(Afstudeerscriptie 313)

Object-Oriëntatie vanuit theorie en praktijk

Een studie naar de theoretische basis en de praktische invoering van Object-georiënteerde technologie

Willem Nabuurs

Katholieke Universiteit Nijmegen
Opleiding Bedrijfsgerichte informatica

Zeist, Rabobank Nederland
INFORMATISERINGSBELEID EN STRUCTURERING AB
Richtlijnen Systeemontwikkeling WAB/AB
juni 1994

Object-Oriëntatie vanuit theorie en praktijk

Een studie naar de theoretische basis en de praktische invoering van Object-georiënteerde technologie

Afstudeerscriptie 313,

ter verkrijging van de graad doctorandus
aan de Katholieke Universiteit Nijmegen,
Opleiding Bedrijfsgerichte informatica,
Faculteit der Wiskunde en Informatica,
Vakgroep Informatica
Afdeling Informatiesystemen

door

W.J.A. Nabuurs

geboren te Wanroij, 3 oktober 1970

Afstudeerdocent:	dr. A.H.M. ter Hofstede
Afstudeerbegeleider:	dr. M.P.H. Thurlings
Datum:	20 juni 1994

Zeist, Rabobank Nederland
INFORMATISERINGSBELEID EN STRUCTURERING AB
Richtlijnen Systeemontwikkeling WAB/AB
juni 1994

Voorwoord

Deze scriptie vormt de afronding van mijn studie bedrijfsgerichte informatica aan de Katholieke Universiteit Nijmegen. In deze scriptie wordt een onderzoek beschreven, dat door mij voor de afdeling Richtlijnen Systeemontwikkeling WAB/AB is uitgevoerd binnen de automatiseringsafdelingen van Rabobank Nederland. Het onderwerp van dat onderzoek was Object-Oriëntatie in de ruimste betekenis van de term.

Bijna vijf jaar geleden kwam ik als eerstejaars student naar Nijmegen, niet gehinderd door enige kennis op het gebied van de informatica. Ik wist, hoe ik programma's op een Commodore 64 kon 'runnen' (lees: spelletjes kon spelen) en ik had een jaar lang een cursus programmeren in Commodore 32 Basic gevolgd, maar al tijdens de eerste week van mijn studie werd mij duidelijk, dat dat alles nauwelijks iets met informatica te maken had. Sindsdien heb ik het ene wonder van de informatietechnologie na het andere mogen aanschouwen en heb ik het één en ander geleerd van informatica, maar ook van andere vakgebieden, zoals bedrijfskunde en wiskunde. Het bewijs daarvan hebt u nu in handen.

Graag wil ik op deze plaats een aantal personen bedanken, die mij in de afgelopen maanden hebben geholpen bij mijn onderzoek en bij het schrijven van mijn scriptie:

- mijn afstudeerbegeleiders bij Rabobank Nederland en op de universiteit: dr. Thijs Thurlings, hoofd van de afdeling Richtlijnen Systeemontwikkeling binnen de activiteit Informatiseringsbeleid en Structurering Aangesloten Banken, en dr. Arthur ter Hofstede, universitair docent bij de afdeling Informatiesystemen van de vakgroep Informatica
- Hans van den Engel, die zes maanden lang zijn kamer met mij heeft moeten delen, die altijd bereid was om in discussie te treden over zaken, die mij niet geheel duidelijk waren, en die steeds weer de moeite nam om de fouten uit het door mij geproduceerde werk te halen
- mr. J.A.R. Kelleter, algemeen directeur van Rabobank Raamland, die mij op het spoor van deze opdracht zette, en dhr. A.H. Mariën, werkzaam bij de afdeling Stages en Onderwijscontacten van Rabobank Nederland, die mij de kans heeft geboden om mijn afstudeeropdracht bij Rabobank Nederland uit te voeren
- mijn zus Rian, die zich, ondanks (of misschien vanwege) alle kritiek die ik heb geleverd op haar scriptie, door meer dan honderd pagina's voor haar minder interessante en soms onbegrijpelijke tekst heeft geworsteld en aan wie het te danken is, dat het aantal fouten tegen de Nederlandse taal beperkt is gebleven
- alle medewerkers van Rabobank Nederland en Rabofacet, die mij hebben geholpen bij mijn onderzoek door het geven van interviews, het voeren van discussies over Object-georiënteerde onderwerpen tijdens de lunchpauze of in de wandelgangen en het verschaffen van 'leesvoer'
- alle medewerkers van en (oud)studenten aan de Katholieke Universiteit Nijmegen, die mij op de één of andere manier hebben geholpen bij mijn onderzoek, met name de cognitiewetenschappers, die (helaas vergeefs) hebben geprobeerd om mij in te wijden in hun wondere wereld

Tenslotte wil ik ook al die mensen bedanken, die misschien geen actieve bijdrage hebben geleverd aan mijn onderzoek of aan mijn scriptie, maar zonder wie ik nooit tot dit resultaat was gekomen: mijn vrienden in Nijmegen en in Mill, Rian, Joris en last but definitely not least mijn ouders.

Willem Nabuurs
Zeist, 20 juni 1994

Inhoud

Inleiding

Aanleiding van het onderzoek

Sinds het einde van de jaren zestig is binnen het gebied van de systeemontwikkeling sprake van de zogenaamde 'software crisis'. Te ontwikkelen systemen zijn dermate complex geworden, dat er behoefte aan betere methoden en technieken voor systeemontwikkeling is ontstaan. Sindsdien zijn tal van dergelijke methoden en technieken ontwikkeld. Een aantal van die methoden is gebaseerd op de zogenaamde Object-georiënteerde benadering.

De afgelopen jaren is het begrip Object-Oriëntatie (OO) uitgegroeid tot een modewoord binnen de automatiseringswereld. OO wordt door voorstanders gezien als de oplossing van de software crisis. Het bedrijfsleven is echter nog niet massaal overgestapt op OO. Zolang onderzoek niet heeft bevestigd, dat de overstap naar Object-georiënteerde systeemontwikkeling lonend is, zal dit waarschijnlijk ook niet gebeuren.

Ook binnen de Coöperatieve Centrale Raiffeisen-Boerenleenbank B.A., in het vervolg van deze scriptie kortweg Rabobank Nederland genoemd, bestaat twijfel over de praktische bruikbaarheid van OO. Binnen sommige automatiseringsafdelingen wordt reeds geëxperimenteerd met OO, terwijl binnen andere afdelingen de gedachte leeft, dat OO een modeverschijnsel is, dat uiteindelijk wel zal overwaaien. Om die reden heeft dr. M.P.H. Thurlings, hoofd van de afdeling Richtlijnen Systeemontwikkeling binnen het Werkgebied Aangesloten Banken, ons gevraagd een onderzoek te doen naar de betekenis van OO en naar de praktische bruikbaarheid van OO binnen een onderneming, toegespitst op Rabobank Nederland.

De praktische bruikbaarheid van een technologie is sterk afhankelijk van de hulpmiddelen, die de ontwikkelaars bij het gebruiken van die technologie ter beschikking staan. De mate, waarin een hulpmiddel automatische ondersteuning kan bieden aan een systeemontwikkelaar, wordt op haar beurt beïnvloed door de mate, waarin het ontwikkelproces is geformaliseerd [HOF93]. Daar vrijwel alle OO methoden een formele basis ontberen [CF92], omvat het hierboven genoemde onderzoek tevens een aanzet tot de formalisatie van de modellen, waarmee tijdens Object-georiënteerde systeemontwikkeling wordt gewerkt.

Probleemstelling van het onderzoek en onderzoeksvragen

Het doel van dit onderzoek is het verkrijgen van inzicht in de betekenis van OO en in de rol, die OO in de praktijk kan spelen, alsmede het opstellen van een wetenschappelijk onderbouwd algemeen Object Model.

Het hierboven geformuleerde onderzoeksdoel willen wij bereiken door de beantwoording van een drietal onderzoeksvragen.

- Wat is de betekenis van het begrip Object-Oriëntatie en wat zijn de verschillen tussen conventionele en Object-georiënteerde systeemontwikkeling?
- Wat zijn de voorwaarden voor implementatie van Object-Oriëntatie binnen de organisatiestructuur van een onderneming, in dit geval Rabobank Nederland, en waarop dient een keuze voor Object-georiënteerde methoden en programmeertalen gebaseerd te worden?
- Welke complicaties doen zich voor, wanneer vanuit een wetenschappelijke invalshoek wordt gekeken naar Object-georiënteerde methoden?

Opzet van de scriptie

Deze scriptie is gesplitst in drie delen: een algemeen informatief deel, een op de praktijk gericht deel en een op de theorie gericht deel. Wij hebben gekozen voor deze indeling, omdat die naar onze mening het beste de beantwoording van de drie onderzoeksvragen mogelijk maakt. De verschillende delen vertonen een hoge onderlinge samenhang, maar kunnen eventueel onafhankelijk van elkaar worden gelezen.

Het eerste deel beslaat vier hoofdstukken. In deze hoofdstukken wordt het begrip Object-Oriëntatie omschreven en wordt een overzicht gegeven van de meest relevante met Object-Oriëntatie samenhangende onderwerpen.

Hoofdstuk 1 begint met een korte beschrijving van de geschiedenis van Object-Oriëntatie. Vervolgens wordt het begrip Object-Oriëntatie omschreven aan de hand van de verschillende definities, die door een aantal auteurs zijn gegeven, gevolgd door de definitie, waar wij in deze scriptie van zijn uitgegaan. Daarna wordt een aantal kernbegrippen van de Object-georiënteerde benadering uitgewerkt. Tenslotte worden de belangrijkste karakteristieken van Object-georiënteerde systeemontwikkeling beschreven.

In hoofdstuk 2 worden de voordelen behandeld, die kunnen worden behaald met het gebruik van Object-Oriëntatie. Zoals iedere technologie kent ook Object-georiënteerde technologie een aantal nadelen. Ook deze nadelen worden in hoofdstuk 2 behandeld. Tenslotte besteden wij aandacht aan de gevaren, die zijn verbonden aan slordig gebruik van Object-Oriëntatie.

In hoofdstuk 3 wordt ingegaan op de rol van het management bij Object-georiënteerde systeemontwikkeling. In dit hoofdstuk komen het plannen van hergebruik en iteratieve en incrementele ontwikkeling, een alternatieve werkverdeling en methoden voor het meten van kosten en baten van Object-georiënteerde projecten aan bod.

In hoofdstuk 4 wordt aandacht besteed aan Object-georiënteerde technologie. Achtereenvolgens worden Object-georiënteerde methoden, Object-georiënteerde programmeertalen, Object-georiënteerde database management systemen, Object-georiënteerde tools en de standaardisatie organisatie OMG belicht. Het hoofdstuk wordt afgesloten met een aantal aan de literatuur ontleende resultaten van Object-georiënteerde projecten.

In deel II ligt de nadruk op de rol, die Object-Oriëntatie kan spelen in het bedrijfsleven. Dit deel bestaat uit twee hoofdstukken, waarin respectievelijk de kritieke succesfactoren bij de invoering van Object-Oriëntatie in een onderneming en richtlijnen voor die invoering centraal staan.

In hoofdstuk 5 wordt eerst een vijftal factoren, die naar onze mening een kritieke rol spelen bij de invoering van Object-Oriëntatie in een onderneming, behandeld. Vervolgens wordt een methode gepresenteerd, waarmee de stand van zaken van die kritieke succesfactoren met betrekking tot de onderneming en de Object-georiënteerde technologie met elkaar kunnen worden vergeleken.

In hoofdstuk 6 wordt een aantal richtlijnen voor de invoering van Object-Oriëntatie in een bedrijf gegeven. Deze richtlijnen hebben betrekking op de aard van eventuele proefprojecten en de keuze voor bepaalde methoden, programmeertalen, database management systemen en tools. Daarnaast wordt het ISA raamwerk [SZ92], dat op dit moment binnen een aantal afdelingen van Rabobank Nederland wordt gebruikt voor de evaluatie van tools, voorzien van een uitbreiding, die het mogelijk maakt ook Object-georiënteerde tools met behulp van dit raamwerk te evalueren.

Het derde deel beschrijft de formalisatie van een Object Model en een bijbehorend Transactie Model. Deze modellen zijn niet gebaseerd op een bepaalde methode, maar zijn zodanig opgesteld, dat met name de objectmodellen van de meeste methoden eenvoudig naar het door ons voorgestelde Object Model kunnen worden vertaald.

In hoofdstuk 7 worden objectstructuren geïntroduceerd. Objectstructuren zijn Object Modellen zonder grafische constraints. Met behulp van een aantal axioma's wordt de rol van de verschillende onderdelen van een objectstructuur vastgelegd, en wordt de invloed, die overerving heeft op de verbanden tussen de verschillende objectklassen binnen een objectstructuur, beschreven.

In hoofdstuk 8 wordt ingegaan op de wijze, waarop dergelijke objectstructuren kunnen worden 'gevuld' met object instanties. Daartoe worden dergelijke vullingen, in deze scriptie populaties genoemd, gedefinieerd en worden met behulp van axioma's een aantal beperkingen aan dergelijke populaties opgelegd. Vervolgens worden grafische constraints, waarmee aan populaties van een objectstructuur nog meer beperkingen kunnen worden opgelegd, formeel gedefinieerd. Tenslotte wordt aandacht besteed aan de wijze, waarop object instanties kunnen worden geïdentificeerd.

In hoofdstuk 9 wordt de interne structuur van object instanties beschreven. Daarbij ligt de nadruk op de dynamiek. Eerst wordt een aantal atomaire taken beschreven, waarmee de populatie van een objectstructuur en de waarden van de variabelen binnen het model kunnen worden gewijzigd. Vervolgens wordt met behulp van deze atomaire taken een aantal transacties beschreven, die er voor zorgen, dat dergelijke wijzigingen niet leiden tot ongeldige populaties. Tenslotte wordt aangegeven, hoe met behulp van transacties en atomaire taken methoden, ofwel de implementaties van de operaties van object instanties, kunnen worden gespecificeerd.

De scriptie besluit met een samenvatting en een aantal conclusies.

In bijlage A wordt een verklaring van de gebruikte wiskundige en grafische notaties gegeven.

Bij deze scriptie hoort nog een tweede bijlage, die alleen is opgenomen in de scripties, die eigendom blijven van Rabobank Nederland. In deze bijlage wordt de in hoofdstuk 6 en 7 behandelde stof vertaald naar de huidige stand van zaken bij Rabobank Nederland.

Deel I

Een overzicht van Object-Oriëntatie

1 Object-Oriëntatie

1.1 Inleiding

In dit hoofdstuk wordt de term *Object-Oriëntatie* verduidelijkt. Hiertoe wordt eerst een korte beschrijving gegeven van de geschiedenis van de Object-georiënteerde technologie. Vervolgens wordt de betekenis van de term Object-Oriëntatie omschreven aan de hand van definities, die door andere auteurs zijn gegeven, en worden de door ons gebruikte definities van de begrippen Object-Oriëntatie, Object-georiënteerde benadering, Object-georiënteerde technologie en Object-georiënteerde systeemontwikkeling gegeven. Daarna wordt een aantal binnen Object-Oriëntatie gebruikte begrippen geïntroduceerd. Tenslotte wordt een overzicht gegeven van de belangrijkste karakteristieken van Object-georiënteerde systeemontwikkeling.

1.2 De geschiedenis van Object-Oriëntatie

De afgelopen tien jaar is veel geschreven over de Object-georiënteerde benadering. Daardoor ontstaat de indruk, dat Object-Oriëntatie (OO) een modern verschijnsel is. OO wordt echter al meer dan vijfentwintig jaar gebruikt. De meeste auteurs plaatsen het beginpunt van het 'Object-georiënteerde tijdperk' in 1967, bij het verschijnen van de programmeertaal *Simula 67* [SIG92]. Anderen, zoals Hull en King [HK87], menen, dat OO voortkomt uit het onderzoek naar abstracte datatypen van onder andere Liskov en Zilles [LZ74]. [SOM89] geeft, naast de twee eerder genoemde redenen, een derde reden: het onderzoek naar afscherming van informatie (*information hiding*), zoals bijvoorbeeld is gedaan door Parnas (zie [PAR72], [PAR76]).

Simula 67 werd ontwikkeld aan het Norwegian Computing Center (NCC) door Kristen Nygaard en Ole-Johan Dahl. Begin jaren zestig ontwikkelden zij een simulatietaal, Simula 1, waarmee zij in staat waren processen op een natuurgetrouwe wijze te simuleren. Beïnvloed door de in diezelfde periode ontwikkelde programmeertaal ALGOL 60, breidden Nygaard en Dahl hun simulatietaal uit tot een op ALGOL gebaseerde, volledige programmeertaal. Bjarne Stroustrup, de ontwikkelaar van C++, noemt Simula 67 daarom "ALGOL 60 with Classes" [SIG92]. Hoewel deze taal het commercieel gezien nooit heeft gehaald, mag Simula 67 worden beschouwd als de bakermat van OO. Bertrand Meyer, ontwikkelaar van de OO programmeertaal Eiffel en een vooraanstaand persoon in de OO wereld, stelt in [SIG92]: "anyone doing object-oriented analysis, design, or programming today is a Simula programmer".

In de jaren zeventig groeide de OO aanhang langzaam. Nygaard wijt dit vooral aan de slechte marketingstrategie van het NCC [SIG92]. In deze periode werd door Alan Kay en Dan Ingalls, verbonden aan XeroxParc, een tweede OO taal ontwikkeld, *Smalltalk-72* genaamd. Deze taal evolueerde via tal van tussenversies tot het tegenwoordig gebruikte *Smalltalk-80*.

Begin jaren tachtig begon de populariteit van Object-georiënteerd programmeren (*OOP*) te groeien. Het aantal programmeertalen nam snel toe (CLOS in 1983, Object Pascal en Objective-C in 1984, C++ en Eiffel in 1985). In deze periode verschenen ook de eerste Object-georiënteerde database management systemen (*OODBMSs*) en in 1986 werd voor het eerst de *OOPSLA conferentie* (Conference on Object-Oriented Programming, Systems, Languages and Applications) georganiseerd [SIG92].

Met het populair worden van OOP ontstond langzaam maar zeker ook aandacht voor het voortraject: de analyse- en ontwerp-fase. In [BOO86] werd voor het eerst aandacht besteed aan een OO methode voor het gehele ontwikkeltraject. Sindsdien is een groot aantal methoden verschenen, waarvan er een aantal in sectie 4.2 (Methoden en technieken) wordt behandeld. Op basis van deze methoden zijn de eerste *OO-CASE tools* ontwikkeld.

Op dit moment lijkt OO snel terrein te winnen. Inmiddels is een standaardisatie organisatie in werking gesteld, de *Object Management Group*. De OMG is een niet-commerciële, internationale organisatie, die in het leven is geroepen door een collectief van meer dan tweehonderd computer- en softwarebedrijven [MAR92]. Doelen van de OMG zijn het vergroten van portabiliteit, herbruikbaarheid en uitwisselbaarheid van OO software, het standaardiseren van een referentie architectuur en het voorzien in een platform voor discussie, onderwijs en promotie van Object-georiënteerde technologie.

Daarnaast worden jaarlijks conferenties georganiseerd, zoals de eerder genoemde OOPSLA conferentie en *ECOOP*, de European Conference on Object-Oriented Programming, en verschijnen geheel aan OO gewijde tijdschriften (onder andere het Journal of Object-Oriented Programming en de Hotline on Object-Oriented Technology).

1.3 Definities van Object-Oriëntatie

In de voorgaande sectie is beschreven, hoe OO zijn intrede heeft gedaan in de wereld van de systeemontwikkeling. Een omschrijving van het begrip OO is echter nog niet gegeven. In deze sectie wordt het begrip 'Object-Oriëntatie' nader omschreven.

Hoewel veel schrijvers de term OO op de één of andere manier gebruiken, neemt slechts een klein aantal de moeite om een definitie van het begrip Object-Oriëntatie te formuleren. [BOO91] omschrijft Object-georiënteerde decompositie als een modelleringstechniek, waarin de wereld wordt beschouwd als een verzameling objecten, die samenwerken om een gewenste functionaliteit te bereiken. In [RBP$^+$91] wordt Object-georiënteerde systeemontwikkeling omschreven als een nieuwe manier van denken over software, gebaseerd op abstracties, die bestaan in de echte wereld. In [CY91a] wordt OO gedefinieerd met behulp van de vergelijking

Object-georiënteerd = Klassen en Objecten
+ Overerving
+ Communicatie met behulp van boodschappen.

Voor een omschrijving van de gebruikte termen verwijzen wij naar sectie 1.4 (Kernbegrippen van Object-Oriëntatie).

Een aantal schrijvers spreekt van het *Object-georiënteerde paradigma*, waarbij, in tegenstelling tot bijvoorbeeld procedurele systeemontwikkeling, de nadruk ligt op de data en op de relaties tussen de objecten [KM90]. In [RID94] wordt het Object-georiënteerde paradigma informeel omschreven als "een ordeningsprincipe, waarvan de essentie als volgt wordt weergegeven: een systeem is te beschouwen als een dynamische verzameling autonome objecten, die voldoende van zichzelf en elkaar weten om hun eigen verantwoordelijkheid waar te maken en om elkaar diensten te vragen en te leveren zonder van elkaars interne organisatie kennis te hebben".

In het vervolg van deze scriptie verstaan wij onder het begrip *Object-georiënteerde benadering* die benadering van het ontwikkelen van geautomatiseerde systemen, waarbij

- de werkelijkheid wordt gemodelleerd als een verzameling objecten, die allemaal bepaalde eigenschappen hebben en die allemaal een bepaald gedrag vertonen, wanneer andere objecten hen daar door het sturen van boodschappen om vragen
- de waarden van de eigenschappen van objecten samen met de operaties op die waarden zijn ingekapseld in die objecten en daardoor slechts door andere objecten kunnen worden benaderd, wanneer deze objecten door het sturen van boodschappen om een bepaald gedrag en dus om uitvoering van die operaties vragen

- objecten zijn gegroepeerd op basis van gemeenschappelijke eigenschappen en gemeenschappelijk gedrag, zodat zij gebruik kunnen maken van dezelfde implementaties
- groepen objecten met vergelijkbaar gedrag en dus met vergelijkbare operaties door middel van overerving en polymorfisme gebruik kunnen maken van dezelfde implementatie van die operaties

Met de term *Object-georiënteerde technologie* bedoelen wij in deze scriptie het geheel van methoden, programmeertalen, database management systemen, tools en standaarden, die uitgaan van een Object-georiënteerde benadering.

Onder *Object-Oriëntatie* wordt in deze scriptie zowel de Object-georiënteerde benadering als Object-georiënteerde technologie verstaan. Als wij spreken van *Object-georiënteerde systeemontwikkeling*, dan wordt gedoeld op een ontwikkelproces, waarbij wordt uitgegaan van de Object-georiënteerde benadering en waarbij gebruik wordt gemaakt van Object-georiënteerde technologie.

1.4 Kernbegrippen van Object-Oriëntatie

In de verschillende OO methoden wordt een aantal nieuwe begrippen geïntroduceerd en wordt aan een aantal oude begrippen een nieuwe betekenis toegekend. De naamgeving is in de verschillende methoden niet consistent, maar een aantal begrippen komt in bijna iedere methode voor. Deze begrippen, die in het vervolg worden aangeduid als de *kernbegrippen* van Object-Oriëntatie, zijn:

- object instantie
- klasse
- attribuut
- operatie
- methode
- relatie
- generalisatie
- aggregatie

In het vervolg van deze sectie worden deze kernbegrippen nader omschreven.

1.4.1 Object instantie

In [SNY93] wordt een *object instantie*, vaak gewoon *object* genoemd, gedefinieerd als een identificeerbare entiteit, die een zichtbare rol speelt in het leveren van een dienst, waar een klant (gebruiker of programma) om verzoekt. In [CY91a] en [RBP+91] wordt daaraan toegevoegd, dat een object een inkapseling van attribuutwaarden en de bijbehorende exclusieve diensten is. Een belangrijk verschil tussen objecten in conventionele datamodellering en objecten in OO is het feit, dat bij OO objecten een identiteit hebben, die onafhankelijk is van de waarde van hun attributen, terwijl twee objecten met precies dezelfde datawaarden bij conventionele datamodellering hetzelfde object zijn (het principe van 'zwakke identificatie') [HOF93]. Andere namen, die door auteurs voor hetzelfde begrip worden gebruikt, zijn *instantie*, *klasse instantie*, *surrogaat* en *entiteit* [SNY93].

1.4.2 Klasse

[RBP+91] definieert een *klasse* als een beschrijving van een groep objecten met gelijke eigenschappen, hetzelfde gedrag, dezelfde relaties en dezelfde semantiek. [CY91a] voegt daaraan toe, dat een klasse een beschrijving bevat van de wijze, waarop nieuwe, tot de klasse behorende object instanties kunnen worden gecreëerd. [SNY93] gebruikt in plaats van de term klasse de term *type*. In het vervolg van deze scriptie wordt door ons meestal de term *objectklasse* gebruikt.

1.4.3 Attribuut

Een *attribuut* wordt in [RBP+91] omschreven als een benoemde eigenschap van een klasse, die wordt gedeeld door alle objecten in die klasse en die door middel van een datawaarde wordt beschreven. Door de inkapseling van objecten (zie paragraaf 1.5.1, Inkapseling van proces en data) zijn deze attributen niet voor de omgeving zichtbaar. De waarden van attributen kunnen alleen met behulp van de bijbehorende operaties worden gelezen en veranderd. [SNY93] spreekt van *toestandvariabelen* en geeft als in gebruik zijnde synoniemen de termen *instantie variabele, data member, veld* en *slot*.

1.4.4 Operatie

Een *operatie* is in [RBP+91] gedefinieerd als een functie of transformatie, die mag worden toegepast op object instanties in een klasse. In [CY91a] wordt gebruik gemaakt van het voor OO karakteristieke antropomorfisme (zie paragraaf 1.5.2, Antropomorf ontwerp) en wordt gesproken van een *dienst*, ofwel het gedrag, waarvoor een object verantwoordelijk is. [SNY93] spreekt van *generische operatie* en legt zo de nadruk op het feit, dat één operatie tal van verschillende implementaties kan hebben, die elk een bepaalde methode realiseren (zie paragraaf 1.4.5, Methode). Andere synoniemen zijn *message selector*[1], *methode*, *generische functie*, *overbeladen functie* en *virtual member function* [SNY93].

1.4.5 Methode

[RBP+91] definieert de term *methode* als de implementatie van een operatie voor een specifieke klasse. Hierbij wordt uitgegaan van *polymorfisme*: het mechanisme, waarbij dezelfde naam kan worden gegeven aan verschillende functies. Polymorfisme mag worden gebruikt voor operaties in klassen, die geen onderlinge samenhang vertonen, maar ook voor dezelfde operatie in één generalisatiestructuur (zie paragraaf 1.4.7, Generalisatie). In dat geval erft de subklasse wel de operatie van zijn superklasse, maar niet de methode.

Polymorfisme is niet kenmerkend voor OO. Ook een aantal niet-OO programmeertalen, zoals Elan, is met dit mechanisme uitgerust. Het combineren van polymorfisme en *dynamische binding*, waarmee het pas tijdens executie binden van een methode aan een operatie wordt bedoeld, met het klassemechanisme levert extra voordelen op. Op deze voordelen gaan wij in in paragraaf 4.3.2 (Eigenschappen van programmeertalen).

[1] Wij hebben zoveel mogelijk geprobeerd logisch klinkende Nederlandse termen te gebruiken in plaats van de Engelse termen, maar in enkele gevallen, zoals in het geval van 'message selector', klinkt de Nederlandse vertaling zo 'gemaakt', dat wij van vertaling hebben afgezien.

De *signatuur* van een methode wordt gevormd door de parameters en het resultaat van de methode. Het hangt van de gebruikte vorm van typering (zie paragraaf 4.3.2, Eigenschappen van programmeertalen) af of de verschillende delen van de signatuur getypeerd dienen te zijn.

Volgens [SNY93] wordt ook de term *member function* gebruikt als synoniem voor methode.

1.4.6 Relatie

Hoewel de verschillende auteurs over het algemeen, ondanks verschillende naamgevingen, op dezelfde manier tegen de verschillende kernbegrippen aankijken, worden relaties tussen verschillende objecten door vrijwel iedere auteur anders behandeld. In [BOO91] wordt gewoon de term *relatie* gebruikt. Alle verbindingen tussen objecten worden beschouwd als relaties, ook de in de volgende paragrafen beschreven generalisatie- en aggregatiestructuren. [BOO91] onderscheidt verschillende soorten relaties, zoals uses relations, inherits relations, instantiates relations en metaclass relations, die elk een eigen notatie kennen.

In [RBP+91] is sprake van *associaties*, die worden omschreven als relaties tussen instanties van twee of meer klassen, die een groep van verbindingen met dezelfde structuur en dezelfde semantiek beschrijven.

[CY91a] gebruikt ook in dit geval meer antropomorfe termen, te weten *instance connection* en *message connection*. Een instance connection vormt de verbinding tussen attributen van verschillende objecten, terwijl een message connection diensten van verschillende objecten verbindt.

[WWW90] spreekt over *samenwerking* tussen objecten. Die samenwerking wordt beschreven met behulp van *contracten*. Ook [MEY88] gebruikt de term contract.

[SNY93] spreekt in dit geval van *verzoek*, terwijl als synoniemen de termen *boodschap*, *methode aanroep* en *functie aanroep* worden gegeven.

Naar onze mening moet in ieder geval onderscheid worden gemaakt tussen statische relaties en dynamische relaties. Met behulp van statische relaties worden verbanden tussen de attributen van verschillende klassen gelegd en met behulp van dynamische relaties kan worden beschreven, welke klassen de operaties van welke andere klassen aan mogen roepen. In deel III worden statische en dynamische relaties met de in [JCJ+92] gebruikte termen *instantie associatie* en *communicatie associatie* aangeduid.

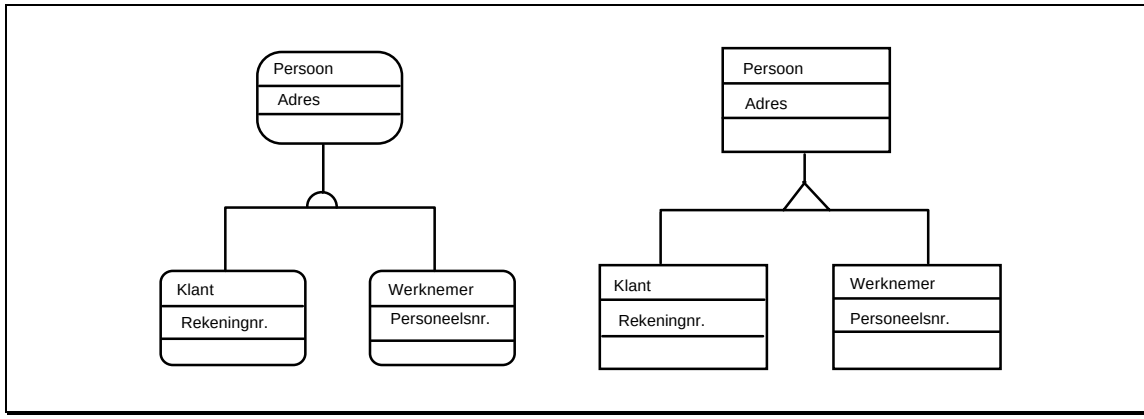
1.4.7 Generalisatie

De laatste twee termen in het rijtje, *generalisatie* en *aggregatie*, zijn in tegenstelling tot de eerder genoemde termen geen in modellen voorkomende elementen, maar structuurvormen, die een onderlinge samenhang tussen die elementen weergeven.

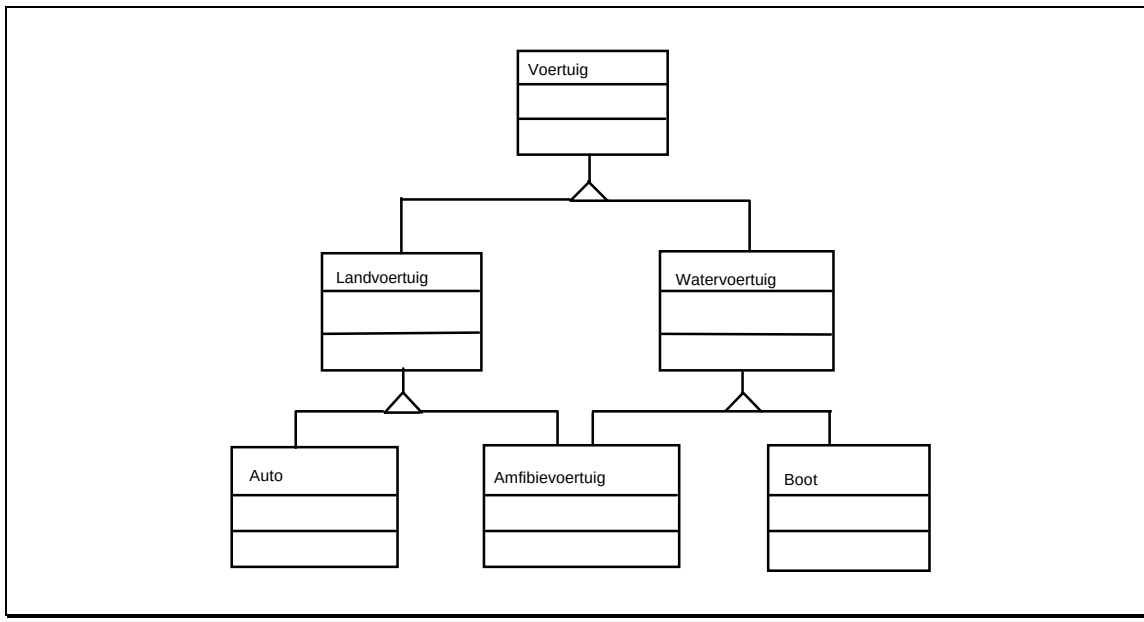
[RBP+91] geeft de volgende definitie van generalisatie: de relatie tussen een klasse en één of meer verfijnde of gespecialiseerde versies van zichzelf. In [CY91a] wordt gesproken over *Gen-Spec* structuren. Bij een generalisatiestructuur is sprake van *superklassen* en *subklassen*, waarbij de subklassen de operaties en attributen van hun superklassen erven. In dit verband wordt ook wel gesproken van *overervingsstructuur* of *inheritance*.

Voorbeeld 1.4.1

Figuur 1.1 is een voorbeeld van een generalisatiestructuur. In dit voorbeeld is sprake van twee verschillende klassen, Klant en Werknemer, die een vergelijkbaar attribuut hebben: Adres. Daarom is een superklasse Persoon ingevoerd, waarin dat attribuut wordt gedefinieerd. Daar zowel Klant als Werknemer subklassen zijn van Persoon, erven zij beide het attribuut Adres van Persoon.



Figuur 1.1: Een voorbeeld van een generalisatiestructuur. Links is gebruik gemaakt van de notatie van Coad en Yourdon [CY91a], rechts van de notatie van Rumbaugh et al. [RBP+91].



Figuur 1.2: Een voorbeeld van veelvuldige generalisatie. Dit voorbeeld is ontleend aan [RBP+91].

Er kunnen twee verschillende vormen van generalisatiestructuren worden onderscheiden: *de generalisatiehiërarchie*, waarin iedere subklasse slechts één directe superklasse heeft, hetgeen bijvoorbeeld in figuur 1.1 het geval is, en *veelvuldige generalisati*e, waarin een subklasse meerdere superklassen mag hebben. [RBP+91] duidt dit aan met de term *veelvuldige overerving*, ofwel *multiple inheritance*. [CY91a] spreekt van *Gen-Spec Lattices*. Bij veelvuldige generalisatie erft een subklasse de operaties van al zijn superklassen.

Voorbeeld 1.4.2

De klasse Voertuig in figuur 1.2 kan worden gesplitst in de klassen Landvoertuig en Watervoertuig. Beide klassen kunnen verder worden gesplitst: Landvoertuig in Auto en Amfibievoertuig en Watervoertuig in Boot en Amfibievoertuig. De klasse Amfibievoertuig is zowel subklasse van de klasse Landvoertuig als van de klasse Watervoertuig en erft van beide superklassen eigenschappen.

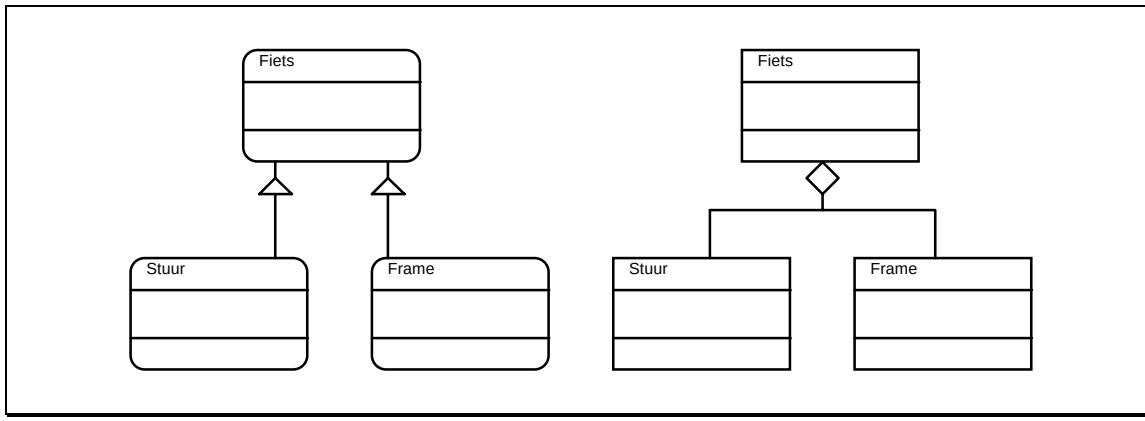
[BRO93] splitst generalisatiestructuren in abstracties en specialisaties. Deze scheiding wordt gebaseerd op het in [RBP+91] aangevoerde onderscheid tussen abstracte klassen en concrete klassen. Object instanties kunnen in [RBP+91] alleen instantie zijn van een *concrete klasse*. *Abstracte klassen* zijn klassen, die zelf geen object instanties kunnen hebben, maar die slechts fungeren als veralgemenisering van een aantal subklassen. Een *abstractie* is een generalisatie, waarbij de superklasse een abstracte klasse is, en een *specialisatie* is een generalisatie, waarbij de superklasse een concrete klasse is.

Voorbeeld 1.4.3

Als de klasse Persoon in de generalisatiestructuur in figuur 1.1 eigen object instanties mag hebben, dan moet de klasse Persoon een concrete klasse zijn en is er sprake van specialisatie. Als de klasse Persoon geen eigen object instanties mag hebben, omdat het een abstracte klasse is, dan is er sprake van abstractie.

1.4.8 Aggregatie

Aggregatie is een structuur, die de samenhang tussen een geheel en zijn componenten beschrijft. Een aantal auteurs, waaronder Rumbaugh et al. [RBP+91], beschouwt aggregatie als een speciaal soort relatie, terwijl bijvoorbeeld Coad en Yourdon in [CY91a] aggregatie zien als een structuur: de *Whole-Part structuur*. Andere auteurs, waaronder Shlaer en Mellor [SM88] en Booch [BOO91], beschouwen aggregatie als een gewone relatie en hebben geen speciale notatie voor aggregatiestructuren.



Figuur 1.3: Een voorbeeld van een aggregatiestructuur. Links is gebruik gemaakt van de notatie van Coad en Yourdon [CY91a], rechts van de notatie van Rumbaugh et al. [RBP+91].

Voorbeeld 1.4.4

Figuur 1.3 is een voorbeeld van een aggregatiestructuur. De klasse Fiets is in dit voorbeeld samengesteld uit object instanties uit de klasse Stuur en de klasse Frame. Object instanties uit de klasse Fiets kunnen op deze manier gebruik maken van de attributen en operaties van hun componenten uit de klassen Stuur en Frame. Niet relevante componenten hoeven niet gemodelleerd te worden. Zo is in dit voorbeeld bijvoorbeeld de verlichting niet gemodelleerd.

1.5 Karakteristieken van Object-Oriëntatie

[COC93] geeft de volgende karakteristieke eigenschappen van Object-georiënteerde systeemontwikkeling:

1. inkapseling van proces en data in zowel de applicatiestructuur als de ontwikkelmethode
2. antropomorf ontwerp
3. modellering van het probleemgebied gedurende het gehele ontwikkeltraject
4. nadruk op hergebruik van ontwerp en code door uitbreidbaarheid
5. incrementele en iteratieve ontwikkeling

In het vervolg van deze sectie wordt nader op de genoemde karakteristieken ingegaan.

1.5.1 Inkapseling van proces en data

Inkapseling, vaak aangeduid met de Engelse term *encapsulation*, is een modellerings- en implementatietechniek, die de externe aspecten van een object scheidt van de interne implementatiedetails [COC93]. In dit geval wordt ook wel gesproken van het *localiteitsprincipe*. Inkapseling is geen nieuw begrip in de informaticawereld. Een taal als Modula2 is geheel gebaseerd op het mechanisme van inkapseling. Het grote verschil tussen inkapseling in een gestructureerde taal als Modula2 en Object-georiënteerde inkapseling is het feit, dat bij Modula2 de inkapseling beperkt blijft tot de functies, terwijl in een OO object zowel de operaties als de attributen van de omgeving worden afgeschermd.

