Andere paradigma's en concepten

Leereenheid 16

Andere paradigma's en concepten

INTRODUCTIE

In de cursus hebben we in detail de paradigma's imperatief, objectgeoriënteerd, functioneel, logisch en parallel bestudeerd. Naast deze 'traditionele' paradigma's zijn in de loop der tijd nieuwe paradigma's ontstaan. Vaak zijn het varianten op de netgenoemde paradigma's. Scripting, die in de vorige leereenheid aan de orde is geweest, is zo'n geval. Maar er zijn ook paradigma's ontstaan die zich duidelijk onderscheiden van de bestaande paradigma's. Als voorbeeld kunnen we het aspectgeoriënteerde paradigma noemen. In de eerste paragraaf van deze leereenheid bekijken we wat aspectgeoriënteerd programmeren inhoudt en bestuderen we de terminologie die in verband hiermee gebruikt wordt.

In deze leereenheid komen ook nieuwe concepten aan de orde. De taal Scala is een mooi voorbeeld van een programmeertaal waarin nieuwe concepten zijn toegepast. Scala is een multi-paradigmataal waarin bestaande concepten gebruikt worden om de taal zowel een object-georiënteerde als een functioneel karakter te geven en waarin nieuwe concepten gebruikt worden om de uitdrukkingsmogelijkheid van de taal te verhogen. Paragraaf 2 is een case study van de taal Scala. In deze pararaaf maakt u kennis met de taal en met nieuwe concepten die in de taal geïmplementeerd zijn. In Scala is het mogelijk om functioneel te programmeren. Ook dat bekijken we in deze paragraaf. Verder buigen we ons over het actor-model van Scala. Dat is een concurrency-model dat gebaseerd is op een gedistribueerd geheugen.

LEERDOELEN

Na het bestuderen van deze leereenheid wordt verwacht dat u

- een omschrijving kunt geven van het aspectgeoriënteerde programmeerparadigma
- de begrippen concern, cross-cutting concern en aspect kunt uitleggen
- kunt aangeven wat de rol is van de begrippen advice, joint point en pointcut in aspectgeoriënteerd programmeren
- een aantal voor- en nadelen van aspectgeoriënteerd programmeren kunt noemen
- een aantal kenmerken van de taal Scala kunt noemen
- een gegeven stukje code in Scala kunt interpreteren
- het verschil tussen de toegangsspecificaties val en var in Scala kunt verklaren
- de Scala-begrippen singleton object, companion object en companion klasse, case-klasse, trait en mixin compositie kunt uitleggen
- kunt aangeven welke concepten een rol spelen bij het functioneel programmeren in Scala en deze concepten in een gegeven stukje Scala-code kunt aanwijzen

CC BY NC ND

- het actor-model kunt beschrijven en kunt aangeven hoe het in Scala is geïmplementeerd
- de functionaliteit van een gegeven codefragment van een actor in Scala kunt omschrijven.

Studeeraanwijzingen
Deze leereenheid maakt geen gebruik van het tekstboek van Watt.

Deze leereenheid is niet bedoeld als programmeercursus in aspectgeoriënteerd programmeren en Scala. Deze leereenheid is bedoeld om u een idee te geven van wat deze twee vormen van programmeren inhouden. We geven, vaak zonder verdere uitleg, codevoorbeelden die u moet kunnen begrijpen en interpreteren. Er wordt van u niet verwacht dat u zelf code kunt schrijven.

De studielast van deze leereenheid is circa 5 uur.

LEERKERN

1 Aspectgeoriënteerd programmeren

Concern

Bij de analyse van de broncode van een softwaresysteem kunnen we verschillende onderdelen aanwijzen die een eigen verantwoordelijkheid dragen die niet ook door andere onderdelen in de code worden uitgevoerd. Voorbeelden zijn de uitvoer van business-logica, het regelen van transacties of het afhandelen van exceptions. Meestal zijn deze verantwoordelijkheden geïmplementeerd in één klasse of module. In termen van aspectgeoriënteerd programmeren (AOP) zijn dat *concerns* (probleemgebieden). De business-logica is een kernverantwoordelijkheid terwijl andere concerns (zoals logging) aspecten (aspects) representeren die in interactie zijn met de kernverantwoordelijkheid. Vaak worden deze aspecten *cross-cutting concerns* genoemd omdat ze 'haaks' staan op de kernverantwoordelijkheid.

Cross-cutting concern

Aspectgeoriënteerd programmeren

Aspectgeoriënteerd programmeren is een paradigma die bedoeld is om met cross-cutting aspecten om te gaan. AOP probeert in een systeem de cross-cutting concerns van de concerns te scheiden en te representeren in aparte beheersbare codemodulen. De winst hiervan is dat deze cross-cutting concerns nu maar op één plek zijn gedefinieerd in een systeem.

Studeeraanwijzing

Lees het artikel uit de bijlage: A look at aspect-oriented programming, Gary Police.

We geven hier een definitie van de belangrijkste concepten die gebruikt worden in verband met AOP. De Engelse termen blijven hier onvertaald.

Aspect

Een *aspect* is een abstractie van programmacode die een cross-cutting concern definieert, dat wil zeggen een verantwoordelijkheid die in verschillende delen van het softwaresysteem moet voorkomen. Een aspect bevat de definitie van pointcuts en de advice die geassocieerd is met de concern.

Advice

Een *advice* is de code die een concern implementeert.

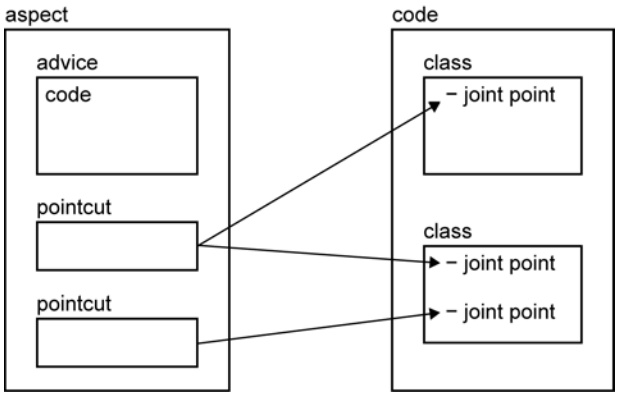
Joint point

Een *joint point* is een punt in een programma in uitvoering waar de advice geassocieerd met een aspect uitgevoerd zou moeten worden.

Pointcut

Een *pointcut* is een opdracht binnen een aspect die aangeeft welke joint points aan de aspect gekoppeld moeten worden.

Figuur 16.1 visualiseert de betekenis van deze concepten.



FIGUUR 16.1 Aspects, advice, pointcuts en joint points

OPGAVE 16.1
a Welke cross-cutting concern dient in het artikel als voorbeeld?
b Noem een ander voorbeeld van een cross-cutting concern die met behulp van AOP goed zou kunnen worden geprogrammeerd?

OPGAVE 16.2
Wat zijn de drie verschillende typen advices en waarom is deze differentiatie nodig?

OPGAVE 16.3
Wat gebeurt er met de aspecten tijdens compilatie van de code?

OPGAVE 16.4
Waarom is unit-testen lastiger bij AOP?

