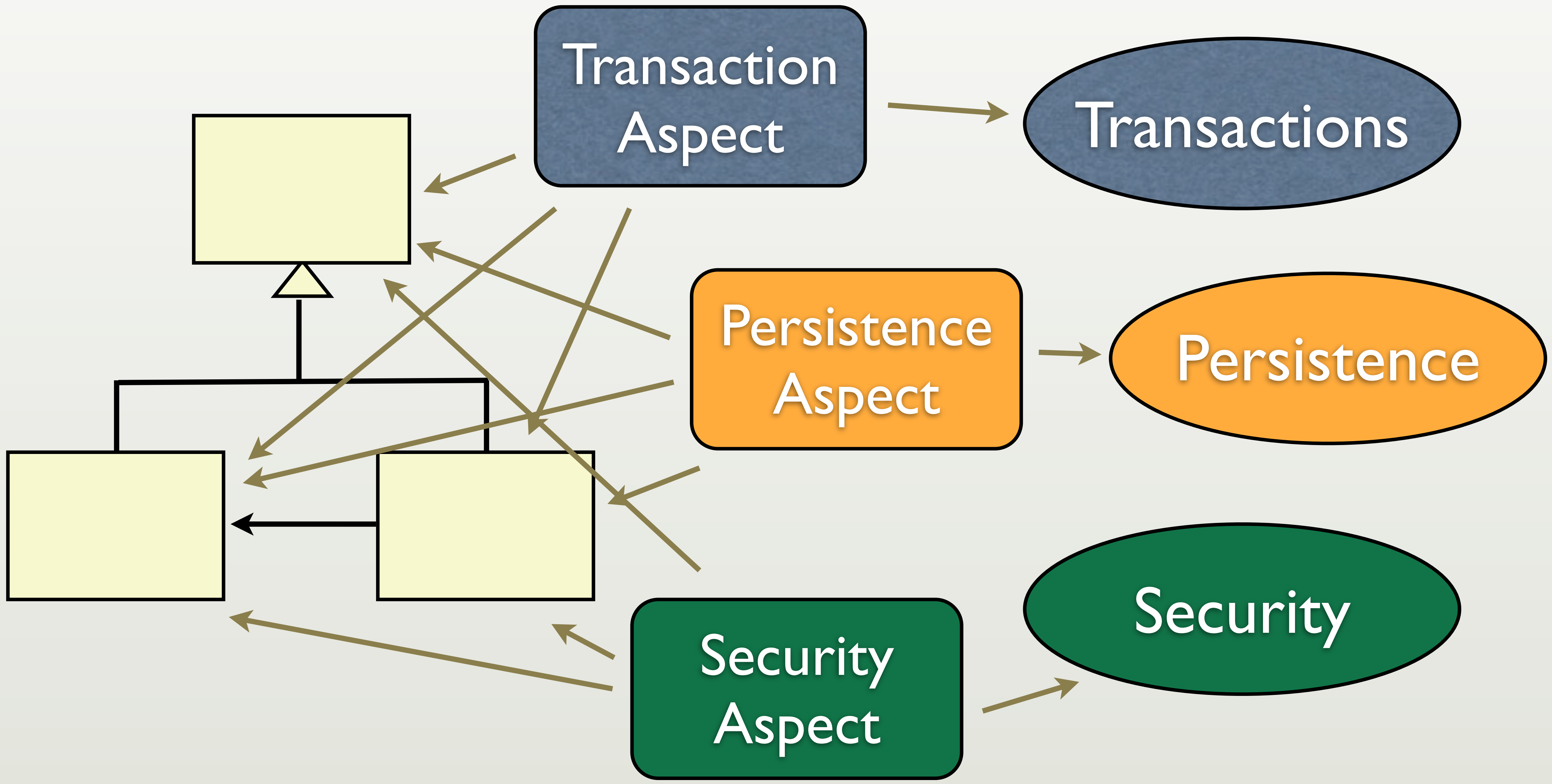


In principle, I can reason about transactions, etc. in isolation, but in reality, the code for transactions is scattered over the whole system. Similarly, the once-clean domain model code is tangled with code from the other concerns.
Objects don't prevent this problem (in most cases).

