

Aspects of AOP and Related Tools

Venkat Subramaniam

venkats@agiledeveloper.com

HJUG - August 2003

Presentation and examples can be downloaded from
<http://www.agiledeveloper.com/download.aspx>

Abstract

Abstract OOP is currently the most popular and practical software development approach. However, OOP has its limitations, especially when it comes to separation of concerns that are global and crosscutting in a large application. Aspect Oriented Programming addresses this issue of managing the complexity and AspectJ is an extension to Java to realize AOP. In this presentation, we will introduce AOP and show how you can implement the concepts using AspectJ and its Eclipse plugin. Audience with strong OO programming knowledge will benefit the most from this presentation.

Speaker Dr. Venkat Subramaniam, founder of Agile Developer, Inc., has taught and mentored more than 2,500 software developers around the world. He has significant experience in architecture, design, and development of distributed object systems. Venkat is an adjunct professor at the University of Houston and teaches the Professional Software Developer Series at Rice University's Technology Education Center. He may be reached at venkats@agiledeveloper.com.



Examples Any page with a  has an example attached
Download from <http://www.agiledeveloper.com/download.aspx>

Aspects of AOP and Related Tools

- **Limitation of OO**
- Separation of Concerns
- Aspect Oriented programming
- AspectJ
- PointCut
- Advice
- Pitfalls
- Conclusion

Software Development

- Various methodologies have evolved
- Object-Oriented Paradigm is the most popular and practical in effect currently
- System composed of objects/ entities
- Used in all kinds of application development

Reasons to use OO

- Helps us manage complexity
- If done well, easier to make change
- Component based approach to developing systems
- But, what are the limitations of OO?

Limitations of OO

- OO advocates decomposing a system into entities
- As complexity increases, the limitations surface
- Breaking system into objects helps manage complexity
- However, can all the system concerns be decomposed into an object?
 - Not really
 - Move commonality into a base class?
 - How about spreading them across several objects?
 - Makes it harder to keep up with the change

Aspects of AOP and Related Tools

- Limitation of OO
- **Separation of Concerns**
- Aspect Oriented programming
- AspectJ
- PointCut
- Advice
- Pitfalls
- Conclusion

Separation of Concerns

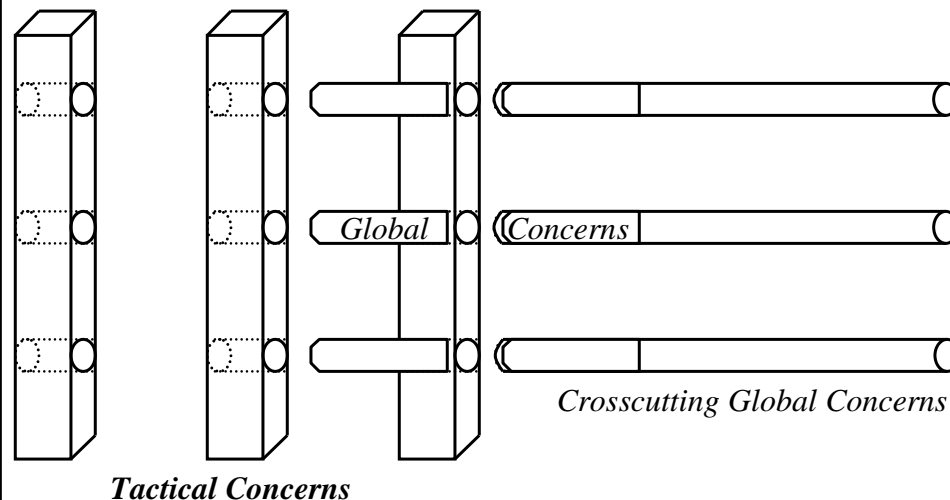
- We have heard this in OOP
- We want to separate the concerns in our system into manageable pieces
- OO does this to a certain extent
- But what about concerns at global level
- Concerns like security, transaction, tracing, logging, error handling, etc.



Crosscutting Concerns

- Some concerns are fairly localized within entities
- Other concerns cut across multiple elements in the system
- How about keeping these cross cutting concerns separately and weaving them horizontally into the system?

