A Brief Introduction to Aspect-Oriented Programming

Historical View Of Languages

• Procedural language
• Functional language
• Object-Oriented language

Acknowledgements

• Zhenxiao Yang
• Gregor Kiczales
**Procedural Language**

- Also termed imperative language
- Describe
  - An explicit sequence of steps to follow to produce a result
- Examples: Basic, Pascal, C, Fortran

**Functional Language**

- Describe everything as a function (e.g., data, operations)
- (+ 3 4); (add (prod 4 5) 3)
- Examples
  - LISP, Scheme, ML, Haskell

**Logical Language**

- Also termed declarative language
- Establish causal relationships between terms
  - Conclusion :- Conditions
  - Read as: If Conditions then Conclusion
- Examples: Prolog, Parlog
Object-Oriented Programming

• Describe
  – A set of user-defined objects
  – And communications among them to produce a (user-defined) result
• Basic features
  – Encapsulation
  – Inheritance
  – Polymorphism

OOP (cont’d)

• Example languages
  – Smalltalk, C++, Java
  – Many other:
    • Ada, Object Pascal, Objective C, DRAGOON, BETA, Emerald, POOL, Eiffel, Self, Oblog, ESP, POLKA, Loops, Perl, VB
• Are OOP languages procedural?

We Need More

• Major advantage of OOP
  – Modular structure
• Potential problems with OOP
  – Issues distributed in different modules result in tangled code.
  – Example: error logging, failure handling, performance optimizations
• Potential result: Tangled Code
  – Change in the way logging is done affects many classes
Example of Tangled Code

Untangling the Tangled Code

- Constraint:
  - Want to preserve the benefits of OOP (encapsulation, modularity, inheritance, etc.)
- Potential Solution:
  - Aspect-Oriented Programming

Basic Concepts in AOP

- Crosscutting: straddle across functional and hierarchical boundaries
- Aspect:
  - Property cannot be cleanly encapsulated into a single procedure
  - Tend to affect performance or semantics of components in systematic ways

Red indicates the error-logging code
AOP: Languages

- Components:
  - Component program
  - Aspect definition
  - Aspect Weaver

- Constructs:
  - Join point: execution point in component program for integrating aspects
  - Pointcuts: refers to collection of join points and values of variables at those points
  - Advice: method-like constructs that define additional behavior at join points
  - Aspects: “units of modular cross-cutting implementation”
    - Pointcuts
    - Advice
    - Regular (Java) code

Pictoral Representation

AspectJ by Example
AspectJ by Example (cont’d)

• Define pointcuts
• Define advice
• Introduction

Pointcuts

• pointcut
  – pointcut move():
    call(void FigureElement.setXY(int, int))
    || call(void Point.setX(int))
    || call(void Point.setY(int))
    || call(void Line.setP1(Point))
    || call(void Line.setP2(Point));
  – pointcut produce (void Figure.make*(..))
  – pointcut setXY(FigureElement fe, int x, int y):
    call(void fe.setXY(x, y));

Advice

• Advice
  – after(): move(){
    System.out.println("A figure element moved.");
  }
  – after(FigureElement fe, int x, int y):setXY(fe,x,y){
    System.out.println(fe + " moved to " +x+ ", " +
y);
  }
Aspects

aspect FigureLog{
  pointcut setXY(FigureElement fe, int x, int y):
    calls(void fe.setXY(x, y));
  after(FigureElement fe, int x, int y): setXY(fe, x, y){
    System.out.println(fe + " moved to (" + x + ", " + y + ").");
  }
}

Introduction

• Introduction
  – Add members to a set of Classes
  – Change inheritance structure of classes

Introduction (cont’d)

aspect PointObserving {
  private Vector Point.observers = new Vector();

  public static void addObserver(Point p, Screen s) {
    p.observers.add(s);
  }

  public static void removeObserver(Point p, Screen s) {
    p.observers.remove(s);
  }
}
public class A1{
    function foo(){…}
}
public class A2{
    function foo(){…; super.foo();…}
}
public class A3{
    function foo(){…; super.foo();…}
}
aspect A1A2A3{
    declare parents: A2 extends A1;
    declare parents: A3 extends A2;
}

aspect A1A3A2{
    declare parents: A3 extends A1;
    declare parents: A2 extends A3;
}

What Can AspectJ Do for Us

- Developing
  - Tracing, Logging, and Profiling
  - Pre- and Post-Conditions
  - Contract Enforcement
- Production
  - Change Monitoring
  - Synchronization
  - Context Passing
  - Providing Consistent Behavior
- Introduction
Conclusion and Open Issues

- AOP vs OOP
  - AOP is not substitute for OOP
  - AOP makes OOP more powerful
- AOP also has runtime overhead
  - We should use OOP as much as possible
- Reuse of aspects
- Tool support

Related Work

- AspectJ
- Reflection and meta-object protocols
  - Meta-object provides mechanism to control over base-objects
- Subject-Oriented Programming
- Intentional Programming