Objects alone *don't* prevent *tangling*.

Aspect-Oriented Programming: restore *modularity* for *cross-cutting concerns*.

Aspects restore *modularity* by encapsulating the *intersections*.



See “extra” slides

If you have used the *Spring Framework*, *you have* used *aspects*.

Parting Thought...

Metaprogramming can be used for some *aspect-like* functionality.

DSLs can solve some *cross-cutting concerns*, by localizing behaviors expressed by the DSL.

Our service must be *available 24x7* and highly *scalable*.

Problem #4

Symptoms

- Only *one* of our developers *really knows* how to write *thread-safe* code.
- The system *freezes* every few *weeks* or so.

Solution

Functional Programming

(At least, it’s one solution...)

Functional Programming

Modeled after *mathematics*.

```
y = sin(x)
```

Functional Programming

Values are *immutable*.
Variables are assigned *once*.

```
y = sin(x)
```

Functional Programming

Functions are *side-effect free*.
Functions don’t alter *state*.
The *result* depends *solely*
on the *arguments*.

```
y = sin(x)
```

Functional Programming: *Concurrency* Is *Easier*

No *writes*, so no *synchronization*.
Hence, *no* locks, semaphores, mutexes...

```
y = sin(x)
```

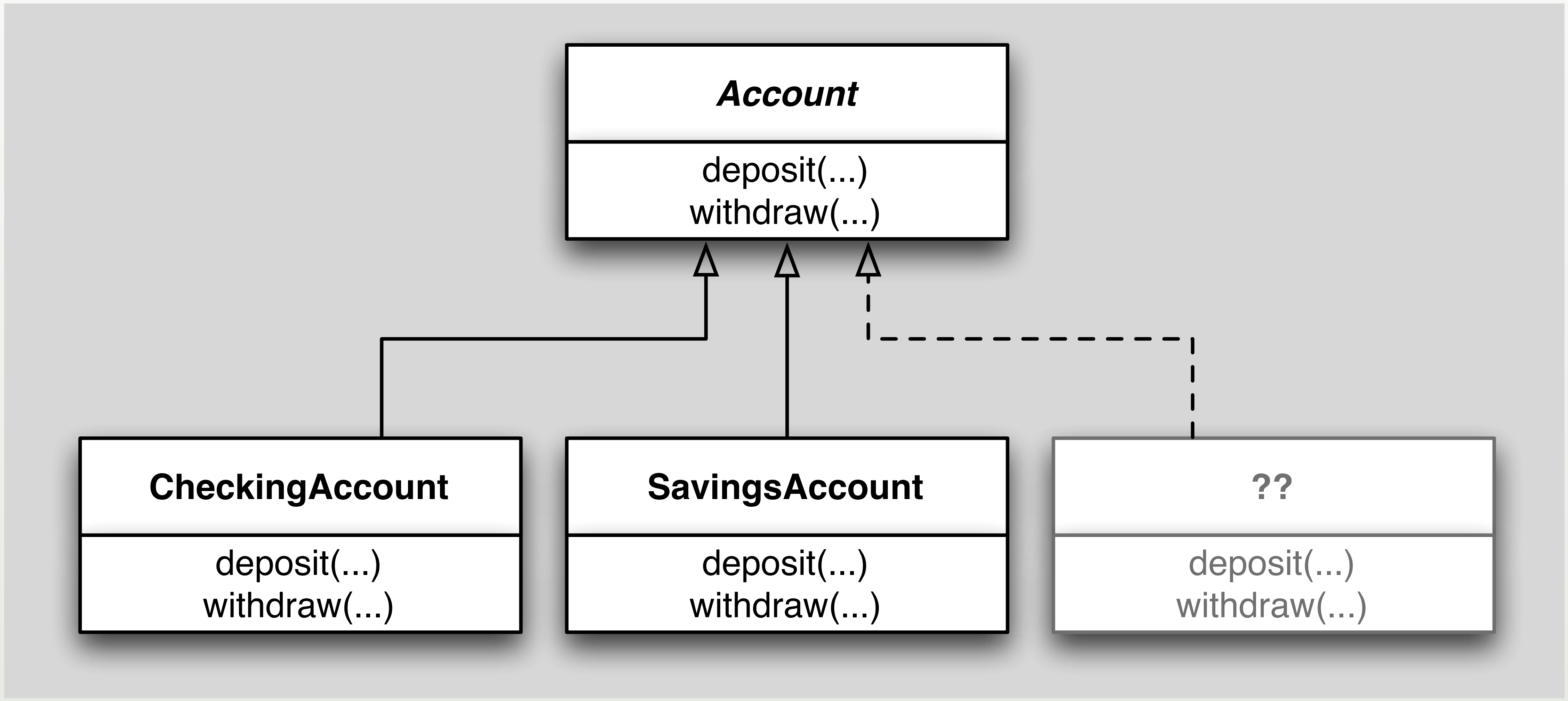
Functional Programming: *Reasoning* is *Easier*

