“Is modularity the key principle to organizing software?”

**Learning outcomes**

- AOP Motivation and Weaving basics
- Bundling aspects with static crosscutting
- Join points, Pointcuts and Advice
- Composing Pointcut Designators
- Implementation of Advices and Pointcuts
Motivation

- Traditional modules directly correspond to code blocks
- Aspects can be thought of separately but are smeared over modules
  ~~~ Tangling of aspects
- Focus on *Aspects of Concern*

~~~ Aspect Oriented Programming

- Express a system’s aspects of concerns cross-cutting modules
- Automatically combine separate Aspects with a *Weaver* into a program

System Decomposition in Aspects

Example concerns:
- Security
- Logging
- Error Handling
- Validation
- Profiling
System Decomposition in Aspects

Example concerns:
- Security
- Logging
- Error Handling
- Validation
- Profiling

\[ \text{AspectJ} \]

Adding External Definitions

\begin{verbatim}
inter-type declaration
class Expr {}
class Const extends Expr {
    public int val;
    public Const(int val) {
        this.val=val;
    }
}
class Add extends Expr {
    public Expr l,r;
    public Add(Expr l, Expr r) {
        this.l=l;this.r=r;
    }
}
aspect ExprEval {
    abstract int Eval eval();
    int Const eval(){ return val; }
    int Add eval() { return l.eval() + r.eval(); }
}
equivalent code
// aspectj-patched code
abstract class Expr {
    abstract int eval();
}
class Const extends Expr {
    public int val;
    public int eval(){ return val; }
    public Const(int val) {
        this.val=val;
    }
}
class Add extends Expr {
    public Expr l,r;
    public int eval() { return l.eval() + r.eval(); }
    public Add(Expr l, Expr r) {
        this.l=l;this.r=r;
    }
}
\end{verbatim}
Join Points

Well defined points in the control flow of a program

- `method/constr. call` executing a statement, invoking a call
- `method/constr. execution` an individual method is invoked
- `field get` a field is read
- `field set` a field is set
- `exception handler execution` an exception handler is invoked
- `class initialization` static initializers are run
- `object initialization` dynamic initializers are run

Pointcuts and Designators

Definition (Pointcut)
A pointcut is a set of join points and optionally some of the runtime values when program execution reaches a referred join point.

Pointcut designators can be defined and named by the programmer:

\[
(\text{userdef}) := \text{'pointcut'} \langle \text{id} \rangle \left\langle \begin{array}{c}
\text{\textbackslash (idlist) } \Rightarrow \\text{\textbackslash (expr)} \\
\text{\textbackslash (id)} \left\langle \begin{array}{c}
\text{\textbackslash (idlist) } \Rightarrow \text{\textbackslash (expr)} \\
\text{\textbackslash (id)} \Rightarrow \text{\textbackslash (expr)} \\
\text{\textbackslash (expr) } \Rightarrow \text{\textbackslash (expr)} \\
\text{\textbackslash (primitive)}
\end{array}\right}\end{array}\right)
\]

Example:

```java
pointcut dfs() execution (void Tree dfs()) || execution (void Leaf dfs());
```

Advice

... are method-like constructs, used to define additional behaviour at joinpoints:

- `before(formal)`
- `after(formal)`
- `after(formal) returning (formal)`
- `after(formal) throwing (formal)`

For example:

```java
aspect Doubler {
    before(): call(int C.foo(int)) {
        System.out.println("About to call foo");
    }
}
```

Binding Pointcut Parameters in Advices

Certain pointcut primitives add dependencies on the context:

- `arg(arglist)`

This binds identifiers to parameter values for use in in advices.

```java
aspect Doubler {
    before(int i): call(int C.foo(int)) & arg(i) {
        i = i*2;
    }
}
```

arglist actually is a flexible expression:

\[
(\text{arglist}) := (\langle \text{arg} \rangle \left\langle \begin{array}{c}
\text{\textbackslash (identifier)} \\
\text{\textbackslash (typename)} \\
\text{\textbackslash (arg)} \\
\text{\textbackslash (arg)} \\
\text{\textbackslash (arg)} \\
\text{\textbackslash (arg)} \\
\text{\textbackslash (arg)}
\end{array}\right)\right)
\]

binds a value to this identifier
filters only this type
matches all types
matches several arguments
**Around Advice**

