An Overview of Aspect-Oriented Programming

Presented by
Jarrell W. Waggoner

College of Engineering and Computing
University of South Carolina

03/19/08

Terminology of Aspect-Oriented Programming (AOP)

Term: **Concern**

Definition: A feature or essential operation that is part of a larger program or solution

Terminology of Aspect-Oriented Programming (AOP)

Term: **Concern**

Definition: A feature or essential operation that is part of a larger program or solution

Term: **Code Entanglement**

Definition: Code that cannot be separated into more than one concern

Terminology of Aspect-Oriented Programming (AOP)

Term: **Concern**

Definition: A feature or essential operation that is part of a larger program or solution

Term: **Code Entanglement**

Definition: Code that cannot be separated into more than one concern

Term: **Cross-Cutting Concern**

Definition: A concern that is entangled with one or more other concerns

Division of Concerns

How concerns are divided in different paradigms:

$$\text{Functional Programming} \Rightarrow \text{Functions}$$
$$\text{Object-Oriented Programming} \Rightarrow \text{Objects}$$

- Functions: Code separation
- Objects: Concern separation

Division of Concerns

How concerns are divided in different paradigms:

$$\text{Functional Programming} \Rightarrow \text{Functions}$$
$$\text{Object-Oriented Programming} \Rightarrow \text{Objects}$$
$$\text{Aspect-Oriented Programming} \Rightarrow \text{Aspects}$$

- Functions: Code separation
- Objects: Concern separation
- Aspects: Cross-Cutting concern separation

Cross-Cutting Concern Examples

List of cross-cutting concerns:

- System logging and tracing
- Error handling
- Statistics gathering
- Security handling
- Managed garbage collection

Cross-Cutting Concern Examples

List of cross-cutting concerns:

- System logging and tracing
- Error handling
- Statistics gathering
- Security handling
- Managed garbage collection

Most OOP languages are extended to support Aspect-Orientation rather than creating entirely new languages:

Common Lisp $\Rightarrow$ AspectL
Java $\Rightarrow$ AspectJ
C#/VB.Net $\Rightarrow$ Aspect.NET
C/C++ $\Rightarrow$ AspectC/Aspect C++

Aspects

Aspect-Oriented Programming Languages:

- Are fully object-oriented
- Add the “aspects” construct

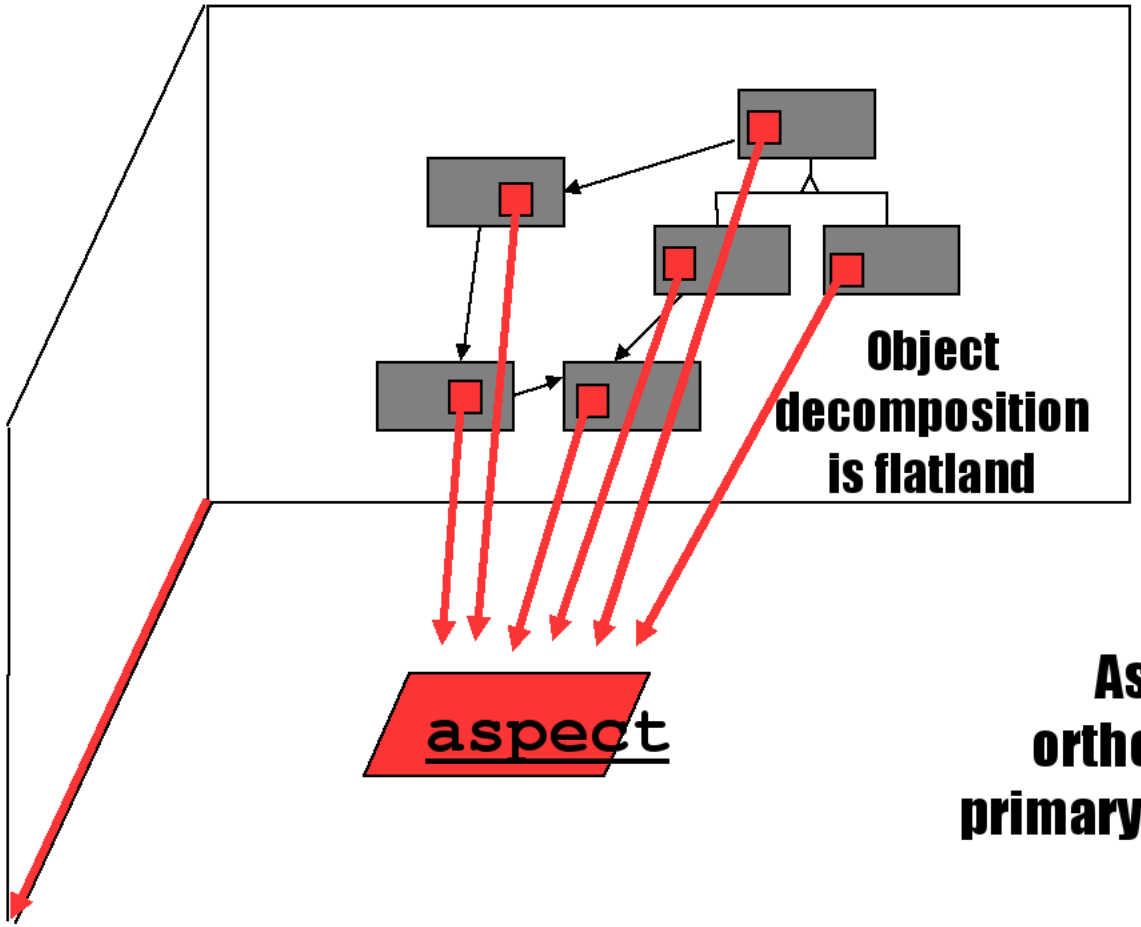
Aspects

Aspect-Oriented Programming Languages:

- Are fully object-oriented
- Add the "aspects" construct

Aspects:

- Encapsulate cross-cutting concerns that cannot be captured by traditional objects
- Generically applied to multiple objects
- No direct modification to the objects themselves
- Applied to all the objects in a program, or just a single object
- Can **add methods**, or run code around existing methods
- Can implement the methods defined by an interface (instead of requiring an implementing object to do this)



Aspects are orthogonal to the primary decomposition

1

[1] www.volantec.biz/Untangle_AOP.ppt

Aspect Example

Example of an aspect **SystemLog.aj**:

Example

```
public aspect SystemLog {

  private long Object.attribute;

  public void Object.methodToAdd(){
      // actions here
  }
}
```

Tree Traversal

Options for traversing a tree with varying types of nodes:

- "Traditional" OO approach
- "Functional" approach
- Visitor approach
- Aspect oriented approach

Note: Traversing our AST is a **cross-cutting** concern!

Anything besides Aspect-Oriented Programming is going to require redundant code, clumsy "hacks" or special patterns that "abuse" features of object-orientation.

Tree Traversal

Options for traversing a tree with varying types of nodes:

- "Traditional" OO approach
- "Functional" approach
- Visitor approach
- **Aspect oriented approach**

Note: Traversing our AST is a cross-cutting concern!

Anything besides Aspect-Oriented Programming is going to require redundant code, clumsy "hacks" or special patterns that "abuse" features of object-orientation.

Tree Traversal

Options for traversing a tree with varying types of nodes:

- "Traditional" OO approach
- "Functional" approach
- Visitor approach ← **modify this with AOP**
- Aspect oriented approach

Note: Traversing our AST is a cross-cutting concern!

Anything besides Aspect-Oriented Programming is going to require redundant code, clumsy "hacks" or special patterns that "abuse" features of object-orientation.

Tree Traversal (cont.)

Anything besides Aspect-Oriented Programming is going to require redundant code, clumsy "hacks" or special patterns that "abuse" features of object-orientation.

Example

The visitor pattern uses (and some would argue abuses) the polymorphic features of object-oriented languages to reduce the code that is required to be part of a collection of objects. Every object still has to have a minimal visit() method, however.

Aspect-Orientation will eliminate the need to ever touch the original objects. No need for a visit() method!

Visitor Pattern with AOP

Implementing the Visitor Pattern with AspectJ:

Example

```
aspect VisitAspect {
    void IfCommand.acceptVisitor(Visitor v) {
        v.visit(this);
    }
}
```

Visitor Pattern with AOP (cont.)

Or better yet:

Example

```
aspect VisitAspect {
    void AST+.acceptVisitor(Visitor v) {
        v.visit(this);
    }
}
```

AST+ means any object that inherits from the abstract AST class.