1.5.2 Antropomorf ontwerp

De term *antropomorfisme* staat voor de toeschrijving van menselijke motivaties, karakteristieken of gedragingen aan levenloze objecten, dieren of natuurverschijnselen. Bij OO worden de operaties van een object beschouwd als diensten, die dat object kan leveren aan andere objecten. Andere objecten kunnen die diensten aanvragen door het leverende object te verzoeken die diensten te verzorgen. Een aantal auteurs ([MEY88], [WWW90]) gaat nog verder en spreekt van *contracten*: garanties, dat een object een dienst correct zal leveren. Door hen wordt aan objecten een *verantwoordelijkheid* toegekend, waarmee het doel van het object, de plaats van het object in het systeem en alle diensten, die het object levert, en de daaraan verbonden contracten worden bedoeld.

1.5.3 Modellering van het probleemgebied gedurende het ontwikkeltraject

Bij gestructureerde ontwikkelmethoden wordt in de analysefase het probleemdomein gemodelleerd. Vervolgens wordt dit model gebruikt tijdens de ontwerpfase. Hiervoor is een vertaalslag nodig, omdat de modellen uit de analysefase meestal gebruik maken van andere notaties dan de modellen uit de ontwerpfase. De modellen uit de analysefase dienen in dat geval vooral ter verificatie van de modellen uit de ontwerp-

fase, maar de kans, dat er fouten in een ontwerp sluipen, die niet bij de verificatie worden ontdekt, is groot.

De meeste OO methoden zijn gebaseerd op OOP en beslaan zowel OOA als OOD. Tijdens OOA en OOD wordt gebruik gemaakt van dezelfde of vergelijkbare notaties en technieken. De tijdens de analysefase vervaardigde modellen kunnen tijdens de ontwerpfase worden gebruikt en worden aangevuld met ontwerpaspecten. Bij implementatie met behulp van een OO programmeertaal kan zelfs de implementatie aan de hand van diezelfde modellen geschieden. In een enkel geval is een omzetting nodig van analyse- naar ontwerpmodellen, maar een dergelijke omzetting is in dat geval, in tegenstelling tot de omzettingen bij de meeste conventionele methoden, in detail beschreven. De methode van Shlaer en Mellor (zie [SM88], [SM92]) en OOSE, de methode van Jacobson et al. (zie [JCJ+92]), zijn voorbeelden van dergelijke methoden. Meestal wordt dus gedurende het gehele ontwikkeltraject met één en hetzelfde model gewerkt, dat constant wordt uitgebreid en aangepast. Moeilijke vertaalslagen zijn niet meer nodig, waardoor de kans op een hoge kwaliteit van het eindprodukt wordt vergroot.

1.5.4 Nadruk op hergebruik van ontwerp en code door uitbreidbaarheid

Uitbreidbaarheid, vaak aangeduid met de Engelse term *extensibility*, wordt door OO op drie manieren ondersteund, namelijk in de vorm van klassenbibliotheken, generalisatie en polymorfisme [COC93]. Deze uitbreidbaarheid biedt zowel tijdens het ontwikkelproces zelf als tijdens latere projecten, waarin code en mogelijk ook analyse- en ontwerpmodellen kunnen worden hergebruikt, aanzienlijke voordelen.

Onder klassenbibliotheken worden verstaan verzamelingen van voorgedefinieerde klassen, die niet meer opnieuw ontworpen en geïmplementeerd dienen te worden, maar direct in het ontwikkelproces kunnen worden gebruikt. Voor de betekenis van de termen polymorfisme en generalisatie verwijzen wij naar respectievelijk paragraaf 1.4.7 (Generalisatie) en paragraaf 1.4.5 (Methode).

1.5.5 Incrementele en iteratieve ontwikkeling

Bij incrementele ontwikkeling worden verschillende delen van het systeem ontwikkeld op verschillende tijdstippen of met verschillende snelheden en na voltooiing geïntegreerd in het reeds gecompleteerde deel van het systeem [COC93]. Bij iteratieve ontwikkeling wordt een reeks van probleemoplossingen, die steeds beter voldoen aan de voor het probleem geldende vereisten, uitgewerkt [PIT93]. In dat geval is er sprake van gecontroleerd overdoen van werk. De meeste Object-georiënteerde methoden gaan uit van zowel een iteratieve als een incrementele stijl van systeemontwikkeling.

2 De Voor- en Nadelen van Object-Oriëntatie

2.1 Inleiding

In het vorige hoofdstuk is een beeld geschetst van de Object-georiënteerde benadering. In dit hoofdstuk gaan wij dieper in op de voordelen, die het gebruik van Object-georiënteerde technologie kan bieden, mits zij goed wordt gebruikt. Natuurlijk is ook bij OO sprake van een keerzijde van de medaille. Daarom worden ook de nadelen en de gevaren, die zijn verbonden aan gebruik van Object-georiënteerde technologie, in dit hoofdstuk geschetst.

2.2 Voordelen van OO

Het gebruik van Object-georiënteerde technologie kan aanzienlijke voordelen bieden ten opzichte van traditionele (gestructureerde) systeemontwikkeling. Het is natuurlijk wel noodzakelijk, dat OO technieken correct worden toegepast. [CY91a] gebruikt in dit verband de uitdrukking: "a fool with a tool is still a fool". OO staat niet garant voor betere software; het is alleen een hulpmiddel om tot betere software te komen.

Vrijwel alle auteurs geven in dit verband een opsomming van de door hen geconstateerde voordelen. [MAR92] komt zelfs met een lijst van tweeëndertig voordelen. In deze sectie wordt aandacht besteed aan de belangrijkste aan het gebruik van OO verbonden voordelen.

2.2.1 Betere integratie van data en processen

Het manco van de meeste conventionele analyse- en ontwerpmethoden is de slechte onderlinge afstemming tussen datamodellen en procesmodellen. Vanwege de inkapseling van attributen en operaties in een object is bij OO de onderlinge samenhang tussen data en proces op een natuurlijke wijze geïntegreerd [HOF93]. Hierdoor wordt de kans op inconsistenties tussen de datamodellen en de procesmodellen aanzienlijk gereduceerd.

Een ander gevolg van de betere integratie tussen data en processen is de eliminatie van door meerdere processen gebruikte globale data [SOM89]. Door de inkapseling van data in objecten kunnen alle gegevens alleen worden benaderd via een boodschap aan het verantwoordelijke object. De kans op onverwachte wijzigingen in door meerdere processen gebruikte gegevens wordt hierdoor nihil. Object-georiënteerde systemen vertonen dan ook een lage graad van koppeling.

2.2.2 Betere ondersteuning van hergebruik

Eén van de grootste voordelen van Object-Oriëntatie is de mogelijkheid om beter gebruik te maken van bestaande code en zelfs van bestaande produkten uit de analyse- en ontwerpfase. Het betreft hier vooral hergebruik van klassen. Vanwege de inkapseling hoeft bij hergebruik van klassen geen rekening gehouden te worden met de implementatie van die klassen en vanwege de structuren, die in OO kunnen bestaan tussen verschillende klassen, is het creëren van nieuwe klassen op basis van oude klassen eenvoudiger dan bij gestructureerde ontwikkeling. Het werken met klassen biedt de mogelijkheid om zogenaamde *klassenbibliotheken* aan te maken. Hierin kunnen klassen worden opgeslagen, die zijn gemaakt in het kader van eerdere projecten en die geschikt leken voor hergebruik in andere projecten. Deze klassen kunnen op verschillende manieren in nieuwe systemen worden gebruikt.

- De specificatie van een klasse in de klassenbibliotheek komt overeen met de specificatie van de gewenste klasse. De klasse kan zonder problemen worden gebruikt in het nieuwe systeem. Testen van de klasse zelf is niet meer nodig, omdat dat bij de creatie van de klasse reeds is gebeurd. Wel moet worden getest op de wisselwerking tussen klassen onderling, maar dat zou ook bij de creatie van een nieuwe klasse het geval zijn geweest.
- De specificatie van een klasse in de klassenbibliotheek vertoont een zelfde structuur als de specificatie van de gewenste klasse. De gewenste klasse kan worden geïmplementeerd als subklasse van de reeds bestaande klasse, zodat slechts de operaties, die de reeds bestaande klasse nog niet kende, geïmplementeerd hoeven te worden. De overige operaties worden geërfd van de reeds bestaande klasse.
- De specificatie van een klasse in de klassenbibliotheek komt overeen met de specificatie van de gewenste klasse, maar een aantal componenten van de klasse kan effectiever ontworpen of geïmplementeerd worden. Vanwege het in OO gebruikte polymorfisme (zie paragraaf 1.4.5, Methode) hoeft in de 'geheel' klasse niets veranderd te worden, zolang het interface van de 'componenten' maar niet verandert.

Hergebruik kan op een aantal terreinen voordelen bieden.

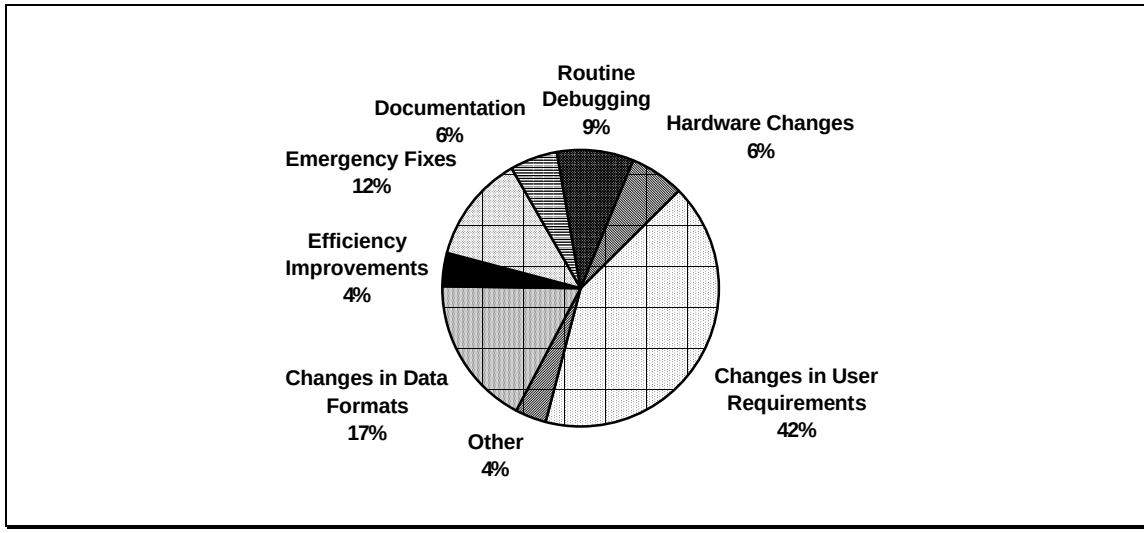
1. Projecten kunnen sneller worden afgerond, omdat gedeelten van het systeem al bestaan en gemakkelijk kunnen worden verwerkt in het nieuwe systeem.
2. Projecten kunnen ten gevolge van de hierboven genoemde tijdwinst tegen lagere kosten worden gerealiseerd. De kosten, die zijn verbonden aan het gebruik van klassenbibliotheken, moeten dan wel lager zijn dan de kosten, die hadden moeten worden gemaakt om de hergebruikte klassen zelf te ontwikkelen.
3. De kans op een hoog kwaliteitsniveau van het nieuwe systeem is groot, omdat de nieuwe software grotendeels is opgebouwd uit bestaande, in de praktijk gebruikte componenten. Object-georiënteerde technologie levert systemen met een hogere kwaliteit, met minder fouten en vaak met betere oplossingen voor de problemen, waar zij voor zijn gebouwd [TAY92].

2.2.3 Beter onderhoud

Vanwege het gebruik van inkapseling in OO heeft een verandering van de implementatie van de ene klasse geen invloed op het gedrag van andere klassen. Hierdoor is het mogelijk om klassen te wijzigen, zonder dat dat effect heeft op andere klassen. Natuurlijk blijft het verstandig om de samenwerking tussen de verschillende klassen opnieuw te testen. Object-georiënteerde software is hierdoor gemakkelijker te wijzigen en te onderhouden dan conventionele software. Dit beperkt de kosten van onderhoud en biedt geautomatiseerde systemen de mogelijkheid om te groeien en te evolueren in antwoord op veranderende omstandigheden [TAY92].

Hoe belangrijk dit voordeel is, kan worden opgemaakt uit figuur 2.1. Uit deze figuur blijkt, dat bijna 60% van alle onderhoudskosten voortkomt uit veranderingen in de wensen van de gebruikers en veranderingen in het formaat van gegevens.

OO biedt daarnaast nog op een andere manier betere mogelijkheden tot onderhoud. Conventioneel ontwikkelde informatiesystemen zijn ontworpen vanuit het oogpunt van interne representatie door de hardware. Veranderingen in de systeemomgeving konden daarom leiden tot aanpassingen in alle delen van een informatiesysteem. Object-georiënteerd ontwikkelde systemen daarentegen pogen de werkelijkheid zo natuurgetrouw mogelijk af te beelden. Veranderingen in de systeemomgeving zullen daarom alleen leiden tot



Figuur 2.1: Indeling van onderhoudskosten (ontleend aan [MEY88]).

aanpassing van die objecten, die de veranderde situatie in de werkelijkheid representeren [WMH93]. Object-georiënteerde systemen kunnen daarom beter worden aangepast aan veranderingen in de systeemomgeving.

2.2.4 Betere communicatie tussen ontwikkelaar en gebruiker

Bij het analyseren van probleemsituaties is de kans op communicatieproblemen tussen de systeemanalisten enerzijds en de toekomstige gebruikers anderzijds altijd aanwezig. Hiervoor kunnen onder andere de onderstaande redenen worden gegeven.

- Verschillende mensen kijken verschillend tegen situaties aan.
- Door zowel de ontwerper als de toekomstige gebruiker worden vaktermen gebruikt, die door de ander niet worden begrepen.
- Verschillende mensen kennen soms verschillende betekenissen toe aan hetzelfde woord.

De meeste, op dit moment gebruikte, conventionele methoden bieden wel mogelijkheden om deze problemen te voorkomen, maar in de praktijk blijken veel eindgebruikers door de scheiding van data en functies toch problemen te hebben met het begrijpen van de vervaardigde modellen.

In paragraaf 1.5.3 (Modellering van het probleemgebied gedurende het ontwikkeltraject) is beschreven, dat één van de karakteristieken van OO de modellering van het probleemdomein tijdens het gehele ontwikkeltraject is. De basis van de vervaardigde modellen van het probleemdomein wordt gevormd door objecten, die een afbeelding vormen van in het probleemdomein voorkomende abstracte en concrete begrippen. In deze objecten zijn zowel de gegevens als de functies, die bij die concepten behoren, ondergebracht. De zo ontstane modellen sluiten beter aan op de manier, waarop de eindgebruikers over het probleemdomein denken. "OOA organizes analysis and specification using the methods of organization which pervade people's thinking" [CY91a].

Wanneer door deze andere manier van modelleren de eindgebruikers daadwerkelijk beter de gepresenteerde modellen begrijpen, zullen zij beter aan kunnen geven, wat er aan die modellen schort. Hierdoor wor-

den de geproduceerde modellen beter gevalideerd, wat bijdraagt aan een verhoogde efficiëntie van het ontwikkelproces.

2.2.5 Betere aansluiting tussen de verschillende ontwikkelfasen

Het in de vorige paragraaf reeds aangehaalde feit, dat bij OO tijdens het gehele ontwikkeltraject het probleemdomein wordt gemodelleerd, heeft nog een tweede positief gevolg: de verschillende fasen in het ontwikkeltraject sluiten beter op elkaar aan dan bij conventionele systeemontwikkeling. Hoewel bij conventionele systeemontwikkeling de eindprodukten van een bepaalde fase wel worden gebruikt als basis voor de produkten van een volgende fase, is meestal een vertaalslag nodig, omdat de verschillende modelleringstechnieken niet goed op elkaar aansluiten. Tijdens deze omzettingen kunnen fouten optreden.

Bij OOA en OOD worden dezelfde modelleringstechnieken gebruikt, waardoor de produkten uit deze fasen een betere consistentie zullen vertonen. Daar de meeste OOD methoden zijn gebaseerd op OOP, is ook de aansluiting tussen OOD en OOP goed en is ook bij die overgang de kans op fouten kleiner.

Deze grotere consistentie geldt niet alleen voor de eindprodukten van de verschillende fasen in het ontwikkeltraject, maar ook voor de produkten, die in één fase worden geproduceerd. Terwijl bij conventionele systeemontwikkeling proces- en datamodellen nauwelijks onderlinge samenhang vertonen, is bij OO die samenhang juist essentieel, omdat attributen en operaties als één intrinsiek geheel worden behandeld [CY91a].

2.3 Nadelen van OO

Ondanks de lofzangen van een aantal auteurs met betrekking tot OO zijn er ook nadelen, verbonden aan het gebruik van OO, aan te geven. Deze nadelen hebben veelal te maken met de overstap van conventionele naar Object-georiënteerde systeemontwikkeling en spelen daarom maar tijdelijk een rol. Hieronder wordt een overzicht gegeven van de belangrijkste aan het gebruik van OO verbonden nadelen.

2.3.1 Informele karakter van de meeste methoden

Hoewel voor de meeste conventionele methoden een formele basis ontbreekt, is de afgelopen jaren vanuit de wetenschappelijke wereld een aantal methoden en technieken van een formele onderbouwing voorzien. Een voorbeeld van een dergelijke formalisatie is [HOF93]. De meeste Object-georiënteerde methoden zijn nog vrij jong en zijn ontstaan in de praktijk. Ook voor deze methoden ontbreekt een formele basis. Een aantal methoden, zoals bijvoorbeeld de methode van Shlaer en Mellor (beschreven in [SM88] en [SM92]), claimt een formele basis te hebben, maar die formele basis wordt ontleend aan het feit, dat gebruik wordt gemaakt van een vastgelegde notatie. Een echte formele basis, bijvoorbeeld in de vorm van algebraïsche specificaties, ontbreekt echter. Een positieve uitzondering op deze regel vormt de OSA methode van Embley et al. De beschrijving van OSA [EKW92] besluit met een meer dan dertig pagina's omvattende formele definitie in de vorm van onder andere eerste-orde predicatenlogica en metamodellen.

Naast het informele karakter wordt in [HOF93] nog een aantal andere modeltechnische nadelen genoemd, zoals het ontbreken van verschillende typeconstructoren, die semantische datamodellen vaak wel kennen (bijvoorbeeld de verzamelingconstructor), het ontstaan van problemen met het localiteitsprincipe, wanneer wordt geprobeerd constraints over meerdere objecten te definiëren, en de onmogelijkheid complexe constraints of dynamisch gedrag formeel te beschrijven.

2.3.2 Nieuwe Investeringen

Om optimaal te kunnen profiteren van de voordelen van OO moet een onderneming, die overstapt naar OO, niet alleen het ontwikkelproces inrichten op basis van de Object-georiënteerde benadering, maar moet zij ook op OO methoden en technieken overgaan. Dit betekent, dat een deel van de huidige hulpmiddelen moet worden vervangen door OO hulpmiddelen. Dat geldt voor programmeeromgevingen, maar ook voor CASE tools en andere hulpmiddelen. Een bedrijf, dat overstapt op OO, zal moeten investeren in nieuwe software.

Daarnaast moet een onderneming, die wil overstappen op OO, mogelijk ook investeren in hardware, zoals opslagcapaciteit, processorsnelheid en intern geheugen. Dit is geen vereiste, maar omdat OO een jonge technologie is, die nog niet is vervolmaakt, kunnen de prestaties op het gebied van responsietijden en dergelijke van Object-georiënteerd ontwikkelde systemen achterblijven bij die van conventioneel ontwikkelde systemen. Wanneer een dergelijke situatie zich voordoet, moet worden gekozen tussen mindere prestaties en investeringen in hardware.

2.3.3 Training van het personeel

De aan een bedrijf verbonden ontwikkelaars zullen meestal nog niet hebben gewerkt met OO methoden en technieken. Om over te kunnen stappen op OO moeten deze mensen worden getraind in het werken met OO. In dit verband wordt in [COC93] onderscheid gemaakt tussen drie strategieën om te migreren naar OO.

1. In eerste instantie worden slechts gedeelten van te ontwikkelen systemen, die uitermate geschikt zijn voor Object-georiënteerde ontwikkeling (bijvoorbeeld gebruikersinterfaces), Object-georiënteerd ontwikkeld. Deze subsystemen worden ontwikkeld door het eigen personeel onder begeleiding van een externe OO expert. Door de training, die de ontwikkelaars op die wijze krijgen, worden zij zelf OO experts, zodat zij in een volgend project als begeleiders van andere ontwikkelaars kunnen worden ingezet. Bij een dergelijke werkwijze wordt de introductie van Object-georiënteerde systeemontwikkeling gespreid over een langere periode en krijgen alle ontwikkelaars een grondige training in OO methoden en technieken.
2. In eerste instantie wordt gekozen voor implementatie in een hybride OO programmeertaal (zie paragraaf 4.3.1, Indeling van programmeertalen), zodat in een project bepaalde subsystemen OO en andere subsystemen conventioneel kunnen worden ontworpen en geïmplementeerd. Bij een dergelijke werkwijze kunnen ontwikkelaars kennis maken met OO, maar wanneer gebruik van OO grote problemen oplevert, kan worden teruggevallen op conventionele technieken. Na verloop van tijd zullen alle ontwikkelaars een grondig inzicht in OO hebben verworven. Op dat moment kan de overstap naar pure OO systeemontwikkeling worden gemaakt.
3. In eerste instantie wordt gekozen voor ontwikkeling met behulp van een hybride OO methode, die volledig compatibel is met zijn niet-OO tegenhanger, bijvoorbeeld een adaptieve OO methode (zie paragraaf 4.2.1, Indeling van methoden). Deze strategie vertoont overeenkomsten met de vorige strategie, maar kent ook een groot verschil: het is mogelijk om tijdens de ontwikkeling van een subsysteem over te stappen van conventionele naar Object-georiënteerde systeemontwikkeling. Ook bij deze werkwijze zullen na verloop van tijd de ontwikkelaars een grondig inzicht in OO hebben verworven, waarna de overstap naar pure OO systeemontwikkeling kan worden gemaakt.

2.3.4 Training van het management

Naast de ontwikkelaars moet ook het management worden getraind in het gebruik van Object-georiënteerde methoden en technieken. Het gaat in dit geval voornamelijk om nieuwe managementtechnieken. Bij Object-georiënteerde systeemontwikkeling speelt een aantal aspecten, die bij conventionele systeemontwikkeling niet aan de orde waren, een rol, zoals het in de volgende paragraaf besproken ontwikkelen met het oog op hergebruik. Hoofdstuk 3 (Management van Object-georiënteerde projecten) is geheel gewijd aan het managen van projecten, die gebruik maken van Object-georiënteerde methoden en technieken.

2.3.5 Ontwikkelen met het oog op hergebruik

Om hergebruik van klassen in toekomstige projecten mogelijk te maken, kan het nodig zijn, dat een aantal abstracte klassen, die voor het huidige project niet noodzakelijk zijn, maar die de kans op hergebruik vergroten, wordt ontworpen en geïmplementeerd. Hierdoor kan een gedeelte van de tijd- en kostenbesparing, die wordt geboekt door hergebruik van klassen, teniet worden gedaan.

2.4 Gevaren bij slordig gebruik

Zoals eerder is opgemerkt, geldt ook bij het gebruik van Object-georiënteerde technologie: "a fool with a tool is still a fool" [CY91a]. Wanneer gebruik wordt gemaakt van OO in plaats van een gestructureerde methode, kan een aantal steeds terugkerende problemen worden voorkomen. Daar staat tegenover, dat bij slordig gebruik van OO nieuwe problemen op kunnen treden. Een aantal van die problemen wordt in deze sectie behandeld.

2.4.1 Hergebruik van slechte produkten

Daar hergebruik van bestaande modellen en code één van de grote voordelen van OO is, moeten alle produkten, die geschikt zijn voor hergebruik, kwalitatief gezien van een hoog niveau zijn. Een niet opgemerkte fout kan niet alleen het huidige systeem ontregelen, maar ook toekomstige systemen, die van dezelfde klassen gebruik maken. Daarom moet extra aandacht worden besteed aan het testen van modellen en klassen, die worden opgenomen in de bibliotheek van opnieuw te gebruiken produkten. Daarnaast moet bij hergebruik iedere keer de koppeling met andere klassen worden getest [COC93].

2.4.2 Onnodige generalisaties

Als het generalisatiemechanisme goed wordt gebruikt, dan kan de totale hoeveelheid code aanzienlijk worden beperkt. Het is echter onverstandig om te generaliseren, als daar geen reden toe is. Het model ziet er door toepassing van generalisatie misschien beter uit, maar onnodige generalisatie brengt wel de nodige gevaren met zich mee. Wanneer tijdens het onderhoud blijkt, dat de implementatie van een superklasse een wezenlijke fout bevat, moeten normaal gesproken ook alle subklassen opnieuw worden bekeken en getest. Het onnodig toepassen van generalisatie kan daardoor leiden tot extra onderhoud, dat normaal gesproken achterwege had kunnen blijven. Het generalisatiemechanisme mag daarom nooit worden toegepast, als een verzameling objecten toevallig tijdelijk een vergelijkbare structuur bezit [COC93].

2.4.3 Ongewenst polymorfisme

Dankzij het polymorfismemechanisme kunnen operaties efficiënt worden geïmplementeerd. Polymorfisme brengt echter wel enkele gevaren met zich mee. De situatie, die wordt beschreven in voorbeeld 2.4.1, is een voorbeeld van het ongewenst optreden van polymorfisme. De in dit voorbeeld beschreven fout wordt in [COC93] aangeduid met de term *superoverriding*.

Voorbeeld 2.4.1

Subklasse B kent een operatie o1 met de naam "O", die niet gedefinieerd is in klasse A, een superklasse van B. Tijdens onderhoud wordt aan klasse A een nieuwe operatie o2 toegevoegd, die ook de naam "O" krijgt. Deze operatie wordt door alle subklassen van A geërfd, maar omdat subklasse B al een operatie met de naam "O" kent, ziet de compiler de methode, waarmee o1 in B wordt geïmplementeerd, als een alternatieve methode voor de operatie o2. De aanroep van de operatie met de naam "O" leidt daarom altijd tot de uitvoering van de methode van operatie o1, ook als het vragende object verwacht, dat operatie o2 zal worden uitgevoerd. In dit geval is dus sprake van het ongewenst optreden van polymorfisme.

2.4.4 Hybride methoden en talen

Bij het gebruik van adaptieve en combinatieve methoden (zie paragraaf 4.2.1, Indeling van methoden) en hybride programmeertalen (zie paragraaf 4.3.1, Indeling van programmeertalen) treedt er nog een extra gevaar op. Gebruik van dergelijke hulpmiddelen staat namelijk niet garant voor Object-georiënteerde systeemontwikkeling. Het blijft mogelijk om in bijvoorbeeld C++ conventionele C programmatuur te schrijven. Ervaren programmeurs blijken in de praktijk wel een aantal handige eigenschappen van OOP over te nemen, maar hun denkwijze blijft in veel gevallen procedureel [TAY92].

Het gebruik van hybride methoden en programmeertalen heeft het voordeel, dat de training niet zo lang hoeft te duren (zie paragraaf 2.3.3, Training van het personeel), maar voor een optimaal resultaat dient de wijze, waarop die methoden en programmeertalen worden gebruikt, nauwgezet gecontroleerd te worden.

3 Management van Object-georiënteerde projecten

3.1 Inleiding

Zoals in het vorige hoofdstuk is gesteld, moet niet alleen het ontwikkelproces worden aangepast bij invoering van Object-georiënteerde systeemontwikkeling. Om effectief gebruik van OO mogelijk te maken moet ook het management zich aan deze nieuwe manier van denken aanpassen. "It turns out that serious object-oriented systems are not grown. They are managed, using techniques and skills based on existing project-management experience. The management of such systems, however, must take into account the nature of object-oriented development" [PIT93].

Een aantal auteurs heeft zich beziggehouden met het management van OO en met verwante zaken als kosten/baten analyses. In dit hoofdstuk geven wij een overzicht van hun bevindingen.

3.2 Plannen van hergebruik

Zoals eerder is gebleken, is beter hergebruik van bestaande produkten één van de belangrijkste voordelen, die gebruik van Object-georiënteerde technologie met zich meebrengt. Het is dan wel noodzakelijk, dat hergebruik expliciet als één van de projectdoelen wordt genoemd. Als dat niet gebeurt, dan zal een project geen gebruik maken van herbruikbare produkten en evenmin herbruikbare produkten opleveren. "Reuse happens by design, not by accident" [PIT93].

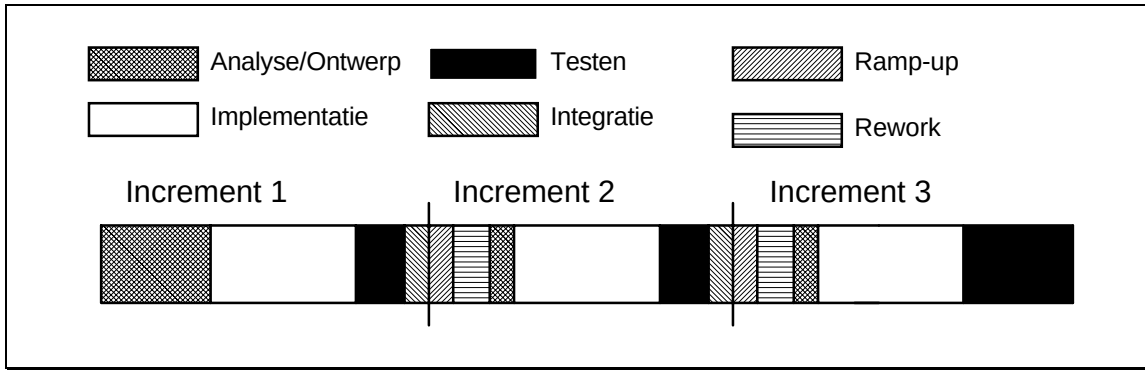
In het projectplan dient duidelijk omschreven te worden, wat er met het oog op hergebruik gedaan dient te worden en op welke manier dit dient te gebeuren. [PIT93] geeft hiervoor een aantal tips en richtlijnen. De belangrijkste zijn:

- stel goed gedefinieerde, haalbare doelen met betrekking tot hergebruik
- weeg de kosten, die gepaard gaan met hergebruik, goed af tegen de gestelde doelen
- maak gebruik van in het probleemgebied gespecialiseerde mensen
- onderken de extra inspanning, die hergebruik met zich meebrengt
- kijk goed naar de mogelijkheden om door anderen ontwikkelde, herbruikbare componenten te gebruiken

3.3 Plannen van iteratieve en incrementele ontwikkeling

Zoals uit paragraaf 1.5.5 (Incrementele en iteratieve ontwikkeling) is gebleken, is een iteratieve en incrementele stijl van ontwikkelen één van de karakteristieken van OO. Hier moet in het projectplan rekening mee worden gehouden. In [PIT93] worden hiervoor vrij gedetailleerde richtlijnen gegeven. De belangrijkste richtlijnen worden hieronder genoemd.

- Het project moet worden verdeeld in incrementen. Het aantal incrementen en het criterium, waarop die indeling is gebaseerd, zijn afhankelijk van het project, al kunnen enkele algemene benaderingen worden aangegeven.
- Ieder increment moet worden verdeeld in fasen voor analyse, ontwerp, implementatie en testen, waarbij geldt, dat 80% van het totale analyse- en ontwerpwerk in het eerste increment moet worden gepland en 50% van het testwerk in het laatste increment.



Figuur 3.1: De in [PIT93] voorgestelde fasering van een project.

- In de planning moet tussen ieder paar elkaar opvolgende incrementen een periode voor integratie, een periode voor 'ramp-up' en een periode voor het verbeteren van fouten (in de meeste literatuur 'rework' genoemd) worden opgenomen.

Figuur 3.1 is een aan [PIT93] ontleende schematische weergave van de indeling van een willekeurig project in incrementen en fasen.

Over de verdeling van het totale werk over de verschillende fasen in het ontwikkeltraject hebben bijna alle auteurs eigen, vaak tegengestelde, denkbeelden.

3.4 Personeelsbezetting

[BOO91] onderscheidt de volgende categorieën systeemontwikkelaars:

- systeemarchitecten, die de strategische beslissingen op het gebied van de systeemarchitectuur nemen
- klasseontwerpers, die de klassestructuren en de onderlinge relaties tussen de klassen bepalen
- klasseïmplementatoren, die de interfaces van de klassen en de interne implementaties daarvan bepalen
- applicatieprogrammeurs, die het systeem programmeren

Jonge, onervaren ontwikkelaars zouden moeten beginnen als applicatieprogrammeurs en in de loop van de tijd moeten doorstromen naar meer verantwoordelijke functies.

De ervaring leert, dat het niveau van ervaring en training van de ontwikkelaars een kritieke succesfactor is bij Object-georiënteerde systeemontwikkeling [PIT93].

3.5 Meten aan Object-georiënteerd ontwikkelen

Op basis van de voorgaande secties zal het duidelijk zijn, dat traditionele methoden voor het voorspellen van de kosten van systeemontwikkeling niet zomaar kunnen worden overgenomen bij Object-georiënteerde systeemontwikkeling.

[PIT93] beschrijft een aangepaste versie van het Constructive Cost Model (CoCoMo) van Barry Boehm [BOE81]. In [BG90] worden, eveneens op basis van CoCoMo, formules voor het schatten van de kosten van hergebruik en prototyping afgeleid.

[BS93] geeft een aantal parameters, waarmee de kwaliteit van klassen kan worden bepaald, zoals het gemiddelde aantal methoden per klasse, het gemiddelde, minimale en maximale aantal variabelen per methode, de gemiddelde, minimale en maximale cyclomatische complexiteit van methoden, het gemiddelde aantal regels code per methode, het gemiddelde aantal boodschappen aan de eigen object instantie per methode, het gemiddelde aantal boodschappen aan andere object instanties per methode en het gemiddelde aantal locale variabelen per methode. Daarnaast wordt beschreven, hoe dergelijke parameters met behulp van geautomatiseerde hulpmiddelen uit de code kunnen worden afgeleid.

[HEN93] geeft een model, waarin de kosten van hergebruik worden afgezet tegen de opbrengsten. De kosten *S* van een project, waarin niet wordt gekeken naar hergebruik, kunnen op de onderstaande wijze worden berekend.[2]

$$S = \sum_{i=1}^{N} s_i = \sum_{i=1}^{N} (A_i W_A + M_i W_M),$$

waarbij *N* staat voor het aantal klassen en s_i voor de kosten om klasse *i* te ontwikkelen. A_i en M_i staan voor respectievelijk het aantal attributen en het aantal methoden van klasse *i*, die met behulp van bijvoorbeeld Functie Punt Analyse (FPA) kunnen worden bepaald, en W_A en W_M zijn de daarbij behorende weegfactoren.

Voor de kosten C_D om *m* klassen nieuw te ontwikkelen geldt in dat geval

$$C_D = \sum_{i=1}^{m} (A_i W_A + M_i W_M).$$

Als de kosten om een klasse in een klassenbibliotheek te vinden *F* bedragen, dan geldt voor de kosten C_R om *k* klassen opnieuw te gebruiken

$$C_R = \sum_{i=1}^{k} F = kF.$$

Voor de kosten C_M om *l* klassen uit de klassenbibliotheek te modificeren geldt

$$C_M = \sum_{i=k+1}^{k+l} [s_i (1 - J_i) + F],$$

waarbij J_i staat voor het deel van de code van klasse *i*, dat niet gemodificeerd hoeft te worden, en s_i kan worden bepaald op de wijze, die hierboven is beschreven.

De totale kosten *C* om een systeem te ontwikkelen, zonder rekening te houden met toekomstig hergebruik, kunnen in dat geval op de onderstaande wijze worden berekend.

$$C = C_R + C_M + C_D.$$

[2] In [HEN93] wordt gebruik gemaakt van een minder formele notatie, waarschijnlijk om de leesbaarheid van de formules te verhogen. Wij hebben met het oog op de toepasbaarheid gekozen voor een meer formele notatie.

Voor de kosten C_G om van de N klassen, die het systeem vormen, P klassen te ontwikkelen met het oog op toekomstig hergebruik geldt

$$C_G = \sum_{i=1}^{P} g_i ,$$

waarbij g_i de kosten zijn, die gemaakt moeten worden om klasse i te ontwikkelen voor hergebruik. In [HEN93] wordt niet ingegaan op de wijze, waarop g_i kan worden bepaald. Er wordt alleen opgemerkt, dat g_i normaal gesproken afhangt van s_i. In [PIT93] wordt in dit verband opgemerkt, dat het documenteren, ontwikkelen en testen van herbruikbare componenten ruwweg drie keer zo veel werk vergt als van vergelijkbare niet-herbruikbare componenten.

De verhouding tussen de besparingen door het gebruik van herbruikbare klassen en de kosten om een aantal klassen van het systeem geschikt te maken voor toekomstig hergebruik wordt R genoemd.

$$R = \frac{S - (C + C_G)}{C_G} = \frac{S - C}{C_G} - 1$$

De gemiddelde waarde van R, genomen over Q projecten, wordt gegeven door

$$\overline{R} = \frac{1}{Q} \sum_{q=1}^{Q} \frac{S_q - C_q}{C_{Gq}} - 1.$$

Hierbij stellen S_q, C_q en C_{Gq} respectievelijk S, C en C_G in project q voor.

De verhouding tussen de cumulatieve besparingen door hergebruik en de daaraan verbonden cumulatieve kosten wordt gegeven door

$$R_Q = \frac{\sum_{q=1}^{Q} S_q - C_q - C_{G_q}}{\sum_{q=1}^{Q} C_{G_q}}.$$

Deze drie verhoudingen worden in [HEN93] de *return on investment* (ROI) indices genoemd.

In [HEN93] wordt tevens beschreven, hoe op basis van dit model een simulatiemodel is gemaakt, waarin voor bepaalde waarden van de parameters de waarden van de ROI indices worden berekend. Op basis van een aantal experimenten wordt geconcludeerd, dat over het algemeen de besparingen door hergebruik pas zullen optreden vanaf het tweede of derde project. Voor een gedetailleerde beschrijving verwijzen wij naar [HEN93].

4 Methoden, talen en tools

4.1 Inleiding

In dit hoofdstuk wordt een globaal overzicht gegeven van de huidige stand van zaken op het gebied van Object-georiënteerde technologie. Daarbij wordt aandacht besteed aan methoden en technieken, programmeertalen, database management systemen, tools en standaarden. Aangezien het te ver voert om in deze scriptie dit alles in detail te beschrijven en een groot gedeelte tijdstipgebonden is en over een paar jaar verouderd kan zijn, wordt veelal volstaan met korte beschrijvingen en verwijzingen naar relevante literatuur.

4.2 Methoden en technieken

In sectie 1.2 (De geschiedenis van Object-Oriëntatie) is opgemerkt, dat OO aanvankelijk beperkt bleef tot het programmeren en dat pas in 1986 in [BOO86] voor het eerst aandacht werd besteed aan OOA en OOD. Daarna is de ene na de andere methode voor OOA en OOD verschenen. Zo wordt in [CF92] aandacht besteed aan twaalf gepubliceerde methoden, terwijl in [MP92], dat nog in hetzelfde jaar verscheen, al drieëntwintig methoden worden vergeleken en melding wordt gemaakt van het feit, dat sinds de afronding van het onderzoek nog minimaal vier nieuwe methoden zijn verschenen.

In deze sectie wordt eerst een in [MP92] voorgestelde indeling van methoden behandeld. Vervolgens wordt een aantal methoden, waarnaar vaak wordt verwezen, beschreven. De verschillende methoden worden in alfabetische volgorde behandeld.

4.2.1 Indeling van methoden

In [MP92] worden Object-georiënteerde methoden gegroepeerd op basis van een drietal benaderingen, te weten:

- de adaptieve benadering
- de combinatieve benadering
- de pure benadering

De *adaptieve benadering* omvat die methoden, die op een nieuwe, Object-georiënteerde manier gebruik maken van bestaande technieken, of bestaande technieken uitbreiden om Object-Oriëntatie te kunnen bevatten [MP92]. Deze methoden verdienen vooral aanbeveling, als tijdens het ontwikkelproces gebruik moet worden gemaakt van bestaande modellen.

De *combinatieve benadering* omvat die methoden, die gebruik maken van Object-georiënteerde, functiegeoriënteerde en/of dynamiek-georiënteerde technieken, die los van elkaar de structuur, de functionaliteit en/of het dynamisch gedrag modelleren en die voorzien in een methode om die verschillende modellen te integreren [MP92]. [COC93] gebruikt voor deze methoden de afkorting *C-R-A-B* (Classes, Relationships, Attributes, Behavior). Het grote voordeel van de combinatieve benadering is het gebruik van bestaande technieken zoals dataflow diagrammen en entity-relationship diagrammen.

De *pure benadering* omvat die methoden, die gebruik maken van nieuwe technieken om objectstructuur, functionaliteit en dynamisch gedrag te modelleren [MP92]. Deze groep methoden wordt in [COC93] aangeduid met de afkorting *B-O-R-D* (Behavior, Objects, Relations, Dynamics), omdat deze methoden de na-

druk leggen op het gedrag van objecten. Het antropomorfe karakter (zie paragraaf 1.5.2, Antropomorf ontwerp) van objecten speelt vooral in deze benadering een belangrijke rol.

