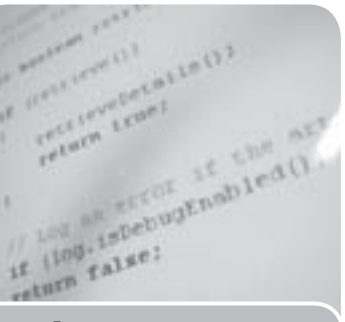


thema

Aspect Oriented Programming (AOP) definieert een nieuw programmeerconstruct, dat een aspect wordt genoemd, en dat gebruikt wordt om de aspecten van een software systeem die door het hele systeem gaan, in afzonderlijke programma-entiteiten op te delen. Dit artikel is opgedeeld in drie delen: het eerste deel legt de principes van AOP uit, het tweede introduceert AspectJ, een implementatie van de AOP-concepten in Java, en het derde deel vergelijkt de AOP-aanpak met programmeren op metaniveau.

Aspect-Oriented Programmeren in Java

Concepten ontleend aan metalevel programming

Objectgeoriënteerd programmeren is de laatste jaren gemeengoed geworden, en heeft vrijwel volledig de procedurele aanpak vervangen. Een van de grootste voordelen van OO is dat een software systeem gezien kan worden als een systeem dat opgebouwd is uit een verzameling van discrete (op zich staande, red.) klassen. Ieder van die klassen heeft een goed gedefinieerde taak, de verantwoordelijkheden zijn duidelijk vastgelegd. In een OO-applicatie werken die klassen samen om het algemene doel van de applicatie te bereiken. Er zijn echter delen van een systeem die niet gezien kunnen worden als de verantwoordelijkheid van een een klasse, maar zij komen door het hele systeem voor en betreffen delen van vele klassen. Locking in een gedistribueerde applicatie zou een voorbeeld kunnen zijn, exception handling, of het loggen van method calls. Natuurlijk kan de code die deze delen behandelt aan al die klassen afzonderlijk worden toegevoegd, maar dat gaat in tegen het principe dat iedere klasse duidelijk gedefinieerde verantwoordelijkheden heeft. Dat is waar AOP op het toneel komt. In AOP behouden de applicatieklassen hun duidelijk gedefinieerde verantwoordelijkheden. Daarnaast zorgt ieder aspect voor het gedrag dat door het hele systeem heen gaat.

De basis Laten we AOP introduceren met behulp van een voorbeeld. Stelt u zich een applicatie voor die voortdurend werkt met gedeelde data. De gedeelde data zouden in een **Data** object (een instance van de klasse **Data**) ingekapseld kunnen zijn. In deze applicatie zijn er meerdere objecten van verschillende klassen die op een enkel data-object werken, terwijl slechts één van deze objecten tegelijk toegang tot de data zou mogen hebben. Om het gewenste gedrag te bereiken, moet een of andere manier van locking worden geïntroduceerd. Dat betekent, dat steeds wanneer één van die objecten toegang wil tot de data, het **Data** object gelocked moet worden (en ge-unlocked nadat het object ermee klaar is). De traditionele aanpak is het introduceren van een (abstracte) basisklasse, waar alle werk-klassen van erven. Deze klasse definieert een methode **lock()** en een methode **unlock()** die aangeroepen moet worden voor- en nadat het eigenlijke werk gedaan is (semaphoren, in wezen). Deze aanpak heeft de volgende nadelen:

- Iedere methode die werkt op de data moet zich met locking bezig houden. De code staat vol met statements, die met de locking verband houden.
- In een single inheritance wereld is het niet altijd mogelijk alle werk-klassen te laten erven van een gemeenschappelijke basisklasse, omdat de enkele inheritance links al gebruikt zou kunnen zijn door een ander concept. Dit is vooral het geval wanneer de locking feature geïntroduceerd moet worden in een klasse hiërarchie nadat de hiërarchie ontworpen is, mogelijk door een andere ontwikkelaar (bijvoorbeeld de ontwikkelaars van een class library).
- Hergebruik komt in het gedrang: de werk-klassen zouden hergebruikt kunnen worden in een andere context waar ze geen locking nodig hebben (of waar ze een ander locking schema nodig hebben). Door de locking code in de werk-klassen op te nemen, zitten ze vast aan de locking aanpak die in die specifieke applicatie nodig is.

