
Een objectgerichte taal gebaseerd op LISP

Koenraad De Smedt

Objecten zijn voorstellingen van afzonderlijke entiteiten in een computermodel van de werkelijkheid. De kennis over deze objecten kan bestaan uit data, maar ook uit procedures die toepasbaar zijn op deze objecten. In *objectgerichte* (of objectgeoriënteerde) talen worden procedures en data dan ook ingekapseld in objecten. Door middel van *erving* kunnen objecten kennis delen met andere. Op deze manier kan men nieuwe objecten creëren als specialisaties of combinaties van andere objecten. Tevens ondersteunt erving een manier van programmeren door verfijning en draagt het bij tot het vermijden van redundantie. De programmeertaal *CommonORBIT* is een objectgerichte uitbreiding van Common LISP. De kenmerken van deze taal worden vergeleken met die van andere objectgerichte talen om zo tot een genuanceerd overzicht te komen van enkele architecturen binnen het objectgerichte paradigma.

1 Het objectgerichte programmeerparadigma

Wanneer een kennistechnoloog wordt geconfronteerd met de taak, een eerste aanzet te geven voor een complex kennissysteem, dan zal hij vaak de soorten van entiteiten in het probleemdomen op een rijtje zetten. Deze *objecten* kunnen zowel abstracte ideeën als concrete dingen zijn. In een expertsysteem voor medische diagnose kunnen deze objecten bijvoorbeeld bestaan uit patiënten, ziekten, bacteriën, symptomen, enzovoorts. In een systeem voor de verwerking van natuurlijke taal kunnen de objecten bestaan uit zinnen, woorden, klanken, enzovoort. In een gebruikersinterface kunnen iconen, vensters, enz. voorkomen, en ook kunnen de computer en de gebruiker zelf als objecten worden voorgesteld.

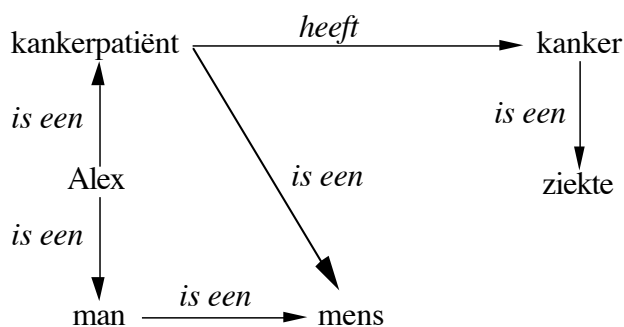
Objecten zijn gestructureerd: ieder object heeft een aantal *aspecten* die kennis bevatten over het object. Een *patiënt* heeft bijvoorbeeld *symptomen*, een *medische achtergrond*, een *bloeddruk*, enz. Bovendien heeft een patiënt een aantal aspecten die ook gelden voor personen in het algemeen, zoals een *naam*, *geslacht*, *gewicht*, *leeftijd*, enz. Het object *patiënt* kan deze aspecten *erven* van met het object *persoon*. Sommige aspecten, bijvoorbeeld *naam*, bevatten statische kennis en zouden kunnen worden opgeslagen als in een databank. Maar andere aspecten, bijvoorbeeld *bloeddruk*, zijn dynamisch en kunnen voortdurend veranderen tijdens een computersimulatie. Sommige aspecten

kunnen worden afgeleid van andere kennis, bijvoorbeeld de *leeftijd* van een persoon op een zeker moment kan worden afgeleid van de *geboortedatum*.

Objectgerichte talen bieden een representatie waarbij deze objecten stuk voor stuk worden voorgesteld en gemanipuleerd. Objecten worden tijdens de loop van het programma aangemaakt en gewijzigd. Zij zijn afzonderlijke computationele eenheden: ieder object bevat de kennis die relevant is voor een entiteit in de wereld. Deze kennis vormt de interne toestand van een object.

Moon (1986) beschouwt objectgericht programmeren als een techniek om zeer grote programma's te organiseren. Het distribueren van zowel declaratieve als procedurele kennis in hanteerbare eenheden maakt complexe systemen overzichtelijk. Objecten zijn een *uniforme* manier om kennis te structureren. Programma's worden geleidelijk opgebouwd door bestaande objecten uit te breiden of te hergebruiken door middel van erving. De organisatie van kennis in objecten staat ook dicht bij de manier waarop wetenschapslui denken over hun wetenschapsdomein. In een grammaticaboek vindt men bijvoorbeeld een apart hoofdstuk over *zelfstandige naamwoorden*, een hoofdstuk over *werkwoorden*, enz. We kunnen die organisatie nabootsen met een object *zelfstandig naamwoord*, een object *werkwoord*, enz. Deze organisatie helpt om de stap van een theorie naar een computerprogramma te vergemakkelijken.

Objectgerichte kennisrepresentatie behoort tot een familie van kennisrepresentatietalen die ook *frames* en *semantische netwerken* omvat. Deze formalismen zijn erop gericht om kennis in ietwat grotere eenheden te formuleren dan bijvoorbeeld *productieregels*. Semantische netwerken stellen kenniseenheden in een lange-termijn-geheugen grafisch voor als knopen; associatieve relaties tussen deze eenheden worden voorgesteld met pijlen (*arcs*). Pijlen met een 'is een' relatie stellen specialisatierelaties voor waarlangs kennis kan worden geërfd (Brachman, 1979). Figuur 1 is een voorbeeld van een semantisch netwerk, waaruit bijvoorbeeld kan worden afgeleid dat *Alex* een *ziekte* heeft.



Figuur 1:
Een semantisch netwerk

Frames werden naar voren gebracht door Minsky (1975) met de bedoeling een context te scheppen voor kennisverwerking. In de kennisrepresentatietalen die later op dit idee zijn gebaseerd lijken frames eigenlijk heel sterk op gestructureerde objecten. De verschillende kennisaspecten van een frame worden gewoonlijk *slots* genoemd. In sommige op frames gebaseerde systemen bevatten deze slots alleen gegevens en geen procedures. Als frames ook procedures kunnen bevatten—wat standaard is in

objectgerichte talen—dan spreekt men van *procedural attachment* (Bobrow & Winograd, 1977).

In het volgende worden de fundamentele concepten van objectgericht programmeren uiteengezet. Binnen dit paradigma bestaan nog verschillende benaderingen, en het zou te ver voeren om een encyclopedisch overzicht daarvan te geven. De discussie beperkt zich hoofdzakelijk tot objectgerichte uitbreidingen van LISP. In het bijzonder worden verschillende manieren waarop objecten kennis delen met elkaar vergeleken. Ook worden verschillende manieren om objecten te activeren—door middel van boodschappen of generische functies—met elkaar vergeleken.

2 Erving

Een belangrijke eigenschap van objectgerichte talen is dat ze voorzien in een mechanisme waarmee objecten zowel procedurele als declaratieve kennis kunnen *erven* van andere objecten. Zo'n ervingsrelatie betekent dat een object in principe alle kennis in een algemener object kan gebruiken—behoudens uitzonderingen.

2.1 Waarom erving?

Het nut van erving in objectgericht programmeren kan verdedigd worden vanuit verschillende gezichtspunten. Enkele belangrijke redenen voor erving zijn de volgende: 1 *Specialisatie*. Een object kan worden gedefinieerd als een specialisatie (of subtype) van een ander, algemener object. Zo kan een *specialisatienetwerk* van concepten worden gevormd. Er kan door herhaalde specialisatie bijvoorbeeld een taxonomie van diersoorten worden opgesteld: de *mens* en de *aap* zijn specialisaties van het object *primaat*, dat zelf weer een specialisatie is van *zoogdier*, enz.

2 *Combinatie*. Een ander gebruik van erving is om de eigenschappen van verschillende objecten te combineren. Als bijvoorbeeld Alex een mannelijke patiënt is, dan worden de eigenschappen van *man* en *patiënt* gecombineerd. Wanneer van meerdere objecten waarvan wordt geërfd, spreekt men van *meervoudige erving*. Men moet hierbij wel opletten voor het feit dat een objectgerichte taal de eigenschappen van gecombineerde objecten simpelweg bij elkaar voegt. Een object voor *klein sterrenstelsel* is echter niet zomaar voor te stellen als de combinatie van *klein* en *sterrenstelsel*; ook is *speelgoedhijskraan* niet een eenvoudige combinatie van *speelgoed* en *hijskraan*.

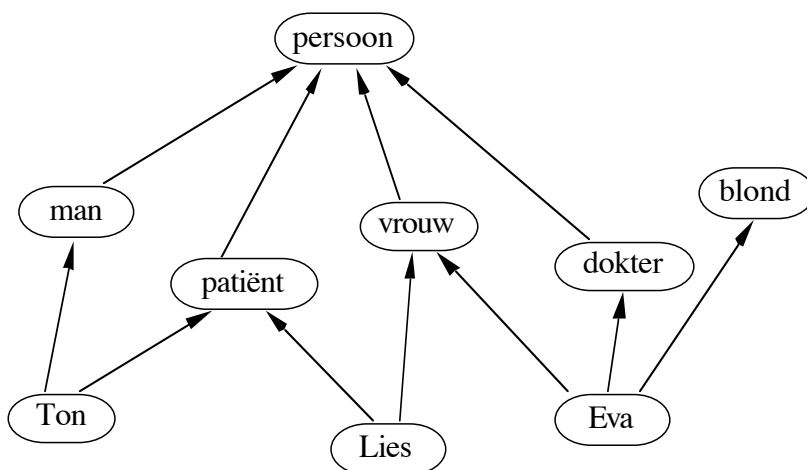
3 *Stapsgewijze verfijning*. Een programma kan worden geschreven door eerst de meest eenvoudige objecten in het domein te definiëren en dan de meer ingewikkelde aan te pakken als extensies van de eenvoudige. De programmeur denkt hierbij niet zozeer aan het opbouwen van een taxonomie, maar aan stapsgewijze verfijning door specialisatie als een techniek voor het ontwikkelen van software. Dit kan enigszins worden vergeleken “stepwise refinement by decomposition” (Wirth, 1971). Het is significant dat de inspanning om een object te definiëren evenredig is met de mate waarin het verschilt van andere objecten. Door deze vaststelling komt stapsgewijze verfijning door specialisatie neer op het cognitieve principe van de ‘minste moeite’ (*principle of least effort*).

