Aspect-Oriented Programming

David Talby

What is it?

- A new addition to the world of programming
  - New concepts & language constructs
  - New tools (aspect compiler, browser, debugger)
  - Many examples & patterns of practical use
  - A lot of hype
- Key advantage: separation of concerns
  - Cross-cutting concerns: those that are not well encapsulated in the usual OOD way – classes
  - Often appear in the form of 'code guidelines'

For Example

```
public class SomeClass extends OtherClass {
        // Core data members
        // Other data members: Log stream, consistency flag

        public void DoSomething(OperationInformation info) {
            // Ensure authentication
            // Ensure info satisfies contracts
            // Lock the object in case other threads access it
            // Ensure the cache is up to date
            // Log the start of operation
            // ==== Perform the core operation ====
            // Log the completion of operation
            // Unlock the object
            // Do Standard Exception Handling
        }
        // More operations similar to above
    }
```

Cross-Cutting Concerns

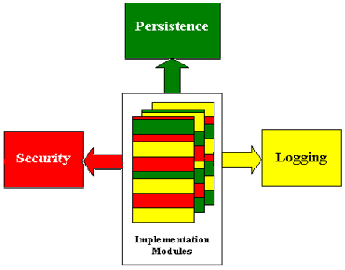
- Logging
- Debugging
- Profiling (Performance)
- Security & Authentication
- Exception Handling
- Design by Contract
- Event Handling
- Synchronization
- Resource Pooling
- Others…

Current Solutions

- Problems: Code Tangling, Code Scattering
  - Reduced reuse, speed, quality, ability to change
- Design patterns can solve some problems
  - Proxy, Template Method solve some cases
  - Visitor, Strategy solve other cases
- Frameworks provide domain-specific solutions
- But it's not a solution for cases in which:
  - Polymorphism can't be used (exceptions, DbC)
  - Concerns are only used during debug, and change a lot
  - The designer didn't plan for a given concern
  - The framework wasn't designed to consider a concern

Separation of Concerns

- Separate logical concerns should be in separate modules of code – called aspects



OOD & AOP

- Object-Oriented Programming
  - Basic concept of modularity : the class
  - Good for common concerns (inheritance)
  - A program is a set of classes
- Aspect-Oriented Programming
  - Basic concept of modularity: the aspect
  - Good for unrelated concerns (pointcuts)
  - A program is a set of aspects
- AOP complements OOD

AspectJ

- AspectJ is the leading AOP implementation, and the only full, stable and widely used one
- It includes a language specification
  - A set of additions to the Java language
  - A compiler that creates standard Java bytecode
- It includes a set of tools
  - Aspect-aware debugger and documentation tool
  - Visual aspect browser
  - Integration with popular IDEs

Hello, World

- Let's start with a simple example

```
// HelloWorld.java
public class HelloWorld {
    public static void say(String message) {
        System.out.println(message);
    }

    public static void sayToPerson(
                        String message, String name) {
        System.out.println(name + ", " + message);
    }
}
```

Polite Hello, World

- Guess what the following aspect does

```
// MannersAspect.java
public aspect MannersAspect {
    pointcut callSayMessage() :
        call(public static void HelloWorld.say*(..));

    before() : callSayMessage() {
        System.out.println("Good day!");
    }

    after() : callSayMessage() {
        System.out.println("Thank you!");
    }
}
```

Running the Example

- Just Compile and Run
  - *ajc HelloWorld.java MannersAspect.java* (or *.aj)
  - *ajc –argfile PoliteHelloWorld.lst*
- What's in the example
  - A Pointcut defines at which points in the dynamic execution of the program – at what Join Points – extra code should be inserted
  - An Advice defines when, relative to the join point, the new code runs, and that actual code
  - An Aspect encapsulates pointcuts and advices

Join Points

- Well-defined points in a program's execution
- AspectJ makes these join points available:
  - Method call and execution
  - Constructor call and execution
  - Read/write access to a field
  - Exception throwing or handler execution
  - Object and class initialization execution
- A join point may include other join points
- A join point may have a context

Pointcuts

- Definition of a collection of join points
- Most common kind – the call pointcut:
  - call(public void MyClass.myMethod(String))
  - call(void MyClass.myMethod(..))
  - call(* MyClass.myMethod*(..))   // * means wildcard
  - call(* MyClass.myMethod*(String,..))
  - call(* *.myMethod(..))
  - call(MyClass.new(..))
  - call(MyClass+.new(..))   // + is subclass wildcard
  - call(public * com.mycompany.*.*(..))