Weaving the system



Advantages

- Could we not write these as functions & call?
 - Results in code permeating through the system at various places – hard to maintain
 - Harder to express concerns this way
 - intrusive – you modify your code to invoke these concerns
 - requires understanding at each level
- In this approach
 - You can focus on the concerns at one place
 - Easier to add and remove concerns
 - Easier to modify or fine tune concerns
 - Easier to understand
 - Efficient to implement
 - More efficient

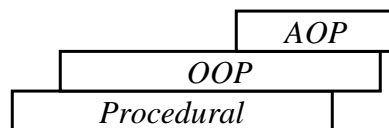
Aspects of AOP and Related Tools

- **Limitation of OO**
- Separation of Concerns
- **Aspect Oriented programming**
- AspectJ
- PointCut
- Advice
- Pitfalls
- Conclusion

What is an Aspect?

- Aspects are
 - Collection of crosscutting concerns in a system
 - the crosscutting implementations
 - These are generally present among several layers or levels of class hierarchy in a OO system
 - Concerns that are orthogonal to the system

AOP vs. OOP



- AOP does not replace OOP
- It handles separation of concerns better than OOP
- Much like how OOP still uses concepts that are procedural, AOP uses concepts that are OOP
- It extends OOP
- AOP has
 - Functions, Classes and Aspects

Goals of AOP

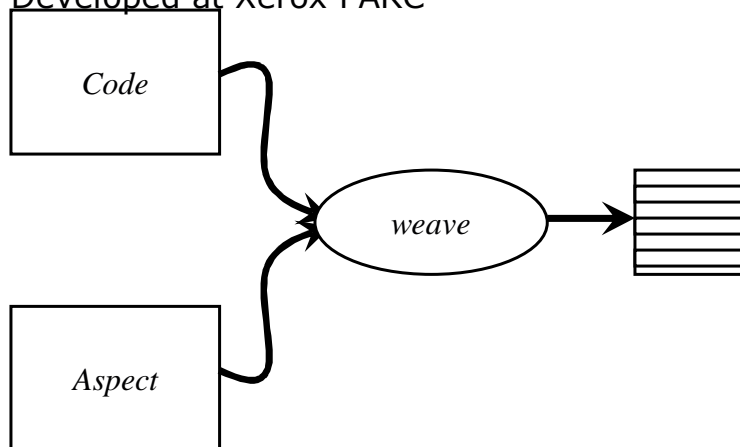
- To
 - separate expression of behavioral concerns from structural ones
 - make design and code more modular
 - not scatter the concerns though out your code
 - isolate the concerns for separate development and
 - be able to plug and unplug these concerns at will

Aspects of AOP and Related Tools

- Limitation of OO
- Separation of Concerns
- Aspect Oriented programming
- **AspectJ**
- PointCut
- Advice
- Pitfalls
- Conclusion

What does AspectJ do?

- General purpose aspect oriented extension to Java
 - Developed at Xerox PARC

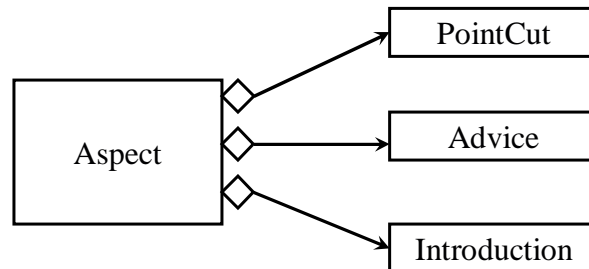


AspectJ Concepts & Constructs

- Join Point
 - well defined points in the execution flow of the code
 - method calls
 - constructor invocation
 - field access
- PointCut
 - selects certain join points and values at those points
- Advice
 - defines code that is executed when a pointcut is reached
- Introduction
 - modifies static structure –classes relationship

Aspect in AspectJ

- Module of crosscutting concerns



```
public aspect MenuEnabling {
    pointcut CreationOfMenuItem() : call(JMenuItem.new(..));

    after() returning(JMenuItem item) : CreationOfMenuItem() {
        // advice definition code goes here
    }

    after() returning(JMenuItem item) : CreationOfMenuItem() {...}
}
```

