Aspect Oriented Programming in Java

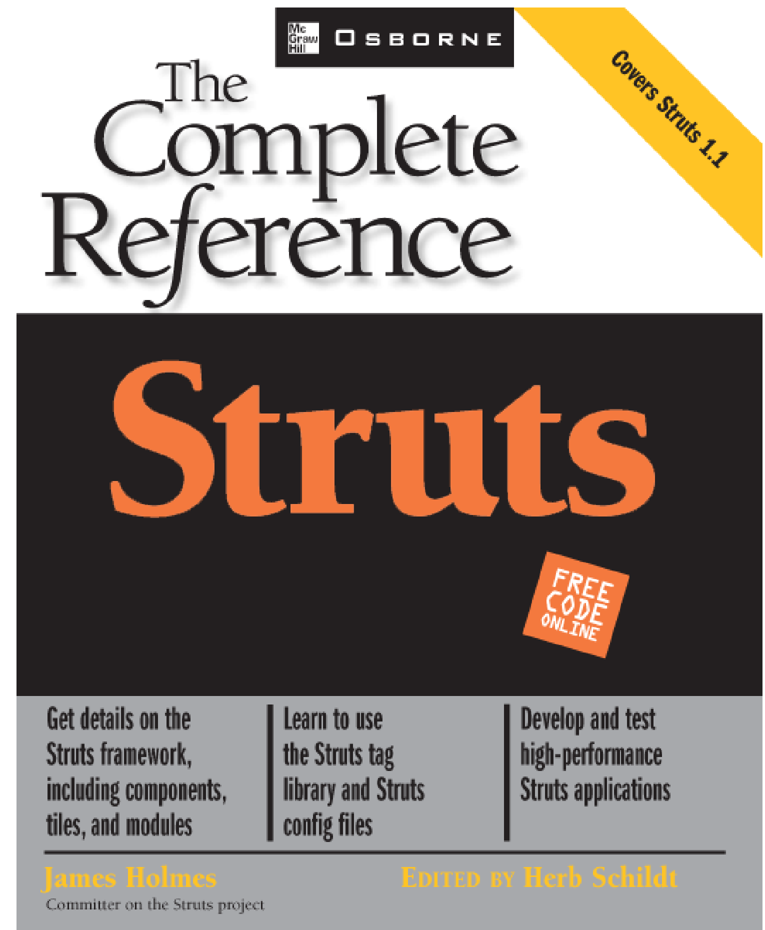
Atlanta Java Users Group

July 15, 2003

James Holmes

Speaker Introduction

- Independent Java Consultant
- Struts project committer
- Creator of popular Struts Console tool
- Author of upcoming Struts: The Complete Reference from McGraw-Hill Osborne
- Co-Author of Art of Java from McGraw-Hill Osborne
- 2002 Oracle Magazine Java Developer of the Year



- Programming background / need for AOP
  - What’s wrong / can be improved with current methodologies
- Overview of Aspect Oriented Programming
- Applying AOP with Java

Programming Evolution

- Procedural programming
  - C, Pascal, Fortran
- Object Oriented programming
  - C++, Java
- Aspect Oriented programming
  - AspectJ (several other frameworks)

    possibly a new language??

Benefits of OOP

- OOP builds on existing paradigms
- Modularity of code
- Code reuse/extensibility
  - Inheritance
  - Polymorphism
  - Design patterns
  - Frameworks

Problems with OOP

- OOP does not allow strong implementation reuse
  - Each class must either inherit **one** base class or use delegation (which breaks instanceof tests)
- Certain software properties cannot be isolated in a single functional unit, instead they *crosscut* multiple components
  - Logging, security checks, transaction management, etc. are all over the place
- Such crosscutting concerns result in *tangled code* that is hard to develop and maintain

Cost of Tangled Code

- Redundant Code
  - Same fragment of code in many places
- Difficult to understand
  - Non-explicit structure
  - Big picture of the tangling is unclear
- Difficult to change
  - Have to find all the code involved
  - and be sure to change it consistently
  - and be sure not to break it by accident