Unusual treatment is necessary for

- `type around(formal)`

⚠️ Here, we need to pinpoint, where the advice is wrapped around the join point – this is achieved via `proceed();`

```java
aspect Doubler {
    int around(int i): call(int C::foo(Object, int)) && args(i) {
        int newi = proceed(i+2);
        return newi/2;
    }
}
```

**Method Related Designators**

- `call(signature)`
- `execution(signature)`

Matches call/execution join points at which the method or constructor called matches the given `signature`. The syntax of a method constructor `signature` is:

```java
ResultType Name RecvrTypeName.method_id(ParamTypeName, ...)
NewObjectTypeName.new(ParamTypeName, ...)
```

**Method Related Designators**

```java
class MyClass{
    public String toString() {
        return "silly me ";
    }
    public static void main(String[] args){
        MyClass c = new MyClass();
        System.out.println(c.toString());
    }
}
```

```java
aspect CallAspect {
    pointcut calltoString(): call(MyClass.toString());
    pointcut exectoString(): execution(MyClass.toString());
    before(): calltoString() || exectoString() {
        System.out.println("advice!");
    }
}
```

```java
class MyClass{
    public String toString() {
        return "silly me ";
    }
    public static void main(String[] args){
        MyClass c = new MyClass();
        System.out.println(c.toString());
    }
}
```

```java
aspect CallAspect {
    pointcut calltoString(): call(MyClass.toString());
    pointcut exectoString(): execution(MyClass.toString());
    before(): calltoString() || exectoString() {
        System.out.println("advice!");
    }
}
```

```java
advice!
advice!
advice!
silly me silly me
```
Field Related Designators

- `get(fieldQualifier)`
- `set(fieldQualifier)`

Matches field get/set join points at which the field accessed matches the signature. The syntax of a field qualifier is:

```
FieldName::ObjectName::field_id
```

⚠️: However, set has an argument which is bound via `arg`:

```java
aspect GuardedSetter {
  before(int newval): set(static int MyClass.x) & args(newval) {
    if (Math.abs(newval - MyClass.x) > 100)
      throw new RuntimeException();
  }
}
```

Type based

- `target(typeId)`
- `within(typePattern)`
- `withIncode(methodPattern)`

Matches join points of any kind which

- are referring to the receiver of type `typeId`
- is contained in the class body of type `typePattern`
- is contained within the method defined by `methodPattern`

Flow and State Based

- `cflow(arbitrary_pointcut)`

Matches join points of any kind that occur strictly between entry and exit of each join point matched by `arbitrary_pointcut`.

- `if(boolean_expression)`

Picks join points based on a dynamic property:

```java
aspect GuardedSetter {
  before() if (thisJoinPoint.getKind().equals("call")) {
    System.out.println("What an inefficient way to match calls");
  }
}
```

Which advice is served first?

**Advices are defined in different aspects**

- If statement `declare precedence: A, B;` exists, then advice in aspect A has precedence over advice in aspect B for the same join point.
- Otherwise, if aspect A is a subsaspect of aspect B, then advice defined in A has precedence over advice defined in B.
- Otherwise, (i.e. if two pieces of advice are defined in two different aspects), it is `undefined` which one has precedence.

**Advices are defined in the same aspect**

- If either are `after advice`, then the one that appears `later` in the aspect has precedence over the one that appears earlier.
- Otherwise, then the one that appears `earlier` in the aspect has precedence over the one that appears later.
Implementation

Aspect Weaving:
- Pre-processor
- During compilation
- Post-compile-processor
- During Runtime in the Virtual Machine
- A combination of the above methods

Woven JVM Code

```
expr one = new Const(1);
one.val = 42;

aspect MyAspect {
    pointcut settingconst():
        set (int Const.val);
    before () : settingconst() {
        System.out.println("setter");
    }
}
```

Woven JVM Code

```
expr one = new Const(1);
expr e = new Add(one, one);
String s = e.toString();
System.out.println(s);

aspect MyAspect {
    pointcut callingtoString():
        call (String Object.toString())
        && target(Expr);
    before () : callingtoString() {
        System.out.println("calling");
    }
}
```