Diagram 1

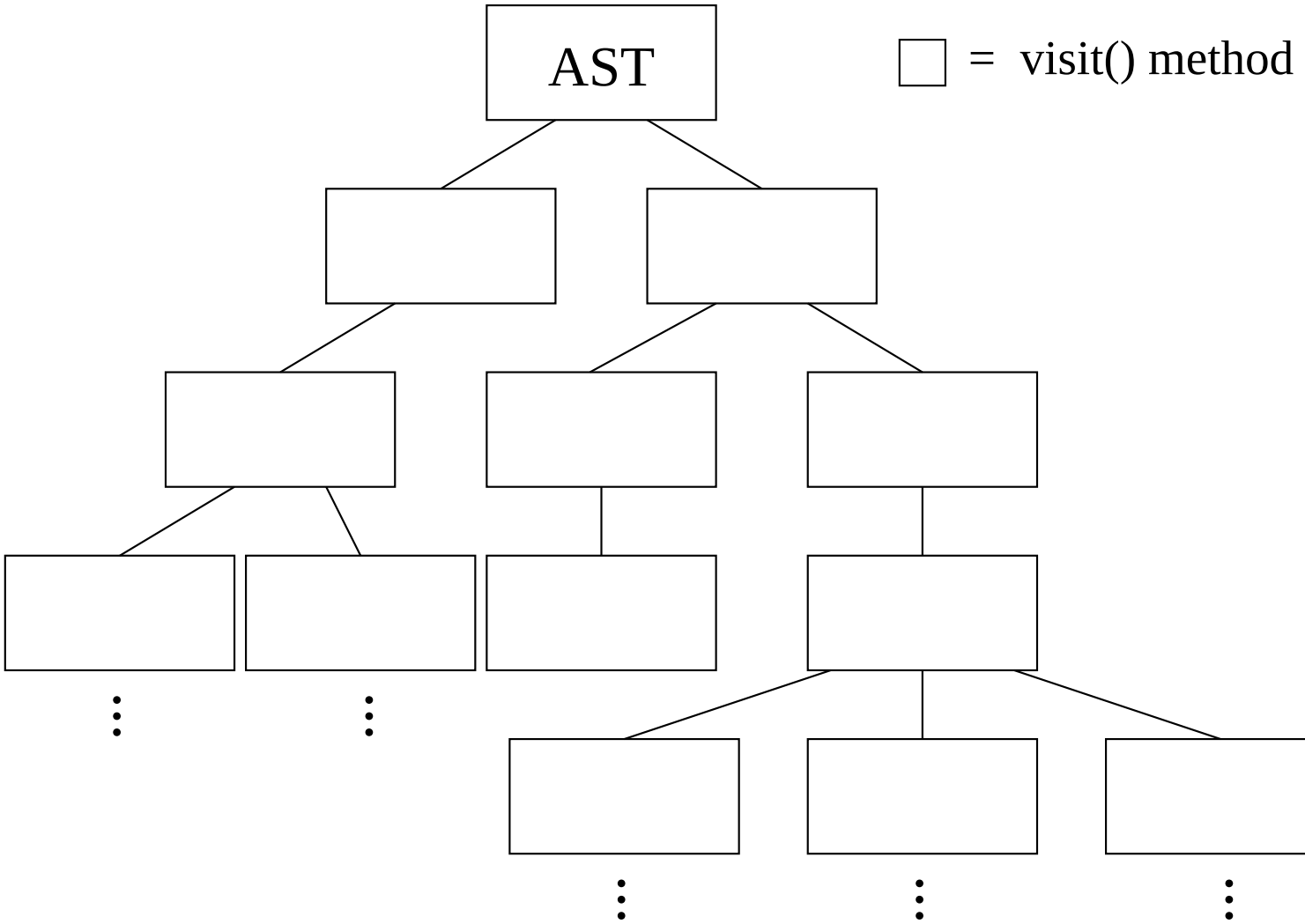


Diagram 2