Het locking-concept in onze voorbeeldapplicatie kan met de volgende eigenschappen beschreven worden:

- Het is niet de primaire taak van de werk-klassen
- Het locking schema is onafhankelijk van de primaire job van de werkers
- Locking gaat dwars door het systeem heen, vele klassen en waarschijnlijke vele methoden van die klassen worden erdoor beïnvloed

Om met dit soort problemen om te gaan, gebruikt AOP een alternatieve aanpak: een nieuw programmeerconstruct zou gedefinieerd moeten worden dat rekening houdt met de door het gehele systeem werkende aspecten. Het is niet verbazingwekkend dat dit nieuwe programmeerconstruct een aspect genoemd wordt. In onze voorbeeldapplicatie zou het aspect **Lock** de volgende verantwoordelijkheden hebben:

- Het leveren van de noodzakelijke eigenschappen om objecten te locken en unlocken voor de klassen die moeten kunnen locken en unlocken (in ons voorbeeld het toevoegen van lock() en unlock() aan de Data klasse)
- Ervoor zorgen dat alle methodes die het Data object veranderen, vooraf lock() aanroepen en unlock() wanneer ze ermee klaar zijn (in ons voorbeeld werk-klassen).

Wat kunnen aspecten verder doen? Ze kunnen handig zijn, wanneer een software systeem bepaalde method calls moet loggen (bijvoorbeeld constructors die het aanmaken van een object moeten volgen). Ook hier is een traditionele methode noodzakelijk ((**log()**), en deze methode moet op bepaalde plaatsen in de code aangeroepen worden. Vanzelfsprekend wil niemand de inheritance link van compleet verschillende klassen verspillen om alleen maar een **log()** methode aan de klasse hiërarchie toe te voegen. AOP kan hier weer van dienst zijn, door een aspect te creëren dat een **log()** methode aanbiedt aan de klassen die er een nodig hebben, en door deze methode aan te roepen wanneer het maar nodig is. Een derde en vrij interessant voorbeeld is exception handling. Een aspect zou **catch()** clausules voor methoden van diverse klassen definiëren, waarmee een consistente exception handling voor de hele applicatie verzekerd zou zijn.

Wanneer je naar aspecten kijkt, rijzen er twee vragen. De eerste is: zijn aspects echt noodzakelijk? Natuurlijk niet, in die zin dat een bepaald probleem ook zonder aspects opgelost zou kunnen worden. Ze leveren echter een nieuw, hoger abstractieniveau dat het gemakkelijker zou kunnen maken software systemen te ontwerpen en te begrijpen. Naarmate software systemen groter en groter worden, wordt het "begrijpen" meer en meer een groot probleem. Daarom zouden aspects een waardevol hulpmiddel kunnen zijn.

Een tweede vraag zou kunnen zijn: gaan aspects de inkapseling van objecten niet tegen?" Ja, in zekere zin doen ze dat. Maar dat gebeurt op een gecontroleerde manier. Aspects hebben toegang tot het private gedeelte van de objecten waarmee ze geassocieerd zijn. Maar dat compromitteert de inkapseling tussen objecten van verschillende klassen niet.

ASPECTJ AOP is een concept en is als zodanig niet gebonden aan een bepaalde programmeertaal of een paradigma. Het kan helpen de tekortkomingen op te vangen van alle talen die enkel hiërarchische decompositie gebruiken. Dat kunnen procedurele, OO of functionele talen zijn. AOP is geïmplementeerd in vele talen waaronder Smalltalk en Java. De Java-implementatie van AOP wordt *AspectJ* genoemd en is ontwikkeld bij Xerox PARC. Zoals iedere andere aspect-geörienteerde compiler, bevat de AspectJ compiler een 'weef-fase' die ondubbelzinnig de cross-cutting tussen klassen en aspects oplost. AspectJ implementeert deze extra fase door eerst aspects te verweven met reguliere code en vervolgens de standaard Java compiler aan te roepen.

