Radical Simplification through Polyglot and Poly-paradigm Programming

Dean Wampler
dean@objectmentor.com
Object Mentor, Inc.

Object Mentor
Industry experts in software best practices

I am speaking at QCon 2008
San Francisco Nov. 17 - 21

Polyglot:

many languages

Poly-paradigm:

many modularity paradigms

Today's applications:

- Are *networked*,
- Have graphical and "service" *interfaces*,
- *Persist* information,
- Must be *resilient* and *secure*,
- Must *scale*,
- ... and must do all that by *next Friday*.

twitter.com/photos/jerryjohn

Mono-paradigm:

Object-Oriented Programming:

right for all problems?

Monolingual

Is one *language*
best for all domains?

twitter.com/photos/watchsmart

Symptoms of *Monocultures*

- *Why is there so much XML in my Java?*
- *Why do I have similar persistence code scattered all over my code base?*
- *I can't scale my application by a factor of 1000!*
- *My application isn't extensible enough!*
- *I can't respond quickly enough when requirements change!*

```
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;
    } ...
```

Pervasive IT problem:

Too much code!

thedailywtf.com

Solutions

The symptoms reflect
common root *problems*
with *similar solutions*.

I need *extensibility* and *agility*.

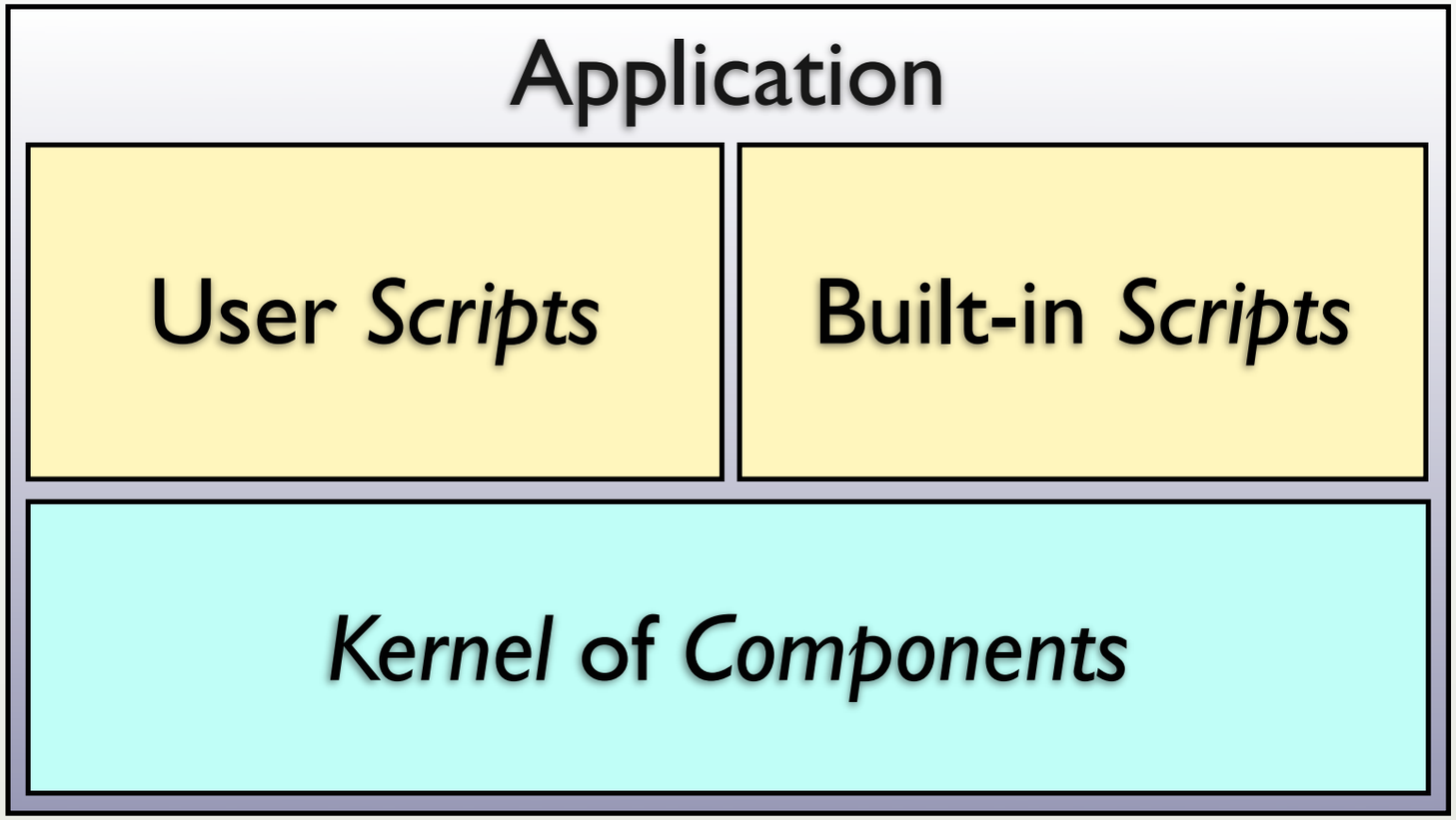
Specific problem #1

twitter.com/photos/arrrika

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

Kernel Components

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

Scripts

- Written in a *dynamically-typed language*.
  - Ruby, Python, JavaScript, Lua, Perl, Tcl, ...
- *Interpreted* for extensibility and agility.
  - Runtime *performance* is less important.
- Glue together components.
- Higher developer *productivity*.

Ola Bini's *Three Layers*

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

Other Examples

- *UNIX/Linux* + *shells*.
  - Also *find*, *make*, *grep*, ...
    - Have their own *DSL's*.
- *Tektronix Oscilloscopes*: C + Smalltalk.

Other Examples

- *Adobe Lightroom*: C++ + Lua.
  - 40-50% written in Lua.
- *NRAO Telescopes*: C + Python.
- *Google Android*: Linux+libraries (C) + Java.

XML in Java

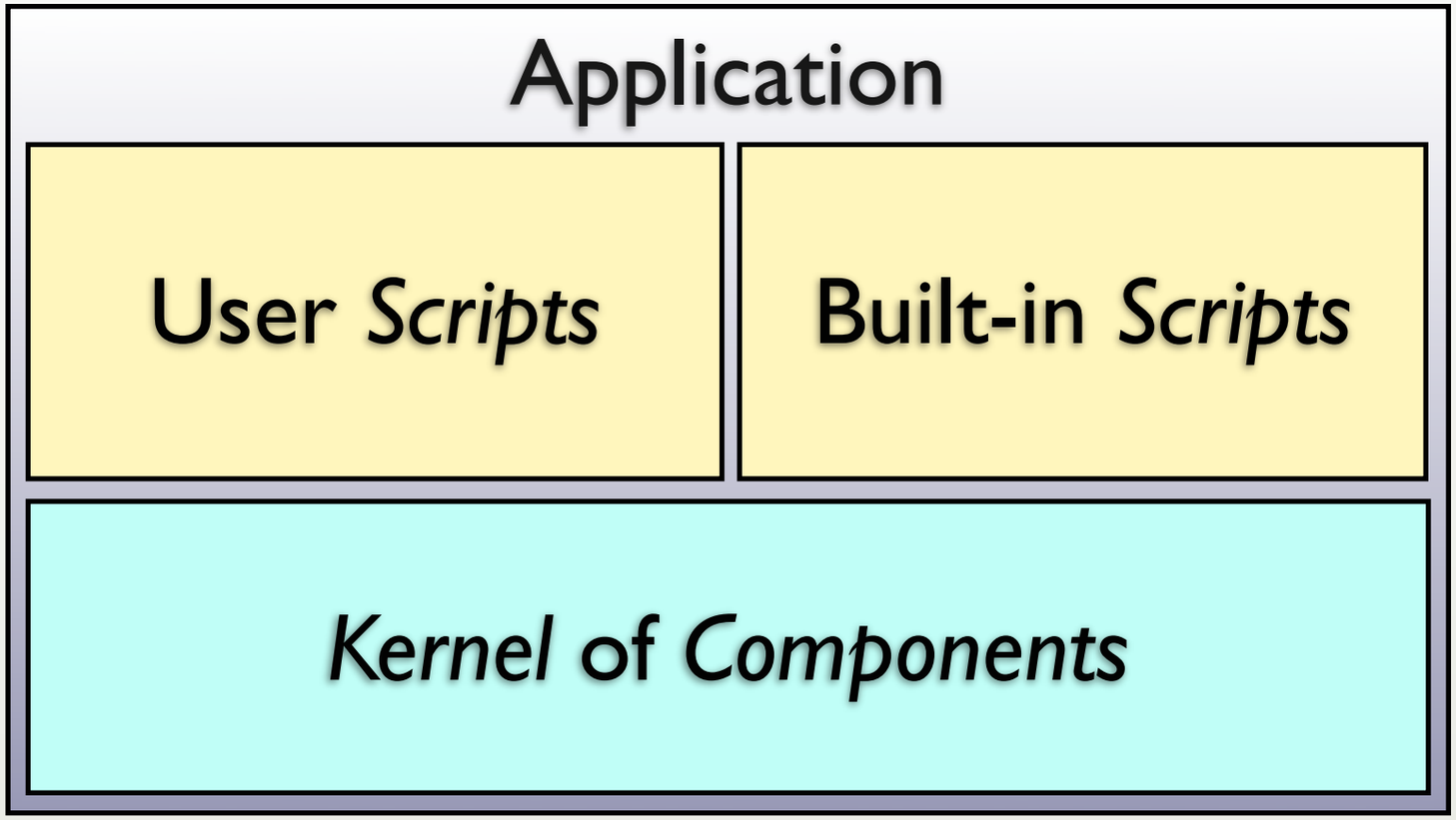
Why not *replace* XML with *JavaScript*, *Groovy* or *JRuby*??

