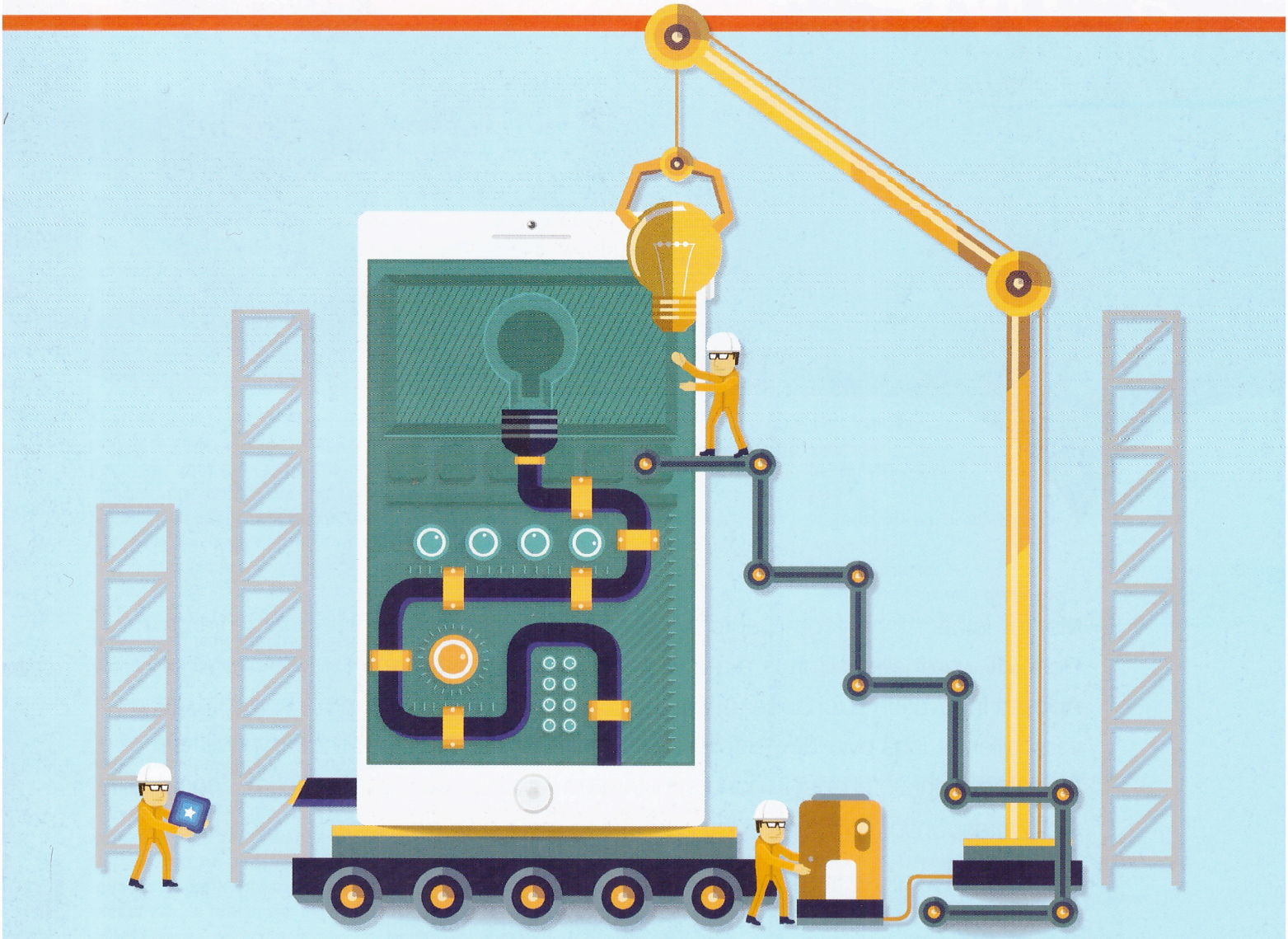
Nummer 2 - 2016

JAVA MAGAZINE

.nl.jug

Onafhankelijk tijdschrift voor de Java-professional



Software Architecture as code

Features

IoT Tech Day 2016 programmaboekje

DSL kweken voor hypothekenmarkt

Spot the security bugs

Een DSL kweken

Voor de hypothekenmarkt

Het verstrekken van hypotheken is één van de belangrijkste diensten van de Rabobank. Voor de vernieuwing van het hypotheekproces is het Rabobank Hypotheekdossier ontwikkeld. Hier kunnen klanten zich oriënteren op de mogelijkheden van een hypotheek op basis van gegevens als hun inkomen en eventuele schulden. Hiervoor wordt gebruik gemaakt van een groot aantal complexe berekeningen. De Java-code, die de berekeningen uitvoert, is in de loop der tijd lastig onderhoudbaar geworden en moest vervangen worden door code die 1) makkelijker onderhoudbaar is, 2) leesbaar is voor de business (zodat zij minimale reviews kunnen geven), en 3) minder fouten bevat.

Vanuit Codestar hebben wij geadviseerd om dit met Scala te doen, vanwege de mogelijkheden die deze taal biedt om DSL's te definieren. Het grote voordeel om hiervoor Scala te gebruiken, is dat de code leesbaarder wordt voor de business, zonder dat je meteen vast zit aan extra tooling, zoals ANTLR. Een nadeel is dat voor het onderhoud van de DSL zelf specialistische Scala-kennis nodig is.

We hebben deze DSL op een organische manier opgebouwd, door stapje voor stapje features toe te voegen: we hebben hem gekweekt, als het ware. In **Listing 1** zie je de originele Java-code. (Deze code is fictief, maar geeft wel een goede indruk van de echte code.) Dit hebben we omgezet naar de code in **Listing 2**: een groot verschil! Laten we eens kijken hoe dit is gegaan.

Voor de rest van dit artikel ga ik uit van een basiskennis van Scala.

Scala