4.2.2 OOD van Booch

De OOD methode van Grady Booch werd voor het eerst beschreven in [BOO86] en is uitvoeriger beschreven in het boek "Object Oriented Design with Applications" [BOO91]. Inmiddels is een tweede editie van dat boek verschenen onder de titel "Object-Oriented Analysis and Design with Applications" [BOO94], waarover in een boekbespreking in de Journal of Object-Oriented Programming [WIL94] wordt opgemerkt: "De nieuwe editie besteedt veel meer aandacht aan de analysefase, legt een aantal van de onderliggende concepten beter uit en presenteert een geëvolueerde notatie"[3]. Aangezien wij, gezien de recente verschijningsdatum, niet de gelegenheid hebben gehad om [BOO94] te bestuderen, beperken wij ons hier tot [BOO91]. De methode van Booch maakt gebruik van een geheel nieuwe notatie en kan daarom worden gezien als een pure methode.

OOD beslaat, zoals de naam doet vermoeden, alleen de ontwerpfase. Hierbij moet wel worden aangetekend, dat Booch de grens tussen analyse en ontwerp anders plaatst dan de meeste auteurs. Voor Booch beperkt de analysefase zich tot het beschrijven van de door de klant gewenste vereisten en komt modelleren pas in de ontwerpfase aan de orde.

Booch onderscheidt vier activiteiten:

- het identificeren van klassen en objecten op een gegeven niveau van abstractie
- het identificeren van de semantiek van deze klassen en objecten
- het identificeren van de relaties tussen deze klassen en objecten
- het implementeren van deze klassen en objecten

Deze activiteiten moeten in de gegeven volgorde worden doorlopen. Iteratieslagen zijn toegestaan en waarschijnlijk zelfs onvermijdelijk.

De vier activiteiten van Booch leiden tot modellen voor een aantal gezichtspunten:

- het logische model, waarin de belangrijkste abstracties, waaruit het ontwerp is opgebouwd, worden beschreven en dat bestaat uit *class diagrams* en *object diagrams*
- het fysieke model, dat de fysieke structuur van het systeem beschrijft en dat bestaat uit *module diagrams* en *process diagrams*
- het statische model, dat wordt gevormd door het logische model en het fysieke model
- het dynamische model, dat de dynamiek van het systeem beschrijft en dat bestaat uit *state transition diagrams* en *timing diagrams*

Voor elk van de hierboven genoemde diagrammen komt Booch met een eigen notatie, waarin niet alleen voor typisch Object-georiënteerde aspecten als abstractie, inkapseling, modulariteit en hiërarchie symbolen worden geïntroduceerd, maar ook voor bijvoorbeeld parallellisme, persistentie (wat gebeurt er met objecten, die blijven bestaan, wanneer het systeem, waarin zij bestonden, termineert) en typering. Hierdoor worden de modellen en dus ook de methode vrij complex.

---

[3] Voor de duidelijkheid is de originele tekst van het citaat door ons naar het Nederlands vertaald.

Zoals de titel van [BOO91] reeds aangeeft, besteed Booch ruimschoots aandacht aan voorbeelden. Hij beschrijft gedetailleerd zes praktijkvoorbeelden, waaraan bijna de helft van de bijna 500 pagina's tekst is gewijd. Daarnaast gaat Booch kort in op het managen van Object-georiënteerde projecten.

4.2.3 OOA/OOD van Coad en Yourdon

De OOA/OOD methode van Coad en Yourdon wordt gepresenteerd in [CY91a] en [CY91b]. In [CN93] wordt een aansluitende methode voor OOP gegeven, die wij hier buiten beschouwing laten. In [PRO94] wordt melding gemaakt van het verschijnen van [YOU94]. De titel daarvan ("Object-Oriented Systems Design: An Integrated Approach") doet vermoeden, dat het hier de opvolger van [CY91b] betreft. Aangezien wij, gezien de recente verschijningsdatum, niet de gelegenheid hebben gehad om [YOU94] te bestuderen, beperken wij ons hier tot [CY91a] en [CY91b].

De methode van Coad en Yourdon is een combinatieve methode, die "builds upon the best concepts from Information Modeling, Object-Oriented Programming Languages, and Knowledge-Based Systems - *the concepts which have a solid basis in underlying principles for managing complexity*" [CY91a].

Tijdens de analysefase worden vijf activiteiten onderscheiden:

- het vinden van *Class-&-Objects* (klassen en bijbehorende object instanties)
- het identificeren van structuren
- het identificeren van subjecten (verzamelingen van klassen, die onderlinge samenhang vertonen)
- het definiëren van attributen
- het definiëren van diensten (operaties)

Coad en Yourdon benadrukken, dat deze activiteiten niet sequentieel uitgevoerd hoeven te worden en dat de bovenstaande volgorde alleen de meest voor de hand liggende volgorde is. De analysefase levert een drietal produkten op:

- een vijflagen OOA model, met een laag voor iedere hierboven genoemde activiteit
- Class-&-Objects specification templates voor alle klassen
- aanvullende documentatie, zoals bijvoorbeeld een lijst met kritieke executiepaden, aanvullende systeemeisen en *services/states tables*

Class-&-Objects specification templates bevatten een specificatie van de bij de klasse behorende attributen, externe input, externe output, additionele vereisten en diensten en eventueel commentaar, traceability codes, applicable state codes en tijd- en geheugeneisen. Verder bevat een template een *object state diagram* voor de betreffende klasse en *service charts* voor alle diensten in de klasse.

In OOD wordt het vijflagen OOA model uitgebouwd tot een viercomponenten/vijflagen OOD model. De vier componenten zijn:

- de problem domain component
- de human interaction component
- de task management component
- de data management component

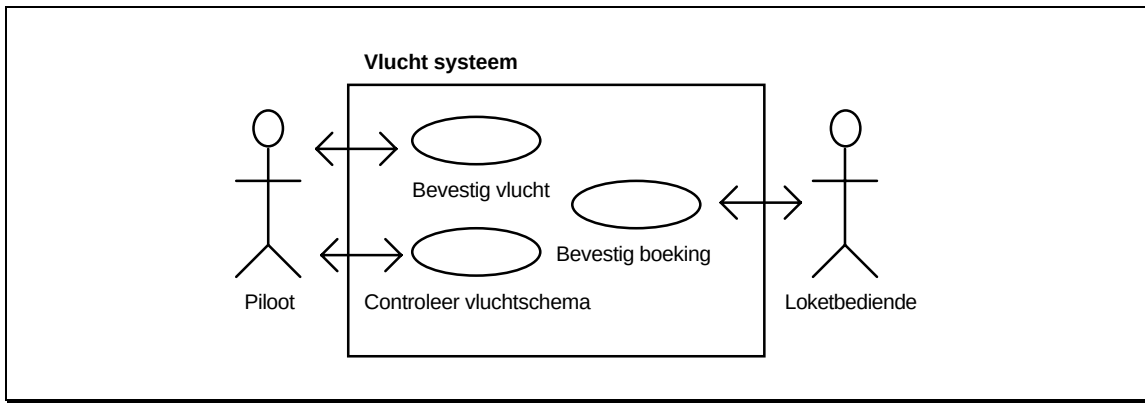
Iedere component bestaat uit de vijf eerder genoemde lagen. De eindprodukten van de analysefase vormen de basis voor de ontwerpfase. In beide fasen wordt dezelfde notatie gebruikt.

De gehele OOA/OOD methode beschouwend, kan worden opgemerkt, dat tijdens de analysefase het probleemdomein wordt gemodelleerd, waarna tijdens de ontwerpfase implementatiedetails, zoals gebruikersinterface, data management en task management, worden toegevoegd. Voor OOA en OOD worden dezelfde technieken en notaties gebruikt, waardoor OOA modellen zonder vertaalslag tijdens OOD kunnen worden gebruikt. Maar ook binnen een fase is de relatie tussen objectmodellering, procesmodellering en functiemodellering duidelijk vastgelegd, zodat consistentie tussen de verschillende modellen is gewaarborgd. Voor zowel OOA als OOD zijn tools verkrijgbaar (respectievelijk OOATool en OODTool). Een minpunt is het ontbreken van goed uitgewerkte praktijkvoorbeelden, waardoor een aantal kleine details onduidelijk blijft.

4.2.4 OOSE van Jacobson et al.

Object-Oriented Software Engineering (OOSE), de door Ivar Jacobson, Magnus Christerson, Patrik Jonsson en Gunnar Øvergaard voorgestelde methode, wordt beschreven in [JCJ+92]. OOSE is een verkorte presentatie van de door de auteurs ontwikkelde Objectory methode. Deze methode is ontwikkeld in de context van telecommunicatiesystemen (Ericsson) [PRO94] en wordt in Europa al een aantal jaren gebruikt door Objective Systems, een door Jacobson opgericht bedrijf, waaraan ook de drie co-auteurs zijn verbonden.

OOSE beslaat het gehele ontwikkelproces, dat door de schrijvers wordt opgedeeld in analyse, constructie en testen. Centraal tijdens het hele proces staat de *use case*, een beschrijving van een functionele vereiste van het systeem op basis van entiteiten buiten het systeem (*actors*). Deze use cases worden gedefinieerd tijdens het eerste deel van de analysefase en blijven daarna een rol spelen in het gehele ontwikkelproces. Naast use cases introduceert OOSE een aantal andere nieuwe technieken, zodat OOSE mag worden gezien als een pure Object-georiënteerde methode.



Figuur 4.1: Een voorbeeld van een use case model. Het systeem wordt begrensd door een rechthoek. De actors worden voorgesteld door personen buiten het systeem, de use cases door ellipsen binnen het systeem. Dit voorbeeld is ontleend aan [JCJ+92].

Voorbeeld 4.2.1

Figuur 4.1 is een voorbeeld van een use case model. De entiteiten buiten het systeem, ofwel de actors, zijn de Piloot en de Loketbediende. De use cases Bevestig vlucht en Controleer vluchtschema zijn functionele vereisten van het Vlucht systeem, waarmee de Piloot te maken heeft. De use case Bevestig boeking is een functionele vereiste van het systeem, waar de Loketbediende mee moet kunnen werken.

Tijdens de analysefase wordt op basis van een eerder opgestelde *requirements specification* begonnen met de *requirements analysis*. Deze activiteit resulteert in een *requirements model*, dat bestaat uit een *use case model* (zie voorbeeld 4.2.1), *interface descriptions*, die de interfaces tussen het systeem en de actors beschrijven, en een *problem domain model*, een eerste model van de in het systeem voorkomende objecten. Vervolgens wordt op basis van dit *requirements model* een *robustness analysis* uitgevoerd, die resulteert in het *analysis model*, waarin de drie dimensies van een systeem, te weten informatie, presentatie en gedrag, met behulp van respectievelijk *entiteit objecten*, *interface objecten* en *controle objecten* worden beschreven.

De constructiefase begint met de ontwerpfase, waarin op basis van het requirements model en het analysis model een *design model* wordt vervaardigd, dat naast een nieuw objectmodel bestaat uit *interaction diagrams*, die de interactie tussen objecten beschrijven, en *state transition graphs*, die het gedrag van objecten beschrijven. Vervolgens wordt tijdens de implementatiefase dit design model omgezet in een *implementation model*, de code van het te ontwikkelen systeem. Tijdens de implementatiefase kan door het werken met componenten de herbruikbaarheid van (delen van) de code worden vergroot.

Tenslotte wordt het implementation model op basis van het requirements model en het design model tijdens de testfase door middel van unit tests, integratietests en een systeemtest getest en worden de specificaties en resultaten van deze tests beschreven in een *test model*.

OOSE kent een aantal sterke punten. Het use case model, een middel om functionele vereisten te modelleren, is een concept, dat in geen enkele andere methode terug te vinden is, maar dat wel degelijk waarde kan hebben tijdens het ontwikkelproces, omdat de vereisten beter kunnen worden vastgelegd en daardoor beter kan worden gecontroleerd, of een systeem aan de vereisten voldoet. Het use case model wordt daarom niet alleen gebruikt tijdens de aansluitende robustness analysis, maar ook tijdens de ontwerpfase en de testfase, waarin respectievelijk het design model en het test model getoetst dienen te worden aan het requirements model.

Tijdens de analysefase wordt gewerkt met slechts één model, waarin zowel informatie (datamodellering), presentatie (interfacemodellering) als controle (procesmodellering) worden gemodelleerd. Hierdoor wordt een goede samenhang tussen de verschillende dimensies van een systeem gewaarborgd. Dit model wordt rechtstreeks afgeleid uit het use case model en dient zelf als basis voor het design model. De eerste versie van het design model is het analysis model, waarin de cirkels zijn vervangen door rechthoeken. Het implementation model kan op zijn beurt rechtstreeks worden afgeleid uit het uiteindelijke design model. De vertaalslagen, die nodig zijn om produkten van de ene fase te kunnen gebruiken in de volgende fase, zijn dus duidelijk vastgelegd, waardoor de kans op fouten kleiner wordt.

Daarnaast kent OOSE subsystemen. Het subsysteemmechanisme is een recursief mechanisme, waardoor beter kan worden omgegaan met complexe systemen. OOSE kent ook een tegenhanger van subsystemen tijdens de implementatiefase: de hierboven reeds genoemde componenten. Met behulp van deze componenten kan ook tijdens de implementatiefase de complexiteit van systemen in de hand worden gehouden.

Om het ontwikkelen van real-time systemen met behulp van OOSE mogelijk te maken is aan de modelleringstechnieken een aantal concepten toegevoegd, die in afzonderlijke hoofdstukken worden behandeld.

Verder is een hoofdstuk gewijd aan de koppeling van systemen met zowel Object-georiënteerde als relationele DBMSs en ook aan het managen van OO is een hoofdstuk gewijd. Tenslotte worden twee voorbeelden uit de praktijk in een grote mate van detail uitgewerkt.

Natuurlijk zijn op OOSE enkele aanmerkingen te maken. Hoewel nergens een voorkeur voor een bepaalde programmeertaal wordt uitgesproken, kunnen bepaalde concepten, zoals bijvoorbeeld meervoudige overerving of implementatie van componenten, gemakkelijker in bijvoorbeeld C++ dan in Smalltalk worden gerealiseerd. In voorbeelden, waarin delen van het ontwerp in code worden uitgewerkt, wordt consequent gebruik gemaakt van C++. Op deze manier wordt de indruk gewekt, dat OOSE alleen geschikt is voor implementatie met behulp van C++, hetgeen niet het geval is.

OOSE claimt te beschikken over een formele basis, die wordt ontleend aan het bestaan van metamodellen. Een wiskundige basis wordt in [JCJ+92] echter niet gepresenteerd.

Overigens is OOSE een verkorte vorm van de door Objective Systems gebruikte Objectory methode, waarin alleen de fundamentele ideeën achter de Objectory methode worden gepresenteerd. Volgens de auteurs is voor een bruikbare implementatie van de methode de complete Objectory documentatie vereist, die bestaat uit meer dan 1200 pagina's. Objectory wordt ondersteund door OrySE, de Objectory Support Environment.

4.2.5 De methode van Martin en Odell

De methode van James Martin en James Odell wordt gepresenteerd in twee los van elkaar te lezen boeken. In [MO92] ligt de nadruk op de tijdens de analyse- en ontwerpfase gebruikte methoden en technieken, terwijl in [MAR92] de nadruk ligt op de wijze, waarop de methode in een onderneming moet worden ingevoerd. Beide boeken wijden enkele hoofdstukken aan elkaars centrale onderwerpen, maar voor een gedetailleerd overzicht van de gehele methode is bestudering van beide boeken noodzakelijk.

De methode van Martin en Odell sluit nauw aan op het eerder gepubliceerde werk van James Martin en is ingepast in het *Information Engineering* raamwerk ([MAR90a], [MAR90b]). De gebruikte notaties zijn uitbreidingen op de notaties, die in [MAR87] worden voorgesteld. De methode omvat het gehele ontwikkelproces, maar de nadruk ligt op de analysefase.

De analysefase wordt opgedeeld in twee nauw samenhangende activiteiten, te weten objectstructuuranalyse en objectgedraganalyse. De objectstructuur wordt gemodelleerd met behulp van *Object-Relationship diagrams*: Entity-Relationship diagrams met een uitbreiding, die het mogelijk maakt om generalisatie te modelleren. Voor het modelleren van objectgedrag worden twee soorten diagrammen gebruikt: *Event diagrams* voor een gedetailleerde beschrijving van de processen en *Object-Flow diagrams* voor een globale beschrijving. De twee soorten diagrammen zijn uitwisselbaar: een Object-Flow diagram kan zowel in nieuwe Object-Flow diagrams als in Event diagrams worden gedecomponeerd. Martin en Odell staan zelfs het decomponeren van het ene deel van de processen in Object-Flow diagrams en het andere deel in Event diagrams toe.

Martin en Odell gaan uit van *meervoudige classificatie*: een object instantie kan instantie zijn van meerdere objectklassen tegelijk. Meervoudige classificatie wordt door geen enkele andere methode ondersteund en kan niet rechtstreeks worden geïmplementeerd met behulp van de huidige OO programmeertalen. Daarom zijn omwegen nodig om dit concept te implementeren. Die omwegen worden beschreven in één van de aan de ontwerpfase gewijde hoofdstukken van [MO92]. Door het toestaan van meervoudige classificatie moeten Martin en Odell een andere invulling geven aan het begrip *object toestand*. In de methode van Martin en Odell is de toestand van een object op een bepaald moment gedefinieerd als de verzameling objectklassen, waarvan dat object op dat moment instantie is.

Het analyseproces wordt door Martin en Odell opgedeeld in een zestal activiteiten. Tijdens deze activiteiten dient de nadruk te liggen op het objectgedrag. Tijdens het analyseren van het objectgedrag krijgt de objectstructuur gelijktijdig vorm. De zes activiteiten zijn:

0. het definiëren van de focus van de analyse
1. het verduidelijken van de event types
2. het generaliseren van de event types
3. het definiëren van de operatiecondities
4. het identificeren van de operatiegevallen
5. het verfijnen van de resultaten van stap 1 tot en met 4

Vervolgens moet verder worden gegaan met stap 1, totdat de vervaardigde modellen in voldoende detail zijn uitgewerkt. Deze incrementele manier van ontwikkelen wordt door Martin en Odell aangeduid met de termen *goal-directed* en *behavior-driven*.

De methode van Martin en Odell heeft een aantal opvallende kenmerken, waaraan zowel voor- als nadelen kleven. De methode maakt gebruik van bekende notaties, waardoor ontwikkelaars geen geheel nieuwe notaties hoeven te leren en de inwerktijd kort kan zijn. Het risico, dat de ontwikkelaars met de nieuwe methode toch op de oude manier blijven werken, wordt echter door het gebruik van bekende notaties vergroot.

Het toestaan van meervoudige classificatie heeft als voordeel, dat tijdens de analysefase nog niet gekeken hoeft te worden naar kunstmatige subklassen, die moeten worden gecreëerd om voor object instanties, die eigenlijk van meer objectklassen instantie zijn, toch een unieke objectklasse te hebben. Nadeel is, dat tijdens de ontwerp- en met name de implementatiefase omwegen nodig zijn om het analysemodel implementeerbaar te maken, omdat meervoudige classificatie op dit moment nog niet wordt ondersteund door programmeertalen.

Pluspunt van met name [MO92] is de grondige manier, waarop de gepresenteerde technieken worden uitgewerkt. De technieken worden gedetailleerd beschreven en geïllustreerd aan de hand van een groot aantal voorbeelden. Ook bij deze methode ontbreekt een formele basis.

4.2.6 OMT van Rumbaugh et al.

De *Object Modeling Technique* (OMT) van James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy en William Lorensen is ontwikkeld door General Electric en wordt beschreven in [RBP$^+$91]. OMT beslaat volgens de auteurs het gehele ontwikkelproces van probleemformulatie via analyse, ontwerp, implementatie en testen tot onderhoud. In [RBP$^+$91] worden echter alleen analyse, systeemontwerp en objectontwerp beschreven. OMT is een combinatieve methode, die gebruik maakt van een aantal traditionele modelleringstechnieken.

Tijdens de analysefase worden drie activiteiten onderscheiden, te weten:

- objectmodellering, resulterend in een *object diagram*
- dynamische modellering, resulterend in *structured state diagrams* voor alle actieve objecten
- functionele modellering, resulterend in *data flow diagrams*

Hoewel de hierboven gegeven volgorde de aanbevolen volgorde is, plaatsen de auteurs de opmerking, dat van die volgorde kan worden afgeweken en dat iteratie zelfs vrijwel onvermijdelijk is.

Tijdens de systeemontwerpfase wordt de systeemarchitectuur vastgelegd. Rumbaugh et al. beperken zich tot het beschrijven van de te specificeren kenmerken. Over te gebruiken notaties en technieken wordt nauwelijks gesproken.

De objectontwerpfase gaat verder, waar de analysefase was blijven steken. De tijdens de analysefase vervaardigde modellen worden uitgebreid met implementatiedetails. Tijdens de objectontwerpfase worden dezelfde technieken en notaties gebruikt als tijdens de analysefase.

Hoewel OMT op zich degelijk in elkaar steekt en de verschillende modelleringstechnieken gedetailleerd worden beschreven, lijkt het geheel losser in elkaar te zitten dan bijvoorbeeld de methode van Coad en Yourdon. De samenhang tussen objectmodellen, dynamische modellen en functionele modellen wordt vrij informeel beschreven. Hetzelfde geldt voor de samenhang tussen analyse en objectontwerp enerzijds en systeemontwerp anderzijds. Hierdoor is de kans op inconsistente modellen groter dan bij bijvoorbeeld de methode van Coad en Yourdon. Wel worden in [RBP+91] drie vrij gedetailleerd uitgewerkte voorbeelden gegeven.

Tools, die OMT ondersteunen, zijn verkrijgbaar via onder andere GE, Love, SoftwareThroughPictures, Cadre en Protosoft [PRO94]. Een partiële formalisatie van het in [RBP+91] gepresenteerde object diagram is te vinden in [BRO93].

4.2.7 De methode van Shlaer en Mellor

De methode van Sally Shlaer en Stephen J. Mellor werd voor het eerst gepresenteerd in [SM88]. De methode, die daarin wordt beschreven, is echter niet veel meer dan een enigszins uitgebreide vorm van Entity-Relationship modellering. Aan procesmodellering en functiemodellering wordt nauwelijks aandacht besteed. Het vier jaar later verschenen [SM92] gaat dieper in op die delen van het analyseproces, die in eerste instantie wat ondergeschoven waren, zoals procesmodellering en functiemodellering. In [SM92] wordt daarnaast, zij het minder uitgebreid, een methode beschreven om de tijdens de analysefase geproduceerde modellen om te zetten in een ontwerp en wordt een ontwerpmethode gepresenteerd.

Shlaer en Mellor onderscheiden in de analysefase drie activiteiten:

- information modelling, met als eindprodukten een *information model*, *object and attribute descriptions* voor alle objecten in het information model en *relationship descriptions* voor alle relaties in het information model
- state modelling, met als eindprodukten *state models* voor alle objecten en relaties in het information model, die interessant dynamisch gedrag vertonen, en bijbehorende *event lists* en een *object communication model*, dat het dynamische gedrag van het systeem als geheel beschrijft
- process modelling, met als eindprodukten *action data flow diagrams* (ADFDs) voor alle acties in de state models, *process descriptions* voor de in de ADFDs voorkomende processen, *state process tables* en een *object access model*

Daarnaast bieden Shlaer en Mellor een methode om deelsystemen te onderscheiden. Eindprodukten daarvan zijn *domain charts*, een *project matrix*, een *subsystem relationship model*, een *subsystem communication model* en een *subsystem access model*. Op basis van de zo ontstane indeling kunnen de verschillende subsystemen afzonderlijk worden geanalyseerd.

Voor de ontwerpfase maken Shlaer en Mellor gebruik van een door hen ontwikkelde notatie, OODLE (Object-Oriented Design LanguagE) genaamd. De notatie omvat vier verschillende diagrammen, te weten:

class diagrams, class structure charts, dependency diagrams en *inheritance diagrams*. Bij hun ontwerpmethode gaan Shlaer en Mellor uit van de volgende architectuur:

- een hoofdprogramma
- vier architecturele klassen, te weten de *transition class*, de *finite state model class*, de *active instance class* en de *timer class*
- een aantal applicatieklassen, die overeenkomen met de in het information model onderscheiden objecten

De eerste drie architecturele klassen zorgen voor mechanismen, die het initialiseren en doorlopen van state machines mogelijk maken, terwijl de timer class de tijdsaspecten verzorgt.

Shlaer en Mellor beschrijven vervolgens kort de manier, waarop voor deze klassen class diagrams kunnen worden ontworpen of uit het information model kunnen worden gegenereerd. Als dat nodig mocht zijn, dan moeten ook dependency diagrams en inheritance diagrams worden ontworpen of gegenereerd. Vervolgens worden op basis van de class diagrams en de ADFDs uit de analysefase class structure charts ontworpen.

De analysemethode van Shlaer en Mellor lijkt goed in elkaar te zitten. Informatiemodellering, procesmodellering en functiemodellering sluiten goed op elkaar aan. Bovendien bieden Shlaer en Mellor veel praktische handvaten, die tijdens de analyse kunnen worden gebruikt. De ontwerpmethode is daarentegen vrij summier: een vrij lastig proces wordt in één hoofdstuk en een bijbehorende appendix beschreven. Hoewel in de boeken tal van voorbeelden worden gepresenteerd, ontbreken echte grote, het gehele ontwikkelproces beslaande voorbeelden.

Tools, die de methode van Shlaer en Mellor ondersteunen, zijn via ProjectTechnology, Cadre en IPSYS leverbaar [PRO94]. Een beschrijving en een evaluatie van het gebruik van de methode van Shlaer en Mellor in een praktijkgeval zijn te vinden in [FHR+93].

4.2.8 De methode van Wirfs-Brock et al.

Rebecca Wirfs-Brock, Brian Wilkerson en Lauren Wiener presenteren in [WWW90] een methode, die vaak wordt aangeduid als de methode van Wirfs-Brock et al. en die in deze studie ook door ons zo zal worden genoemd. Evenals de methode van Booch en de OOSE methode van Jacobson et al. is deze methode een pure methode, die geen gebruik maakt van bestaande technieken en notaties. De methode vertoont nog een andere overeenkomst met de methode van Booch: hoewel de titel van het boek ("Designing Object-Oriented Software") suggereert, dat alleen het ontwerpproces wordt beschreven, behandelen Wirfs-Brock et al. wel degelijk de analysefase. Ook Wirfs-Brock et al. verstaan namelijk onder de term analyseproces alleen het vastleggen van de specificaties en niet het modelleren op basis van die specificaties. Vergeleken met andere methoden is de methode van Wirfs-Brock et al. juist vooral een analysemethode. Aan het eigenlijke ontwerpproces wordt slechts één hoofdstuk besteed.

Wirfs-Brock et al. onderscheiden in het ontwerpproces de volgende activiteiten:

- het vinden van klassen
- het bepalen van verantwoordelijkheden ('responsibilities') van klassen
- het bepalen van samenwerking ('collaboration') tussen klassen
- het bepalen van klassehiërarchieën
- het definiëren van contracten voor de klassen

- het samenbrengen van klassen in subsystemen en het definiëren van contracten voor die subsystemen
- het beschrijven van protocollen voor iedere klasse
- het specificeren van klassen, subsystemen en contracten

Doordat subsystemen contracten kunnen hebben en zo naar de omgeving toe als één geheel kunnen optreden, gaat het mechanisme veel verder dan bijvoorbeeld het subjectmechanisme van Coad en Yourdon of het subsysteemmechanisme van Shlaer en Mellor, waar respectievelijk subjects en subsystemen slechts op het hoogste niveau kunnen worden gebruikt. Met behulp van het subsysteemmechanisme van Wirfs-Brock et al. is het mogelijk om een ontwerp op een zelf te bepalen aantal abstractieniveaus uit te werken.

Tijdens de eerste vier activiteiten gebruiken Wirfs-Brock et al. de in [BC89] geïntroduceerde *CRC kaarten* (Class, Responsibility, Collaboration). Op basis van deze kaarten worden tijdens het ontwerpproces de volgende eindprodukten vervaardigd:

- *hierarchy graphs* voor alle klassehiërarchieën
- *collaboration graphs* voor het gehele systeem en alle subsystemen
- *class specifications* voor alle klassen
- *subsystem specifications* voor alle subsystemen
- *contract specifications* voor alle contracten

Het uiteindelijke ontwerp voor een met behulp van de methode van Wirfs-Brock et al. ontwikkeld systeem ziet er heel anders uit dan bijvoorbeeld een met behulp van de OOA/OOD van Coad en Yourdon of de OMT methode van Rumbaugh et al. vervaardigd ontwerp. Dit is deels het gevolg van de afwijkende notatie en deels het gevolg van de antropomorfe benadering van systeemontwikkeling. Rumbaugh et al. bijvoorbeeld spreken met hun combinatieve aanpak over attributen, relaties en dergelijke, terwijl Wirfs-Brock et al. spreken over verantwoordelijkheden, samenwerking en contracten. Een dergelijke beschouwing van systeemontwikkeling wordt door enkele auteurs, waaronder [TAY92], *responsibility-driven design* genoemd. Hoewel de ideeën achter een dergelijke beschouwing het bekijken waard zijn en mogelijk tot betere systemen kunnen leiden, lijkt de methode van Wirfs-Brock et al. nog niet vervolmaakt en dient nog een aantal zaken toegevoegd te worden, voordat de methode 'volwassen' kan worden genoemd. Hierbij kan worden gedacht aan de manier, waarop protocollen worden ontworpen en vervolgens gespecificeerd.

[WWW90] besluit met een drietal gedetailleerd uitgewerkte voorbeelden, die meer dan een derde van de totale tekst beslaan en die een goed beeld geven van de gepresenteerde methode.

4.3 Programmeertalen

De eerste OO programmeertaal was, zoals in sectie 1.2 (De geschiedenis van Object-Oriëntatie) reeds naar voren is gekomen, Simula 67. In de jaren zeventig kwam daar Smalltalk bij, in de jaren tachtig gevolgd door onder andere Object Pascal, Objective-C, CLOS, C++ en Eiffel en Object-georiënteerde versies van bijvoorbeeld Turbo Pascal. Een Object-georiënteerde versie van COBOL, Object COBOL genaamd, was in 1992 nog in ontwikkeling [TAY92][4]. [SAU89] komt met een lijst van in totaal 88 Object-georiënteerde

[4]Tijdens één van de door ons in april 1994 binnen Rabobank Nederland afgenomen interviews werd opgemerkt dat de afdeling, waarvan de geïnterviewde persoon hoofd is, op korte termijn de beschikking zou krijgen over een β-versie van Object COBOL.

programmeertalen en in [BOO91] is een lijst met 112 Object-georiënteerde en Object-based programmeertalen opgenomen.

In deze sectie wordt eerst een indeling van Object-georiënteerde programmeertalen in twee verschillende groepen gegeven. Vervolgens worden enkele eigenschappen van Object-georiënteerde programmeertalen beschreven. Tenslotte wordt een aantal talen in meer detail besproken. Ook de talen worden in alfabetische volgorde gepresenteerd.

4.3.1 Indeling van programmeertalen

De meeste auteurs onderscheiden twee groepen Object-georiënteerde programmeertalen, te weten:

- pure OO talen
- hybride OO talen

Pure OO talen zijn programmeertalen, die speciaal voor OOP zijn ontwikkeld. Het is niet mogelijk om in dergelijke talen niet-Object-georiënteerde software te schrijven. Voorbeelden van dergelijke talen zijn Smalltalk, Actor en Eiffel.

Hybride OO talen zijn programmeertalen, die zijn gebouwd op bestaande talen en die alle in de originele taal bestaande concepten in zich hebben. Het is daardoor mogelijk om in een hybride OO taal niet-Object-georiënteerde programma's te schrijven. Voorbeelden van hybride OO talen zijn Objective-C en C++ (gebaseerd op C), Object Pascal, Quick Pascal en Turbo Pascal (gebaseerd op Pascal) en CLOS (gebaseerd op Lisp).

Zoals in paragraaf 2.4.4 (Hybride methoden en talen) is beschreven, brengen hybride programmeertalen het gevaar van slordig gebruik met zich mee. Daar staat tegenover, dat programmeurs vaak bekend zijn met een aantal belangrijke concepten van de conventionele taal en dus bij gebruik van hybride talen minder training nodig zullen hebben.

Tenslotte onderscheidt een aantal auteurs *Object-based* programmeertalen, die wel de inkapseling van operaties en attributen in een object toestaan, maar die geen mechanismen bieden, die generalisatie en het gebruik van klassen ondersteunen. Een voorbeeld van een dergelijke taal is Ada.

4.3.2 Eigenschappen van programmeertalen

Naast het onderscheid, dat kan worden gemaakt tussen hybride en pure programmeertalen, kan ook onderscheid worden gemaakt op basis van een aantal andere eigenschappen van programmeertalen. Deze eigenschappen zijn:

- statische of dynamische binding
- zwakke of sterke typering
- enkelvoudige of veelvuldige generalisatie
- interpretatie of compilatie

Binding betreft het moment, waarop een operatie wordt gekoppeld aan een methode. Bij *statische binding*, volgens [TAY92] soms ook *vroege binding* of *compile-time binding* genoemd, wordt een operatie tijdens de compilatie gekoppeld aan een methode. Bij *dynamische binding*, ook wel *late binding* of *run-time binding* genoemd, gebeurt dit pas tijdens de executie. De combinatie van polymorfisme en dynamische bin-

ding maakt het mogelijk implementaties van objecten te veranderen, zonder de noodzaak van hernieuwde compilatie [COC93].

Typering betreft de declaratie van het soort informatie, dat variabelen mogen bevatten. *Sterke typering* vereist, dat van alle variabelen het type is gedeclareerd, voordat zij kunnen worden gebruikt. *Zwakke typering* staat toe, dat een variabele het type aanneemt, dat op dat moment het meest geschikt is. Programmeertalen met zwakke typering worden daarom ook wel *untyped* genoemd [TAY92].

Het verschil tussen enkelvoudige en veelvuldige generalisatie is in paragraaf 1.4.7 (Generalisatie) aan bod gekomen. Een aantal programmeertalen staat alleen enkelvoudige generalisatie toe.

Een *interpreter* vertaalt een programma instructie voor instructie naar compacte, interne bytecode, waarna tijdens executie van het programma een run-time interpreter die code omzet in machinecode. Een *compiler* daarentegen vertaalt een programma direct naar executeerbare code. Deze compilers worden ook wel *conventionele compilers* genoemd, in tegenstelling tot de *dynamische compilers*, die zijn verschenen na de introductie van OOP. Dynamische compilers vertalen net als interpreters instructies in eerste instantie naar interne bytecode, maar tijdens de executie worden om de snelheid te verhogen hele methoden in een keer omgezet naar machinecode. Tenslotte kunnen programmeertalen worden vertaald met behulp van *preprocessors*, die programma's omzetten naar een conventionele programmeertaal, waarna vervolgens een bij die taal behorende compiler kan worden gebruikt [TAY92].

4.3.3 C++

C++ is een hybride programmeertaal, die is gebaseerd op de programmeertaal C. Het relatief grote succes van C++ is met name te danken aan die verwantschap met C. C++ werd ontwikkeld door Bjarne Stroustrup bij AT&T's Bell Labs. De meeste C++ versies maken gebruik van een preprocessor en een C compiler of een C++ compiler. [TAY92] maakt echter melding van het verschijnen van de eerste C++ interpreters en C++ dynamische compilers.

C++ wordt gekenmerkt door sterke typering en ondersteuning van veelvuldige generalisatie. Dynamische binding is slechts mogelijk voor klassen, die een superklasse gemeen hebben.

De meeste C++ versies kennen geen uitgebreide programmeeromgeving. Uitzonderingen hierop vormen ObjectWorks\C++ van ParcPlace, dat is gebaseerd op ObjectWorks\Smalltalk van dezelfde onderneming (zie paragraaf 4.3.6, Smalltalk), en Borland C++ van Borland International.

Voor meer informatie over C++ verwijzen wij naar bijvoorbeeld [ES90] of [STR91].

4.3.4 CLOS

CLOS, wat staat voor *Common Lisp Object System*, is voortgekomen uit een aantal al dan niet Object-georiënteerde Lisp dialecten als Common Lisp, Common LOOPS en New Flavors. CLOS wordt vooral veel gebruikt voor de ontwikkeling van systemen op het gebied van kunstmatige intelligentie (artificial intelligence, AI) zoals expertsystemen.

CLOS wordt gekenmerkt door ondersteuning van veelvuldige generalisatie en metaklassen en door zwakke typering, hoewel sterke typering optioneel is. CLOS wordt vertaald met behulp van een interpreter, hoewel sommige versies van CLOS ook een compiler kennen. De meeste versies van CLOS worden ondersteund door een programmeeromgeving.

Meer informatie over CLOS is te vinden in onder andere [KG89].

4.3.5 Eiffel

Eiffel is een sterk door Simula beïnvloede, pure Object-georiënteerde programmeertaal. Eiffel werd ontwikkeld door Bertrand Meyer. De evolutie van de taal wordt gecontroleerd door het Nonprofit International Consortium for Eiffel (NICE) [MEY92]. [RBP+91] noemt Eiffel de beste commerciële Object-georiënteerde programmeertaal in termen van technische capaciteiten. Eiffel wordt gekenmerkt door dynamische binding, statische type checking, ondersteuning van veelvuldige generalisatie en geparametriseerde klassen. Een unieke eigenschap van Eiffel is de mogelijkheid tot het aangeven van *asserties* als pre- en postcondities en invarianten. Schending van deze condities of invarianten leidt tot executie van een exceptie.

[TAY92] wijt het uitblijven van commercieel succes van Eiffel aan het feit, dat er nog maar één versie van Eiffel op de markt verkrijgbaar is.

Een uitgebreidere beschrijving van Eiffel is te vinden in [MEY88] en [MEY91].

4.3.6 Smalltalk

Smalltalk staat bekend als de eerste pure Object-georiënteerde programmeertaal. Smalltalk is ontwikkeld door een aantal onderzoekers van de Xerox Corporation onder leiding van Alan Kay en Adele Goldberg. Smalltalk is gebouwd op twee simpele concepten:

1. alles wordt behandeld als een object
2. objecten communiceren met elkaar door middel van boodschappen

De oorspronkelijke versie van Smalltalk, Smalltalk-72, is inmiddels via Smalltalk-74, Smalltalk-76 en Smalltalk-78 geëvolueerd tot Smalltalk-80. [TAY92] vermeld twee commerciële Smalltalk produkten: ObjectWorks\Smalltalk van ParcPlace en Smalltalk/V van Digitalk. ObjectWorks\Smalltalk maakt gebruik van een dynamische compiler. Smalltalk/V daarentegen maakt gebruik van een interpreter, terwijl in de OS/2 versie ook een conventionele compiler is opgenomen. Beide produkten zijn uitgebreide programmeeromgevingen met class browsers, debuggers en dergelijke.

Smalltalk maakt gebruik van zwakke typering en dynamische binding. Smalltalk kent geen veelvuldige generalisatie. Wel kunnen in Smalltalk metaklassen worden gedefinieerd, waarin informatie over bepaalde klassen kan worden opgeslagen, zoals het aantal object instanties.

Voor meer informatie over Smalltalk verwijzen wij naar de over Smalltalk verschenen literatuur, bijvoorbeeld [GR89] of het werk van LaLonde en Pugh ([LP90], [LP91] en [LP93]).

4.4 Database Management Systemen

Naast Object-georiënteerde programmeertalen is inmiddels ook een aantal *Object-georiënteerde database management systemen* op de markt verschenen. In het vervolg van deze sectie gebruiken wij de afkorting *OODBMS*, hoewel deze afkorting niet door iedereen wordt gebruikt. In [RBP+91] bijvoorbeeld wordt gesproken van OO-DBMS, terwijl [TAY92] de afkorting ODBMS gebruikt.

In deze sectie wordt eerst aangegeven, waarom er naast relationele DBMSs ook behoefte is aan OODBMSs. Vervolgens wordt een indeling van deze OODBMSs gegeven. Daarna wordt aandacht besteed aan interfaces naar programmeertalen. Tenslotte wordt een aantal op de markt aangeboden OODBMSs besproken.

4.4.1 Relationele versus Object-georiënteerde DBMSs

[TAY92] geeft een aantal redenen voor het verschijnen van OODBMSs.

- Relationele DBMSs kampten met nadelen, zoals structurele en fysieke beperkingen. De toepassing van OOP leidde tot nieuwe ideeën over de wijze, waarop die nadelen kunnen worden ondervangen.
- Door de toepassing van OOP ontstond behoefte aan systemen, waarin objecten kunnen worden opgeslagen, die blijven bestaan na terminatie van een programma; de zogenaamde *persistent objects*. Relationele databases waren daarvoor minder geschikt, omdat zij geen mogelijkheden boden voor het bij elkaar opslaan van de gegevens en de functies van een object.
- Door het streven naar herbruikbare software ontstond behoefte aan als *repository* te gebruiken DBMSs, waarin herbruikbare klassen en andere softwarecomponenten kunnen worden opgeslagen. Deze repository zou na voltooiing van het project moeten kunnen worden gebruikt als klassenbibliotheek. Ook in dit geval gold, dat relationele DBMSs geen mogelijkheden boden om gegevens en functies gezamenlijk op te slaan.

