Aquarium: AOP for Ruby

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

AOSD 2008
April 3, 2008

Goals and Features

- Provide an *intuitive* syntax.
- Support *runtime addition* and *removal* of advice.
- Advise *Java* through *JRuby*.
- Demonstrate the value of AOP in *dynamically-typed* languages.

Why Ruby??

- It's what the *cool* kids are using.
- *"Revenge of the Smalltalkers."*

Language trends…
and waves of innovation

- Late 80's - early 90's: *C++*
- Late 90's: *Java*
- Late 00's: *Ruby*

Groovy *vs.* Ruby

- *Groovy* might be better for *advising Java*.
- *Ruby* is better, otherwise. ☺

Provide an *intuitive* syntax.

Domain-specific language for *aspect-like* behavior?

```ruby
class BankAccount
  attr_reader :balance
  def initialize
    @balance = 0
  end
  def deposit(amount)
    @balance += amount
  end
  def withdraw(amount)
    @balance -= amount
  end
end
```

creates getter `balance()`

`@balance` *attribute*

Let’s add a persistence aspect...

The requirements are:

Before reading the balance,
read the current value from the database.

After changing the balance,
write the current value to the database.

Before accessing the account,
authenticate and authorize the user.

Can I “compile” those requirements?

Aquarium: AOP for Ruby

```
class BankAccount
  ...
  before \
    (:calling => [:deposit, :withdraw]) \
      do |...|
        # read object state from database
  end
  ...
```

reopen the class

advice type

pointcut

"," works like "or"

advice (do ... end block)

2nd Requirement

```
...
after \                                    :calling => [:deposit, :withdraw] \
    do |...|
    # write object state to database
end
...
```

advice type

pointcut

advice

3rd Requirement

```
…
before \
    :calling => [:deposit, :withdraw], \
    :accessing => :balance \
  do |…|
    raise "…" unless user_permitted
  end
end
```

pointcut *"," works like "or"*

Small print...

include library

```
require “aquarium”
class BankAccount
  include Aquarium::AspectDSL
```

add methods to class

```
  before … do |jp, object, *args|
    …
  end
```

join point context

active object

parameters passed to method

Runtime addition and *removal* of advice.

Not limited to static weaving...

Temporary aspects

```
foo = FooBar.new(…)
foo.non_critical_method
aspect = Aspect.new :before, \
  :calls_to => :all_methods, \
  :in_object => foo do |join_point|
    puts “Entering #{join_point}”
end
foo.critical_method   # output happens...
aspect.unadvise       # stop the output...
```

