Advanced AspectJ & AspectJ 5

Wim Vanderperren & Bruno De Fraine
wvdperre@vub.ac.be
Outline

* Aspect Instantiation (customization)
* Aspect Inheritance (reuse)
* AspectJ 5 for Java 5
  * Generics
  * Annotations
  * Annotation-based style
Aspect instantiation
Aspect Instantiation

public aspect Profiling {
    int count = 0;
    pointcut toProfile(): execution(public * *(..));
    after(): toProfile() {
        count++;
    }
}

* Similar to classes, aspects can have instance data
* Advice is executed in the context of an instance
* However, the instantiation is done by AspectJ: no new
Instantiation Strategies

- AspectJ has predefined instantiation strategies, selected with an aspect modifier

- Three main kinds of strategies:
  - Singleton: one, global aspect instance (default)
  - Per-object: one aspect instance per object
  - Per-control-flow: one aspect instance per complete join point
public aspect Profiling issingleton() {
    int count = 0;
    pointcut toProfile():
        execution(public * *(..));
    after(): toProfile() {
        count++;
    }
}

System.out.println(Profiling.aspectOf().count);

- Modifier may be omitted
- Base program can reach the global instance using: Profiling.aspectOf()
Per-Object Aspects (1)

* **Modifiers:** perthis(Pointcut) or pertarget(Pointcut)

* At each join point selected by the Pointcut, an aspect instance is set-up for advice execution

* One aspect instance is maintained per executing object (resp. receiving object)

* **Warning:** an advice is only applied at a join point if an aspect instance is set-up for that join point

* Advice pcs should be ‘included’ in modifier pc
public aspect Profiling perthis(toProfile()) {
    int count = 0;
    pointcut toProfile():
        execution(public * (Client || Server).*(..));
    after(): toProfile() {
        count++;
    }
}

System.out.println(
    {
        Profiling.aspectOf(client1).count,
        Profiling.aspectOf(server2).count
    });

* Modifier can use named pointcuts from aspect body
* Aspect instances reachable from base code using Profiling.aspectOf(obj)
Per-Control-Flow Aspects (1)

* **Modifiers:** `percflow(Pointcut)` or `percflowbelow(Pointcut)`

* At (resp. below) each join point selected by the Pointcut, an aspect instance is set-up

* The aspect instance is maintained during the entire control-flow of the join point

* Again: an advice is only applied at a join point if an aspect instance is set-up for that join point

* Advice pcs should lie within cflow of modifier pc
Per-Control-Flow Aspects (2)

```java
public aspect Profiling percflow(major()) {
    int count = 0;
    pointcut toProfile():
        execution(public * *(..));
    after(): toProfile() {
        count++;
    }
    pointcut major():
        execution(void Server.processRequest(..));
    after(): major() {
        System.out.println(count);
    }
}

* Aspect instances reachable from base code, but only within the relevant control flow, using Profiling.aspectOf()
```
AspectJ Instantiation: Evaluation

- Instantiation modifiers allows essentially the same advice to function very differently (good)
- Although explicit instantiation could be more powerful (bad)
- Not limited to predefined strategies
- Constructor can take configuration arguments
- No need for the `aspectOf` method, which only works when there is some global key
Aspect Inheritance and Reuse
Pointcut Reuse

- Named pointcuts are static members
- Can be referenced from other aspects (if access modifiers allow it)
- Common idiom: aspect that only defines pointcuts, to share among

```java
public aspect ProgramPoints {
  public pointcut main():
    execution(public * be.ac.vub..*.*(..));
  public pointcut expensive():
    execution(* (be.ac.vub.Calculator || be.ac.vub.Renderer).*(..));
}

public aspect Profiling {
  after(): ProgramPoints.expensive() { ... }
}
```
Advice Reuse