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

Multilingual VM's

- *Jython*, *JRuby*, *Groovy*, *Scala*.
  - On the *JVM*.
  - Ruby on Rails on JRuby (*Oracle Mix*).
- *Dynamic Language Runtime* (DLR).
  - Ruby, Python, ... on the *.NET CLR*.

Benefits

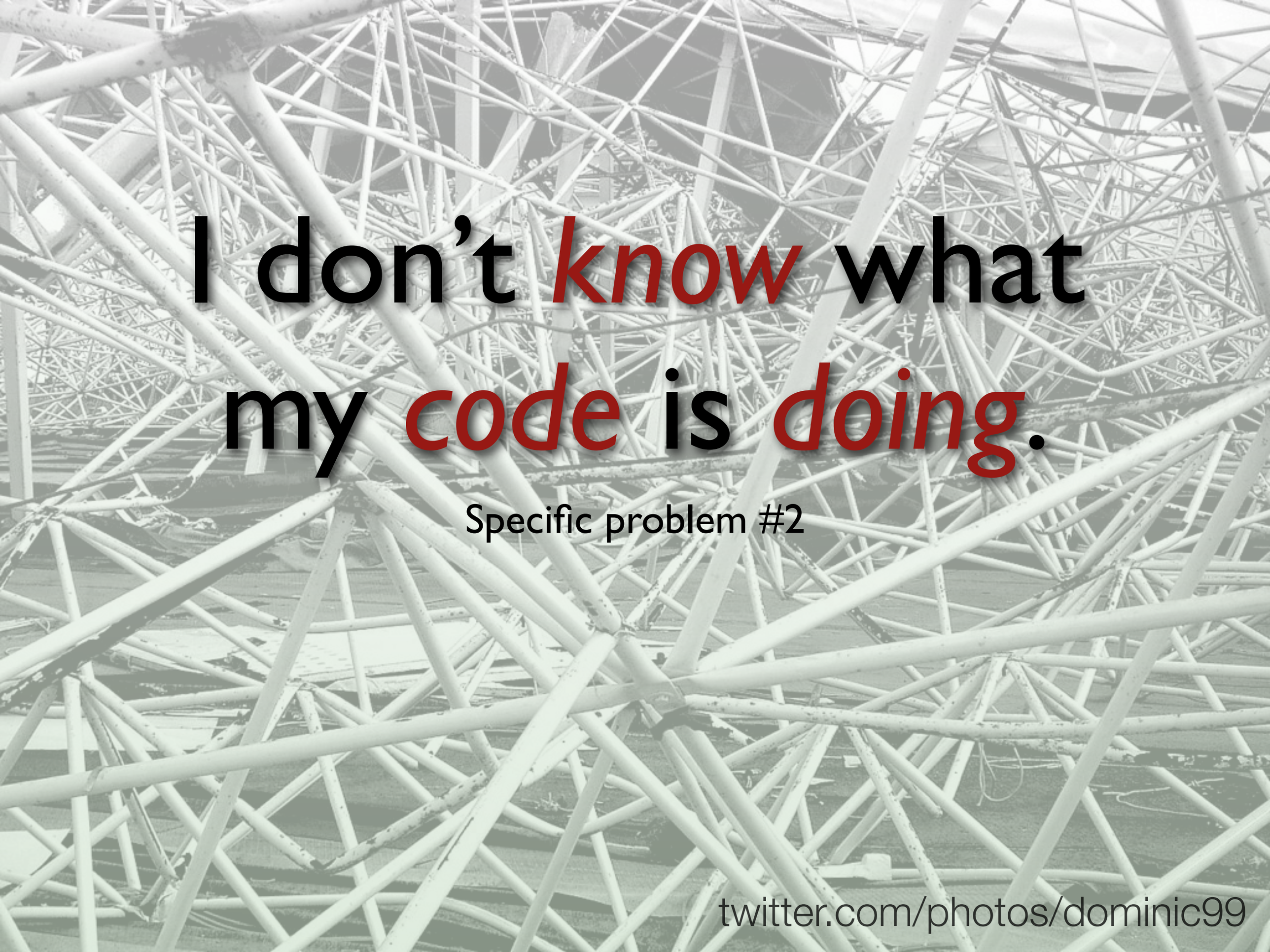


- Optimize *performance* where it matters.
- Optimize *productivity*, *extensibility* and *agility* everywhere else.

Parting Thought...

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

(Why *IPhone* and *Android* are interesting.)


I don't *know* what
