In de beperkingen toont zich de meester

Het unieke nut van klassen

Henk Jan Pels

Beperkingen zijn het enige middel dat klassen hebben om eigenschappen aan objecten op te leggen. Beperkingen zijn nodig voor het aanbrengen van onderscheid tussen zin en onzin. Er is maar één middel om een object te onderwerpen aan de regels van een klasse: maak het object lid van een bepaalde klasse.

Objecten hebben geen klasse nodig om te kunnen bestaan. Dat betoogde ik in het vorige inleidende artikel. Betekent dit dat ik een klassenloze informatiemaatschappij nastreef? Zeker niet, want klassen zijn een onmisbare vorm van abstractie om orde te scheppen in de informatiechaos om ons heen. Het betekent wel dat ik niet wil leven met een model waarin elk object wordt gecreëerd in een klasse waar het vervolgens nooit meer uit kan.

KLASSEN ALS MIDDEL

In mijn benadering is elk object uniek en leidt zijn eigen bestaan. Klassen zijn een middel om op een iets hoger abstractieniveau te kunnen spreken over groepen objecten met overeenkomstige eigenschappen. Afhankelijk van het standpunt van waaruit je naar een object kijkt, zijn andere eigenschappen relevant en daarom moet je het door de bril van verschillende klassen kunnen beschouwen. Klassen maken het bovendien mogelijk om programma's te schrijven die bepaalde operaties op objecten uitvoeren, zonder dat de programmeur die objecten vooraf hoeft te kennen.

Eigenlijk hebben we maar drie dingen nodig om een conceptueel databasemodel te beschrijven:

- een database met de verzameling gegevens;
- een schema met de verzameling beperkingen;
- een lidmaatschapsrelatie met de verzameling lidmaatschappen.

HET SCHEMA

Een object bestaat omdat het voorkomt als eerste coördinaat van een gegeven in de database. Een beperking bestaat omdat hij voorkomt in het schema. Een klasse bestaat omdat hij genoemd wordt als eerste coördinaat in een beperking, of als tweede in een lidmaatschap.

Het schema is de verzameling van alle beperkingen. Voor een conceptueel model doet het er niet zoveel toe hoe die genoteerd worden. Het zou zelfs in natuurlijke taal kunnen, zij het dat het lastig is om daarin eenduidig te zijn. Bovendien moet een schema overzichtelijk zijn. Grafische notaties zijn overzichtelijk, en ook UML biedt een grafische notatie. Een nadeel van plaatjes is dat overzichtelijkheid en volledigheid niet altijd samengaan. Ook de UML-plaatjes kunnen maar een deel van alle mogelijke beperkingen uitdrukken. Voor de rest is er de object constraint language (OCL). Om het conceptuele model te kunnen gebruiken als discussiemiddel, hebben we een middel nodig om beperkingen in tekst uit te drukken, maar dan wel op een bondige manier. Daarom hebben we ervoor gekozen beperkingen te noteren als 4-tupels, net als gegevens. Ter onderscheid van de gegevens gebruiken we daarbij dubbele haken:

```
<<klasse, beperkingtype, beperkingexpressie,
gesteld>>
```

De eerste coördinaat geeft aan op welke klasse de beperking van toepassing is. De tweede, het beperkingtype, geeft aan hoe de

Modellering van scratch af aan (2)

Nieuwe producten en de productielijnen om die te maken, worden in 3D gevisualiseerd, lang voordat ze fysiek gemaakt worden. Verder worden producten in steeds meer 'smaken' en 'kleuren' aangeboden. Om die variëteit te kunnen beheersen, moeten varianten in productfamilies worden geklasseerd. Conceptuele datamodellen zijn goed bruikbaar bij de communicatie over zulke abstracte zaken, mits zij voldoen aan de nieuwe eisen die aan ze gesteld worden. Ze moeten met name meer steun bieden bij het vormen van abstracties.

beperkingsexpressie geïnterpreteerd moet worden. Wordt OCL als beperkingstype aangegeven, dan is de beperking in de derde coördinaat in OCL genoteerd. Voor veel voorkomende beperkingtypen, zoals attributen, sleutels en associaties, introduceren we speciale verkorte notaties. Stel dat we een klasse willen onderscheiden waartoe objecten behoren die een persoon representeren waarvan de haarkleuren komen uit de verzameling {blond, bruin, grijs, rood}, dan is de betreffende beperking te noteren als:

```
<<persoon, OCL, self.haarkleur ( {blond, bruin,
grijs, rood}, 1>>
```

Met de vierde coördinaat kan, net als bij gegevens, worden aangegeven of een beperking expliciet door een gebruiker is vastgesteld, of dat hij van andere beperkingen is afgeleid. Hierdoor wordt het mogelijk net te doen alsof alle afleidbare beperkingen vanzelf ook in het schema staan. Dat blijkt vooral handig bij uitleggen van semantiek, zoals aan het einde van deze aflevering zal blijken. Vaak wordt de semantiek van bepaalde notaties gekarakteriseerd door de beperkingen en gegevens die erdoor zijn af te leiden.