> AOP is een concept en is niet gebonden aan een programmeertaal of een paradigma

IMPLEMENTATIE VAN EEN VOORBEELD Ik zou graag een gedeelte van de implementatie tonen van het locking voorbeeld, zoals hiervoor beschreven. De klasse die de data representeert waarop het systeem draait, wordt **Data** genoemd. Het heeft een paar instance variabelen (waarvan sommige statisch zouden kunnen zijn) en een paar methoden die werken op de instance variabelen. Dan is er nog de klasse **Worker** die een soort van aanpassing van de data uitvoert, dat wil zeggen: deze modificaties kunnen tijd-getriggered zijn. Codevoorbeeld 11 toont deze **Worker** klasse. De teruggegeven boolean-waarden in de uitvoer-methoden geven aan of de aanpassingen succesvol zijn afgehandeld of niet.

```
01 aspect DataLog {
02       advise * Worker.performActionA(..), *
Worker.performActionB(..) {
03           static after {
```

```
04                if ( thisResult == true )
05                  System.out.println(
“Executed “+thisMethodName+
06                  “successfully!” );
07                else
08                  System.out.println( “Error
Executing “+
09                  thisMethodName );
10            }
11        }
12 }
```

Codevoorbeeld 1

Codevoorbeeld 1 definieert het aspect **DataLog**. Dit bevat een zogenaamd een zogenaamd advies dwars door het systeem heen. Het advies beïnvloedt alle methoden die overeenkomen met de handtekening * **performAction(..)**. Ze moeten bijvoorbeeld beginnen met performAction, of ze kunnen willekeurige parameters of return typen bevatten (regel 02). Nadat de methoden zijn uitgevoerd, wordt de code in het **after**-gedeelte in het **DataLog** advies uitgevoerd. Dus na iedere aanroep van **performActionA()** of **performActionB()** print het systeem een boodschap die aangeeft of de methode succesvol is uitgevoerd of niet. Binnen de aspect-code kunnen we een aantal speciale variabelen gebruiken, zoals **thisResult**, die het resultaat bevat van de geadviseerde method of **thisMethodName**, welke de naam van de methode bevat die op dat moment wordt uitgevoerd. Naast after adviezen, is het ook mogelijk om **before** adviezen toe te voegen. Met behulp van de code weaver, introduceert AspectJ automatisch de noodzakelijke code aan de corresponderende klassen.

> Aspects hebben toegang tot het private gedeelte van de objecten waarmee ze geassocieerd zijn

Nu gaan we kijken naar locking. Het is niet problematisch wanneer slechts één werker-klasse bezig is met **sharedDataInstance**, omdat locking niet nodig is wanneer slechts één werker-klasse werkt aan **sharedDataInstance**. Stel je nu **AnotherWorker** voor, een klasse die volledig los staat van **Worker** (i.e. geen gemeenschappelijke basisklasse), die aanpassingen uitvoert op dezelfde **sharedDataInstance**. Er moet dan een mechanisme worden geïntroduceerd om de **Data** instance te locken. Dit is noodzakelijk om te voorkomen dat beide werkerklassen tegelijkertijd aanpassingen verrichten aan **sharedDataInstance**, en daarmee een inconsistente status creëren of verkeerde data lezen. Ook hier voorzien aspects in een zeer elegante manier om dit probleem op te lossen. Alle code die nodig is voor locking is ingekapseld in een aspect.

```
01 aspect Lock {
02
03      Data sharedDataInstance;
04      Lock( Data d ) {
05          sharedDataInstance = d;
06      }
07
08      introduce Lock Data.lock;
09
10      advise Data() {
11          static after {
12              thisObject.lock = new Lock(
thisObject );
13          }
14      }
15
17 }
```