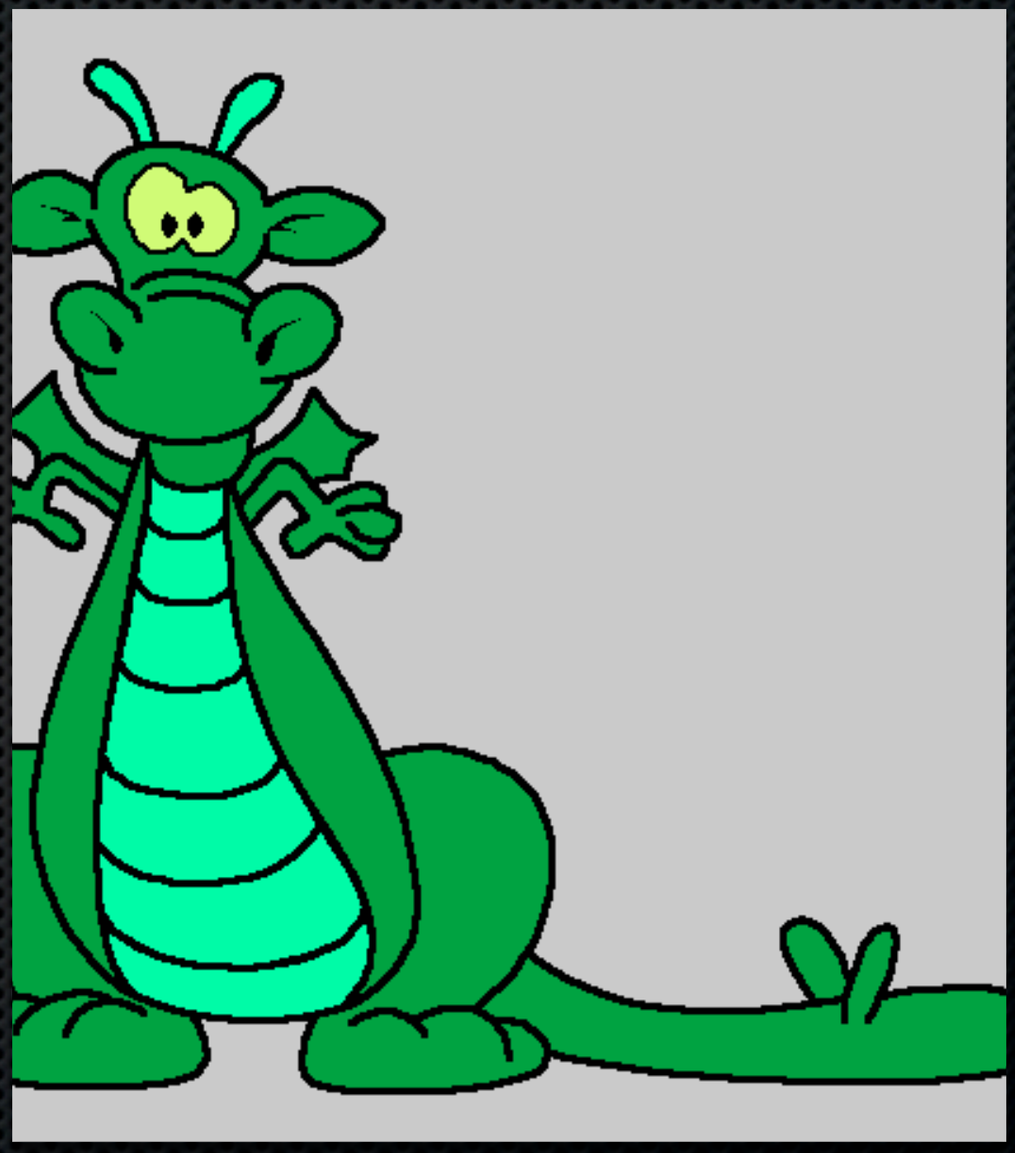
Advise *Java* thru *JRuby*.

The performance of Java,

the flexible power of Ruby.

Hic sunt dracones

Bleeding-edge, juggling-knives approach to Java AOP...



Java aspects in Ruby!

```
foo = Java::com.demo.FooBar.new(…)
aspect = Aspect.new :before, \
  :calls_to => :critical_method, \
  :in_object => foo do |join_point|
    puts “Entering #{join_point}”
end
foo.critical_method
aspect.unadvise
```

AOP for *dynamically-typed* languages.

Metaprogramming isn't enough.