Voor vaak voorkomende beperkingen is OCL niet de meest leesbare vorm. Door het onderscheiden van verschillende beperkingtypen wordt het mogelijk per type een handige, compacte, leesbare notatie te gebruiken. In het vervolg zullen we verschillende beperkingtypen introduceren.

LIDMAATSCHAP EN ATTRIBUTEN

Aangezien objecten niet vanzelf tot een klasse behoren, moeten ze er expliciet aan worden toegekend. Ook dat gebeurt met een tupel. In dit geval volstaan drie coördinaten:

```
<(object, klasse, gesteld)>
```

Zo'n tupel noemen we een lidmaatschap en de verzameling van alle lidmaatschappen heet de lidmaatschapsrelatie. Zo drukt bijvoorbeeld het lidmaatschap:

```
<(jan, persoon, 1)>
```

uit dat object jan instance is van de klasse persoon. In de vorige aflevering waren al een paar attributen aan object jan toegekend.[1]

Hoewel we in ons model vrij zijn willekeurige attributen aan willekeurige objecten toe te kennen, door simpelweg een gegeven met een waarde voor dat attribuut te stellen, is het toch vaak gewenst dat de objecten van een bepaalde klasse tenminste bepaalde attributen hebben. Willen we bijvoorbeeld dat elke persoon precies één waarde heeft voor het attribuut achternaam, dan wordt dat bondig genoteerd als:

```
<<persoon, attribuut, achternaam, 1>>
```

Zijn meer waarden toegestaan voor één attribuut, dan wordt dat bijvoorbeeld genoteerd als:

```
<<persoon, attribuut, haarkleur[1..*], 1>>
```

Willen we ook nog het domein aangeven waaruit de waarden gekozen kunnen worden, dan noteren we bijvoorbeeld:

```
<<persoon, attribuut, haarkleur[1..3]: {blond,
bruin, grijs, rood}, 1>>
```

Omdat gegevensmodelleerders bekend staan om hun luiheid -en ook schema's overzichtelijk willen houden- proberen we steeds voor de meest voorkomende beperkingen de kortst mogelijke notatie te kiezen. Vandaar dat de kortste (zonder vermelding van het aantal toegestane waarden) geldt voor de situatie waarin precies één waarde vereist wordt. Die situatie komt namelijk het meest voor. Helemaal consequent is deze keuze niet, want een ander principe is dat beperkingen alleen gelden als ze expliciet genoemd zijn of uit andere beperkingen zijn af te leiden. Eigenlijk zouden we dus moeten schrijven: <<persoon, attribuut, achternaam[1], 1>>.

Eveneens uit luiheid kiezen we voor nog een inconsequentie. In de vorige aflevering is gesteld dat de puntnotatie de verzameling waarden oplevert. Ook als voor object jan één voornaam gegeven is, bijvoorbeeld:

```
<jan, voornaam, 'jan', 1>
```

levert de puntnotatie een verzameling:

```
jan.voornaam = {'jan'}
```

Als er echter een beperking is die vereist dat er maar één waarde is, zoals voor het attribuut achternaam, spreken we af dat de puntnotatie die enkele waarde oplevert. Hierboven is het object jan tot lid van de klasse persoon verklaard. Bovendien geldt voor die klasse dat achternaam precies één waarde heeft. Daarom gaat het volgende op:

```
jan.achternaam = 'jansen'
```

Nog een veel voorkomende beperking is de sleutel. Ook daarvoor is gemakkelijk een notatie te bedenken:

```
<<persoon, sleutel, {naam}>>
```

betekent dat geen twee instances van de klasse persoon dezelfde waarde voor attribuut naam mogen hebben.