4 *Vermijden van redundantie.* De opslag van kennis wordt efficiënter door erving doordat *redundantie* (het meermaals voorkomen van dezelfde informatie) wordt vermeden. Een stuk informatie dat hoort bij verscheidene objecten hoeft slechts bij één ervan te worden opgeslagen. Dit vergroot bovendien de modulariteit omdat de geërfde informatie bij veranderingen op slechts één plaats hoeft te worden gewijzigd.

2.2 Prototypen versus klassen

Twee architecturen voor specialisatie worden onderscheiden. De eenvoudigste bestaat hierin, dat objecten *model* kunnen staan voor andere objecten. Men zegt ook wel dat objecten als *prototypen* (of *stereotypen*) kunnen fungeren voor andere objecten. CommonORBIT is gebaseerd op prototypen en gebruikt hiervoor de metafoor van *delegeren* (*delegation*): een object (in CommonORBIT de *client*) kan zich laten vertegenwoordigen door een ander (de vertegenwoordiger of *proxy*). Als een object een boodschap krijgt waarop het niet weet hoe te reageren, dan delegeert het object de boodschap aan zijn vertegenwoordiger (of prototype): “Kan jij dit voor mij afhandelen?”. In termen van (LISP) functies zou dit kunnen betekenen: “Ik heb geen definitie voor deze functie, heb jij er een die ik mag gebruiken?” Een object kan verschillende prototypes hebben, die elk verschillende definities bijdragen.

Een specialisatienetwerk is voor deze architectuur een gerichte graaf waarvan de knopen objecten voorstellen die gerelateerd zijn door de specialisatielatie. De relatie “*x* is een prototype voor *y*” wordt geschreven als $x \rightarrow y$. Om circulariteit te vermijden moet de graaf acyclisch zijn. Een voorbeeld van een specialisatienetwerk wordt getoond in figuur 2.

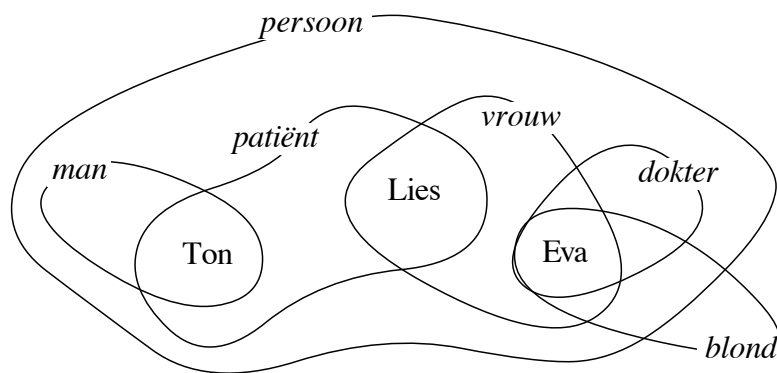


Figuur 2:
Schematisch voorbeeld van een specialisatienetwerk van objecten

Een prototype kan verwijzen naar een typische situatie of een ideaal of stereotypisch object. Door andere objecten te vergelijken met het prototype kunnen gelijkenissen en verschillen aan het licht komen. Het is mogelijk dat geen enkel ander object in het domein *alle* eigenschappen van het prototype heeft. Andere objecten kunnen worden gedefinieerd door hun verschillen met het prototype aan te geven. Deze verschillen

kunnen uitzonderingen zijn. Bijvoorbeeld, een vogel kan vliegen, maar een struisvogel niet. De verschillen kunnen ook details zijn die helemaal niet aanwezig zijn in het prototype. Stel dat een zekere struisvogel een zwarte stip heeft op de linkerpoot, dan is dit bijkomende kennis, maar geen uitzondering.

Een totaal andere aanpak bestaat hierin, dat de structuur en het gedrag dat een aantal objecten gemeen hebben worden gedefinieerd in een *klasse* (ook wel een *type* of *flavor* genoemd). Ieder object moet een *instantie* zijn van een klasse, waarvan ze kennis erven. In tegenstelling tot prototypen zijn klassen echter zelf geen objecten. Een klasse kan worden opgebouwd uit een of meer andere klassen (*superklassen*) waarvan de kennis dan wordt verenigd in de nieuwe klasse. Een schematisch voorbeeld wordt gegeven in figuur 3.



Figuur 3:
Schematisch voorbeeld van een specialisatienetwerk met klassen en instanties

Door zijn uniformiteit is een architectuur gebaseerd op prototypen algemener en krachtiger dan een architectuur gebaseerd op het onderscheid tussen klassen en instanties. De laatste kan enkel 'is een' relaties uitdrukken tussen, omdat objecten zelf geen kennis van andere objecten kunnen erven. Prototypen laten dit wel toe; zij zijn gebaseerd op de 'is als' relatie, die algemener van aard is. Een prototype kan eenvoudig een klasse voorstellen, maar niet andersom. Klassen zijn immers geen volwaardige objecten; zij zijn enkel nuttig voor het creëren van instanties. Sommige talen, bijvoorbeeld SIMULA, laten zelfs niet toe dat er nieuwe klassen worden gecreëerd tijdens de uitvoering van het programma; klassen worden gedefinieerd bij het compileren en blijven daarna onveranderd.

3 Objectgerichte uitbreidingen van LISP

Als programmeerparadigma staat de objectgerichte aanpak haaks op andere classificaties van programmeertalen, bijvoorbeeld op het onderscheid tussen imperatieve, functionele en logische talen. Omdat een objectgerichte aanpak al deze stijlen kan aanvullen is niet zo verwonderlijk dat vele objectgerichte talen uitbreidingen zijn van diverse niet-objectgerichte talen. Sommige van de eerste objectgerichte talen werden geïmplementeerd als uitbreidingen van een imperatieve taal, bijvoorbeeld SIMULA (Dahl & Nygaard, 1966; Dahl, Myrhaug & Nygaard, 1971). Later werden

ook objectgerichte uitbreidingen van applicatieve en logische talen gerealiseerd. Voor uitbreidingen van LISP zijn er twee belangrijke architecturen, die nu worden besproken.

3.1 Message passing

Binnen het objectgerichte paradigma werden talen gebaseerd op *message passing* (het sturen van boodschappen) reeds vroeg een dominante groep. Voorbeelden zijn SMALLTALK (Ingalls, 1978; Goldberg & Robson, 1983) en ACT-1 (Hewitt, 1979). *Message passing* is een cognitieve architectuur waarbij ieder object wordt beschouwd als een zelfstandige verwerkingseenheid, soms een *actor* genoemd, die boodschappen kan ontvangen en tevens kan sturen naar andere objecten. Objecten hebben een aantal *methoden* waarmee ze kunnen reageren op ontvangen boodschappen. Objecten reageren op boodschappen door meer objecten aan te maken, meer boodschappen te sturen, en/of hun eigen interne toestand aan te passen.

De metafoor van *message passing* is zeer krachtig. Een systeem gebaseerd op het sturen van boodschappen kan zeer goed worden gerealiseerd in hardware door een netwerk van processoren aan te leggen. Een programmeur die schrijft in een message-passing-taal kan zich goed inleven in de objecten die hij definieert. Door message passing worden objecten namelijk verpersoonlijkt, zoals het volgende voorbeeld aangeeft:

“Als ik een Koorts-object ben, en ik krijg een boodschap die vraagt naar mijn Meest-Waarschijnlijke-Oorzaak, dan vraag ik mijn Temperatuur of die groter is dan 100, en ik vraag mijn Duur of die groter is dan 3 dagen. Als dat het geval is, dan antwoord ik met een Ernstige-Infectie. Anders antwoord ik met de Griep.” (vertaald uit Lieberman, 1987).

Om LISP uit te breiden met message passing moeten we beschikken over een operator om daadwerkelijk boodschappen te versturen. In FLAVORS (Weinreb & Moon, 1980) is dat bijvoorbeeld de SEND operator, zoals in het volgende voorbeeld:

```
(SEND PATIENT1 'SYMPTOMEN)
```

Ondanks de voordelen van message passing kan men aanvoeren dat een message passing operator als SEND een vreemd element is in een *applicatieve* taal als LISP. Een applicatieve taal is namelijk gebaseerd op de toepassing van functies op argumenten. De basistechniek voor het maken van nieuwe functies is abstractie en in principe is de definitie van nieuwe functies het enige neveneffect (McCarthy, 1960). In FLAVORS werd dit ten dele gerealiseerd door een object te zien als een soort van functie en SEND te implementeren als FUNCALL.

Een elegantere manier om het applicatieve karakter van LISP te behouden is het voorstel van Steels (1983) om de metafoor van message passing geheel te vermijden. In plaats daarvan wordt het begrip *functie* uitgebreid met *generische functies*. Op die manier kan een functie worden aangeroepen met een object als argument:

```
(SYMPTOMEN PATIENT1)
```

Generische functies hebben geresulteerd in het ontwerp van talen als ORBIT (Steels, 1983; De Smedt, 1984), CommonORBIT (De Smedt, 1987), New Flavors (Moon, 1986), CommonLoops (Bobrow e.a., 1985) en CLOS (Gabriel, 1987; Keene, 1989).

3.2 Generische functies

Een functie waarvan de definitie afhangt van het type van zijn argumenten wordt vaak *generisch*¹ (generic), *overloaded* of *polymorf* genoemd. Generische functies worden ingepast in het objectgerichte paradigma door de definitie niet te laten bestaan uit een enkel stuk programma maar hem te *verspreiden* over een aantal objecten. Net als een gewone LISP functie wordt een generische functie toegepast op argumenten, worden bepaalde operaties uitgevoerd en worden resultaten teruggegeven. In tegenstelling tot normale functies is de eigenlijke operatie die moet worden uitgevoerd niet opgeslagen in de functiedefinitie zelf, maar in de argumenten waarop de functie wordt toegepast. Generische functies zijn nuttig omdat ze toelaten om één enkele naam te gebruiken voor een aantal gerelateerde operaties op verschillende objecten. Dit maakt de interface eenvoudiger, omdat de differentiatie naar verschillende operaties automatisch wordt uitgevoerd op basis van de argumenten.