Reusable advice is defined using an abstract pointcut

```java
public abstract aspect Profiling {
    int count = 0;
    abstract pointcut toProfile();
    after(): toProfile() { count++; }
}

public aspect MyProfiling extends Profiling {
    pointcut toProfile():
        ProgramPoints.expensive();
}
```

* Reusable advice is defined using an abstract pointcut
* The advice is used by providing a definition for this pointcut
abstract Rules

- Can be declared abstract:
  - Named pointcuts and methods (no implementation)
  - Aspects (not instantiable, not applied / woven)
  - Subaspects can provide implementations for abstract members
  - When an aspect has abstract members left, it must itself be declared abstract
Inheritance Rules

* An aspect can implement one or more interfaces, or extend a class or another abstract aspect

* Still only argumentless constructor

* An aspect cannot extend a concrete aspect, classes cannot extend aspects

* Concrete aspect is directly instantiated (woven)

* Classes could be explicitly instantiated
AspectJ 5
AspectJ 5 (2005)

- AspectJ release with support for the new features of Java 5 (2004)
  - Generics
  - Annotations
  - (Variable arguments)
  - (Autoboxing)
Aside: Generics
Subtype Polymorphism

```java
output(o) {
    write(o.toString());
}
output(book);
output(person);
```

- "output" accepts different types
- These types share a contract
Parametric Polymorphism

choose(a,b) {
    if(cond.holds()) return a;
    else return b;
}
choose(book1,book2).read();
choose(pers1,pers2).marry();

* “choose” accepts parameters of any type
* Types of arguments and return value are related
interface Stringable {
    String toString();
}
void output(Stringable o) {...}

class Book implements Stringable {
    ...
}
output(book);

* Classes inherit from a common supertype
* Represents the contract
Typing Parametric PM

Object choose(Object a, Object b) {...}
((Book)choose(book1, book2)).read();

Object choose(Object o1, Object o2) {return anything;}
((Book)choose(pers, book)).read();

* General typing does not express type relations
* Requires casts
* Allows violations by callee and caller
Idea: Parametric Typing

<T> T choose(T a, T b){...}

<Book>choose(book1,book2).read();

* Type variable can be declared to capture the type relations
* Substitution of a type value
* No casts required
Java 5 Generics

interface List<E> {
    E get(int index);
    void add(int index, E element);
}

void process(List<String> l) {
    l.get(0).startsWith("foo"); //OK
    l.add(1, new Object()); //Err
}

* Type variables at the level of classes, bound per instantiation

* Generic containers: killer app for parametric polymorphism

* Type variables at the level of methods, bound per invocation

* (Mechanisms to interact with subtype polymorphism)

* Type variables and arguments are erased after compilation
Generics in AspectJ 5 (1)

Goal 1: Coping with Generics in the base language

- Type patterns can match generic types and their instantiations
  - (Quite complex because of erasure, see AspectJ 5 developer notebook for details)

- Inter-type declarations can introduce new generic members
abstract aspect GenericTracer<T> {  
    abstract pointcut trace(T t);  
    abstract void report(String d, T t);  
    before(T t): trace(t) { report("Entering",t); }  
    after (T t): trace(t) { report("Leaving",t); }  
}

aspect A extends GenericTracer<Server> {  
    pointcut trace(Server s):  
        execution(public Server.*(..)) && this(s);  
    void report(String d, Server s) {  
        System.out.println(d + "at" + s.getName());  
    }  
}

* Goal 2: Generic aspects  
* Aspects declare type variables  
* When used as type annotations, very similar to generic classes
Generics in AspectJ 5 (3)

interface Observer { void notify(); }

abstract aspect ObserverProtocol<Subject> { 
  Collection<Observer> Subject.observers;
  void Subject.notifyObservers() { 
    for(Observer o: observers) 
      o.notify(); 
  } 
  after(Subject s): this(s) && execution(void Subject.set*(..)) { s.notifyObservers(); } 
}

aspect MyObserverInstance extends ObserverProtocol<Server> { } // No body

* Type vars can also be used in the type patterns of pointcuts and ITDs

* New kind of aspect reuse, based on types

* Reuse of aspect behavior by simply providing type(s)
abstract / Inheritance Rule

* Caveat: type variables can only be declared for abstract aspects
* Reason: implicit instantiation
* Type variable of generic class is normally provided at instantiation
Annotations
Annotations

* Metadata (=structured extra information)
  * Added by the programmer to source tree nodes (classes, methods, fields,...)

