Aquarium:
Aspect-Oriented Programming for Ruby

Dean Wampler, Object Mentor
dean@objectmentor.com
April 7, 2008
What is AOP?
Is it necessary for Ruby?
Aquarium in action
class Account

attr_reader :balance

def credit amount
  raise "..." unless amount >= 0
  @balance += amount
end

def debit amount
  raise "..." unless amount < @balance
  @balance -= amount
end
end
Clean and Simple
But, Real Applications Need:

```ruby
class Account
  attr_reader :balance
  def credit amount; ...; end
  def debit  amount; ...; end
end
```

Transactions
Persistence
Security
Scattered Persistence, Transactions, Security, ... Code
Modularity is Compromised.
Rails Solution

class Account < ActiveRecord::Base

... 
end
But what if you want “PORO’s”??

(Plain Old Ruby Objects)
I would like to say...

Before returning the Account balance, read the current balance from the persistence store.

After the Account balance changes, update the new balance in the persistence store.

Before changing the Account balance, authenticate and authorize the user.
require 'aquarium'

class Account
  # reopen Account
  include Aquarium::DSL  # add “DSL” methods

  before :calls_to => [:credit, :debit]  \n    do [join_point, object, *args]  
    object.balance = read_from_database ...
  end

...

The type the aspect acts on is inferred to be Account
...  

after_returning_from :calls_to=>[:credit, :debit] \  
do |join_point, object, *args|  

update_in_database (object.balance,...)  

end  

...
before :calls_to => [:credit, :debit] do |jp, *args|
  raise "..." unless user_authorized
end
end
Can’t we just use Metaprogramming?
Ruby’s metaprogramming gives us the *mechanisms* we need...
..., but it’s nice to implement our *design* concepts using the same idioms.
Some AOP Terms:
Aspect

A modularity construct that incorporates Pointcuts and Advice.

Alternative to before method used before.

```ruby
Aspect.new :before, :calls_to => :credit, \ 
  :in_type => Account do |jp, obj, *args|
  # do something
end
```
Aspect

Can advise individual objects

```ruby
Aspect.new
  :before, :calls_to => :credit, 
  :in_object => my_account
  do |jp, obj, *args|
    # do something
  end
```

Monday, April 7, 2008
Aspect

Aspect.new :before, :calls_to => :credit, \
:in_object => my_account, \
:advice => proc

Use a Proc instead of a block
Join Point

A single execution point.

JoinPoint.new :type => Account,
    :method => :credit

JoinPoint.new :object => account1,
    :method => :credit

:type is one of many synonyms for :in_types
:method is one of many synonyms for :calls_to
Pointcut†

A “query” over all Join Points.

Pointcut.new :types => /.*Account$/,
:calls_to => [:credit, :debit]

Pointcut.new :in_object => account1,
:calls_to => /it$/

†Yes, no space, unlike Join Point
Advice:

- Before
- After returning
- After raising
- After (… returning or raising)
- Around
Before Advice

Do something **before** the Join Point.

```ruby
Aspect.new :before, :pointcut => ... do |jp,o,*a|
  log "Entering: #{jp.inspect}"
end

include Aquarium::DSL
before :pointcut => ... do |jp, object, *args|
  log "Entering: #{jp.inspect}"
end
```
After Returning Advice

Do something **after returning from** the Join Point.

Or :after_returning

Aspect.new :after_returning_from, ... do |jp,o,*a|
  log "Leaving: #{jp}" end

include Aquarium::DSL
after_returning_from :pointcut => ... do |jp,o,*args|
  log "Leaving: #{jp}" end
After Raising Advice

Do something **iff** the Join Point *raises*.

```ruby
Aspect.new :after_raising, ... do |jp, obj, *args|
  log "ERROR: #{jp.context.raised_exception}"
end

include Aquarium::DSL
after_raising :pointcut => ... do |jp, obj, *args|
  log "  ERROR: #{jp.context.raised_exception}"  
end
```
After Advice

Do something after a return or raise...

```ruby
Aspect.new :after, ... do |jp, obj, *args|
  log "Escaped from: #{jp}"
end

include Aquarium::DSL

after :pointcut => ... do |jp, obj, *args|
  log "Escaped from: #{jp}"
end
```
Around Advice

“Wrap” a Join Point.

Aspect.new :around, ... do |jp, obj, *args|
  log “before: #{jp}”
  jp.proceed  # You decide to invoke join point
  log “after: #{jp}”
end

include Aquarium::DSL
around :pointcut => ... do |jp, obj, *args|
  ...
end
A Few Other AOP Terms:
Introduction†

Adding new attributes, methods to a class.
We already have it with Ruby!!

class MyClass
  def do_this; ...; end
end

class MyClass  # reopen MyClass
  def do_that; ...; end
end

†a.k.a Inter-Type Declaration
Cross-Cutting Concerns

When the natural boundaries of different domains *cut across* each-other’s natural boundaries.
Example: Refactor Rails

~175 uses of alias_method in Rails
module ActiveRecord::Associations::ClassMethods
def has_and_belongs_to_many (assoc_id, options => {}, &extension)
  reflection =
  create_has_and_belongs_to_many_reflection (assoc_id, options, &extension)
  ...
  old = "destroy_without_habtm_shim_for_#{reflection.name}"
  class_eval <<-END
    # next slide...
  END
  end
alias_method '#{old}', destroy_without_callbacks

def destroy_without_callbacks
  #{reflection.name}.clear
  #{old}
end
Refactoring with Aquarium

... reflection = create_has_and_belongs_to_many_reflection (assoc_id, options, &extension)

...

before :calls_to => :destroy_without_callbacks \ do |jp, obj, *args|
  class_eval "#{reflection.name}.clear"
end
end
Refactoring Account

Handle “overdraft” requirements as an aspect, so we can vary it independently, possibly per client, per type of account, etc.
class Account
  attr_reader :balance
  def credit amount
    raise "..." unless amount >= 0
    @balance += amount
  end

  def debit amount
    raise "..." unless amount < @balance
    @balance -= amount
  end
end

Move this logic to an aspect
class Account
  attr_reader :balance
  def credit amount
    raise "..." unless amount >= 0
    @balance += amount
  end

  def debit amount
    @balance -= amount
  end
end
class Account  # Reopen class
attr_accessor :max_overdraft
before :calls_to=> :debit, :in_type=> :Account \do |jp, account, *args|
  if (account.balance - args[0]) < -max_overdraft
    raise "..."
  end
end
end
end
Exercises

- aspectprogramming.com/papers/
- Aquarium_RubyAOP_exercises.zip

1. Method tracing
2. Advising method_missing
3. AO design - safer pointcuts
References

- This presentation:
  - aspectprogramming.com/papers
  - aquarium.rubyforge.org
  - aosd.net