

Aspect-Oriented Programming in Academia and Industry

Dean Wampler

Object Mentor

dean@objectmentor.com



- What is AOP?
- History of AOP
- Academia vs. industry
- Future work

```
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:

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



end
end

Transactions

Persistence

Security

Tangled Account Code

Scattered
Persistence,
Transactions,
Security, ...
Code

Modularity
is
Compromised.

We 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::Aspects::DSL::AspectDSL  
  
before :attribute => :balance,  
:attribute_options => [:reader] do |jp, *args|  
  jp.context.advised_object.balance =  
    read_from_database(...)  
end  
  
...
```

aquarium.rubyforge.org

jp: *Join Point*

```
...
after_returning :attribute => :balance,
:attribute_options => [:writer] do |jp, *args|
  update_in_database (
    jp.context.advised_object.balance,...)
end
...

```

```
...
before :methods => [:credit, :debit],
:attributes => [:balance] do |jp, *args|
  raise "..." unless userAuthorized
end
end
```

Can't we just use

Metaprogramming?

(when available)

Languages that support our *paradigms* yield:

- Higher Productivity
- Higher Quality

Refactoring Account

Handle “overdraft” requirements as an aspect

```
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
```

```
class Account
  attr_reader :balance

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

  def debit(amount)
    @balance -= amount
  end
end
```

```
module AllowableOverdraftAccount
attr_accessor :max_overdraft
before :type => :Account,
       :method => :debit do |jp, *args|
  account = jp.context.advised_object
  if (account.balance - args[0]) <
      -max_overdraft
    raise "..."
  end
end
end
```

Some History

A Personal Perspective

"Open Implementation, Analysis and Design of Substrate Software"

OOPSLA '95 Tutorial

G. Kiczales, R. DeLine, A. Lee, C. Maeda

“Black Box” Problems

- Limits of Object-Oriented Modularity
- Need controlled access to internals
 - Often at the “meta-level”

Tutorial Reflected Work On...

- Metaobject protocols (MOPs) and reflection
- MOPs for
 - File system cache management
 - Virtual memory management tuning
 - Process scheduler tuning

At the same time...

The Internet Bubble!!

Industry developers were feeling the pain of cross- cutting concerns (CCC)

- Persistence
- Transactions
- High availability
- Security
- ...

Common Problems Led to AspectJ

AspectJ

- Xerox PARC
- Extension of Java
- Modularizes the *cross-cutting concerns* (CCC)

```
aspect AllowableOverdraftAccount {  
    float Account.maxOverdraft;  
    before (Account account, float amount) :  
        execution (* Account.debit(..)) &&  
        target(account) && args(amount) {  
            if ((account.balance - amount) <  
                - maxOverdraft)  
                throw new OverdraftException(...);  
        }  
    }  
}
```

Why Java?

- Most web/enterprise software is statically typed
 - Where the pain is felt

Why Java?

- Java's "MOP" is insufficient for CCC
 - Rise of byte-code engineering tools
 - Configured with XML!
- But sufficient as a base for AOP tools

An Aside...

- Java's Virtual Machine (and maybe the API's) may become more important than Java itself!

Generative Programming

Czarnecki and Eisenacker

Generative Programming

- Analysis and Design
 - Domain engineering
 - Feature modeling

Generative Programming

- Implementation Technologies
 - Generic programming
 - C++ template metaprogramming
- AOP
- Intentional programming

Multidimensional Separation of Concerns

- IBM Research
- Morphed from “Subject-Oriented Programming”
- Hyper/J
- More ambitious than AspectJ

Multidimensional Separation of Concerns

- Symmetric AOP
 - Aspects as first-class citizens, like classes
- Asymmetric AOP
 - Aspects as “adjuncts”
 - AspectJ’s *de facto* model

Industry Landscape Today

- AOP pervasive in open-source Java enterprise frameworks
 - Spring
 - JBoss

Industry Landscape Today

- Lots of .NET/CLR AOP projects
- Industry adoption still “tepid”

Aspect-Oriented Design

Relearning Object-Oriented Principles

Quantification and Obliviousness

R. Filman and D Friedman (OOPSLA 2000)

AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers.

Open-Closed Principle (Meyer):

- Modules should be
 - *open for extension,*
 - but *closed for modification*

Persistence Aspect

after set (Account.name)



Account
name

Version 1



Account
first_name
last_name

Version 2

Aspects make initial version easier,
but subsequent versions harder!

AOSD-Evolution Paradox

*On the Existence of the
AOSD-Evolution Paradox.*
Tom Tourwé, Johan Brichau,
Kris Gybels.



Next Generation of
Thought...

Non-invasiveness vs. Obliviousness

G. Kiczales, *et al.*

Modules should be aware of possible *advices*, without assuming specifics...

Advice: The new behavior invoked at the join point.

... and modules should
expose *pointcuts*...

... and maybe restrict
access,...

Pointcut: The set of “interesting”
join points.

... but we should still be
able to *advise* modules
without modification.

```
class Account
  attr_reader :balance
  def credit(amount)
    ...
  end
  def debit(amount)
    @balance -= amount
  end
end
```

```
STATE_CHANGE = Pointcut.new
  :methods => [:credit, :debit]
```

aquarium.rubyforge.org

...

```
Aspect.new :pointcut =>
  Account.STATE_CHANGE do | ... |
    # Persist the change...
end
```

We're rediscovering
OO Design Principles

Using Abstractions!

For Completeness...

- Open Modules
- *Modular Reasoning About Advice*
 - J. Aldrich
- Cross-Cutting Programming Interfaces (XPI)
- *Modular Software Design with Crosscutting Interfaces*
 - Griswold, Sullivan, et al.

What Industry Cares About

Industry Criteria for Technology

- Simple (enough) to understand and use
- Strong tool support
- Maintainability of long-lived software
- We must get paid, ASAP!

What Academia Cares About

Academia's Criteria

- Non-trivial, interesting problems
- Theoretical rigor
- Publish or perish!
- But longer time lines are acceptable

Industry and Academia Working Together

Some current and future growth areas for AOP

Language-Oriented Programming

- Raise the abstraction level by constructing Domain Specific Languages (DSLs)
- Could hide the complexity of aspects, objects, metaprogramming, etc.

Contrived Example:

```
...
for_types(with_pointcut(PERSISTABLE))
do |type|
  map_attributes_of(type)
    .excluding.attributes_marked(:transient)
  on_state_changes(:write_to_store)
  use_cache(:memcached)
end
```

What Industry Will Do...

- Invent lots of little, *ad hoc* DSL's
- Create a "Tower of Babel" situation
- Developers will struggle to learn all the DSLs of all the libraries/tools they need

What Academia Could Do...

- You understand language design, AI, etc.
- Help industry understand
 - Globally-applicable DSL design principles
 - Mapping DSLs to object, aspect, ... assembly code

Massively Large Systems

- How would you build a city?
- How would you build a software system of the same complexity?

What Industry Will Do...

- Incremental improvements on what we already know how to do
- Build systems whose complexity exceeds the capabilities of our modularity tools
- Struggle to maintain these systems...

What Academia Could Do...

- Understand the unique characteristics of massive systems
- Find new ways to build them in a modular, manageable way

Some Final Thoughts

- Don't worry too much about industry relevance!
- We need people working on longer-term problems
- Instead of incremental improvements...
- Focus on fundamental problems and innovation!

Thank You!

- dean@aspectprogramming.com
- aquarium.rubyforge.org
- contract4j.org