Introducing AOP

- AOP introduced to solve crosscutting problems of OOP
- AOP is to OOP what OOP was to procedural languages (think C++ to C)
- Originated in academic/research community in 1997
- Primarily researched and developed by Xerox PARC

The Idea Behind AOP

- Crosscutting is inherent in complex systems
- Crosscutting concerns
  - Have a clear purpose
  - Have a natural structure
    - Defined set of methods, module boundary crossings, points of resource utilization, lines of data flow
- AOP captures the structure of crosscutting concerns explicitly
  - In a modular way
  - With linguistic and tool support
- Aspects are well modularized crosscutting concerns

AOP Benefits

- All benefits of OOP discussed earlier
- Separation of components and aspects
  - Better understanding of software because of high level of abstraction
  - Easier development and maintenance because tangling of code is prevented
  - Increased potential for reuse for both components as well as aspects

AOP Benefits

- Modularized implementation of crosscutting concerns
  AOP addresses each concern separately with minimal coupling, resulting in modularized implementations

- Easier-to-evolve systems
  Aspected modules can be unaware of crosscutting concerns, that way it's easy to add newer functionality by creating new aspects

- Late binding of design decisions
  Architects can delay making design decisions for future requirements since they can be implemented as aspects

- More code reuse
  Because AOP implements each aspect as a separate module, each individual module is more loosely coupled