Hoewel relationele databases beter geschikt zijn voor simpele structuren en eenvoudige transacties en queries [SZ92] en over betere faciliteiten voor het werken met relaties en sleutels beschikken, blijken OODBMSs veel efficiënter om te gaan met complexe informatie. [TAY92] geeft als voorbeeld een test bij de US Navy, waarin een uit 25.000 componenten bestaande printplaat met behulp van een CAD systeem moest worden getoond en opgeslagen. Terwijl het relationele DBMS daar een kwartier voor nodig had, deed het OODBMS er slechts negen seconden over, een factor 100 verschil. [TAY92] maakt ook melding van een studie bij Sun Microsystems, waaruit blijkt, dat alle geteste OODBMSs in de door hen uitgevoerde tests betere prestaties leveren dan een aantal veel gebruikte relationele DBMSs.

4.4.2 Indeling van OODBMSs

Ook voor databases geldt het onderscheid tussen pure en hybride OODBMSs. Een puur OODBMS is alleen geschikt voor opslag en bewerking van objecten. Een hybride OODBMS daarentegen is een uitbreiding van een bestaand DBMS met Object-georiënteerde aspecten. Het voordeel van een hybride OODBMS is de betere ondersteuning door tools en dergelijke. Nadelen zijn de nog steeds geldende fysieke beperkingen van relationele databases en de kunstmatige en daardoor minder efficiënte manier van omgaan met objecten, operaties en klassen.

4.4.3 Interfaces naar programmeertalen

Hoewel de meeste OODBMSs een eigen *data description language* (DDL) en een eigen *data manipulation language* (DML) hebben, hebben zij meestal ook één of meerdere interfaces naar andere programmeertalen. De meeste ontwikkelaars hebben hun OODBMS zelfs voorzien van een interface voor *OSQL*, een (overigens niet gestandaardiseerde) Object-georiënteerde versie van SQL. Er worden pogingen ondernomen om tot een gestandaardiseerde versie van OSQL te komen [TAY92].

4.4.4 Op de markt aangeboden OODBMSs

[TAY92] maakt melding van een zestal op de markt te koop aangeboden OODBMSs.

- *Gem Stone* van Servio Corporation. Gem Stone heeft een eigen DDL/DML, *Opal*, en interfaces voor C, C++ en Smalltalk.

- *Itasca* van Itasca Systems. Itasca is gebouwd ter ondersteuning van Lisp, maar heeft ook een interface voor C.
- *Objectivity/DB* van Objectivity. Objectivity/DB heeft eveneens een interface voor C++ en is vooral geschikt voor computer-aided design (CAD).
- *ObjectStore* van ObjectDesign. ObjectStore heeft interfaces voor C++ en C.
- *Ontos* van Ontos Corporation. Ontos heeft interfaces voor C++, C en SQL.
- *Versant ODBMS* van Versant Object Technology. Versant heeft interfaces voor C++, C en Smalltalk-80 en wordt ondersteund door een aantal tools, waaronder een versie van OSQL.

Voor een uitgebreidere omschrijving van de verschillende DBMSs en voor meer informatie over Object-georiënteerde databases en database management systemen verwijzen wij naar [TAY92].

4.5 Hulpmiddelen

Tools en dan met name Computer Assisted Software Engineering (*CASE*) tools zijn een gewild hulpmiddel geworden bij systeemontwikkeling. Object-georiënteerde systeemontwikkeling leent zich voor ondersteuning met behulp van CASE tools. Het aantal bruikbare CASE tools is echter nog beperkt. In [CF92] wordt dit geweten aan een soort 'kip en ei' situatie: verkopers van CASE tools aarzelen met het op de markt brengen van Object-georiënteerde tools vanwege de geringe vraag en de onduidelijkheid over gebruikte methoden en technieken, terwijl gebruikers aarzelen om over te stappen op OO vanwege hun onbekendheid met methoden en technieken en het ontbreken van geschikte tools om er bekend mee te raken.

In deze sectie wordt eerst een aantal mogelijke indelingen van tools gegeven. Daarna wordt een aantal verschillende soorten tools behandeld. Het laatste deel is gewijd aan op de markt beschikbare tools.

4.5.1 Indeling van tools

Zoals bij methoden, programmeertalen en databases het geval is, kan ook bij tools onderscheid worden gemaakt tussen pure OO tools, hybride OO tools en traditionele tools, die OO ondersteunen [MAR92]. *Pure OO tools* ondersteunen alleen Object-georiënteerde systeemontwikkeling, terwijl *hybride OO tools* zowel Object-georiënteerde als traditionele systeemontwikkeling ondersteunen. De laatste categorie tools dient ter ondersteuning van traditionele systeemontwikkeling, maar is uitgebreid met een aantal Object-georiënteerde kenmerken, zoals generalisatie. Deze tools bieden in tegenstelling tot pure en hybride tools geen mogelijkheden om gebruik te maken van Object-georiënteerde methoden en technieken.

Daarnaast kunnen nog andere indelingen worden gemaakt, zoals een indeling in *I-CASE tools* en *gefragmenteerde CASE tools*. Deze laatste categorie tools ondersteunt slechts een gedeelte van de totale systeemontwikkeling, bijvoorbeeld de analysefase of automatische codegeneratie. I-CASE tools daarentegen ondersteunen het gehele ontwikkelproces en maken het mogelijk om op basis van een analyse een ontwerp te maken en op basis van dat ontwerp code te genereren. Object-georiënteerde systeemontwikkeling, die wordt gekarakteriseerd door het in elkaar overvloeien van de analyse-, de ontwerp- en de implementatiefase, leent zich daarom bijzonder goed voor ondersteuning door middel van I-CASE tools.

4.5.2 Soorten hulpmiddelen

[TAY92] noemt een aantal soorten hulpmiddelen, die Object-georiënteerde systeemontwikkeling kunnen ondersteunen, te weten:

- editors
- class browsers
- debuggers
- OO-CASE tools
- Object-georiënteerde vierde generatie talen
- code management tools

Editors en *debuggers* worden al geruime tijd gebruikt bij systeemontwikkeling. Vooral in een programmeeromgeving geïntegreerde editors en debuggers kunnen het ontwikkelproces vergemakkelijken.

Class browsers maken het mogelijk klassestructuren te bekijken en van geselecteerde klassen interfaces en implementaties te bekijken en aan te passen. De Smalltalk omgeving was de eerste omgeving, die met een dergelijk hulpmiddel uitgerust was. Tegenwoordig hebben de meeste Object-georiënteerde programmeeromgevingen een class browser.

Ook *CASE tools* en *vierde generatie talen* (fourth generation languages, 4GLs) worden al langer gebruikt om systeemontwikkeling te ondersteunen. Daar de huidige CASE tools en vierde generatie talen geen Object-Oriëntatie ondersteunen, bestaat er behoefte aan OO-CASE tools en Object-georiënteerde 4GLs, die zijn gebaseerd op Object-georiënteerde methoden en technieken en Object-georiënteerde programmeertalen. Het aantal tot nu toe op de markt verschenen OO-CASE tools en Object-georiënteerde 4GLs is, zoals in de inleiding van deze sectie reeds is opgemerkt, nog vrij beperkt.

Code management systemen tenslotte ondersteunen het gebruik van klassenbibliotheken en al ontwikkelde code door meerdere mensen tegelijkertijd. In projecten, waarin een groot aantal teams van ontwikkelaars tegelijkertijd aan het werk is, zijn zij onmisbaar. Vooral voor Smalltalk is reeds een aantal code management systemen op de markt verschenen, zoals de Application Manager van Coopers & Lybrand, de ENVY/Developer van Object Technology International en de Convergence/Team van Instantiations Inc. [TAY92].

4.5.3 Op de markt aangeboden tools

Hoewel een aantal auteurs spreekt over een beperkt aantal tools, doet een blik op een tijdschrift als de Journal of Object-Oriented Programming vermoeden, dat die situatie snel zal veranderen. In de JOOP van januari 1994 worden tal van tools en workbenches aangeboden. De meeste van deze tools ondersteunen slechts één methode. Booch, Shlaer/Mellor, Rumbaugh et al., Coad/Yourdon en Martin/Odell werden met name genoemd. Wat programmeertalen betreft genereren de meeste tools C++ en/of Smalltalk code. Als door tools ondersteunde OODBMSs worden onder andere ObjectStore, Objectivity en Versant genoemd.

4.6 Standaardisatie

In sectie 1.2 (De geschiedenis van Object-Oriëntatie) is reeds opgemerkt, dat inmiddels een standaardisatie organisatie is opgericht, de *Object Management Group* (OMG). Bij deze organisatie zijn meer dan tweehonderd computer- en softwarebedrijven aangesloten, waaronder een aantal marktleiders, zoals Apple, AT&T/NCR, Borland, DEC, Hewlett-Packard, IBM, Intel, Microsoft, Motorola, Nixdorf, Philips, Sun, Texas Instruments, Unisys, Wang en Xerox. Ook niet direct voor de hand liggende concerns als Boeing, Citibank en Volvo zijn aangesloten bij de OMG [TAY92].

In deze sectie worden eerst de doelstellingen van de OMG beschreven. Vervolgens wordt de door de OMG voorgestelde referentiearchitectuur behandeld.

4.6.1 Doelstellingen van de OMG

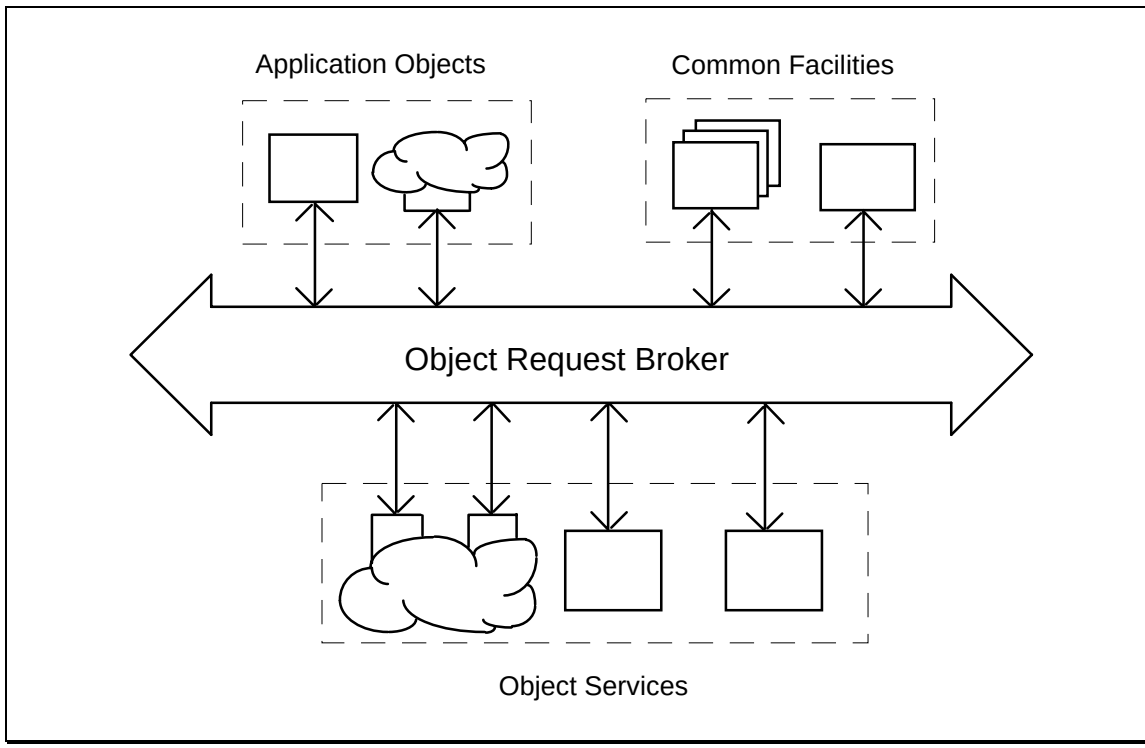
De OMG heeft de volgende doelstellingen [MAR92]:

- het maximaliseren van de portabiliteit, herbruikbaarheid en uitwisselbaarheid van software
- het voorzien in een referentiearchitectuur met termen en definities, waarop alle specificaties zijn gebaseerd
- het voorzien in een open forum voor industriële discussie over, onderricht in en promotie van door de OMG onderschreven Object-georiënteerde technologie

De OMG streeft naar de creatie van industriële standaarden voor commercieel beschikbare Object-georiënteerde systemen en richt zich daarbij op remote object network access, inkapseling van bestaande applicaties en object database interfaces.

4.6.2 CORBA

Inmiddels is door de OMG een referentiearchitectuur opgesteld: de *common object request broker archi-*



Figuur 4.2: De CORBA architectuur, gebaseerd op Figure 21.1 in [MAR92]. De rechthoeken stellen klassen voor, de wolken niet-OO software en de rechthoekjes op de wolken OO interfaces op niet-OO applicaties.

tecture (CORBA) [BEA93]. In figuur 4.2 is de CORBA architectuur weergegeven.

De architectuur is opgebouwd uit vier hoofdcomponenten.

- De *object request broker* (ORB) maakt het mogelijk, dat objecten boodschappen en antwoorden kunnen verzenden en ontvangen.
- De *object services* (OS) vormen een verzameling diensten met OO interfaces, die voorzien in standaardfuncties om objecten te realiseren en te onderhouden. Object Services zijn bijvoorbeeld operating systems, database management, opslag, klassemanagement, versiecontrole, integriteitsbewaking, beveiliging en opvraagtalen. Object services hoeven niet noodzakelijkerwijs Object-georiënteerde applicaties te zijn, zolang zij maar zijn voorzien van een OO interface, zodat de ORB hen als objecten kan beschouwen.
- De *common facilities* (CF) vormen een verzameling applicatie objecten, die standaardvoorzieningen bieden. Common facilities zijn bijvoorbeeld printen, mail, link management, tekst editors en hulpfuncties.
- De *application objects* (AO) zijn specifieke applicaties van de gebruiker. Ook zij hoeven niet Object-georiënteerd te zijn, zolang zij maar zijn voorzien van een OO interface.

In deze architectuur is de ORB de centrale component. ORBs kunnen met andere ORBs worden verbonden, zodat een netwerk ontstaat, waarin application objects via hun ORB gebruik kunnen maken van common facilities of object services, die aan andere ORBs zijn gekoppeld. De componenten communiceren met elkaar in Interface Definition Language (IDL), een deelverzameling van ANSI C++, waaraan een aantal ondersteunende mechanismen is toegevoegd [MAR92].

4.7 Met OO behaalde resultaten

Over met OO behaalde resultaten is in de literatuur weinig te vinden. Daarnaast zijn de projecten, die worden beschreven, meestal geslaagde projecten. De enige auteur, die daadwerkelijk aandacht besteed aan het gebruik van OO in commerciële projecten, is Taylor, die in [TAY92] achttien projecten beschrijft, variërend van geografische informatiesystemen tot produktiesystemen voor bijvoorbeeld halfgeleiders. Ook financiële applicaties komen aan bod: Taylor beschrijft de ontwikkeling van een portfolio management systeem, dat door Wyatt Software werd ontwikkeld voor een aantal banken, een makelarijsysteem voor Hambrecht & Quist en een 'technical trading environment' voor Midland Bank.

Een uitgebreidere beschrijving van het door Taylor genoemde 'customer management system', dat door Brooklyn Gas Union in samenwerking met Andersen Consulting werd ontwikkeld, is te vinden in [DM93].

Op basis van de door hem besproken projecten trekt Taylor de volgende conclusies:

- OO maakt zijn beloftes waar
- OO vereist veranderingen
- OO is rijp voor massale acceptatie

De door Taylor besproken projecten geven inderdaad de indruk, dat deze conclusies gerechtvaardigd zijn. Feit blijft, dat nergens voorbeelden van gefaalde projecten te vinden zijn en dat het gevaarlijk en zelfs onverstandig is om conclusies te trekken op basis van eenzijdige informatie.

In [CCH+94] worden drie Object-georiënteerde projecten bij IBM beschreven en geëvalueerd. De drie projecten vertonen grote verschillen in applicatiedomein, omvang, platform en gebruikte programmeertaal.

De enige overeenkomst is het gebruik van Object-georiënteerde technologie. Op basis van de evaluatie werd een aantal al dan niet verwachte conclusies getrokken.

- De kwaliteit van de geproduceerde code is uitstekend. Dit is met name het gevolg van het hergebruik door middel van overerving en van de inkapseling van gegevens. Desondanks merken de onderzoekers op, dat de vaardigheden en ervaring van de ontwikkelaars een grotere invloed hebben op de kwaliteit van de code dan welke technologie ook.
- Er is sprake van een duidelijke afname van het aantal door de eindgebruikers gewenste veranderingen, zowel tijdens de ontwikkeling als na de oplevering. Deze afname is het gevolg van de grotere betrokkenheid van de eindgebruikers bij het ontwikkelproces.
- De gebruiksvriendelijkheid van een systeem wordt verhoogd, met name door het gebruik van Object-georiënteerde gebruikersinterfaces. Het betreft hier zowel het gemak, waarmee een gebruiker leert omgaan met het systeem, als het gebruiksgemak en de interne consistentie.
- De relatieve kosten om een systeem aan te passen aan veranderde functionele vereisten, het zogenaamde adaptieve onderhoud, en aan niet-functionele vereisten, het zogenaamde perfectieve onderhoud, zijn lager bij Object-georiënteerde systemen dan bij vergelijkbare conventionele systemen. Dit is het gevolg van de hogere modulariteit van Object-georiënteerde systemen.
- Het kost relatief minder moeite om in een Object-georiënteerd systeem fouten te detecteren en te verbeteren, het zogenaamde correctieve onderhoud. Ook dit is het gevolg van de hogere modulariteit.
- De prestaties van een Object-georiënteerd systeem hoeven niet onder te doen voor de prestaties van een vergelijkbaar conventioneel systeem, maar er is wel meer inspanning nodig om die prestaties te bereiken. Een positief punt is in dit geval het vroege tijdstip, waarop problemen met de prestaties van een systeem worden gesignaleerd, vanwege het gebruik van prototyping.

Daarnaast is enige literatuur verschenen betreffende experimenten met Object-georiënteerde systeemontwikkeling. In [HH93] wordt een project beschreven, waarbij vierentwintig studenten een half jaar lang hebben gewerkt aan Object-georiënteerde en niet-Object-georiënteerde opdrachten. De uitkomsten van het experiment ondersteunden de door de onderzoekers gestelde hypothese, dat met OOP (Objective-C) beter onderhoudbare code wordt geschreven.

In [COM93] wordt een project van het SERC en het tijdschrift Computable in samenwerking met de vakgroep Informatica van de Universiteit van Utrecht beschreven. In dit project testten acht teams, elk bestaande uit twee studenten of twee in de automatisering werkzame personen, een viertal Object-georiënteerde ontwikkelomgevingen. Uit dit onderzoek kwamen de volgende punten naar voren:

- bij de meeste tools komen Object-georiënteerde aspecten alleen terug bij het ontwikkelen van het gebruikersinterface
- de meeste tools gaan uit van een relationele database, waardoor geen gebruik kan worden gemaakt van typisch Object-georiënteerde zaken als overerving en object identiteit
- er bestaan nog tal van onduidelijkheden in de terminologie en het begrippenkader rond Object-Orientatie
- de meeste deelnemers misten ondanks een korte cursus OO duidelijk ervaring, wat zich vertaalde in door conventionele systeemontwikkeling beïnvloede oplossingen op plaatsen, waar een van Object-georiënteerde concepten gebruik makende oplossing beter was geweest

- hoewel hergebruik door de deelnemers als belangrijkste pluspunt werd genoemd, bleek het niet altijd even eenvoudig om met hergebruik rekening te houden
- bij een incrementele manier van ontwikkelen bleek het moeilijk om zicht te houden op de voortgang en op de omvang van het resterende werk

De onderzoekers concludeerden, dat er meer ondersteuning voor ontwikkelactiviteiten nodig is, dan de in het experiment gebruikte tools momenteel bieden.

Deel II

Invoering van Object-georiënteerde technologie in de praktijk

5 Succesfactoren bij invoering van Object-Oriëntatie

5.1 Inleiding

In Deel I hebben wij een algemeen beeld geschetst van het begrip Object-Oriëntatie. Daarbij is ingegaan op vragen als: wat houdt OO in, welke begrippen spelen een rol, wat zijn de voor- en nadelen en wat voor middelen zijn er op dit moment om OO toe te passen? In dit deel gaan wij dieper in op de bruikbaarheid van Object-georiënteerde technologie binnen de automatiseringsafdelingen van een onderneming. Vragen, die in dat verband kunnen worden gesteld, zijn bijvoorbeeld: wanneer is een overstap naar Object-georiënteerde technologie lonend voor een onderneming, wanneer zijn de automatiseringsafdelingen binnen een onderneming toe aan een dergelijke overstap en hoe moet de overstap van conventionele naar Object-georiënteerde systeemontwikkeling worden gemaakt?

In dit hoofdstuk worden de voorwaarden geschetst, waaraan een organisatie moet voldoen om tot een rendabele invoering van Object-georiënteerde technologie te komen. In het verleden is door een aantal auteurs gekeken naar de bruikbaarheid van een methodiek of technologie in de praktijk, onder andere uit het oogpunt van de zogenaamde *tool kit benadering*, waarbij wordt geprobeerd bij ieder te ontwikkelen informatiesysteem gebruik te maken van de meest effectieve methodiek [VER93].

In [AW91] wordt in dit verband opgemerkt, dat het ontwikkelproces afhankelijk is van de gebruikte methodiek, de probleemsituatie en het team van ontwikkelaars en eventueel eindgebruikers, waardoor het systeem wordt ontwikkeld. Voor iedere factor wordt vervolgens een aantal contingentiefactoren, die bepalend zijn voor het proces, aangewezen. In het geval van ontwikkelaars en eindgebruikers worden in [AW91] bijvoorbeeld training, opleiding, culturele achtergrond en ervaring genoemd. In [AW91] wordt toegegeven, dat er nog onvoldoende onderzoek is gedaan naar de redenen, waarom deze afhankelijkheden bestaan, de situaties, waarin zij bestaan, en de manier, waarop zij het ontwikkelproces beïnvloeden.

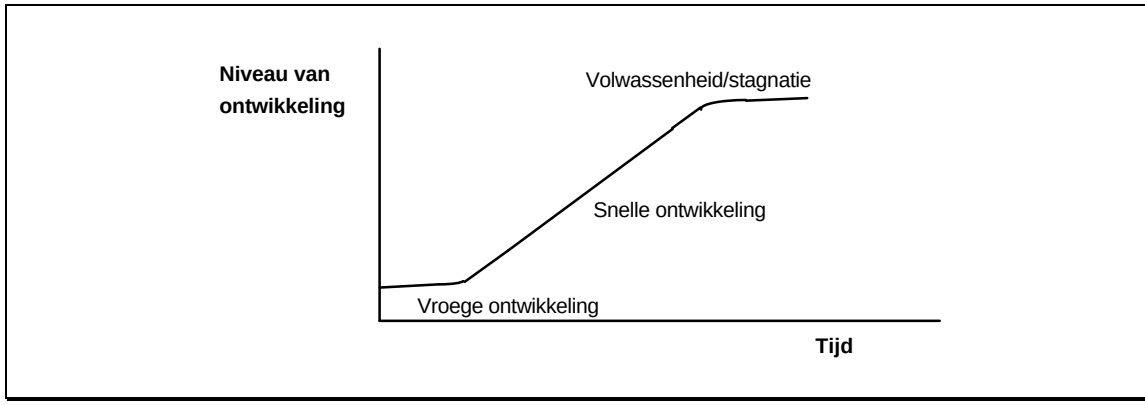
[YOU90] komt met een viertal vragen, die het management van een onderneming zich dient te stellen, wanneer het overweegt over te gaan op Object-georiënteerde technologie.

- Is het Object-georiënteerde paradigma voldoende gerijpt en uitgewerkt?
- Is de Object-georiënteerde technologie efficiënt genoeg te gebruiken, ofwel zijn er voldoende hulpmiddelen beschikbaar?
- Is de organisatie toe aan Object-georiënteerde technologie?
- Is de Object-georiënteerde benadering de meest geschikte benadering om de gewenste systemen mee te ontwikkelen?

Wij gaan in deze studie uit van een aantal factoren, die volgens ons essentieel zijn voor een optimaal gebruik van Object-Oriëntatie. De door ons voorgestelde kritieke succesfactoren zijn:

- de volwassenheid van de technologie
- de organisatie van de automatisering
- de houding van het management
- de systeemontwikkelaars
- het applicatiedomein

In de volgende secties worden deze succesfactoren nader belicht.



Figuur 5.1: De evolutie van een technologie, zoals is beschreven in [YOU90].

5.2 Volwassenheid van de technologie

In [YOU90] wordt onderscheid gemaakt tussen de volwassenheid van de Object-georiënteerde benadering en de mate, waarin de benadering wordt ondersteund door hulpmiddelen als methoden, programmeertalen, tools en DBMSs. Naar onze mening hangen deze factoren nauw samen. In deze sectie worden zij dan ook gezamenlijk bekeken.

Volgens [YOU90] doorloopt bijna iedere nieuwe technologie een drietal fasen, zoals te zien is in figuur 5.1. [YOU90] plaatste Object-georiënteerde technologie op dat moment (1990) ergens in het grensgebied tussen 'vroege ontwikkeling' en 'snelle ontwikkeling'. Inmiddels is, naar onze mening, Object-georiënteerde technologie in de fase 'snelle ontwikkeling' beland, gezien de hoeveelheid literatuur, die op dit moment aan de technologie wordt gewijd, en de hoeveelheid hulpmiddelen, die op de markt verschijnen. Volwassen is de technologie echter zeker nog niet.

Het hangt daarom van de organisatie af, of de technologie rijp genoeg is om er gebruik van te maken. Op 'cutting-edge' technologie georiënteerde ondernemingen zullen eerder geneigd zijn in een vroeg stadium over te stappen op nieuwe technologieën dan meer conservatieve ondernemingen.

5.3 Organisatie van de automatisering

In de voorgaande sectie is gebleken, dat de volwassenheid van een technologie bepalend kan zijn voor de geschiktheid van de technologie voor een onderneming. Maar omgekeerd is ook de volwassenheid van een organisatie bepalend voor het al dan niet slagen van de invoering van een nieuwe technologie.

Nolan ([NG74], [NOL79]) stelt een zestal groeifasen voor, waarin de automatisering binnen een onderneming zich kan bevinden. Deze fasen zijn:

- *initiatie*, gekenmerkt door kostenbesparende, batch-georiënteerde toepassingen, management en specialisatie gericht op technische hulpmiddelen, geringe automatiseringsplanning en gering automatiseringsbeheer, afzijdige gebruikers en centrale financiering

- *diffusie*, gekenmerkt door proliferatie van toepassingen (eilanden van automatisering), management en specialisatie gericht op toepassingsmogelijkheden, geringe automatiseringsplanning en gering automatiseringsbeheer, bovenmatig enthousiaste gebruikers en centrale financiering
- *beheersing*, gekenmerkt door een upgrade van documentatie, herstructurering van bestaande toepassingen, interesse van het middle management voor de beheersbaarheid van automatisering, geformaliseerde planning en geformaliseerd beheer met als doel de explosieve groei in te dammen, ingedamd enthousiasme bij de gebruikers om de automatisering betaalbaar te houden en centrale financiering
- *integratie*, gekenmerkt door herbouw van bestaande toepassingen met behulp van database en netwerktechnologie, metingen door management en gebruikers van systemen naar hun effectiviteit, planning om vraag naar en aanbod van automatiseringsfaciliteiten in evenwicht te brengen en eigen budgetten voor gebruikers om deze te leren kostenbewust te handelen
- *data-oriëntatie*, gekenmerkt door organisatorische integratie van toepassingen, waarvan nog slechts 50% batch-georiënteerd is, data administratie, management, dat zich bewust wordt, dat informatie een belangrijke produktiefactor is, een eerste aanzet tot strategisch beleid en een effectief systeem, waarin gebruikers zelf beslissen over hun automatiseringsbudget
- *verzadiging*, gekenmerkt door de aanwezigheid van een infrastructuur, waarin personal computing, real time en on line toepassingen domineren, data resource management, onderkenning van informatie als strategisch wapen door het top management, strategisch informatiebeleid en gebruikers, die zich gezamenlijk verantwoordelijk voelen voor de infrastructuur van de informatievoorziening

Humphrey komt in [HUM89] tot een vijftal niveaus van volwassenheid, te weten:

- het *initiële niveau*, waarbij nog geen sprake is van formele methodieken, consistentie en standaarden
- het *herhaalbare niveau*, waarbij consensus is bereikt over 'de manier, waarop wij de dingen doen'
- het *gedefinieerde niveau*, waarbij een geformaliseerd en gedocumenteerd ontwikkelproces is ontstaan
- het *gemanagede niveau*, waarbij de organisatie geformaliseerde methoden heeft ingesteld om het ontwikkelproces en de resulterende produkten te meten
- het *geoptimaliseerde niveau*, waarbij de resultaten van de metingen worden gebruikt als feedback bij de verbetering van het ontwikkelproces

Humphrey geeft hier de waarschuwing, dat organisaties, die zich nog niet op het gedefinieerde niveau bevinden, beter niet kunnen overgaan op nieuwe technologieën. "Unless they are introduced with great care, new tools and methods will affect the process, thus destroying the relevance of the intuitive historical base on which the organization relies. Without a defined process framework in which to address these risks, it is even possible for a new technology to do more harm than good."

Deze waarschuwing wordt des te belangrijker, wanneer wij kijken naar door Yourdon in [YOU90] gepresenteerde cijfers, waaruit blijkt, dat in 1990 in de Verenigde Staten nog 85% van de bedrijven zich op het initiële niveau bevond. 12% bevond zich op het herhaalbare niveau en slechts 3% op het gedefinieerde niveau. De hoogste niveaus waren door nog geen enkele Amerikaanse onderneming bereikt.

In [JCJ+92] worden de theorieën van Humphrey en Yourdon enigszins afgezwakt. Jacobson stelt, dat hij een aantal ondernemingen, die zich nog op het herhaalbare niveau bevonden en die toch volwassen ge-

noeg bleken om de overstap naar Object-georiënteerde technologie aan te kunnen, heeft gezien. Een gedegen voorbereiding en voldoende inspanning bleken in die gevallen vereisten te zijn.

5.4 Houding van het management

In hoofdstuk 3 (Management van Object-georiënteerde projecten) is gebleken, dat Object-georiënteerde systeemontwikkeling een andere vorm van management vereist dan conventionele systeemontwikkeling. In [PIT93] wordt opgemerkt, dat dit in de praktijk nog nauwelijks het geval is.

Voor geslaagde invoering van Object-georiënteerde technologie is ondersteuning en medewerking van het management onontbeerlijk. Het management moet zich in de technologie verdiepen en moet zich een nieuwe manier van projectmanagement eigen maken. Hierbij draait het in hoofdzaak om een viertal veranderingen: een nieuwe manier van doelstellingen bepalen, een nieuwe manier van projecten plannen, een nieuwe manier van voortgang bepalen en een nieuwe manier van middelen alloceren [PIT93]. Voor meer informatie over de rol van het management bij Object-georiënteerde systeemontwikkeling wordt verwezen naar hoofdstuk 3 (Management van Object-georiënteerde projecten) en naar relevante literatuur ([PIT93] en [SM93], alsmede hoofdstukken van [BOO91] en [JCJ$^+$92]).

Daarnaast is de medewerking van het management op een ander gebied nodig. Zoals in sectie 2.3 (Nadelen van OO) is gebleken, brengt een overstap naar Object-georiënteerde technologie de nodige kosten op het gebied van opleiding van personeel en management en investeringen in hard- en software met zich mee. Deze kosten kunnen worden terugverdiend door de lagere onderhoudskosten en door de betere mogelijkheden tot hergebruik van modellen en code (zie sectie 2.2, Voordelen van OO). Aangezien in sectie 3.5 (Meten aan Object-georiënteerd ontwikkelen) is gebleken, dat ontwikkelen met het oog op hergebruik pas voordelen oplevert na twee tot drie projecten, en onderhoud pas een rol speelt na oplevering van het systeem, wordt een Object-georiënteerde manier van ontwikkelen pas na verloop van tijd winstgevend. Het management moet daarom bereid zijn te investeren in een technologie, die pas op langere termijn commerciële voordelen gaat bieden.

5.5 Systeemontwikkelaars

Zoals in de inleiding van dit hoofdstuk reeds is opgemerkt, wordt in [AW91] een aantal contingentiefactoren, die betrekking hebben op de leden van projectteams, gegeven. Concreet worden genoemd training, opleiding, culturele achtergrond en ervaring. In sectie 3.4 (Personeelsbezetting) is in dit verband reeds opgemerkt, dat de ervaring leert, dat het niveau van ervaring en training van de ontwikkelaars een kritieke succesfactor is bij Object-georiënteerde systeemontwikkeling [PIT93].

Aan het hierboven opgesomde rijtje dient naar onze mening motivatie toegevoegd te worden. De kans is groot, dat een systeem, dat tegen de wil van de ontwikkelaars met behulp van Object-georiënteerde hulpmiddelen is ontwikkeld, kwalitatief minder goed zal zijn dan een systeem, dat door diezelfde ontwikkelaars op de oude vertrouwde manier is ontwikkeld.

5.6 Applicatiedomein

In de inleidende sectie van dit hoofdstuk is reeds de tool kit benadering aan de orde gekomen. Deze benadering gaat uit van een gereedschapskist vol methodieken, waarover een ontwikkelafdeling zou moeten beschikken. Voor ieder project wordt uit deze gereedschapskist de meest geschikte methodiek gekozen. Hoewel voorstanders van een dergelijke benadering toegeven, dat er nog weinig onderzoek is gedaan naar

de criteria, waarop een dergelijke keuze moet worden gebaseerd (zie onder meer [BS87], [AW91]), valt voor de idee achter de benadering wel degelijk wat te zeggen.

Object-georiënteerde technologie moet in een dergelijke benadering niet worden gezien als de oplossing voor de problemen binnen alle bestaande toepassingsgebieden, maar veeleer als een technologie, die voor het ontwikkelen van een bepaald type informatiesystemen het meest geschikt is. In dit verband willen wij nog eens duidelijk aangeven, dat er verschil bestaat tussen de Object-georiënteerde benadering, ofwel het oplossen van problemen op basis van Object-georiënteerde uitgangspunten, en Object-georiënteerde technologie, ofwel het geheel van Object-georiënteerde methoden, technieken, talen en hulpmiddelen. Naar onze mening moeten systemen vanuit een Object-georiënteerde benadering worden ontwikkeld, maar kan hierbij, in het geval, dat adequate Object-georiënteerde hulpmiddelen ontbreken, worden gekozen voor niet-Object-georiënteerde technologieën.

Daar de tool kit benadering op dit moment nog onvoldoende is uitgewerkt, volgen wij in dit onderzoek een omgekeerde werkwijze. Geautomatiseerde systemen kunnen naar onze mening worden ondergebracht in een aantal toepassingsgebieden of applicatiedomeinen, die niet noodzakelijk disjunct zijn. Voor elk applicatiedomein kunnen vervolgens één of meer methodieken worden aangewezen, die het meest geschikt zijn om te gebruiken bij de ontwikkeling van tot dat applicatiedomein behorende systemen. Wanneer de Object-georiënteerde technologie volwassen is geworden en de Object-georiënteerde benadering op dat moment nog niet is achterhaald door een nieuwe benadering, zullen naar onze mening deze methodieken allemaal een Object-georiënteerde basis hebben. Op dit moment is Object-georiënteerde technologie vooral goed toepasbaar binnen de applicatiedomeinen, die in het vervolg van deze sectie worden behandeld.

Een eerste groep toepassingen, waarvoor Object-georiënteerde technologie zich uitstekend leent, wordt gevormd door toepassingen met een *grafisch gebruikersinterface*. In dit verband worden in [YOU90] de woorden van een software manager bij IBM geciteerd: "attempts to use standard structured techniques to build applications in an OS/2 Extended Edition Presentation Manager (PM) environment had been dismal failures; the only successes they could point to were applications developed with object-oriented techniques."

Een tweede groep toepassingen wordt gevormd door *dynamische informatiesystemen*. Hieronder verstaan wij systemen met een groot aantal interne toestanden en een groot aantal met elkaar communicerende deelsystemen. Dergelijke systemen passen precies in de Object-georiënteerde zienswijze van objecten, die via boodschappen met elkaar communiceren. [SOM89] beschrijft dit toepassingsgebied aan de hand van een tegenvoorbeeld: typisch niet-dynamische systemen zijn automaten, zoals geld-, betaal- en koffieautomaten. Dergelijke systemen kennen slechts een beperkt aantal interne toestanden en herhalen telkens hetzelfde proces, waarbij getermineerde processen nauwelijks invloed hebben op de actieve processen.

Voorbeelden van dynamische systemen zijn *bestuurlijke informatiesystemen*, die informatie moeten produceren op basis van steeds veranderende gegevens. Bij deze groep speelt de onderhoudbaarheid een grote rol. Voor bestuurlijke informatiesystemen kan het van groot belang zijn, dat zij actueel zijn. Zij moeten de structuur en de omgeving van de organisatie op het moment van gebruik beschrijven en niet een situatie, die in het verleden heeft bestaan. Veranderingen in de werkelijkheid (het Universum van Discussie) moeten daarom zo snel mogelijk worden verwerkt in het informatiesysteem. In paragraaf 2.2.3 (Beter onderhoud) is gebleken, dat Object-georiënteerde technologie dergelijk onderhoud beter ondersteunt dan conventionele technologieën.

Een derde applicatiedomein wordt gevormd door systemen met een hoog '*herbruikbaarheidsgehalte*'. Hiermee bedoelen wij zowel systemen, die zijn ontwikkeld op basis van herbruikbare componenten als systemen, waarvan de componenten zich lijken te lenen voor toekomstig hergebruik. Tot de eerste groep behoort onder andere een groot deel van de eerder genoemde toepassingen met grafische gebruikersinterfa-

ces, terwijl bij de tweede groep kan worden gedacht aan informatiesystemen in een decentrale organisatie, die zijn opgebouwd uit afdelingsspecifieke en door het gehele bedrijf bruikbare componenten.

5.7 Een methode om de rijpheid van een onderneming voor OO te bepalen

In de vorige secties is een aantal factoren, die een kritieke rol spelen bij het al dan niet slagen van de invoering van Object-georiënteerde technologie in een onderneming, geschetst. Om een beter beeld te krijgen van de mate, waarin de betreffende factoren een rol spelen bij invoering op een bepaald tijdstip in een bepaalde onderneming, wordt in deze sectie een methode gepresenteerd, die de rol van een aantal door de succesfactoren te beïnvloeden eigenschappen van onderneming en technologie weergeeft. Hierbij wordt gebruik gemaakt van een bestaande methode, die eerst wordt beschreven.

5.7.1 De CASE/IM methode

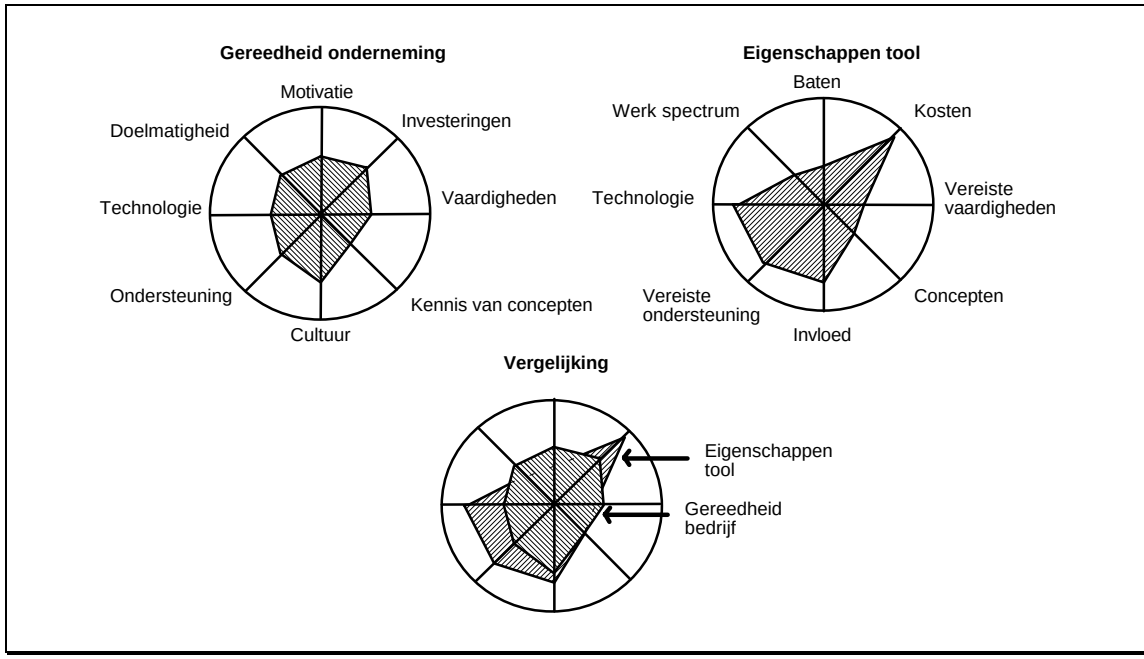
In [JON91] wordt de *CASE/IM* methode, waarmee kan worden bepaald of een onderneming 'klaar' is voor het gebruik van een bepaald CASE tool, beschreven. Deze methode, die is ontwikkeld door Howard Rubin Associates in New York en wordt gepresenteerd in [RUB91], zet het *gereedheidsniveau* van een bedrijf af tegen de eigenschappen van het tool. Dit gebeurt met behulp van *Kiviat charts*, die worden gebruikt om een aantal gerelateerde eigenschappen in hetzelfde diagram te vertonen. De Kiviat chart is een cirkel, waarin iedere eigenschap wordt gerepresenteerd door een lijnstuk van het middelpunt van de cirkel naar de cirkel zelf. De score van een bepaalde eigenschap wordt weergegeven door een punt op het bijbehorende lijnstuk; hoe hoger de score, des te verder is het punt verwijderd van het middelpunt. De punten op de verschillende lijnstukken worden met elkaar verbonden en vormen zo een veelhoek. Zaken, in dit geval het gereedheidsniveau van het bedrijf en de eigenschappen van het tool, kunnen nu met elkaar worden vergeleken door de corresponderende veelhoeken met elkaar te vergelijken.