my *code* is *doing*.
Specific problem #2
twitter.com/photos/dominic99

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

Symptoms

- The *Business logic* doesn't *jump out* at me when I *read* the code.
- The system *breaks* when we *change* it.
- Translating *requirements* to *code* is *error prone*.

Solution #1

Write
less code.

Profound statement.

Less Code

- Means *problems* are *smaller*:
  - Maintenance
  - Duplication (DRY)
  - 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

+

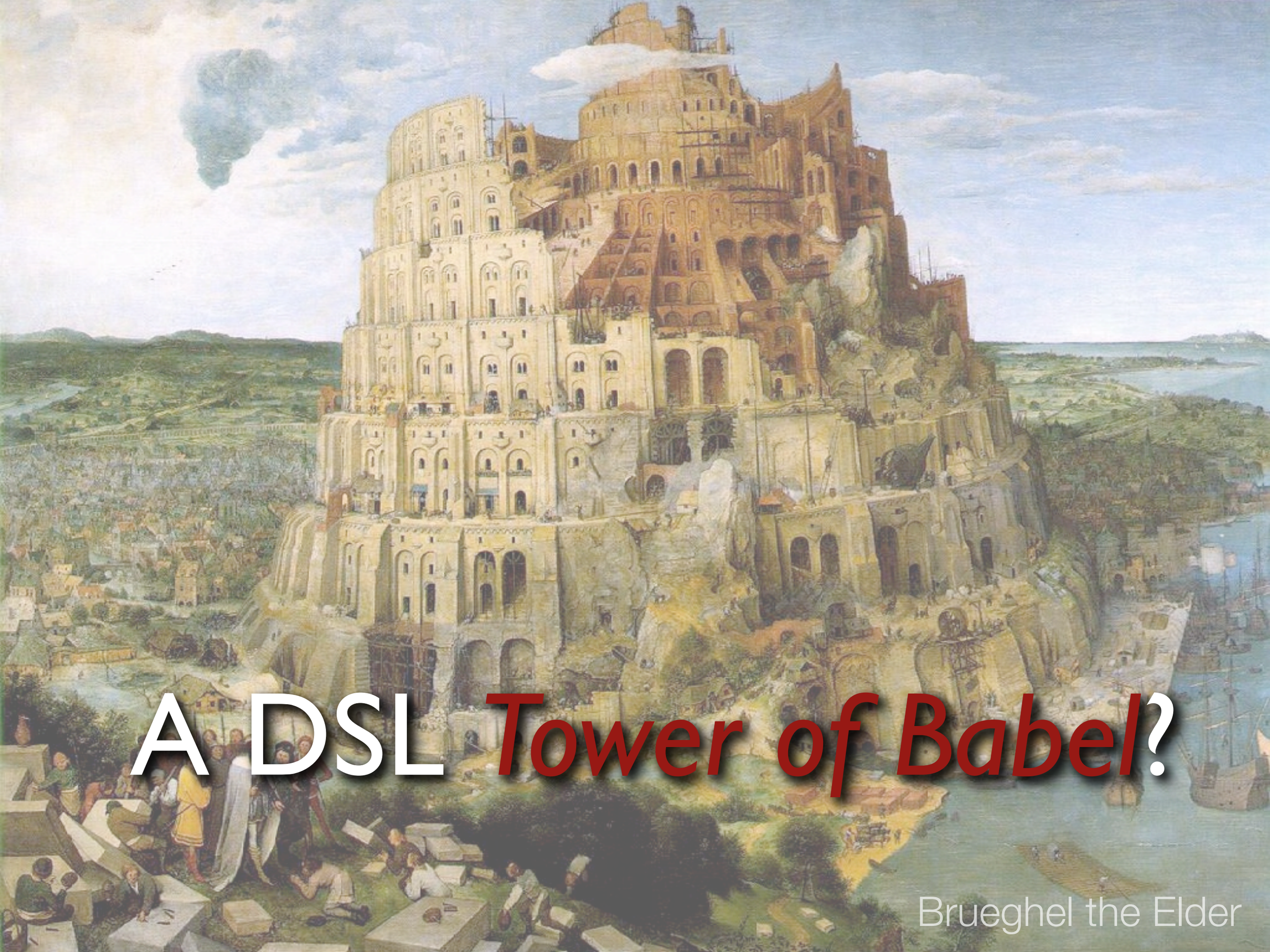
...