Codevoorbeeld 2

Codevoorbeeld 2 definieert een nieuw aspect genaamd **Lock**. In regel 08 wordt de nieuwe variabele lock geïntroduceerd in de **Data** klasse met behulp van een zogenaamde **introduce** cross-cut. Dan volgt een **advise** cross-cut: In regels 10 tot en met 14 wordt de constructor van de **Data** klasse aangepast, zodat na iedere aanroep de code binnen het advies wordt uitgevoerd. Dit creëert een nieuw **Lock** aspect voor ieder aangemaakt **Data** object. Zoals u kunt zien, kan een aspect zijn eigen status hebben (**sharedDataInstance**). De volgende stap is om de klassen die werken aan de **Data** instance (**Worker**, **AnotherWorker**) te adviseren. Hiervoor moet het **Lock** aspect worden uitgebreid zoals te zien is in Codevoorbeeld 3.

```
15      boolean locked = false;
16
17      advise Worker.perform*(..),
AnotherWorker.perform*(..) {
18          before {
19              if ( thisObject.sharedDataIn-
stace.lock.locked ) // enqueue, wait
20              thisObject.sharedDataInstace.
lock.locked = true;
21          }
22          after {
23              thisObject.sharedDataInstace.
lock.locked = false;
24          }
25      }
26 }
```

Codevoorbeeld 3

Dit introduceert de variabele **locked** in het **Lock** aspect. Het doel daarvan is om te onthouden of het geassocieerde **Data** object al dan niet gelocked is. Opnieuw worden de werk-klassen aangepast. Voordat zij gaan werken op de **Data** instance, locken ze het **Lock** object (regel 18 tot en met 20), en wanneer ze klaar zijn, unlocken ze het weer (regel 21 tot en met 23). Een alternatieve implementatie zou de gelockede variabele direct "introduceren" in het geassocieerde **Data** object.

Een derde voorbeeld regelt de foutafhandeling:

```
25      advise Worker.perform*(..),
AnotherWorker.perform*(..) {
26          static catch ( Exception ex ) {
ex.printStackTrace(); }
27          static finally { thisObject.
sharedDataInstace.lock.locked = false; }
28      }
29 }
```

Codevoorbeeld 4

Dit stukje code introduceert twee nieuwe adviezen, beiden voor de verschillende perform methoden. Het advies in regel 26 heeft als consequentie dat iedere exception die in de perform methode terecht komt, wordt afgedrukt. Regel 27 zorgt ervoor dat de lock onder alle omstandigheden wordt uitgevoerd.

De beschreven voorbeelden laten zien dat AOP interessante nieuwe concepten biedt voor objectgeoriënteerde ontwikkeling. Het volgende gedeelte toont de relatie tussen AOP en programmeren op metaniveau, een concept dat significante invloed heeft gehad op de ontwikkeling van het AOP paradigma.

PROGRAMMEREN OP METANIVEAU AOP heeft veel gemeenschappelijk met programmeren op metaniveau. Beiden leggen aspects door het hele systeem heen vast op een schone, gecontroleerde manier. Een van de meest fundamentele eigenschappen van programmeren op meta-niveau is dat de programmeur toegang heeft tot de structuren die het programma representeren: een programma dat in een bepaalde taal geschreven is, wordt in de runtime in exact dezelfde taal gerepresenteerd. De meest populaire taal die metalevel programming concepten implementeert is CLOS, the Common Lisp Object System [CLOS]. De implementaties daarvan zijn gebaseerd op het zogenaamde Metaobject Protocol (MOP). De MOP kan worden beschouwd als een standaard interface voor de CLOS interpreter [MOP]. Met behulp van MOP is het mogelijk om het gedrag van de interpreter op een gecontroleerde manier aan te passen. Het volgende gedeelte toont enige features van MOP in pseudo-Java syntax en laat zien hoe ze zich verhouden tot AOP.