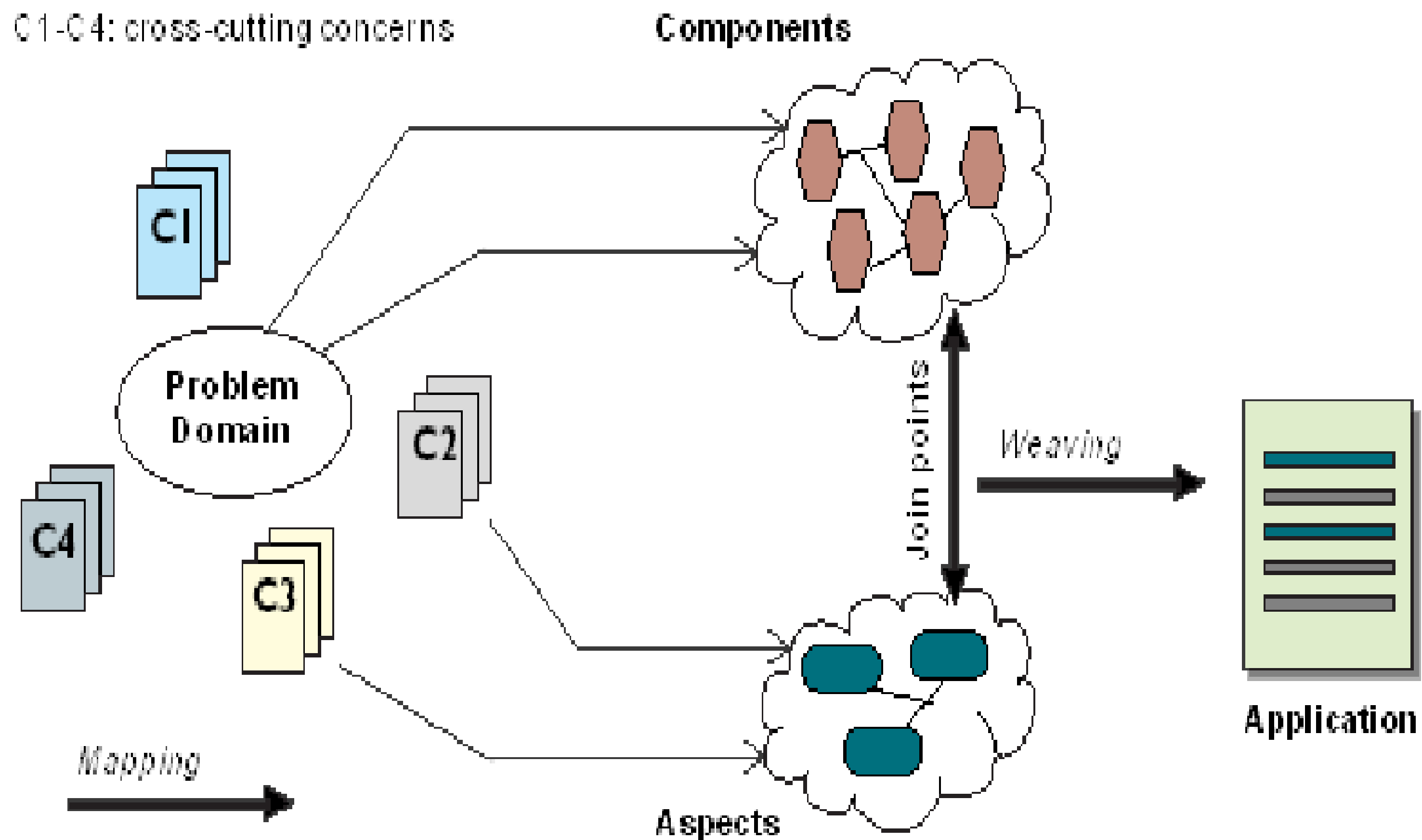
Software Properties

- Components: properties that can be cleanly encapsulated in a unit of function or behavior; like a procedure or object

- Aspects: cross-cutting concerns that cannot be captured in a functional decomposition; properties affecting the performance or semantics of components

- Components
  - GUI elements
  - Business logic
  - Database access
- Aspects