Example 1: Tracing

- Print debug traces of method calls and their timing for all methods of class MyClass
- Note the use of anonymous pointcuts

```
public aspect MyClassTrace {
   before() : call(public * MyClass.*(..)) {
      System.out.println("Before: " + thisJoinPoint + " " +
                                 System.currentTimeMillis()); }
   after() : call(public * MyClass.*(..)) {
       System.out.println("After: " + thisJoinPoint + " " +
                                 System.currentTimeMillis()); }
}
```

thisJoinPoint

- A useful reflection-like feature, can provide:
- the kind of join point that was matched
- the source location of the current join point
- normal, short and long string representations of the current join point
- actual argument(s) to the method / field of the join point
- signature of the method or field of the current join point
- the target object
- the currently executing object
- a reference to the static portion of the object holding the join point; also available in thisJoinPointStaticPart

Example 2: Tracing Revisited

- First solution using an aspect:

```
aspect TraceEntities {
   pointcut myClasses():
        within(MyClass+);
   pointcut myConstructors():
        myClasses() && call(new(..));
   pointcut myMethods():
        myClasses() && call(* *(..));
   before (): myConstructors() {
       Trace.traceEntry("Before Constructor: "+
                thisJoinPointStaticPart.getSignature()); }
   before (): myMethods() {
       Trace.traceEntry("Before Method: " +
                thisJoinPointStaticPart.getSignature()); }
```

Within and CFlow Pointcuts

- Be inside lexical scope of class or method
  - within(MyClass)   // of class
  - withincode(* MyClass.myMethod(..))   // of method
- Be inside the control flow of another pointcut
  - If a() calls b(), then b() is inside a()'s control flow
  - cflow (call(* MyClass.myMethod(..))
  - Any pointcut can be used as the base of cflow
  - Control flow is decided in runtime, unlike within
  - cflowbelow(Pcut) is similar, but ignores join points that are already in PCut

Example 3: Contract Enforcement

- Useful to check assertions, use Design by Contract, or validate framework assumptions
- The following checks that only certain factory methods can put objects in a central Registry

```
aspect RegistrationProtection {
   pointcut register():
        call(void Registry.register(Element));
   pointcut canRegister():
        withincode(static * Element.make*(..));
   before(): register() && !canRegister() {
        throw new IllegalAccessException("Illegal call " +
                                               thisJoinPoint); } }
```

Example 4: Profiling

- It’s easy to ask very specific questions, and quickly modify them, all outside the real code
- Note that withincode wouldn’t work here

```
aspect SetsInRotateCounting {
  int rotateCount = 0;
  int setCount = 0;
  before(): call(void Line.rotate(double)) {
    rotateCount++; }
  before():
    call(void Point.set*(int)) &&
    cflow(call(void Line.rotate(double))) {
      setCount++; }   }
```

Context-Based Pointcuts

- Pointcuts based on dynamic, runtime context
  - this(JComponent+)   // ‘this’ object inherits from JComponent
  - target(MyClass)   // match target object of current method call
  - args(String,..,int)   // match order & type of arguments
  - args(IOException)   // type of argument or exception handler
- Dynamic – so these are not equal:
  - call(* Object.equals(String))
  - call(* Object.equals(Object)) && args(String))
- Always used in conjunction with other pointcuts

Exposing Context in Pointcuts

- A pointcut can define arguments
  - Each argument must have a type
  - Each must be bound by a context-based pointcut
  - The arguments can be passed to the advice
- Here’s another custom tracing example:

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

Example 5: Pre- and Post-Conditions

- Verify that *setX()* and *setY()* in class *Point* do not receive out-of-bound arguments

```
aspect PointBoundsChecking {
  pointcut setX(int x): call(void Point.setX(int)) && args(x));
  pointcut setY(int y): call(void Point.setY(int)) && args(y));
  before(int x): setX(x) {
    if ( x < MIN_X || x > MAX_X )
      throw new IllegalArgumentException("x out of bounds"); }
  before(int y): setY(y) {
    if ( y < MIN_Y || y > MAX_Y )
      throw new IllegalArgumentException("y out of bounds"); } }
```