De CASE/IM methode gebruikt voor het bepalen van de gereedheid van een onderneming de volgende eigenschappen:

1. *motivatie*: de behoefte aan verbetering van de produktiviteit en de kwaliteit
2. *investeringen*: de bereidheid om vereiste investeringen te doen
3. *vaardigheden*: de in de onderneming aanwezige vaardigheden op het gebied van systeemontwikkeling en de mogelijkheden en bereidheid om die vaardigheden daadwerkelijk te gebruiken
4. *kennis van concepten*: de kennis van de concepten, waarop het tool is gebaseerd, en de mogelijkheden en bereidheid om die kennis daadwerkelijk te gebruiken
5. *cultuur*: de bereidheid om nieuwe hulpmiddelen optimaal te gebruiken
6. *ondersteuning*: de aanwezigheid van afdelingen voor bijvoorbeeld kwaliteitszorg en research & development
7. *technologie*: de op het moment van vergelijking gebruikte technologie
8. *doelmatigheid*: het percentage van de investeringen, dat wordt besteed aan nieuwe ontwikkelingen

Tegenover deze eigenschappen worden de volgende eigenschappen van het tool geplaatst:

1. *baten*: de verwachte verbetering van de produktiviteit en de kwaliteit
2. *kosten*: de aan het tool verbonden kosten, waartoe ook trainingskosten en dergelijke worden gerekend



Figuur 5.2: Een aan [JON91] ontleend voorbeeld van het gebruik van de CASE/IM methode.

3. *vereiste vaardigheden*: de voor efficiënt gebruik van het tool benodigde vaardigheden
4. *concepten*: het aantal concepten, waarop het tool is gebaseerd, en de bekendheid van die concepten
5. *invloed*: de invloed van het tool op het totale ontwikkelproces
6. *vereiste ondersteuning*: de ondersteuning, die gebruik van het tool vereist
7. *technologie*: de voor gebruik van het tool vereiste technologie
8. *werk spectrum*: de hoeveelheid door het tool ondersteunde nieuwe ontwikkelingen

Deze eigenschappen worden uitgedrukt in een score, die kan variëren van 0 tot 10.

Voorbeeld 5.7.1

Figuur 5.2 is een voorbeeld van het gebruik van de CASE/IM methode voor het vergelijken van de eigenschappen van een tool en de gereedheid van een bedrijf voor het gebruik van dat tool. Uit de vergelijking blijkt, dat er problemen zijn met de kosten van het tool, de benodigde ondersteuning, de invloed van het tool op het ontwikkelproces en de benodigde technologie. Voor wat de overige eigenschappen betreft, is het bedrijf gereed voor gebruik van het tool.

5.7.2 De CASE/IM methode en Object-Oriëntatie

Hoewel de CASE/IM methode bedoeld is voor het bepalen van de gereedheid van een onderneming voor het gebruik van een bepaald CASE tool, is de methode in een aangepaste vorm ook te gebruiken voor het bepalen van de gereedheid van een onderneming voor het gebruik van Object-georiënteerde technologie.

Hetzelfde geldt voor decentrale eenheden binnen een onderneming, maar voor het gemak wordt in deze paragraaf alleen gekeken naar gehele ondernemingen.

Om CASE/IM geschikt te maken voor dergelijk gebruik moet een iets andere invulling worden gegeven aan de verschillende eigenschappen van respectievelijk onderneming en OO technologie. Daarbij gaat het om de volgende eigenschappen voor de onderneming:

1. *motivatie*: de behoefte aan verbetering van de produktiviteit en de kwaliteit
2. *investeringen*: de bereidheid om vereiste investeringen te doen
3. *vaardigheden*: de in de onderneming aanwezige kennis van Object-georiënteerde methoden, technieken en programmeertalen en de mogelijkheden en bereidheid om die kennis te gebruiken
4. *cultuur*: de bereidheid om over te stappen op Object-georiënteerde technologie
5. *ondersteuning*: de aanwezigheid van afdelingen voor bijvoorbeeld kwaliteitszorg en research & development
6. *hergebruik*: de behoefte aan herbruikbare ontwerpen en software

De volgende eigenschappen van OO technologie kunnen daar tegenover worden geplaatst:

1. *baten*: de verwachte verbetering van de produktiviteit en de kwaliteit
2. *kosten*: de aan de overstap naar OO verbonden kosten
3. *vaardigheden*: de voor efficiënt gebruik van OO benodigde vaardigheden
4. *invloed*: de invloed van een overstap naar OO op het totale ontwikkelproces
5. *vereiste ondersteuning*: de ondersteuning, die het gebruik van OO technologie vereist
6. *hergebruik*: de invloed van hergebruik op het totale ontwikkelproces

In vergelijking met de CASE/IM methode is een drietal eigenschappen, die vooral te maken hebben met specifieke eigenschappen van tools, namelijk ondersteunde methoden en technieken, technologie en werk spectrum, buiten beschouwing gelaten, terwijl een nieuwe eigenschap, hergebruik, is toegevoegd. Hergebruik is namelijk één van de karakteristieke eigenschappen van OO en voor een aantal mensen dé reden om over te stappen op OO. Het is daarom belangrijk om ook de mogelijkheden tot hergebruik van ontwerp en software en de daarmee gepaard gaande veranderingen bij de vergelijking te betrekken.

Toekenning van waarden aan de verschillende eigenschappen van de onderneming kan geschieden op basis van een evaluatie van de eerder in dit hoofdstuk gepresenteerde succesfactoren. De waarden van de eigenschappen van OO kunnen worden bepaald op basis van relevante literatuur.

Als aan de hand van de bovenstaande methode kan worden geconcludeerd, dat de beschouwde onderneming gereed is voor een overstap naar OO, dan dient een keuze gemaakt te worden voor bepaalde OO methoden, OO programmeertalen, OO tools en eventueel OODBMSs. Voor de evaluatie van OO tools kan gebruik worden gemaakt van het originele CASE/IM model, waaraan één extra eigenschap moet worden toegevoegd:

9. *hergebruik*: de behoefte aan herbruikbare ontwerpen en software (gereedheid bedrijf) tegenover de mogelijkheden om herbruikbare ontwerpen en software te ontwikkelen en te beheren (eigenschappen tool)

6 Wat te doen bij invoering van Object-Oriëntatie?

6.1 Inleiding

In deze sectie wordt een aantal aanbevelingen gedaan voor het geval, dat een onderneming besluit over te gaan tot invoering van OO. Een aantal van deze aanbevelingen blijft ook gelden in het geval, dat niet wordt overgegaan tot invoering van OO. Ook dan dient namelijk de nodige aandacht besteed te worden aan de technologie, bijvoorbeeld in de vorm van proefprojecten.

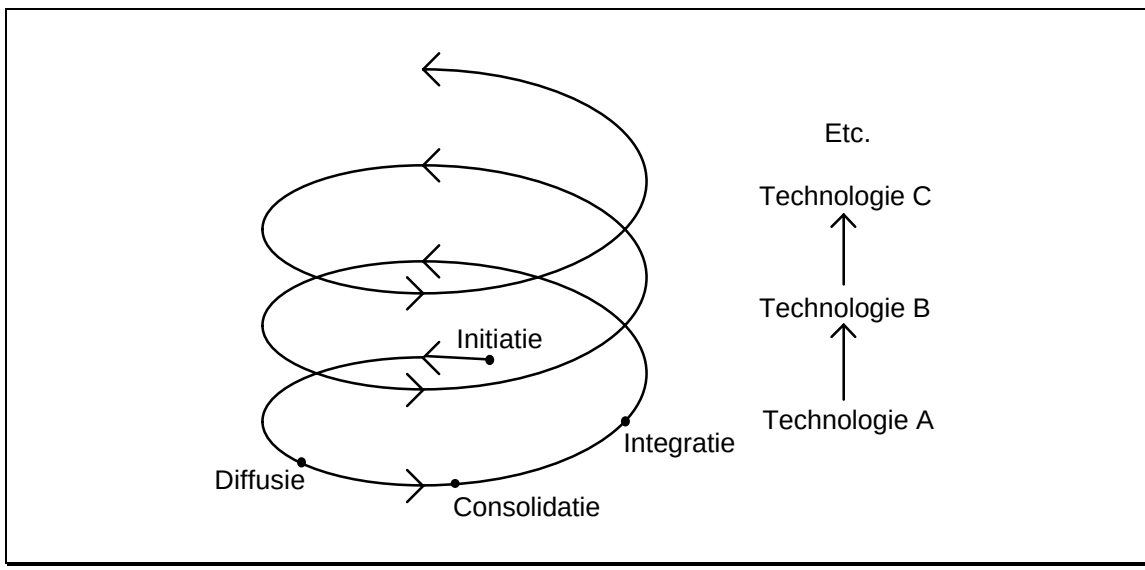
In dit hoofdstuk wordt aandacht besteed aan de manier, waarop de technologie geïntroduceerd dient te worden, en aan de hulpmiddelen, die bij een dergelijke introductie moeten worden gekozen.

Voor algemene regels over het gebruik van OO binnen een onderneming wordt verwezen naar deel I en naar [CCH+94], waarin op basis van drie Object-georiënteerde projecten uitspraken worden gedaan over de opzet van Object-georiënteerde projecten. Daarbij komen onder andere prototyping, hergebruik van code, overgangen tussen verschillende fasen in het ontwikkelproces en de samenstelling van projectgroepen aan bod.

6.2 Introductie van Object-Oriëntatie

In [BEM91] wordt een op Nolan ([NG74], [NOL79]) gebaseerd genuanceerd fasenmodel geïntroduceerd, dat de *technologische spiraal* wordt genoemd. Dit model gaat uit van het voortdurend beschikbaar komen van nieuwe technologieën. Voor elke technologie kunnen vier stadia worden onderscheiden, te weten:

- *initiatie*: het op kleine schaal experimenteren met een nieuwe technologie, ten einde kennis en ervaring op te bouwen



Figuur 6.1: De fasen in automatiseringstechnologieën: de technologische spiraal. Deze figuur is overgenomen uit [BEM91].

- *diffusie*: het op brede schaal toepassen van de nieuwe technologie
- *consolidatie*: de economische uitnutting van de nieuwe technologie
- *integratie* of *substitutie*: het vervangen van verouderde technologieën door de inmiddels vertrouwde nieuwe technologie (substitutie) of het integreren van de nieuwe technologie in de gehele infrastructuur van hulpmiddelen (integratie)

Deze technologische spiraal is grafisch weergegeven in figuur 6.1.

Iedere nieuwe technologie moet daarom worden ingevoerd met behulp van *proefprojecten*. Bij een positieve evaluatie van die proefprojecten moet de technologie op brede schaal worden toegepast.

Ook in het geval van invoering van Object-georiënteerde technologie bij een onderneming is het verstandig om de nieuwe technologie te introduceren door middel van proefprojecten. Een proefproject moet naar onze mening voldoen aan een aantal voorwaarden.

- *Plaats binnen de organisatie*. Het proefproject moet worden ondergebracht in een projectgroep, die buiten de bestaande organisatiestructuur wordt geplaatst. De leden van de projectgroep moeten tijdelijk uit hun afdeling worden gehaald. De projectgroep moet een eigen locatie krijgen, zodat de projectgroepleden elkaar altijd snel kunnen vinden en een goede onderlinge communicatie gegarandeerd is.
- *Leden van de projectgroep*. De projectgroep moet niet te groot zijn: een projectmanager, een klein aantal systeemontwikkelaars (vier à vijf), waaronder één projectleider, en een OO specialist, die zelf geen ontwikkelwerk doet, maar die fungeert als begeleider van de ontwikkelaars en die de kwaliteit van de geleverde produkten bewaakt. De ontwikkelaars hoeven niet noodzakelijkerwijs afkomstig te zijn van dezelfde afdeling. De belangrijkste factor bij de keuze van de leden van de projectgroep is motivatie: de ontwikkelaars moeten de wil hebben het project tot een goed einde te brengen. Als binnen de organisatie geen geschikte kandidaat voor de functie van specialist kan worden gevonden, dan moet daarvoor extern iemand worden aangetrokken.
- *Methoden, talen en tools*. De ontwikkelaars moeten gebruik maken van een zo klein mogelijk aantal methoden, talen en hulpmiddelen. Deze dienen voor aanvang van het proefproject vastgesteld te zijn. Deze beslissing reikt verder dan het proefproject: om hergebruik van bestaande produkten te bevorderen zal in de toekomst waarschijnlijk gebruik worden gemaakt van dezelfde methoden, talen en tools. In de volgende sectie wordt dieper op dit onderwerp ingegaan.
- *Scholing*. Voordat het project een aanvang neemt, moeten de projectgroepleden, óók de projectmanager, een gedegen opleiding in Object-Oriëntatie in het algemeen en de gebruikte methodiek, de gebruikte programmeertaal en de gebruikte hulpmiddelen in het bijzonder krijgen. Hierbij kan bijvoorbeeld worden gedacht aan een cursus van een week. De specialist hoeft een dergelijke cursus niet te volgen, maar moet wel op de hoogte zijn van de behandelde stof. Als een dergelijke opleiding intern kan worden verzorgd, dan wordt aangeraden van die mogelijkheid gebruik te maken, maar vermoedelijk is het noodzakelijk om hiervoor een externe organisatie in te schakelen.
- *Het systeem*. Het is het beste om een reeds bestaand systeem opnieuw te ontwikkelen, maar dit keer op een Object-georiënteerde manier. Dit maakt het mogelijk om de conventionele en de Object-georiënteerde wijze van systeemontwikkeling optimaal te vergelijken. Als het niet mogelijk is om een bestaand systeem opnieuw te ontwikkelen en wordt gekozen voor de ontwikkeling van een nieuwe toepassing, dan moeten aan die toepassing de volgende voorwaarden worden gesteld:

  - het aantal interfaces met bestaande systemen moet zo klein mogelijk zijn, zodat zo min mogelijk tijd besteed hoeft te worden aan de ontwikkeling van interfaces naar conventionele systemen en aan het documenteren en onderhouden van die interfaces
  - het systeem moet representatief zijn voor bancaire applicaties, zodat een aantal van de ontwikkelde klassen in de toekomst kan worden hergebruikt en daarom met het oog op herbruikbaarheid ontwikkeld dient te worden

- *Voorbereiding*. Voor aanvang van het project dient de nodige zorg besteed te zijn aan kwaliteitsnormen en aan methoden om complexiteit, kosten en dergelijke van het proefsysteem te meten. Als dat niet wordt gedaan, dan is een goede evaluatie achteraf onmogelijk.
- *Uitvoering*. Tijdens het proefproject moet de nodige zorg worden besteed aan project management en object management, ook in het geval, dat de aard van het project dat niet vereist. Een proefproject moet namelijk een goed beeld geven van de technologie en daarom dienen alle aspecten aan bod te komen. Op basis van de bevindingen moeten na afloop van het proefproject algemene richtlijnen worden opgesteld.
- *Evaluatie*. Het proefproject dient achteraf grondig geëvalueerd te worden. Hierbij moet niet alleen worden gekeken naar het opgeleverde systeem, maar ook naar de wijze, waarop het project is gemanaged, naar het aantal herbruikbare klassen, dat is opgeleverd, en naar de kwaliteit van die klassen, naar de herbruikbaarheid van de vervaardigde analyse- en ontwerpmodellen, naar de invloed, die methodiek, programmeertaal en hulpmiddelen hebben gehad op het project, enzovoort.

6.3 Methoden, talen en tools opnieuw bekeken

In hoofdstuk 4 (Methoden, talen en tools) is uitgebreid aandacht besteed aan de verschillende methoden en technieken, programmeertalen, hulpmiddelen, DBMSs en standaarden. Wij hebben getracht in dat hoofdstuk een zo volledig mogelijk beeld te schetsen van de huidige situatie. Vergelijkingen tussen bijvoorbeeld verschillende methoden zijn daarbij alleen gemaakt, indien zij een bijdrage leveren aan dat algemene beeld.

In deze sectie wordt een aantal aanbevelingen gedaan met betrekking tot beslissingen over de aanschaf van verschillende methoden, programmeertalen, DBMSs en tools. Een aantal van deze aanbevelingen is gebaseerd op in deze sectie gepresenteerde vergelijkingen. Die vergelijkingen zijn, gezien de beperkte omvang van deze voorstudie, waar mogelijk gebaseerd op uitspraken van andere auteurs. In deze sectie wordt niet gekozen voor een bepaalde methode of programmeertaal. Dergelijke keuzen zouden op het tijdstip, waarop de daadwerkelijke beslissingen worden genomen, achterhaald kunnen zijn, gezien de snelheid, waarmee op dit moment nieuwe produkten worden geïntroduceerd.

6.3.1 Onderlinge samenhang

Bij de introductie van Object-georiënteerde technologie binnen een onderneming moet worden gekozen voor methodieken, programmeertalen, tools en DBMSs. Het is onverstandig om deze keuzen onafhankelijk van elkaar te maken, omdat:

- methoden vaak zijn gebaseerd op programmeertalen, waardoor met behulp van een bepaalde methode gemaakte ontwerpen vaak het beste met behulp van de programmeertaal, die aan de basis stond van de methode, kunnen worden gecodeerd.

- tools vaak slechts één of enkele methoden ondersteunen en code genereren in één of enkele programmeertalen, zodat de keuze voor een methode of programmeertaal het aantal bruikbare tools sterk beïnvloedt. Een methode of programmeertaal kan daarom niet worden gekozen zonder evaluatie van de ondersteunende tools.
- het bij een aantal methoden wenselijk is om bij de implementatie gebruik te maken van een OODBMS, terwijl andere methoden de wijze voorschrijven, waarop ontwerpen kunnen worden afgestemd op een relationeel DBMS. Kiezen voor een methode, die een OODBMS vereist, is daarom niet mogelijk zonder eerst de beschikbare OODBMSs te evalueren.
- OODBMSs, in tegenstelling tot bijvoorbeeld relationele DBMSs, interfaces naar OO programmeertalen bieden. Het is daarom niet verstandig om tot de keuze voor een programmeertaal over te gaan, voordat duidelijk is wat voor soort DBMS zal worden gebruikt. Omgekeerd dient bij de keuze voor een OODBMS rekening gehouden te worden met de programmeertalen, die door het DBMS worden ondersteund.

De hierboven genoemde redenen maken duidelijk, dat een coherent stelsel van op elkaar aansluitende methoden, programmeertalen, tools en DBMSs gekozen dient te worden. Als die aansluiting niet volledig is, dan dient zij ondersteund te worden met behulp van zo nodig zelf ontwikkelde geautomatiseerde conversiesystemen.

6.3.2 Methoden

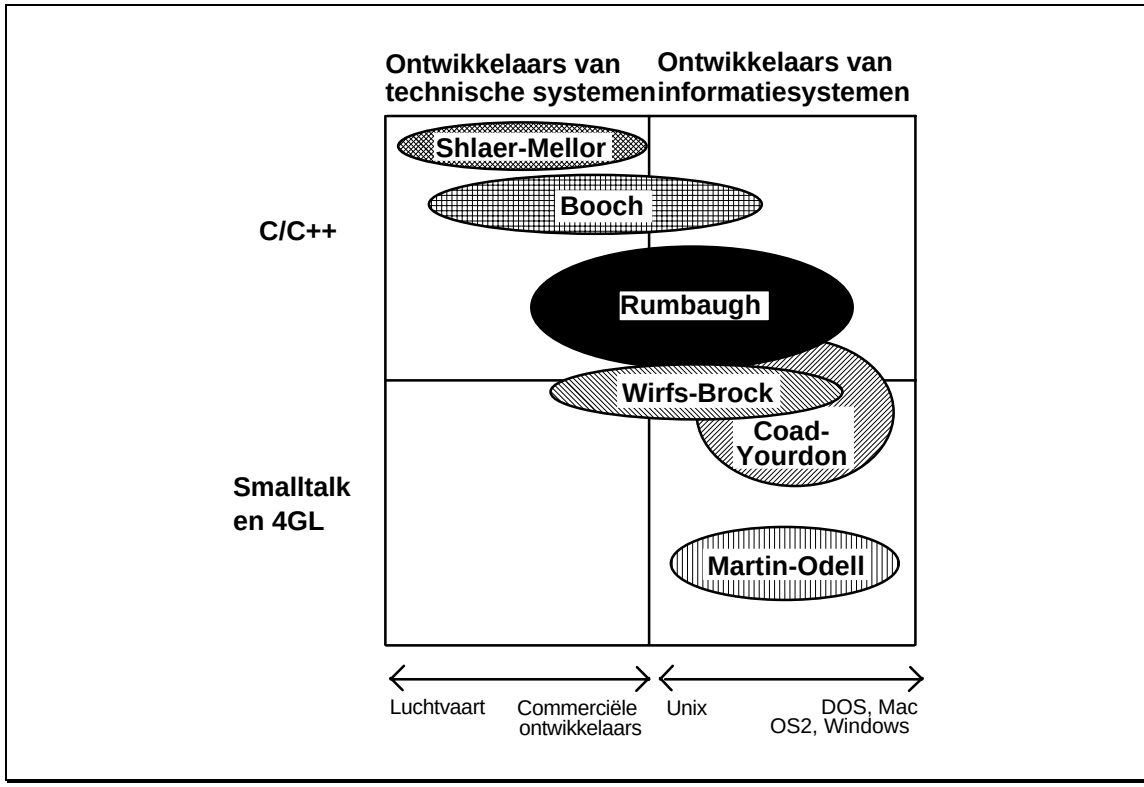
In sectie 4.2 (Methoden en technieken) hebben wij een zevental methoden beschreven. In een door de GartnerGroup verzorgde lezing (terug te vinden in [WES93]) werden zes van die zeven methoden met elkaar vergeleken. In figuur 6.2 is de uitkomst daarvan weergegeven. De methoden zijn hierin gerangschikt naar de applicatiedomeinen, waarin zij het beste tot hun recht komen, en naar de programmeertalen, die het beste op de methoden aansluiten. Hierbij moet worden opgemerkt, dat de zevende door ons behandelde methode, te weten de OOSE methode van Jacobson et al., niet in de grafiek is opgenomen. Deze methode kan naar onze mening in het middengebied worden geplaatst.

In [PRO94] wordt door Pronk een vergelijking tussen een achttal methoden gepresenteerd. Het resultaat daarvan is weergegeven in tabel 6.1.

De meeste in de vergelijking betrokken methoden worden door ons in sectie 4.2 (Methoden en technieken) behandeld. Objectory is de uitgebreide variant van OOSE, de methode van Jacobson et al. OMT is de methode van Rumbaugh et al. en OOA/OOD is de methode van Coad en Yourdon. Bij de evaluatie van de

Methode	Vereisten		Ontwerp	Codering	Formalisme	Beknoptheid	Ondersteunende tools
	Verkrijgen	Analyse					
Objectory	+	++	+	+	-	laag	1
OMT	•	+	+	•	•	gemiddeld	3^+
Booch 91/94	-	-	+	•	-	laag	1
Wirfs-Brock	-	•	+	•	–	gemiddeld	-
KISS	-	•	•	-	–	laag	1
OOA/OOD	-	•	-	–	-	laag	2^+
Shlaer/Mellor	-	+	•	-	-	gemiddeld	3^+
Fusion	+	++	++	+	+	hoog	3^+

Tabel 6.1: Een deel van de resultaten van de in [PRO94] gemaakte vergelijking van acht methoden.



Figuur 6.2: Een plaatsing van methodieken, gerelateerd aan de programmeertalen, die er het beste op aansluiten, en het applicatiedomein, waarin zij het beste tot hun recht komen. Deze figuur is ontleend aan [WES93].

methode van Booch is het door ons niet besproken [BOO94] meegenomen. KISS, de met name voor administratieve toepassingen bedoelde methode van Kristen (zie [KRI93]), en Fusion, de bij Hewlett-Packard ontwikkelde methode van Coleman et al., die vooral geschikt is voor technische applicaties (zie [CAB+93]), worden niet in deze scriptie besproken.

Opmerkelijke resultaten van deze vergelijking zijn de hoge score van Objectory en Fusion en de uitermate lage score van de methode van Coad en Yourdon, die alleen voor hun analysefase een voldoende krijgen. Pronk ziet de breedsprakigheid, waarvan de schrijvers zich bedienen, als het grote minpunt van de methode van Coad en Yourdon. Evenals ons constateert Pronk de onvolwassenheid van de methode van Wirfs-Brock et al., die hij wel aanraadt als een goede introductie voor de leek op het gebied van OO, en de grote kans op inconsistente modellen bij het gebruik van de methode van Rumbaugh et al. Van de methode van Booch wordt opgemerkt, dat zij nog te instabiel is, als zwak punt van de methode van Jacobson et al. wordt het toestandsmodel gezien en van zowel de methode van Shlaer en Mellor als de Fusion methode wordt opgemerkt, dat zij vrij veel verschillende modellen kennen, zodat de kans op inconsistentie groot is.

6.3.3 Programmeertalen en hulpmiddelen

In de secties 4.3 (Programmeertalen) en 4.5 (Hulpmiddelen) is aandacht besteed aan de belangrijkste Object-georiënteerde programmeertalen en tools van dit moment. Daar tools meer en meer evolueren tot

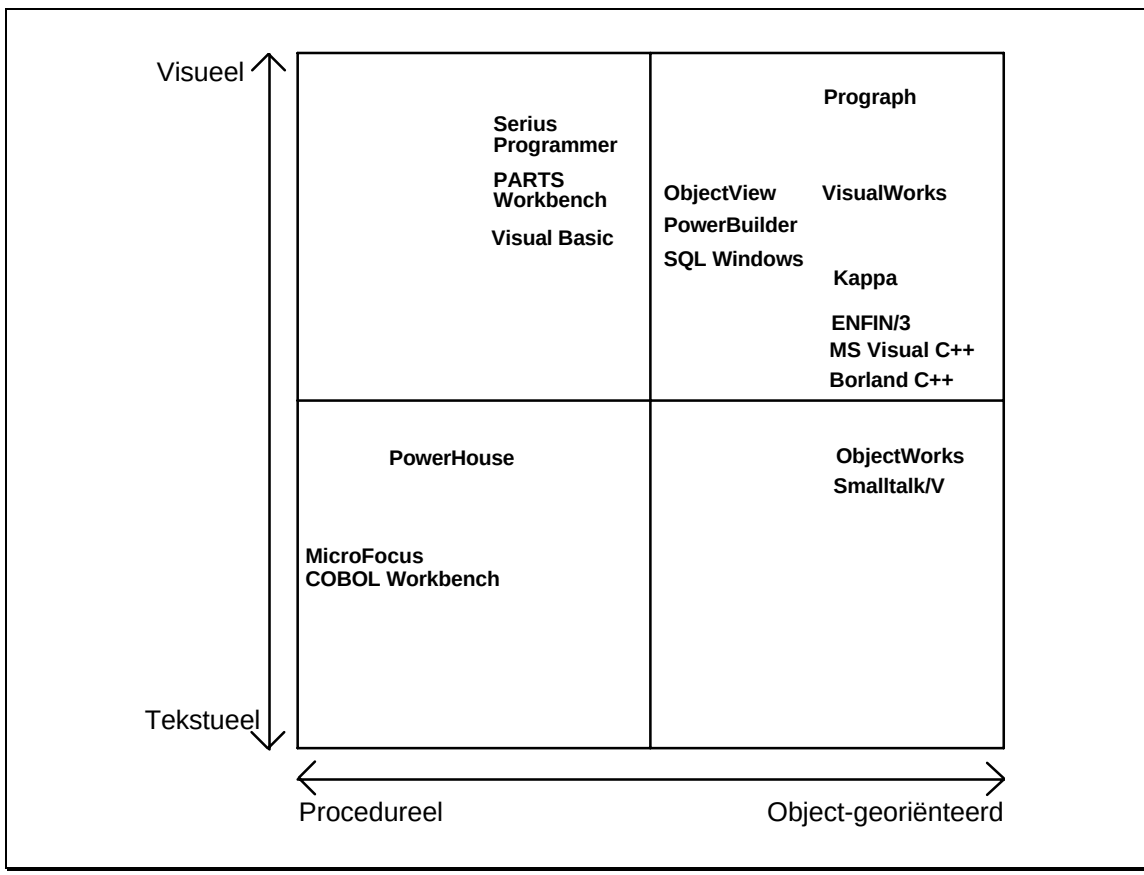
I-CASE tools, kunnen programmeertalen en tools bijna niet meer los van elkaar worden beschouwd. In deze paragraaf wordt daarom tegelijkertijd gekeken naar programmeertalen en naar bijbehorende tools.

Figuur 6.3 is eveneens uit de in de vorige paragraaf genoemde lezing van de GartnerGroup (zie [WES93]) overgenomen. Deze figuur geeft een overzicht van de belangrijkste hulpmiddelen voor het ontwikkelen van geautomatiseerde systemen op het moment van de lezing (juni 1993). De hulpmiddelen zijn gerangschikt op basis van twee eigenschappen, te weten de mogelijkheden voor de ontwikkeling van grafische gebruikersinterfaces en de aanwezigheid van Object-georiënteerde eigenschappen.

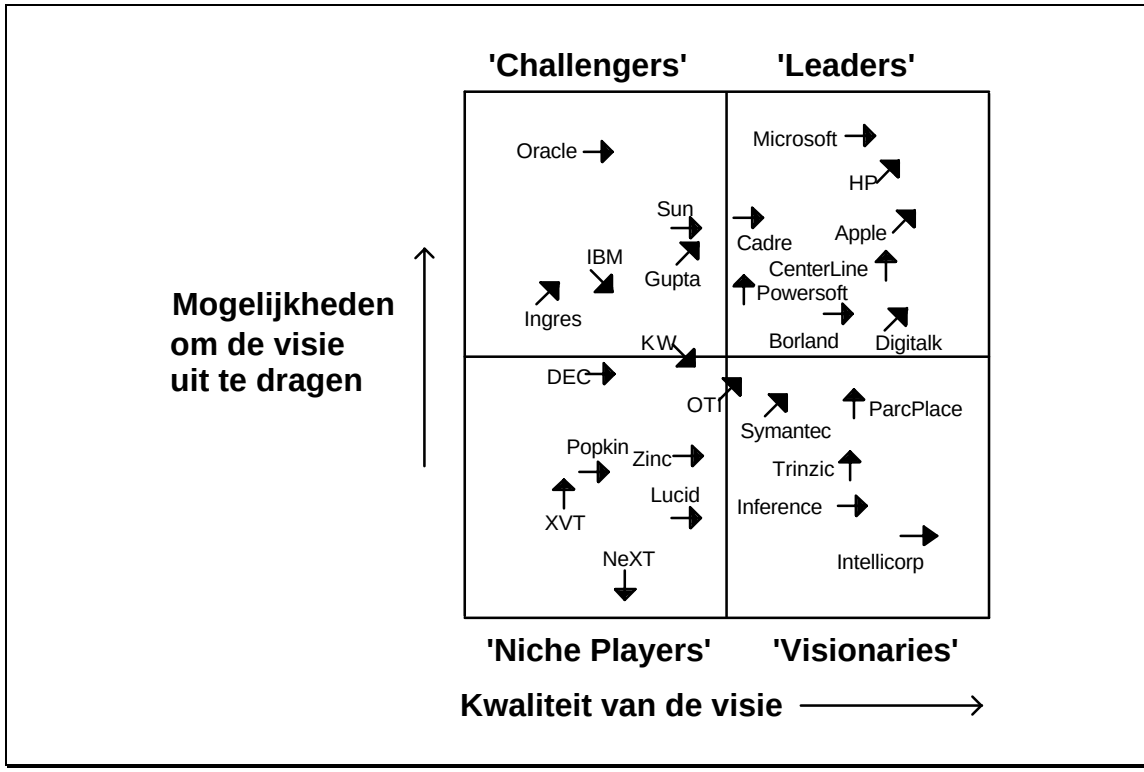
Een voorspelling over de toekomstverwachtingen voor ondernemingen op Object-georiënteerd gebied is weergegeven in figuur 6.4. Ook deze figuur is afkomstig uit [WES93]. In deze figuur worden ondernemingen beoordeeld op hun visie op het gebied van Object-georiënteerde technologie en de mogelijkheden om die visie uit te dragen. De door de GartnerGroup hoog ingeschatte ondernemingen zijn dus in de rechterbovenhoek te vinden.

Het verband tussen figuur 6.3 en figuur 6.4 kan worden verduidelijkt middels het volgende overzicht:

- zowel de PARTS Workbench als Smalltalk/V zijn Smalltalk omgevingen van Digitalk
- ObjectWorks is de benaming voor zowel de Smalltalk als de C++ omgeving van ParcPlace



Figuur 6.3: Een overzicht van de belangrijkste hulpmiddelen bij de systeemontwikkeling. Deze figuur is ontleend aan [WES93].



Figuur 6.4: De marktleiders op het gebied van hulpmiddelen met betrekking tot Object-georiënteerde systeemontwikkeling. Deze figuur is ontleend aan [WES93].

- Visual Basic en Visual C++ zijn produkten van Microsoft
- Borland C++ is, evenals het niet in figuur 6.3 opgenomen Borland Pascal, een produkt van Borland
- Enfin, een produkt van het in figuur 6.4 ontbrekende Easel, is een Smalltalk omgeving
- PowerHouse is een produkt van Cognos
- van de overige produkten is ons op dit moment de aanbieder niet bekend
- niet opgenomen in figuur 6.3 zijn de C++ produkten van Lucid, CenterLine, IBM, Hewlett-Packard, Apple en Sun

Opvallende afwezige in figuur 6.4 is Texas Instruments. Uit [SD93] blijkt, dat deze organisatie wel werkt aan een Object-georiënteerde versie van het Information Engineering tool (Information Engineering \with Objects, ofwel IE\O).

6.4 Evaluatie van ontwikkelmethoden met behulp van het ISA raamwerk

Op dit moment wordt binnen Rabobank Nederland aanbevolen bij de evaluatie van ontwikkelmethoden en CASE tools gebruik te maken van een vereenvoudigde vorm van het door Sowa en Zachman [SZ92] geïntroduceerde *Information Systems Architecture* (ISA) raamwerk. Binnen Rabobank Nederland wordt dit

raamwerk meestal aangeduid als het *Sowa/Zachman schema*. Wij gebruiken in deze scriptie de officiële benaming.

Naar onze mening mist het ISA raamwerk een aantal concepten, waardoor het onmogelijk is Object-georienteerde ontwikkelmethoden in het schema te plaatsen. Wij stellen daarom een uitbreiding van het raamwerk voor, die een dergelijke plaatsing mogelijk maakt. Voordat wij deze uitbreiding beschrijven, geven wij voor de duidelijkheid eerst een overzicht van het ISA raamwerk.

6.4.1 Het ISA raamwerk

Met behulp van het ISA raamwerk kunnen de verschillende concepten, die tijdens het ontwikkelen van informatiesystemen worden gebruikt om verschillende aspecten van de werkelijke situatie en van het informatiesysteem te beschrijven, aan elkaar worden gerelateerd. Het ISA raamwerk is daartoe onderverdeeld in dertig cellen, gerangschikt in een matrix van zes kolommen en vijf rijen. Een grafische weergave van het ISA raamwerk is te vinden in figuur 6.5.

Zoals uit de figuur kan worden opgemaakt, representeert iedere rij het model, dat ontstaat door vanuit een bepaald abstractieniveau naar het te beschouwen informatiesysteem te kijken. De verschillende abstractieniveaus zijn:

1. het *bereik* van de onderneming
2. het *ondernemingsmodel*
3. het *systeemmodel*
4. het *technologiemodel*
5. de *componenten*

De zes kolommen stellen de verschillende perspectieven voor, van waaruit het informatiesysteem kan worden beschouwd. Iedere kolom geeft antwoord op een vraag, die over het systeem kan worden gesteld:

1. de *data (wat?)* kolom
2. de *functie (hoe?)* kolom
3. de *netwerk (waar?)* kolom
4. de *personen (wie?)* kolom
5. de *tijd (wanneer?)* kolom
6. de *motivatie (waarom?)* kolom

Binnen het ISA raamwerk gelden de volgende regels:

Regel 1. De kolommen hebben geen onderlinge volgorde.

Regel 2. Iedere kolom heeft een eenvoudig basismodel.

Regel 3. Het basismodel van iedere kolom moet uniek zijn.

	DATA Entity Relation	**FUNCTION** Function Argument	**NETWORK** Node Link	**PEOPLE** Agent Work	**TIME** Time Cycle	**MOTIVATION** Ends Means
SCOPE Planner	**List of things important to the business** E = Class of Business thing	**List of processes the business performs** F = Class of Business Process	**List of locations in which the business operates** N = Major Business Location	**List of organizations/agents important to the business** A = Major Organization Unit	**List of events significant to the business** T = Major Business Event	**List of business goals/strategy** E/M = Major Business Goal/ Critical Success Factor
ENTERPRISE MODEL Owner	**E.G., Entity/ Relationship Model** E = Business Entity R = Business Constraint	**E.G., Process Flow Diagram** F = Business Process A = Business Resources	**E.G. Logistics Network** N = Business Location L = Business Linkage	**E.G., Organization chart** A = Orginization Unit W = Work Product	**E.G., Master Schedule** T = Business Event C = Business Cycle	**E.G., Business Plan** E = Business Objective M = Business Strategy
SYSTEM MODEL Designer	**E.G., Data Model** E = Data Entity R = Data Relationship	**E.G., Data Flow Diagram** F = Application Function A = User View	**E.G., Distributed System Architecture** N = I/S Function (Storage, etc.) L = Line characteristics	**E.G., Human Interface Architecture** A = Role W = Deliverable	**E.G., Processing Structure** T = System Event C = Processing Cycle	**E.G., Knowledge Architecture** E = Criterion M = Option
TECHNOLOGY MODEL Builder	**E.G., Data Design** E = Segment/ Row R = Pointer/Key	**E.G., Structure Chart** F = Computer Function A = Screen/ Device Format	**E.G., System Architecture** N = Hardware/ System Software L = Line Specifications	**E.G., Human/ Technology Interface** A = User W = Job	**E.G., Control Structure** T = Execute C = Component Cycle	**E.G., Knowledge Decision** E = Condition M = Action
COMPONENTS Subcontractor	**E.G., Data Definition Description** E = Field R = Address	**E.G., Program** F = Language Statement A = Control Block	**E.G., Network Architecture** N = Address L = Protocol	**E.G., Security Architecture** A = Identity W = Transaction	**E.G., Timing Definition** T = Interrupt C = Machine Cycle	**E.G., Knowledge Definition** E = Subcondition M = Step
FUNCTIONING SYSTEM	**E.G., Data**	**E.G., Function**	**E.G. Network**	**E.G., Organization**	**E.G., Schedule**	**E.G. Strategy**

Figuur 6.5: Het volledige ISA raamwerk, ook wel Sowa/Zachman schema genoemd [SZ92].

Regel 4. Iedere rij representeert een uniek perspectief.

Regel 5. Iedere cel is uniek.

Regel 6. De verzameling van de modellen in alle cellen van een rij vormt een compleet model van het perspectief van die rij.

Regel 7. De logica van het raamwerk is recursief.

De eerste zes regels spreken voor zich. De zevende regel stelt, dat met het raamwerk niet alleen de ontwikkeling van informatiesystemen kan worden beschreven, maar dat het kan worden toegepast op bijna alle systemen, die een eigenaar, een ontwerper en een bouwer hebben. Zo kan het raamwerk worden gebruikt om met behulp van een informatiesysteem een produktiebedrijf te modelleren, maar ook om het produktieproces in dat produktiebedrijf te modelleren, of om de ontwikkeling van het CASE tool, waarmee het betreffende informatiesysteem wordt ontwikkeld, te modelleren.

Om het ISA raamwerk beter toepasbaar te maken, wordt binnen Rabobank Nederland gebruik gemaakt van een beperkte vorm van het raamwerk, waarbinnen de netwerk, personen, tijd en motivatie kolom worden ondergebracht in één nieuwe kolom, de *controle* kolom.

6.4.2 Een aan Object-Oriëntatie aangepaste versie van het ISA raamwerk

Als wij in plaats van conventionele systeemontwikkeling uitgaan van Object-georiënteerde systeemontwikkeling, dan is het in de vorige paragraaf geschetste raamwerk maar ten dele bruikbaar. Met dat raamwerk kan de interne opbouw van de verschillende objectklassen worden beschreven, maar zowel de statische als de dynamische verbanden tussen de objectklassen onderling kunnen er niet in worden ondergebracht. Daarom stellen wij voor de binnen de Rabobank gebruikte beperking van het ISA raamwerk onder te brengen in een uitgebreider schema, waarin deze verbanden wel zichtbaar kunnen worden gemaakt. Dit uitgebreide raamwerk geven wij de naam OO ISA raamwerk.