Aspects of AOP and Related Tools

- Limitation of OO
- Separation of Concerns
- Aspect Oriented programming
- AspectJ
- **PointCut**
- Advice
- Pitfalls
- Conclusion

Pointcuts

- Defines arbitrary number of points in a program
- However, defines finite number of kinds of points
 - method invocation
 - method execution
 - exception handling
 - object instantiation
 - constructor execution
 - field reference

PointCut Designators

- execution
 - execution(void X.foo()) – when X.foo's body executes
- call
 - call(void X.foo()) – when method X.foo is called
- handler
 - handler(OutOfMemoryException) – execution of the exception handler
- this
 - this(X) – object currently executing is of type X
- within
 - within(X) – executing code belongs to class X
- target
 - target(X) – target object is of type X
- cflow
 - cflow(void X.foo()) - This special pointcut defines all joint points between receiving method calls for the method and returning from those calls, i.e., points in the control flow of the call to X.foo()

PointCut Examples


- name-based crosscutting
 - call (void MyClass.foo(int))
 - any call to foo(int) on any object of MyClass
 - call (void MyClass1.f1(int)) ||
call (void MyClass2.f2(double))
 - any call to either f1 on object of MyClass1 or f2 on object of MyClass2
 - pointcut pc1() : call (void MyClass.foo(int))
 - named pointcut with name pc1
- property-based crosscutting (not exact name)
 - call (void MyClass.f*(..)) || call (*
MyClass2.*(..))
 - void methods of MyClass starting with f or any method of MyClass2

PointCut Examples...

- pointcut pc3(X ref) : target(ref) &&
call(public * *(..))
 - calls to any methods, on an object of X, with any args
- I want to find which methods of my class are invoked during a certain execution of my program



Quiz Time

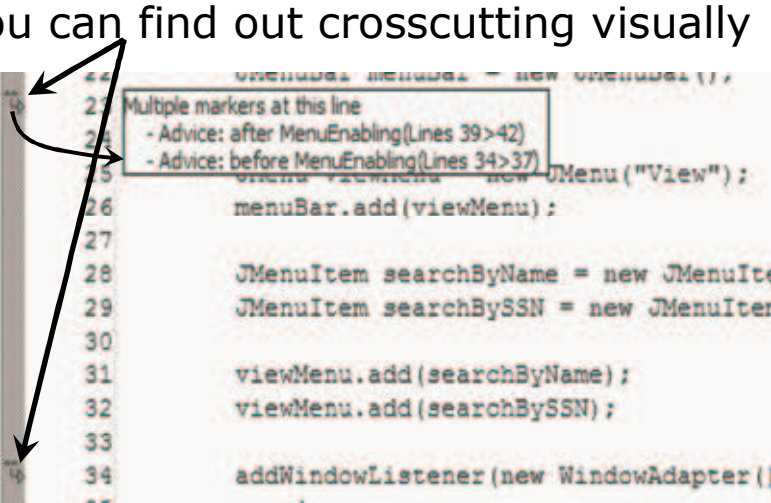


Agile Developer

Aspects of AOP- 25

Eclipse Plugin Support

- You can find out crosscutting visually



The screenshot shows a Java code editor with a method `createMenuBar()`. A red box highlights a call to `viewMenu.add()` on line 31. A tooltip is displayed over this call, listing two advice points: `- Advice: after MenuEnabling(Lines 39>42)` and `- Advice: before MenuEnabling(Lines 34>37)`. A vertical bar on the left side of the editor indicates the scope of the advice, spanning from line 34 to line 42. Arrows point from the text 'You can find out crosscutting visually' to the tooltip and the vertical bar.

```
24 createMenuBar() {
25     Multiple markers at this line
26     - Advice: after MenuEnabling(Lines 39>42)
27     - Advice: before MenuEnabling(Lines 34>37)
28     JMenu viewMenu = new JMenu("View");
29     menuBar.add(viewMenu);
30
31     JMenuItem searchByName = new JMenuItem
32     JMenuItem searchBySSN = new JMenuItem(
33
34     viewMenu.add(searchByName);
35     viewMenu.add(searchBySSN);
36
37     addWindowListener(new WindowAdapter()
38 }
39 }
```