Programmeurs die aan financiële berekeningen hebben gewerkt, weten dat die worden uitgevoerd met BigDecimals in plaats van doubles, om afrondingsfouten te voorkomen. Helaas betekent dat in Java dat je constructies krijgt als `a.add(b.add(c))` in plaats van `a + b + c`. Dit maakt het moeilijk voor een ontwikkelaar om in één oogopslag te zien wat een berekening doet. Bovendien is de business gewend om `a + b + c` te zien. Het is lastig om aan hen uit te leggen waarom je het in Java niet zo kunt opschrijven.

In Scala kun je wél `a + b + c` schrijven. Een 1-op-1 vertaling van de bestaande Java-code

Jan Ouwens is senior Scala ontwikkelaar bij Codestar. Hij houdt zich bezig met codekwaliteit en het oplossen van puzzels. Ook is hij de maker van EqualsVerifier.

```java
public BigDecimal berekenAnnuiteit(BigDecimal kapitaal, BigDecimal rente, int aantalTermijnen, int termCode) {
    BigDecimal maandelijseRentefactor = rente.divide(new BigDecimal(1200).divide(new BigDecimal(termCode), MC), MC);
    BigDecimal pow = BigDecimal.ONE.add(maandelijseRentefactor).pow(-aantalTermijnen, MC);
    BigDecimal annuiteitenfactor = maandelijseRentefactor.divide(BigDecimal.ONE.subtract(pow), MC);
    return kapitaal.multiply(annuiteitenfactor);
}

public boolean kanIkDitHuisBetalen(BigDecimal prijs, List<BigDecimal> inkomens) {
    BigDecimal rente = new BigDecimal(6);
    int aantalTermijnen = 30 * 12;
    BigDecimal percentageWonen = new BigDecimal(25);

    BigDecimal last = berekenAnnuiteit(prijs, rente, aantalTermijnen, 1);
    BigDecimal totaalInkomen = new BigDecimal(0);
    for (BigDecimal b : inkomens) {
        totaalInkomen = totaalInkomen.add(b);
    }
    BigDecimal schatting = (percentageWonen.divide(new BigDecimal(100), MC)).multiply(totaalInkomen);
    return last.compareTo(schatting) < 0;
}
```