De rijen van het OO ISA raamwerk blijven gelijk aan de rijen in het originele ISA raamwerk, maar in

	Statische Externe Object-structuur Object Statische verbinding	**Dynamische Externe Object-structuur** Object Dynamische verbinding	**Interne Objectstructuur** Object		
			Data Entiteit Relatie	**Functie** Functie Argument	**Controle** Event Trigger
BEREIK					
ONDERNEMINGS-MODEL					
SYSTEEM MODEL					
TECHNOLOGIE MODEL					
COMPONENTEN					

Figuur 6.6: Een schematische weergave van het OO ISA raamwerk.

plaats van de originele indeling in kolommen gaan wij uit van de volgende indeling:

1. de *statische externe objectstructuur* kolom
2. de *dynamische externe objectstructuur* kolom
3. de *interne objectstructuur* kolom, die uiteenvalt in de volgende drie subkolommen:
   a. de *data* subkolom
   b. de *functie* subkolom
   c. de *controle* subkolom

Een schematische weergave hiervan is te vinden in figuur 6.6.

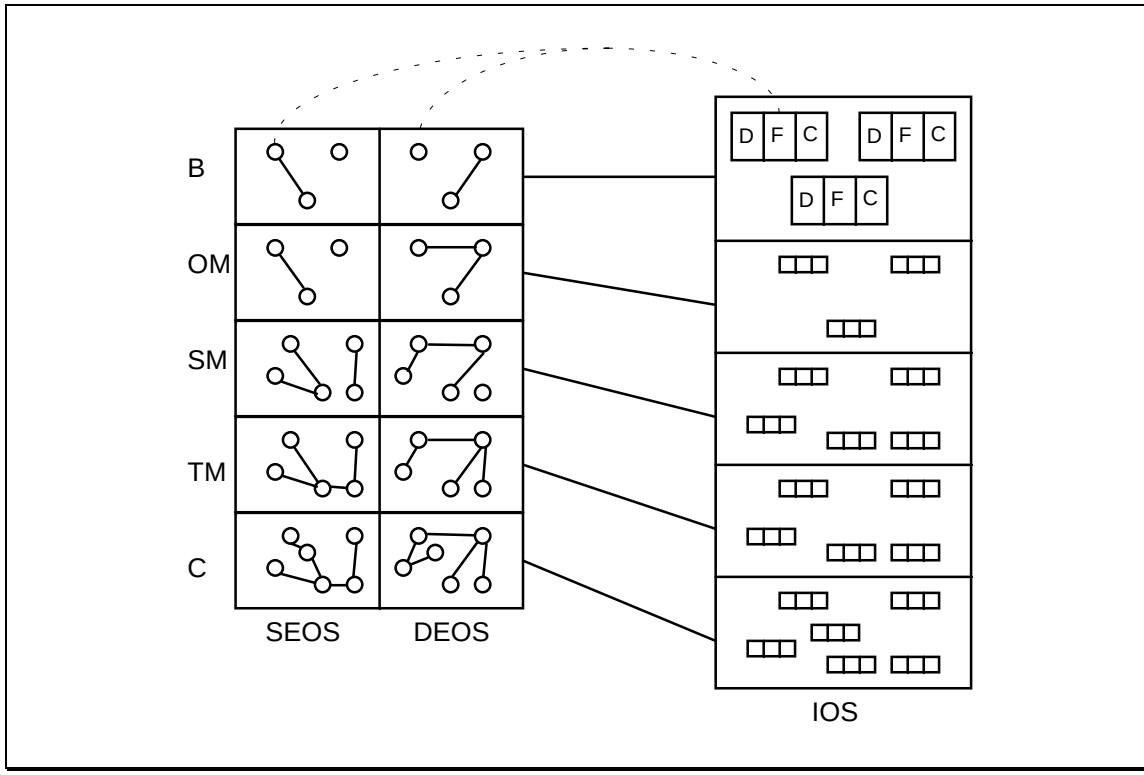
De statische externe objectstructuur kolom beschrijft de statische verbanden tussen de verschillende objectklassen, te weten instantie associaties, generalisaties en aggregaties, alsmede de attributen en de operaties, die de verschillende objectklassen kennen. Dit kan bijvoorbeeld in objectmodellen worden gemodelleerd. In de statische externe objectstructuur kolom vormen de objectklassen de entiteiten en de statische verbanden tussen de objectklassen de connectoren.

De dynamische externe objectstructuur kolom beschrijft de dynamische verbanden tussen de verschillende objectklassen. Het betreft hier de boodschappen, die de ene objectklasse naar de andere kan sturen. Ook deze verbanden worden in de meeste methoden in het objectmodel gemodelleerd. Wij hebben daar geen bezwaar tegen, zolang duidelijk blijft welke delen van het objectmodel de statische en welke de dynamische verbanden beschrijven. Een dergelijk objectmodel moet te allen tijde probleemloos kunnen worden gesplitst in twee verschillende modellen, die respectievelijk de dynamische en de statische externe objectstructuur beschrijven. In de dynamische externe objectstructuur kolom vormen de objectklassen de entiteiten en de communicatie associaties de connectoren.

Iedere objectklasse vertoont gedrag. Als dat niet het geval was, dan zouden tussen de betreffende objectklasse en de andere objectklassen in het systeem geen dynamische verbanden bestaan en zouden de objecten, die instanties waren van de betreffende objectklasse, geen rol spelen in het systeem en net zo goed kunnen worden weggelaten. Als wij het ontbreken van statische verbanden als een vorm van structuur beschouwen, dan heeft daarnaast iedere objectklasse een statische objectstructuur. Iedere objectklasse heeft dus zowel een dynamische als een statische objectstructuur. De verzameling objectklassen in de statische externe objectstructuur kolom is dus gelijk aan de verzameling objectklassen in de dynamische externe objectstructuur kolom. Hierdoor is regel 3 (het basismodel van iedere kolom moet uniek zijn) voor de externe objectstructuur kolommen niet van toepassing. Voor deze kolommen geldt de volgende regel:

Regel 3a. De entiteiten in de externe objectstructuur kolommen zijn gelijk. In deze kolommen is objectklasse de entiteit. Het verschil tussen de twee kolommen wordt gevormd door de connectoren. Voor iedere in deze kolommen voorkomende objectklasse moeten in de interne objectstructuur kolom bijbehorende data, functie en controle cellen voorkomen.

De interne objectstructuur kolom is te vergelijken met de binnen Rabobank Nederland gebruikte beperking van het ISA raamwerk. Het beschrijft **voor iedere objectklasse** in de externe objectstructuur kolommen de interne opbouw. Hierbij kan onderscheid worden gemaakt tussen drie verschillende perspectieven: data, functies en controle. In feite is er dan ook geen sprake van één interne objectstructuur kolom, maar van een per rij variërend aantal parallel voorkomende interne objectstructuur cellen, welk aantal gelijk is aan het aantal objectklassen, dat in de externe objectstructuur kolommen in de betreffende rij is gemodelleerd. In figuur 6.7 wordt dit schematisch weergegeven.



Figuur 6.7: Goed beschouwd is in het OO ISA raamwerk geen sprake van drie gelijksoortige kolommen. De statische en de dynamische externe objectstructuur kolom bieden een bepaald perspectief op het geheel van objectklassen. De interne objectstructuur kolom bevat voor iedere objectklasse, die in de beide andere kolommen op een bepaald niveau is opgenomen, op datzelfde niveau een interne objectstructuur cel, die de opbouw van de objectklasse in kwestie beschrijft. Deze interne objectstructuur cellen kunnen worden gesplitst in een drietal subcellen, die elk hun eigen perspectief bieden op de opbouw van de betreffende objectklasse. De afkortingen in de figuur verwijzen naar de beginletters van de verschillende kolommen en rijen.

Bij het modelleren van de drie verschillende perspectieven kan eventueel gebruik worden gemaakt van conventionele modelleringstechnieken. Binnen de interne objectstructuur kolom zijn alle zeven regels, die gelden binnen het ISA raamwerk, van kracht; ook regel 3:

Regel 3b. De basismodellen van de drie subkolommen in de interne objectstructuur kolom moeten uniek zijn.

Voor een precieze beschrijving van deze regels en voor een precieze formulering van de verbanden tussen de entiteiten en de connectoren in de verschillende kolommen is een meer formele basis van de verschillende modellen vereist. In Deel III wordt een dergelijke formele basis van Object-georiënteerde begrippen beschreven.

Deel III

Een formalisatie van Object-Oriëntatie

7 Formalisatie van de kernbegrippen

7.1 Inleiding

In de vorige delen heeft de nadruk gelegen op de toepassing van Object-georiënteerde technologie in de praktijk. Wij hebben beschreven, wat Object-Oriëntatie betekent en welke begrippen daarbij een rol spelen, wij hebben een overzicht gegeven van de hulpmiddelen, die het Object-georiënteerd ontwikkelen van systemen kunnen ondersteunen, en wij hebben gekeken naar de invoering en toepassing van Object-georienteerde technologie in ondernemingen.

In die delen is op enkele plaatsen naar voren gekomen, dat een goede formele basis van Object-Oriëntatie ontbreekt. Daar de aanwezigheid van een dergelijke basis grote invloed kan hebben op het succes, dat met Object-Oriëntatie wordt behaald, wordt in dit deel een eerste aanzet gegeven tot het tot stand komen van een formele basis.

In dit deel wordt een gedeeltelijke formalisatie van de Object-georiënteerde benadering gepresenteerd. Wij beginnen met een formalisatie van de in sectie 1.4 (Kernbegrippen van Object-Oriëntatie) voorgestelde kernbegrippen. In dit hoofdstuk wordt een formele syntax voor objectstructuren gepresenteerd, waarbij wij met de term objectstructuren doelen op Object Modellen zonder grafische constraints. In het volgende hoofdstuk wordt gekeken naar populaties van dergelijke objectstructuren, naar voorwaarden, die aan de populatie van een objectstructuur kunnen worden gesteld, en naar de wijze, waarop de leden van een populatie kunnen worden geïdentificeerd.

De in dit deel in de voorbeelden gebruikte notatie is voor wat de Object-georiënteerde concepten betreft gebaseerd op de door Coad en Yourdon voorgestelde notatie (zie [CY91a]), maar modelleert instantie associaties en de daarvoor geldende constraints met behulp van de op NIAM gebaseerde PSM notatie (zie [HOF93]). De keuze voor Coad en Yourdon is gemaakt, omdat de door hun gebruikte notatie het beste de door ons gebruikte concepten ondersteunt. In bijlage A (Verklaring van de wiskundige en grafische notaties) wordt de door ons gebruikte notatie nader verklaard.

Een verklaring van de gebruikte wiskundige notaties is eveneens te vinden in bijlage A (Verklaring van de wiskundige en grafische notaties).

7.2 Objectstructuren

Een objectstructuur van een Object Model bestaat uit de hieronder genoemde basiscomponenten.

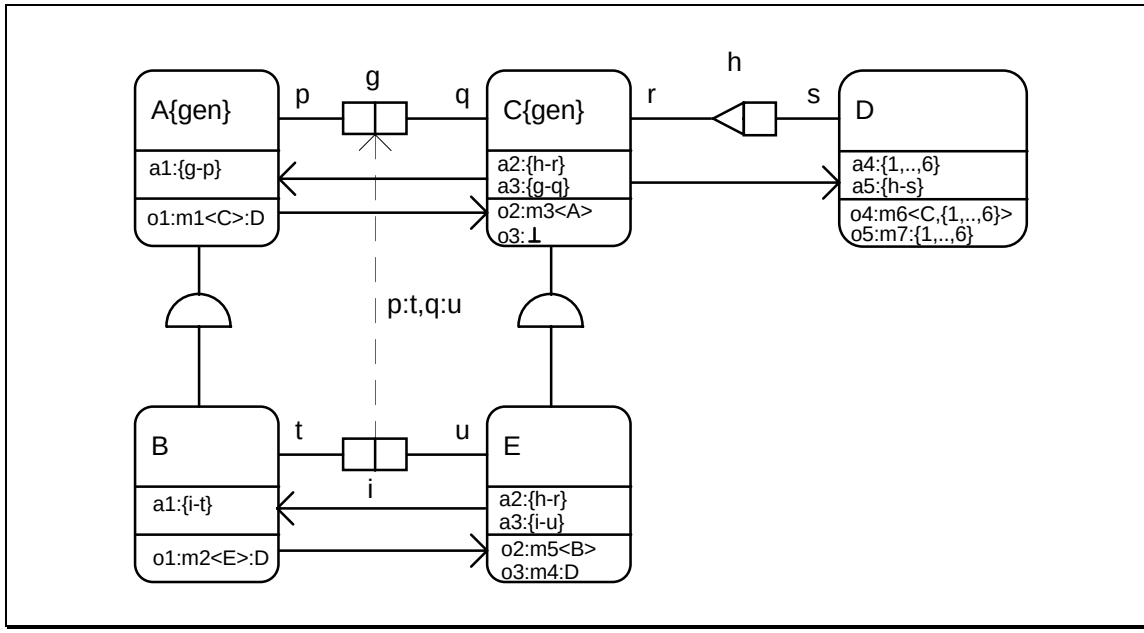
1. Een niet-lege eindige verzameling $\mathcal{K}$ van objectklassen.
2. Een verzameling $\mathcal{AK}$ van abstracte objectklassen. Abstracte objectklassen zijn objectklassen: $\mathcal{AK} \subseteq \mathcal{K}$.
3. Een verzameling $\mathcal{CK}$ van concrete objectklassen. Ook concrete objectklassen zijn objectklassen: $\mathcal{CK} \subseteq \mathcal{K}$.
4. Een verzameling $\mathcal{GK}$ van generalisatie objectklassen. Deze verzameling is een echte deelverzameling van de verzameling objectklassen: $\mathcal{GK} \subset \mathcal{K}$.
5. Een niet-lege eindige verzameling $\mathcal{P}$ van predicatoren.
6. Een partitie $\mathcal{F}$ van de verzameling $\mathcal{P}$. De elementen van $\mathcal{F}$ worden instantie associaties genoemd.

7. Een verzameling $\mathcal{AG}$ van aggregaties. Aggregaties zijn instantie associaties: $\mathcal{AG} \subseteq \mathcal{F}$.
8. Een niet-lege verzameling $\mathcal{A}$ van attributen.
9. Een niet-lege verzameling $\mathcal{O}$ van operaties.
10. Een niet-lege verzameling $\mathcal{M}$ van methoden.
11. Een meersoortige algebra $\mathcal{AL} = \langle D, F \rangle$, waarbij D een verzameling concrete domeinen is (bijvoorbeeld karakter, string, geheel getal) en F een verzameling operaties (bijvoorbeeld +).
12. Een functie **Base:** $\mathcal{P} \to \mathcal{K}$. De basis van een predicator is de objectklasse, waarmee de predicator is geassocieerd.
13. Een functie **Aggr:** $\mathcal{AG} \to \mathcal{P}$. Deze functie levert de predicator, die een gegeven aggregatie verbindt met de objectklasse, die in die aggregatie de rol van aggregaat speelt.
14. Een relatie **Prop** $\subseteq \mathcal{A} \times \mathcal{K}$, die het verband beschrijft tussen attributen en de objectklassen, waartoe deze attributen behoren.
15. Een functie **Dom:Prop** $\to D \cup \left(\wp(\mathcal{F}) \setminus \{\varnothing\}\right)$. Deze functie levert het domein van een gegeven attribuut in een gegeven objectklasse.
16. Een partiële relatie **Pred:Prop** $\mapsto \left(\wp(\mathcal{P}) \setminus \{\varnothing\}\right)$. Deze functie levert de predicatoren, die de instantie associaties uit het domein van een gegeven attribuut in een gegeven objectklasse verbinden met generalisaties van die objectklasse of die objectklasse zelf. **Pred**$(a,k)\downarrow$ als **Dom**$(a,k) \subseteq \mathcal{F}$.
17. Een relatie **Serv** $\subseteq \mathcal{O} \times \mathcal{K}$, die het verband beschrijft tussen operaties en de objectklassen, waartoe deze operaties behoren.
18. Een functie **Method:Serv** $\to \mathcal{M}$, die de methode levert, waardoor een gegeven operatie in een gegeven objectklasse wordt geïmplementeerd.
19. Een partiële functie **Parms:** $\mathcal{M} \mapsto (\mathcal{K} \cup D)^+$. Deze functie levert de reeks objectklassen en concrete domeinen, die fungeert als parameterlijst van de gegeven methode.
20. Een partiële functie **Res:** $\mathcal{M} \mapsto \mathcal{K} \cup D$. Deze functie levert de objectklasse of het concrete domein, waarvan respectievelijk de object instanties en de elementen als resultaat kunnen fungeren van de gegeven methode.
21. Een binaire relatie **Client** $\subseteq \mathcal{K} \times \mathcal{K}$ op objectklassen, die communicatie associaties beschrijft.
22. Een binaire relatie **Gen** $\subset \mathcal{K} \times \mathcal{K}$ op objectklassen, die generalisatiestructuren beschrijft.
23. Een binaire relatie **Spec** $\subset \mathcal{F} \times \mathcal{F}$ op instantie associaties, die specialisatie beschrijft.

Voorbeeld 7.2.1

Figuur 7.1 is de grafische weergave van een objectstructuur, die wordt gedefinieerd door

$\mathcal{K} = \{A,B,C,D,E\}$ $\quad$ $\mathcal{F} = \{g,h,i\}$
$\mathcal{AK} = \{A,B,C,E\}$ $\quad$ $\mathcal{AG} = \{h\}$
$\mathcal{CK} = \{D\}$ $\quad$ $\mathcal{A} = \{a1,a2,a3,a4,a5\}$
$\mathcal{GK} = \{A,C\}$ $\quad$ $\mathcal{O} = \{o1,o2,o3,o4,o5\}$
$\mathcal{P} = \{p,q,r,s,t,u\}$ $\quad$ $\mathcal{M} = \{\perp,m1,m2,m3,m4,m5,m6,m7\}$



Figuur 7.1: Een voorbeeld van een objectstructuur. De bij de instantie associaties in een domein behorende predicatoren en de parameterlijsten en resultaten van methoden zijn hier voor de volledigheid in het algemene Object Model opgenomen. Meestal worden zij niet in het Object Model zelf, maar in bij het Object Model behorende klassebeschrijvingen, die grafisch of tekstueel kunnen zijn, gemodelleerd.

met $g = \{p,q\}$, $h = \{r,s\}$ *en* $i = \{t,u\}$. *Verder geldt*

$\mathbf{Base}(p) = A$ $\quad$ $\mathbf{Base}(r) = C$ $\quad$ $\mathbf{Base}(t) = B$
$\mathbf{Base}(q) = C$ $\quad$ $\mathbf{Base}(s) = D$ $\quad$ $\mathbf{Base}(u) = E$

$\mathbf{Aggr}(h) = r$

$a1\,\mathbf{Prop}\,A$ $\quad$ $a2\,\mathbf{Prop}\,C$ $\quad$ $a3\,\mathbf{Prop}\,C$ $\quad$ $a4\,\mathbf{Prop}\,D$
$a1\,\mathbf{Prop}\,B$ $\quad$ $a2\,\mathbf{Prop}\,E$ $\quad$ $a3\,\mathbf{Prop}\,E$ $\quad$ $a5\,\mathbf{Prop}\,D$

$\mathbf{Dom}(a1,A) = \{g\}$ $\quad$ $\mathbf{Dom}(a2,E) = \{h\}$ $\quad$ $\mathbf{Dom}(a4,D) = \{1,..,6\}$
$\mathbf{Dom}(a1,B) = \{i\}$ $\quad$ $\mathbf{Dom}(a3,C) = \{g\}$ $\quad$ $\mathbf{Dom}(a5,D) = \{h\}$
$\mathbf{Dom}(a2,C) = \{h\}$ $\quad$ $\mathbf{Dom}(a3,E) = \{i\}$

$\mathbf{Pred}(a1,A) = \{p\}$ $\quad$ $\mathbf{Pred}(a2,E) = \{r\}$ $\quad$ $\mathbf{Pred}(a5,D) = \{s\}$
$\mathbf{Pred}(a1,B) = \{t\}$ $\quad$ $\mathbf{Pred}(a3,C) = \{q\}$
$\mathbf{Pred}(a2,C) = \{r\}$ $\quad$ $\mathbf{Pred}(a3,E) = \{u\}$

$o1\,\mathbf{Serv}\,A$ $\quad$ $o2\,\mathbf{Serv}\,C$ $\quad$ $o3\,\mathbf{Serv}\,C$ $\quad$ $o4\,\mathbf{Serv}\,D$

$o1$ **Serv** B $\quad$ $o2$ **Serv** E $\quad$ $o3$ **Serv** E $\quad$ $o5$ **Serv** D

Method$(o1,A) = m1$ $\quad$ **Method**$(o2,E) = m5$ $\quad$ **Method**$(o4,D) = m6$
Method$(o1,B) = m2$ $\quad$ **Method**$(o3,C) = \perp$ $\quad$ **Method**$(o5,D) = m7$
Method$(o2,C) = m3$ $\quad$ **Method**$(o3,E) = m4$

Parms$(m1) =< C >$ $\quad$ **Parms**$(m3) =< A >$ $\quad$ **Parms**$(m6) =< C,\{1,..,6\} >$
Parms$(m2) =< E >$ $\quad$ **Parms**$(m5) =< B >$

Res$(m1) = D$ $\quad$ **Res**$(m2) = D$ $\quad$ **Res**$(m4) = D$ $\quad$ **Res**$(m7) = \{1,..,6\}$

A **Client** C $\quad$ C **Client** A $\quad$ E **Client** B
B **Client** E $\quad$ C **Client** D

A **Gen** B $\quad$ C **Gen** E

i **Spec** g met $\phi(p) = t, \phi(q) = u$

7.2.1 Objectklassen

In paragraaf 1.4.2 (Klasse) is een klasse gedefinieerd als "een beschrijving van een groep objecten met gelijke eigenschappen, hetzelfde gedrag, dezelfde relaties en dezelfde semantiek". De eindige verzameling $\mathfrak{K}$ is de verzameling van alle objectklassen. Iedere objectklasse is een al dan niet lege verzameling object instanties met vergelijkbare eigenschappen en hetzelfde gedrag.

Vanwege de verschillende interpretaties, die worden gegeven aan abstracte objectklassen en concrete objectklassen, worden zij als verschillend beschouwd. Bovendien is iedere objectklasse of abstract of concreet.

[OM1] $\mathcal{A}\mathfrak{K} \cap \mathcal{C}\mathfrak{K} = \varnothing$

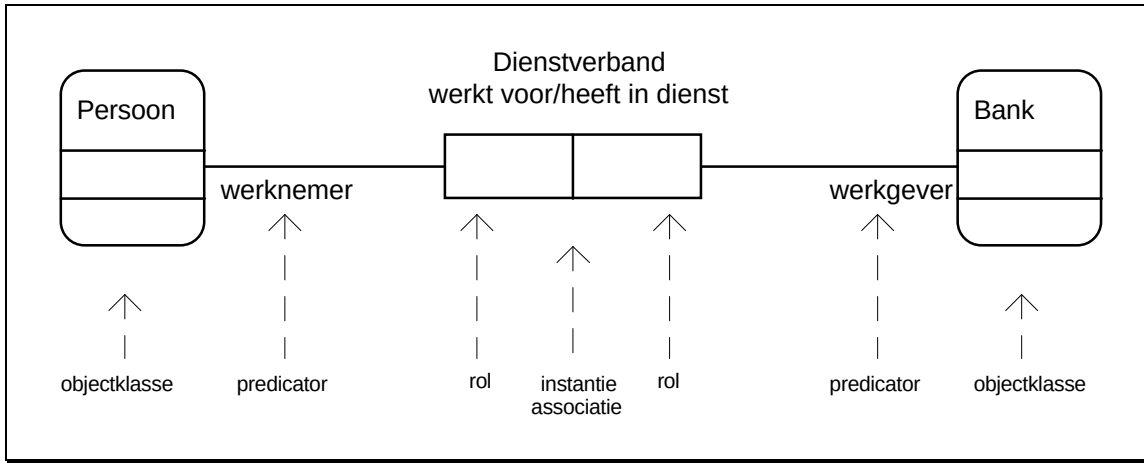
[OM2] $\mathcal{A}\mathfrak{K} \cup \mathcal{C}\mathfrak{K} = \mathfrak{K}$

De verzameling generalisatie objectklassen is een deelverzameling van de verzameling objectklassen. Generalisatie objectklassen dienen ter bevordering van de herbruikbaarheid van een objectstructuur en mogen niet worden gepopuleerd (zie paragraaf 8.2.1, Populaties van objectklassen). Generalisatie objectklassen kunnen zowel abstract als concreet zijn.

Overigens worden de objectklassen, die in onze benadering generalisatie objectklassen worden genoemd, door anderen (bijvoorbeeld [RBP+91]) abstracte objectklassen genoemd. De term concrete objectklassen wordt door deze auteurs gebruikt voor niet-abstracte objectklassen. Het door ons gemaakte onderscheid tussen de objectklassen, die een concreet domein representeren, en de objectklassen, die geen concreet domein representeren, wordt door deze auteurs niet gemaakt.

7.2.2 Instantie Associatie

In paragraaf 1.4.6 (Relaties tussen objecten) is opgemerkt, dat er twee verschillende soorten relaties tussen objectklassen bestaan, te weten statische relaties en dynamische relaties. In dit deel wordt met een instantie associatie op een statische relatie tussen objectklassen gedoeld en met een communicatie associatie op



Figuur 7.2: Een voorbeeld van de hier gevolgde benadering van een instantie associatie, ontleend aan [BRO93].

een dynamische relatie. In deze paragraaf gaan wij in op instantie associaties, terwijl communicatie associaties in de volgende paragraaf aan de orde komen.

Evenals [BRO93] gaan wij bij de formalisatie van het Object Model uit van de *mapping-oriented benadering*. In de mapping-oriented benadering wordt een instantie associatie opgesplitst in een aantal *rollen*. De connecties tussen de rollen en de objectklassen, waarmee de rollen zijn geassocieerd, worden *predicatoren* genoemd. Een instantie associatie kan in dat geval worden gezien als een verzameling predicatoren [HOF93]. Een voorbeeld van de manier, waarop wij instantie associaties benaderen, is te vinden in voorbeeld 7.2.2.

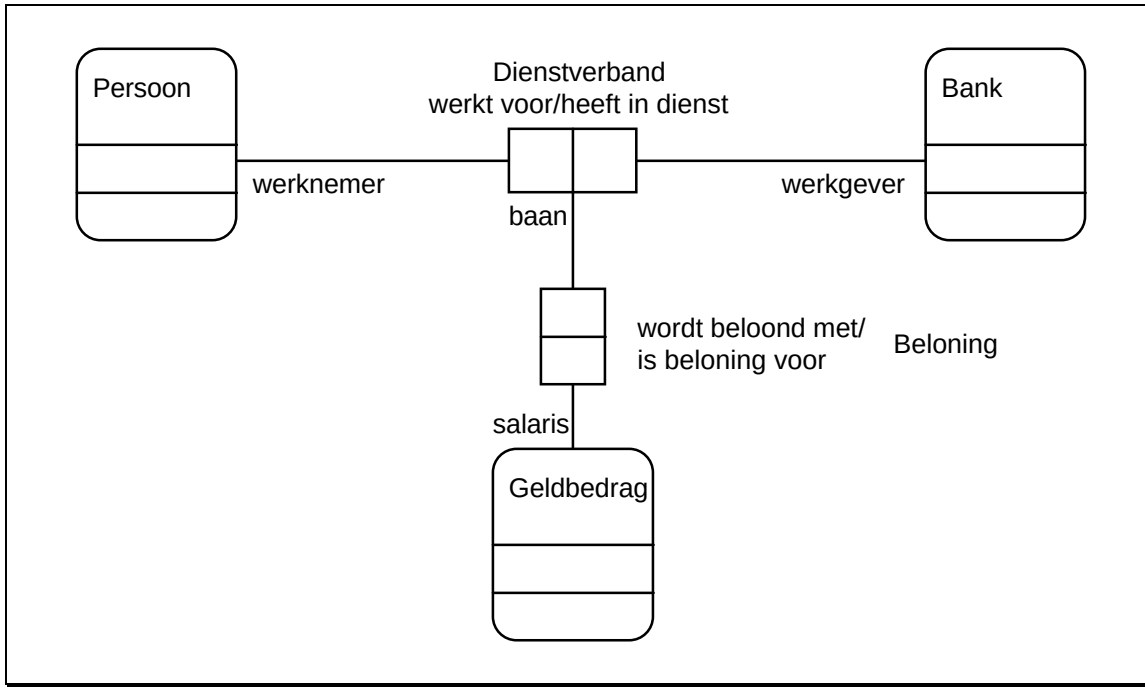
Voorbeeld 7.2.2

In figuur 7.2 is het dienstverband van een werknemer, die werkt voor een bank, gemodelleerd. De instantie associatie Dienstverband legt een relatie tussen de objectklasse Persoon en de objectklasse Bank. De rol 'werkt voor' associeert een object instantie, behorend tot de objectklasse Persoon, met de bank(en), waar hij/zij voor werkt, terwijl de rol 'heeft in dienst' een object instantie, behorend tot de objectklasse Bank, associeert met de persoon/personen, die zij in dienst heeft. De predicator 'werknemer' verbindt Persoon met Dienstverband en de predicator 'werkgever' verbindt Bank met Dienstverband.

In [BRO93] kan een instantie associatie zelf weer geassocieerd zijn met objectklassen. Wij staan dit niet toe, omdat anders op basis van onze definities het localiteitsprincipe (zie paragraaf 1.5.1, Inkapseling van proces en data) wordt geschonden. Dit wordt verduidelijkt in voorbeeld 7.2.3.

Voorbeeld 7.2.3

Figuur 7.3 is een voorbeeld van een geobjectificeerde instantie associatie: een instantie associatie, die een objectklasse met een andere instantie associatie associeert. Een Dienstverband is een instantie associatie tussen een Persoon en een Bank. Dit Dienstverband wordt beloond door middel van een Geldbedrag, dat als salaris door Bank aan Persoon wordt gegeven. Als een werknemer een salarisverhoging krijgt, dan moet de associatie Beloning worden gewijzigd. Volgens het localiteitsprincipe zijn alle gegevens ingekapseld en kunnen zij alleen worden benaderd via bijbeho-



Figuur 7.3: Een voorbeeld van een geobjectificeerde instantie associatie.

rende operaties. De operatie Salarisverhoging moet dus op de één of andere manier in de instantie associatie Dienstverband worden ondergebracht. Maar daarmee wordt Dienstverband een inkapseling van data en functies en dus een objectklasse. Geobjectificeerde instantie associaties zijn daarom objectklassen en dienen dan ook als zodanig gemodelleerd te worden.

De functie **Base** levert bij een gegeven predicator de objectklasse, die door die predicator wordt verbonden met een instantie associatie. De hulpfunctie **Fact:** $\mathcal{P} \to \mathcal{I}$ levert bij een gegeven predicator de instantie associatie, die door die predicator wordt verbonden met een objectklasse, en is gedefinieerd als $\mathbf{Fact}(p) = i \Leftrightarrow p \in i$.

Een speciale soort instantie associaties, aggregaties, wordt beschreven in paragraaf 7.2.4 (Aggregatie).

7.2.3 Communicatie associaties

Communicatie associaties beschrijven de dynamische verbanden tussen objectklassen. Een communicatie instantie geeft aan, dat een instantie van de ene objectklasse gebruik mag maken van de diensten van een instantie van de andere objectklasse, ofwel dat de instanties van de ene objectklasse klant zijn van de instanties van de andere objectklasse. Een communicatie associatie is daarom geen verzameling predicatoren, maar een geordend paar (dienstvrager, dienstverlener).

Om te voorkomen, dat modellen te complex worden, mogen communicatie associaties, die de communicatie tussen een abstracte en een concrete objectklasse beschrijven, in een model worden weggelaten. Ook communicatie associaties, waarin één objectklasse zowel dienstvrager als dienstverlener is, hoeven om die reden niet in een model opgenomen te worden.

Ieder attribuut heeft een *domein*, een verzameling waarden, die het attribuut mag aannemen. Deze waarden representeren de eigenschap, die het attribuut beschrijft. Het domein van een attribuut is ofwel een verzameling instantie associaties, ofwel een concreet domein. De functie **Dom** levert bij een gegeven attribuut en een objectklasse. waartoe dat attribuut behoort, het domein van dat attribuut.

Alleen concrete objectklassen hebben een attribuut met als domein een concreet domein, een verzameling concrete waarden. De concrete objectklasse fungeert in dat geval als inkapseling van de operaties op die concrete waarden en de concrete waarde zelf. Iedere concrete objectklasse heeft precies één attribuut met een concreet domein. Een concrete objectklasse heeft daarnaast een aantal attributen, die als domein een verzameling instantie associaties hebben.

[OM8] $k \in \mathbb{CK} \Rightarrow \exists!_{a \in \mathcal{A}} \left[a \,\mathbf{Prop}\, k \wedge \mathbf{Dom}(a,k) \in D \right]$

Ieder concreet domein moet kunnen worden bewerkt en is daarom domein van minimaal één attribuut.

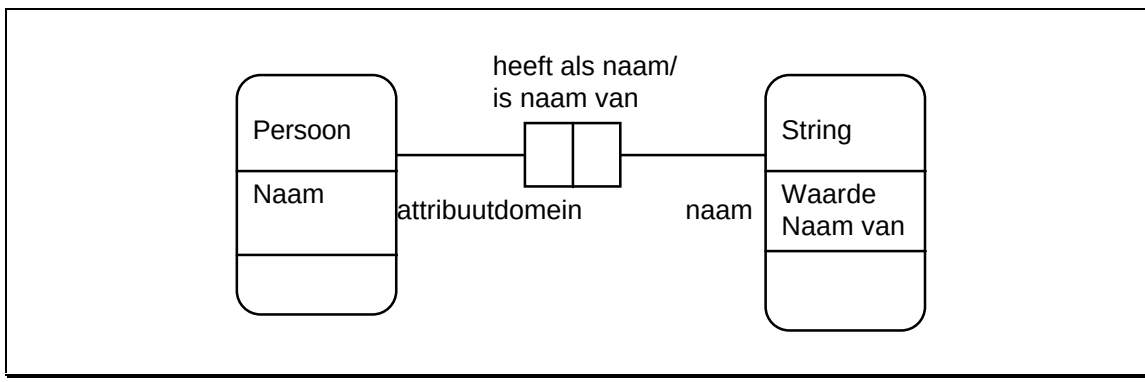
[OM9] $d \in D \Rightarrow \exists_{a \in \mathcal{A}} \exists_{k \in \mathbb{K}} \left[\mathbf{Dom}(a,k) = d \right]$

Concrete domeinen zijn bijvoorbeeld Cijfer, Geheel Getal, Reëel Getal, Karakter en String.

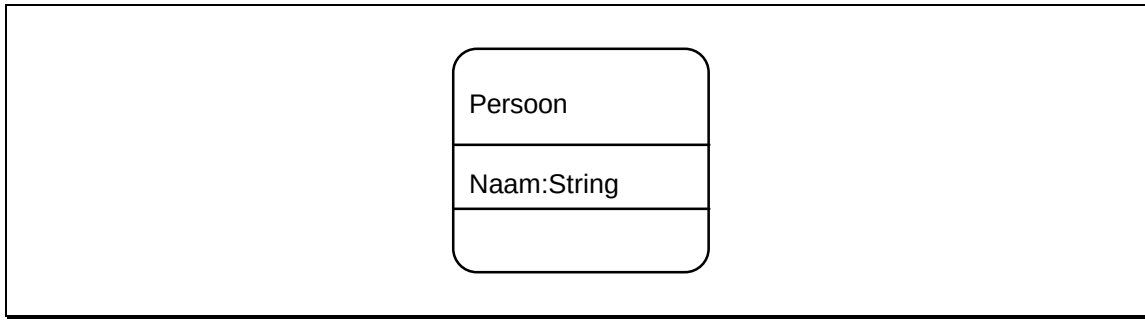
Als het domein van een attribuut een verzameling instantie associaties is, dan zijn die instantie associaties vergelijkbaar: de rollen van de ene instantie associatie zijn gelijk aan de rollen van de andere instantie associatie. In paragraaf 7.3.1 (Overerving van attributen en domeinen) wordt dit formeel beschreven. De eigenschap, die het attribuut beschrijft, wordt gerepresenteerd door de object instanties, die via die instantie associaties zijn geassocieerd. Op deze manier blijft het localiteitsprincipe geldig: wijzigen van één van de object instanties, die samen een eigenschap representeren, is alleen mogelijk via de door die object instantie toegestane operaties. Dit wordt verduidelijkt in voorbeeld 7.2.5.

Voorbeeld 7.2.5

Alle object instanties, die behoren tot de objectklasse Persoon, hebben de eigenschap 'persoon heeft naam'. Die eigenschap kan worden gerepresenteerd door de combinatie van een object instantie, behorend tot de objectklasse Persoon, en een object instantie, behorend tot de objectklasse String. Deze objectklassen moeten dus met elkaar geassocieerd zijn. In figuur 7.4 is de instantie associatie tussen de objectklasse Persoon en de objectklasse String gemodelleerd. Wijzigen van de naam van een persoon kan alleen met behulp van door de objectklasse String toegestane operaties:



Figuur 7.4: Een voorbeeld van een instantie associatie tussen een gegeven objectklasse en een primitieve objectklasse, die fungeert als domein voor één van de attributen van die gegeven objectklasse.



Figuur 7.5: Een voorbeeld van een impliciete weergave van een binaire instantie associatie tussen een abstracte objectklasse en een concrete objectklasse.

gegeven en bijbehorende functies zijn ingekapseld in één object instantie.

Voor iedere predicator van een instantie associatie is er een attribuut, dat behoort tot de basis van die predicator en dat als domein de betreffende instantie associatie heeft.

[OM10] $f \in \mathfrak{F} \wedge p \in f \Rightarrow \exists_{a \in \mathcal{A}} \exists_{k \in \mathfrak{K}} \left[f \in \mathbf{Dom}(a,k) \wedge \mathbf{Base}(p) = k \right]$

De partiële functie **Pred** beschrijft het verband tussen een verzameling predicatoren en een gegeven attribuut en een gegeven objectklasse, waartoe het attribuut behoort. De predicatoren verbinden de instantie associaties, die het domein van het gegeven attribuut voor de gegeven objectklasse vormen, met die objectklasse of met generalisaties van die objectklasse. Voor iedere instantie associatie in het domein van een attribuut is er een unieke predicator, die deze verbinding legt.

[OM11] $f \in \mathfrak{F} \wedge f \in \mathbf{Dom}(a,k) \Rightarrow \exists!_{p \in f} \left[p \in \mathbf{Pred}(a,k) \right]$

Een dergelijke verzameling predicatoren mag geen overbodige predicatoren bevatten.

[OM12] $p \in \mathbf{Pred}(a,k) \Rightarrow \mathbf{Fact}(p) \in \mathbf{Dom}(a,k)$

Om te voorkomen dat modellen te complex worden, staan wij toe, dat binaire instantie associaties tussen een abstracte objectklasse en een concrete objectklasse niet expliciet in het model worden opgenomen, maar worden gemodelleerd door achter de naam van het bijbehorende attribuut van de abstracte objectklasse de naam van de concrete objectklasse, die via de instantie associatie is verbonden met die abstracte objectklasse, te vermelden. Ook het bijbehorende attribuut van de concrete objectklasse hoeft niet in het model opgenomen te worden. Figuur 7.5 is een op deze manier vereenvoudigde versie van figuur 7.4.

Vanwege haar speciale eigenschappen moet een aggregatie altijd expliciet in het model worden opgenomen, ook als de component van de aggregatie een concrete objectklasse is.

De relatie **Spec**, die specialisatie van instantie associaties beschrijft, is geïntroduceerd met het oog op herdefiniëring van domeinen van attributen en is dus verbonden met overerving van domeinen. Wij behandelen deze relatie daarom in de aan dat onderwerp gewijde paragraaf 7.3.1 (Overerving van attributen en domeinen).

7.2.6 Operaties

De operaties van een object instantie bepalen het gedrag van die object instantie. Object instanties, die behoren tot dezelfde objectklasse, vertonen hetzelfde gedrag en hebben dus dezelfde operaties. De relatie **Serv** beschrijft welke operaties behoren tot welke objectklassen. Iedere object instantie vertoont gedrag, dus iedere objectklasse moet minimaal één operatie hebben.

[OM13] $k \in \mathfrak{K} \Rightarrow \exists_{o \in \mathbb{O}} \left[o \, \mathbf{Serv} \, k \right]$

Iedere operatie verzorgt een bepaalde functionaliteit en moet daarom behoren tot minimaal één objectklasse.

[OM14] $o \in \mathbb{O} \Rightarrow \exists_{k \in \mathfrak{K}} \left[o \, \mathbf{Serv} \, k \right]$

Een operatie kan worden uitgevoerd door executie van een methode. Op methoden gaan wij in de volgende paragraaf in.

7.2.7 Methoden

In paragraaf 1.4.5 (Methode) is een methode gedefinieerd als "de implementatie van een operatie voor een specifieke klasse". De functie **Method** levert bij een gegeven operatie en een gegeven objectklasse, waartoe de operatie behoort, de methode, die die operatie voor de gegeven objectklasse implementeert.

Een methode kan een aantal parameters hebben en kan een resultaat leveren. Een parameter is altijd een object instantie van een voorgeschreven objectklasse of een waarde uit een concreet domein. Daar de volgorde van de parameters van belang is, worden de parameters gegroepeerd in een reeks. De partiële functie **Parms** levert van een gegeven methode de bijbehorende reeks objectklassen en concrete domeinen, waartoe de parameters moeten behoren. Bij een methode zonder parameters is **Parms** niet gedefinieerd.