Execution Pointcuts

- Join point in which a method starts executing
  - execution(* MyClass.myMethod*(..));
  - execution(MyClass+.new(..))
- Behaviors different form call pointcuts
  - In execution, the within and withincode pointcuts will refer to the text of the called method
  - In execution, The dynamic context pointcuts will refer to the context of the called method
  - call does not catch calls to (non-static) super methods
- Use call to match calling a signature, use execution for actually running a piece of code

Advice

- Defines the code to run, and when to run it
- Advide kinds: before(), after() and around()
- Before advice runs before the join point
- After advice has three variants
  - after(): register()   { registry.update(); }
  - after() returning move()   { screen.update(); }
  - after() throwing (Error e): { log.write(e);   }
- Around advice surrounds original join point
  - Can replace it completely, and return a different value
  - Can run it one or more times with proceed()
  - Can run it using different arguments

Example 6: Resource Pooling

- A global connection pool should be used
  - Original code is oblivious of the pool, so the following code surrounds Connection.close()
  - To complete the implementation, the constructor of class Connection must be surrounded as well

```
void around(Connection conn) :
   call(Connection.close()) && target(conn) {
      if (enablePooling) {
         connectionPool.put(conn);
      } else {
         proceed();
      }
   }
```

More Pointcut Kinds

- Field access
  - get(PrintStream System.out)
  - set(int MyClass.x)
- Exception handling (entering *catch* execution)
  - handler(RemoteException)
  - handler(IOException+)
  - handler(CreditCard*)
- Conditional tests
  - if(EventQueue.isDispatchThread())
  - The Boolean expression can use static methods and fields, fields of the enclosing aspect, and thisJoinPoint

Example 7: Error Logging

- Log all errors (not exceptions) thrown out of package com.acme.* to a log
- Use cflow() to prevent logging an error twice, in case it was raised internally in com.acme.*

```
aspect PublicErrorLogging {
   pointcut publicMethodCall():
      call(public * com.acme.*.*(..));
   after() throwing (Error e):
      publicMethodCall() &&
      !cflow(publicMethodCall())
      if (Logger.traceLevel() > 0) {
         Logger.write(e);  }
}
```

Aspects

- Unit that combines pointcuts and advices
- Can contain methods and fields
- Can extend classes or implement interfaces
- Cannot create an 'aspect object' using *new*
- Aspects and pointcuts can be abstract
- Classes can define pointcuts too
  - These must be declared static
  - This is not recommended practice
  - Advices can't be declared inside classes

Fields in Methods in Aspects

- Fields can be used to collect data
  - See example 4 – profiling
- Methods can be used as in any regular class

```
aspect YetAnotherLoggingAspect {
   private static Log log = new Log();
   public static void clearLog() { log.clear(); }
   pointcut publicMethodCall(): call(public * com.acme.*.*(..));
   after() throwing (Error e):
      publicMethodCall() { log.write(e);  }  }
```

- Aspects are by default singletons
  - But there are other supported association types: perthis, pertarget, percflow, percflowbelow

Example 7: Authentication

- Abstract aspects allow even more reuse
- Here's a generic aspect for authentication through a singleton Authenticator:

```
// AbstratcAuthenticationAspect.java
   public abstract aspect AbstractAuthenticationAspect {
     public abstract pointcut opsNeeddingAuthentication();
     before() : opsNeeddingAuthentication() {
         // Perform authentication. If not authenticated,
         // let the thrown exception propagate.
         Authenticator.authenticate();
      }  }
```

Example 7: Authentication II

- A concrete aspect for a database app:

```
// DatabaseAuthenticationAspect.java
   public aspect DatabaseAuthenticationAspect
      extends AbstractAuthenticationAspect {

      public pointcut opsNeeddingAuthentication():
         call(* DatabaseServer.connect());
   }
```