AOP is een paradigma dat sterk van het bestaande afwijkt en een element van reflectie en metaprogrammeren inbrengt. Meer nog dan door het directe gebruik van een AOP-taal, oefent AOP invloed uit doordat allerlei frameworks en programmeeromgevingen met AOP zijn gebouwd. We geven een paar voorbeelden. Enterprise Javabeans zijn vanaf versie 3.0 door AOP enorm vereenvoudigd. Het Java-framework Spring biedt AOP en is behoorlijk populair. Ten slotte noemen we de annotaties die in Java 5 werden ingevoerd. Annotaties zijn speciaal bedoeld als aangrijpingspunten voor aspecten.

cc BY NC ND

2 Case study: Scala, een multi-paradigmataal

Multi-paradigma

Een *multi-paradigmaprogrammeertaal* is een programmeertaal die meer dan één programmeerparadigma ondersteunt of in ieder geval concepten uit verschillende paradigma's implementeert. Op de Engelstalige versie van Wikipedia wordt een zeer uitgebreide en genuanceerde lijst gegeven van paradigma's en van talen waarin deze paradigma's terug te vinden zijn; zie de link op de cursussite.
In deze paragraaf bestuderen we de programmeertaal Scala. Volgens de lijst van Wikipedia implementeert Scala zeven verschillende paradigma's. Zo uitgebreid willen we het hier niet aanpakken. Wij beschouwen de taal Scala als een objectgeoriënteerde taal en een functionele taal.

2.1 INTRODUCTIE TOT DE PROGRAMMEERTAAL SCALA

Scala is een taal die zowel functioneel als objectgeoriënteerd programmeren ondersteunt. De taal is ontwikkeld door Martin Odersky. De eerste versie is in 2003 verschenen. De naam Scala is afgeleid van 'Scalable Language' vanwege de vele mogelijkheden tot abstractie en modularisatie.

OO-taal

Scala is een *objectgeoriënteerde taal* in de zin dat elke waarde een object is. Scala kent geen primitieve waarden. De functionele eigenschappen van de taal worden in de volgende paragraaf bekeken.

JVM

De compiler van Scala vertaalt de broncode naar bytecode die op een *Java Virtuele Machine* draait. Daardoor heeft de taal Scala de beschikking over alle bibliotheken van Java.

Statisch getypeerd

Scala is een *statisch getypeerde* taal maar het type van de elementen hoeft niet altijd gegeven te worden. De compiler is in staat om door type-inferentie het type uit de context te extraheren.

Syntaxis

Er zijn *syntactische* verschillen met Java. Behalve het weglaten van typering en van 'onnodige' puntkomma's kunnen underscores gebruikt worden als wildcard in import-opdrachten (Java gebruikt hiervoor het sterretje). Het sleutelwoord override is verplicht bij herdefinitie van methoden. Parametertypen staan achter de parameters. Het sleutelwoord def markeert een methodedefinitie waarvan de implementatie uit één expressie of één blok bestaat. Meerdere expressies kunnen gegroepeerd worden door blokaccolades. In dat geval geldt de laatste expressie binnen het blok als terugkeerwaarde. Een expliciete return-opdracht is niet nodig. Methoden met een terugkeerwaarde van type Unit (het tegenhanger van void in Java) worden alleen uitgevoerd voor de neveneffecten. In de code kan het type Unit weggelaten worden.

We geven in figuur 16.2 ter illustratie en zonder verdere uitleg een voorbeeld in Scala afkomstig van de officiële site van Scala, scala-lang.org. Daarna bespreken we een paar nieuwe concepten uit de taal.

Voorbeeld

```
package examples

/** Quick sort, imperative style */

object sort {

  /** Nested methods can use and even update everything
   *  visible in their scope (including local variables or
   *  arguments of enclosing methods).
   */
  def sort(a: Array[Int]) {

    def swap(i: Int, j: Int) {
      val t = a(i); a(i) = a(j); a(j) = t
    }

    def sort1(l: Int, r: Int) {
      val pivot = a((l + r) / 2)
      var i = l
      var j = r
      while (i <= j) {
        while (a(i) < pivot) i += 1
        while (a(j) > pivot) j -= 1
        if (i <= j) {
          swap(i, j)
          i += 1
          j -= 1
        }
      }
      if (l < j) sort1(l, j)
      if (j < r) sort1(i, r)
    }

    if (a.length > 0)
      sort1(0, a.length - 1)
  }

  def println(ar: Array[Int]) {
    def print1 = {
      def iter(i: Int): String =
        ar(i)
        + (if (i < ar.length-1) ","
        + iter(i+1) else "")
      if (ar.length == 0) "" else iter(0)
    }
    Console.println("[" + print1 + "]")
  }

  def main(args: Array[String]) {
    val ar = Array(6, 2, 8, 5, 1)
    println(ar)
    sort(ar)
    println(ar)
  }
}
```

FIGUUR 16.2 Een voorbeeld in Scala

cc BY NC ND

OPGAVE 16.5
Bekijk de code van methoden swap en iter in figuur 16.2.
a Welke informatie is bevat in de kop van methode swap?
b Hoeveel parameters heeft methode iter en van welk type zijn ze?
c Wat is het type van de terugkeerwaarde van methode iter en welke waarde wordt teruggegeven?

Sleutelwoorden val en var

Variabelen kunnen worden gedeclareerd met het sleutelwoord val of var. Met de toegangsspecificatie var wordt een variabele gedeclareerd zoals in Java, dus een veranderbare variabele die als neveneffect heeft dat het toestand van het geheugen gewijzigd kan worden. Een val-declaratie levert een onveranderbare (immutable) variabele op, dus een variabele zonder neveneffecten. Dit komt overeen met een variabele in Java die de aanduiding final heeft.

Singleton object

Een klasse in Scala kan geen static-elementen bevatten. In plaats daarvan gebruikt Scala *singleton-objecten* die met het sleutelwoord object gedeclareerd worden; zie figuur 16.2. Een singleton-object kan niet geïnstantieerd worden.

Companion object

Wanneer een singleton-object dezelfde naam heeft als een Scala-klasse, dan is er sprake van een *companion object*. De klasse is dan de companion klasse. Beide moeten dan in hetzelfde bronbestand worden gedefinieerd. Een companion klasse en een companion object hebben toegang tot elkaars private attributen en methoden. De reden voor deze keuze voor companion objecten in plaats van klasseattributen en klassemethoden, is dat in Scala alles een object is: Scala is consequent objectgeoriënteerd. Er bestaat at run-time niet zoiets als een klasse: er bestaan alleen objecten.

Case-klasse

Een *case-klasse* is een gewone klasse die de parameters van haar constructor exporteert in de kop van de klasse.

```
abstract class Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
case class App(f: Term, v: Term) extends Term
```

FIGUUR 16.3 Case-klassen

De klasse Var, Fun en App zijn drie case klassen.
De compiler voegt automatisch het volgende toe aan een case-klasse:
– Een methode met als naam de naam van de klasse (een constructor-functie) om hiermee instanties te creëren. Voorbeeld: `val v = Var("x")`
– Alle parameters worden attributen van de klasse met de toegangs-specificatie val. Het zijn dus onveranderbare waarden. Voor elk gegenereerd attribuut komt een get-methode.
– De klasse krijgt automatisch een implementatie van de methoden toString, hashCode en equals.