BEFORE & AFTER CLOS bevat de feature 'methode combinatie'. Voor iedere methode is het mogelijk om een methode te definiëren die onmiddellijk wordt uitgevoerd voordat de primaire methode wordt uitgevoerd (de *before-method*), en een methode die wordt uitgevoerd na de primaire methode (de *after-method*). Deze methoden kunnen ook gedefinieerd worden in subklassen. Een voorbeeld daarvan is te zien in codevoorbeeld 5.

```
01 class Shape {
02      public void paint() {
03          // paint shape
04      }
05 }
06
07 class ColoredShape extends Shape {
08      public before void paint() {
09          // set brush color
10      }
11 }
12
13 class Test {
14      public void run() {
15          Shape shape = new Shape();
16          shape.paint();
17          Shape shape2 = new
ColoredShape();
18          shape2.paint();
19      }
20 }
```

Codevoorbeeld 5

In regel 16 roept de **Test** klasse de **paint()** method van een **Shape** object aan. Het resultaat is, dat **Shape.paint()** wordt aangeroepen. In regel 18, wanneer **shape2.paint()** wordt aangeroepen, wordt de before methode van de **ColoredShape** klasse aangeroepen voordat de **paint()** methode van de superklasse wordt aangeroepen. Het is interessant om te zien dat een after-methode geen invloed heeft op de return-waarde van de primaire methode. Wanneer een klasse meerdere niveaus van parents heeft, wordt de aanroep-semantiek van een methode als volgt:

> In AOP behouden de applicatieklassen hun duidelijk gedefinieerde verantwoordelijkheden

- Voer alle before-methoden uit van de klasse en van alle parent klassen
- Voer de primaire methode uit
- Voer alle after-methoden uit van de klasse en van alle parent klassen

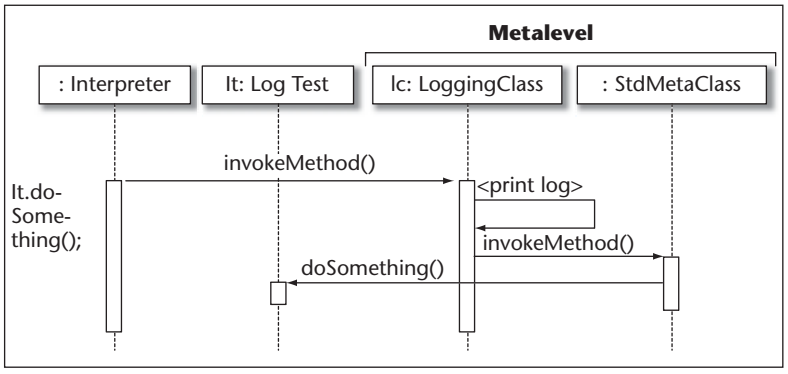
Vanwege de gegeven volgorde van uitvoer, worden before-methoden vaak gebruikt om fouten te checken (pre-conditie validatie) of om voorbereidend werk te doen, en after-methoden kunnen worden gebruikt om de omgeving te verduidelijken (en om post-condities te verzekeren). AOP introduceert dezelfde features: Ze worden aangeroepen voor en na adviezen.

METAKLASSEN Centraal in het meta-object protocol van CLOS staat het concept van de *metaklassen*. Een meta-klasse is de klasse van een klasse. Dat betekent dat een klasse die gedefinieerd is door de programmeur een instance is van een andere klasse: de meta-klasse van die klasse. Deze metaklasse is verantwoordelijk voor het implementeren van het protocol van de klasse: het aanroepmechanisme van de methode, het aanmaakproces voor het object, et cetera. Iedere klasse in een systeem heeft zijn eigen meta-klasse. Meta-klassen kunnen worden onderverdeeld om hun gedrag te definiëren, net als iedere andere klasse.

```
01 class Test extends Object {
02       public void doSomething() {
03            // do something
04       }
05 }
```

Codevoorbeeld 6

Laten we kijken naar het voorbeeld in codevoorbeeld 6, de **Test** klasse. Daarin is de methode **doSomething()** te zien. De metaklasse voor deze klasse is **StdMetaClass**, omdat geen andere metaklasse in de klasse declaratie gespecificeerd wordt. Zo heeft **Metaclass** (en dus ook zijn subklasse **StdMetaclass**) onder meer de methode **invokeMethod()**, welke wordt aangeroepen door de interpreter, iedere keer als een methode van een object wordt aangeroepen door een programma. De **invokeMethod()** methode is verantwoordelijk voor het implementeren van de semantiek van methode-aanroep, bijvoorbeeld het doorzoeken van de superklassen wanneer geen implementatie van de aangeroepen methode wordt gevonden in de klasse zelf. Stel dat u de aanroepen van alle methoden van bepaalde klassen wilt loggen. De aangewezen manier om dit te doen met behulp van MOP zou het volgende kunnen zijn: maak een nieuwe metaklasse aan die **StdMetaclass** uitbreidt en zet de **invokeMethod()** opzij om een log message te tonen. Codevoorbeeld 7 laat dit zien.



