A Brief Introduction to Aspect-Oriented Programming

Historical View Of Languages

- Procedural language
- Functional language
- Object-Oriented language
Acknowledgements

• Zhenxiao Yang
• Gregor Kiczales
• Eclipse website for AspectJ (www.eclipse.org/aspectj)

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**
  - First OOP language: SIMULA-67 (1970)
  - 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

Red indicates the error-logging 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
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

---

**Pictorial Representation**

```
  Compiler -> Executable
  Compiler -> Executable

  Aspect

  Weaver

  Compiler -> Executable
```

Hook points for cross-cutting concern
AspectJ by Example

- Define pointcuts
- Define advice
- Introduction

UML for the FigureEditor example

http://www.eclipse.org/aspectj
Name-Based Pointcuts

- Identify specific join point within a program flow
  
  \[
  \text{pointcut } \text{call}(\text{void Point.setX(int)})
  \]
  - Looks for specific signature

- Identify a collection of multiple signatures according to Boolean operators (&&, ||, !)
  
  \[
  \text{pointcut}
  \begin{align*}
  &\text{call}(\text{void Point.setX(int)}) || \\
  &\text{call}(\text{void Point.setY(int)})
  \end{align*}
  \]

http://www.eclipse.org/aspectj

Name-based Pointcut

- Naming a more complicated pointcut
  
  \[
  \text{pointcut } \text{move}():
  \begin{align*}
  &\text{call}(\text{void FigureElement.setXY(int,int)}) \\
  &|| \text{call}(\text{void Point.setX(int)}) \\
  &|| \text{call}(\text{void Point.setY(int)}) \\
  &|| \text{call}(\text{void Line.setP1(Point)}) \\
  &|| \text{call}(\text{void Line.setP2(Point)});
  \end{align*}
  \]

- Perform action if any of these pointcuts are encountered

http://www.eclipse.org/aspectj
Property-based Pointcuts

• Identify pointcuts by properties of methods (e.g., use wildcards *)
  – Identify by name prefix:
    ```
    pointcut call(void Figure.make*(..))
    Pick out join point whose method starts with "make"
    ```
  – Identify by visibility
    ```
    pointcut call(public * Figure.* (..))
    ```

Source: www.eclipse.org
http://www.eclipse.org/aspectj

Advice

• Advice links together a pointcut with a segment of code (to be run at each join point)

• 3 types of advice:
  – before: runs as join point is reached, before transferring control at join point
  – after: after execution of join point (before control is returned to caller)
    ```java
    after(): move(){
        System.out.println("A figure element moved. ");
    }
    ```
  – around: runs as the join point is reached, maintains control of program.

http://www.eclipse.org/aspectj
Advice and Context

• Pointcuts:
  – Pick out join points
  – Also expose part of execution context at join point
  – Values from context can be used in advice body.

Advice

• Example of context from pointcut
  ```java
  pointcut setXY(FigureElement fe, int x, int y):
  call(void FigureElement.setXY(x, y))
  && target (fe)
  && args (x, y);
  – Naming FigureElement as fe, identify parameters (arguments) x and y
  ```

• Context used in Advice
  ```java
  after (FigureElement fe, int x, int y)returning:
  setXY(fe,x,y){
    System.out.println(fe + " moved to " +x,+ " ", + y);
  }
  ```

http://www.eclipse.org/aspectj
Advice and pointcut

```java
after(FigureElement fe, int x, int y)
    returning:
        call(void FigureElement.setXY(int, int))
        && target(fe)
        && args(x, y) {
            System.out.println(fe + " moved to (" + x + ", " + y + ")");
        }
```

http://www.eclipse.org/aspectj

Aspect Definition Example

```java
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 + ").");
        }
}
```

http://www.eclipse.org/aspectj
Introduction: Inter-type Declarations

- Declarations that cut across classes and their hierarchies
  - Declare members across multiple classes
  - Change inheritance relationship

- Operates statically at compile-time.

Introduction (cont’d)

```java
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);
    }

    pointcut changes(Point p): target(p) && call(void Point.set*(int));
    after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
            updateObserver(p, (Screen)iter.next());
        }
    }
    static void updateObserver(Point p, Screen s) {
        s.display(p);
    }
}
```

http://www.eclipse.org/aspectj
aspect PointObserving {

...  
pointcut changes(Point p): target(p) && call(void Point.set* (int));
after(Point p): changes(p) {
    Iterator iter = p.observers.iterator();
    while ( iter.hasNext() ) {
        updateObserver(p, (Screen)iter.next());
    }
}
static void updateObserver(Point p, Screen s) {
    s.display(p);
}
}

http://www.eclipse.org/aspectj

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;
}
public class A1{
    function foo(){…}
}
public class A2{
    function foo(){…;super.foo();…}
}
public class A3{
    function foo(){…;super.foo();…}
}
aspect A1A3A2{
    declare parents: A3 extends A1;
    declare parents: A2 extends A3;
}
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