Case-klassen kunnen in tegenstelling tot normale klassen gebruikt worden bij patroonherkenning. De expressie Fun("f", t) correspondeert met alle instanties van type Fun waarvan de eerste argument de string "f" is en de tweede argument de variabele t is.

Naast case-klassen zijn er ook case-objecten.

OPGAVE 16.6
Welk structureringsconcept is van toepassing in de code van figuur 16.3?

Trait

Naast klassen kent Scala *traits*. Een trait is vergelijkbaar met een interface in Java, maar kan in tegenstelling tot interfaces wel attributen en implementatie van methoden bevatten. Een trait kan geen constructoren met parameters hebben.
Traits zijn ook verwant met abstracte klassen. Het verschil is dat abstracte klassen wel een constructor met parameters kunnen hebben.

Mixin compositie

Een trait kan met behulp van *mixin compositie* gecombineerd worden met een klasse. Dat gebeurt met een van de twee sleutelwoorden extends of with.
Mixin compositie is een vorm van meervoudige overerving, maar is niet hetzelfde. Bij een trait is het type van de referentie naar super (het type van de superklasse dus) niet bekend op het moment dat de trait gedefinieerd wordt. Het type van de superklasse wordt telkens bepaald wanneer een trait gemengd wordt met een klasse of een andere trait.

We laten dat met een codevoorbeeld zien:

Voorbeeld

```
trait Philosophical {
  def philosophize() {...}
}

trait HasLegs {
   ...
}

class Animal {
  ...
}

class Frog extends Animal with HasLegs with Philosophical {
  override def toString = "green"
}
```

De klasse Frog erft van de klasse Animal en van de traits HasLegs en Philosophical. Hierbij erft de klasse onder andere de implementatie van de methode philosophize van trait Philosophical.

Stel, een methode komt met dezelfde naam voor in twee of meer traits of klassen en de gegeven implementaties van de methode verschillen. Welke implementatie zou dan gebruikt worden? Dat hangt af van de volgorde waarin de traits worden genoemd: de trait die het laatst genoemd wordt, overschrijft eventuele methoden van de traits en klassen ervoor. In ons voorbeeld maakt het niet uit of methode philosophize ook gedefinieerd is in Animal of in Haslegs of in beide. De implementatie is die van Philosophical.

Het is ook mogelijk om tijdens het instantiëren van een klasse traits te mengen. We geven een voorbeeld van deze vorm van dynamische mixin compositie:

```
new String("tekst") with RichIterator[Char]
```

CC BY NC ND

Dit geeft een instantie van de klasse String met als extra de attributen en de methoden gedefinieerd in de trait RichIterator. Dit voorbeeld illustreert de kracht van deze vorm van late binding.

2.2 FUNCTIONEEL PROGRAMMEREN IN SCALA

De twee belangrijkste kenmerken van het functionele paradigma zijn onveranderbare variabelen en functies als eerste-klaswaarden en zonder neveneffecten. Daarbij komen ook de concepten hogere-orde functies, anonieme functies, currying en partiële parametrisatie, pattern matching, lazy evaluatie, closures en recursie als eerste middel voor iteratie. In deze paragraaf bekijken we in welke vorm deze concepten aanwezig zijn in Scala en hoe we hiermee functioneel kunnen programmeren in deze taal.

Geen neveneffecten

Hoe kunnen we in Scala code *zonder neveneffecten* programmeren, dat wil zeggen code die de toestand van het geheugen niet wijzigt?
Ten eerste moeten we ervoor zorgen dat de variabelen onveranderbaar zijn. Dat zijn in Scala variabelen met de aanduiding val.
Ten tweede moeten samengestelde waarden ook onveranderbaar zijn. De klasse List (een automatische import van scala.collection.immutable.List) en de andere datastructuren van de package scala.collection.immutable zijn onveranderbaar. De bekende methoden als het toevoegen, weghalen of updaten retourneren telkens een nieuw object.
Ten derde moeten we methoden definiëren die een waarde teruggeven. Een terugkeerwaarde van type Unit is niet goed omdat een dergelijke methode altijd neveneffecten impliceert. Die willen we juist vermijden.
Ten slotte mogen geen lussen geprogrammeerd worden met behulp van de gebruikelijke for- en while-lussen waarin gebruik wordt gemaakt van een veranderbare variabele voor het bijhouden van de iteraties. In plaats daarvan moeten we recursie en *lijstcomprehensies* inzetten. In het volgende voorbeeld wordt in methode even met behulp van comprehensie een lijst van even getallen opgebouwd:

Lijstcomprehensie

```
def even(from: Int, to: Int): List[Int] =
  for (i <- List.range(from, to) if i % 2 == 0) yield i
```

Comprehensie heeft de vorm

```
for (enum) yield e
```

waar enum een enumerator is. Een comprehensie evalueert de expressie e voor elke binding die door de enumerator is gegenereerd en retourneert een lijst met alle verkregen waarden.

OPGAVE 16.7
Wat is het resultaat van de aanroep even(0, 10)?

Functies als eerste-klaswaarden

Functieobject

Hoe kunnen we in Scala functies gebruiken als *eerste-klaswaarden*? We kunnen functies gebruiken als gewone objecten. De volgende opdracht declareert een onveranderbare variabele add dat een *functieobject* is met twee parameters van type Int en die als resultaat de som van de twee parameters heeft:

```
val add = (x: Int, y: Int) => x + y
```

Een anonieme versie van dezelfde functie is:

```
(x: Int, y: Int) => x + y
```

Dergelijke functies kunnen meegegeven worden als argument of kunnen het resultaat zijn van een functie- of methodeaanroep.

Currying

Currying is bij methodedefinitie mogelijk. In plaats van alle parameters in een structuur te bundelen en in één keer aan de methode aan te bieden (zoals bij functie add), kunnen de parameters ook los aan de functie aangeboden worden:

```
def add1(x: Int) = (y: Int) => x + y
```

of

```
def add2(x: Int)(y: Int) = x + y
```

Partiële parametrisatie

Partiële parametrisatie kan bereikt worden door een parameter te vervangen door een underscore. In het volgende voorbeeld wordt de methode sum gedefinieerd en daarna met behulp van partiële parametrisatie de functie sum':

```
def sum(a: Int, b: Int, c: Int) = a + b + c

val sum'= sum(1, _:Int, 3)
```

De functie sum' is een functieobject die één parameter van type Int verwacht en een resultaat van type Int heeft.

In het geval van een gecurryde methodedefinitie (zie de definitie van sum1 en sum2 hierboven) kan partiële parametrisatie als volgt worden bereikt:

```
val f1 = add1(5)
```

of

```
val f2 = add2(5) _
```

Let op het verschil in syntaxis bij de twee varianten: bij de definitie van f1 mag geen underscore staan terwijl het bij de definitie van f2 wel moet.

Hogere-ordefuncties

In Scala kunnen we ook *hogere-ordefuncties* definiëren. We geven hier een voorbeeld:

```
def apply(f: Int => String, v: Int) = f(v)
```

De hogere-ordefunctie apply heeft twee parameters waarvan de eerste een functie van Int naar String is.