Stel dat men een tekstverwerker ontwerpt en er is een *high-level*-operatie nodig om delen van de tekst weg te halen. Uiteraard zijn er verschillende acties nodig naargelang het gaat om een letterteken, woord, alinea, enz. De interface kan worden vereenvoudigd door een generische functie DELETE die de correcte taak selecteert op basis van de teksteenheid. De conventionele manier om zo'n functie te schrijven is om uit te splitsen naar een aantal gevallen (met bijv. een COND of CASE statement):

Definitie van WEGHALEN:

Als het object een *letterteken* is, gebruik dan procedure *A*,
Als het object een *woord* is, gebruik dan procedure *B*, enz.

Stel dat we nog een functie toevoegen voor een andere operatie, dan moeten we opnieuw de verschillende gevallen beschouwen, bijvoorbeeld:

Definitie van VERPLAATSEN:

Als het object een *letterteken* is, gebruik dan procedure *C*,
Als het object een *woord* is, gebruik dan procedure *D*, enz.

Deze aanpak is typerend voor een *actiegerichte* stijl. Men denkt in termen van welke acties er zijn, analyseert de mogelijke gevallen, en definieert een actie voor elk geval. Bijgevolg is in een actiegerichte taal de kennis over iedere actie gegroepeerd, maar de kennis over een object verspreid over verschillende functies. Actiegerichte programmeertalen stimuleren deze aanpak door primitieven voor subroutines, conditionele uitdrukkingen, enz.

¹ 'Generisch' heeft in sommige talen een gerelateerde, maar toch andere betekenis. Generische subprogramma's in ADA zijn abstracties van gewone subprogramma's, die toelaten om te programmeren door middel van stapsgewijze verfijning.

In een objectgerichte aanpak gaat het precies omgekeerd. Generische functies worden gedefinieerd in de context van de objecten waarop zij van toepassing zijn, bijvoorbeeld:

Definitie van een LETTERTEKEN:

Gebruik procedure *A* om het object *weg* te *halen*,
Gebruik procedure *C* om het object te *verplaatsen*, enz.

Definitie van een WOORD:

Gebruik procedure *B* om het object *weg* te *halen*,
Gebruik procedure *D* om het object te *verplaatsen*, enz.

Als we deze aanpak gebruiken, dan beschouwen we eerst welke objecten er zijn in het domein, en groeperen dan de mogelijke acties voor elke soort. Bijgevolg kunnen definities van functies verspreid zijn over meerdere objecten. Objectgerichte talen ondersteunen deze aanpak door primitieven om objecten te creëren, er generische functies aan toe te voegen, enz. Het grote voordeel van generische functies is dat zij worden aangeroepen net als gewone functies, waardoor de uniformiteit van LISP gewaarborgd blijft.

3.3 Objecten en datatypen in LISP

Vele programmeertalen laten de definitie van abstracte gestructureerde datatypen toe. Door procedures zowel als data in te kapselen in datatypen kunnen deze datatypen worden gebruikt als objectklassen. De ingekapselde procedures worden dan getypeerde generische functies. Dergelijke uitbreidingen van LISP (bijvoorbeeld New Flavors, CommonLoops en CLOS) zijn goed geïntegreerd in de kern van de programmeertaal omdat het specialisatienetwerk een deel vormt van de hiërarchie van datatypen. KRS (Van Marcke, 1987) beschikt zelfs over *dataconcepten* die standaard LISP datatypen voorstellen op een objectgerichte manier. Een dergelijk gebruik van datatypen heeft als consequentie dat de objectgerichte taal een architectuur heeft die een onderscheid maakt tussen klassen (typen) en instanties.

Het gebruik van de typehiërarchie van de onderliggende programmeertaal is dan ook niet geschikt voor het implementeren van talen als CommonORBIT, die geen formeel onderscheid maken tussen klassen en instanties. Daarom voegen deze talen in principe maar één enkel datatype toe aan LISP, namelijk het gestructureerde type OBJECT. Alle objecten zijn van dit type, en specialisatierelaties worden onafhankelijk van de datatypehiërarchie opgeslagen. Generische functies in CommonORBIT zijn dan ook strikt genomen niet getypeerd, maar met ieder object kan een functiedefinitie geassocieerd zijn. Een object delegeert aan zijn prototypen wanneer het zelf geen definitie voorhanden heeft. Ook Objective-C (Cox, 1986), een taal gebaseerd op C, voegt maar één enkel type toe aan de onderliggende programmeertaal.

4 Objecten in CommonORBIT

Een object in CommonORBIT is geïmplementeerd als een Common LISP *structure* van het type OBJECT. Objecten hebben een aantal *aspecten* (slots) die kennis bevatten over het object en die geactiveerd worden als generische functies.

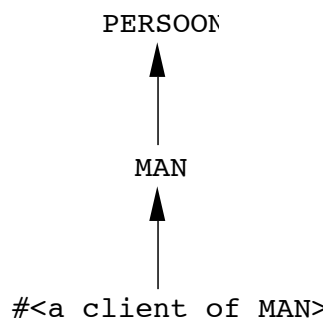
4.1 Objecten met namen en anonieme objecten

Objecten kunnen een naam hebben of anoniem blijven. Gewone objecten zijn anoniem: het zijn structuren die enkel zijn toegankelijk door ze te binden aan variabelen of als argumenten te passeren aan functies. Een object met een naam is bovendien ten allen tijde toegankelijk via een unieke identifier (een *symbol* in LISP). Objecten met een naam worden gedefinieerd met behulp van de DEFOBJECT macro, terwijl anonieme objecten worden gemaakt met behulp van A of AN. Enkele voorbeelden (—> geldt als *prompt*):

```
—> (DEFOBJECT MAN PERSOON
      (GESLACHT :VALUE 'MANNELIJK))
#<object MAN>

—> (SETQ X
      (A MAN
        (VOORNAAM :VALUE 'JANTJE)
        (GEBORTEJAAR :VALUE 1954)))
#<a client of MAN>
```

Het bovenstaande definieert drie objecten. Een object met de naam MAN heeft als prototype een object met de naam PERSOON (als dit laatste nog niet bestond, dan wordt het automatisch aangemaakt). De tweede expressie maakt een anoniem object met als prototype MAN en bindt dit anonieme object aan de variabele X. Het specialisatienetwerk ziet er op dit moment uit als in figuur 4.



Figuur 4:
Enkele objecten in CommonORBIT

4.2 Declaratieve en procedurele kennis in aspecten

Alle kennis over objecten wordt geactiveerd door middel van generische functies. Locale definities van generische functies worden *aspecten* genoemd in

CommonORBIT. Alhoewel in principe kan volstaan worden met het specificeren van een *closure* in een aspect, wordt terwille van het programmeergemak toch een onderscheid gemaakt tussen verschillende soorten van aspecten. Aspecten waarbij door middel van :VALUE een waarde wordt gespecificeerd drukken declaratieve kennis uit. Deze waarde wordt gewoon teruggegeven als resultaat van de generische functie, bijvoorbeeld:

```
→ (GEBORTEJAAR X)
1954
```

```
→ (GESLACHT X)
MANNELIJK
```

In de eerste functieaanroep werd de waarde voor de generische functie GEBORTEJAAR direct gevonden bij het object waarop de functie wordt toegepast. Bij de tweede functieaanroep werd de waarde voor de functie GESLACHT verkregen door te delegeren aan het prototype MAN.

Een tweede aspecttype is :FUNCTION, dat procedurele kennis uitdrukt. De inhoud van het aspect is een closure die wordt toegepast op de gegeven argumenten. Een voorbeeld wordt hieronder gegeven. Hieronder wordt een functie gedefinieerd voor het berekenen van de leeftijd die een persoon bereikt in een gegeven jaar door van dit jaar het geboortjaar van de persoon af te trekken.

```
→ (DEFOBJECT PERSOON
    (LEEFTIJD :FUNCTION
      #'(LAMBDA (ZELF JAAR)
          (- JAAR (GEBORTEJAAR ZELF))))))
#<object PERSOON>
```

```
→ (LEEFTIJD X 1992)
38
```

De gebruikelijke *lambda-binding* in LISP wordt ook hier gebruikt voor het toepassen van de closure op de argumenten van de generische functie. Alleen het eerste argument—hier X—hoeft een object te zijn; deze is immers de leverancier van de definitie voor de functie. De andere argumenten—hier het getal 1992—mogen tot willekeurige LISP datatypen behoren.

Enkele vergelijkingen met andere talen zijn nu aan de orde. Ten eerste is er in sommige objectgerichte talen (bijvoorbeeld SMALLTALK) een speciale (pseudo)variabele, meestal SELF genoemd, die naar het object verwijst. In CommonORBIT echter wordt gewoon gebruik gemaakt van de lambda-variabelen van de closure, die willekeurige namen mogen hebben.

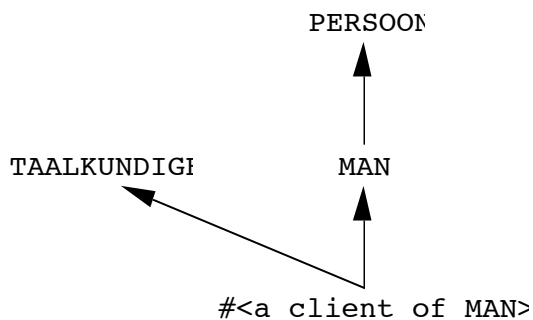
Ten tweede hebben talen als CommonLoops en CLOS het idee van generische functies verder gegeneraliseerd dan CommonORBIT. Zij selecteren namelijk een definitie op basis van *alle* argumenten, niet alleen het eerste. Op die manier zijn er geen geprivilegieerde argumentposities voor objecten. Bovendien kunnen generische functies dan ook worden gedefinieerd voor standaard LISP datatypen zowel als voor door de gebruiker gedefinieerde objecttypen.

Ten derde bieden de sleutelwoorden :VALUE en :FUNCTION in CommonORBIT een syntaxis voor het definiëren van verschillende soorten aspecten (en verderop zullen nog meer specifieke soorten aspecten worden geïntroduceerd). Alle aspecten worden echter geactiveerd middels generische functies. In sommige objectgerichte talen ligt dat anders. FLAVORS (en New Flavors) bijvoorbeeld maken een onderscheid tussen *instantievariabelen* waaraan declaratieve kennis wordt gebonden, en *methoden* voor procedurele kennis; alleen de laatste worden aangeroepen als generische functies, wat de interface helaas minder transparant maakt.