DSL Advantages

- When code *looks* like domain prose,
  - It is easier to understand by *everyone*,
  - It is easier to *align* with the *requirements*,
  - It is more *succinct*.

DSL Disadvantages

- DSL's are *hard* to *design*, *test* and *debug*.
- Some people are *bad* API designers,
  - They will be even *worse* DSL designers!


A DSL *Tower of Babel*?
Brueghel the Elder

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

Parting Thought #2...

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

-- Albert Einstein

Corollary:

Entia non sunt multiplicanda praeter necessitatem.

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

-- Occam's Razor

We have *code duplication* everywhere.

Specific 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.
  - *DSL's*.
  - *Aspects*.

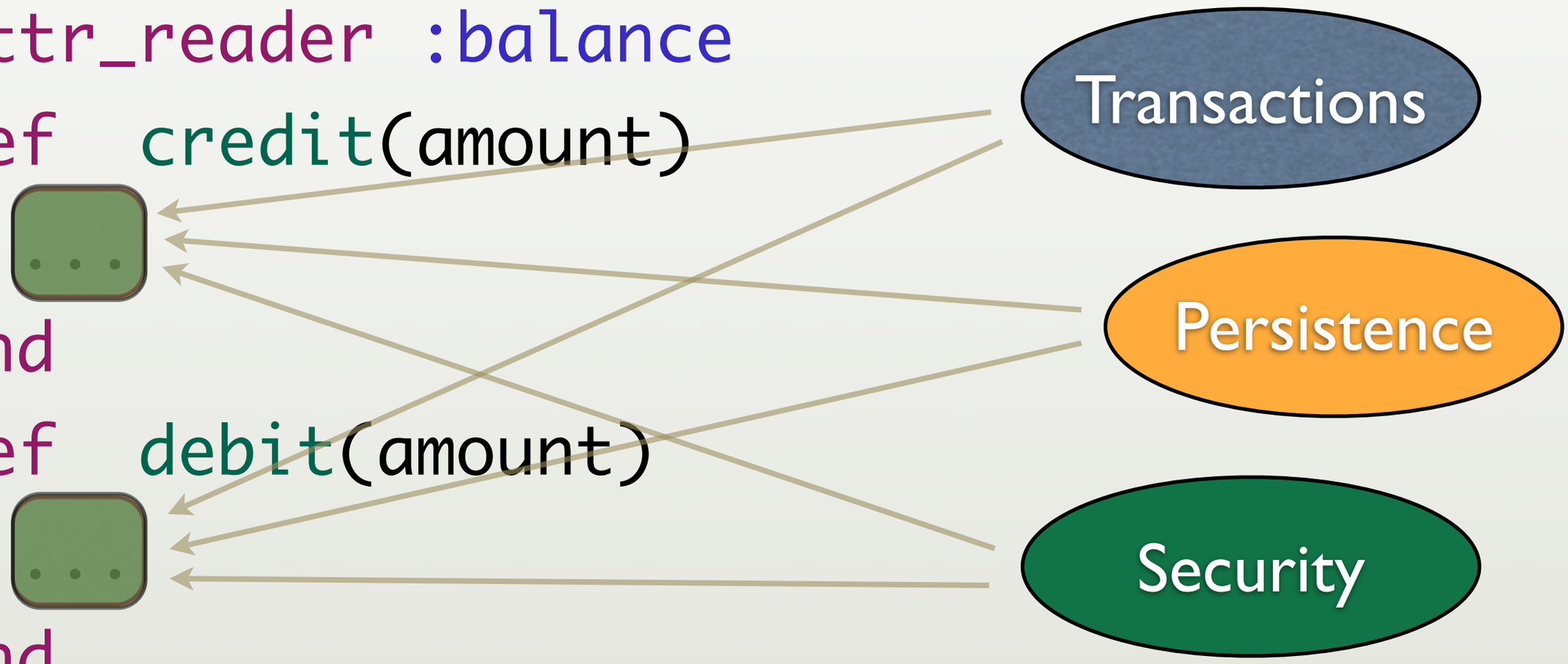
An Example...

```ruby
class BankAccount
  attr_reader :balance

  def  credit(amount)
    @balance += amount
  end

  def  debit(amount)
    @balance -= amount
  end

  …
end
```