Example 8: Functional Guidelines

- “Every time a slow operation is used, the cursor should turn into a wait cursor”

```
public abstract aspect SlowMethodAspect {
      abstract pointcut slowMethods(UIManager ui);
      void around(UIManager ui) :
          slowMethods(ui) {
              Cursor originalCursor = ui.getCursor();
              Cursor waitCursor = Cursor.WAIT_CURSOR;
              ui.setCursor(waitCursor);
              try {
                     proceed(ui);
              } finally {
                   ui.setCursor(originalCursor);
              }   }    }
```

Functional Guidelines

- Code of aspected classes doesn’t change
- Multiple aspects can co-exist
- Same pattern is useful for many other cases
  - Security
  - Resource Pooling, Caching, Copy on write, …
  - Creation by Factory, Lazy Creation, …
  - Multi-Thread Synchronization
  - Transaction Definition
  - Monitoring System Notification
  - Standard Exception Handling

Introductions

- Modify the static form of a class
- Add fields to an existing class
  - private boolean Server.disabled = false;
  - public String Foo.name;
- Add methods to an existing class
  - public int Point.getX() { return x; }
  - public String (Point || Line).getName() { return name; }
- Add Constructors
  - public Point.new(int x, int y) { this.x = x; this.y = y; }

Introductions II

- Extend an existing class with another
  - declare parents: Point extends GeometricObject;
- Implement an interface with an existing class
  - declare parents: Point implements Comparable;
- “Soften” Exception
  - Convert checked exceptions to unchecked ones
  - Wraps exceptions in org.aspectj.lang.SoftException
  - declare soft: CloneNotSupportedException:
    execution(Object clone());

Example 9: Adding Mixins

- Given a standard *Point* class, with private fields *x,y* we can make it cloneable:

```
aspect CloneablePoint {
    declare parents: Point implements Cloneable;
    declare soft: CloneNotSupportedException:
          execution(Object clone());
    Object Point.clone() { return super.clone(); }
}
```

- Being Cloneable is an example of a mixin, like Comparable, Serializable or Persistent

Introductions: Compiler Warnings

- Add a compile-time warning or error
- Issued if there is a chance that code will reach a given pointcut
- Warning / error string can be defined
- declare warning: Pointcut: String;
- declare error: *Pointcut*: *String*;
- The pointcuts must be statically determinable
  - Not allowed: this, target, args, if, cflow, cflowbelow

Example 10: Flexible Access Control

- Control method access beyond *private*, *protected* and *public* declarations
- Violations must be found at compile time
- For example, class Product can only be initialized and configured by specific classes

```
public class Product {
        public Product() {
            /* constructor implementation  */  }
        public void configure() {
            /* configuration implementation   */   }
```

Example 10: Flexible Access Control II

- Use declare error to define access policy

```
aspect FlagAccessViolation {
    pointcut factoryAccessViolation()
        : call(Product.new(..)) && !within(ProductFactory+);
    pointcut configuratorAccessViolation()
        : call(* Product.configure(..)) &&
          !within(ProductConfigurator+);
    declare error
                :  factoryAccessViolation() ||
                   configuratorAccessViolation()
                : "Access control violation";
    }
```

Summary: The Syntax

- Pointcuts
  - call, execution, within, withincode, cflow, cflowbelow
  - this, target, args, if
  - thisJoinPoint, thisJoinPointStaticPart
- Advices
  - before, after (throwing & returning), around (proceed)
- Aspects
  - Fields & methods, Abstract aspects & pointcuts
- Introductions
  - Add fields, methods and constructor
  - declare parents, declare soft
  - declare error, declare warning

Summary: The Examples

- Development Time Examples
  - 1,2: Tracing - Printing “Debug Messages”
  - 3: Contract enforcement
  - 4: Profiling with fine-grained control
  - 5: Pre- and post-conditions
  - 10: Flexible method access control
- Production Time Examples
  - 6: Resource pooling
  - 7: Logging (of errors)
  - 8: Modularizing functional guidelines
  - 9: Implementing Mixins: Making classes Cloneable

Summary

- AOP is a strong complement to OOD
  - Separation of concerns for unrelated aspects
  - Less code, more modular, easier to modify
  - Many practical uses, a lot of hype
- AspectJ is the primary implementation today
  - Many features, good tools and active support
  - Yet the entire platform is still in ‘beta version’
- A good tool, during development for now