Aspect Oriented Programming

G. Kiczales, J. Lamping, A Mendhekar, C. Maeda,
C. Lopes, J. Loingtier, J. Irwin

CS780 – Adv. Software Engineering
October 8, 2003

Outline

- Introduction
- Why AOP?
- AOP fundamentals
- AOP: Early Visions
- Assessments of AOP
- Related Work
- AOP For Security
- Conclusions

Introduction

- Aspect Oriented Programming:
  - **Problem Statement**: POP and OOP do not allow all concerns that must be implemented in a system to be clearly and cleanly expressed
  - **Approach**: AOP attempts to provide a solution for encapsulating these concerns that have been difficult to capture

Why AOP?

Why AOP?

- Design methods decompose a system down into smaller and smaller units
- Programming languages give a way to define these abstract decompositions
- Traditional programming paradigms provide functional decomposition
  - e.g. Units encapsulated by **general procedures** such as functions, objects, etc.

Why AOP?

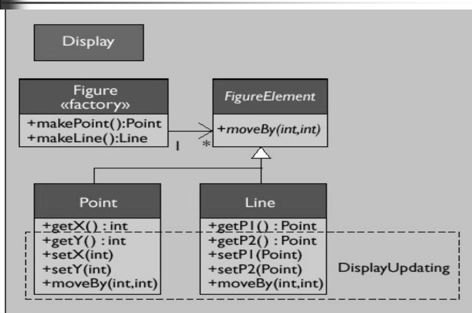
- But what happens when the programming paradigm does not provide clear way to express a certain design decision?
  - Implementation of decision is scattered throughout code
- "Real life" example of scattering:
  - Disney's Encyclopedia of Animated Characters

Why AOP?

- Such design decisions are difficult to express because they *cross cut the basic functionality of a system*
- AOP seeks to ensure that these "problematic" design decisions, called *aspects*, are expressed in a clear fashion
- AOP attempts to make design and implementation more modular

AOP Fundamentals

AOP Fundamentals



AOP Fundamentals

- Properties that must be implemented fall into two categories
  - Component: can be clearly encapsulated in a generalized procedure (i.e. Object, method, procedure, API)
  - Aspect: can't be cleanly encapsulated in a generalized procedure

AOP Fundamentals

- Goal of AOP:
  - "To support the programmer in cleanly separating components and aspects from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system"
  - Or more simply, to support "separation of concerns"

AOP Fundamentals

- Let's apply AOP to the figure editor example
  - Want to be able to reason about individual components and the display update feature
- Use AspectJ
  - Simple aspect oriented extension to Java
  - Allows cross cutting concerns to be implemented in a modular manner separately from the main source code
  - Meant to be easy to use for Java programmers

AOP Fundamentals

- But first, some AspectJ terminology
  - Join point: well defined points in execution of a program
  - Pointcut: collection of join points
  - Advice: a construct indicating code that should run at each join point in a pointcut
  - Aspect: program units encapsulating an implementation of a cross cutting property

AOP Fundamentals

- An aspect for updating the display using AspectJ:

```
aspect DisplayUpdating {
      pointcut move():
      call (void FigureElement.moveBy(int, int)) ||
                call(void Point.setX(int)) ||
                call(void Point.setY(int)) ||
                call(void Line.setP1(Point) ||
                call(void Line.setP2(Point);
     after(): move() {
                Display.needsRepaint();
      }
   }
```

AOP Fundamentals

- Benefits of the AOP approach
  - The cross cutting concern is explicitly captured
  - Evolution of the code is simplified
  - The encapsulated concern is now “pluggable”
  - Stabilizes the implementation

AOP Fundamentals

- **Aspect Weaving**: process of coordinating and combining the aspect code and the program code
- **Aspect Weaver**: tool used combine the aspect and non-aspect programs to produce the final system
  - weaver relies on the join points to map the aspect code to the program code
  - Weaving can occur at load time, during compilation, or can be handled by a pre-processor

AOP Fundamentals

- AOP is NOT a replacement for Object-Oriented Programming (OOP)!
  - Meant to build on OOP and provide techniques for solving problems for which OOP is insufficient
  - Procedural programming was not abandoned during the shift to OOP
  - AOP uses procedures, objects and aspects
  - POP, OOP, AOP, ????