  - Logging
  - Caching
  - Debugging

How AOP works



AOP Mechanisms

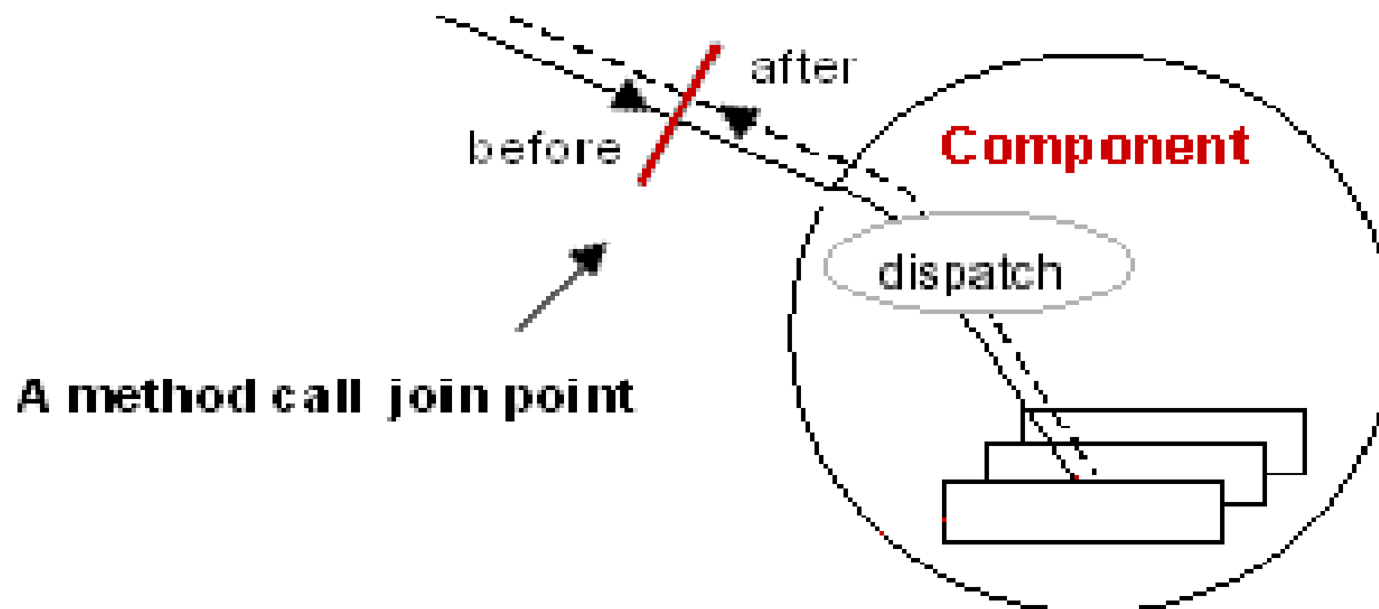
- Abstraction Mechanism:
  An *aspect description language* is used to encapsulate crosscutting concerns into modules according to the *join point model*
- Composition Mechanism:
  A *weaver* is used to merge the aspects and components into an application that only contains traditional language constructs