Een methode manipuleert één of meerdere attributen, maar kan daarnaast een resultaat leveren. Het resultaat van een methode is altijd een object instantie van een voorgeschreven objectklasse of een waarde uit een concreet domein. De partiële functie **Res** levert van een gegeven methode de objectklasse of het concreet domein, waartoe het resultaat van de methode moet behoren.

Het is toegestaan, dat de methode van een operatie voor een bepaalde klasse ongespecificeerd blijft. Wij introduceren hiervoor de lege methode $\perp$, met $\perp \in \mathcal{M}$, $\mathbf{Parms}(\perp)\uparrow$ en $\mathbf{Res}(\perp)\uparrow$. Om te voorkomen dat het gedrag van object instanties niet is gedefinieerd, omdat de operaties, die dat gedrag veroorzaken, geen implementatie kennen, is een operatie zonder methode alleen toegestaan in generalisatie objectklassen; klassen, die per definitie geen object instanties hebben.

[OM15] $\mathbf{Method}(o,k) = \perp \Rightarrow k \in \mathcal{G}\mathfrak{K}$

7.2.8 Generalisatie

De binaire relatie **Gen** beschrijft de generalisatiestructuur. Als de ene klasse een generalisatie is van de andere klasse, dan worden eigenschappen en gedrag van die ene objectklasse geërfd door de object instanties van de andere objectklasse. Generalisatiestructuren zijn acyclisch.

[OM16] *(geen cykels)* $a \, \mathbf{Gen}^+ \, b \Rightarrow \neg b \, \mathbf{Gen}^+ \, a$

7.2.4 Aggregatie

In paragraaf 1.4.8 (Aggregatie) wordt aggregatie opgevoerd als een structuurvorm. Daar wordt echter ook opgemerkt, dat bepaalde auteurs, waaronder Shlaer en Mellor [SM88] en Booch [BOO91], een aggregatie beschouwen als een instantie associatie. Ook wij beschouwen een aggregatie als een instantie associatie, maar wel als een instantie associatie met bepaalde eigenschappen. Hoewel een aantal auteurs aggregatie modelleert als een structuurvorm, waarbij meer dan twee objectklassen betrokken kunnen zijn, is aggregatie naar onze mening altijd een binaire instantie associatie. Dit wordt verduidelijkt aan de hand van voorbeeld 7.2.4.

Voorbeeld 7.2.4

In figuur 1.3 is een fiets gemodelleerd als een aggregatie van een frame en een stuur. Als het stuur versleten is en moet worden vervangen, dan heeft dat geen invloed op het frame. Er is dus niet sprake van één aggregatie, waarbij een frame en een stuur samen een fiets vormen, maar van twee aggregaties, waarbij zowel het frame als het stuur onderdelen van de fiets zijn met een bepaalde functionaliteit. Vervanging van het ene onderdeel hoeft geen invloed te hebben op de overige onderdelen van het aggregaat.

[OM3] $f \in \mathcal{AG} \Rightarrow |f| = 2$

Verder is bij aggregatie de rol, die de verschillende objectklassen spelen, bekend: één van de geassocieerde objectklassen is het geheel (het *aggregaat*) en de ander het onderdeel (de *component*). De functie **Aggr** levert de predicator, die een gegeven aggregatie verbindt met de objectklasse, die in die aggregatie de rol van aggregaat speelt.

[OM4] $\mathbf{Aggr}(f) \in f$

Met behulp van deze functie kan een operatie **Comp** worden gedefinieerd, die de predicator levert, die een gegeven aggregatie *i* verbindt met de objectklasse, die in die aggregatie de rol van component speelt, als $\mathbf{Comp}(f) \in f \wedge \mathbf{Comp}(f) \neq \mathbf{Aggr}(f)$. Een aggregatie mag nooit een object instantie, die behoort tot een concrete objectklasse, als aggregaat hebben en een object instantie, die behoort tot een abstracte objectklasse, als component.

[OM5] $\mathbf{Base}(\mathbf{Aggr}(f)) \in \mathcal{CK} \Rightarrow \mathbf{Base}(\mathbf{Comp}(f)) \in \mathcal{CK}$

7.2.5 Attributen

De eigenschappen van een object instantie kunnen worden beschreven met behulp van attributen. Object instanties, die tot dezelfde objectklasse behoren, hebben dezelfde eigenschappen en dus dezelfde attributen. De waarden van de attributen van object instanties, die behoren tot dezelfde objectklasse, hoeven natuurlijk niet gelijk te zijn.

De relatie **Prop** legt verband tussen attributen en objectklassen, waarvan de object instanties die eigenschappen hebben. Iedere object instantie wordt gekenmerkt door bepaalde eigenschappen. Iedere objectklasse moet daarom minimaal één attribuut hebben.

[OM6] $k \in \mathcal{K} \Rightarrow \exists_{a \in \mathcal{A}}[a \,\mathbf{Prop}\, k]$

Ieder attribuut beschrijft de eigenschap van de object instanties van minimaal één objectklasse.

[OM7] $a \in \mathcal{A} \Rightarrow \exists_{k \in \mathcal{K}}[a \,\mathbf{Prop}\, k]$

In dit axioma staat $\mathbf{Gen}^+$ voor de transitieve afsluiting van **Gen**, die op de gebruikelijke wijze kan worden gedefinieerd. $\mathbf{Gen}^*$ kan in dat geval worden gedefinieerd als

$$\mathbf{Gen}^* = \mathbf{Gen}^+ \cup \{\langle k,k\rangle \mid k \in \mathfrak{K}\}.$$

Met behulp van $\mathbf{Gen}^*$ kan de **Root** relatie worden gedefinieerd als

$$\mathbf{Root}(a,b) = b\,\mathbf{Gen}^*\,a \wedge \neg\exists_{k\in\mathfrak{K}}[k\,\mathbf{Gen}\,b].$$

Het volgende lemma stelt, dat iedere objectklasse minimaal één stamvader heeft.

Lemma 7.2.1 $\forall_{k\in\mathfrak{K}}\exists_{l\in\mathfrak{K}}[\mathbf{Root}(k,l)]$

Bewijs:

Daar $\mathfrak{K}$ eindig is en **[OM16]** stelt, dat generalisatiestructuren acyclisch zijn, kunnen wij een functie $\mathbf{depth}: \mathfrak{K} \to \aleph$ definiëren.

$$\mathbf{depth}(k) = \begin{cases} 0 & \text{if} \quad \neg\exists_{l\in\mathfrak{K}}[l\,\mathbf{Gen}\,k] \\ 1 + \min\limits_{l\,\mathbf{Gen}\,k} \mathbf{depth}(l) & \text{otherwise} \end{cases}$$

Het lemma kan nu worden bewezen met behulp van inductie naar de diepte van een objectklasse *k*. Als $\mathbf{depth}(k) = 0$, dan geldt $\neg\exists_{l\in\mathfrak{K}}[l\,\mathbf{Gen}\,k]$. $k = k$, dus $k\,\mathbf{Gen}^*\,k$. Dus $\mathbf{Root}(k,k)$.
Als $\mathbf{depth}(k) = n+1$, dan bestaat een objectklasse *l*, zodanig dat $l\,\mathbf{Gen}\,k$ en $\mathbf{depth}(k) = n$. Uit toepassing van de inductiehypothese volgt dan, dat een objectklasse *m* bestaat, zodanig dat $\mathbf{Root}(l,m)$. Maar dan geldt $m\,\mathbf{Gen}^*\,l$ en $\neg\exists_{n\in\mathfrak{K}}[n\,\mathbf{Gen}\,m]$. Uit de transitiviteit van $\mathbf{Gen}^*$ volgt dan $m\,\mathbf{Gen}^*\,k$ en dus $\mathbf{Root}(k,m)$. Hieruit volgt $\forall_{k\in\mathfrak{K}}\exists_{l\in\mathfrak{K}}[\mathbf{Root}(k,l)]$. •

Een objectklasse kan meerdere stamvaders hebben.

Een generalisatie objectklasse is altijd een generalisatie van één of meer andere objectklassen.

[OM17] $k \in \mathcal{GK} \Rightarrow \exists_{l\in\mathfrak{K}}[k\,\mathbf{Gen}\,l]$

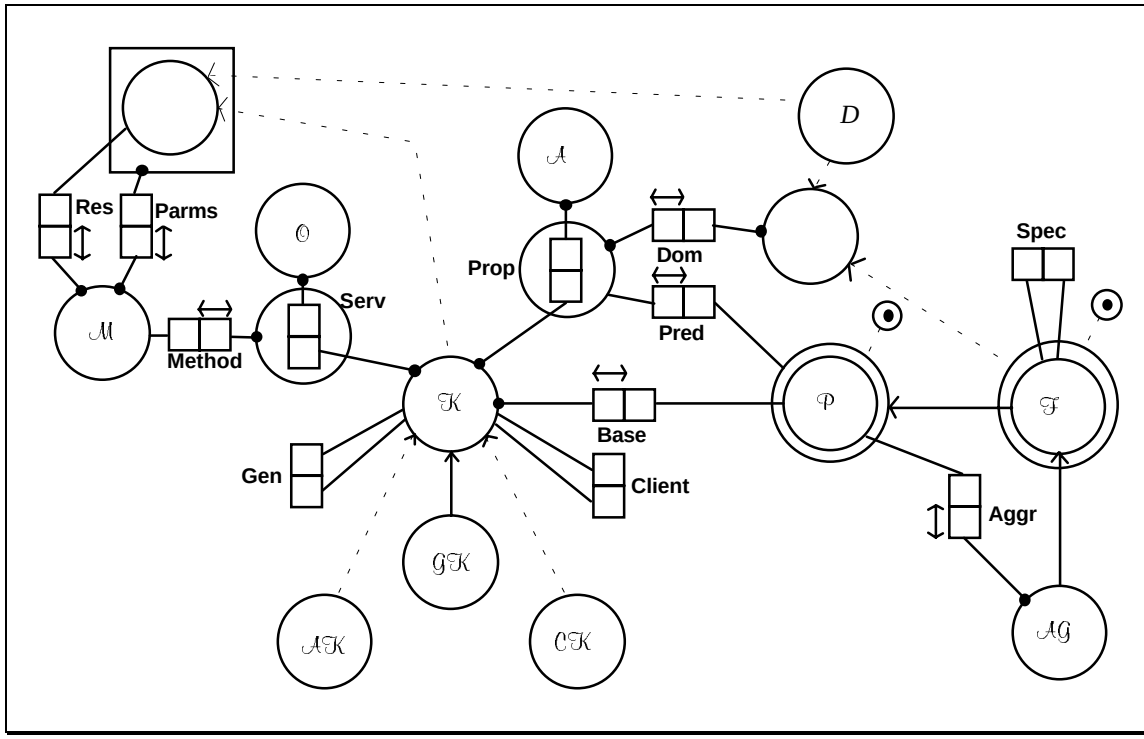
7.2.9 Metamodel

In figuur 7.7 is een metamodel voor objectstructuren weergegeven. Hiervoor is de PSM notatie gebruikt Voor een verklaring van die PSM notatie wordt verwezen naar [HOF93].

In het metamodel ontbreken subtype definiërende regels voor de gebruikte subtypen ($\mathcal{AG}$, $\mathcal{F}$ en $\mathcal{GK}$), omdat het hier geen echte subtypen betreft, maar deelverzamelingen, die uit praktische overwegingen een eigen naam hebben gekregen. De populaties van deze entiteittypen zijn niet afleidbaar uit de informatiestructuur, maar dienen door de analist zelf vastgelegd te worden.

7.3 Overerving

In de vorige sectie is een objectstructuur gepresenteerd en is een aantal axioma's gegeven, die de verbanden tussen de verschillende componenten van de objectstructuur beschrijven. In deze sectie wordt een aantal aanvullende regels, die overerving van eigenschappen en gedrag beschrijven, gepresenteerd. Maar eerst



Figuur 7.6: Een metamodel van objectstructuren.

wordt de ~ relatie geïntroduceerd, waarmee het verband tussen domeinen, parameterlijsten en resultaten kan worden vastgelegd.

7.3.1 De ~ relatie

In de vorige sectie is gebleken, dat zowel domeinen van attributen in objectklassen als parameters en resultaten van methoden waarden kunnen aannemen uit verschillende soorten verzamelingen. Domeinen van attributen kunnen concrete domeinen of verzamelingen instantie associaties zijn. Parameters en resultaten van methoden kunnen instanties van objectklassen zijn of waarden uit een concreet domein. Om vergelijkingen, die op verschillende verzamelingen betrekking kunnen hebben, eenvoudiger te kunnen formuleren, wordt de binaire relatie ~ gedefinieerd.

$$\sim \subseteq (\mathcal{K} \times \mathcal{K}) \cup (D \times D) \cup \left(\wp(\mathcal{F}) \times \wp(\mathcal{F})\right) \cup \left((\mathcal{K} \cup D)^+ \times (\mathcal{K} \cup D)^+\right).$$

Op deze relatie zijn een aantal afleidingsregels van toepassing.

[T1] $k,l \in \mathcal{K} \wedge k\ \mathbf{Gen}^*\ l \vdash l \sim k$

[T2] $c,d \in D \wedge c \subseteq d \vdash c \sim d$

[T3] $A,B \subseteq \mathcal{F} \wedge \forall_{f \in A} \exists_{g \in B} [f\ \mathbf{Spec}\ g] \wedge \forall_{g \in B} \exists_{f \in A} [f\ \mathbf{Spec}\ g] \vdash A \sim B$

[T4] $S,T \in (\mathcal{K} \cup D)^+ \wedge |S| = |T| \wedge \forall_{i \in \{1,..,|S|\}} [S[i] \sim T[i]] \vdash S \sim T$

De **Spec** relatie wordt in de volgende paragraaf behandeld.

7.3.2 Overerving van attributen en domeinen

Een objectklasse erft het gedrag van zijn generalisaties. Alle attributen van een objectklasse worden dus geërfd door de objectklassen, waarvan die objectklasse een generalisatie is.

[I1] $a\,\mathbf{Prop}\,k \wedge k\,\mathbf{Gen}\,l \Rightarrow a\,\mathbf{Prop}\,l$

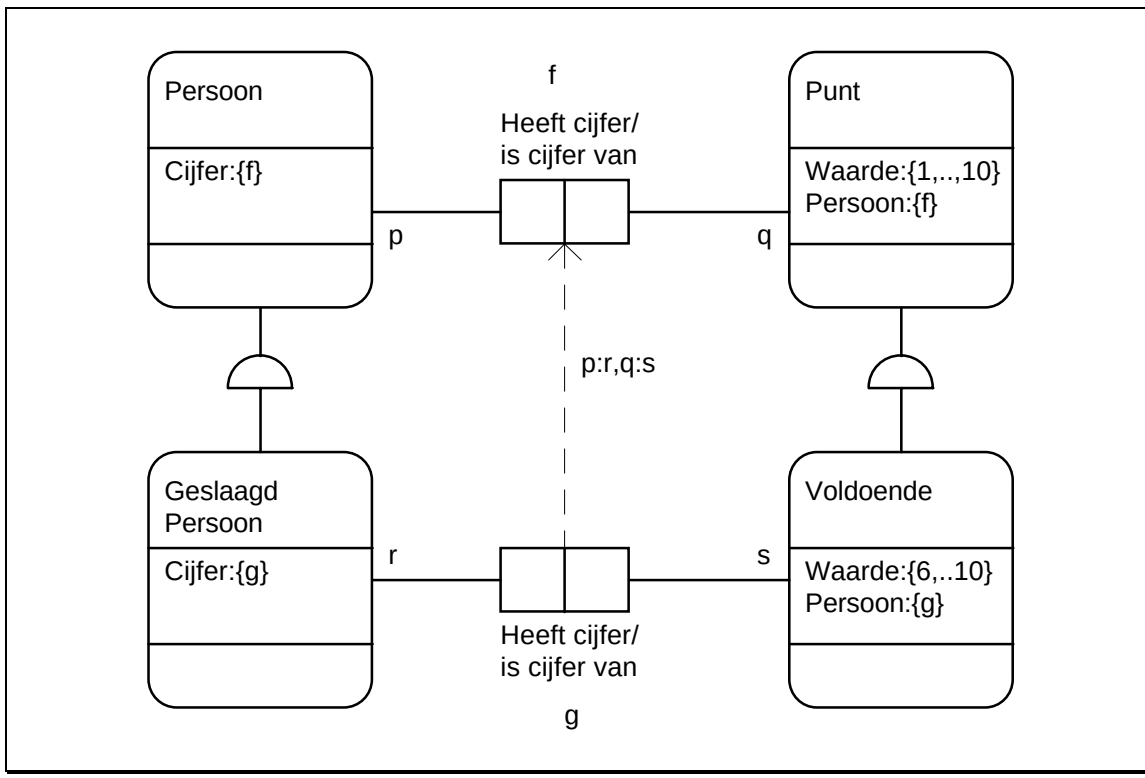
Het predicaat $\mathbf{PropInh}(k,a)$ geeft aan of een attribuut, dat behoort tot een objectklasse, is geërfd van een generalisatie van die objectklasse en is gedefinieerd als

$$\mathbf{PropInh}(k,a) = \exists_{l \in \mathbb{K}}\left[l\,\mathbf{Gen}\,k \wedge a\,\mathbf{Prop}\,l\right].$$

Het is toegestaan, dat de domeinen van geërfde attributen opnieuw worden gedefinieerd. In voorbeeld 7.3.1 is een situatie gemodelleerd, waarin een dergelijke herdefiniëring zinvol is.

Voorbeeld 7.3.1

In figuur 7.7 is een situatie gemodelleerd, waarin objectklassen attributen erven, maar met een domein, dat opnieuw is gedefinieerd. Een Persoon kan een Punt halen. Punt is hierbij een concrete objectklasse, de inkapseling van het domein {1,..,10}*. Een Persoon is geslaagd, als zijn punt voldoende is. Een punt is voldoende, als het hoger is dan een* 5*. Wanneer een geslaagd persoon nieuwe eigenschappen en nieuw gedrag krijgt (bijvoorbeeld het mogen aanvragen van een rijbewijs), dient er een nieuwe objectklasse Geslaagd Persoon gecreëerd te worden, waartoe alle personen*



Figuur 7.7: Een voorbeeld van het herdefiniëren van het domein van een geërfd attribuut.

behoren met een cijfer hoger dan 5. *Bovendien moet er een objectklasse Voldoende worden gecreeerd, de inkapseling van het domein* {1,..,6}. *Tenslotte moet er een nieuwe instantie associatie (in dit geval aangeduid met de naam g) worden gecreëerd, die de nieuwe objectklassen met elkaar associeert. De instantie associaties f en g associëren verschillende objectklassen, maar zij modelleren een vergelijkbare instantie associatie en zij kennen dezelfde rollen. Met andere woorden:* g **Spec** f.

Als een instantie associatie wordt gespecialiseerd, dan moet minimaal één predicator van de gespecialiseerde instantie associatie een basis hebben, die een generalisatie is van de basis van een predicator van de specialisatie. De overige predicatoren hebben een basis, die gelijk is aan de basis van een predicator uit de specialisatie.

[I2] $g\,\mathbf{Spec}\,f \Rightarrow \exists_{\phi: f \leftrightarrow g} \exists_{p \in f} \left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi(p))\right]$

Een functie ϕ is een matching van f naar g (ϕ**:** $f \leftrightarrow g$) als

$$\phi\mathbf{:}\ f \leftrightarrow g = \phi\mathbf{:}\ f \rightarrow g \wedge f, g \in \mathfrak{F} \wedge \mathbf{bijection}(\phi) \wedge \forall_{p \in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{*}\mathbf{Base}(\phi(p))\right],$$

waarbij het predicaat $\mathbf{bijection}(\phi)$ op de gebruikelijke manier kan worden gedefinieerd.

Het volgende lemma stelt, dat specialisatie een irreflexieve relatie is.

Lemma 7.3.1 $\forall_{f \in \mathfrak{F}}\left[\neg f\ \mathbf{Spec}\ f\right]$

Bewijs

Stel $\exists_{f \in \mathfrak{F}}\left[f\ \mathbf{Gen}\ f\right]$. Uit **[I2]** volgt, dat er een bijectie ϕ**:** $f \leftrightarrow f$ is, zodanig dat $\exists_{p \in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi(p))\right]$. ϕ beeldt f af op zichzelf en is dus een transformatie. Dus ϕ is een bijectieve transformatie, ofwel een permutatie, van f.
$\exists_{p \in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi(p))\right]$, dus er is minimaal één predicator p, die element is van f en waarvoor geldt $\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi(p))$. Definieer $f' = \left\{p \in f \,|\, \mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi(p))\right\}$.
Dan geldt $|f'| \geq 1$ en is ϕ een permutatie van f'.
Wij bewijzen nu eerst het onderstaande lemma.
Lemma: $\forall_{i \in \aleph}\forall_{p \in f'}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi^{i+1}(p))\right]$
Bewijs met inductie naar i. Het geval $i=0$ is hierboven reeds bewezen.
Stel $\forall_{p \in f'}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi^{i+1}(p))\right]$. ϕ is een permutatie van f', dus $\forall_{p \in f'}\left[\phi^{i+1}(p) \in f'\right]$.
Verder geldt ϕ**:** $f \leftrightarrow f$ en dus $\forall_{p \in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{*}\mathbf{Base}(\phi(p))\right]$.
$f' \subseteq f$, dus $\forall_{p \in f'}\left[\mathbf{Base}(p)\mathbf{Gen}^{*}\mathbf{Base}(\phi(p))\right]$.
Maar dan geldt $\forall_{p \in f'}\left[\mathbf{Base}(\phi^{i+1}(p))\mathbf{Gen}^{*}\mathbf{Base}(\phi^{i+2}(p))\right]$. Toepassing van de inductiehypothese levert $\forall_{p \in f'}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi^{i+1}(p)) \wedge \mathbf{Base}(\phi^{i+1}(p))\mathbf{Gen}^{*}\mathbf{Base}(\phi^{i+2}(p))\right]$. Uit de transitiviteit van $\mathbf{Gen}^{+}$ volgt dan $\forall_{p \in f'}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi^{i+2}(p))\right]$. Dus geldt het lemma.
Dus $\forall_{p \in f'}\forall_{i \in \aleph}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi^{i+1}(p))\right]$.
ϕ is een permutatie van f', dus $\forall_{p \in f'}\exists_{k \in \aleph}\left[\phi^{k+1}(p) = p\right]$.

Maar dan geldt $\forall_{p\in f'}\exists_{i\in\aleph}\left[\phi^{i+1}(p) = p \wedge \mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}\left(\phi^{i+1}(p)\right)\right]$, ofwel
$\forall_{p\in f'}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(p)\right]$.
Dus $\forall_{p\in f'}\left[\mathbf{Base}(p)\mathbf{GenBase}(p)\vee\exists_{k\in K}\left[\mathbf{Base}(p)\mathbf{Gen}k \wedge k\,\mathbf{Gen}^{+}\mathbf{Base}(p)\right]\right]$.
Uit **[OM16]** kan eenvoudig worden afgeleid, dat **Gen** een irreflexieve relatie is. Toepassing van dat gegeven leidt tot $\forall_{p\in f'}\exists_{k\in K}\left[\mathbf{Base}(p)\mathbf{Gen}k \wedge k\,\mathbf{Gen}^{+}\mathbf{Base}(p)\right]$. Uit de definitie van $\mathbf{Gen}^{+}$ volgt dan $\forall_{p\in f'}\exists_{k\in K}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\,k \wedge k\,\mathbf{Gen}^{+}\mathbf{Base}(p)\right]$. En dat is in tegenspraak met **[OM16]**.

Dus $\neg\exists_{f\in\mathfrak{F}}[\,f\ \mathbf{Gen}\ f\,]$, ofwel $\forall_{f\in\mathfrak{F}}[\neg f\ \mathbf{Gen}\ f\,]$. •

Bovendien is specialisatie asymmetrisch.

Lemma 7.3.2 $\forall_{f,g\in\mathfrak{F}}[\,f\ \mathbf{Spec}\ g \Rightarrow \neg g\ \mathbf{Spec}\ f\,]$

Bewijs

Stel $f\ \mathbf{Spec}\ g$ en $g\ \mathbf{Spec}\ f$. Dan zijn er twee bijecties $\phi_1 : f \leftrightarrow g$ en $\phi_2 : g \leftrightarrow f$, zodanig dat $\exists_{p\in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}\left(\phi_1(p)\right)\right]$ en $\exists_{p\in g}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}\left(\phi_2(p)\right)\right]$. ϕ_1 en ϕ_2 zijn bijecties, dus $\phi_2\circ\phi_1 : f \to f$ is een bijectie.
$\phi_1 : f \leftrightarrow g$ en $\phi_2 : g \leftrightarrow f$, dus uit de definitie van $\leftrightarrow$ volgt $\forall_{p\in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{*}\mathbf{Base}\left(\phi_1(p)\right)\right]$ en $\forall_{p\in g}\left[\mathbf{Base}(p)\mathbf{Gen}^{*}\mathbf{Base}\left(\phi_2(p)\right)\right]$. Uit de transitiviteit van $\mathbf{Gen}^{*}$ volgt dan $\forall_{p\in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{*}\mathbf{Base}\left(\phi_2\circ\phi_1(p)\right)\right]$. Dus $\phi_2\circ\phi_1 : f \leftrightarrow f$.
$\exists_{p\in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}\left(\phi_1(p)\right)\right]$ en $\forall_{p\in g}\left[\mathbf{Base}(p)\mathbf{Gen}^{*}\mathbf{Base}\left(\phi_2(p)\right)\right]$. Uit de transitiviteit van $\mathbf{Gen}^{+}$ volgt dan $\exists_{p\in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}\left(\phi_2\circ\phi_1(p)\right)\right]$. Dus $f\ \mathbf{Spec}\ f$. Maar dat is in tegenspraak met Lemma 7.3.1.
Dus $\neg\exists_{f,g\in\mathfrak{F}}[\,f\ \mathbf{Spec}\ g \wedge g\ \mathbf{Spec}\ f\,]$. Dus $\forall_{f,g\in\mathfrak{F}}[\,f\ \mathbf{Spec}\ g \Rightarrow \neg g\ \mathbf{Spec}\ f\,]$. •

Tenslotte kan worden bewezen, dat **Spec** transitief is.

Lemma 7.3.3 $\forall_{f,g,h\in\mathfrak{F}}[\,f\ \mathbf{Spec}\ g \wedge g\ \mathbf{Spec}\ h \Rightarrow f\ \mathbf{Spec}\ h\,]$

Bewijs

Stel $f\ \mathbf{Spec}\ g$ en $g\ \mathbf{Spec}\ h$. Dan zijn er twee bijecties $\phi_1 : f \leftrightarrow g$ en $\phi_2 : g \leftrightarrow h$, zodanig dat $\exists_{p\in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}\left(\phi_1(p)\right)\right]$ en $\exists_{p\in h}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}\left(\phi_2(p)\right)\right]$. ϕ_1 en ϕ_2 zijn bijecties, dus $\phi_2\circ\phi_1 : f \to h$ is een bijectie.
$\phi_1 : f \leftrightarrow g$ en $\phi_2 : g \leftrightarrow h$, dus uit de definitie van $\leftrightarrow$ volgt $\forall_{p\in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{*}\mathbf{Base}\left(\phi_1(p)\right)\right]$ en $\forall_{p\in g}\left[\mathbf{Base}(p)\mathbf{Gen}^{*}\mathbf{Base}\left(\phi_2(p)\right)\right]$. Uit de transitiviteit van $\mathbf{Gen}^{*}$ volgt dan $\forall_{p\in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{*}\mathbf{Base}\left(\phi_2\circ\phi_1(p)\right)\right]$. Dus $\phi_2\circ\phi_1 : f \leftrightarrow h$

$\exists_{p \in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi_1(p))\right]$ en $\forall_{p \in g}\left[\mathbf{Base}(p)\mathbf{Gen}^{*}\mathbf{Base}(\phi_2(p))\right]$. Uit de transitiviteit van $\mathbf{Gen}^{+}$ volgt dan $\exists_{p \in f}\left[\mathbf{Base}(p)\mathbf{Gen}^{+}\mathbf{Base}(\phi_2 \circ \phi_1(p))\right]$.
Dus $f\ \mathbf{Spec}\ h$. •

Met het oog op de complexiteit van de modellen is het voldoende om alleen rechtstreekse specialisaties in het model aan te geven en transitieve specialisaties weg te laten. Om dezelfde reden hoeft de bijectie ϕ alleen in het model opgenomen te worden, als zij niet afleidbaar is uit de inhoud van het model.

Voorbeeld 7.3.2

In figuur 7.7 is de gegeven bijectie de enige bijectie van f naar g, die voldoet aan **[I2]**. *Zij mag daarom worden weggelaten.*

De pater familias relatie Π is gedefinieerd als

$$\Pi(f,g) = (f = g \vee f\ \mathbf{Spec}\ g) \wedge \neg\mathbf{spec}(g).$$

Hierin is $\mathbf{spec}(g)$ een afkorting voor $\exists_{x \in \mathfrak{F}}[x\ \mathbf{Spec}\ g]$. Het volgende lemma stelt, dat iedere instantie associatie minimaal één pater familias heeft.

Lemma 7.3.4 $\forall_{f \in \mathfrak{F}}\exists_{g \in \mathfrak{F}}[\Pi(f,g)]$

Bewijs:

Daar $\mathfrak{F}$ eindig is en uit Lemma 7.3.2 en Lemma 7.3.3 eenvoudig af te leiden is, dat specialisatiestructuren acyclisch zijn, kunnen wij een functie $\mathbf{depth}: \mathfrak{F} \to \aleph$ definiëren.

$$\mathbf{depth}(f) = \begin{cases} 0 & \text{if} \quad \neg\mathbf{spec}(f) \\ 1 + \min\limits_{f\ \mathbf{Spec}\ g} \mathbf{depth}(g) & \text{otherwise} \end{cases}$$

Het lemma kan nu worden bewezen met behulp van inductie naar de diepte van instantie associatie f.
Als $\mathbf{depth}(f) = 0$, dan geldt $\neg\mathbf{spec}(f)$. $f = f$, dus $\Pi(f,f)$.
Als $\mathbf{depth}(f) = n+1$, dan bestaat er een instantie associatie g, zodanig dat $f\ \mathbf{Spec}\ g$ en $\mathbf{depth}(g) = n$. Uit toepassing van de inductiehypothese volgt dan, dat er een instantie associatie h bestaat, zodanig dat $\Pi(g,h)$. Maar dan geldt $g = h \vee g\ \mathbf{Spec}\ h$ en tevens $\neg\mathbf{spec}(h)$. Uit de transitiviteit van $\mathbf{Spec}$ volgt dan $f\ \mathbf{Spec}\ h$. Dus $\Pi(f,h)$. Dus $\forall_{f \in \mathfrak{F}}\exists_{g \in \mathfrak{F}}[\Pi(f,g)]$. •

Met behulp van het volgende axioma wordt gezorgd, dat een specialisatie nooit meer dan één pater familias heeft.

[I3] $\Pi(f,g) = \Pi(f,h) \Rightarrow g = h$

[I3] en Lemma 7.3.4 zorgen er voor, dat iedere specialisatie een unieke pater familias heeft, ofwel $\forall_{f \in \mathfrak{F}}\exists!_{g \in \mathfrak{F}}[\Pi(f,g)]$. Voor iedere f wordt deze unieke g genoteerd als $\Pi(a)$. Uit de definitie van Π is direct af te leiden dat $\neg\mathbf{spec}(\Pi(f))$.

Een specialisatie heeft dezelfde pater familias als de instantie associaties, waarvan het een specialisatie is.

Lemma 7.3.5 $\forall_{f,g\in\mathfrak{F}}\left[f\ \mathbf{Spec}\ g \Rightarrow \Pi(f)=\Pi(g)\right]$

Bewijs

Stel $f\ \mathbf{Spec}\ g$. Dan volgt uit de definitie van Π dat $\Pi(g)=g\vee g\ \mathbf{Spec}\ \Pi(g)$. Uit de transitiviteit van **Spec** volgt dan $f\ \mathbf{Spec}\ \Pi(g)$. Verder geldt $\neg\mathbf{spec}(\Pi(g))$, dus $\Pi(f)=\Pi(g)$. •

Aan herdefiniëring van domeinen van attributen en van associaties wordt in de literatuur nauwelijks aandacht besteed. Een enigszins met onze benadering vergelijkbaar concept is de in [WJ92] uitgewerkte subklassering van rollen.

Een attribuut, dat in een gegeven objectklasse een domein heeft met daarin instantie associaties, die geen predicatoren bevatten, die als basis de gegeven objectklasse hebben, is geërfd.

[I4] $f\in\mathbf{Dom}(a,k)\wedge\forall_{p\in f}\left[\mathbf{Base}(p)\neq k\right]\Rightarrow\mathbf{PropInh}(k,a)$

Als in een gegeven objectklasse een attribuut is geërfd van één of meer generalisaties van die objectklasse, dan wordt van één van die generalisaties het domein geërfd of gespecialiseerd.

[I5] $\mathbf{PropInh}(k,a)\Rightarrow\exists_{l\in\mathfrak{K}}\left[l\ \mathbf{Gen}\ k\wedge\mathbf{Dom}(a,k)\sim\mathbf{Dom}(a,l)\right]$

Een domein, dat bestaat uit instantie associaties, moet bovendien samenhangend zijn. Hiermee bedoelen wij, dat alle instantie associaties in een dergelijk domein dezelfde pater familias moeten hebben.

[I6] $\mathbf{Dom}(a,k)\subseteq\mathfrak{F}\wedge f\in\mathbf{Dom}(a,k)\wedge g\in\mathbf{Dom}(a,k)\Rightarrow\Pi(f)=\Pi(g)$

Specialisatie kan nog op een tweede manier worden gebruikt, namelijk voor het herdefiniëren van constraints. Constraints worden behandeld in sectie 8.3 (Grafische Constraints).

7.3.3 Overerving van operaties en methoden

Iedere objectklasse erft alle operaties van zijn generalisaties.

[I7] $a\ \mathbf{Serv}\ k\wedge k\ \mathbf{Gen}\ l\Rightarrow a\ \mathbf{Serv}\ l$

Het predicaat $\mathbf{ServInh}(k,a)$ is gedefinieerd als

$$\mathbf{ServInh}(k,a)=\exists_{l\in\mathfrak{K}}\left[l\ \mathbf{Gen}\ k\wedge o\ \mathbf{Serv}\ k\right].$$

Evenals bij het overerven van attributen en domeinen geldt bij het overerven van operaties en methoden, dat een objectklasse alle operaties erft van zijn generalisaties, maar dat de methoden, die die operaties implementeren, opnieuw kunnen worden gedefinieerd.

Als twee objectklassen een operatie met behulp van dezelfde methode implementeren en die methode is niet de lege methode, dan hebben zij de methode geërfd van een gezamenlijke voorouder.

[I8] $\mathbf{Method}(o,k1)=\mathbf{Method}(o,k2)\wedge\mathbf{Method}(o,k1)\neq\perp\Rightarrow$
$\exists_{l\in\mathfrak{K}}\left[l\ \mathbf{Gen}^{*}\ k1\wedge l\ \mathbf{Gen}^{*}\ k2\wedge\mathbf{Method}(o,k1)=\mathbf{Method}(o,l)\right]$

Een objectklasse, die een operatie van één van zijn generalisaties erft, erft ook de methode, of implementeert de operatie met behulp een methode, die vergelijkbaar is met de methode van die generalisatie. De enige uitzondering is het geval, waarin in de generalisatie de operatie is geïmplementeerd met behulp van de lege methode. Met vergelijkbare methode wordt bedoeld, dat of het resultaat is gespecialiseerd, of de parameters, of beide.

[I9]
$$\begin{aligned}\mathbf{ServInh}(k,a) \Rightarrow \forall_{l\in\mathbb{K}}\Big[l\,\mathbf{Gen}\,k \Rightarrow \mathbf{Method}(o,l)\uparrow \vee \mathbf{Method}(o,l) = \bot\Big] \vee \\ \exists_{l\in K}\Big[l\,\mathbf{Gen}\,k \wedge \mathbf{MethodSpec}\big(\mathbf{Method}(o,k),\mathbf{Method}(o,l)\big)\Big]\end{aligned}$$

$\mathbf{MethodSpec}(m1,m2)$ is gedefinieerd als

$$\mathbf{MethodSpec}(m1,m2) = \mathbf{ParmsSpec}(m1,m2) \wedge \mathbf{ResSpec}(m1,m2).$$

De hierbij gebruikte predicaten zijn op hun beurt gedefinieerd als

$$\mathbf{ParmsSpec}(m1,m2) = \big(\mathbf{Parms}(m1)\uparrow \wedge \mathbf{Parms}(m2)\uparrow\big) \vee \mathbf{Parms}(m1) \sim \mathbf{Parms}(m2);$$
$$\mathbf{ResSpec}(m1,m2) = \big(\mathbf{Res}(m1)\uparrow \wedge \mathbf{Res}(m2)\uparrow\big) \vee \mathbf{Res}(m1) \sim \mathbf{Res}(m2).$$

8 Populaties, Constraints en Identificatie

8.1 Inleiding

In het vorige hoofdstuk is een syntax voor objectstructuren gegeven. In dit hoofdstuk besteden wij eerst aandacht aan het semantische aspect van dergelijke objectstructuren en geven wij een aantal regels met betrekking tot populaties van objectstructuren. Vervolgens gaan wij in op de voorwaarden, die aan dergelijke populaties kunnen worden gesteld, en de wijze, waarop wij deze voorwaarden in het model op kunnen nemen. De syntax en semantiek van deze zogenaamde grafische constraints wordt formeel beschreven. Tenslotte wordt ingegaan op de wijze, waarop object instanties, die behoren tot de populatie van een objectstructuur, kunnen worden geïdentificeerd.

8.2 Populaties

In het vorige hoofdstuk is een syntax voor objectstructuren gegeven. In deze sectie gaan wij in op populaties voor die objectstructuren. In onze benadering is een populatie **Pop** van een objectstructuur

$$\mathbb{OS} = \left\langle \begin{array}{l} \mathcal{K},\mathcal{AK},\mathcal{CK},\mathcal{GK},\mathcal{P},\mathcal{F},\mathcal{AG},\mathcal{A},\mathcal{O},\mathcal{M},\mathcal{D},\mathbf{Base},\mathbf{Aggr},\mathbf{Prop}, \\ \mathbf{Dom},\mathbf{Pred},\mathbf{Serv},\mathbf{Method},\mathbf{Parms},\mathbf{Res},\mathbf{Client},\mathbf{Gen},\mathbf{Spec} \end{array} \right\rangle$$

een toewijzing, die een eindige verzameling object instanties toewijst aan alle objectklassen in $\mathcal{K}$ en die aan de attributen van die object instanties waarden uit het domein van die attributen toewijst. Deze toewijzing moet voldoen aan een aantal regels, die in deze sectie worden gepresenteerd. De uitdrukking $\mathbf{IsPop}(\mathbb{OS},\mathbf{Pop})$ geeft aan, dat **Pop** een populatie is van objectstructuur $\mathbb{OS}$. In dat geval is **Pop** een geordend paar $\langle \mathbf{CPop},\mathbf{IPop} \rangle$, waarbij **CPop** een functie $\mathbf{CPop}: \mathcal{K} \to \wp(\Theta)$ is en **IPop** een partiële functie $\mathbf{IPop}: \Theta \mapsto \wp(\Omega) \setminus \{\varnothing\}$, met $\mathbf{IPop}(x)\downarrow \Leftrightarrow \exists_{k \in \mathcal{K}}\, x \in \mathbf{CPop}(k)$. De verzameling van alle populaties van een objectstructuur $\mathbb{OS}$ is gedefinieerd als $\mathbf{POP}_{\mathbb{OS}} = \{\mathbf{Pop} | \mathbf{IsPop}(\mathbb{OS},\mathbf{Pop})\}$.

De elementen van de populatie van een objectklasse, ofwel de object instanties, komen uit het domein Θ. De verzameling Θ is een eindige verzameling ongestructureerde waarden. De verzameling Ω is een verzameling geordende paren, bestaande uit een attribuut en een waarde uit het domein van dat attribuut. Hierbij fungeert Ω als synoniem voor $\mathcal{A} \times \left(\bigcup D \cup \wp(P \times \Xi)\right)$. Voor de definitie van de verzameling Ξ wordt verwezen naar paragraaf 8.4.1 (Object identiteit op machineniveau).

8.2.1 Populaties van objectklassen

Met betrekking tot de populaties van objectklassen is een aantal regels van kracht. Een object instantie behoort tot ten hoogste één objectklasse.

[P1] $\quad k,l \in \mathcal{K} \wedge k \neq l \Rightarrow \mathbf{CPop}(k) \cap \mathbf{CPop}(l) = \varnothing$

Uit paragraaf 4.2.5 (De methode van Martin en Odell) is gebleken, dat de methode van Martin en Odell meervoudige classificatie ondersteunt. Daarmee wordt bedoeld, dat een object instantie tot meer objectklassen kan behoren (zie verder [MO92]). Meervoudige classificatie vereist aanvullende regels met betrekking tot het toewijzen van domeinen aan attributen en methoden aan operaties.

Wij beschouwen bij wijze van voorbeeld een object instantie, die behoort tot twee objectklassen, die allebei een bepaald attribuut hebben. De domeinen van het attribuut voor de twee objectklassen zijn verschillend.