Agile Developer

Aspects of AOP- 26

-
- ```

24 JMenuItem menuBar = new JMenuItem();
25 Multiple markers at this line
26 - Advice: after MenuEnabling(Lines 39>42)
27 - Advice: before MenuEnabling(Lines 34>37)
28 JMenuItem searchByName = new JMenuItem();
29 JMenuItem searchBySSN = new JMenuItem();
30
31 viewMenu.add(searchByName);
32 viewMenu.add(searchBySSN);
33
34 addWindowListener(new WindowAdapter()

```

## call vs. execution

- In the case of a call, the context is in the caller of the method
- In the case of execution, the context is within the method of interest
- call will not capture super calls to non-static methods of the base, execution will
- Use call if you want an advice to run when the call is made. Use execution if you want an advice to run when ever a code is executed



## Pointcut Context

- Execution context at the join point
- advice declarations may use these values
- ```
pointcut pc2(MyClass obj, int a) :  
    call (void MyClass.foo(int)) && target(obj)  
    && args(a);
```
- ```
after(MyClass obj, int a) : pc2(obj, a)
{
 System.out.println("method foo called on "
+ obj +
 " with arg " + a);
}
```

## Aspects of AOP and Related Tools

- Limitation of OO
- Separation of Concerns
- Aspect Oriented programming
- AspectJ
- PointCut
- **Advice**
- Pitfalls
- Conclusion

## Advice

- Defines code that should run at join points
- Types of Advices:
  - Before
    - runs when joint point is reached, but before computation proceeds
  - After
    - runs after computation finishes and before the control returns to the caller
  - Around
    - controls if the computation under joint point is allowed to run
- Example
  - before() : pc1()
    - {
    - the code to run
    - }

## Advice and call execution

- `after()` : `call(int X.foo(int) {...})`
  - executes after the call to `X.foo(int)`, irrespective of successful completion or not
- `after() returning(int result)` : `call(int X.foo(int){...})`
  - executes after the successful completion of the call. The returned result may be accessed by advice definition
- `after() throwing(Exception e)` : `call(int X.foo(int))`
  - executes only if `foo` throws exception of type `Exception`. After the advice runs, the exception is re-thrown.



## Bypassing calls

- Using `around` you may bypass calls to methods
- You may check for conditions and let the call go through or simply refuse to allow the call as well
- `around(X ref, int a)` : `call(int X.foo(int) && args(a) && target(ref)`  
    {  
        if (`a > 2`) `proceed(ref, a)`;  
        return 4;  
    }



## Aspects of AOP and Related Tools

- Limitation of OO
- Separation of Concerns
- Aspect Oriented programming
- AspectJ
- PointCut
- Advice
- **Pitfalls**
- Conclusion

## Pitfalls

- While concept is very simple, syntax is confusing
- Has some learning curve, especially to implement some complex cross cuttings
- Easy to write a pointcut that puts your code in recursive calls –  
StackOverflowException
- Different tools for different languages

## Quiz Time



## Aspects of AOP and Related Tools

- Limitation of OO
- Separation of Concerns
- Aspect Oriented programming
- AspectJ
- PointCut
- Advice
- Pitfalls
- **Conclusion**

## Conclusion

- AOP seems to be the next logical step in handling
  - complexity
  - separation of concerns
- Has a lot of promise
- This is a beginning and not the end to the next phase of refinement

## Further Reading

1. Aspect-Oriented Software Development:  
<http://www.aosd.net>
2. Aspect J: <http://www.eclipse.org/aspectj/>
3. Aspect J Development Tools (Eclipse Plugin):  
<http://www.eclipse.org/ajdt/>
4. AOP Focus issue: Communications of the ACM,  
October 2001- Volume 44, Number 10.
5. Examples, slides are for your download at  
**<http://www.agiledeveloper.com/download.aspx>**