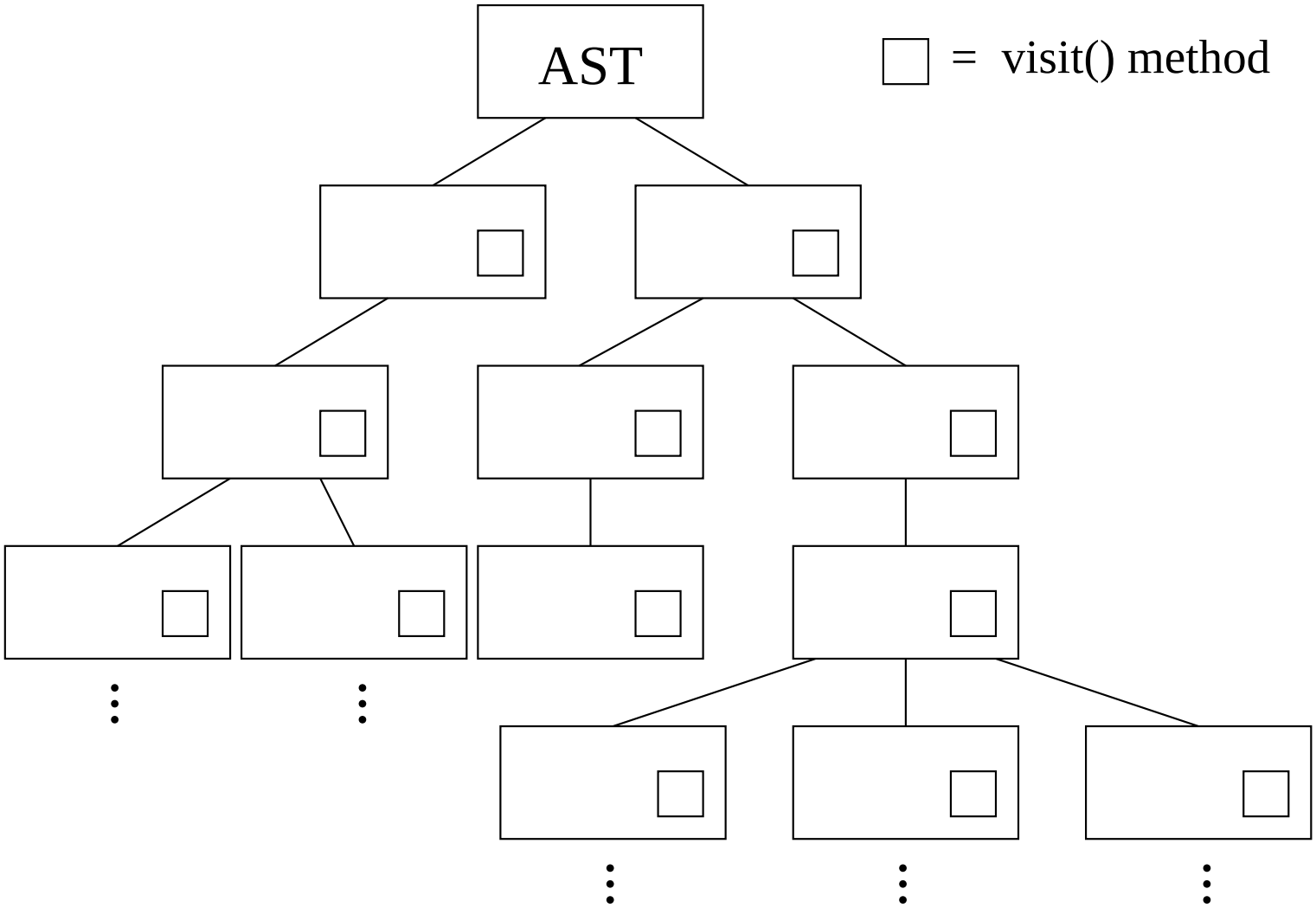
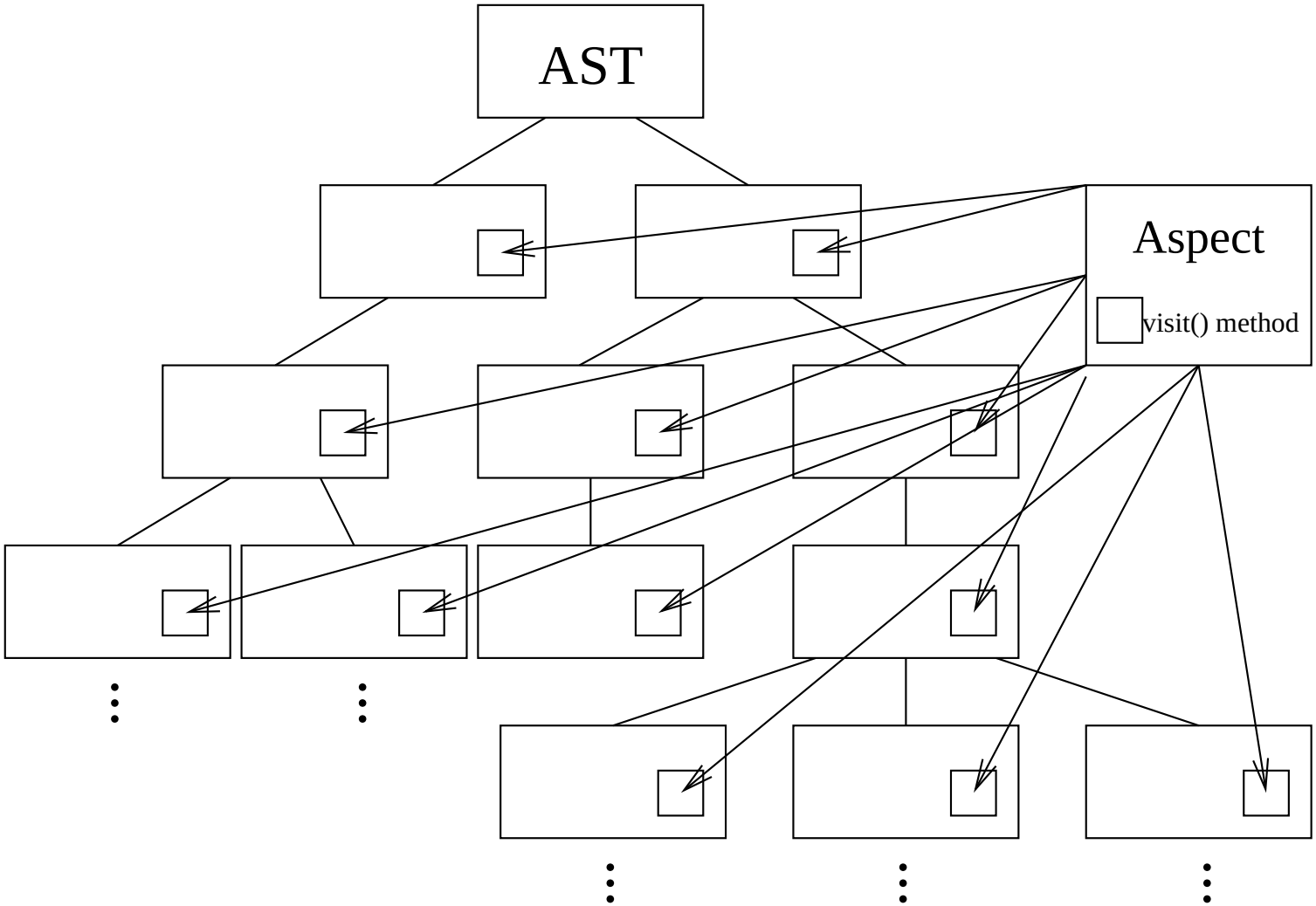