* Predefined annotations understood by javac
  * e.g. @Override

* Other annotations can be inspected through reflection
Annotations

- **Good pointcut (observer protocol):**
  - `set(* Point.*)`

- **Fragile pointcut (caching):**
  - `call(String X.a()) || call(Object Y.b*(..)) || call(* C.*(*)) ..`

- **Solution: annotations**
  - `@Cachable`
Annotations to tag methods that could be cached

@Cachable(timeToLive=500)

pointcut enableCaching(Cachable c) : call(* *(..)) &&
    @annotation(c) && if(c.timeToLive()>treshold);
Annotations & Pointcuts

- Pointcuts can reason on the presence of annotations

- Examples:
  - `within(@Secure *)`
  - `call(@Cachable *.*(..))`
  - `handler(!@Catastrophic *)`
Annotations & Pointcuts

- Allows new style of aspect application based on semantic tags
- Solves possibly fragile pointcuts
  - e.g. execution(* a()) || execution (* b()) ..... becomes call(@Cachable *.*(..))
- Base code is no longer oblivious
- BUT base code is no longer oblivious!
Library Aspects

- AspectJ libraries with reusable aspects.
- Weg Isberg’s library of 30+ aspects
- Invariant Example: idempotent methods
  - Methods that should only run once
- Use annotations to tag them
  - Annotation defines semantic of method, not aspect specific
  - Developer takes into account this semantics
public abstract aspect IdempotentMethod pertarget(idempotentMethod()) {

    protected pointcut idempotentMethod();

    declare error : idempotentMethod() && execution(!void *(..)) :
    "Idempotent methods must return void";

    declare error : idempotentMethod() && execution(* *(*, ..)) :
    "Idempotent methods must not take an argument";

    boolean hit;
    void around() : idempotentMethod() {
        if (!hit) {
            proceed();
            hit = true;
        }
    }
}

static aspect IdempotentMethodTest extends IdempotentMethod {

    protected pointcut idempotentMethod() : execution(@Idempotent * *(..));
}

//Example method
public class MyBeanFactory {

    @Idempotent public void init() {....}

    ...
}
public aspect Const {
    declare error : withincode(@ReadOnly * *(..)) && set(* *)
        : "no set in read-only method";
    declare error : withincode(@ReadOnly * *(..)) && call(!@ReadOnly * *(..))
        : "no calls to non-read-only methods inside read-only method";
    declare error : set(@ReadOnly * *) && !withincode(new(..))
        : "no writes to read-only fields outside of constructors";
    declare error : within(@ReadOnly *) && call(!@ReadOnly * *(..))
        : "no calls to non-read-only methods inside read-only classes";
    declare error : within(@ReadOnly *) && set(* *)
        : "no sets inside read-only classes";
    // this is duplicative - omit or reinforce the point?
    declare error : within(@ReadOnly *) && execution(!@ReadOnly * *(..))
        : "all methods in read-only classes should be read-only";
}
Other invariants

- Re-throw VM error
- Naming conventions (e.g. fields start with "f")
- No calls from production into test packages
- Forces getter-setter methods
- No System.out or System.err
- No null parameters
- Enforce singleton pattern
Annotation-based development style
Annotation-based development style

- AspectJ aspects in pure Java
- Uses annotations to specify non-Java constructs
- Weaving can happen at load-time or at linking time

Advantages:
- A java compiler suffices
- Integrates better with existing toolchain/IDE
- There is no official new language
The following AspectJ pointcut:

```java
pointcut anyCall() : call(* *.*(..));
```