Patroonherkenning

Scala heeft een ingebouwd mechanisme voor *patroonherkenning*. Ook hier volstaan we met het geven van een voorbeeld:

```
def second(xs : List[Int]) = xs match {
  case x :: y :: _ => y
  case _ => null
}
```

Met gebruik van het sleutelwoord match kunnen we een lijst van alternatieven geven. De eerstgevonden match bepaalt de waarde. In het voorbeeld ziet u dat de operator om een waarde op kop van een lijst te zetten de dubbele dubbelepunt (`::`) is. In de eerste case-opdracht wordt de lijst beschreven met behulp van patroonherkenning. De underscore heeft hier de betekenis van 'don't care'. Het gaat om een lijst met op kop x, dan y en dan de rest van de lijst. Oftewel, elke lijst die uit minstens twee elementen bestaat.

Closure

Scala kent het concept *closure*. De waarde van een functieobject wordt tijdens verwerking bepaald en vormt een closure die de binding van de vrije variabelen bevat. In het volgende voorbeeld is more een vrije variabele:

```
(x: Int) => x + more
```

Om de anonieme functie te kunnen evalueren voor een gegeven argument is de waarde van more ook nodig. Daarom wordt deze waarde tijdens verwerking toegevoegd aan de closure van de functie. Wanneer variabele more van een waarde is voorzien, is de anonieme functie een gesloten term (closed term) geworden.

Lazy evaluatie

Verder vermelden we zonder verdere uitleg dat *lazy evaluatie* in Scala mogelijk is met behulp van het sleutelwoord lazy.

In deze paragraaf hebben een aantal functionele concepten van Scala de revue laten passeren. Er zijn natuurlijk nog meer mogelijkheden voor het functioneel programmeren in Scala, maar die vallen buiten het bestek van deze cursus.
We eindigen de paragraaf met een voorbeeld van het quicksort algoritme op functionele wijze; zie figuur 16.4. Het voorbeeld is, net als het voorbeeld van figuur 16.2 afkomstig van de officiële site van Scala, scala-lang.org.

Voorbeeld

```
package examples

/** Quick sort, functional style */

object sort1 {
  def sort(a: List[Int]): List[Int] = {
    if (a.length < 2)
      a
    else {
      val pivot = a(a.length / 2)
      sort(a.filter(_ < pivot)) :::
           a.filter(_ == pivot) :::
           sort(a.filter(_ > pivot))
    }
  }
```

```
  def main(args: Array[String]) {
    val xs = List(6, 2, 8, 5, 1)
    println(xs)
    println(sort(xs))
  }
}
```

FIGUUR 16.4 Quicksort functioneel

OPGAVE 16.8
Vergelijk de implementaties van quicksort uit de figuren 16.2 en 16.4. Waarom is de implementatie van figuur 16.2 niet functioneel terwijl die van figuur 16.4 dat wel is?
Aanwijzingen:
– Methode filter wordt aangeroepen op een lijst en heeft als terugkeerwaarde een nieuwe lijst die alle elementen bevat die voldoen aan het predicaat dat gegeven is als argument.
– De operator `:::` is de operator voor concatenatie van twee lijsten.

Scala verplicht ons niet om functies zonder neveneffecten en onveranderbare variabelen te gebruiken maar moedigt ons aan om waar mogelijk daar gebruik van te maken. Wanneer neveneffecten en het programmeren van een status nodig zijn (een programma dat daar geen gebruik van maakt zou van weinig nut zijn), moeten we proberen om ze te isoleren in gescheiden modulen.

2.3 ACTORS EN CONCURRENCY IN SCALA

Scala kent twee modellen voor parallel programmeren, namelijk het concurrency-model en het actor-model.

Concurrency-model

Het concurrency-model met gemeenschappelijk geheugen is al bekend van Java. In dit model worden codefragmenten, die meestal als aparte threads zijn geïmplementeerd, concurrent uitgevoerd om in kritieke secties de gemeenschappelijke variabelen te inspecteren en te wijzigen. Synchronisatie en wederzijdse uitsluiting spelen hierbij een rol.

Actor-model

Daarnaast is er een model waarbij het geheugen gedistribueerd is en waarbij gecommuniceerd wordt met asynchrone berichten door middel van message passing. Dit laatste model wordt in Scala gerealiseerd met het *actor-model*. Actors zijn aan Scala toegevoegd in de vorm van een bibliotheek, de package scala.actors. Actors zijn niet eigen aan Scala. Ze komen voor in verschillende programmeertalen waaronder de functionele taal Erlang. De eenvoud van het concept maakt het veel makkelijker om robuuste parallelle programma's te schrijven.

Actor

Actors zijn parallelle processen die communiceren door berichten uit te wisselen. Een actor beslist welke actie wordt ondernomen na het ontvangen van een bericht. De actie kan intern afgehandeld worden, een aanleiding zijn tot het zenden van een bericht aan een andere actor of tot het creëren van een nieuwe actor die dan de afhandeling voor zijn rekening neemt.

Mailbox

Een actor heeft een *mailbox* (een soort van buffer) waarin de berichten voor de actor worden geplaatst.

Trait Actor

De package scala.actors bevat de *trait Actor*. Een actor is een subklasse van Actor. Van de trait Actor bespreken we kort de vier methoden start, act, receive en react en een operator !.
Methode start (geïmplementeerd in Actor) wordt aangeroepen om een actor te starten. Binnen de methode start wordt methode act aangeroepen. Het gedrag van de actor wordt bepaald door de implementatie van de methode act. De standaardimplementatie in Actor dient dus te worden geherdefinieerd in onze subklasse; zie bijvoorbeeld figuur 16.5. De twee methoden receive en react worden beide gebruikt voor het lezen en afhandelen van een bericht. Methode receive wordt gebruikt bij een thread-based actor: elke actor wordt in een aparte thread uitgevoerd. De methode receive blokkeert totdat een bericht van het verwachte type is ontvangen. Voor een voorbeeld van methode receive zie figuur 16.5. De methode react wordt gebruikt bij een event-based actor: het gedrag van de actor is gedefinieerd in event handlers. Over de verschillen van beide implementaties gaan we hier niet in. Onthoud alleen dat thread-based actors eenvoudiger maar minder efficiënt zijn dan event-based actors. Een instantie kan een bericht naar een actor sturen door de operator ! te gebruiken.

We geven in figuur 16.5 een voorbeeld van een actor waarbij een teller wordt opgehoogd. In de linker kantlijn hebben we regelnummers toegevoegd.

```
01  import scala.actors.Actor
02  import scala.actors.Actor._
03
04  case class Inc(amount: Int)
05  case class Value()
06
07  class Counter extends Actor {
08    var value: Int = 0;
09
10    def act() = {
11      loop {
12        receive {
13          case Inc(amount) =>
14            value += amount
15          case Value =>
16            println("Value is " + value)
17            exit()
18        }
19      }
20    }
21  }
22
23  object ActorTest extends Application {
24    val counter = new Counter
24    counter.start()
26
27    for (i <- 0 until 100000) {
28      counter ! Inc(1)
29    }
30    counter ! Value
31    // Output: Value is 100000
32  }
```

FIGUUR 16.5 Actor in Scala

OPGAVE 16.9

Bekijk de code van figuur 16.5 en beantwoordt de volgende vragen.

a Wat is de naam van de actor-klasse en op welke regel wordt een instantie gecreëerd?

b Wat is de naam van de berichten en hoe zijn ze geïmplementeerd?

c Op welke regel(s) worden berichten gestuurd en wie stuurt de berichten?

d Op welke regel(s) leest de actor de berichten?

e Waarom is het attribuut value met toegangsspecificatie var gedeclareerd?

f Beschrijf het gedrag van de actor?

g Hoe wordt het probleem van wederzijdse uitsluiting door de actor opgelost?