Without *side effects*,
functions are *easier* to *test*, *understand*, ...

and *reuse*!

```
y = sin(x)
```

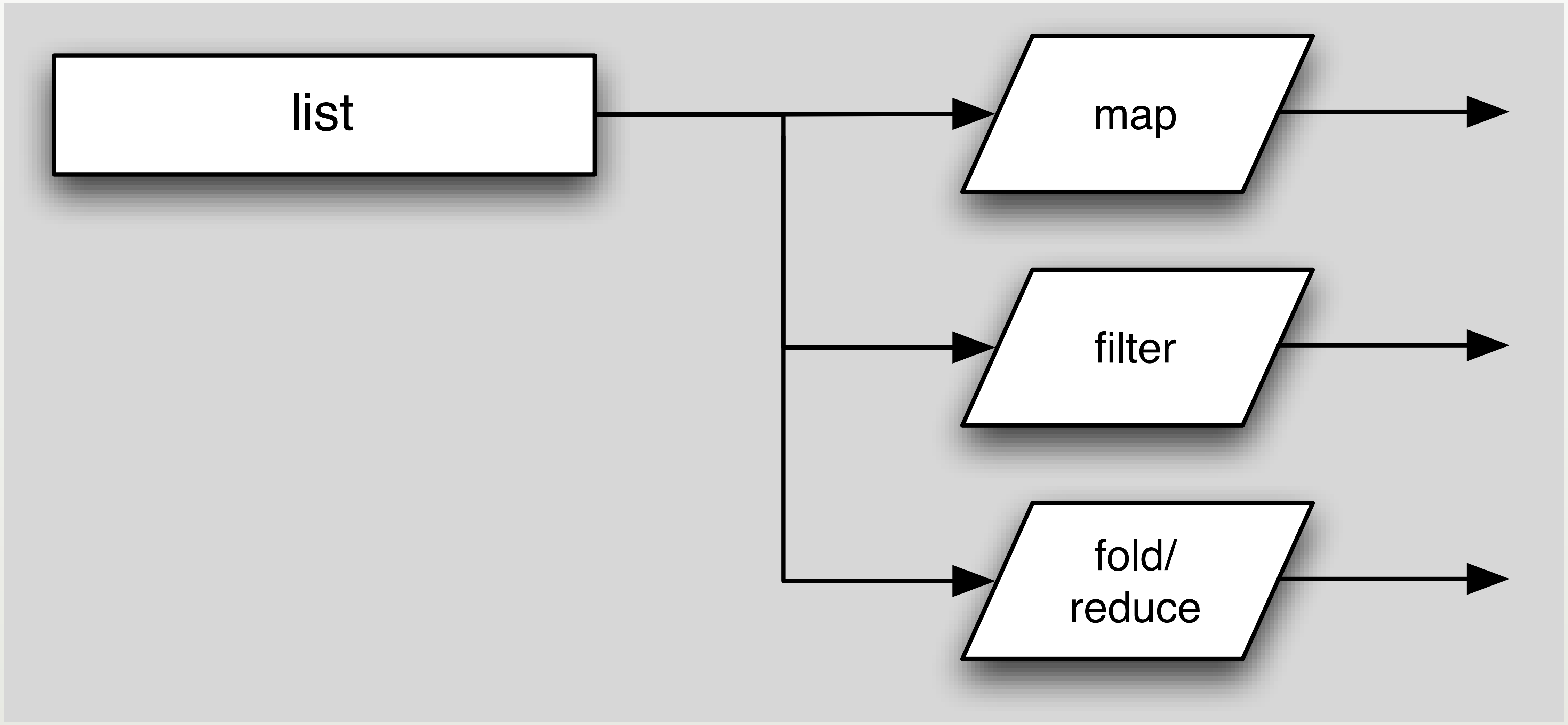
Which fits your needs?