Translates to:

```java
@Pointcut("call(* *.*(..))") void anyCall() {}
```
// Pointcut arguments:
@Pointcut("call(* *(int)) && args(i) && target(callee)")
void someCall(int i, Foo callee) {}
//Simple Advice
@Before("execution(*.*(..))")
void anyExec() {
    logger.info("Something happened");
}

//Advice with arguments
@After("execution(*.*(..)) && target(myTarget)")
public void anyExec(Object myTarget) {
    logger.info("Something happened on " + myTarget);
}

//Advice with joinpoint reflection
@After("execution(*.*(..)) && target(myTarget)")
public void anyExec(JoinPoint thisJoinPoint, Object myTarget) {
    logger.info(thisJoinPoint.getSignature() + " happened on " + myTarget);
}
//Around Advice
@Around("execution(@Idempotent void *.*(..))")
public Object skipMethod(ProceedingJoinPoint thisJoinPoint) {
    Object result = null;
    if(!hasAlreadyExecuted) {
        hasAlreadyExecuted=true;
        result = thisJoinPoint.proceed();
    }
    return result;
}
//Pointcut-Advice Binding
@Pointcut("call(* *.*(..)) && @annotation(info) && if()")
protected static boolean cachableMethods(Cachable info) {
    return info.timeToLive() > treshold;
}

@Around("cachableMethods(info)")
public Object cache(ProceedingJoinPoint thisJoinPoint, Cachable info) {
    Object result = getFromCache(thisJoinPoint);
    if(result==null) {
        result = thisJoinPoint.proceed();
        storeInCache(thisJoinPoint, result, info.timeToLive());
    }
    return result;
}
//Declare Warning/Error
@DeclareWarning("call(* javax.sql..*(*..)) && !within(org.xyz.daos..*)")
static final String aMessage = "Only DAOs should be calling JDBC."

@DeclareError("execution(* IFoo+.*(*..)) && !within(org.foo..*)")
static final String badIFooImplementors = "Only foo types can implement IFoo"

//Declare Precedence
@DeclarePrecedence("Security*, org.xyz.TransactionSupport,
org.xyz.Persistence")
public class SystemArchitecture {
}

Inter-type declarations

Only supports injecting interfaces (aka mixins)

//Declare Marker Interface
@DeclareParents("org.xyz.domain..*")
Serializable implementedInterface;

//Declare Parents
public interface IDGen{ public long getId(); }
public class IDGenImpl implements IDGen { .... }

@DeclareParents(value="xy..*",defaultImpl=org.IDGenImpl.class)
private IDGen implementedInterface;

@Before("execution(* *.*(..)) && this(m)")
void feelingMoody(IdGen m) {
    System.out.println("this is my id "+m.getId());
}
Aspects

@Aspect
class MyLoggingAspect {
    .... }

// Controlling Instantiation
@Aspect("perthis(execution(* abc..*(..)))")
class MyLoggingAspect {}
Limitations

- Types have to be fully referenced in pointcuts
  - NOT: @Pointcut("call(* List.*(..))")
  - BUT: @Pointcut("call(* java.util.List.*(..))")

- Limited intertype declarations

- No privileged aspects
# Overview

<table>
<thead>
<tr>
<th>Element</th>
<th>Becomes</th>
<th>with Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aspect</td>
<td>Class</td>
<td>@Aspect</td>
</tr>
<tr>
<td>Named pointcut</td>
<td>Method</td>
<td>@Pointcut</td>
</tr>
<tr>
<td>Advice</td>
<td>Method</td>
<td>@Before, @After, @Around,...</td>
</tr>
<tr>
<td>Inter-type parent declaration</td>
<td>Field</td>
<td>@DeclareParents</td>
</tr>
<tr>
<td>Error/warning declaration</td>
<td>Static final String field</td>
<td>@DeclareWarning, @DeclareError</td>
</tr>
</tbody>
</table>