Programming Languages

Aspect Oriented Programming

Dr. Michael Petter
Winter 2019/20
“Is modularity the key principle to organizing software?“

Learning outcomes

1. AOP Motivation and Weaving basics
2. Bundling aspects with static crosscutting
3. Join points, Pointcuts and Advice
4. Composing Pointcut Designators
5. Implementation of Advices and Pointcuts
Motivation

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

$\rightsquigarrow$ **Aspect Oriented Programming**
Motivation

- Traditional modules directly correspond to code blocks
- Aspects can be thought of separately but are smeared over modules \(\leadsto\) Tangling of aspects
- Focus on Aspects of Concern

\(\leadsto\) Aspect Oriented Programming

Aspect Oriented Programming

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

Aspect oriented decomposition
System Decomposition in Aspects

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

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

⇝ AspectJ
Static Crosscutting
Adding External Definitions

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 Expr.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;
    }
}
Dynamic Crosscutting
Join Points

Well-defined points in the control flow of a program

- method/constr. call: executing the actual method-call statement
- method/constr. execution: the individual method is executed
- 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
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:

\[
\langle \text{userdef} \rangle ::= \text{pointcut} \ \langle \text{id} \rangle \ \langle \langle \text{idlist} \rangle \ ? \ \rangle \ : \ \langle \text{expr} \rangle \ ;
\]

\[
\langle \text{idlist} \rangle ::= \langle \text{id} \rangle \ (\ , \ \langle \text{id} \rangle)^
\]

\[
\langle \text{expr} \rangle ::= \ ! \ \langle \text{expr} \rangle \\
| \ \langle \text{expr} \rangle \ &\ & \ \langle \text{expr} \rangle \\
| \ \langle \text{expr} \rangle \ | \ | \ \langle \text{expr} \rangle \\
| \ \langle \text{expr} \rangle \ ? \\
| \ \langle \text{primitive} \rangle
\]

Example:

```plaintext
pointcut dfs(): execution (void Tree.dfs()) ||
                   execution (void Leaf.dfs())
```
... 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:

- `args(arglist)`

This binds identifiers to parameter values for use in advice.

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

`arglist` actually is a flexible expression:

```
⟨arglist⟩ ::= (⟨arg⟩ (‘,’ ⟨arg⟩)*)?
```

```
⟨arg⟩ ::= ⟨identifier⟩
       | ⟨typename⟩
       | ‘*’
       | ‘..’
```

- `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;
  }
}
```
Pointcut Designator Primitives
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:

\[
\text{ResultTypeName} \ \text{RecvrTypeName}.\text{meth\_id}(\text{ParamTypeName}, \ldots) \\
\text{NewObjectTypeName}.\text{new}(\text{ParamTypeName}, \ldots)
\]
class MyClass{
    public String toString() {
        return "silly me ";
    }
    public static void main(String[] args){
        MyClass c = new MyClass();
        System.out.println(c + c.toString());
    }
}

aspect CallAspect {
    pointcut calltostring() : call (String MyClass.toString());
    pointcut exectostring() : execution(String MyClass.toString());
    before() : calltostring() || exectostring() {
        System.out.println("advice!");
    }
}
class MyClass{
    public String toString() {
        return "silly me ";
    }
    public static void main(String[] args){
        MyClass c = new MyClass();
        System.out.println(c + c.toString());
    }
}

aspect CallAspect {
    pointcut calltostring() : call (String MyClass.toString());
    pointcut exectostring() : execution(String MyClass.toString());
    before() : calltostring() || exectostring() {
        System.out.println("advice!");
    }
}

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:

```
FieldTypeName ObjectTypeName.field_id
```

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

```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**(typeorid)
- within(typepattern)
- withincode(methodpattern)

Matches join points of any kind which
- are referring to the receiver of type typeorid
- 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(METHOD_CALL)) && within(MyClass) {
        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 subaspect 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
Implementation

Aspect Weaving:
- Pre-processor
- During compilation
- Post-compile-processor
- During Runtime in the Virtual Machine
- A combination of the above methods
aspect MyAspect {
    pointcut settingconst(): set(int Const.val);
    before () : settingconst() {
        System.out.println("setter");
    }
}
Woven JVM Code

```
aspect MyAspect {
    pointcut callingtostring():
        call (String Object.toString()) && target(Expr);
    before () : callingtostring() {
        System.out.println("calling");
    }
}
Expr one = new Const(1);
Expr e = new Add(one, one);
String s = e.toString();
System.out.println(s);
...
Poincut 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);
}
```
✓ 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;
}
```
However, instead of being used for a direct call, `proceed()` and its parameters may escape the calling context:
⚠️ `proceed()` might not even be in the same scope as the original method!
⚠️ even worse, the scope of the exposed parameters might have expired!

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

aspect Doubler {
    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

✓ creates a shadow, carrying the advice
✓ creates a closure, carrying the context/parameters

```java
// aspectj patched code
public static void main(String[] str){
    int itemp = 42;
    Doubler shadow = Doubler.aspectOf();
    Object[] params = new Object[]
    { new C(), Conversions.intObject(itemp) };
    C_AjcClosure1 closure = new C_AjcClosure1(params);
    shadow.ajc$around$Doubler$1$9158ff14(itemp, closure);
}
```
Shadow Classes and Closures

// aspectj patched code

class Doubler { // shadow class, holding the fields for the advice
    Future<Integer> f;
    ExecutorService executor;
    ...

    public int ajc$around$Doubler$1$9158ff14(int i, AroundClosure c){
        Callable<Integer> c = lambda$0(i,c);
        f = executor.submit(c);
        return i/2;
    }

    public static int ajc$around$Doubler$1$9158ff14proceed(int i, AroundClosure c)
        throws Throwable{
        Object[] params = new Object[] { Conversions.intObject(i) };
        return Conversions.intValue(c.run(params));
    }

    static Integer lambda$0(int i, AroundClosure c) throws Exception{
        return Integer.valueOf(ajc$around$Doubler$1$9158ff14proceed(i*2, c)/2);
    }
}

class C_AjcClosure1 extends AroundClosure{ // closure class for poincut params
    C_AjcClosure1(Object[] params){ super(params); }
    Object run(Object[] params) {
        C c = (C) params[0];
        int i = Conversions.intValue(params[1]);
        return Conversions.intObject(C.foo_aroundBody0(c, i));
    }
}
Property Based Crosscutting

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

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

Even more optimizations in practice
- state-sharing, counters,
- static analysis
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 *explicitely 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*
# 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
Further reading...


