Understanding Aspects

Mitchell Wand
Northeastern University
August, 2003
Goals for the talk

• Report on my efforts to figure out what AOP is about
• Suggest some ways in which PL research can be applied to AOP
Outline

1. Background: what problems was AOP intended to address?
2. Examples
3. Shortcomings of current efforts
4. Reconceptualizing AOP
5. Implications for future research
The problem

- Limitations of traditional layered architectures
- Different research groups tell different motivating stories:
  - Tyranny of the primary decomposition
  - Crosscutting concerns lead to scattered and tangled code
Tyranny of the primary decomposition

- Want to assemble programs from different subsystems
- Each subsystem has its own idea of how program should be organized and who’s in control
- Multiple views of program lead to combinatorial explosion of methods
- Want effect of multiple inheritance
Example systems

- HyperJ [Ossher-Tarr et al]
- Jiauzzi [Flatt et al]
- Mixin Layers, GenVoca [Batory et al]
- Composition Filters [Aksit et al]
Crosscutting concerns lead to complexity

- Applications typically need multiple services:
  - logging, locking, display, transport, authentication, security, etc
- These services don't naturally fit in usual module boundaries ("crosscutting")
Scattering and tangling

• These services must be called from many places ("scattering")
• An individual operation may need to refer to many services ("tangling")
Example of scattering

logging in org.apache.tomcat
- each bar shows one module
- red shows lines of code that handle logging
- not in just one place, not even in a small number of places
So what's the problem?

- Functional programmers know the answer: use proxies or wrappers

```
(define doit
  (let ((old-doit doit))
    (logged-version old-doit)))
```
Why isn’t that enough?

• How to make sure an application calls the right proxy?
• Potential for conflict with calls to multiple services
  - combinatorial explosion of wrappers
  - tangling
The real problem

- Each application has a **policy** about when each service is required
- But the policy is built into the structure of the program
- Hard to understand, modify, etc, etc
A solution

- Add a **policy language** that describes where each service needs to be called
  - policy language is declarative
  - localizes knowledge about policy
Examples

• D [Lopes-Kiczales 97]
  - had policy languages for several kinds of services
    • locking/mutual exclusion (COOL)
    • transport (RIDL)
    • proposals for others
  - Each such service became an “aspect”

• QuO [BBN 00]
  - policy language for network transport
COORDINATOR BoundedBuffer {
    selfex put, take;
    mutex {put, take};
    condition empty = true, full = false;

    put: requires !full;
    on_exit {
        if (empty) empty = false;
        if (usedSlots == capacity) full = true;
    }
    take: requires !empty; ...
}
QuO Example

```quodl
contract UAVdistrib {
    sysconds ValueType actualFrameRate, ... ;
callbacks sourceControl, timeInRegion ;
region HighLoad (actualFrameRate >= 8) {
    state Test until (timeInRegion >= 3) {
        transition any->Test {
            sourceControl.setFrameRate(30);
            timeInRegion.longValue(0);
        }
    }
    ...
}
```
Limitations of this approach

• What are aspects, anyway?
• Is there some fixed finite set of aspects?
  – Might even want to express some functional behavior as aspects
• Need to analyze each aspect, then develop and maintain a language for it
• Proliferation of languages for individual aspects
• Bad for technology transfer
AspectJ

[Kiczales et al 01]

- Kiczales’ strategy: develop a single language in which all aspects could be expressed
- Use Java as base language
- Allow the community to concentrate its efforts on a single tool
Ideas of AspectJ

• Policy specified in terms of join points at which actions could be attached
• Join points: events in the program execution:
  - method calls
  - method executions
  - constructor calls
  - etc
AspectJ, cont’d

- Policies expressed as sets of join points, called **point cuts**
- Language of **point cut descriptors** allows declarative specification of point cuts
- Action at a point cut expressed as **advice** before/after/around each join point in the point cut
Example

[AspectJ manual]

aspect LogPublicErrors {
  pointcut publicInterface():
    instanceof(mypackage.*) &&
    executions(public * *(..));

  static after() throwing(Error e): publicInterface()
  {logIt(e); throw e;}

  static void logIt(Error e) { ... }

  each aspect packages a policy

  pointcut declaration

  advice on this pointcut
What's the difficulty?