Clean Code

But, real applications need:

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

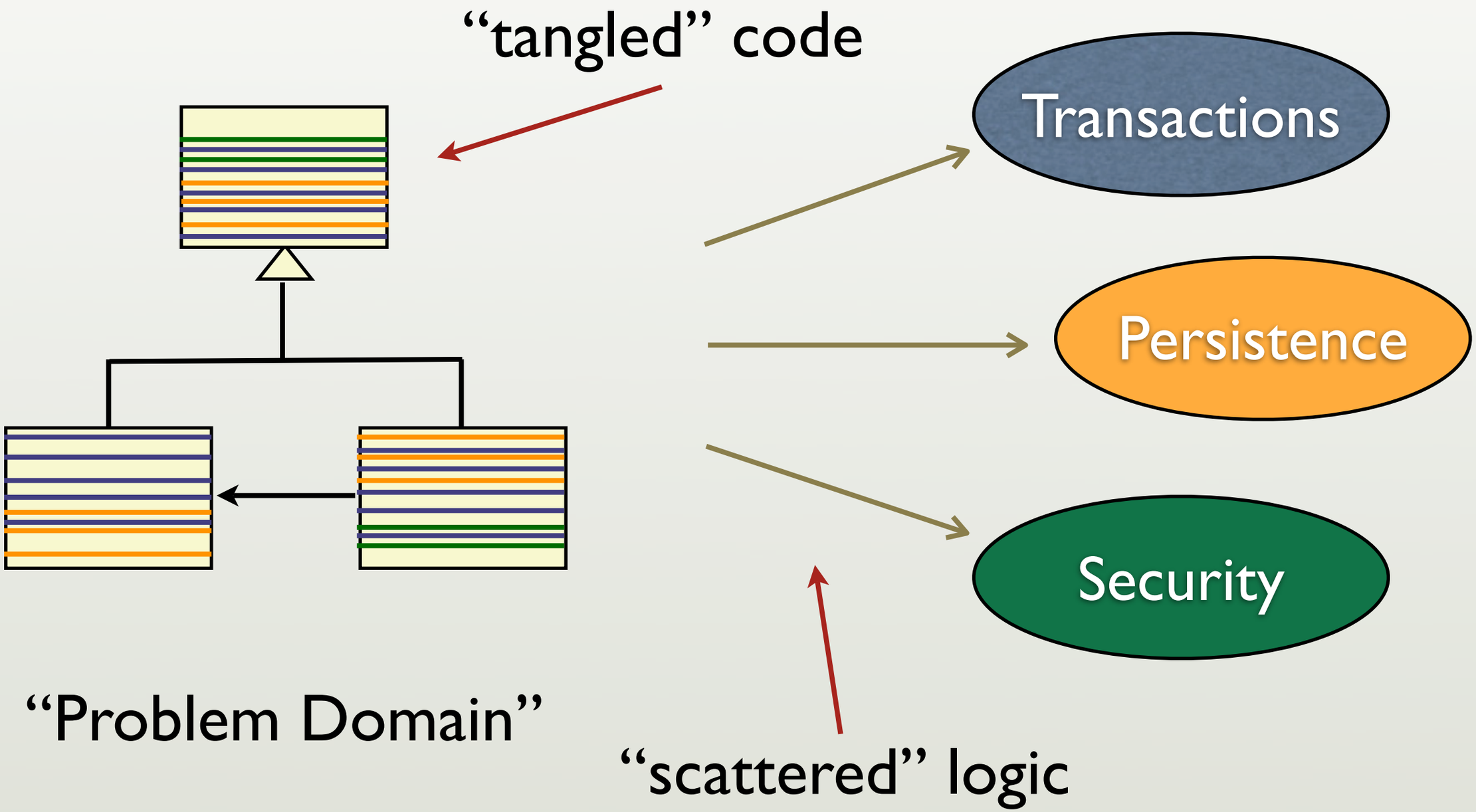


So credit becomes…

```
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*.



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
  ...
```

reopen class

add new behavior

aquarium.rubyforge.org

Back to *clean code*

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

Parting Thought...