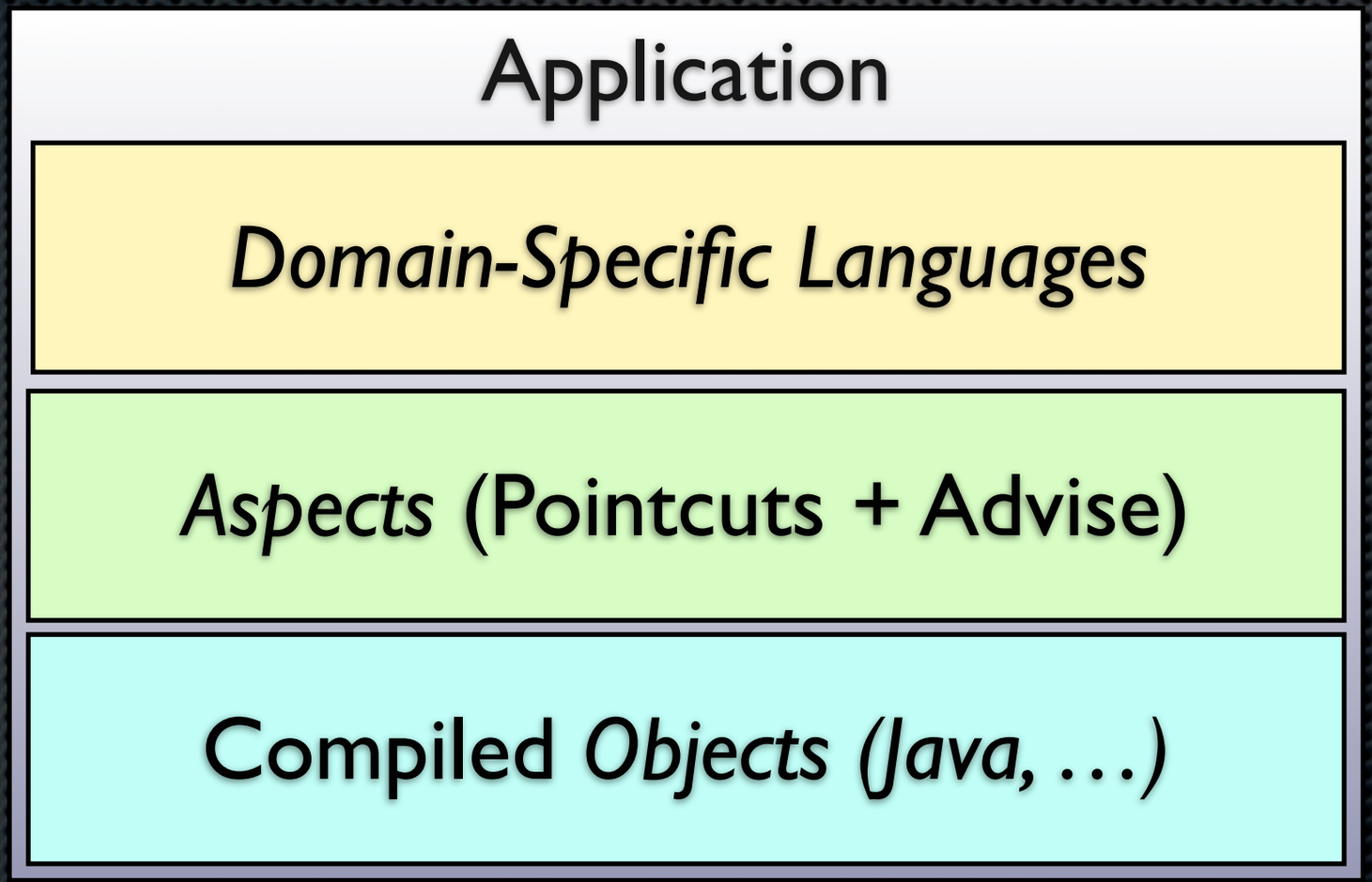
Drawbacks of metaprogramming alone

- Have to map AOP *design* to metaprogramming *code*.
- No *Pointcut Language*.

The future of aspects...

- Radical statements:
  - Languages like *Java*, *.NET* will *limit* aspects to *pointcuts+advise*.
  - Dynamic languages promise *real innovation* in AOSD.

Language Oriented Programming



Improve *Ruby on Rails*

- What the *cool kids* use for web apps.
- Nice API (effective *DSL's*).
- *Complex* MP code inside.

One example.
What you write:

```
class Customer < ActiveRecord::Base
  has_many BankAccounts
  ...
end
```

What Rails does:

```
module ActiveRecord::Associations::…
def has_many(…)
  reflection =
    create_has_many(…)
    # “alias_person_has_many_bank_accounts”
    name2 = “alias_#{reflection.name}”
    …
```

continued...

```
…
eval <<-EOF
  alias_method #{name2},
               destroy_without_callbacks
  def destroy_without_callbacks
    #{reflection.name}.clear
    #{name2}
  end
EOF
…
```

It's just before *advice!*

Original method

Refactored with Aquarium

```
reflection = …
before :calling =>
      :destroy_without_callbacks do
    eval “#{reflection.name}.clear”
end
```

Aquarium:

- Provides an *intuitive* syntax.
- Supports *runtime addition* and *removal* of advice.
- Advises *Java* through *JRuby*. (sort of…)
- Demonstrates the value of AOP in *dynamically-typed* languages.

3.5 out of 4!

Thank you!

- For more information:
  - http://aquarium.rubyforge.org
  - dean@objectmentor.com
  - http://objectmentor.com
  - http://aspectprogramming.com/papers