Join Point Model

- Identify join points
  - Points in the execution of components where aspects should be applied
  - i.e. method invocation or object construction
- Describe behavior at join points
  - Wrap join points with extra code just before or just after execution
  - i.e. log before and after method or lock and unlock shared data

Join Point Illustration

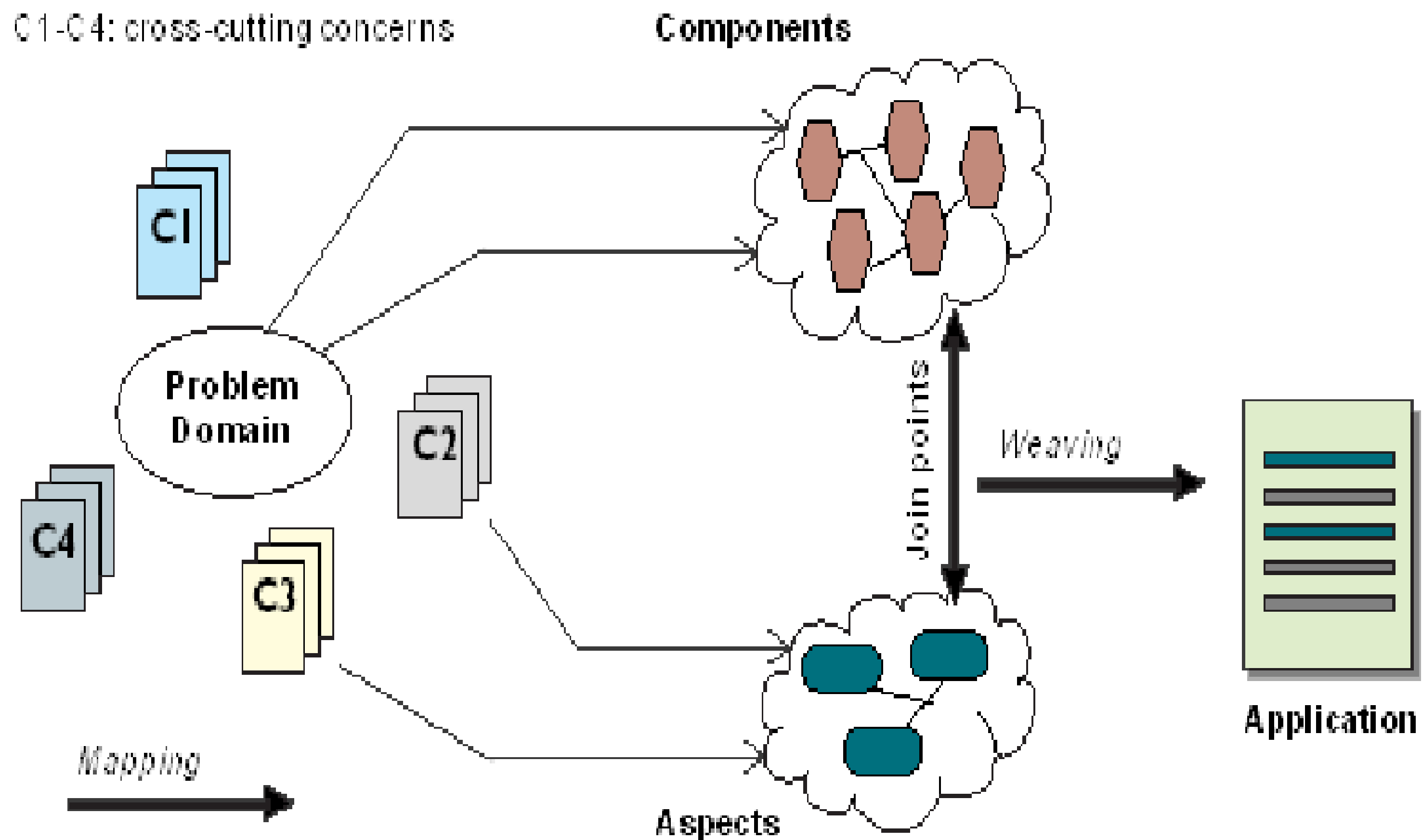
- A method call join point:



Weaving

- Aspect description languages cannot be processed by traditional compilers
- Therefore, aspects and components are woven into intermediate source code
- Weaving is no longer needed if there is an aspect-oriented compiler available
- This can be compared to preprocessing of C++ code to generate a C representation

How AOP works



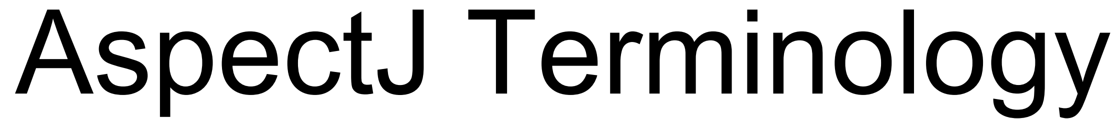
Applying AOP with Java

- First: AOP can be applied to a number of different languages; it’s not just for Java
- Several AOP frameworks to use with Java
  - AspectJ
    (most popular, several books and articles)
  - JBossAOP
    (heavily integrated into JBoss)
  - AspectWerkz

AspectJ

- Initially developed by Xerox PARC with DARPA funding
- Most popular Java AOP framework
- Well Documented
  - 3 – 5 books out or in progress on AspectJ
  - Several articles published on AspectJ
- GUI support
  - Eclipse, NetBeans, JBuilder, IDEA, Emacs and so on

AspectJ Terminology



- Pointcuts:
  used to define join points
- Advice:
  used to bind pointcuts to code
- Inter-type Declarations:
  used to statically add code to classes
- Aspects:
  used to wrap Pointcuts, Advice and Inter-type Declarations into modular units

Pointcuts

Used to define join points for an aspect

Basic Example

```
pointcut log():
  call(void BizObj.meth1());
```

Expanded Basic Example

```
pointcut log():
  call(void BizObj.meth1()) ||
  call(void BizObj.meth2(int));
```

Pointcuts

Wildcard Examples

```
pointcut log():
  call(void BizObj.meth*(..));
pointcut log():
  call(public * BizObj.meth*(..));
```

CFlow Examples

```
pointcut log2():
  cflow(call(void BizObj.meth1()));
pointcut log2():
  cflow(log());
```

Pointcuts

- We've covered most basic pointcut functionality, however, there are infinite more possibilities that are more complex
  - poincut parameters
  - execution() vs. call()
    different behavior for within(), etc.
  - target(), within(), this()
    used for narrowing pointcut to specific types

Advice

Used to bind pointcuts to code for an aspect

Before Example

```
before(): log() {
   System.out.println(“logging before method call");
}
```

After Example

```
after(): log() {
   System.out.println(“logging after returning/throwing”);
}
after() returning: log() {
   System.out.println(“logging after returning”);
}
after() throwing: log() {
   System.out.println(“logging after throwing”);
}
```