Het actor-model verandert het denkproces over concurrency en de veiligheid van threads. Het denkproces evolueert van een model waarin de focus ligt op gemeenschappelijke data (data concurrency) naar een model waarin de focus ligt op de structuur van de code die de data bewerkt (task concurrency) en waarbij zo weinig mogelijk data gemeenschappelijk is.

ZELFTOETS

1 Geef in eigen woorden een omschrijving van het aspectgeoriënteerde programmeerparadigma.

2 a Noem drie voordelen van AOP.
b Noem drie nadelen van AOP.

3 Waaraan herkent u functionele programmacode in Scala?

4 Gegeven zijn de volgende Scala-codefragmenten:

```
abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}

class BasicIntQueue extends IntQueue {
  ...
}

trait Doubling extends IntQueue {
  abstract override def put(x: Int) { super.put(2 * x) }

  ...
}

trait Incrementing extends IntQueue {
  abstract override def put(x: Int) { super.put(x + 1) }

  ...
}
```

Leg in eigen woorden uit wat de betekenis is van de volgende twee opdrachten en wat het verschil tussen beide is:

```
val queue1 = new BasicIntQueue with Doubling
                               with Incrementing

val queue2 = new BasicIntQueue with Incrementing
                               with Doubling
```

5 Gegeven is de volgende Scala-code:

```
import scala.actors.Actor
import Actor._

case class MoreValues(values: Array[Int])
case class GetValues(sender: Actor)
case class Reply(array: Array[Int])

class MyArrayActor extends Actor {
  private var array = new Array[Int](0)

  def act = loop {
    receive {
      case MoreValues(values) =>
        array ++= values
        println ("Got: " + values.deep.mkString(" "))
      case GetValues(sender)  =>
        sender ! Reply(array)
    }
  }
}

object MyArrayTest extends Application with Actor{

  val myArray = new MyArrayActor
  myArray.start()
  this.start()
  myArray ! MoreValues(Array(1, 2, 3))
  myArray ! MoreValues(Array(6, 5, 4, 3))
  myArray ! GetValues(this)

  def act = loop {
    receive {
      case Reply(values) =>
        println ("Array: " + values.deep.mkString(" "))
    }
  }
}
```

In deze code betekent de aanroep van values.deep.mkString(" ") dat er een string wordt opgebouwd met alle elementen van de array, met telkens een spatie ertussen.

a Is MyArrayTest een companion object?
b Hoeveel Actor-klassen worden in de code gedefinieerd en hoeveel actors worden gestart?
c Beschrijf welke acties de actors implementeren.
d Beschrijf de werking van de applicatie.
e Stopt het programma?

Bijlage

A look at aspect-oriented programming

Gary Police

IBM

the Rational edge
e-zine for the rational community

developerWorks

A look at aspect-oriented programming

e-mail it!

Gary Pollice
Worcester Polytechnic Institute
27 0 2004

from The Rational Edge: Pollice provides an overview of AOP and offers some observations on its promise, what we need to do to realize that promise, and some issues and obstacles along the way.

Contents:

Subscriptions:

In a perfect world, there would be no such thing as software rework. We would get the object model right the first time. The waterfall approach to development would work just fine. But alas, we don't live in a perfect world. That's why we continue to seek better ways to build our software systems.

As realists, we acknowledge that no one process, technique, language, or platform is good for all situations. We increase the number of tools in our repertoire so that when we need that special tool or technique, we are well-prepared to use it. The history of our industry is rife with examples of improvements in our approach to building software, from the introduction of high-level languages, structured programming, and the object-oriented approach, to the development of spiral and iterative methods, and so on. One of the latest entrants in this lineup is aspect-oriented programming (AOP), which represents one facet of aspect-oriented software development (AOSD). In this month's column, I'll provide an overview of AOP and offer some observations on its promise, what we need to do to realize that promise, and some issues and obstacles along the way. We'll learn a little bit about AOSD, too.

What is AOP?

Aspect-oriented programming is one of those ideas that seems new but has actually been around for quite a while. Many people contributed to the body of knowledge available today, and many can lay claim to being a founder. The person most commonly associated with AOP is Gregor Kiczales, currently at the University of British Columbia, where he works on software modularity research. From 1984 to 1999, Kiczales worked on AOP at the Xerox Palo Alto Research Center (PARC) and was a leader in developing implementations of it.

CC BY NC ND

The best way to describe AOP is by example. Suppose you were responsible for maintaining a large software system that managed the payroll and personnel functions for your organization. What if management issued a new requirement that you create a log of all changes to an employee's data? This would include payroll changes, such as a change in the number of deductions, raises, overtime, and so on. It would also include any changes to the employee's title, personal data, and any other information associated with an employee. How would you go about meeting this requirement?

Most people, in the absence of a better way, would search through the code and insert calls to a logging method in appropriate places. If the system were well-designed and constructed, the code might require changes in only a few places. For most systems, though, insertions would be necessary in many places. If the system were object-oriented, we might build a logging class and then use an instance of it to handle the logging. We might need a hierarchy of classes to handle different files and databases. Attempting to fulfill the requirement in this way would not be a trivial job, and it would not be easy to ensure that we'd made all of the necessary insertions, either.

Even if we had a superior, object-oriented system implementation, we would encounter some problems. In their book, *Mastering AspectJ*,™ Joseph Gradecki and Nicholas Lesiecki note that OO systems often produce classes that are difficult to change, and code that can't be reused and is difficult to trace through (i.e., numerous inter-object messages make it difficult to follow a logical thread for the code).[1]

Cross-cutting concerns

The type of logging required in our example is a *cross-cutting* concern. It cannot be isolated or encapsulated in one or two specific classes; the logging involves changes in many places across the system. Our desire is to keep the system maintainable, and therefore we want our logging approach to keep the code as clean and simple as possible. We would also like to avoid significant structural changes to the system's architecture. So how do we implement a cross-cutting concern such as logging? We could refactor all of the code by creating a logging class and performing the appropriate insertions. However, in a large system, this would be a time-consuming, error-prone job.

AOP is designed to handle cross-cutting concerns by providing a mechanism, the *aspect*, for expressing these concerns and automatically incorporating them into a system. AOP does not *replace* existing programming paradigms and languages; instead, it works *with* them to improve their expressiveness and utility. It enhances our ability to express the separation of concerns necessary for a well-designed, maintainable software system. Some concerns are appropriately expressed as encapsulated objects, or components. Others are best expressed as cross-cutting concerns.

The logging example in AspectJ

Let's continue looking at the logging problem and see how it might be solved using AspectJ, an AOP implementation that extends the Java language. AspectJ is perhaps the best known and most widely used AOP implementation. Even if you are not familiar with Java, you should be able to understand the major ideas I will present below.