AOP: Early Visions

AOP: Early Visions

- Aspect language customized for a particular problem
- Multilple aspect languages may be required for a single application
- Customized aspect weaver needed

AOP: Early Visions

- Example #1: Image Processing
  - Without AOP, there is the easily understandable implementation or the efficient implementation
  - AOP implementation provides component language for functionality and aspect language for efficiency
  - Data flow graph of component program is used as the guide for fusing loops at weave time

AOP: Early Visions

- Example #2:
  Document Processing/Distributed Objects
  - Component language represents documents, repositories, printers, etc.
  - Communication aspect to control bandwidth used
  - Aspect language allows for specification of how much of an object to copy using RMI

Assessments of AOP

Assessments of AOP: Image Processing

- Qualitative evaluation of applying AOP to image processing program – easier to understand
  - Easy to understand how components compose
  - Can easily see how aspect will affect program
  - Changing components or aspects is simple – reweave
  - Programmer is unconcerned with the details of the final program

Assessments of AOP: Image Processing

- Performance of AOP and “tangled” versions comparable:
  - AOP version: 1039 lines of code
  - “Tangled version”: 35215 lines of code
- Metric: Reduction in code bloat = (tangled code size – component program size)/sum of aspect program sizes
- Result for Image processing example: (35213-756)/352=98

Assessments of AOP: “Don't Bother”

- Adam Kolawa, CEO of Parasoft:
  - AOP compelling on the surface but....
  - Views AOP as a general solution to performance optimization
  - Not worth training programmers in a new art when computers keep getting faster
  - Put extra effort into solving business/scientific problems

Assessments of AOP: A Study of AspectJ

- Exploratory experiments for two programming tasks:
  - Debugging
  - Changing the code
- Digital Library served as the test system
- For each experiment:
  - One group used AspectJ with an OO language;
  - Other group used only the OO language

Assessments of AOP: A Study of AspectJ

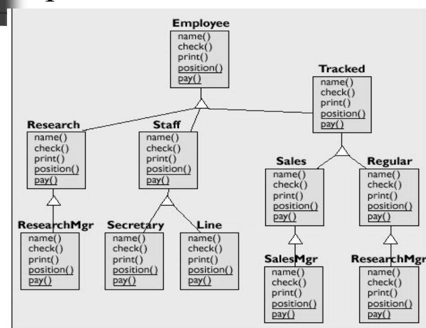
- Experiment #1: Debugging
  - AspectJ users finished debugging tasks faster
  - Improvement more pronounced when fault code was “modularized”
- Experiment #2: Code Modification
  - Non-AspectJ users finished changes faster
  - AOP users tried to code changes quickly using aspects, resulting in more failures

Related Work

Related Work: Multidimensional Separation of Concerns

- Addresses the need for the separation of concerns in programming (e.g., feature concern, data concern)
- Goal: to end the “Tyranny of the dominant dimension”
  - In OO, dominant dimension is the class
- Allow developers to decompose and integrate these various dimensions
  - Tool: HyperJ for Java developers

Related Work: Multidimensional Separation of Concerns



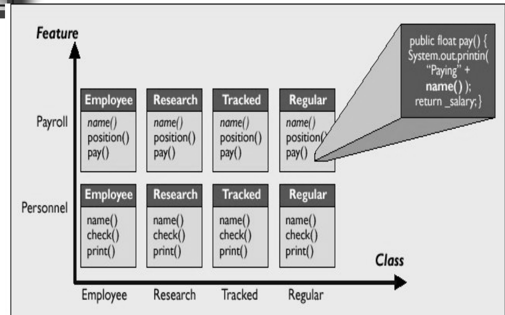
Decomposing the System

- Using Hyper/J:

```
package personnel: Feature.Personnel
operation position: Feature.Payroll
operation pay: Feature.Payroll
```

- Default: everything in Personnel package part of Personnel feature
- Override: methods named position or pay part of Payroll feature

Related Work:Multidimensional Separation of Concerns



Re-integrate the System

- Using Hyper/J:

```
hypermodule PayrollPlusPersonnel
 hyperslices: Payroll, Personnel;
 relationships: mergeByName;
end hypermodule
```

- Merge Payroll and Personnel hyperslices back together to form original system

AOP for Security

AOP for Security

- The Aspect-Oriented Security Solution:
  - Build security into a system from the start; don't add it on later
  - Create an aspect language for adding security primitives to code
  - Separates the "functionality" and "security" concerns
  - Aspect weaver combines primary program and security aspects

AOP for Security

- Created an aspect language, C-saw, and weaver for C
- Wrote several aspects to deal with general security-related issues:
  - TOCTOU
  - Buffer overflow
  - Format string attack

Conclusions

- AOP addresses limitations of current programming paradigms:
  - Some properties can be represented with traditional programming paradigms; others cannot
  - Encapsulating these "difficult" properties with aspects can improve modularity
- Authors see AOP as the next step in the evolution of programming

References

- An Overview of AspectJ. G. Kiczales, et. al.
- Getting Started with AspectJ. G. Kiczales et. al.
- Discussing Aspects of AOP. ACM panel discussion
- Ghosts from the Past. A. Kolawa
- An Initial Assessment of AOP. R. Walker, E. Baniassad, G. Murphy

References

- Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. H. Ossher, P.Tarr
- www.cigital.com
- Employment at Cigital, Inc.

Discussion

- In what other types of situations could aspects be useful?
- Does anyone see major drawbacks of AOP?
- What do you think of Kolawa's critique?
- Do you think this will be a step forward in programming that is as important the shift to OOP?