Object Oriented

Do operations vary significantly, depending on data type or ...

Which fits your needs?



Functional

… or do operations more or less work the same independent of the data type?

What if you’re doing *cloud computing*?

E.g., is map-reduce object-oriented or functional?

FP Code:
more *declarative* than *imperative*.

```
F(n) = F(n-1) + F(n-2)
where: F(0) = 0 and F(1) = 1
```

```
0, 1, 1, 2, 3, 5, 8, 13, ...
```

The Fibonacci Sequence.
I tell the system what I want (e.g., what are the relationships between data, the constraints, etc.) and let the system figure out how to do it.

… and so are DSLs.

```
class Customer < ActiveRecord::Base
 has_many :accounts
 validates_uniqueness_of :name,
    :on => create,
    :message => ‘Evil twin!’
end
```

By hiding the implementation details, we have much more leeway in implementing aspect behavior, etc.

A Few *Functional* Languages

Haskell

```
module Main where
-- Function f returns the n'th Fibonacci number.
-- It uses binary recursion.
f n | n <= 2 = 1
    | n >  2 = f (n-1) + f (n-2)
-- Print the Fibonacci number F(8)
main = print(show (f 8))
```

x

Note how closely the definition reads compared to the mathematical definition I presented earlier.

Erlang

- Ericsson Functional Language.
- For distributed, reliable, *soft* real-time, *highly* concurrent systems.
- Used in telecom switches.
  - *9-9's reliability* for AXD301 switch.

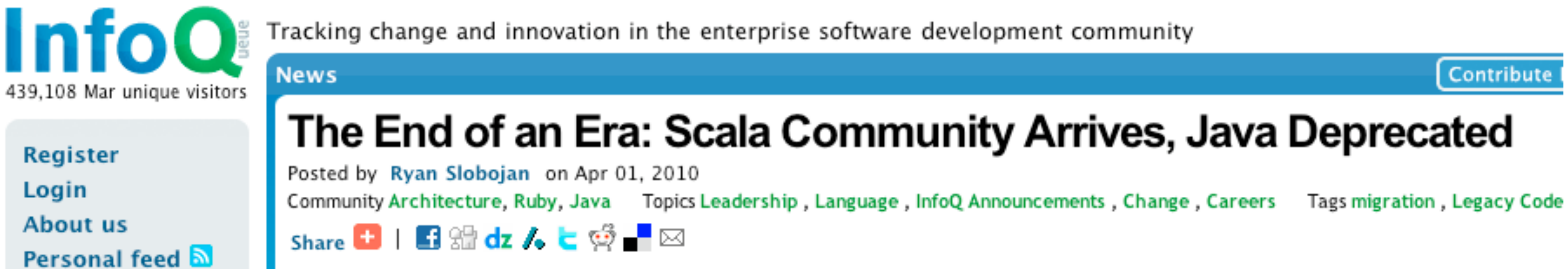
Erlang

- No *mutable variables* and *side effects*.
- Uses the *actor model* of concurrency.
  - All IPC is optimized *message passing*.
  - *Let it fail* philosophy.
- *Very* lightweight and fast *processes*.
  - Lighter than most OS threads.

Scala

- Hybrid: *object* and *functional*.
- Targets the *JVM* and *.NET*.
- *"Endorsed" by James Gosling* at JavaOne.
- Could be the most popular *replacement* for Java.