4.3 Kennis toevoegen aan bestaande objecten

Nadat een object is gedefinieerd kan men er dynamisch kennis aan toevoegen in de vorm van prototypen en aspecten. In het volgende voorbeeld wordt een prototype toegevoegd door middel van een IS-CLIENT operatie. Het effect op de eerder aangemaakte specialisatienetwerk wordt grafisch voorgesteld in figuur 5.

```
→ (IS-CLIENT X 'TAALKUNDIGE)
#<a client of TAALKUNDIGE and MAN>
```



Figuur 5:
Toevoegen van een prototype

Aspecten kunnen worden toegevoegd door middel van DEFASPECT. Deze operatie vereist argumenten voor de aspectnaam, het object, het aspecttype en de inhoud van de definitie. Hier volgen enkele voorbeelden:

```
→ (DEFASPECT ACHTERNAAM X
    :VALUE 'BETON)
    ACHTERNAAM

→ (DEFASPECT SCHRIJF-NAAM 'PERSOON
    :FUNCTION
    #' (LAMBDA (ZELF)
        (PRINC (VOORNAAM ZELF)
        (PRINC " ")
        (PRINC (ACHTERNAAM ZELF))
        T))
    SCHRIJF-NAAM
```

→ (SCHRIJF-NAAM X)
JANTJE BETON
T

Prototypen en aspecten kunnen tijdens de uitvoering van het programma eveneens worden verwijderd. De mogelijkheid om definities toe te voegen en ongedaan te maken geeft extra flexibiliteit in een experimenteel programma, of een programma dat tijdens de uitvoering steeds veranderende situaties modelleert. Ook hier moet worden benadrukt dat deze flexibiliteit niet door alle objectgerichte talen wordt geboden.

5 Erving van defaults

Het mechanisme waarbij objecten kennis delen is gebaseerd op *defaults* en niet op onweerlegbare proposities. Als we zeggen “vogels kunnen vliegen” dan bedoelen we “vogels kunnen normaal gesproken vliegen” of “de stereotiepe vogel kan vliegen”. We hebben geen problemen met uitzonderingen zoals struisvogels en pinguïns. Uitzonderingen kunnen niet goed worden behandeld in een systeem gebaseerd op predicaatlogica. Om te bewijzen dat een zekere vogel kan vliegen zou het immers nodig zijn om eerst te bewijzen dat het niet om een uitzondering gaat, dus dat het geen struisvogel is, geen pinguïn, enzovoort. Maar onze kennis is vaak beperkt. Wat we eigenlijk willen is dat we kunnen afleiden dat een vogel kan vliegen als het *niet* kan worden bewezen dat het om een uitzondering gaat (cf. Brewka, 1989). In objectgerichte talen gaat het om dit soort van redenering—*default reasoning*. Alle kennis die wordt verkregen door erving geldt slechts voor zover die niet wordt tegengesproken door kennis in het ervende object.

5.1 Erving door delegeren vs. erving door copiëren

Voor ieder systeem dat een deel van de ‘echte’ wereld voorstelt is verandering de grootste vijand. Immers, wanneer verschillende kennisbronnen van elkaar afhankelijk zijn dan maken veranderingen in één kennisbron vaak veranderingen in andere delen. Daarom is het belangrijk dat kennissystemen die worden gebruikt voor het modelleren van veranderende situaties, bijvoorbeeld kantoorssystemen, een efficiënte *updating* toelaten. Omdat objectgerichte talen onder meer de mogelijkheid bieden om een zeer grote specialisatienetwerk voor te stellen moet het ervingsmechanisme in staat zijn om veranderingen snel toegankelijk maken voor gerelateerde objecten in de hiërarchie. Dit is tevens een kwestie van *modulariteit*: een systeem is modulair in zoverre een verandering in één module het gedrag van een afhankelijke module niet verstoort.

Er zijn in wezen twee manieren om erving gestalte te geven. Ik gebruik de term *erving door copiëren* voor de techniek om naar een object defaultinformatie van zijn prototypen te copiëren. Deze techniek wordt ten dele gebruikt door de meeste systemen die zijn gebaseerd op klassen. Een klasse is als een mal die iedere instantie vormt door er zijn defaults in te stoppen. Als bijvoorbeeld een CLOS-object wordt gecreëerd, dan wordt een structuur aangemaakt met slots die alle defaultwaarden van de klasse bevatten. Ook in SMALLTALK wordt een datastructuur met instantievariabelen

gecreëerd bij iedere instantiatie. Het zal duidelijk zijn kopiëren niet zuinig is en dat een verandering van de defaults een grote updatingoperatie tot gevolg heeft².

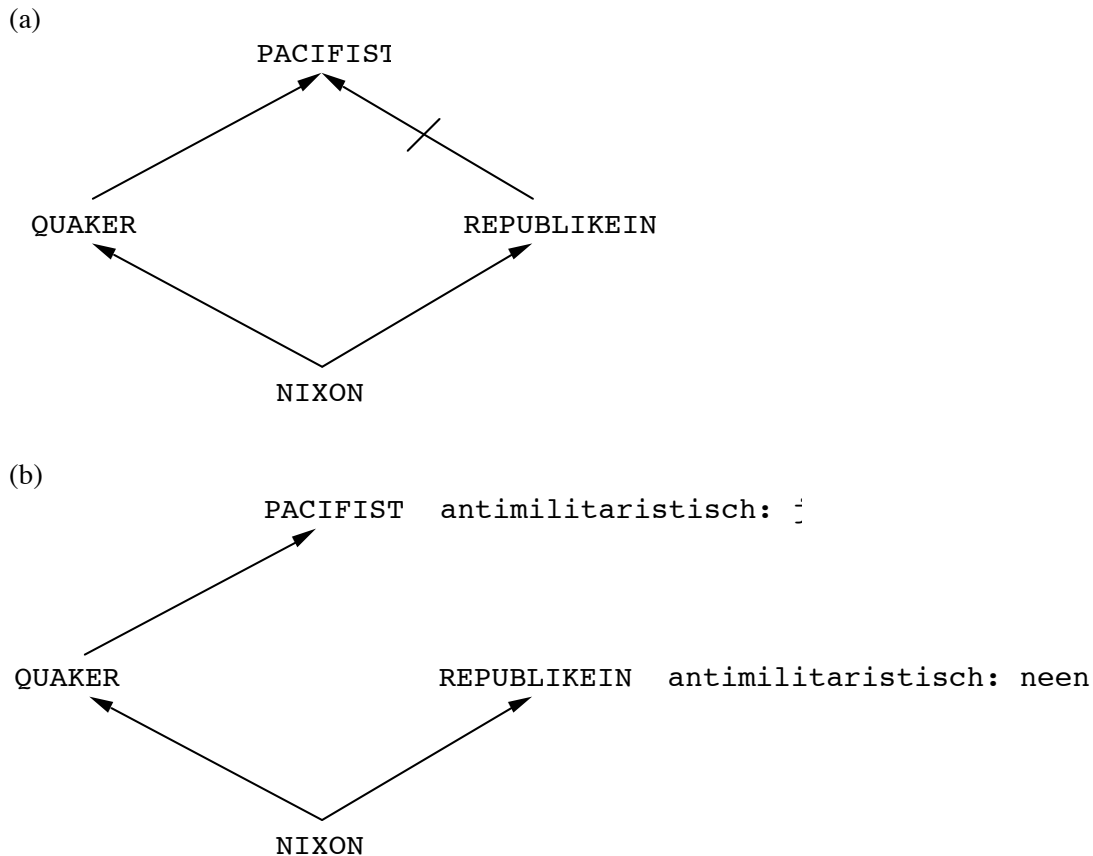
Bij *erving door delegeren* (ook wel *dynamische erving* genaamd) worden geen defaults gecopieerd, zodat het ervende object grotendeels leeg blijft. Delegeren is een techniek waarbij de defaults alleen worden opgevraagd als zij werkelijk nodig zijn. Dit is typerend voor systemen zoals CommonORBIT, die gebaseerd zijn op de notie van een prototype, model of vertegenwoordiger. Wanneer de aspecten van een object veranderen, dan zijn deze veranderingen direct toegankelijk voor de objecten waarvoor het een prototype is. Omdat er geen informatie gecopieerd werd is dus ook geen verdere updating noodzakelijk. Een systeem dat is gebaseerd op delegeren is dus ook modulairder. Overigens worden in systemen als CLOS die zijn gebaseerd op klassen, de *methoden*—in tegenstelling tot instantievariabelen—niet gecopieerd.

5.2 Meervoudige erving

In sommige objectgerichte talen kunnen objecten maar één prototype (of klasse) hebben. Het specialisatienetwerk vormt dan een boomstructuur. Andere systemen, zoals CommonORBIT, laten toe dat een object meerdere prototypen heeft. In dergelijke systemen vormt het specialisatienetwerk een gerichte acyclische graaf. Meervoudige erving laat een krachtiger kennisrepresentatie toe dan enkelvoudige erving. Het grote probleem is echter dat de verschillende prototypen van een object conflicterende informatie kunnen leveren. Hoe zulke conflicten moeten worden opgelost is een netelige vraag, waarop nu verder wordt ingegaan. In ieder geval is het raadzaam om alleen tamelijk onafhankelijke, niet conflicterende prototypen te combineren.

Het schoolvoorbeeld van conflicterende informatie in een specialisatienetwerk is de zogenaamde ‘Nixon ruit’ in figuur 6. Nixon is zowel een quaker als een republikein. Quakers zijn meestal pacifisten terwijl republikeinen dat meestal niet zijn. In een *bipolair* systeem (Touretzky, 1986; Touretzky, Horty & Thomason, 1987) dat zowel positieve (\rightarrow) als negatieve ($\emptyset, /$) verbindingen kent, kan deze kennis worden voorgesteld als figuur 6a. In een *unipolair* systeem, zoals CommonORBIT en de meeste objectgerichte talen, kan deze kennis worden voorgesteld als het blokkeren van een default (zie figuur 6b).

² CLOS biedt een optie aan om slots werkelijk te *delen*. Op die manier wordt kopiëren vermeden, maar dit is geen zuivere manier van erving omdat verandering van een slot in een instantie een verandering in de klasse tot gevolg heeft!