ASSOCIATIES

Associaties zijn zonder meer het belangrijkste structureringsmiddel in databases. In UML wordt een associatie voorgesteld als een binaire relatie tussen twee klassen. Wij streven ernaar ons model zo eenvoudig mogelijk te houden. We hebben gesteld dat een conceptueel databasemodel bestaat uit een database, een schema en een lidmaatschaprelatie. Daarin passen geen binaire relaties. Gelukkig hebben we die ook niet nodig. Twee objecten zijn te koppelen door in een elementair gegeven de één als attribuutwaarde aan de ander toe te kennen. We spreken dan van een link (of schakel, voor de puristen). Stel er is een adres a1 met de volgende attributen:

```
<ad1, straat, 'catslaan', 1>, <ad1, nummer, 12,
1>, <ad1, plaats, 'daalhoven', 1>
```

dan kunnen we dat adres aan jan toekennen met de link:

```
<jan, adres, a1, 1>
```

Een associatie kan nu eenvoudig worden gezien als een beperking die vereist dat een bepaald attribuut altijd een link is. Stel, we creëren naast persoon de klasse adres:

```
<<adres, attribuut, straat, 1>>, <<adres, attri-
buut, nummer, 1>>, <<adres, attribuut, plaats, 1>>
```

met een instance a1:

```
<(a1, adres, 1)>
```

Dan betekent de associatie:

```
<<persoon, associatie, adres[1], 1>>
```

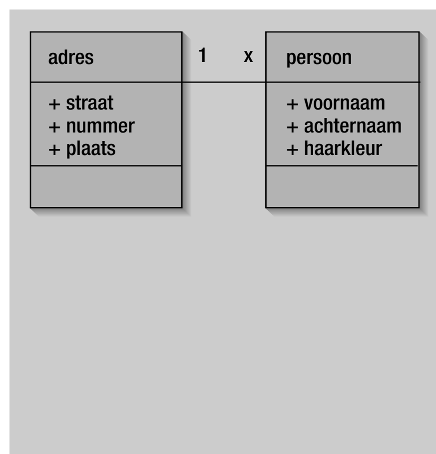
dat elke persoon precies één adres moet hebben. De kardinaliteit van de associatie is op de bekende manier tussen vierkante haken aan te geven met een onder- en bovengrens. Geen kardinaliteit betekent geen beperking, dus [0..*]. Figuur 1 toont de UML-notatie. Staat in de associatie niets anders vermeld, dan krijgt het linkattribuut eenvoudig de naam van de andere klasse.

Elke associatie heeft twee kanten. Als jan gelinkt is met adres a1, volgt daaruit dat omgekeerd adres a1 gelinkt is met jan. Dit stukje semantiek wordt uitgedrukt door een afgeleid gegeven in de database:

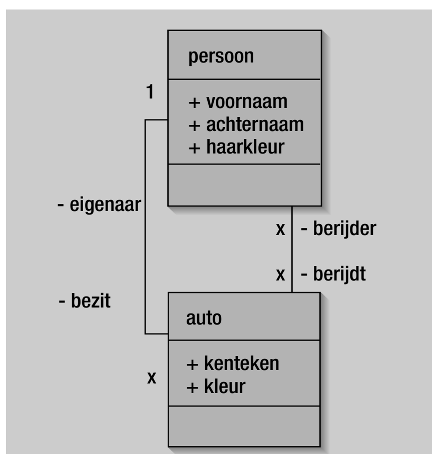
```
<a1, persoon, jan, 0>
```

Hier blijkt voor het eerst het nut van de vierde coördinaat. Daarmee kunnen we aangeven dat de omgekeerde link wordt geacht impliciet in de database aanwezig te zijn. Evenzo is er impliciet een associatie van adres naar persoon:

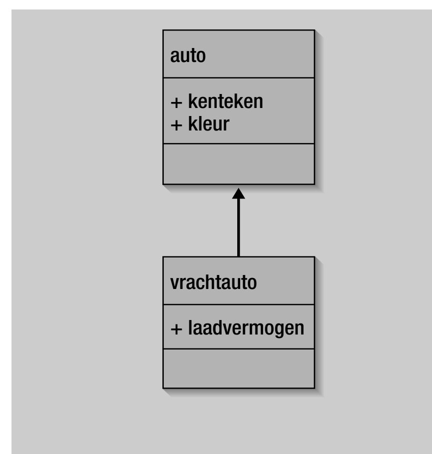
```
<adres, associatie, persoon, 0>
```



Figuur 1: UML-diagram met eenvoudige associatie.



Figuur 2: Een associatie met rolnamen in UML.



Figuur 3: Generalisatie in UML.

Omdat niets bekend is over een beperking van de kardinaliteit, wordt hier niets aangegeven.