Figure 1 depicts one way we might structure our modification of the system. The Finance system has an interface and several methods for updating an employee's financial data. The names of the methods all begin with the word `update` (e.g., `updateFederalTaxInfo`), and each financial update takes an Employee object as an argument. Employees' personnel information is also updated through an Employee object, using the methods shown in Figure 1.

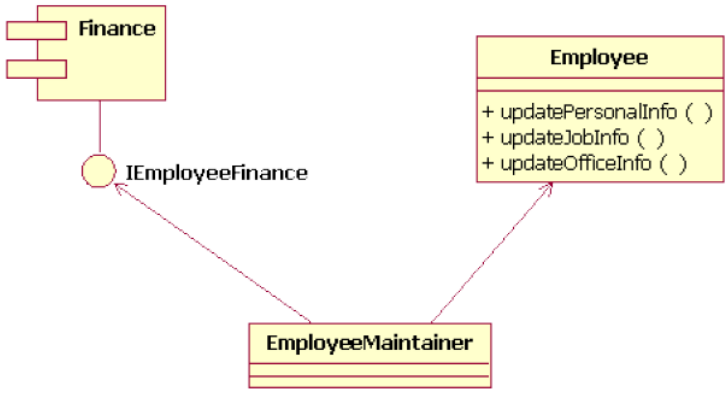


Figure 1: Classes involved in updating employee information

We can describe, in prose, what we need to do. Every time we call any of the updating functions and the update succeeds, we write a logging message. For simplicity, we will say that we print a logging message to the standard output. In the real system we would write to a log file. We would take three steps to implement our solution in AspectJ:

1. Identify places in the code where we want to insert the logging code. This is called *defining join points* in AspectJ.
2. Write the logging code.
3. Compile the new code and integrate it into the system.

I will describe each of these steps in the next three sections.

Define the join points

A *join point* is a well-defined point in the code at which our concerns crosscut the application. Typically, there are many join points for each concern. If there were just one or two, we could manually change the code with little effort.

In AspectJ, we define join points by grouping them into *pointcuts*. (The AspectJ syntax is rich, and we will not attempt to describe it completely here.) To start, we define two pointcuts, one each for grouping the join points in the Employee class and the IEmployeeFinance component. The following code defines the two pointcuts:

```
pointcut employeeUpdates(Employee e):
     call(public void Employee.update*Info()) && target(e);

pointcut employeeFinanceUpdates(Employee e) :
     call (public void update*Info(Employee)) && args(e);
```

The first pointcut, named `employeeUpdates`, describes all join points where we call an Employee object's method that begins with the string `update` and ends with the string `Info`, and has no arguments. It also specifically identifies the method as defined in the Employee class through the `target` designator. The second pointcut, `employeeFinanceUpdates`, describes all join points where there is a call to any method that begins with `update`, ends with `Info`, and has one argument of type Employee. Together, these two pointcuts define all of the join points with which we are concerned. If we add more updating methods to the Employee class or the IEmployeeFinance component, calls to them will automatically be included in the pointcuts, as long as we maintain the same naming conventions. This means that we do not have to deliberately include logging code every time we add an updating method.

Write the logging code

The code that implements logging is similar to any method in Java, but it is placed in a new type, called an aspect. The aspect is the mechanism we use to encapsulate code related to a specific concern. The implementation of the aspect for logging employee changes might look like the following:

```
public aspect EmployeeChangeLogger {
   pointcut employeeUpdates(Employee e) : call(
   public void Employee.update*Info())
   && target(e);

 pointcut employeeFinanceUpdates(Employee e) : call(
   public void update*Info(Employee))
   && args(e);

 after(Employee e) returning : employeeUpdates(e)
   || employeeFinanceUpdates(e) {
   System.out.println("\t>Employee : " +e.getName() +
   " has had a change ");
   System.out.println("\t>Changed by " +
   thisJoinPoint.getSignature());
   }
   }
```

First, notice that the aspect structure is similar to a class in Java. The aspect is typically placed in its own file, just like a Java class. Although the usual method is to include the previously defined pointcuts here in the aspect code, as we have done, we could instead include them closer to code containing the join points.

Following the pointcuts, we have a section of code that is similar to a method in regular Java code. This is called *advice* in AspectJ. There are three different types of advice: before, after, and around. They execute before the join point, after the join point, and instead of the join point, respectively. There are many variations you can use to customize your advice. In our example, we choose to perform the logging immediately after the updating method in the join point returns. Notice also that we combined the two pointcuts by naming each of them immediately after the colon in the advice header and connecting them with a logical "or." We were able to do this easily because both pointcuts have an Employee parameter.

Two statements in the advice print out the fact that the employee's information was changed, along with the name of the employee. This is easy to arrange, since the affected employee object is passed in as an argument to the advice. The second statement identifies the exact join point where the advice is executed and makes use of the AspectJ `JoinPoint` class. Whenever advice executes, it has an associated join point referenced by `thisJoinPoint`.

Compile and test

Now that we have written the code for logging, we need to compile it and integrate it into the existing system. For our example, we have implemented two classes: Employee and EmployeeFinance. We also have a simple test class with a main method, as shown below:

```
public static void main(String[] args) {
     Employee e = new Employee("Chris Smith");
     // Do something to change some of the employee's
     // information here.
     e.updateJobInfo();
     e.updateOfficeInfo();
     EmployeeFinance.updateFederalTaxInfo(e);
     EmployeeFinance.updateStateTaxInfo(e);
     }
```

This code runs just fine without any AOP implementation. For our example, the bodies of all the update methods just contain a print statement. When we run this example we get the following output:

```
Updating job information
Updating office information
Updating federal tax information
Updating state tax information
```

In order to incorporate our aspect into the system, we add the aspect source code to the project and build with the AspectJ compiler, `ajc`. The compiler takes each aspect and creates class files that contain the advice code. Then, calls to the appropriate methods in these class files are *woven* into the original application code. With the current release of AspectJ, this weaving takes place at the Java bytecode level, so there are no intermediate source files you can look at to see the final code. However, you can decompile the Java bytecode if you are curious.

I use the Eclipse environment for development, and the AspectJ plug-in takes care of invoking the compiler with the proper arguments. Once we have compiled the project with the AspectJ compiler, we get the following output:

```
Updating job information
     >Employee : Chris Smith has had a change
     >Changed by void employee.Employee.updateJobInfo()
     Updating office information
     >Employee : Chris Smith has had a change
     >Changed by void employee.Employee.updateOfficeInfo()
     Updating federal tax information
     >Employee : Chris Smith has had a change
     >Changed by void employee.EmployeeFinance.updateFederalTaxInfo(Employee)
     Updating state tax information
     >Employee : Chris Smith has had a change
     >Changed by void employee.EmployeeFinance.updateStateTaxInfo(Employee)
```

Now we know which employee had information changed, and where the changes were made. The logging could certainly be much more elaborate, but the basic technique would not change.

What does this all mean to you?
Aspect-oriented technology has many potential benefits. It provides a way of specifying and encapsulating cross-cutting concerns in a system. This might, in turn, allow us to do a better job of maintaining systems as they evolve — and we do know that they *will* evolve. AOP would let us add new features, in the form of concerns, to existing systems in an organized manner. The improvements in expressiveness and structure might allow us to keep systems running longer, and to incrementally improve them without incurring the expense of a complete rewrite.