Advice

Around Example

```
around(): log() {
  System.out.println("logging 'around'
  method call")
  proceed();
}
```

Traps the execution of the specified join point

Proceed() returns processing to joinpoint's code

Inter-type Declarations

Used to statically (compile time) add/introduce code to classes

Field Introduction Example

```
aspect ObserverAspect {
  private Vector BizObj.observers
  = new Vector();
}
```

Inter-type Declarations

Method Introduction Example

```
aspect ObserverAspect {
  private Vector BizObj.observers = new
  Vector();

  public void addObserver(Observer o) {
     observers.add(o);
  }

  public void removeObserver(Observer o) {
     observers.remove(o);
  }
}
```

Add Parents Example

```
declare parents:
  BizObj implements Comparable;

declare parents:
  BizObj extends BaseBizObj;
```

Aspects

- Used to wrap Pointcuts, Advice and Inter-type Declarations into modular units
- Essentially equivalent to Java's class construct
- Can have fields and methods, just like Java classes, in addition to crosscutting code/semantics
- Each aspect is a singleton

Aspect Example

```
aspect Logger {
  pointcut log(): call(public * BizObj.*(..));

  before(): log() {
      System.out.println("before method call");
  }

  after(): log() {
      System.out.println("after method call");
  }
}
```

Example 1: Logging Aspect

Business Object Component Source

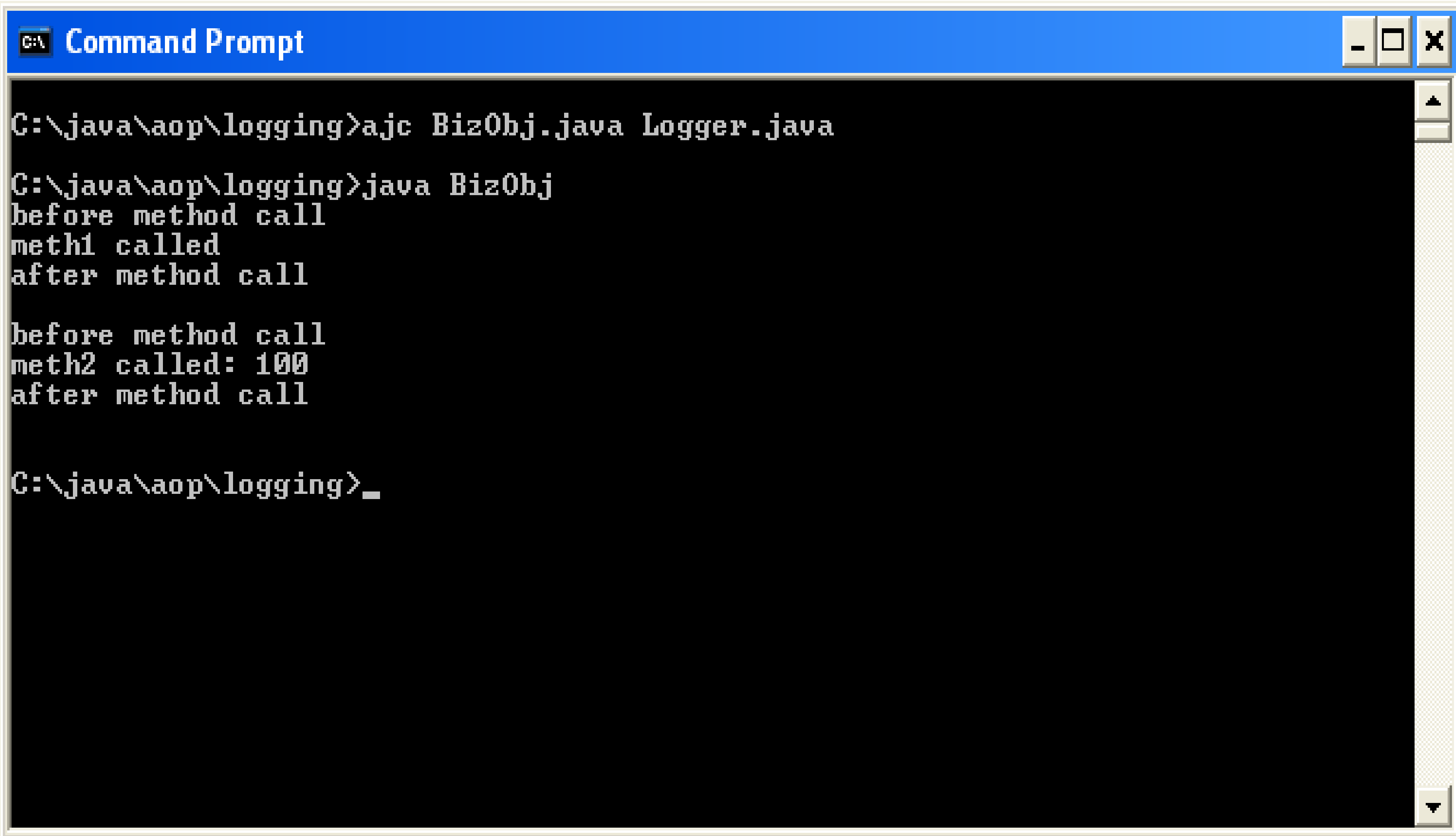
```
public class BizObj {
  public void meth1() {
      System.out.println("meth1 called");
  }

  public void meth2(int value) {
      System.out.println("meth2 called: " + value);
  }

  public static void main(String[] args) {
      BizObj obj = new BizObj();
      obj.meth1();
      obj.meth2(100);
  }
}
```