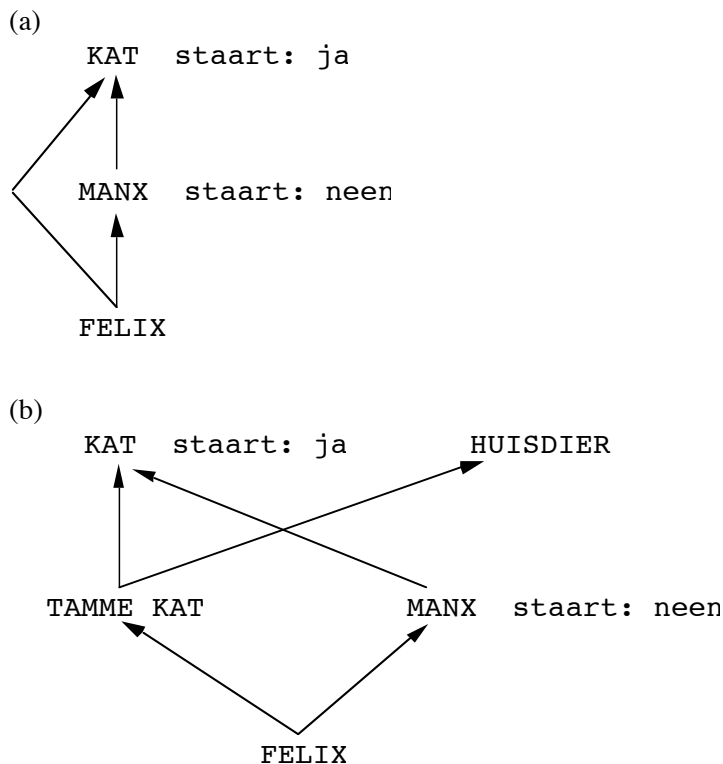
Figuur 6
De 'Nixon ruit'

Wat moeten we concluderen uit deze conflicterende informatie? Als we een skeptische houding aannemen, dan zouden we ons kunnen onthouden van een oordeel over het pacifisme van Nixon. Als we goedgegelovig zijn, dan zouden we twee standpunten kunnen innemen die allebei aanvaardbaar lijken. Een geheel andere oplossing, die wordt aangenomen in vele objectgerichte talen, bestaat uit het opstellen van prioriteiten tussen prototypen. De programmeur moet dus beslissen of het belangrijker is dat Nixon een republikein is dan dat hij een quaker is. Als we de conventie aannemen dat deze volgorde grafisch wordt voorgesteld van links naar rechts, dan zouden we uit figuur 6 kunnen concluderen dat Nixon meer een quaker is dan een republikein. Als we bovendien stellen dat de prioriteiten van prototypen ook geldt voor *hun* prototypen, dan is Nixon ook meer een pacifist dan een republikein. Op die manier wordt de hiërarchie in de diepte eerst doorzocht, en we concluderen dat Nixon antimilitaristisch is. Meervoudige erving in CommonORBIT wordt beschreven door de principes (1). De arbitraire volgorde opgelegd door (1b) is alleen noodzakelijk in een systeem als CommonORBIT, waar prototypen tijdens de uitvoering van het programma kunnen worden toegevoegd.

- (1) *Meervoudige erving:*

- a. (*Locale volgorde*) Een prototype dat meer aan het begin van een DEFOBJECT formule voorkomt heeft een hogere locale prioriteit dan een prototype dat later in die formule voorkomt.
- b. (*Recentheid*) Een prototype dat later wordt toegevoegd tijdens de uitvoering van het programma heeft een hogere locale prioriteit dan een prototype dat eerder was toegevoegd.
- c. (*Diepte eerst*) Als een object *a* direct erft van *b* en *c*, en *b* heeft een hogere prioriteit dan *c*, dan hebben alle prototypen van *b* ook een hogere prioriteit dan *c* en alle prototypen van *c*.

Meervoudige erving is dus een partiële volgorde op alle recursieve prototypen van een object. Omdat specialisatie zelf ook een partiële volgorde is op de recursieve prototypen, blijft de vraag, hoe deze twee relaties in elkaar passen om de uiteindelijke volgorde voor erving te bepalen. Zolang specialisatie en meervoudige erving elkaar niet tegenspreken is dat een eenvoudige zaak. Helaas is het niet moeilijk om voorbeelden te verzinnen van netwerken waarin specialisatie in tegenspraak is met de volgorde van meervoudige erving. Figuur 7a toont een *redundante* verbinding die een prototype ‘overslaat’; figuur 7b toont een voorbeeld zonder redundantie maar waarin een stricte diepte-eerst-zoekmethode toch de specialisatievolgorde onrecht aandoet.



Figuur 7
Specialisatie versus volgorde van meervoudige erving

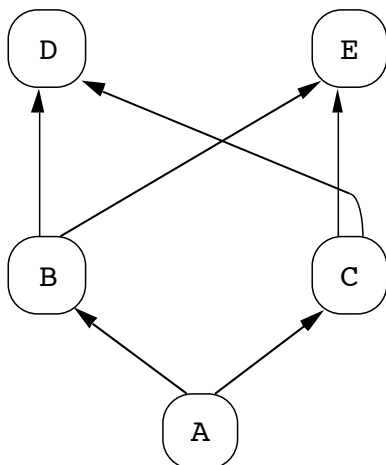
Moeten we het belang van lokale volgorde beperken ten gunste van de correcte specialisatievolgorde? Het wordt inderdaad algemeen aangenomen (bijvoorbeeld Touretzky, 1986) dat erving altijd tenminste volgens de partiële volgorde van specialisatie dient te verlopen. Dit principe, geformuleerd in (2), verhindert ‘kortsluiting’ in het specialisatienetwerk.

(2) *Specialisatie vs. volgorde van meervoudige erving:*

Erving moet altijd verlopen volgens de volgorde van de specialisatierelaties; daarom gaat specialisatie altijd vóór op de volgorde van meervoudige erving.

Benadrukt dient te worden dat dit principe volkomen arbitrair is. We konden net zo goed het omgekeerde kiezen; op die manier hadden we bijvoorbeeld redundante verbindingen als in figuur 7a kunnen gebruiken als een controlestructuur om *bewust* een kortsluiting te veroorzaken. Ook is principe (4) niet altijd in overeenstemming met de intuïtie. Het is niet moeilijk om een voorbeeld van een netwerk te bedenken dat isomorf is aan figuur 7b maar leidt tot een andere intuïtie omtrent volgorde (de lezer wordt aangemoedigd om het te proberen).

Verscheidene algoritmen die voldoen aan principe (2) kunnen worden geconstrueerd. Een efficiënt en tamelijk eenvoudig algoritme bestaat hierin, dat de graaf van prototypen recursief in de diepte eerst wordt doorzocht, maar dat hierbij die objecten worden overgeslagen die prototypen zijn van andere objecten die nog ‘in de wachtrij staan’. Zo blijft gegarandeerd dat de specialisatievolgorde wordt gevolgd. Jammer genoeg zijn er ook situaties denkbaar waarin lokale volgorde conflicteert met een andere lokale volgorde. Een voorbeeld wordt gegeven in figuur 8.



Figuur 8
Conflicterende lokale volgorden bij meervoudige erving

Het voorgestelde algoritme, dat wordt gebruikt in CommonORBIT, zal eerst de prototypen van B bezoeken en daarbij zowel D als E overslaan. Daarna wordt C bezocht en vervolgens E en D. Men kan opwerpen dat deze volgorde niet de plaatselijke volgorde in de meest linkse tak weerspiegelt. Het is mogelijk om de

‘correcte’ volgorde (A B C D E) te verkrijgen met een ander algoritme, dat echter complexer en tijdrovender is (Ducournau & Habib, 1987). Omdat hiërarchieën als die in figuur 8 echter atypisch en tegenintuïtief zijn, is het de vraag of dit de moeite waard is. Ik durf te besluiten dat het mechanisme voor meervoudige erving in CommonORBIT een praktisch systeem is dat overeenstemt met algemeen aanvaarde principes. Toch blijft het probleem van domeinafhankelijke intuïties onopgelost (cf. ook Touretzky, Horty & Thomason, 1987).

5.3 Het combineren van operaties

De eenvoudigste manier van erving bestaat hierin, dat voor een aspect slechts één definitie—de meest specifieke—wordt geërfd en dat alle andere buiten beschouwing blijven. Dit is normaal het geval in CommonORBIT en stemt ook het best overeen met het idee van *default*erving. Toch zijn er gevallen waar dit onvoldoende is. Soms wil men overgeërfd kennis net iets veranderen. Stel bijvoorbeeld dat een naam van een persoon normaal wordt gerepresenteerd als een *lijst* van voornaam en achternaam. Verder wordt de naam van een dokter net zo gerepresenteerd als dat van een ‘normale’ persoon, maar voorafgegaan door de letters “DR.”. Het aspect NAAM wordt dus wel overgeërfd, maar het resultaat wordt nog veranderd. In CommonORBIT wordt zo’n ‘op maat gemaakte’ erving geschreven met behulp van DELEGATE. Laten we eerst een object maken voor PERSOON:

```

-> (DEFOBJECT PERSOON
      (NAAM :FUNCTION
        #'(LAMBDA (ZELF)
            (LIST (VOORNAAM ZELF)
                  (ACHTERNAAM ZELF))))))
#<object PERSOON>

-> (NAAM (A PERSOON
          (FIRST-NAME 'RIA)
          (LAST-NAME 'WILBERS)))
(RIA WILBERS)

```

Nu definiëren we een object DOKTER met een aspect NAAM. De DELEGATE macro delegeert het aspect aan PERSOON en past de verkregen definitie lokaal toe op het argument ZELF³. Door middel van CONS wordt bij dit resultaat het symbool “DR.” toegevoegd.