Listing 1: Java

```
def berekenAnnuiteit[P <: Periode](kapitaal: Bedrag, rente: Percentage,
                                   aantalTermijnen: Int, periode: P): Bedrag Per P = {
  val maandelijseRentefactor = rente / periode.frequentie
  val pow = (1 + maandelijseRentefactor).pow(-aantalTermijnen)
  val annuiteitenfactor = maandelijseRentefactor / (1 - pow)
  (kapitaal * annuiteitenfactor) per periode
}

def kanIkDitHuisBetalen(prijs: Bedrag, inkomens: List[Bedrag Per Maand]): Boolean = {
  val rente = 6.procent
  val aantalTermijnen = 30.jaar
  val percentageWonen = 25.procent

  val last = berekenAnnuiteit(prijs, rente, aantalTermijnen, Maand)
  val totaalInkomen = inkomens.sum
  last < percentageWonen * totaalInkomen
}
```

Listing 2: eindresultaat

leidt al tot een grote verbetering in leesbaarheid. Hierdoor viel opeens de volgende regel op:

```
val maandelijkseRentefactor
  rente / (1200 / termCode)
```

Listing 3

Wat zou die vreemde constante `1200` betekenen?

Domeinobjecten

Maar het kan nog leesbaarder. We starten met een case class voor een veelgebruikt concept: het `Bedrag` (zie **Listing 4**). Hier vallen een aantal dingen op. Ten eerste, door het woord `private[dsl]` tussen de naam van de class en de parameterlijst te zetten, maken we niet de class, maar de constructor private, maar tegelijk wel beschikbaar binnen de `finance.dsl` package. Dit doen we, omdat de constructor niet erg verhelderend is in het gebruik: is `Bedrag(10)` gelijk aan 10 euro, 10 dollar of 10 złoty? De implicit classes zorgen ervoor dat we in plaats daarvan `10.euro` kunnen schrijven. Veel duidelijker. We maken 2 `ToBedrag` classes: een voor `Int` en een voor `BigDecimal`, met een gedeelde abstracte superclass. Merk op dat Scala zelf een impliciete conversie heeft van `Int` naar `BigDecimal`, waardoor we deze conversie niet meer zelf hoeven uit te voeren in de constructor.

De method `+` spreekt voor zich, maar `*` is ingewikkelder. We vermenigvuldigen hier met een `BigDecimal`, niet met een `Bedrag`. Dat doen we omdat bedragen vermenigvuldigen raar is. Ter vergelijking: 10 meter * 10 meter = 100m². Is €10 * €10 dan ook 10 vierkante euro? Het is logischer om bedragen uitsluitend te vermenigvuldigen met "rauwe", eenheidsloze getallen. Als Jan en Pim ieder €10 hebben, hebben zij samen €10 * 2 = €20 euro.

Nu hebben we echter een probleem: `10.euro * 2` zou gelijk moeten zijn aan `2 * 10.euro`, maar dat laatste compileert niet. De `*` method in `ToBedrag` maakt dit alsnog mogelijk, want deze accepteert een `Bedrag` en delegeert de vermenigvuldiging naar de `*` method van `Bedrag`. Probleem opgelost.

Op dezelfde manier kunnen we ook een class `Percentage` uitwerken, maar vermenigvuldiging blijft lastig. Om dit voor alle relevante

```
package finance.dsl

case class Bedrag private[dsl] (waarde: BigDecimal) {
  def + (n: Bedrag): Bedrag = Bedrag(waarde + n.waarde)
  def * (n: BigDecimal): Bedrag = Bedrag(waarde * n)
}
object BedragImplicits {
  abstract class ToBedrag(waarde: BigDecimal) {
    def euro: Bedrag = Bedrag(waarde)
    def * (n: Bedrag): Bedrag = n * waarde
  }
  implicit class BigDecimalToBedrag(waarde: BigDecimal) extends ToBedrag(waarde)
  implicit class IntToBedrag(waarde: Int) extends ToBedrag(waarde)
}
```

Listing 4: case class Bedrag

ZONDER DAT JE METEEN VAST ZIT AAN EXTRA TOOLING ZOALS ANTLR

types werkend te maken, moeten we in `Percentage` drie `*` methods opnemen: voor `Int`, voor `BigDecimal` en voor `Bedrag`. Ook moeten we aan `Bedrag` een `*` method toevoegen voor `Percentage`. Later zullen we zien hoe dit beter kan.