Soms zijn tussen twee klassen meerdere associaties nodig. Dan volstaat het niet het linkattribuut de naam van de andere klasse te geven, maar moeten expliciet namen worden toegekend. Stel dat er in de te modelleren wereld ook auto's zijn en dat personen die auto's kunnen bezitten of berijden. Vijn twee auto's zijn de volgende gegevens bekend:

```
<au1, kenteken, 67-GG-HH, 1>, <au1, kleur, rood, 1>, <au1, eigenaar, jan, 1>, <jan, berijdt, au1, 1>
<au2, kenteken, 67-43-GG, 1>, <au2, kleur, geel, 1>, <jan, bezit, au2, 1>
<(au1, auto, 1)>, <(au2, auto, 1)>
```

Associaties met expliciete rolnamen (end-names in UML), zoals weergegeven in het UML-diagram van figuur 2, zijn nu te specificeren met:

```
<<auto, associatie, eigenaar[1]: persoon omgekeerd bezit, 1>>
<<auto, associatie, berijder: persoon omgekeerd berijdt, 1>>
```

Deze associatie betekent dat elke auto één eigenaar moet hebben en dat die eigenaar de auto bezit. Daarbij kan een persoon nul of meer auto's bezitten. Deze associatie mag aan het schema worden toegevoegd, omdat de huidige databasetoestand eraan voldoet: elke auto heeft een eigenaar - mede omdat de associatie de volgende afgeleide gegevens in de database doet verschijnen:

```
<jan, bezit, au1, 0>, <au1, berijder, jan, 0>, <au2, eigenaar, jan, 0>
```

Zo opgevat zijn associaties niet meer dan gewone beperkingen met een bijzondere semantiek: ze impliceren de omgekeerde associatie, en de bijbehorende links doen impliciet omgekeerde links in de database ontstaan. UML gebruikt binaire relaties om links voor te stellen. Dat is in feite een overbodige constructie in het model. Ook meerwaardige associaties en associaties met attributen zijn eigenlijk overbodig: die kunnen altijd met aparte klassen worden voorgesteld.

GENERALISATIE

Ten slotte nemen we nog even de generalisatie onder de loep. Die dient om aan een klasse subklassen met speciale eigenschappen te kunnen toevoegen. De subklasse erft dan de eigenschappen van de superklasse. Typisch voorbeeld is de vrachtauto die wordt gemodelleerd als een auto met de speciale eigenschap laadvermogen. In UML laat dat zich beschrijven met figuur 3. In de tekstuele notatie schrijven we in plaats hiervan:

```
<<vrachtauto, superklasse, auto, 1>>
```

Dit laat zich lezen als "vrachtauto heeft superklasse auto". En het betekent dat vrachtauto alle beperkingen van auto erft, dus ook de associaties:

```
<<vrachtauto, associatie, eigenaar[1]: persoon omgekeerd bezit, 0>>
<<vrachtauto, associatie, berijder: persoon omgekeerd berijdt, 0>>
```

Voor de instances van vrachtauto betekent het dat ze impliciet lid van auto zijn. Stel, we beschrijven een derde voertuig met:

```
<au3, kenteken, 22-VV-33, 1>, <au3, eigenaar, jan, 1>, <au3, laadvermogen, 2000, 1>
```

en kennen die vervolgens toe aan de klasse vrachtauto met:

```
<(au3, vrachtauto, 1)>
```

Dan volgt op grond van de generalisatie:

```
<(au3, auto, 0)>
```

CONCLUSIE

Klassen zijn te creëren door beperkingen ervoor te definiëren. Objecten worden eerst gecreëerd door attribuutwaarden eraan toe te kennen. Vervolgens kunnen ze aan bepaalde klassen worden toegewezen, mits ze voldoen aan alle beperkingen van die klasse. Gegevens en beperkingen kunnen we eenvoudig opnemen tussen onze tekst. Daardoor kunnen we gemakkelijk over conceptuele modellen schrijven. Structuurdiagrammen blijven we gewoon gebruiken als illustratie bij of als aanvulling op de tekst. De afgeleide gegevens, beperkingen en lidmaatschappen blijken een handig middel om de semantiek van bepaalde beperkingen in het model uit te drukken.

In het volgende artikel zullen we zien hoe we de mogelijkheden van klassen kunnen uitbreiden, door objecten als instance toe te wijzen aan andere objecten. Dat lijkt misschien vreemd, maar nergens staat dat het verboden is. Dus waarom niet? ●

Noot:

1. Aangezien deze reeks als doorlopend verhaal is bedoeld, blijven die gegevens gelden in deze en volgende afleveringen. In plaats van de vreemde term 'is instance van' wordt ook wel gezegd: 'jan behoort tot de klasse persoon', 'jan is lid van de klasse persoon' of 'jan is een persoon'.

Henk Jan Pels (H.J.Pels@tm.tue.nl) is werkzaam aan de Faculteit Technologie Management, Capaciteitsgroep Informatie & Technologie, van de Technische Universiteit Eindhoven.