Woven JVM Code

```
aload_1
iconst_1
dup_x1
invokestatic #73 // Method MyAspect.aspectOf:()LMyAspect;
invokevirtual #79 // Method MyAspect.aspectAspect0f:()MyAspect;
putfield #54 // Field Const.val:
...
...
aload_2
instanceof #1 // class Expr
ifeq 85
invokestatic #67 // Method MyAspect.aspectOf:()MyAspect;
invokespecial #33 // Method java/lang/Object.toString:()Ljava/lang/String;
astore_3
```
Pointcut Parameters and Around/Proceed

Around clauses often refer to parameters and `proceed()`—sometimes across different contexts!

```java
class C {
    int foo(int i) { return 42+i; }
}

aspect Doubler {
    int around(int i): call(int *foo::Object, int) && args(i) {
        int newi = proceed(i+2)
        return newi/2;
    }
}
```

⚠️ Now, imagine code like:

```java
public static void main(String[] args){
    new C().foo(42);
}
```

Around/Proceed – via Procedures

✅ inlining advices in `main`—all of it in JVM, disassembled to equivalent:

```java
// aspectj patched code
public static void main(String[] args){
    C c = new C();
    foo_aroundBody1Advice(c, 42, Doubler.aspectOf(), 42, null);
}
private static final int foo_aroundBody0(C c, int i){
    return c.foo(i);
}
private static final int foo_aroundBody1Advice
    (C c, int i, Doubler d, int j, AroundClosure a){
    int temp = 2*i;
    int ret = foo_aroundBody0(c, temp);
    return ret / 2;
}
```

Escaping the Calling Context

⚠️ However, instead of being used for a direct call, `proceed()` and its parameters may escape the calling context:

⚠️ Use of `proceed()` should be avoided in favor of method parameters if feasible.
### Pointcut parameters and Scope

```java
class C {
    int foo(int i) { return 42 + i; }
    public static void main(String[] str) {
        new C().foo(42);
    }
}
```

```
class Doubl{:
    Executor executor;
    Future<Integer> f;
    int around(int i) call(int *foo(Object, int) & args(i) {
        Callable<Integer> c = () -> proceed(i*2)/2;
        f = executor.submit(c);
        return i/2;
    }
    public int getCachedValue() throws Exception {
        return f.get();
    }
}
```

### Shadow Classes and Closures

```java
// aspectj patched code
public static void main(String[] str) {
    int itemp = 42;
    Doublershadow = Doublers.aspectOf();
    Object[] params = new Object[]
    { new C(), Conversions.intObject(itemp) },
    C_AjcClosure1 closure = new C_AjcClosure1(closure);
    Shadow.Ajc$around$Doublers199558414$199558414(item $closure);
}
```

### Shadow Classes and Closures

```java
// aspectj patched code
public int ajc$around$Doublers199558414(int i, AroundClosure c){
    Callable<Integer> c = lambdas$0(i, c);
    f = executor.submit(c);
    return i/2;
}
```

### Property Based Crosscutting

```java
after(int i) call(void h()) &
   cflow(call(void f(int) &
      args(i)) { ... } );
```

#### Idea 1: Stack based
- At each call-match, check runtime stack for cflo-match
- Naive implementation
- Poor runtime performance

#### Idea 2: State based
- Keep separate stack of states
- Only modify stack at cflo-relevant pointcuts
- Check stack for emptiness

Even more optimizations in practice
- state-sharing, counters, static analysis
Implementation – Summary

Translation scheme implications:

- **before/after Advice** ... ranges from *inlined code* to distribution into several methods and closures
- **Joinpoints** ... in the original program that have advices may get *explicitly dispatching wrappers*
- **Dynamic dispatching** ... can require a *runtime test* to correctly interpret certain joinpoint designators
- **Flow sensitive pointcuts** ... runtime penalty for the naive implementation, optimized version still *costly*

Further reading...


Aspect Orientation

**Pro**
- Un-tangling of concerns
- Late extension across boundaries of hierarchies
- Aspects provide another level of abstraction

**Contra**
- Weaving generates runtime overhead
- nontransparent control flow and interactions between aspects
- Debugging and Development needs IDE Support