Aspect-Oriented Programming

Introduction to Aspect-Oriented Programming

Alesya Sheremet

Center for Software Technology, Utrecht University
http://www.cs.uu.nl/groups/ST/

September 10, 2007
Outline

1. Introduction
2. Cross cutting concerns in an application
3. The meaning of an aspect with respect to AOP
4. Aspect execution, debugging and tracing
5. Weaving
The fundamentals of AOP

1. Developed by Gregor Kiczales and his team at Xerox PARC
2. This team also developed the first and most popular general-purpose AOP language, AspectJ
3. IBM’s research team offered HyperJ
Crosscutting concern problems

- **Code tangling**: Modules in a software system may simultaneously interact with several requirements.
- **Code scattering**: Related implementations spread over many modules.
Terminology

1. **Aspects**: Features of a program that do not relate to the core functionality of the program directly, but are needed for proper program execution.

2. **Cross cutting concerns**: Modules in a software system may simultaneously interact with several requirements. Ex. Logging.

3. **Join point**: Point within a program where the aspect can be applied.

4. **Weaving**: The act of inserting calls to the aspect into the main program.
Aspects are concerns that are cross cutting the system’s functionality. They can’t be encapsulated to a single class.

- A new modularity mechanism with which crosscutting concerns can be expressed.
- An aspect holds all the information (operations and data) that a crosscutting concern uses/manipulates.
- Better modularity. Crosscutting concerns are now localized in an aspect.
- Easier understanding. The information about a concern is localized. It is less scattered and tangled with other information.
- Easier modifiability. Look at one aspect and its binding.
Is AOP really a new programming paradigm or is it just another optimisation technique based on existing languages (the paper talks mostly about optimisations anyway)? How does it resolve the complexity of existing languages? It seems that adding code separation between an aspect and a component will increase the overall complexity of the system.

Robin
Question

Is AOP really a new programming paradigm or is it just another optimisation technique based on existing languages (the paper talks mostly about optimisations anyway)?

How does it resolve the complexity of existing languages? It seems that adding code separation between an aspect and a component will increase the overall complexity of the system.

Robin

AOP is a programming technique aimed at achieving better modularization by separating cross-cutting concerns.

Optimization is not it’s primary goal.

AOP helps resolve complexity of software design.
Crosscutting Concerns

Concern = system property

- Functional property
- Constraint of system behavior
  - Constraints on a system refer to conditions that need to be satisfied when the system runs
    - Logging of read/write operations
    - Authorization
    - Synchronization
    - Checking for error conditions and acting accordingly
    - Exception handling
    -Trace output
Crosscutting concern problems

Code tangling and code scattering affect software design and developments in many ways:

- **Poor traceability**: Modules in a software system may simultaneously interact with several requirements.

- **Lower productivity**: Related implementations spread over many modules.

- **Less code reuse**: A module implements multiple concerns, other systems requiring similar functionality may not be able to readily use the module, further lowering productivity.

- **Poor code quality**: By targeting too many concerns at once, one or more of those concerns will not receive enough attention.

- **More difficult evolution**: Addressing future requirements often requires reworking the implementation.
AOP involves three distinct development steps:

1. **Aspectual decomposition**: Decompose the requirements to identify crosscutting and common concerns.

2. **Concern implementation**: Implement each concern separately.

3. **Weaving**: Compose the final system.
Anatomy of AOP languages

An AOP implementation consists of two parts: a language specification and an implementation.

The AOP language specification

1. Implementation of concerns: Mapping an individual requirement into code so that a compiler can translate it into executable code.

2. Weaving rules specification: How to compose independently implemented concerns to form the final system.

The AOP language implementation

1. Combine the individual concerns.

2. Convert the resulting information into executable code.
Aspect Weaver

Introduction
The fundamentals of AOP
Crosscutting concern problems
Terminology Aspects Question
Crosscutting Concerns
Implications
AOP development steps
Anatomy of AOP languages
Weaver

Different AOP implementation techniques
Aspect execution

Weaver
Different AOP implementation techniques

At the bottom of page 230 they mention that their weaver is not a "smart" compiler and that the programmer has to do everything himself. In the footnote they say that’s not so bad, but could it be done better? Would it be possible to make such a "smart" compiler? And would that be useful or would that actually completely contradict the whole point of AOP? Christiaan

1. **Static source weaving**: The programming language is extended to include constructs for defining aspects and where they get applied.

2. **Static byte weaving**: The source is compiled through the normal language compiler. Then an AOP weaver loads it into memory and looks at the aspect mapping mechanism (e.g. an xml file, attributes etc.) and injects byte code into appropriate places to execute the aspect.

3. **Dynamic weaving**: At runtime, there is some mechanism that emits code into memory to execute the aspects at the appropriate place.
Aspect execution

What kind of 'join points' are possible? Is it limited to matching on (nested) functions calls? Or is it also be possible to hook into things like an assignment? Or when an object is passed around? Peter

In the context of a method:
Aspects can get applied in three different ways: at the beginning of the method, the end of the method, or instead of the method.
But
There is some undergoing work at creating an aspect definition construct to define where specifically in a method to apply an aspect (inside a loop, in the else block of an if-then-else statement etc.)
How do you find your failure points? Is it the original source, the aspect source, or the AOP weaver?

**Code location problem** is usually caused by the removal, insertion, duplication or reordering of instructions. The problem causes the debugger to show the wrong source line or show byte code instead of source code.

- **Annotation**: ability to annotate aspect code to provide rich debug information, to allow the debugger to hide the code in support of debug obliviousness, and to support fault isolation. AspectJ uses byte-code annotation
AOP in Functional Programming

Are there any example of aspects applied to functional programming languages? Mark

- AspectOCaml, aspects are used in interpreter construction
- The integration of AOP and type classes in Haskell.