³ In feite kan de verkregen definitie worden toegepast op willekeurige argumenten. Op die manier kan de functie bewust *niet* gelocaliseerd worden. Lieberman (1986) noemt deze mogelijkheid ‘echt’ delegeren, omdat het prototype de boodschap verwerkt *in de plaats van* een ander object.

```

-> (DEFOBJECT DOKTER
      (NAAM :FUNCTION
        #' (LAMBDA (ZELF)
          (CONS 'DR.
                (DELEGATE
                  (NAAM 'PERSOON)
                  ZELF) ) ) ) )

```

```
#<object DOKTER>
```

```

-> (NAAM (A DOKTER
          (VOORNAAM 'EVA)
          (ACHTERNAAM 'BLOM) ) )
(DR. EVA BLOM)

```

Het gebruik van DELEGATE vereenvoudigt het programma, niet alleen door het vermijden van het onnodig dupliceren van (LIST ...) maar ook doordat de definitie van DOKTER afhankelijk *blijft* van die van PERSOON. Het programma wint daardoor aan modulariteit. Als de definitie van PERSOON verandert, bijvoorbeeld door de voornaam voor de achternaam te plaatsen, dan erft DOKTER deze verandering automatisch.

Soms is het wenselijk om de kennis in meerdere objecten na elkaar te gebruiken. In dat geval kan door middel van DELEGATE een volgorde worden gespecificeerd. Stel bijvoorbeeld dat we een venstersysteem bouwen met definities voor vensters met *scroll bars* en vensters met titel. Stel dat we deze willen combineren in een VENSTER-MET-SCROLL-BARS-EN-TITEL die eerst een venster met scroll bar tekent en er dan een titel aan toevoegt. Een mogelijke definitie van zo'n venster is het volgende:

```

-> (DEFOBJECT
      VENSTER-MET-SCROLL-BAR-EN-TITEL
      VENSTER
      (TEKEN :FUNCTION
        #' (LAMBDA (ZELF)
          (DELEGATE
            (TEKEN 'VENSTER-MET-SCROLL-BAR)
            ZELF)
          (DELEGATE
            (TEKEN 'VENSTER-MET-TITEL)
            ZELF) ) ) )
#<object VENSTER-MET-SCROLL-BAR-EN-TITEL>

```

In tegenstelling tot CommonORBIT, dat DELEGATE aanbiedt als een algemeen en flexibel gereedschap voor het knutselen met erving, bieden andere talen soms een heel arsenaal van meer specifieke gereedschappen. CLOS bijvoorbeeld heeft een aantal voorgedefiniëerde combinaties van methoden. Deze method combination types omvatten onder meer LIST, +, MAX, enz. Naast een *primary method* kunnen bovendien ook andere methoden geactiveerd worden, ofwel *vóór* of *na* de primary method. Zo kan bijvoorbeeld één object een primary method leveren voor een generische functie, een ander levert een bijzondere initialisatie *vóór* de primary method, en een derde doet achteraf nog iets extra's.

6 Rollen en gestructureerde erving

6.1 Omgekeerde evaluatie door middel van rollen

In een objectgericht systeem is het soms nuttig om *omgekeerde* evaluatie te doen. Dit is de mogelijkheid om een lijst op te vragen van die objecten waarvoor een gegeven functie een gegeven resultaat oplevert. Bijvoorbeeld wil men misschien weten wiens broer Peter is, met andere woorden, alle objecten waarvoor de functie broer de waarde PETER oplevert. In CommonORBIT is omgekeerde evaluatie mogelijk wanneer de waarde van een aspect zelf ook een object is. In dat geval wordt het aspecttype :OBJECT gebruikt in plaats van :VALUE. Bij de definitie van het aspect wordt dan een terugwaartse verwijzing aangebracht; men zegt dat het object dan een *rol* heeft. Beschouw de volgende definities van de objecten PETER, KAREL en ANNA:

```
-> (DEFOBJECT PETER PERSON)
#<object PETER>
```

```
-> (DEFOBJECT KAREL
      PERSON
      (BROER :OBJECT 'PETER))
#<object KAREL>
```

```
-> (DEFOBJECT ANNA
      PERSON
      (BROER :OBJECT 'PETER))
#<object ANNA>
```

Normale evaluatie van de generische functie BROER produceert de volgende resultaten:

```
-> (BROER 'KAREL)
#<object PETER>
```

```
-> (BROER 'ANNA)
#<object PETER>
```

Omgekeerde evaluatie wordt uitgevoerd door middel van WHOSE en geeft een lijst terug:

```
-> (WHOSE 'BROER 'PETER)
(#<object KAREL> #<object ANNA>)
```

We zeggen dat PETER de rol heeft van BROER met betrekking tot de objecten KAREL en ANNA. In PETER worden terugwaartse verwijzingen opgeslagen naar KAREL en ANNA. Deze verwijzingen laten toe om snel de omgekeerde evaluatie te kunnen doen zonder *ieder* object in het systeem af te zoeken. Uiteraard worden terugverwijzingen bij herdefinities van de aspecten automatisch up to date gebracht. Zo blijft consistentie tussen normale en omgekeerde evaluatie gegarandeerd.

6.2 Gestructureerde erving

Gestructureerde erving werd wel eens omschreven als de mogelijkheid om “... een complexe verzameling van relaties tussen delen van een beschrijving samen te houden wanneer deze wordt toegepast lager in de specialisatiehiërarchie” (Brachman & Schmolze, 1985:177). Erving is met andere woorden niet beperkt tot het delen van aparte aspecten, maar het modelleert een object als geheel—met alle gerelateerde objecten—naar een prototype. Dit mechanisme, dat centraal staat in bijvoorbeeld KL-ONE, is ook aanwezig in CommonORBIT. Beschouw bijvoorbeeld de onderstaande definities voor VROUW en PERSOON.

```
→ (DEFOBJECT VROUW
    "Een vrouw is een persoon van het
    vrouwelijk geslacht."
    PERSOON
    (GESLACHT :VALUE 'VROUWELIJK))
#<object VROUW>

→ (DEFOBJECT PERSOON
    "De moeder van elke persoon is een vrouw
    die normaal gesproken geen maagd is."
    (MOEDER :OBJECT
      (A VROUW (MAAGD? NIL))))
#<object PERSOON>
```

Delegeren van het aspect MOEDER aan PERSOON omvat meer dan alleen maar het ophalen van een object, anders zou iedere persoon dezelfde moeder hebben. Daarom maakt CommonORBIT voor iedere persoon een uniek object aan dat fungeert als moeder en dat erft van het moeder-object van het prototype PERSOON. De creatie van dit moeder-object is *lazy*, dus gebeurt alleen indien erom gevraagd wordt⁴. Eens dat het aangemaakt is, wordt het echter permanent opgeslagen. Een voorbeeld volgt nu; het specialisatienetwerk wordt voorgesteld in figuur 9.

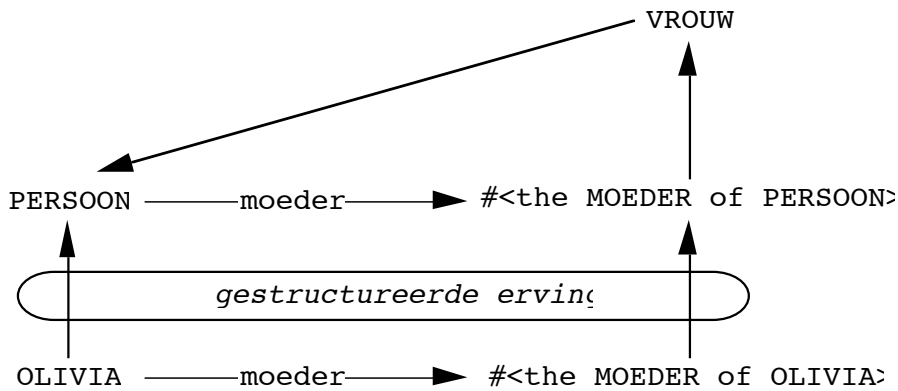
```
→ (DEFOBJECT OLIVIA PERSOON)
#<object OLIVIA>

→ (MOEDER 'OLIVIA)
#<the MOEDER of OLIVIA>

→ (GESLACHT (MOEDER 'OLIVIA))
VROUWELIJK

→ (MAAGD? (MOEDER 'OLIVIA))
NIL
```

⁴ Dit is tevens nodig om oneindige recursie te voorkomen.



Figuur 9
Gestructureerde erving

Gestructureerde erving laat ook door het programma gedefinieerde objecten toe om defaults te erven via hun rollen. In dat geval legt CommonORBIT een specialisatierelatie tussen de moeder van de client en de moeder van het prototype **PERSOON**. Bijvoorbeeld:

```

-> (DEFOBJECT HELGA
      PERSON
      (MOEDER :OBJECT (A DOKTER)))
#<object HELGA>

-> (GESLACHT (MOEDER 'HELGA))
VROUWELIJK

-> (MAAGD? (MOEDER 'HELGA))
NIL

```

Natuurlijk kunnen defaults van een prototype in gestructureerde erving toch worden overschreven. Het volgende voorbeeld spreekt voor zichzelf.

```

-> (DEFOBJECT JESUS
      GOD PERSON
      (MOEDER :OBJECT
        (DEFOBJECT MARIA
          (MAAGD? T))))
#<object JESUS>

-> (MAAGD? (MOEDER 'JESUS))
T

-> (GESLACHT 'MARIA)
VROUWELIJK

```

Samenvattend kan men stellen dat gestructureerde erving betekent dat wanneer een specialisatierelatie geldt tussen twee objecten, er tevens specialisatierelaties gelden tussen de objecten in hun overeenkomstige aspecten. Bijgevolg werkt het ervingsmechanisme effectief op de objectstructuur en niet op geïsoleerde entiteiten.

8 Memo's

Het afwegen van rekentijd tegen opslagruimte is een steeds terugkerende *trade-off* in de informatieverwerking. Wanneer het waarschijnlijk is dat het resultaat van een berekening later nog eens nodig zal zijn, is het vaak gerechtvaardigd om dit resultaat op te slaan om zo rekentijd te besparen. In de plaats van de berekening opnieuw uit te voeren kan dan het eerder berekende resultaat gewoon opgehaald worden. Dergelijke opslag noemt men vaak een *memo* of een *cache*. In objectgerichte talen kunnen memo's deel uitmaken van de interne toestand van een object.

8.1 Lazy aspecten

Soms dient de waarde van een aspect voor een object maar één keer te worden uitgerekend. Die waarde kan dan worden bewaard in een memo en hoeft niet te worden herberekend wanneer hij nog eens nodig is. Het aspecttype :IF-NEEDED is voor dit doel bestemd. Het werkt als :FUNCTION maar slaat het resultaat van de functietoepassing op in het object als een :VALUE. Dit mechanisme is *lazy*, dat wil zeggen dat het berekenen van de waarde en de opslag ervan pas plaats vindt nadat de generische functie voor het eerst is toegepast op het object.