Times Change...



InfoQ
439,108 Mar unique visitors

Tracking change and innovation in the enterprise software development community

Register
Login
About us
Personal feed

News

Contribute

The End of an Era: Scala Community Arrives, Java Deprecated

Posted by Ryan Slobojan on Apr 01, 2010

Community Architecture, Ruby, Java Topics Leadership , Language , InfoQ Announcements , Change , Careers Tags migration , Legacy Code

Share

...

Dean Wampler, Ph.D., the co-author of O'Reilly's "Programming Scala", offered this comment on the sudden industry switch to Scala vs. the less appealing alternatives:

> We all know that object-oriented programming is dead and buried. Scala gives you a 'grace period'; you can use its deprecated support for objects until you've ported your code to use Monads.

Note that date on this InfoQ post...

Clojure

- *Functional*, with *principled* support for mutability.
- Targets the *JVM* and *.NET*.
- Best *buzz*?
- Too many *good ideas* to name here...

Functional Languages in Industry

- *Erlang*
  - *CouchDB*, *Basho Riak*, and Amazon's *Simple DB*.
  - *GitHub*
  - Jabber/XMPP server *ejabberd*.

Functional Languages in Industry

- *OCaml*
  - Jane Street Capital
- *Scala*
  - Twitter
  - LinkedIn
- *Clojure*
  - Flightcaster

Parting Thought...

Which is better:
A *hybrid object-functional* language
for everything?
An *object* language for some code and
a *functional* language for other code?

e.g., Scala *vs.* Java + Erlang??

Scala is more complex than "mono-paradigm" languages, so it's harder to master. However, using multiple languages has it's own challenges.

Recap:

Polyglot and *Poly-paradigm Programming* (PPP)

Disadvantages of PPP

- *N* tool chains, languages, libraries, “ecosystems”, idioms, ...
- *Impedance mismatch* between tools.
  - Different *meta-models*.
  - *Overhead* of calls between languages.

Advantages of PPP

- Can use the *best tool* for a *particular job*.
- Can *minimize* the *amount* of code required.
- Can keep code *closer* to the domain using DSLs.
- Encourages *thinking* about *architecture*.

Is This *New*?