• AspectJ point cuts are a powerful reflection mechanism
• Can use them to detect and modify otherwise unobservable program behavior
• Ordinary local reasoning about programs becomes invalid
Meddling aspects

```java
class C {
    static int foo;
    static final void m1() {foo = 55;}
    static final void m2()
        {m1(); println(foo);}
}

aspect Meddle {
    void after() :
        void call(C.m1())
        {target.foo = 66}
    }
```

Does m2 always print 55?

Ouch! My aching invariant!
Aspects can detect refactoring

```java
class C {void foo () {..} ..}
class D extends C {}

class C {void foo () {..} ..}
class D extends C {
    void foo () {super.foo();}
}

aspect Distinguish {
    void around():
        execution (void D.foo())
        {println("gotcha!");}
}
```

returns w/o calling super
Aspects can distinguish non-terminating programs

```
class C { static final void foo() { foo(); } }  #1
    static final void bar() { bar(); }

aspect Distinguish {
    void around():
        executions(void C.bar())
        { println("gotcha!"); }
}
```

makes `c.foo()` halt in #2, not in #1
Why is this so bad?

- Can no longer do local reasoning about programs; can only do whole-program reasoning
- Defeats encapsulation, which is basic SWE principle
- Tools such as aspect browsers can help, but scalability is a question mark
Where did we go astray?

• Previous AO Languages were *conjunctive* specifications
• Can think of each aspect as a partial specification of behavior of the machine
• *conjunctive* = orthogonal
What AspectJ changed

- But AspectJ is an *exception* specification!
- “Base program” is intended to be a complete specification of a JVM behavior
- Advice *changes* the behavior
- Now reasoning is much more difficult
- Level much too low-- no room for partial specification
Reconceptualizing AOP

- Scattering is inevitable
- Aspects are modular units of specification
- A join point model is a shared ontology
The conspiracy theory of programming

• A specification represents a *conspiracy* between two or more pieces of program.

• \((\text{pop}\ (\text{push} \times s)) = s\) specifies a conspiracy between \text{push} and \text{pop}.

• \text{push} and \text{pop} must agree on the representation of stacks.
Good conspiracies are local

• If we change the representation of stacks, we need only change push and pop to match; client need not change
• This is good if push and pop are in the same module
Distributed conspiracies are harder

- A policy is a cross-module specification
- Changes to representation or to specification require changes in many modules
Example

- Policy: "logging should occur whenever events in set X happen"
- If you change X, you may have to change all the modules in which X may happen
- This is root cause of scattering
- Conclusion: scattering is inevitable!
How to escape

• Don’t think about programming, think about specification
• An aspect is a modular unit of specification
Examples

- Standard examples:
  - Base functionality, logging, security, persistence, etc
- Each of these is best specified in its own language
- Policy language must intersect all of these languages
  - intersections are join points
- So it must know something about each of them. Therefore:
A join point model is a shared ontology

- A join point model is a shared ontology, representing the aspects’ shared understanding of their joint specificand
- The join points are a class of entities in that shared ontology
What is an ontology?

• Specifies a domain of discourse
• Specifies the structure of entities in that domain
• May specify additional constraints
• Can have different ontologies for the same entities
  - different data represented
  - different constraints
• Languages for writing ontologies
  - UML/OCL, RDF, DAML/OIL
Ontologies as Agreements

- Agents agree on names for things and their behaviors
- Each agent may bring additional knowledge about the domain, not in the shared portion of the ontology
Example: lexer/parser communication

- **Agents:**
  - Lexers and parsers
- **Domain of discourse:**
  - lexical items
- **Ontology:**
  - each item has a lexical class, data, and debugging info
- **Join points:**
  - names of lexical classes
  - Lexer and parser must agree on these names
Example: ADT’s

- **Agents:**
  - server (ADT implementation) and its clients

- **Domain of discourse:** procedure calls

- **Ontology:**
  - includes agreement on the semantics of a procedure call

- **Join points:**
  - names of procedures in interface
  - Client and server agree on the names of procedures to be called, and on their behavior
Procedures vs. methods

- In Java, can do the same thing, but domain of discourse is method calls instead of procedure calls.
- A procedure-oriented client can’t use an object-oriented server!
Widening our horizons