8.2 Het bewaren van geërfde waarden

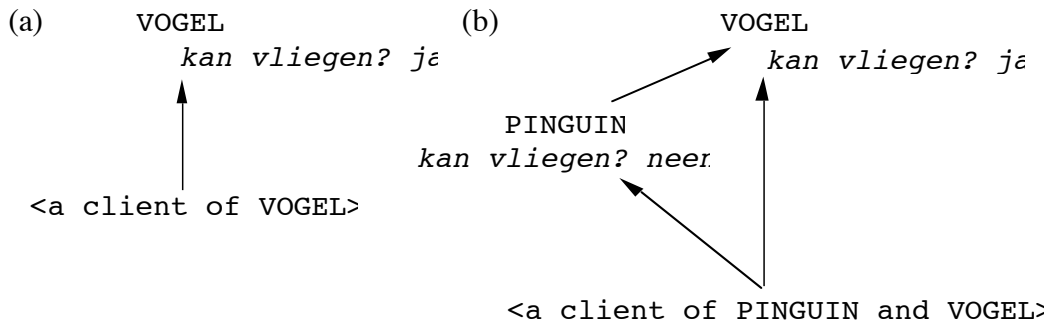
Het dynamisch opzoeken van kennis in een specialisatienetwerk komt ten goede aan de modulariteit, maar kost veel tijd omdat de hiërarchie moet worden doorzocht elke keer dat een object een aspect wil delegeren. Erving door kopiëren kan soms de snelheid van het systeem opvoeren⁵, maar is minder modulair. CommonORBIT lost dit dilemma op door *optioneel* kopiëren van geërfde kennis in aspecten van het type :VALUE, :OBJECT en OBJECTS. Dit mechanisme is eveneens *lazy*: het aanbrengen van memo's in de vorm van het kopiëren van geërfde aspecten vindt niet plaats bij de aanmaak van het object, maar na de eerste aanroep van de generische functie op het object. Volgende aanroepen gaan dan ook meestal sneller. De optie om al dan niet te kopiëren wordt in CommonORBIT aan de programmeur overgelaten om zo het systeem te kunnen afstemmen op een bepaalde toepassing. Voor systemen die vaak veranderingen ondergaan is modulariteit belangrijk en verdient erving door delegeren waarschijnlijk de voorkeur; stabiele systemen waarvoor snelheid belangrijk is kunnen erving door kopiëren goed gebruiken.

8.3 Het up-to-date brengen van memo's

Memo's zijn eigenlijk niet in overeenstemming met het idee van delegeren. Veranderingen in prototypen kunnen de informatie in hun cliënten ongeldig maken. Zolang er geen memo's worden gemaakt is dit probleemloos, want de kennis wordt dan gewoon herberekend in de nieuwe stand van zaken. Stel bijvoorbeeld dat een object van het prototype VOGEL de kennis erft dat het kan vliegen. Als dan later blijkt dat de

⁵ Anderzijds kan de extra opslag een negatieve invloed hebben op *paging* in systemen met virtueel geheugen. Lieberman (1986a,b) gaat in op deze *trade-off* van geheugenruimte tegenover rekentijd.

vogel in feite een PINGUIN is, dan wordt een andere conclusie getrokken. Erving is niet-monotoon: nieuwe kennis kan aantonen dat een geval uitzonderlijk is en dan wordt de *default*conclusie teruggetrokken (zie Brewka, 1989). Dit wordt geïllustreerd in figuur 10.



Figuur 10
Een verandering aan het specialisatienetwerk maakt een vroegere conclusie ongeldig

In een systeem met dynamische erving is non-monotoniciteit eigenlijk geen probleem omdat kennis steeds weer wordt herberekend en dus niet dient te worden herzien. Maar als kennis wordt opgeslagen in memo's dan wordt zij maar één keer berekend en daarna opgeslagen; de memo moet dus verwijderd worden. CommonORBIT houdt een netwerk voor *updating* bij dat memo's automatisch ongedaan maakt wanneer ze niet langer geldig zijn. Deze updating gebeurt zowel voor resultaten van :IF-NEEDED aspecten als voor kennis geërfd en gecopiëerd van prototypen. Natuurlijk kunnen nieuwe waarden opnieuw in memo's worden bewaard.

Updating van memo's is een optie. De programmeur kan er net zo goed voor kiezen om de oude waarden te behouden. De correctheid van het programma kan er zelfs afhankelijk van zijn.

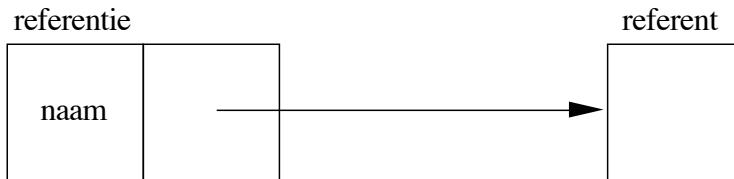
10 Referenten en het samenvoegen van objecten

Soms is het nuttig om de kennis in twee objecten samen te voegen in één object, of omgekeerd aan één object twee namen te geven. Dergelijke representaties zijn mogelijk in CommonORBIT, dat een object voorstelt in twee lagen. De eerste laag van een object heet de *referentie*; deze verwijst naar een tweede laag, de *referent*. Alle feitelijke informatie is opgeslagen in de referent. Normaal heeft iedere referentie zijn eigen referent, die een aparte entiteit in de werkelijkheid denoteert. Als twee referenties naar dezelfde referent verwijzen dan zijn die objecten *coreferentieel*. Figuur 11 geeft een schematische voorstelling van deze representatie.

Anoniem object:



Object met naam:



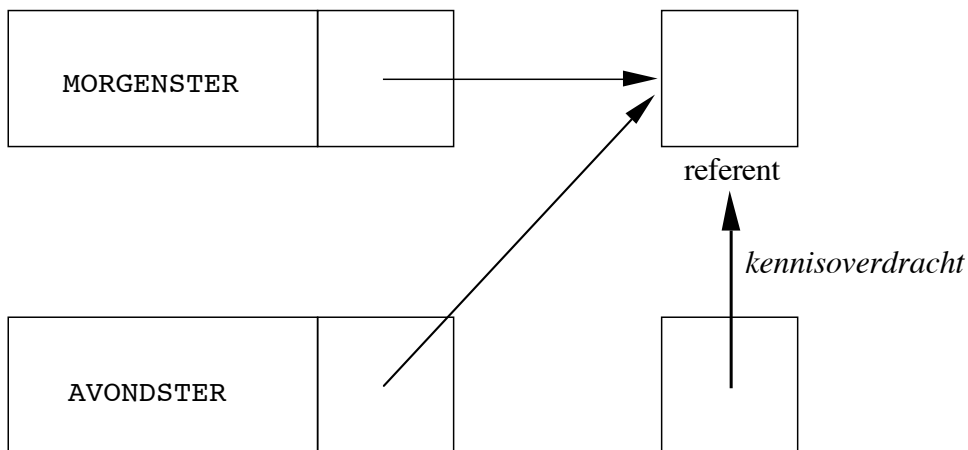
Figuur 11
Objecten en referenten

Door een tweede referentie te creëren naar een bestaande referent kunnen we een object meer dan één naam geven—of, zo men wil, kan men twee objecten maken met dezelfde referent. Als gevolg van de twee lagen zullen alle veranderingen aan het ene object ook het andere beïnvloeden. Deze mogelijkheid bleek bijvoorbeeld nuttig in een model van de Nijmeegse studentenadministratie, die aan sommige studenten meer dan één studentnummer heeft toegekend.

Ook kunnen twee objecten worden samengevoegd door alle kennis in de ene referent over te brengen naar de andere⁶ en dan de verwijzing van het ‘lege’ object te verplaatsen naar het ‘volle’. De twee objecten delen dan een referent, die toegankelijk blijft vanuit beide referenties. Het samenvoegen van objecten is nuttig in situaties wanneer twee objecten met verschillende namen dezelfde entiteit in de werkelijkheid denoteren. Een bekend voorbeeld is dat van de morgenster en de avondster, die aanvankelijk werden voorgesteld als aparte objecten maar uiteindelijk dezelfde planeet bleken te zijn. Een voorbeeld van het samenvoegen van objecten wordt gegeven in figuur 12.

```
→ (MERGE-OBJECTS 'MORGENSTER  
                  'AVONDSTER)  
#<object MORGENSTER>
```

⁶ In het geval van conflict gaat de kennis van het eerste object voor op dat van het tweede.



Figuur 12
Het samenvoegen van objecten

Het losmaken van de representatie van een object en dat van zijn referent brengt de vraag naar voren, how gelijkheid van objecten moet worden gedefinieerd. Twee verschillende gelijkheidspredikaten worden voorgesteld in CommonORBIT. Het predikaat OQL test of twee objecten denotationeel gelijk zijn, dat wil zeggen, of ze dezelfde referent hebben. De normale EQ test van Common LISP wordt gebruikt om te testen of twee objecten dezelfde referenties zijn.

Het gebruik van coreferentie in objectgerichte talen is verder uitgewerkt door Ferber en Volle (1988) die een logica hebben ontworpen voor coreferentieel redeneren. Hun systeem laat ook toe om referenties te adresseren vanuit hun referenten, wat niet mogelijk is in CommonORBIT. Coreferentie is interessant voor een objectgerichte implementatie van op *unificatie* gebaseerde grammatica's (De Smedt, 1990).

11 Besluit en perspectieven

Een objectgerichte programmeertaal verschilt van een niet-objectgerichte programmeertaal in de manier waarop kennis wordt georganiseerd. Een objectgerichte taal distribueert kennis over een aantal objecten die ieder een entiteit of situatie in een domein van de werkelijkheid voorstellen. Toch kunnen binnen het objectgerichte paradigma verschillende architecturen worden onderscheiden. Een vergelijking van bestaande objectgerichte talen brengt heel wat onderlinge verschillen aan het licht met betrekking tot de voorstelling en activering van objecten. Dit artikel is in detail ingegaan op één zo'n systeem, CommonORBIT, en heeft verschillen en overeenkomsten met andere systemen trachten aan te duiden.