Diagram 3



Notes

Implementing the visitor pattern with AOP only uses one limited feature that is provided by the AOP framework.

Pointcuts, joinpoints, code weaving, runtime code weaving, etc...

There are better ways to traverse our AST with more AOP "tricks" that would be even more efficient than modifying the visitor pattern, and use even less code.

Importance of AOP

Is Aspect-Oriented Programming important?

- In 2001, MIT Technology Review listed AOP as one of the top 10 emerging technologies that will change the world
- One of the most dramatic examples of a code layer "above" traditional code
- Possibility of "aspect libraries" that can add high-level features to complex programs easily
- Implementation of security patches/fixes as run-time "aspect layers" for real-time systems

Summary

Much more to AOP than is discussed here:

- Advice Code woven into an object at a joinpoint
- Joinpoint A place where code can be woven into an object (creating a new method/attribute, before or after a current method, etc.)
- Pointcut A collection of joinpoints, perhaps across multiple objects
- Weaving Merging standard code with the associated aspects, either before compile-time, or at run-time
- Many more terms and concepts...

Summary (cont.)

Downsides to using AOP

- Difficult to manage
  Few programmers are trained to understand AOP, so the problem here is difficult to identify
- Difficult to debug
  The open-ended pointcut system can mean advice is being woven into many, many places
- Limited tool support
  Few programs understand AOP code, and even fewer can debug it
- Conceptual issues
  Arguments that AOP undermines fundamental structural and organizational programming properties