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

DSL's can solve some *CCC*.
(We'll come back to that.)

Our application must be *available* 24 x 7 and highly *concurrent*.

Specific problem #4

twitter.com/photos/wolfro54

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

Functional Programming

- Works like *mathematical functions*.

  Fibonacci Numbers:

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

Functional Programming

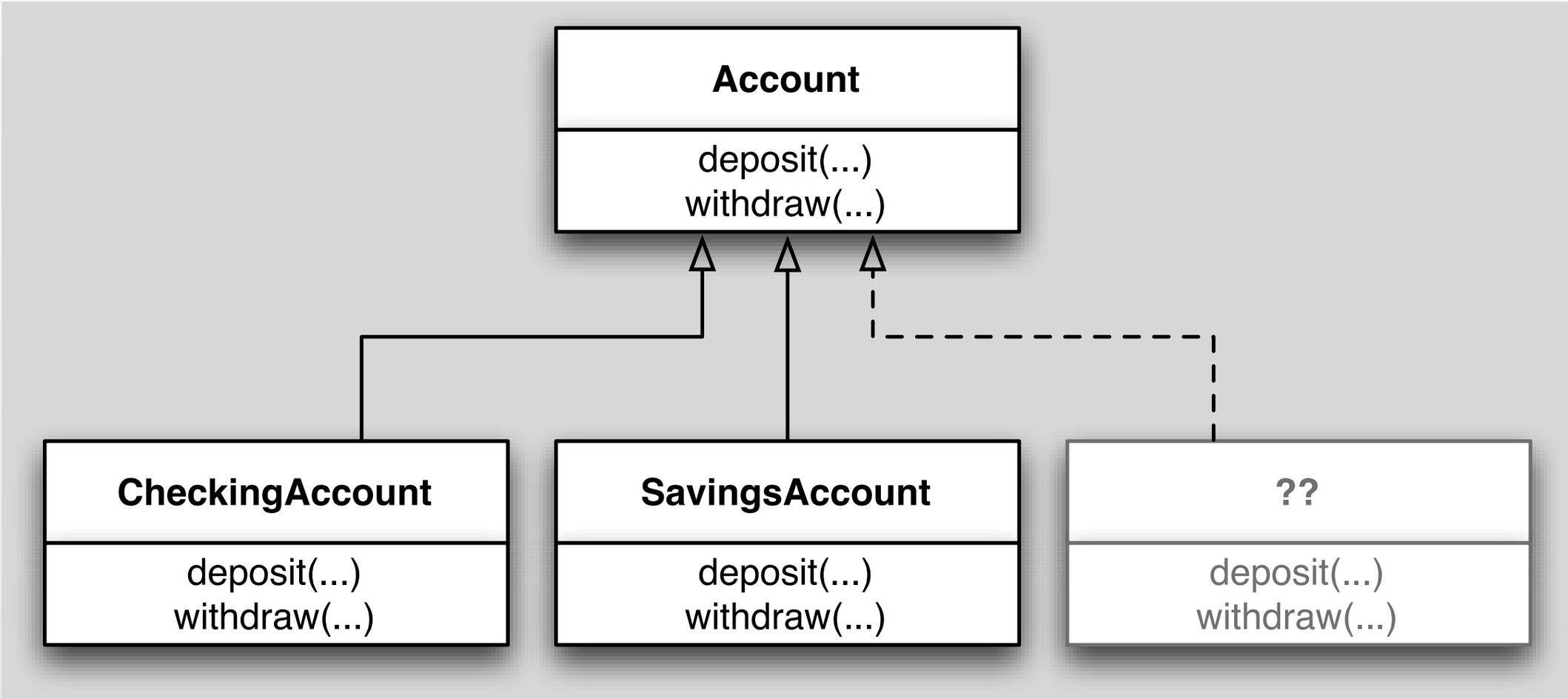
```
y = sin(x)
```

- Variables are assigned *once*.
- Functions are *side-effect free*.
  - They don't alter *state*.

