Script generated by TTT

Title: Petter: Programmiersprachenh (23.01.2019)

Date: Wed Jan 23 14:11:23 CET 2019

Duration: 91:51 min

Pages: 32

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Aspect Oriented Programming

Dr. Michael Petter
Winter 2018/19

"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 seperately but are smeared over modules
  ⇝ *Tangling of aspects*
- Focus on *Aspects of Concern*

⇝ *A*spect *O*riented *P*rogramming

Motivation

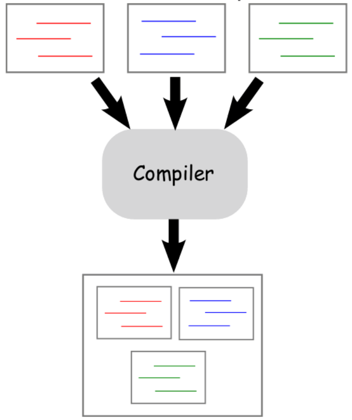
- Traditional modules directly correspond to code blocks
- Aspects can be thought of seperately but are smeared over modules
  ⇝ *Tangling of aspects*
- Focus on *Aspects of Concern*

⇝ *A*spect *O*riented *P*rogramming

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



System Decomposition in Aspects

Example concerns:

- Security
- Logging
- Error Handling
- Validation
- Profiling

Adding External Defintions

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

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 refered join point.

Pointcut designators can be defined and named by the programmer:

⟨*userdef*⟩ ::= ‘pointcut’ ⟨*id*⟩ ‘(’ ⟨*idlist*⟩? ‘)’ ‘:’ ⟨*expr*⟩ ‘;’

⟨*idlist*⟩ ::= ⟨*id*⟩ (‘,’ ⟨*id*⟩)*

⟨*expr*⟩ ::= ‘!’ ⟨*expr*⟩
 | ⟨*expr*⟩ ‘&&’ ⟨*expr*⟩
 | ⟨*expr*⟩ ‘||’ ⟨*expr*⟩
 | ‘(’ ⟨*expr*⟩ ‘)’
 | ⟨*primitive*⟩

Example:

```
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:

```
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 in advices.

```
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*⟩	binds a value to this identifier
| ⟨*typename*⟩	filters only this type
| ‘*’	matches all types
| ‘..’	matches several arguments

Around Advice

Unusual treatment is necessary for

- `type around(formal)`

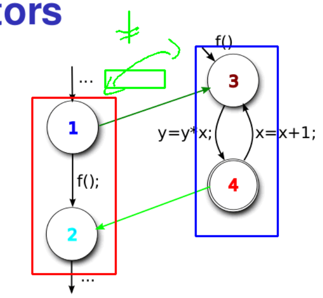
⚠ Here, we need to pinpoint, where the advice is wrapped around the join point – this is achieved via `proceed()`:

```
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:

```
ResultTypeName RecvrTypeName.meth_id(ParamTypeName, ...)
NewObjectTypeName.new(ParamTypeName, ...)
```


Method Related Designators

```
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!");
} }
```

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`:

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

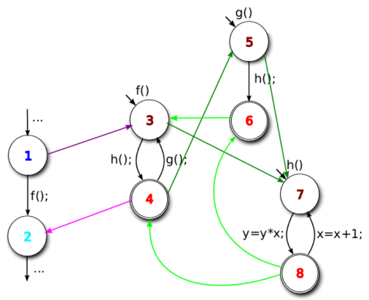

Type based

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

Matches join points of any kind which

- are refering 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:

```
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

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");
} }
```

```
...
 117: aload_1
 118: iconst_1
 119: dup_x1
 120: invokestatic  #73 // Method MyAspect.aspectOf:()LMyAspect;
 123: invokevirtual #79 // Method MyAspect.ajc$before$MyAspect$2$704a2754:()V
 126: putfield      #54 // Field Const.val:I
...
```


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");
} }
```

```
...
 72: aload_2
 73: instanceof    #1  // class Expr
 76: ifeq          85
 79: invokestatic  #67 // Method MyAspect.aspectOf:()MyAspect;
 82: invokevirtual #70 // Method MyAspect.ajc$before$MyAspect$1$4c1f7c11:()V
 85: aload_2
 86: invokevirtual #33 // Method java/lang/Object.toString:()Ljava/lang/String;
 89: astore_3
...
```


Poincut Parameters and Around/Proceed

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

```
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:

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

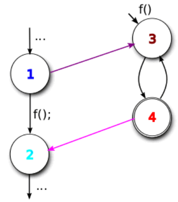
Around/Proceed – via Procedures

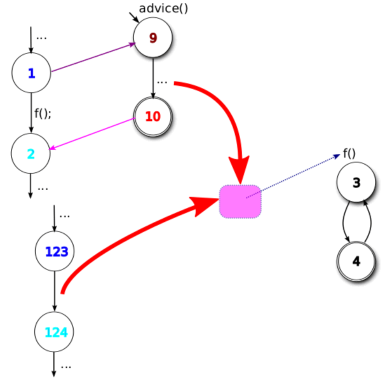
✓ inlining advices in `main` – all of it in JVM, disassembled to equivalent:

```
// 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 beeing used for a direct call, `proceed()` and its parameters may *escape the calling context*:





Pointcut parameters and Scope

⚠ `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

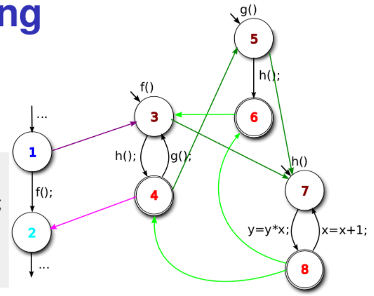
```
// aspectj patched code
public static void main(String[] str){
  int itemp = 42;
  Doubler shadow = Doubler.aspectOf();
  Object[] params = new Object[]
    { new C(),Conversions.intObject(itemp) };
  C_AjcClosure1 closure = new C_AjcClosure1(params);
  shadow.ajc$around$Doubler$1$9158ff14(itemp,closure);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

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

```
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 `cflow`-match
- ⇝ Naive implementation
- ⇝ Poor runtime performance

Idea 2: State based

- Keep seperate stack of states
- ⇝ Only modify stack at `cflow`-relevant pointcuts
- ⇝ Check stack for emptyness

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 *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...