AFBEELDING 1. Iedere keer als doSomething() wordt aangeroepen, wordt een log message

```
01 public class LoggingClass extends
StdMetaclass {
02       public void invokeMethod( Object
dest, String name, Object[] params ) {
03            System.out.println( name+" cal-
led on "+dest+" with "+params );
04            super.invokeMethod( dest, name,
params );
05       }
06 }
```

Codevoorbeeld 7

Iedere klasse, die zijn method calls wil loggen moet nu **LoggingClass** gebruiken als zijn metaklasse (Codevoorbeeld 8). Een alternatieve implementatie zou de methode **invokeMethod()** niet opzij zetten maar zou een before-methode toevoegen aan **invokeMethod()**, die de log message laat zien.

```
01 public class LogTest extends Object meta-
class LoggingClass {
02       public void doSomething() {
03            // do something
04       }
05 }
```

Codevoorbeeld 8

Iedere keer als **doSomething()** wordt aangeroepen, zal een log message getoond worden op stdout. Dit proces wordt getoond in afbeelding 1. In AOP is het mogelijk dit gedrag te bereiken door het aanmaken van een aspect zoals dat in codevoorbeeld 9.

```
01 aspect Log {
02          // list of all classes that want
their method calls logged
03      advise Class1.*(..), Class2.*(..), ...
{
```

```
04          static before {
05              // display message
06          }
07      }
08 }
```

Codevoorbeeld 9

Een ander voorbeeld: we kunnen MOP gebruiken om klassen te implementeren, die tellen hoeveel instances er uit hen gemaakt zijn. Hiertoe moeten we een nieuwe metaklasse introduceren, **CountingClass**:

```
01 public class CountingClass extends
StdMetaclass {
02      private int instanceCount = 0;
03      public after Object createInstance(
Metaclass object ) {
04          instanceCount++;
05      }
06 }
07
08 public class CounterTest metaclass
CountingClass {
09 }
```

Codevoorbeeld 10

De nieuwe metaklasse (regels 01 t/m 06) voegt één attribuut toe, de integer **instanceCount** die gebruikt wordt om het aantal instances op te slaan. Dit werkt als volgt: iedere keer als een nieuwe klasse wordt gedefinieerd, die **CountingClass** specificeert als zijn metaklasse (bijvoorbeeld CounterTest), creëert deze definitie een nieuw metaobject van type **CountingClass**, waarbij zijn lid **instanceCount** geïnitialiseerd wordt naar 0. Wanneer

```
public class Worker extends Thread {
     Data sharedDataInstance;
     public boolean performAlgorithmA() {
          // ... do something with sharedDa-
taInstance
          // returns true when action was
executed
          // successfully, false otherwise.
     }
     public boolean performAlgorithmB() {
          // ... do something else with
sharedDataInstance
     }
     public void run() {
          // schedule calls of
          // performAlgorithmA and performAl-
gorithmB
          // according to some external
          // cicumstances
     }
}
```