• With this new perspective, we can look for hidden aspect-orientation in other languages

• So: what is the world’s most popular aspect-oriented language?
Microsoft Word!
Microsoft Word

- Different aspects:
  - Contents aspect
  - Formatting aspect, with subaspects:
    - Font
    - Indentation/Margin
    - Borders/Shading, etc

- Structure of menus mimics the structure of this ontology
Word example, cont’d

- Not a programming language
- But has some weak abstraction capabilities: styles
- Also has a weak policy language, e.g.: "whenever you reach the end of a paragraph in Style1, start the next paragraph in Style2."
Aspect-oriented programming reconsidered

• Let's see how some AOP languages fit into this framework
AspectJ

• Domain of discourse:
  - execution sequences of an idealized JVM

• Ontology:
  - an execution consists of a sequence of object constructions, method calls, method executions, etc. Each such event takes place in a dynamic context (the \textit{cf}low)

• Actions:
  - execute advice before/after/around each event in the ontology
Composition Filters

[Aksit et al 92]

- Domain of discourse:
  - OOPL execution sequences
  - like AspectJ
- Ontology:
  - method calls
- Action:
  - interpose filters

Same domain of discourse, different ontology
Composition Filters, cont'd

- filter runs an incoming message through a decision tree,
  - based on pattern-matching and boolean "condition variables" set from code
  - so filter can have state
- filter can then dispatch the message to different methods, reify, queue messages, etc
- Does this raise the same difficulties as advice? Good question!
Hyper/J

[Ossher-Tarr 99]

• Domain of discourse:
  - Java program texts

• Ontology:
  - a Java program is a set of packages, each of which consists of a set of classes, each of which consists of a set of methods

• Actions:
  - collect methods into classes
  - associate a method body with each method name
DemeterJ

[Lieberherr 96 et seq]

- Domain of discourse:
  - Graph traversals
- Ontology:
  - A graph traversal is a sequence of node or edge visits
- Action:
  - Call a visitor method at each event
PL research in AOP

- **Descriptive:**
  - [de Meuter 97], [Andrews 01], [Douence-Motelet-Sudholt 01], [Lammel 02], [Wand-Kiczales-Dutchyn 02]

- **Compiler Correctness**
  - [Masuhara-Kiczales-Dutchyn 02], [Jagadeesan-Jeffrey-Riely 03]

- **Core Calculi**
  - [Walker-Zdancewic-Ligatti 03]
Research Directions

• Some ways in which the PL community can make AOP safer
Higher-level join-point models

• AspectJ ontology is that of OO assembly language
  - universal, but too low-level
• Better idea: make the join-point model part of the system design
  - UML represents a system-wide shared ontology of data
  - can we do the same thing for join points?
  - example: Emacs-Lisp hook system
  - example: [Heeren-Hage-Swiertsa 03]
Domain-specific aspect languages

- Each aspect is best specified in its own vocabulary
- First-generation AO languages had it right
  - But development, deployment costs too high
- We can do better:
  - build tools and environments to support DSAL’s
- This is the real long-term win for AO ideas
Example: Scripting Type Inference

- Join point model
  - inference steps in the typechecker
  - inferences contain unifications (= jp's)
- Language for describing them
- Language for advising them
  - action: on failure print <whatever>
- Soundness is guaranteed
  - can't cheat

[Heeren-Hage-Swiertsa 03]
Aspect-oriented reasoning

- Goal: restore possibility of local reasoning
- We reason locally about program fragments by making assumptions about the class of contexts in which they will be executed
  - type assumptions: consider only well-typed contexts
  - evaluation assumptions: we don't consider contexts like `println('[ ]')`

PlI 2003
Specifying contexts

- Can we formalize our assumptions about contexts with aspects? e.g.:
  - which join points are visible to the context
  - what portion of the state the advice is allowed to change
- With such contextual assumptions, we could restore the possibility of local reasoning
Conclusions

• AOP is getting a lot of attention in the SWE world
• Current popular AOP mechanisms (eg global advice) seem flawed
  - too low-level, can't do local program reasoning
• We ought to be able to do better
  - more semantics in the join point model
  - more semantics in the aspect languages
  - more semantics in the contextual assumptions
The End

Slides available soon at http://www.ccs.neu.edu/home/wand