Type classes

In Scala kan je met `List(1,2,3).sum` een lijst integers optellen. `List(1.euro, 2.euro).sum` compileert echter niet. Tochis de method `sum` gedefinieerd op `List`. Hoe werkt dit? En kunnen we dit werkend maken voor lijsten van onze `Bedrag` class?

De signature voor `List.sum` ziet er, versimpeld, als volgt uit:

```
def sum(implicit num: Numeric[A]): A
```

Listing 5

Hierbij is `A` de generic parameter van de `List`. Dit betekent dat de compiler op zoek gaat naar een impliciet object van type `Numeric[A]`. `Numeric[A]` is een trait die operaties definieert, zoals `def plus(left: A, right: A): A`. De implementatie van `sum` kan dus `num.plus(x, y)` aanroepen om twee numerieke waarden bij elkaar op te tellen. Omdat `num` impliciet is, hoeven we hem dus ook niet expliciet te noemen bij het aanroepen van `sum`.

Scala definieert zelf een `Numeric[Int]` en een `Numeric[BigDecimal]`. Om een lijst met bedragen op te tellen, hoeven we dus alleen een `Numeric[Bedrag]` te definiëren en deze als impliciet te markeren: `implicit object NumericBedrag extends Numeric[Bedrag] { ... }`

Dit is het 'type class' design pattern: het definiëren van gedrag op een verzameling types dat aan elkaar gerelateerd is, maar geen gezamenlijke superclass heeft. In feite is dit polymorfisme zonder inheritance (zie **[1]**).

Een type class voor vermenigvuldiging

Nu we gezien hebben hoe je de type class `Numeric` kunt uitbreiden, gaan we zelf een type class definiëren.

Eerder zagen we hoe we in Percentage een `*` method moesten definiëren voor `Int`, `BigDecimal` én `Bedrag`. In onze DSL waren er zelfs nog meer types, waarvoor allemaal

```
trait Kwantiteit[A] {
  def vermenigvuldig(n: A, m: BigDecimal): A
}
```

Listing 6

```
def * [A](n: A)(implicit kw: Kwantiteit[A]): A =
  kw.vermenigvuldig(n, p / 100)
```

Listing 7

overloads zouden moeten bestaan. Met een type class kunnen we dit terugbrengen naar één enkele overload. Helaas kunnen we hier niet `Numeric` gebruiken, omdat die verwacht dat je waarden van dezelfde types met elkaar vermenigvuldigt. Wij willen nu juist een Bedrag met een eenheidsloze waarde vermenigvuldigen. Daarom introduceren we de type class `Kwantiteit` (zie **Listing 6**).

Deze kunnen we op de voor de hand liggende manier implementeren voor `Int`, `BigDecimal` en `Bedrag`. Vervolgens definiëren we `*` in `Percentage` als in **Listing 7**.

Nu kunnen we schrijven: `10.procent * 50.euro`, en daar komt, zoals verwacht, `5.euro` uit. Net als bij `List.sum` zal de Scala-compiler op zoek gaan naar een impliciete waarde van type `Kwantiteit[Bedrag]` en deze automatisch invoegen in de aanroep. We hoeven er enkel voor te zorgen dat deze waarde in scope is.

Overigens komen type classes zo vaak voor, dat Scala ook een andere manier heeft om de `*` method te schrijven:

```
def * [A: Kwantiteit](n: A): A = ...
```

Listing 8

Deze notatie is equivalent aan de eerdere notatie met een `implicit kw` parameter. Persoonlijk vind ik deze notatie mooier, omdat in de generic parameter meteen duidelijk is wat voor type we verwachten. Een voordeel van de eerdere notatie is dat er minder "magie achter de schermen" plaatsvindt, iets waar implicits in Scala natuurlijk al snel last van hebben.

Overigens kunnen we nog niet `50.euro * 10.procent` schrijven. Hiervoor moeten we de class uit **Listing 9** toevoegen.
We hebben nu dus nog maar 2 `*` methods over: één in `Percentage` die een `Kwantiteit` als parameter accepteert en één in `KwantiteitMetPercentage` die een `Percentage` neemt.