Codevoorbeeld 11

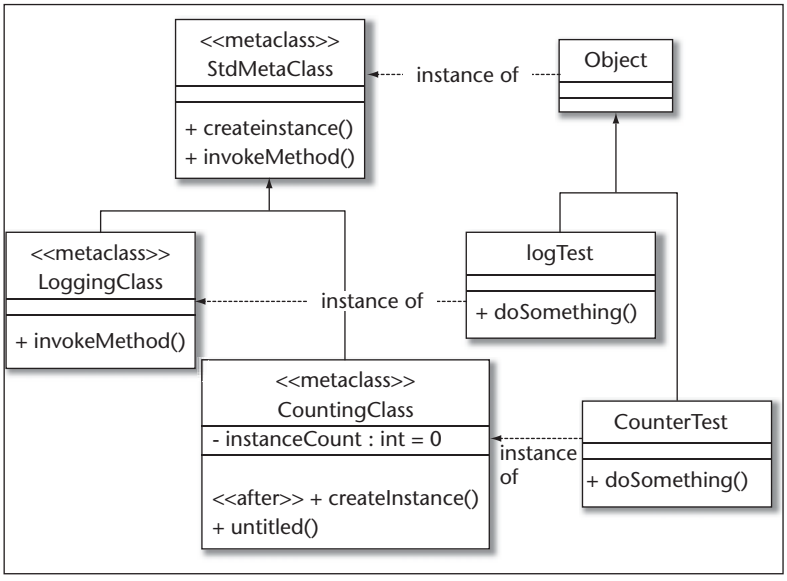
```
public class AnotherWorker extends Thread {
     Data sharedDataInstance;
     public boolean performA() {
          // ... do something with sharedDa-
taInstance
     }
     public void run() {
          // schedule calls of performA
     }
}
```

Codevoorbeeld 12

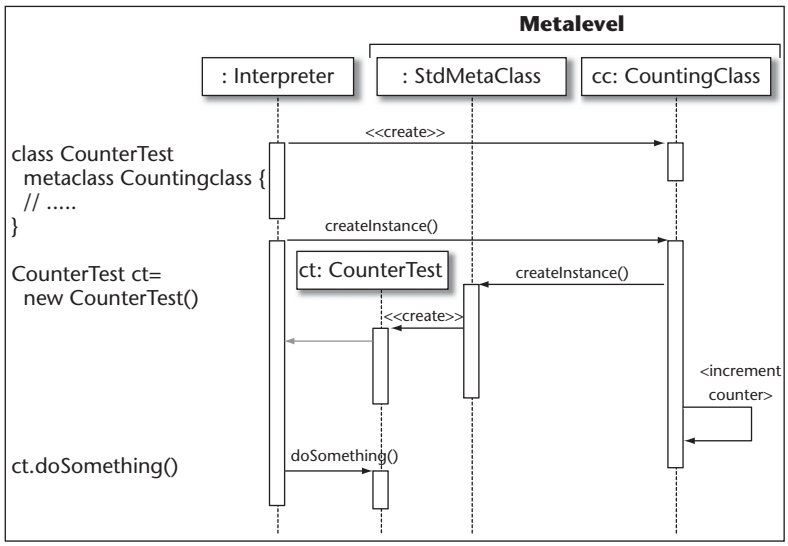
een nieuw object van type **CounterTest** wordt gemaakt (**CounterTest ct = new CounterTest()**) roept de interpreter de methode **createInstance()** van **CountingClass** aan. Omdat deze methode niet opzij gezet wordt in **CountingClass**, wordt de **createInstance()** methode van StdMetaclass uitgevoerd en wordt een instance van de **CounterTest** klasse gecreëerd.

Nadat we echter een after-methode gespecificeerd hebben voor **createInstance()** in **CountingClass**, wordt deze after methode nu uitgevoerd in aanvulling op de **createInstance()** methode zelf, en de teller wordt met één verhoogd. Afbeelding 2 toont de associaties tussen de klassen en de metaklassen en afbeelding 3 toont het aanmaakproces van een instance. In AOP kan dit gedrag bereikt worden door het aanmaken van een aspect, dat after-adviezen bevat voor de constructors van alle klassen die instance counting moeten implementeren.