Example 1: Logging Aspect

Logging Aspect Source

```
aspect Logger {
  pointcut log(): call(public * BizObj.*(..));

  before(): log() {
      System.out.println("before method call");
  }

  after(): log() {
      System.out.println("after method call\n");
  }
}
```

Example 1: Logging Aspect

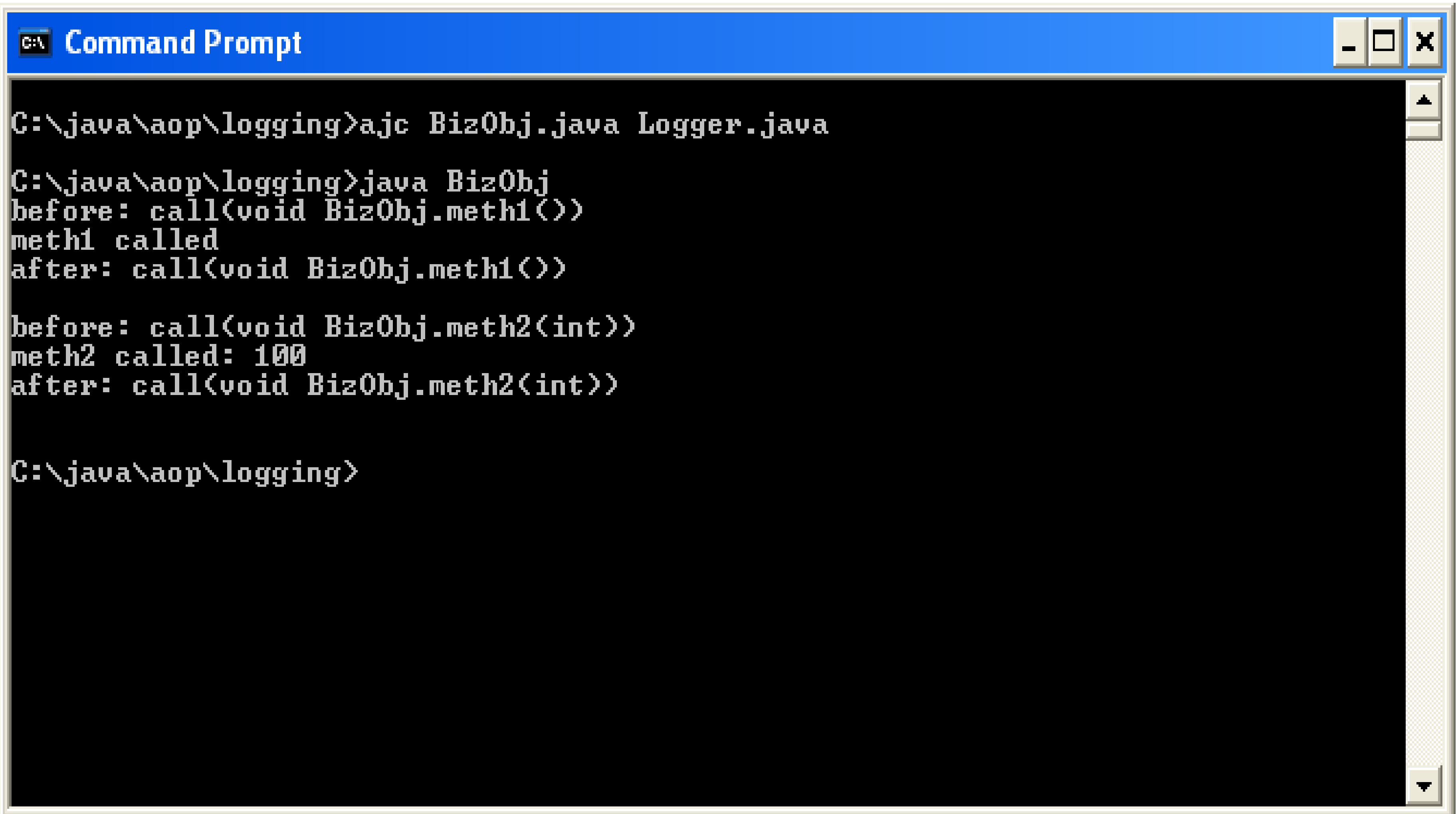


Example 1: Logging Aspect

Updated logging aspect

```
aspect Logger {
  pointcut log(): call(public * BizObj.*(..));

  before(): log() {
      System.out.println("before: " + thisJoinPoint);
  }

  after(): log() {
      System.out.println("after: " + thisJoinPoint +
  "\n");
  }
}
```

Example 1: Logging Aspect



Example 1: Logging Aspect

AspectJ Browser

File Project Tools

Build: Run: Save Options

Global View | package hierarchy

- files.lst
  - BizObj.java
    - BizObj
      - main
      - meth1
      - meth2
  - Logger.java
    - Logger
      - after
        - *advises method call sites*
          - BizObj.main: method-call(void BizObj.meth1())
          - BizObj.main: method-call(void BizObj.meth2(int))
      - before
        - *advises method call sites*
          - BizObj.main: method-call(void BizObj.meth1())
          - BizObj.main: method-call(void BizObj.meth2(int))
      - log

File View (files.lst)

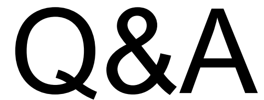
- BizObj.java
  - BizObj
    - meth1
    - meth2
    - main

```
public class BizObj {
        public void meth1() {
                System.out.println("meth1 called");
        }

        public void meth2(int value) {
                System.out.println("meth2 called: " + value);
        }

        public static void main(String[] args) {
                BizObj obj = new BizObj();
                obj.meth1();
                obj.meth2(100);
        }
}
```

Build succeeded in 2 second(s).

Example 2: Observer Pattern Aspect

- Interactive Walk Through



- Where is AOP headed in the Java world?
- Will there be a new language that wraps OOP and AOP more tightly together?

Resources

- Aspect Oriented Software Development
  http://aosd.net/
- AspectJ
  http://eclipse.org/aspectj/
- JBoss AOP
  http://www.jboss.org/
- AspectWerkz
  http://aspectwerkz.codehaus.org/