STAPJE VOOR STAPJE DE DSL KWEKEN, ALS HET WARE

```
implicit class KwantiteitMetPercentage[A: Kwantiteit](waarde: A) {
  def * (p: Percentage): A = p * waarde
}
```

Listing 9

```
case class Per[T](waarde: T, periode: Periode)
```

Listing 10

```
package finance

package object dsl
    extends BedragImplicits
    with PerImplicits
    with PercentageImplicits
    with PeriodeImplicits
    with Ordering.ExtraImplicits {

  val Maand = Periode.Maand
  val Jaar = Periode.Jaar
}
```

Listing 11: package object

Per

Maar het kan nog beter. Om het moeilijker te maken om subtiele foutjes te maken, willen we Scala's type system inzetten om onderscheid te maken tussen bedragen per jaar en bedragen per maand. Hiervoor introduceren we een case class `Per` en een type `Periode` (zie **Listing 10**).

Dat geeft ons types `Per[Bedrag, Maand]` en `Per[Bedrag, Jaar]`. Maar in Scala heeft een type met twee generic parameters ook een infix-notatie. Dat betekent dat je het type ook kunt schrijven als `Bedrag Per Maand`. Nu hebben we een syntax, die intuïtief leesbaar is voor iedereen die Nederlands spreekt.

Uiteraard moeten we `Per` ook aankleden met de `Numeric` en `Kwantiteit` type classes. Dit laat ik als een oefening voor de lezer.

Losse eindjes

We hebben inmiddels een hoop implicits gedefinieerd, die allemaal geïmporteerd moeten worden om ze te kunnen gebruiken. Helaas is het in IDE's lastig om implicits te importeren, want je weet immers niet hoe ze heten. Het helpt dat we ze in objects naast de bijbehorende case classes hebben gezet. We kunnen nu immers `import BedragImplicits._` typen om ze in scope te halen. Maar het blijft onhandig dat we het met de hand moeten doen, en voor elke object afzonderlijk.

In plaats hiervan kunnen we Scala's package object gebruiken. Als `BedragImplicits` en zijn vrienden traits zijn in plaats van objects, kunnen we deze extenden in de package object, zoals te zien is in **Listing 11**. Nu is een simpele `import finance.dsl`. voldoende om de volledige DSL te importeren.

Conclusie

Om een zo leesbaar mogelijke DSL te kweken, hebben we veel geavanceerde Scala-features gebruikt, zoals impliciete conversies, type classes, de infix-notatie voor types en het package object. Hierdoor blijft mijn eerdere kanttekening overeind, dat voor het onderhoud van deze DSL veel Scala-kennis vereist is. Is deze complexiteit het waard?

Uit codereviews met ontwikkelaars zonder Scala-kennis bleek dat zij de berekeningen, die met deze DSL geschreven zijn, goed kunnen lezen. Ditzelfde gold ook voor de business analisten. Beginnende Scala-ontwikkelaars waren bovendien prima in staat om zelf berekeningen te wijzigen en zelfs op te stellen. En omdat Scala op de JVM draait, konden we eenvoudig de test suite van de bestaande Java-implementatie loslaten op onze Scala implementatie, waarmee de correctheid geborgd is. Hiermee hebben we dus de doelen gehaald, die we onszelf aan het begin gesteld hebben.

Het eindresultaat van de berekening is te zien in **Listing 2**. De magic number `1200`, die we aan het begin zagen, is nu verdwenen. Dit bleek de 12 te zijn van het aantal maanden in een jaar en de `100` van het percentage. De DSL heeft ervoor gezorgd dat we deze ondoorgrondelijke waarde niet meer nodig hebben.

In de volgende Java Magazine, vertellen mijn collega's wat we hebben gedaan om te zorgen dat de analisten de berekeningen niet alleen kunnen lezen, maar zelfs ook zelf schrijven. ■

HET TYPE CLASS PATTERN IS POLYMORFISME ZONDER INHERITANCE

REFERENTIES

[1] **http://www.danielwestheide.com/blog/2013/02/06/the-neophytes-guide-to-scala-part-12-type-classes.html**