SAMENVATTING AOP biedt veel interessante nieuwe concepten, waarvan er veel geleend zijn van metalevel programming. Uiteindelijk is AOP echter gemakkelijker te begrijpen dan metalevel programming. Hopelijk zal



AFBEELDING 2. De associaties tussen de klassen en de metaklassen



AFBEELDING 3. Het aanmaakproces van een instance

er een breder publiek voor ontstaan. Tot op heden zijn er geen industrie-compilers die AOP implementeren, maar de concepten voor het identificeren van gedrag door het hele systeem heen en voor het in afzonderlijke entiteiten stoppen van de code, kan helpen bij het doorgronden van de applicatie. Alleen al het je bewust zijn van het onderscheid tussen de feitelijke taak van een klasse en de aanvullende verantwoordelijkheden zoals locking of foutafhandeling, kan helpen bij het begrijpen van de structuur van de applicatie. Bovendien kunnen aspectachtige concepten handmatig worden geïmplementeerd in conventionele talen. Voor degenen die met de "echte AOP" willen experimenteren: de huidige bèta-versie van AspectJ, evenals informatie en tutorials over AOP zijn beschikbaar van de AOP homepage van het Xerox Palo Alto Research Center [AOP].

Referenties

[AOP] *AOP Homepage of Xerox PARC*, http://www.parc.xerox.com/aop

[AJ] *Aspect J Homepage*, http://www.parc.xerox.com/spl/projects/aop/aspectj/

[MOP] Kiczales, Rivieres, Bobrow: *The Art of the Metaobject Protocol*, MIT Press 1995, ISBN 0-262-61074-4

[CLOS] Koschmann, *The Common LISP Companion*, ISBN 0-471-50308-8

Markus Völter is onafhankelijk software engineer en architect (voelter@acm.org).

PATCHES Patches PATCHES Patches PATCHES Patches PATCHES

Optimal J en IntelliJ

Misschien geen zeer recent nieuws, maar niettemin belangrijk: Compuware's nieuwste OptimalJ-versie wordt in de Developer Edition samen met IntelliJ verkocht. Beide partijen (Compuware en JetBrains) verkopen de combinatie van de twee tools, die elkaar op een natuurlijke manier aanvullen. Ook is er nu ondermeer een Eclipse en een JBuilder plug in. In een volgend nummer komen we uitgebreider op vernieuwingen in OptimalJ terug. Iets uitvoeriger kunnen we nu ook al melden dat Compuware OptimalJ, door te voorzien in specifieke functionaliteit voor architecten, senior developers en developers, het voor elk van deze rollen mogelijk maakt een unieke bijdrage te leveren aan de ontwikkeling van een applicatie. Versie 3.2 verbetert de rol-gebaseerde structuur van OptimalJ, waardoor ieder teamlid productiever wordt en de samenwerking tussen teamleden vergemakkelijkt wordt. Daarbij springen de volgende features in het oog:

Om ontwikkelaars flexibiliteit te verschaffen in het gebruik van bekende tools, heeft Compuware Eclipse integratie toegevoegd aan de bestaande library van OptimalJ 3.2 Developer Edition's, bestaande uit IDE plug-ins, waaronder Borland JBuilder en IBM WebSphere Studio Application Developer. Om het organisaties mogelijk te maken bestaande Java applicaties te verbeteren en te integreren in OptimalJ ontwikkelprojecten, wordt de Developer Edition nu geleverd met native integratie met een van de meest populaire Java IDE's, JetBrains IntelliJ IDEA. Ook bevat de Developer Edition profiling mogelijkheden en architectonische analyse en refactoring functionaliteit. Compuware OptimalJ 3.2 maakt het voor architecten en senior developers ook mogelijk om het gedrag van een applicatie te modelleren. Door de mogelijkheden voor code-generatie uit te breiden van structurele naar gedragscode, kunnen OptimalJ 3.2 klanten automatisch een groter percentage van een applicatie genereren, de ontwikkelcycli versnellen, het aantal fouten verminderen en het onderhoud van de applicatie vergemakkelijken. Compuware OptimalJ 3.2's nieuwe model-merge mogelijkheden vereenvoudigen de samenwerking door grote enterprise development teams te voorzien van een visueel tool om de conflicten kunnen oplossen die kunnen optreden tussen concurrerende ontwikkelinspanningen. Versie 3.2 bevat ook een subsystem architectuur die het "subteams" mogelijk maakt om te werken aan verschillende delen van een groter ontwikkelproject. Door de toevoeging van IBM Rational ClearCase-ondersteuning aan de bestaande CVS integratie, verzekert Compuware OptimalJ 3.2 dat alle ontwikkelteams effectief hun source code assets kunnen beheren.