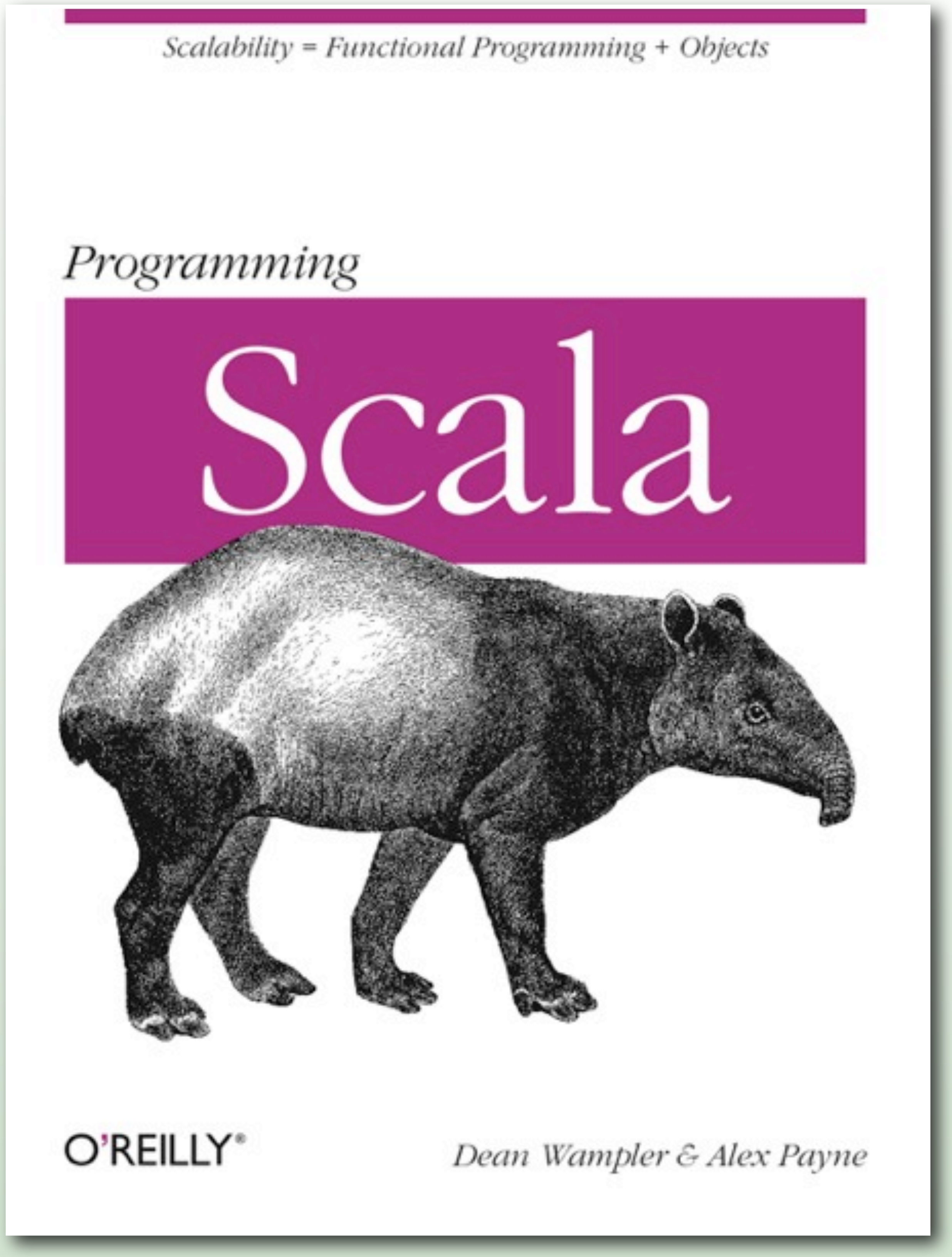
- *Functional Programming Comes of Age.*
  - Dr. Dobbs, 1994
- *Scripting: Higher Level Programming for the 21st Century.*
  - *IEEE Computer, 1998*
- *In Praise of Scripting: Real Programming Pragmatism.*
  - *IEEE Computer, 2008*

Why go *mainstream* now?

- *Rapidly increasing* pace of development,
  - ➔ Scripting (dynamic languages), DSLs.
- *Pervasive concurrency* (e.g., *Multicore CPUs*)
  - ➔ Functional programming.
- *Cross-cutting concerns*
  - ➔ Aspect-oriented programming.

Thank You!

- dean@deanwampler.com
- @deanwampler
- polyglotprogramming.com
- thinkbiganalytics.com


Scalability = Functional Programming + Objects
Programming
Scala
O'REILLY
Dean Wampler & Alex Payne

Extra Slides

Aspect-Oriented Tools

- Java
  - AspectJ
  - Spring AOP
  - JBoss AOP

shameless plug

- Ruby
  - Aquarium
  - Facets
  - AspectR

Options for Java and Ruby. Some other languages have AOP toolkits.

I would like to write…

Before returning the *balance*, read the current value from the database.

After setting the *balance*, write the current value to the database.

Before accessing the *BankAccount*, authenticate and authorize the user.

I would like to write…

Before returning the *balance*, read the current value from the database.

After setting the *balance*, write the current value to the database.

Before accessing the *BankAccount*, authenticate and authorize the user.

Aquarium

```
require 'aquarium'
class BankAccount
  …
  after :writing => :balance \
      do |context, account, *args|
    persist_balance account
  end
  …
```

use aquarium lib.

reopen class

"event" to trigger on

new behavior

aquarium.rubyforge.org

Back to *clean code*

```
def credit(amount)
  @balance += amount
end
```

Common Themes

- *Less* code is *more*.
- Keep the code *close* to the *domain*: DSLs.
- Be *declarative* rather than *imperative*.
- *Minimize* side effects and mutable data.