Functional Programming Makes *Concurrency Easier*

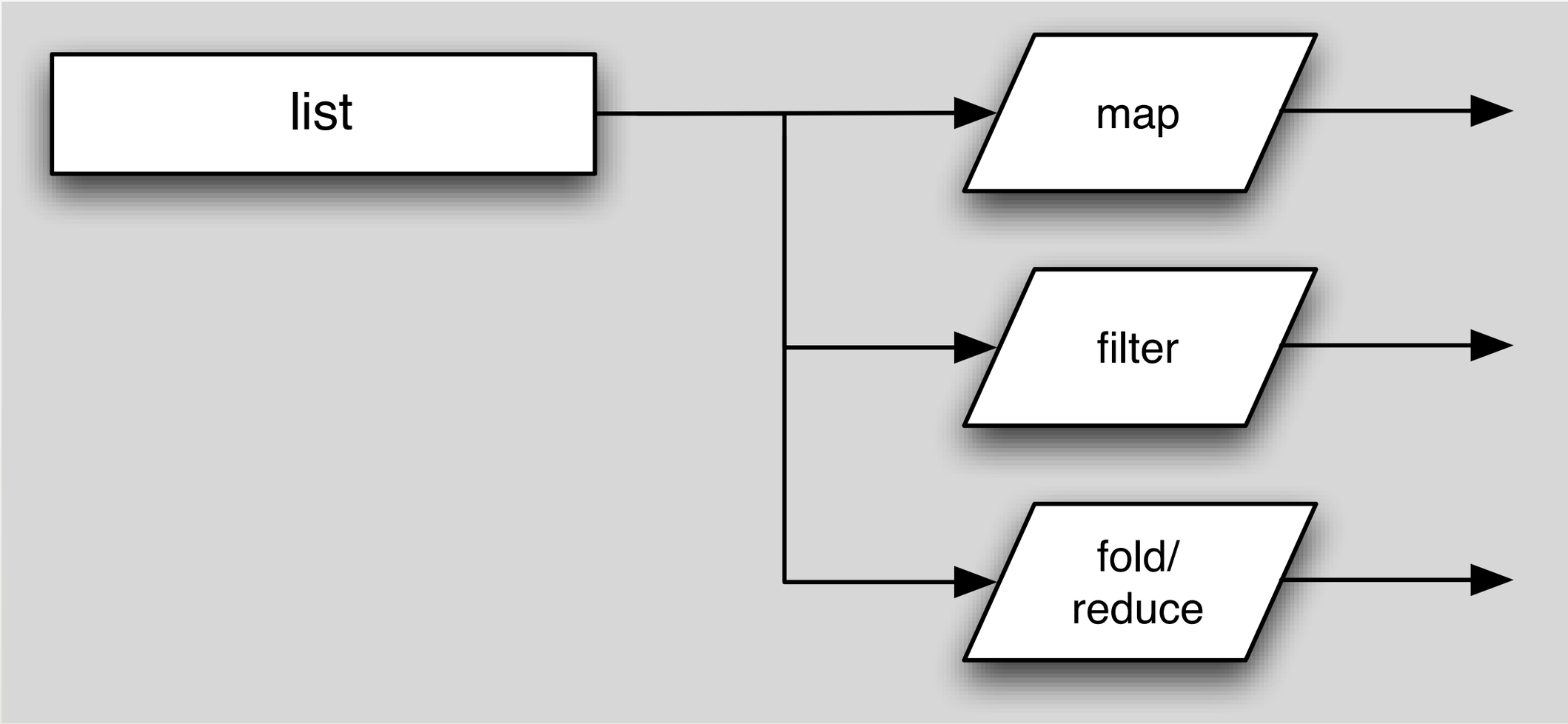
- *Nothing* to *synchronize*.
- Hence *no* locks, semaphores, mutexes...

Which fits your needs?



Object Oriented

Which fits your needs?



Functional

What if you're doing *cloud computing*?

twitter.com/photos/deanwampler

Declarative
rather than
imperative.

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

… and so are DSL's.

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

A Few Functional Languages

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*.
- All IPC is optimized *message passing*.
- *Very* lightweight and fast *processes*.
  - Lighter than most OS threads.

Scala

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

shameless plug

Parting Thought...

Is a *hybrid object-functional* language better than using an *object* language with a *functional* language??

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

Recap:

Simplification through *Polyglot* and *Poly-paradigm Programming* (PPP)

Disadvantages of PPP

- *N* tool chains, languages, libraries, “ecosystems”, ...
- *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.
- Encourages *thinking* about *architectures*.
  - *E.g.*, *decoupling* between "components".

Everything *old* is *new* *again.*

- *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 with dynamic languages?

- *Pervasive concurrency* (e.g., *Multicore CPUs*)

  ➔ Functional programming?

- *Cross-cutting concerns*

  ➔ Aspect-oriented programming?

Common Threads

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

Thank You!

- dean@objectmentor.com
  - Watch for my *Scala* book.
- http://blog.objectmentor.com
- http://polyglotprogramming.com/papers