AOP may also be a great addition to quality professionals' toolboxes. Using an AOP language, we might be able to test application code automatically without disturbing the code. This would eliminate a possible source of error.

We are in the early stages of understanding the full potential of AOSD. It seems clear that the technology offers enough benefits to warrant further exploration and experimentation. How far are we from everyday usage of AOP languages for application development? It depends on whom you ask.

Now that we've seen some benefits, let's look at some of the risks concerning AOSD as well as what is needed to bring it into the software development mainstream. The next sections discuss these issues.

Quality and risks

Having looked at AOSD from a quality viewpoint and done a little exploring with AspectJ, I've seen potential risks along with the benefits. I will discuss three issues that illustrate some of the challenges we may face with respect to quality as AOSD becomes more popular.

The first issue is how we will need to modify our process to accommodate AOP. One of the most effective techniques for detecting software defects is through code inspections and reviews. During a review, a team of programmers critiques code to determine if it meets its requirements. With object-oriented programs, we can review classes, or sets of related classes, and reason about them. We can look at the code and determine if it handles unexpected events properly, has logical flaws, and so on. In an OO system, each class completely encapsulates the data and behavior of a specific concept.

With AOP, however, we can no longer reason about a class just by looking at the code for it. We do not know whether the code might be either augmented by advice from some aspect or completely replaced by such advice. To be able to reason about an application's code, we must be able to look at the code from each class as well as the code for any aspects that might affect the class's behavior. However, it is possible that the aspects have not been written yet. If that is so, then how much can we truly understand about the behavior of the application class's code when we consider it in isolation?

In fact, the way to think about correctness in AOP code is the inverse of how we consider it for object-oriented programming (OOP) code. With OOP, we go from inside out: We consider a class, make assumptions about its context, and then reason about its correctness, both in isolation, and in terms of how it will interact with other classes. With AOP, we need to look from the outside in, and determine the effects of each aspect on every possible join point. Determining how to reason correctly about AOP, and developing the appropriate techniques and tools to help us, is a rich research area.

A second, more practical issue regarding the potential widespread adoption of AOP is the development of tools and techniques for testing, especially unit testing. Because code may be changed by an aspect, unit tests that run perfectly on a class may behave quite differently when the class is integrated into an AOP system. The following example illustrates this point.

As you probably know, a *stack* is a data structure that is used to add and remove items in a last-in, first-out manner. If you push the numbers 2, 4, and 6 onto a stack and then pop the stack twice, you will get 6 and 4, in that order. Writing a unit test for a Stack class is straightforward. We can read the requirements specification and the code and do a very good job of ensuring that the implementation is correct. One of the first things I did when I began to look at AspectJ was to implement a stack and see what I could do with aspects to change the behavior. I implemented a simple change: Increment each item by 1. This is very easy to do. Now, my unit test — which pushes 2, 4, and 6 onto the stack and pops the top two elements off to verify that they are 6 and 4 — fails. The code that the unit test is testing did not change, yet the behavior changed. Instead of 6 and 4, the results are now 7 and 5.

This is a trivial example, and one that probably would not occur in a real-life situation. But it shows that a malicious programmer can cause a lot of harm quite easily. Even if we ignore malicious programmers, many defects occur because there are unwanted side effects of changes we make. It might be very difficult to ensure that an aspect implemented for very valid reasons does not have unwanted effects on existing program functionality.

The third issue is the testing process itself. Once we have a set of tools and techniques, how do we modify our testing process to use these effectively and support our overall development goals? Although this issue may not be a major one, I believe it will need to be addressed before we can really do a good job of testing software built with aspects.

Other barriers to AOSD adoption

Quality issues may be the biggest deterrents to adopting AOSD methods, but they are not the only ones. AOSD is a new paradigm. As did other paradigms when they were new (e.g., object-oriented software development), this one will take time to be adopted widely because it involves a learning curve. First we must learn basic techniques and mechanics, then advanced techniques, and then how to best apply the techniques and when they are most appropriate.

Tools will play a major part in industry adoption of AOP. In addition to compilers and editors, we need tools to help us reason about systems, identify potential cross-cutting concerns, and help us test in the presence of aspects. As we find methods to help us represent the systems better, such as representing aspects in UML, our tools must evolve to support these methods.

cc BY NC ND

Although I discussed AspectJ in this article, it is neither the only AOP implementation nor the only one for Java. Furthermore, other paradigms similar to AOP are also emerging. The multi-dimensional separation of concerns paradigm, for example, has been under development at IBM Research, and an implementation is available in the HyperJ tool (http://www.alphaworks.ibm.com/tech/hyperj). Using any new paradigm is risky until standard implementations are established for the languages you work with. AspectJ, for instance, is a still-evolving AOP implementation. The danger is that you may develop software that incorporates cross-cutting concerns, and your implementation approach will either cease to be supported or will change significantly.

Moving toward a better way to build software

Clearly, we have a way to go in developing tools and processes to make AOP viable for everyday use. However, I do not believe that any of the issues I have discussed in this article represent insurmountable problems. I *do* believe that when we have developed a proven set of tools and processes for AOP, we will have a better way to build software than we do today.

As Barry Boehm said about agile processes,[2] we must approach AOP with care. Whether we are early adopters or we wait for this technology to become more mainstream, we need to ensure that our software investment will provide acceptable returns today and in the future. That's just good business sense.

Resources

If you are interested in AOSD and AOP, here are some places to start your investigation.

Communications of the ACM, October 2001, has many articles on AOSD.

"Improve Modularity with Aspect-oriented Programming," by Nicholas Lesiecki provides a good overview: http://www-106.ibm.com/developerworks/java/library/j-aspectj/.

"Test Flexibility with AspectJ and Mock Objects," by Nicholas Lesiecki shows how AOP can be used to help with unit testing: http://www-106.ibm.com/developerworks/java/library/j-aspectj2/?open&l=007,t=gr

Mastering AspectJ, by Joseph D. Gradecki and Nicholas Lesiecki, is a book published by John Wiley, 2003.

For a more detailed list of resources, visit my Web page at WPI: http://www.cs.wpi.edu/~gpollice/Interests/AOSD.html

Notes

[1] *Mastering AspectJ*,™ by Joseph D. Gradecki and Nicholas Lesiecki, John Wiley, 2003, p. 8.

[2] "Get Ready for Agile Methods, with Care", Barry Boehm, *IEEE Computer*, January, 2002.

About the author

Gary Pollice is a Professor of Practice at Worcester Polytechnic Institute, in Worcester, MA. He teaches software engineering, design, testing, and other computer science courses, and also directs student projects. Before entering the academic world, he spent more than thirty-five years developing various kinds of software, from business applications to compilers and tools. His last industry job was with IBM Rational Software, where he was known as "the RUP Curmudgeon" and was also a member of the original Rational Suite team. He is the primary author of *Software Development for Small Teams: a RUP-Centric Approach*, published by Addison-Wesley in 2004. He holds a B.A. in mathematics and M.S. in computer science.

TERUGKOPPELING

1 Uitwerking van de opgaven

16.1 a Het artikel gebruikt als voorbeeld voor een cross-cutting concern het in een log bijhouden van alle wijzigingen aan de data van werknemers.
b Een ander voorbeeld van een cross-cutting concern is de controle op autorisatie in een systeem met verschillende toegangsrechten.