CommonORBIT komt uit de vergelijking naar voren als een systeem dat gemakkelijk in gebruik is omdat de syntaxis goed past in de normale syntaxis van LISP. Het is een taal met een groot uitdrukkingsvermogen omdat zijn ervingsmechanisme zeer algemeen is. Het is een flexibele taal omdat erwing tot op het niveau van individuele aspecten gaat. Het biedt modulariteit en is toch efficiënt omdat het mechanisme van delegeren wordt aangevuld met de mogelijkheid van memo's. Anderzijds moet toegegeven worden dat CommonORBIT zijn beperkingen kent. Het is een systeem waar vele concepten in de

taal zijn ingewerkt voor zover zij praktisch zijn maar niet tot de uiterste consequenties toe. Zo wordt de definitie van een generische functie enkel op het eerste argument geselecteerd. Ook kent gestructureerde erving zijn beperkingen: het is statisch (aangemaakt bij de definitie van een aspect) en biedt geen meervoudige erving. Tenslotte werkt het updating-mechanisme enkel voor aspecten als zodanig en niet voor willekeurige berekeningen die de kennis in aspecten hebben gebruikt.

De objectgerichte representatie van kennis belichaamt een verschillende *cognitieve architectuur* dan de niet-objectgerichte. De complexiteit van de kennis die een rol speelt in intelligente processen geeft vaak op een natuurlijke wijze aanleiding tot een structurering in objecten en tot het gebruik van erving. Kennissystemen kunnen daar dankbaar gebruik van maken.

Het aantal objectgerichte talen en programmeeromgevingen groeit thans sterk. Het aantal applicaties dat op succesvolle manier gebruik heeft gemaakt van het objectgerichte paradigma is legio, vooral op het gebied van de Artificiële Intelligentie, maar ook op andere terreinen van symbolische gegevensverwerking, bijvoorbeeld user interfaces, grafische toepassingen, en zelfs operating systems. Objectgericht programmeren heeft nieuwe metaforen voor gegevensverwerking in het leven geroepen (bijvoorbeeld *message passing* en *actors*) die niet alleen nuttig zijn gebleken als softwaretechnieken maar die ook aanleiding hebben gegeven tot nieuwe hardware-architecturen.

Momenteel kunnen we enkele trends constateren in het objectgerichte paradigma. Enerzijds ontgroeien objectgerichte talen hun oorspronkelijke substraten en groeien door naar zelfstandige en volwaardige kennisrepresentatiesystemen. KRS bijvoorbeeld is een krachtige programmeeromgeving voor kennisrepresentatie met vele niet-conventionele kenmerken. Het houdt een representatie bij van elk concept op een intensioneel niveau zowel als een extensioneel niveau. Het is voorzien van een compleet *truth-maintenance* mechanisme (Van Marcke, 1986, 1987) en is programmeerbaar op een meta-niveau zodat het zichzelf kan veranderen door introspectie (Maes, 1986)⁷. Anderzijds wordt het objectgerichte programmeerparadigma stilaan geïntegreerd in bestaande programmeerparadigma's. De programmeertaal FORK (Beckstein, Görz & Tieleman, 1987) is een van de vele systemen waar objectgericht en regelgebaseerd programmeren worden verweven. Ook het vermelden waard zijn objectgerichte varianten van logische talen (bijvoorbeeld Kahn, Tribble, Miller & Bobrow, 1987). Ook wordt een objectgerichte representatie steeds vaker samengevoegd met conventionele database-management-technieken (bijvoorbeeld het STATICE-systeem van Symbolics). Deze ontwikkelingen laten zien dat het objectgerichte programmeerparadigma een steeds belangrijker invloed gaat uitoefenen op andere programmerstijlen. In de toekomst voorzie ik dat deze ontwikkelingen zullen leiden tot een kennisrepresentatieomgeving waarin verschillende controlestructuren en organisaties van kennis met elkaar een symbiose vormen.

⁷ Helaas moeten KRS-programmeurs de prijs voor dit *creeping featurism* betalen met een syntaxis die veel gecompliceerder is dan die van CommonORBIT.

LITERATUUR

- Anderson, J.R., Corbett, A.T. & Reiser, B.J. 1987. *Essential LISP*. Reading: Addison-Wesley.
- Beckstein, C., Görz, G. & Tielemann, M. 1987. FORK: a system for object- and rule-oriented programming. *Proceedings of ECOOP'78 (Bigre+Globule 54)*, 303-314. Paris: AFCET.
- Bobrow, D.G. & Winograd, T. 1977. An overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1, 3-46.
- Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M. & Zdybel, F. 1985. *CommonLoops: Merging Common LISP and Object-Oriented Programming*. Report ISL-85-8, Xerox Palo Alto Research Center, Palo Alto (CA).
- Brachman, R.J. & Schmolze, J.G. 1985. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9, 171-216.
- Brachman, R.J. 1979. On the Epistemological Status of Semantic Networks. In: Findler, N.V. (ed.) *Associative Networks: Representation and Use of Knowledge by Computers* (pp. 3-50). New York: Academic Press.
- Brewka, G. 1989. Nonmonotonic logics—a brief overview. *AI Communications* 2, 88-97.
- Cox, B.J. 1986. *Object-oriented programming: an evolutionary approach*. Reading: Addison-Wesley.
- Dahl, O.J. & Nygaard, K. 1966. SIMULA — an ALGOL-based simulation language. *CACM*, 9, 671-678.
- Dahl, O.J., Myhrhaug, B. & Nygaard, K. 1971. *SIMULA 67 Common Base Language*. Pub. S-22, Norwegian Computing Center, Oslo.
- DeMichiel, L.G. & Gabriel, R.P. 1987. The Common Lisp Object System: an overview. In: *Proceedings of ECOOP'78 (Bigre+Globule 54)*, pp. 201-220. Paris: AFCET.
- Ducournau, R. & Habib, M. 1987. On some algorithms for multiple inheritance in object-oriented programming. In: *Proceedings of ECOOP'78 (Bigre+Globule 54)*, 291-300. Paris: AFCET.
- Ferber, J. & Volle, Ph. 1988. Using coreference in object-oriented representations. In: Kodratoff, Y. (ed.) *Proceedings of the 8th European Conference on Artificial Intelligence* (pp. 238-240). London: Pitman.
- Fikes, R. & Kehler, T. 1985. The role of frame-based representation in reasoning. *CACM*, 28, No. 9, 904-920.
- Goldberg, A. & Robson, D. 1983. *Smalltalk-80. The language and its implementation*. Reading (MA): Addison-Wesley.
- Hewitt, C. 1979. Viewing Control Structures as Patterns of Passing Messages. In: Winston, P. & Brown, R. (eds.) *Artificial Intelligence: an MIT Perspective*. Cambridge (MA): MIT Press.
- Ingalls, D. H. 1976. The Smalltalk-76 Programming System: Design and Implementation. In: *Conference record of the Fifth Annual ACM Symposium on Principles of Programming Languages* (pp. 9-16). Tucson (AZ).
- Kahn, K., Tribble, E.D., Miller, M.S. & Bobrow, D.G. 1987. VULCAN: logical concurrent objects. In: Shriver, B. & Wegner, P. (eds.) *Research directions in object-oriented programming*. Cambridge, MA: MIT Press.
- Keene, S. E. 1988. *Object-oriented programming in Common Lisp*. Reading (MA): Addison-Wesley.

- Lieberman, H. 1986. Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object-Oriented Systems. In: *Proceedings of the Third Workshop on Object Oriented Languages, Paris*.
- Lieberman, H. 1987. Languages, object-oriented. In: Shapiro, S. (ed.), *Encyclopedia of Artificial Intelligence*, Vol. 1 (pp. 452-456). New York: Wiley.
- Maes, P. 1986. Introspection in knowledge representation. In: *ECAI-86. Proceedings of the 7th European Conference on Artificial Intelligence* (pp. 256-269).
- Marcke, K. van 1986. FPPD: a consistency maintenance system based of forward propagation of denials. In: *Proceedings of the 7th European Conference on Artificial Intelligence. Brighton* (pp. 278-290).
- Marcke, K. van 1987. KRS: An Object-Oriented Representation Language., *Revue d'Intelligence Artificielle*, 1, No. 4.
- McCarthy, J. 1960. Recursive functions of symbolic expressions. *CACM*, 3, 184-195.
- Minsky, M. 1975. A framework for representing knowledge. In: Winston, P. (ed.) *The psychology of computer vision* (pp. 211-277). New York: McGraw-Hill
- Moon, D. A. 1986. Object-Oriented Programming with FLAVORS. *Proceedings of OOPSLA '86*. ACM.
- Smedt, K. de 1987. Object-oriented programming in Flavors and CommonORBIT. In: Hawley, R. (ed.) *Artificial Intelligence programming environments* (pp. 157-176). Chichester: Ellis Horwood.
- Smedt, K. de 1990. Incremental sentence generation: a computer model of grammatical encoding. Dissertatie, NICI Technical Report 90-01, Universiteit Nijmegen.
- Steele, G.L. 1984. *Common LISP: the language*. Digital Press.
- Steels, L. 1983. ORBIT: An applicative view of object-oriented programming. In: Degano & Sandewall (eds.), *Proceedings of the European Conference On Integrated Interactive Computing Systems, Stresa, Italy, 1-3 Sept. 1982*. North-Holland, Amsterdam.
- Touretzky, D.S. 1986. *The mathematics of inheritance systems*. Los Altos: Morgan Kaufmann.
- Touretzky, D.S., Horty, J.F. & Thomason, R.H. 1987. A clash of intuitions: The current state of nonmonotonic multiple inheritance systems. In: *Proceedings of the 10th IJCAI, Milan*. Los Altos: Morgan Kaufmann.
- Weinreb, D. & Moon, D. 1980. *Flavors: message passing in the Lisp Machine*. Memo AIM-602, MIT, Cambridge, MA.
- Winston, P.H. & Horn, B.K.P. 1988. *Lisp* (derde uitgave). Reading (MA): Addison-Wesley.
- Wirth, N. 1971. Program development by step-wise refinement. *Communications of the ACM* 14, 221-227.

Koenraad de Smedt is universitair docent in de studierichtingen Cognitiewetenschap en Psychologie aan de Universiteit van Nijmegen. Als onderzoeker is hij verbonden aan het Nijmeegs Instituut voor Cognitieonderzoek en Informatietechnologie (NICI).