16.2 De drie verschillende typen advices zijn before, after en around. De differentiatie is nodig om aan te geven wanneer de code van de advice uitgevoerd moet worden bij een joint point.

16.3 Tijdens compilatie wordt de code van de advice geïntegreerd met de code van de applicatie bij alle joint points die gespecificeerd zijn in de point cuts van de aspect.

16.4 Unit-testen is gericht op het testen van de functionaliteit van code. Voor de integratie met de aspecten kan de code goed getest worden terwijl na integratie met de aspecten de code zich totaal anders kan gedragen. Realiseer u daarbij dat aspecten later in de tijd toegevoegd kunnen worden.

16.5 a Methode swap heeft twee parameters i en j van type Int. De methode heeft een terugkeerwaarde van type Unit. Dit is in de code weggelaten. Het komt erop neer dat de methode geen terugkeerwaarde heeft en alleen gebruikt wordt voor de neveneffecten.
NB Het type Int is de klasse die alle gehele waarden representeert.
b Methode iter heeft één parameter i van type Int.
c Methode iter heeft een terugkeerwaarde van type String. De terugkeerwaarde is de waarde van de expressie na het is-teken:

```
ar(i) + (if (i < ar.length-1) "," + iter(i+1) else "")
```

16.6 In figuur 2 herkennen we de disjuncte vereniging. Een instantie van type Term is of een instantie van type Var, of een instantie van type Fun of een instantie van type App.

16.7 Het resultaat van de aanroep van even(0, 10) is de waarde List(0, 2, 4, 6, 8, 10).

16.8 In de implementatie van figuur 16.2 zien we dat gebruik wordt gemaakt van een instantie van type Array[Int]. Op deze array wordt de waarde van afzonderlijke elementen veranderd (selectieve updating). Verder maken while-lussen gebruik van veranderbare variabelen en heeft de methode sort een terugkeerwaarde van type Unit. Methode sort veroorzaakt dus neveneffecten.
In de implementatie van figuur 16.4 wordt gebruikgemaakt van een onveranderbare lijst van type List[Int]. Er worden uitsluitend variabelen gedeclareerd met toegangsspecificatie val. De iteratie wordt bereikt met behulp van recursie en de methode sort1 heeft als terugkeerwaarde een instantie van type List[Int], dat wil zeggen een onveranderbare lijst.

16.9 a De naam van de actor-klasse is Counter; zie regel 7. De instantie met naam counter wordt op regel 24, in de applicatie (een instantie van ActorTest), gecreëerd.
b De berichten zijn case-klassen met de namen Inc en Value; zie de regels 4 en 5.
c De applicatie stuurt de berichten naar de actor. Het bericht Inc wordt in een lus op regel 28 gestuurd en het bericht Value wordt eenmalig op regel 30 gestuurd.
d De actor leest de berichten binnen de methode receive op regel 13 en op regel 15. Er wordt gebruikgemaakt van patroonherkenning.
e Het attribuut value is met het sleutelwoord var gedeclareerd omdat de waarde van het attribuut tijdens verwerking moet kunnen veranderen.
f De actor counter houdt een waarde, de instantie value, bij. De actor wacht in een oneindige lus (loop is een afkorting van while(true)) op berichten. Ontvangt de actor een bericht van klasse Inc, dan wordt de waarde van value verhoogd met de waarde van het argument van het bericht. Ontvangt de actor een bericht van klasse Value, dan print de actor de waarde op de standaarduitvoer en wordt de oneindige lus gestopt.
g Het attribuut counter kan door meerdere instanties gewijzigd en/of geïnspecteerd worden. Nu gebeurt het alleen door de applicatie, maar andere instanties zouden ook een bericht Inc of Val aan de actor kunnen sturen. Wederzijdse uitsluiting is geregeld doordat de actor de berichten één voor één leest en afhandelt.

2 Uitwerking van de zelftoets

1 Aspectgeoriënteerd programmeren is een programmeerparadigma dat als doel heeft het verhogen van de modulariteit van een programma door cross-cutting concerns, dat wil zeggen code dat op diverse plaatsen in het programma moet optreden, te scheiden van de kernfunctionaliteit. De gescheiden code wordt tijdens compilatie samengevoegd.

2 a Drie voordelen van AOP zijn:
– Het is mogelijk om met minder grote ingrepen nieuwe eigenschappen aan een bestaand systeem toe te voegen.
– Het is mogelijk om een systeem modulair te ontwerpen en opbouwen.
– Het is mogelijk om een serie testen uit te voeren op een bestaand systeem.
b Drie nadelen zijn:
– Het is moeilijker om fouten in een programma te detecteren.
– Unit testen is lastiger omdat code zich anders kan gedragen voor- en na integratie met AOP.
– Adoptie van AOP wordt vertraagd door de lange leercurve.

3 Functionele programmacode in Scala is te herkennen aan de volgende kenmerken:
– Gebruikte variabelen zijn onveranderbaar, dus met toegangsspecificatie val in plaats van var.
– Datastructuren zijn onveranderbaar, dus instanties van subklassen van scala.collection.immutable.
– Iteratie wordt geprogrammeerd met behulp van recursie en lijstcomprehensies.
– Functies hebben een terugkeerwaarde anders dan Unit en de terugkeerwaarde is een onveranderbare waarde.

4 Wat beide opdrachten gemeen hebben, is dat ze een onveranderbare variabele creëren als instantie van de klasse BasicIntQueue. Aan de instantie worden de methoden van de traits Doubling en Increment toegevoegd.
Waar beide opdrachten in verschillen, is welke implementatie van de methode put wordt gebruikt. Bij queue1 is dat die van Incrementing (de waarde van het argument wordt met één verhoogd) en bij queue2 is dat die van Doubling (de waarde van het argument wordt verdubbeld).

5 a Nee, het object MyArrayTest is geen companion object. Het is een singleton-object. Een companion object draagt altijd dezelfde naam als de companion klasse.
b Er worden twee Actor-klassen gedefinieerd: de klasse MyArrayActor en de klasse MyArrayTest waarvan slechts een singleton-object wordt gemaakt. Er worden ook twee actor-instanties gecreëerd en gestart: myArray en this, het singleton-object MyArrayTest.
c Er worden drie soorten berichten gedefinieerd: van de klassen MoreValues, GetValues en Reply.
De actor myArray leest in een oneindige lus berichten van de klassen MoreValues en GetValues. Bij een bericht van type MoreValues wordt het argument van het bericht toegevoegd aan de elementen van het attribuut array en wordt een melding geprint. Bij een bericht van type GetValues wordt een bericht teruggestuurd naar de afzender die meegegeven is als argument van het bericht GetValues. Met het bericht wordt de waarde van de array gestuurd.
De actor this leest in een oneindige lus berichten van de klasse Reply. Wordt een bericht ontvangen, dan wordt de inhoud daarvan geprint.
d De werking van de applicatie is gecodeerd in MyArrayTest. Eerst worden de twee actors gestart en daarna worden drie berichten naar de actor myArray gestuurd. Dan is de applicatie zelf afgelopen.
e Nee, het programma stopt niet. Hoewel de applicatie alle code heeft verwerkt, blijven de beide actors in hun oneindige lus wachten op berichten.