Er kan nu niet uit het Object Model worden afgeleid, welk domein het domein van het attribuut moet vormen. Daarom dienen nu aanvullende regels opgesteld te worden, die de toewijzing van domeinen aan attributen en methoden aan operaties regelen, bijvoorbeeld met behulp van een prioriteitenstelsel voor objectklassen.

In deze scriptie beperken wij ons tot enkelvoudige classificatie.

Een generalisatie objectklasse dient alleen ter bevordering van de herbruikbaarheid van modellen en heeft geen functie in het uiteindelijke systeem. Dergelijke objectklassen hoeven geen implementaties voor hun operaties te kennen (zie 7.2.7, Methoden). Het is daarom niet toegestaan om generalisatie objectklassen te populeren.

[P2] $k \in \mathcal{GK} \Rightarrow \mathbf{CPop}(k) = \varnothing$

8.2.2 Toewijzing van waarden aan de attributen van object instanties

Ook voor de toewijzing van waarden aan de attributen van de object instanties, behorend tot concrete objectklassen, geldt een aantal regels. Bij het beschrijven van deze regels wordt gebruik gemaakt van een aantal hulpfuncties, die eerst worden gedefinieerd.

De functies $\mathbf{Attr}:\Omega \to a$ en $\mathbf{Value}:\Omega \to \left(\bigcup D \cup \wp(\mathcal{P} \times \Xi)\right)$ zijn de projecties op respectievelijk het eerste en het tweede element van een geordend paar uit Ω.

$$y \in \Omega \wedge y = \langle a, z \rangle \Rightarrow \mathbf{Attr}(y) = a \wedge \mathbf{Value}(y) = z$$

De verzameling relevante objectklassen, ofwel de verzameling objectklassen, die voor alle predicatoren in een gegeven verzameling predicatoren een attribuut kennen, waaraan die predicator is toegewezen, wordt gegeven door de functie $\mathbf{RelCl}:\wp(\mathcal{P}) \to \wp(\mathcal{K})$.

$$\mathbf{RelCl}(\tau) = \left\{ k \in \mathcal{K} \mid \forall_{p \in \tau} \exists_{a \in \mathcal{A}} \left[a\,\mathbf{Prop}\,k \wedge p \in \mathbf{Pred}(a,k) \right] \right\}$$

In het vervolg laten wij voor het gemak de accolades weg, als de verzameling predicatoren slechts één element bevat.

Als aan een object instantie een attribuut met een bepaalde waarde wordt toegewezen, dan moet dat attribuut een attribuut zijn van de objectklasse, waartoe de object instantie behoort.

[P3] $x \in \mathbf{CPop}(k) \wedge y \in \mathbf{IPop}(x) \Rightarrow \mathbf{Attr}(y)\mathbf{Prop}\,k$

Aan alle attributen van de objectklasse, waartoe een object instantie behoort, moet een waarde worden toegewezen. Een dergelijke waarde moet in overeenstemming zijn met het domein van het attribuut in de betreffende objectklasse. Aan een attribuut met een concreet domein mag bovendien per object instantie slechts één waarde worden toegewezen.

[P4] $x \in \mathbf{CPop}(k) \wedge a\,\mathbf{Prop}\,k \wedge \mathbf{Dom}(a,k) \in D \Rightarrow \exists!_{y \in \mathbf{IPop}(x)} \left[\mathbf{Attr}(y) = a \wedge \mathbf{Value}(y) \in \mathbf{Dom}(a,k) \right]$

[P5] $x \in \mathbf{CPop}(k) \wedge a\,\mathbf{Prop}\,k \wedge \mathbf{Dom}(a,k) \subseteq \mathcal{T} \Rightarrow$
$\exists_{y \in \mathbf{IPop}(x)} \left[\mathbf{Attr}(y) = a \wedge \left(\mathbf{Value}(y) = \varnothing \vee \exists_{f \in \mathbf{Dom}(a,k)} \left[\mathbf{Value}(y) : f \to \Xi \right] \right) \right]$

Als een attribuut geen waarde heeft, dan wordt dit gerepresenteerd door aan het betreffende attribuut de waarde 'nil' of de waarde $\varnothing$ toe te kennen. De waarde 'nil' is een waarde, die element is van alle concrete domeinen ($\forall_{d \in D}[\mathrm{nil} \in d]$) en die daarom kan worden toegekend aan alle attributen met een concreet do-

mein. ∅ kan worden toegekend aan alle attributen met één of meer instantie associaties als domein. De waarde 'nil' mag slechts aan een attribuut worden toegewezen, als alle andere attributen een niet gedefinieerde waarde hebben.

[P6] $y \in \mathbf{IPop}(x) \wedge \mathbf{Value}(y) = \text{nil} \Rightarrow \forall_{z \in \mathbf{IPop}(x)} \left[y \neq z \Rightarrow \mathbf{Value}(z) = \varnothing \right]$

De waarde ∅ mag slechts aan een attribuut worden toegewezen, als aan dat attribuut geen andere waarden zijn toegewezen.

[P7] $y \in \mathbf{IPop}(x) \wedge \mathbf{Value}(y) = \varnothing \Rightarrow \neg \exists_{z \in \mathbf{IPop}(x)} \left[y \neq z \wedge \mathbf{Attr}(y) = \mathbf{Attr}(z) \right]$

Als de waarde van een geordend paar, dat aan een gegeven object instantie is toegewezen, een functie is van een instantie associatie naar waarden uit Ξ, dan moet die functie aan iedere predicator van de instantie associatie de object identiteit van een object instantie, die behoort tot een voor die predicator relevante objectklasse, toekennen. Bovendien behoort één van de predicatoren van die instantie associatie tot de aan het attribuut van dat paar toegewezen predicatoren en moet aan deze predicator de object identiteit van de gegeven object instantie toegewezen zijn.

[P8]
$$x \in \mathbf{CPop}(k) \wedge y \in \mathbf{IPop}(x) \wedge \mathbf{Attr}(y) = a \wedge \mathbf{Value}(y)\colon f \rightarrow \Xi \Rightarrow$$
$$\exists_{p \in f} \left[p \in \mathbf{Pred}(a,k) \wedge \mathbf{Value}(y)(p) = \mathbf{OID}(x) \right] \wedge \forall_{p \in f} \left[\mathbf{Obj}\left(\mathbf{Value}(y)(p)\right) \in \bigcup_{l \in \mathbf{RelCl}(p)} \mathbf{CPop}(l) \right]$$

Voor de definities van de functies $\mathbf{OID}\colon \Theta \rightarrow \Xi$ en $\mathbf{Obj}\colon \Xi \mapsto \Theta$ wordt verwezen naar paragraaf 8.4.1 (Object identiteit op machineniveau).

Tenslotte moet een instantie associatie tussen object instanties aan de bijbehorende attributen van alle betrokken object instanties worden toegewezen.

[P9]
$$\exists_{x \in \Theta} \left[y \in \mathbf{IPop}(x) \wedge \mathbf{Value}(y)\colon f \rightarrow \Xi \right] \Rightarrow$$
$$\forall_{p \in f} \exists_{x \in \Theta} \exists_{z \in \mathbf{IPop}(x)} \left[\mathbf{Value}(y)(p) = \mathbf{OID}(x) \wedge \mathbf{Value}(y) = \mathbf{Value}(z) \right]$$

Voorbeeld 8.2.1

Een voorbeeldpopulatie voor de objectstructuur in figuur 7.1 wordt gegeven door

$$\mathbf{CPop}(A) = \varnothing \qquad \mathbf{CPop}(D) = \{y1, y2\}$$
$$\mathbf{CPop}(B) = \{x1, x2\} \qquad \mathbf{CPop}(E) = \{z1, z2\}$$
$$\mathbf{CPop}(C) = \varnothing$$

$$\mathbf{IPop}(x1) = \left\{ \left\langle a1, \left\{ \left\{ t\colon \mathbf{OID}(x1), u\colon \mathbf{OID}(y1) \right\}, \left\{ t\colon \mathbf{OID}(x1), u\colon \mathbf{OID}(y2) \right\} \right\} \right\rangle \right\}$$
$$\mathbf{IPop}(x2) = \left\{ \left\langle a1, \left\{ \left\{ t\colon \mathbf{OID}(x2), u\colon \mathbf{OID}(y2) \right\} \right\} \right\rangle \right\}$$
$$\mathbf{IPop}(y1) = \left\{ \begin{array}{l} \left\langle a2, \left\{ \left\{ t\colon \mathbf{OID}(x1), u\colon \mathbf{OID}(y1) \right\} \right\} \right\rangle, \\ \left\langle a3, \left\{ \left\{ r\colon \mathbf{OID}(y1), s\colon \mathbf{OID}(z1) \right\}, \left\{ r\colon \mathbf{OID}(y1), s\colon \mathbf{OID}(z2)2 \right\} \right\} \right\rangle \end{array} \right\}$$
$$\mathbf{IPop}(y2) = \left\{ \begin{array}{l} \left\langle a2, \left\{ \left\{ t\colon \mathbf{OID}(x1), u\colon \mathbf{OID}(y2) \right\}, \left\{ t\colon \mathbf{OID}(x2), u\colon \mathbf{OID}(y2) \right\} \right\} \right\rangle, \\ \left\langle a3, \left\{ \left\{ r\colon \mathbf{OID}(y2), s\colon \mathbf{OID}(z1) \right\} \right\} \right\rangle \end{array} \right\}$$
$$\mathbf{IPop}(z1) = \left\{ \langle a4, 2 \rangle, \left\langle a5, \left\{ \left\{ r\colon \mathbf{OID}(y1), s\colon \mathbf{OID}(z1) \right\}, \left\{ r\colon \mathbf{OID}(y2), s\colon \mathbf{OID}(z1) \right\} \right\} \right\rangle \right\}$$

$$\mathbf{IPop}(z2) = \left\{\langle a4,4\rangle, \left\langle a5, \left\{\left\{r{:}\,\mathbf{OID}(y1), s{:}\,\mathbf{OID}(z2)\right\}\right\}\right\rangle\right\}$$

8.3 Grafische Constraints

Meestal zijn niet alle mogelijke populaties van een objectstructuur geldig. Hiermee bedoelen wij, dat die populaties niet corresponderen met een bepaalde toestand in het Universum van Discussie. Met behulp van statische constraints kan de verzameling van mogelijke populaties van een objectstructuur worden beperkt tot die populaties, die corresponderen met een bepaalde toestand in het Universum van Discussie.

Een aantal van deze statische constraints komt zo vaak voor in de beschrijving van een objectstructuur, dat de notatie de mogelijkheid biedt om deze constraints grafisch weer te geven. Een grafische representatie vergemakkelijkt in dergelijke gevallen de specificatie en zorgt er verder voor, dat de constraints beter te begrijpen zijn [HOF93].

In deze sectie formaliseren wij de constraints, die met behulp van onze notatie grafisch kunnen worden weergegeven. Hierbij wordt gebruik gemaakt van een relationele algebra voor Object Modellen, die is gebaseerd op de voor PSM gedefinieerde relationele algebra (zie [HOF93]). Formeel bestaat een Object Model schema $\Sigma = \langle \mathbb{OS}, \mathbb{R} \rangle$ uit een objectstructuur $\mathbb{OS}$ en een verzameling constraints $\mathbb{R}$. Een populatie van een schema moet een populatie van zijn objectstructuur zijn en moet voldoen aan zijn constraints.

$$\mathbf{IsPop}(\Sigma, \mathbf{Pop}) = \mathbf{IsPop}(\mathbb{OS}, \mathbf{Pop}) \wedge \forall_{r \in \mathbb{R}} \left[\mathbf{Pop}|= r\right]$$

De verzameling van alle mogelijke grafische constraints van een objectstructuur $\mathbb{OS}$ wordt genoteerd als $\Gamma(\mathbb{OS})$. Per definitie geldt dan $\mathbb{R} \subseteq \Gamma(\mathbb{OS})$. De verzameling van alle populaties van een schema Σ is gedefinieerd als

$$\mathbf{POP}_{\Sigma} = \left\{\mathbf{Pop} \in \mathbf{POP}_{\mathbb{OS}} \,|\, \mathbf{IsPop}(\Sigma, \mathbf{Pop})\right\}.$$

8.3.1 Een relationele algebra voor Object Modellen

In [HOF93] wordt uitgebreid een relationele algebra voor PSM behandeld. Aangezien wij bij het formaliseren van de grafische constraints kunnen volstaan met een relationele algebra, die grotendeels op die relationele algebra voor PSM is gebaseerd, geven wij hier alleen een definitie voor het schema van een instantie associatie en de waarde van een instantie associatie.

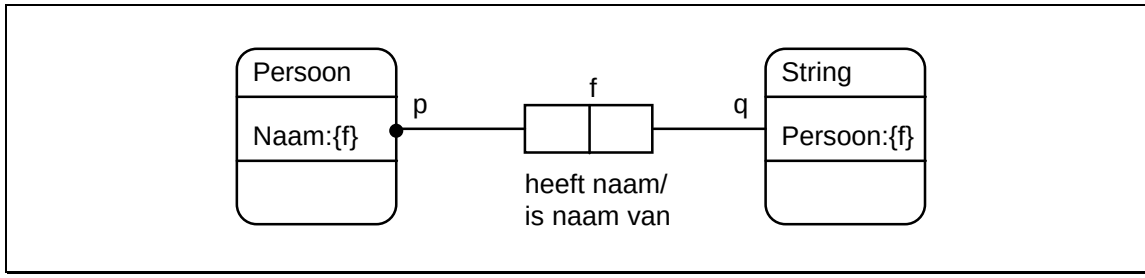
Het schema van een instantie associatie f is de verzameling predicatoren in f ($\mathbf{Schema}(f) = f$).

De populatie van een relationele expressie r wordt geleverd door de operator **Val**, die opereert op een populatie en een verzameling geordende paren oplevert, die stuk voor stuk functies zijn van $\mathbf{Schema}(r)$ naar Θ.

De waarde van een instantie associatie f is gedefinieerd als

$$\mathbf{Val}\left[\!\left[f \right]\!\right](\mathbf{Pop}) = \left\{z{:}\, f \to \Theta \,|\, \exists_{x \in \Theta} \exists_{y \in \mathbf{IPop}(x)} \forall_{p \in f} \left[\mathbf{OID}(z(p)) = \mathbf{Value}(y)(p)\right]\right\}.$$

Voor een uitgebreidere beschrijving van relationele algebra's verwijzen wij naar [HOF93]. Voor een definitie van de operatoren $\cup$, \, π, σ, $\triangleright\!\triangleleft$, χ en θ en het predicaat **IsEmpty** verwijzen wij naar bijlage A (Verklaring van de wiskundige en grafische notaties).



Figuur 8.1: Een voorbeeld van een total role constraint.

8.3.2 Total role constraints

Een instantie associatie kan de eigenschap hebben, dat in iedere populatie de object instanties, die een rol kunnen spelen in die instantie associatie, ofwel de object instanties, behorend tot de relevante objectklassen, die rol minimaal één keer moeten spelen.

Voorbeeld 8.3.1

Figuur 8.1 is een voorbeeld van een total role constraint. Voor iedere object instantie, die behoort tot de objectklasse Persoon, moet aan het attribuut Naam een waarde zijn toewezen, in dit geval een object instantie, behorend tot de objectklasse String. Wij duiden dit aan met behulp van een punt op het snijpunt van de objectklasse en de bij het bewuste attribuut behorende predicator.

Analoog aan de total role constraint in PSM mag de total role constraint in het Object Model betrekking hebben op meerdere predicatoren, die al dan niet tot dezelfde instantie associatie behoren.

Voorbeeld 8.3.2

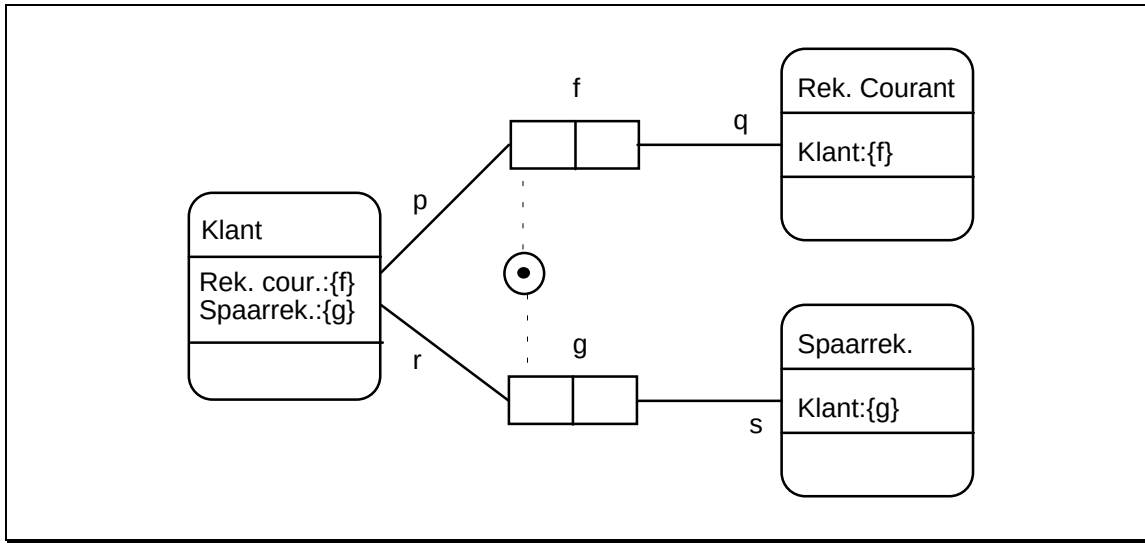
In figuur 8.2 is een total role constraint over meerdere predicatoren weergegeven. Een klant kan verschillende rekeningen hebben, in dit geval een rekening courant of een spaarrekening. Een klant moet altijd minimaal één rekening hebben, maar dit mag of een rekening courant of een spaarrekening zijn. Wij noteren dit door de predicatoren, die behoren bij de attributen, waarop de total role constraint betrekking heeft, te verbinden met een cirkel met daarin een punt.

Syntactisch gezien is een total role constraint τ een niet-lege verzameling predicatoren ($\tau \neq \varnothing \wedge \tau \subseteq \mathcal{P}$). Een total role constraint heeft alleen betekenis, als er relevante objectklassen voor de constraint bestaan, waarvan alle bij de betrokken attributen behorende predicatoren deel uit maken van de constraint. De verzameling objectklassen, waarin een total role constraint van kracht is, duiden wij aan met $\mathbf{TotalCl}(\tau)$.

$$\mathbf{TotalCl}(\tau) = \left\{ k \in \mathbf{RelCl}(\tau) \mid \forall_{a \in \mathcal{A}} \left[a \, \mathbf{Prop} \, k \wedge \exists_{p \in \mathbf{Pred}(a,k)} [p \in \tau] \Rightarrow \forall_{p \in \mathbf{Pred}(a,k)} [p \in \tau] \right] \right\}$$

Een populatie $\mathbf{Pop}$ voldoet aan de total role constraint τ ($\mathbf{Pop} \models \mathbf{total}(\tau)$), dan en slechts dan als geldt

$$\bigcup_{k \in \mathbf{TotalCl}(\tau)} \mathbf{CPop}(k) = \bigcup_{p \in \tau} \mathbf{Val} \left[\left[\theta_p \left(\pi_p \left(\mathbf{Fact}(p) \right) \right) \right] \right] (\mathbf{Pop}).$$



Figuur 8.2: Een voorbeeld van een total role constraint over meerdere predicatoren.

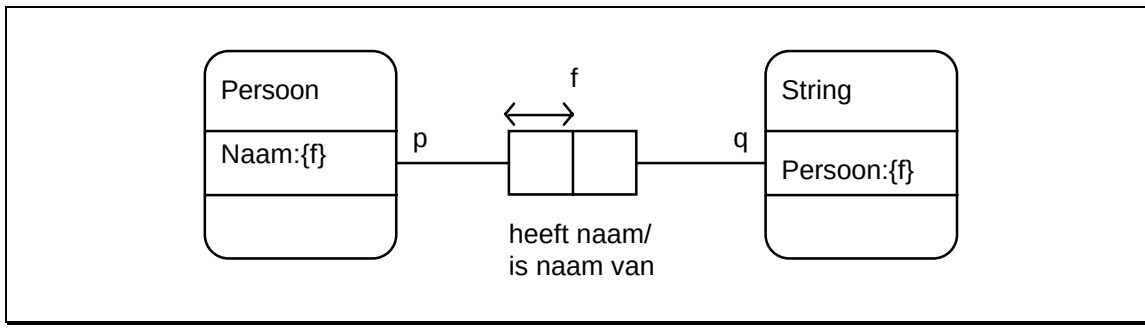
Als voor alle populaties van een schema Σ, waarvoor geldt $\mathbf{IsPop}(\Sigma,\mathbf{Pop})$, geldt $\mathbf{Pop} \models \mathbf{total}(\tau)$, dan is de assertie $\mathbf{total}(\tau)$ waar. Wij schrijven $\mathbf{total}(\tau) \in \mathfrak{R}$, dan en slechts dan als τ een gespecificeerde total role constraint is in schema $\Sigma = \langle \mathfrak{OS}, \mathfrak{R} \rangle$. Het is duidelijk, dat $\mathbf{total}(\tau) \in \mathfrak{R} \Rightarrow \mathbf{total}(\tau)$.

8.3.3 Uniqueness constraints

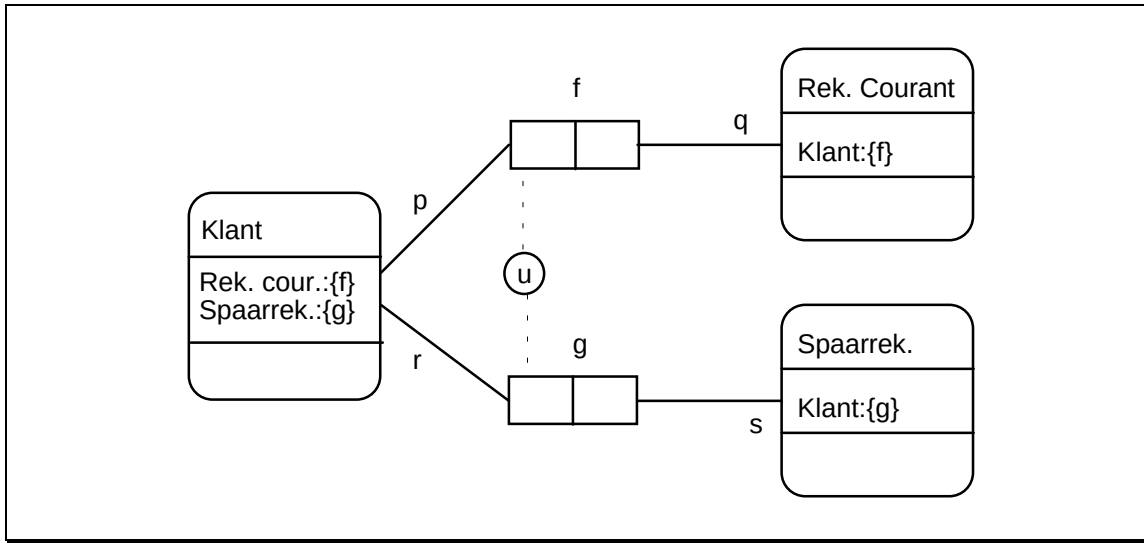
Een instantie associatie kan de eigenschap hebben, dat in iedere populatie de object instanties, die een rol kunnen spelen in die instantie associatie, ofwel de object instanties, die behoren tot de relevante object-klassen, die rol hoogstens één keer spelen.

Voorbeeld 8.3.3

Figuur 8.3 is een voorbeeld van een uniqueness constraint, waarbij één instantie associatie is betrokken. Object instanties, die behoren tot de objectklasse Persoon, hebben een attribuut Naam,



Figuur 8.3: Een voorbeeld van een uniqueness constraint, waarbij één instantie associatie is betrokken.



Figuur 8.4: Een voorbeeld van een uniqueness constraint, waarbij meerdere instantie associaties zijn betrokken.

waarvan de waarde een object instantie is, die behoort tot de objectklasse String. Iedere object instantie, die behoort tot de objectklasse Persoon, mag hoogstens één naam hebben. Wij geven dit grafisch weer door een pijl te plaatsen boven de predicatoren, die behoren tot de uniqueness constraint.

Een uniqueness constraint kan betrekking hebben op meerdere instantie associaties.

Voorbeeld 8.3.4

In figuur 8.4 is een uniqueness constraint over meerdere instantie associaties weergegeven. Een klant kan verschillende rekeningen hebben, in dit geval een rekening courant of een spaarrekening. Een klant kan echter ten hoogste één rekening hebben, of een rekening courant of een spaarrekening. Wij noteren dit door de predicatoren, die behoren bij de attributen, waarop de uniqueness constraint betrekking heeft, te verbinden met een cirkel met daarin een u.

Syntactisch gezien is een uniqueness constraint τ een niet-lege verzameling predicatoren ($\tau \neq \varnothing \wedge \tau \subseteq \mathcal{P}$). Een uniqueness constraint heeft alleen betekenis, als de predicatoren in τ met elkaar zijn verbonden via relevante objectklassen, ofwel als geldt $\mathbf{Jn}(\tau)$.

$$\mathbf{Jn}(\tau) = \left|\mathbf{Facts}(\tau) > 1\right| \Rightarrow \exists_{f,g \in \mathbf{Facts}(\tau)}\left[f \neq g \wedge f \setminus \tau \sim g \setminus \tau \wedge \mathbf{Jn}(\tau \setminus f) \right]$$

Twee verzamelingen predicatoren τ_1 en τ_2 zijn aan elkaar gerelateerd ($\tau_1 \sim \tau_2$), dan en slechts dan als zij een predicator bevatten, zodanig dat voor de combinatie van die predicatoren relevante objectklassen bestaan.

$$\tau_1 \sim \tau_2 \Leftrightarrow \exists_{p \in \tau_1, q \in \tau_2}\left[\mathbf{RelCl}(\{p,q\}) \neq \varnothing\right]$$

De uitdrukking $\mathbf{Facts}(\tau)$ levert de verzameling instantie associaties, waartoe de predicatoren van τ behoren.

$$\mathbf{Facts}(\tau) = \{\mathbf{Fact}(p) \mid p \in \tau\}$$

Hieruit volgt, dat nooit alle predicatoren van één instantie associatie kunnen behoren tot een uniqueness constraint, die op meerdere instantie associaties betrekking heeft.

Als een uniqueness constraint predicatoren bevat van instantie associaties, die tot dezelfde specialisatiestructuur behoren, dan moet bovendien voldaan zijn aan

$$f, g \in \mathbf{Facts}(\tau) \wedge \Pi(f) = \Pi(g) \wedge \phi_1 : \Pi(f) \to f \wedge \phi_2 : \Pi(g) \to g \Rightarrow \forall_{p \in f}\left[p \in \tau \Rightarrow \phi_2 \circ \phi_1{}^{-1}(p) \in \tau\right],$$

waarbij ϕ_1 en ϕ_2 de in de objectstructuur gespecificeerde bijecties zijn.

Een populatie voldoet aan een uniqueness constraint τ ($\mathbf{Pop} \models \mathbf{unique}(\tau)$), dan en slechts dan als geldt $\mathbf{Pop} \models \mathbf{identifier}(\xi(\tau), \tau)$. Voor de formele definitie van $\mathbf{identifier}(r, \sigma)$ wordt verwezen naar [HOF93].

De operator ξ is gedefinieerd als

$$\xi(\tau) = \sigma_{C(\tau)} \underset{f \in \mathbf{Facts}(f)}{\triangleright\!\triangleleft} \left(\pi_{\phi_f{}^{-1} f}\right).$$

Hierbij staat $\phi_f{}^{-1} : f \to \Pi(f)$ voor de bijectie van een instantie associatie naar zijn pater familias. Deze bijectie kan worden afgeleid uit de objectstructuur door de inverse te nemen van de bijectie van de pater familias naar de bewuste instantie associatie. Die bijectie is gegeven of volgt uit de objectstructuur. De selectieconditie is gedefinieerd als

$$C(\tau) = \bigwedge_{p,q \in \bigcup \mathbf{Facts}(\tau) \setminus \tau, \mathbf{RelCl}(\{p,q\}) \neq \varnothing} p = q.$$

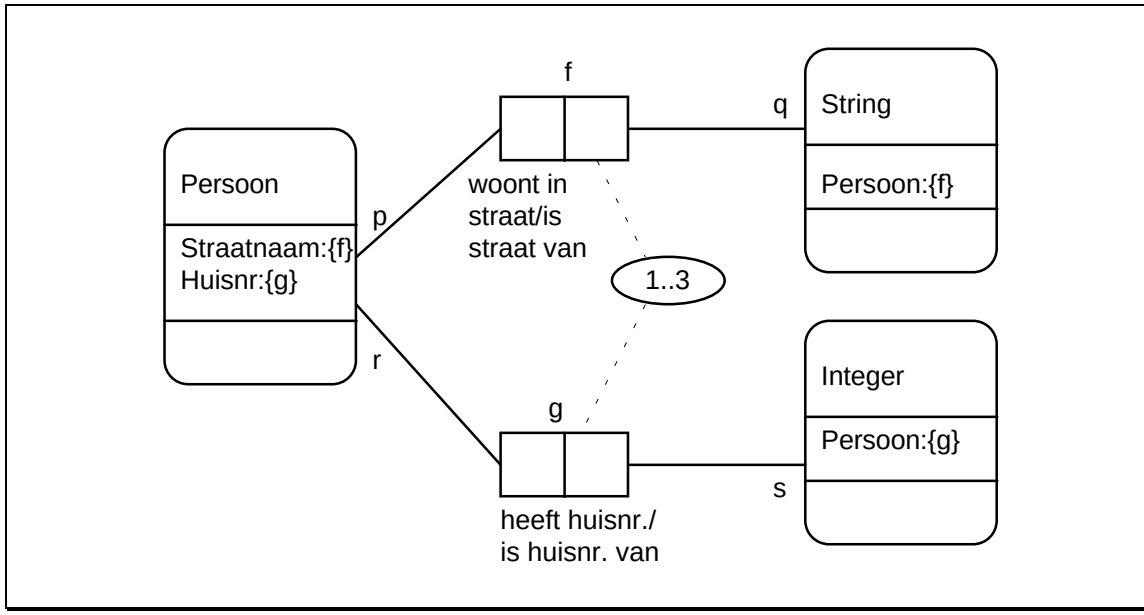
8.3.4 Occurrence frequency constraints

Met behulp van de in de vorige paragraaf gedefinieerde operator ξ kan nog een aantal andere constraints worden gedefinieerd.

De occurrence frequency constraint geeft aan, dat een gegeven combinatie van object instanties, die voorkomt in een verzameling predicatoren σ , minimaal *m* en maximaal *n* maal voorkomt. Dit wordt genoteerd als $\mathbf{frequency}(\sigma, m, n)$.

$\mathbf{Pop} \models \mathbf{frequency}(\sigma, m, n)$ geldt, dan en slechts dan als geldt

$$\mathbf{Pop} \models \neg\mathbf{IsEmpty}(\xi(\sigma)) \Rightarrow \begin{cases} \mathbf{Val}\left[\mathbf{min}\left(\chi(\xi(\sigma), \sigma, a), a\right)\right](\mathbf{Pop}) \geq m \\ \mathbf{Val}\left[\mathbf{max}\left(\chi(\xi(\sigma), \sigma, a), a\right)\right](\mathbf{Pop}) \leq n \end{cases}.$$



Figuur 8.5: Een voorbeeld van een occurrence frequency constraint.

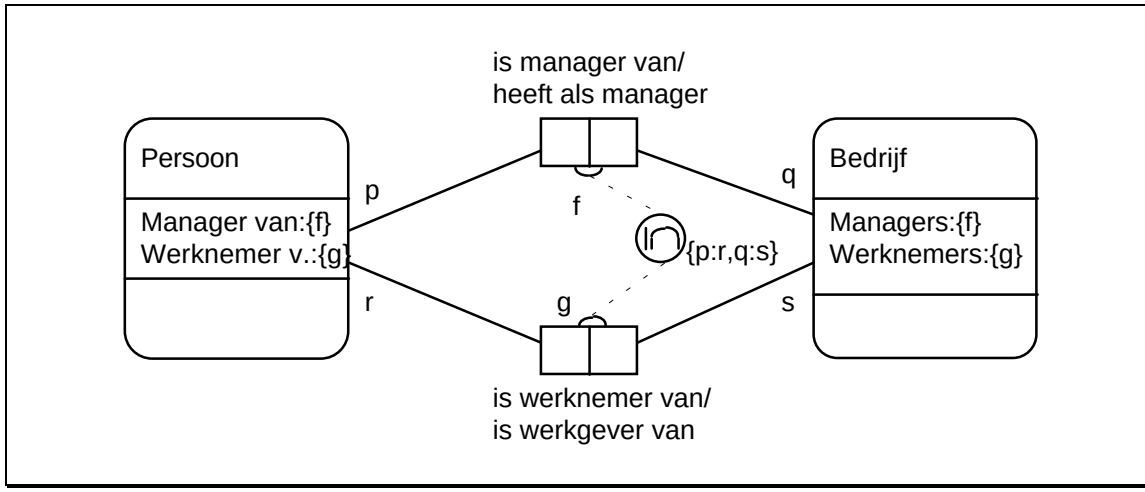
Voorbeeld 8.3.5

Figuur 8.5 is een voorbeeld van een occurrence frequency constraint. Een object instantie, die behoort tot de objectklasse Persoon, kan een attribuut Straatnaam hebben, gerepresenteerd door een object instantie, behorend tot de objectklasse String, en een attribuut Huisnr, gerepresenteerd door een object instantie, behorend tot de objectklasse Integer. In een straat kunnen meer personen wonen, meerdere personen kunnen hetzelfde huisnummer hebben en een persoon kan in meerdere straten wonen en meerdere huisnummers hebben (een persoon mag meerdere adressen hebben, bijvoorbeeld een postadres en een woonadres). Op ieder adres, ofwel iedere combinatie van straatnaam en huisnummer, moet een object instantie, behorend tot de objectklasse Persoon, wonen en op ieder adres mogen ten hoogste drie object instanties wonen, die behoren tot de objectklasse Persoon. Iedere combinatie Straatnaam-Huisnr moet dus minstens één en hoogstens drie keer voorkomen. Dit wordt grafisch genoteerd door de betrokken predicatoren te verbinden met een ellips met daarin de minimum en maximum waarde.

8.3.5 Set constraints

Tenslotte wordt een aantal set constraints gedefinieerd.

1. $\mathbf{Pop} \models \mathbf{subset}_{\phi}(\sigma,\tau) \Leftrightarrow \mathbf{Pop} \models \pi_{\rho(\sigma)}(\xi(\sigma)) \subseteq_{\phi} \pi_{\rho(\tau)}(\xi(\tau))$
2. $\mathbf{Pop} \models \mathbf{equal}_{\phi}(\sigma,\tau) \Leftrightarrow \mathbf{Pop} \models \pi_{\rho(\sigma)}(\xi(\sigma)) =_{\phi} \pi_{\rho(\tau)}(\xi(\tau))$
3. $\mathbf{Pop} \models \mathbf{exclusion}_{\phi}(\sigma,\tau) \Leftrightarrow \mathbf{Pop} \models \pi_{\rho(\sigma)}(\xi(\sigma)) \otimes_{\phi} \pi_{\rho(\tau)}(\xi(\tau))$



Figuur 8.6: Een voorbeeld van een subset constraint.

ϕ is een bijectie van $\rho(\sigma)$ naar $\rho(\tau)$, die mag worden weggelaten, als zij direct volgt uit de context. $\rho(\sigma)$ is de verzameling predicatoren

$$\left\{q \in \mathcal{P} \mid p \in \sigma \wedge \phi_{\mathbf{Fact}(p)}{}^{-1}(p) = q\right\}.$$

Hierbij staat $\phi_f{}^{-1}$**:** $f \rightarrow \Pi(f)$ voor de bijectie van een instantie associatie naar zijn pater familias.

Voorbeeld 8.3.6

Figuur 8.6 is een voorbeeld van een subset constraint. Een object instantie, die behoort tot de objectklasse Persoon, kan werknemer zijn van een object instantie, die behoort tot de objectklasse Bedrijf. Een object instantie, behorend tot de objectklasse Persoon, kan bovendien manager zijn van een object instantie, behorend tot de objectklasse Bedrijf. Als een persoon manager is in een bedrijf, dan is hij tevens werknemer van dat bedrijf. Dit kan worden gemodelleerd met behulp van een subset constraint. Iedere combinatie Persoon-Bedrijf, die voorkomt in de populatie van de instantie associatie 'is manager van/heeft als manager', moet tevens voorkomen in de populatie van de instantie associatie 'is werknemer van/is werkgever van'. Wij modelleren dit grafisch door de predicatoren van een verzameling met elkaar te verbinden en deze verbindingen op hun beurt te verbinden met een cirkel, waarin de bewuste operator (in dit geval $\subseteq$) is afgebeeld. De bijectie ϕ mag worden weggelaten, als zij afleidbaar is uit de context, hetgeen in dit voorbeeld het geval is. De overige set constraints werken analoog.

8.4 Identificatie

Zoals eerder is opgemerkt, wordt de basis van Object-Oriëntatie gevormd door objecten, door ons voor de duidelijkheid object instanties genoemd. Object instanties zijn in paragraaf 1.4.1 (Object instantie) gedefinieerd als "identificeerbare entiteiten, die een zichtbare rol spelen in het leveren van een dienst, waar een klant om verzoekt".

8.4.1 Object identiteit op machineniveau

In sectie 8.2 (Populaties) is de verzameling Θ, de verzameling van alle object instanties, geïntroduceerd. Iedere object instantie heeft zoals gezegd een eigen object identiteit, die de object instantie onderscheidt van iedere andere object instantie. Deze object identiteit is niet afhankelijk van de objectklasse, waartoe de object instantie behoort, omdat een object instantie te allen tijde haar identiteit behoudt, ook als zij migreert van de ene naar de andere objectklasse.

Daar in een populatie eigenschappen en gedrag van een object instantie afhangen van de klasse, waartoe de object instantie behoort, is de identiteit van een object instantie onafhankelijk van de populatie van de objectstructuur. De identiteit van een object instantie wordt in onze benadering gegeven door middel van een functie van Θ naar de eindige verzameling Ξ ($|\Theta| \leq |\Xi|$), de verzameling van object identiteiten. De injectieve functie $\mathbf{OID}: \Theta \rightarrow \Xi$ levert van iedere object instantie de bijbehorende object identiteit. De partiële hulpfunctie $\mathbf{Obj}: \Xi \mapsto \Theta$ is de inverse van **OID**.

Deze object identiteit is alleen zichtbaar op machineniveau en is daarom alleen van belang tijdens het technisch ontwerp en de implementatie van het te ontwikkelen systeem. Tijdens de analysefase wordt de eigen identiteit van een object instantie aanwezig verondersteld en niet in beschouwing genomen en als het systeem in bedrijf is, dan speelt de object identiteit een voor de eindgebruiker onzichtbare rol op machineniveau.

8.4.2 Object identiteit op gebruikersniveau

In de inleiding van deze sectie is reeds opgemerkt, dat het voor de eindgebruiker gemakkelijk kan zijn, dat hij object instanties kan identificeren op basis van eigenschappen. Hierbij moet met name worden gedacht aan object instanties, die behoren tot abstracte objectklassen, omdat bijvoorbeeld het onderscheid tussen twee integers met de waarde '3' voor de gebruiker van minder belang zal zijn. Voor concrete objectklassen geldt zelfs het tegendeel: object instanties met dezelfde concrete waarde mogen door de eindgebruiker niet van elkaar onderscheiden kunnen worden.

In deze paragraaf gaan wij nader in op zwakke identificatie in een objectstructuur. Wij introduceren daartoe de term structurele identificeerbaarheid. Een schema is structureel identificeerbaar ($\mathbf{StructId}(\Sigma)$), dan en slechts dan als iedere objectklasse in dat schema identificeerbaar is ($\forall_{k \in \mathbb{K}} [\mathbf{Identifiable}(k)]$).

Concrete objectklassen zijn identificeerbaar.

[ID1] $k \in \mathcal{CK} \vdash \mathbf{Identifiable}(k)$

De identificeerbaarheid van abstracte objectklassen kan worden afgeleid met behulp van de volgende regel.

[ID2] $k \in \mathcal{AK} \wedge \exists_{\tau \subseteq \mathcal{P}} [\mathbf{Identification}(k, \tau)] \vdash \mathbf{Identifiable}(k)$

Hierin geldt $\mathbf{Identification}(k, \tau)$, dan en slechts dan als geldt

1. $\mathbf{unique}(\tau) \in \mathcal{R}$

   De verzameling predicatoren τ fungeert als identificatie voor objectklasse *k*. Object instanties, behorend tot *k*, kunnen worden genoteerd door middel van een specifieke combinatie van waarden van τ. Daarom is het vereist, dat die combinatie van waarden van τ uniek is voor iedere object instantie, die behoort tot *k*. Dit wordt gegarandeerd, als τ een uniqueness constraint is.

2. $\forall_{p\in\tau}\exists!_{q\in\mathbf{Fact}(p)}\left[q\notin\tau\right]$

   Voor iedere predicator *p* in τ moet precies één predicator *q*, die geen deel uitmaakt van τ, behoren tot dezelfde instantie associatie. Deze unieke predicator wordt genoteerd als $\mathbf{co}(\tau,p)$. Objectklasse *k* moet de basis van deze predicator zijn: $\mathbf{Base}\left(\mathbf{co}(\tau,p)\right)=k$.

3. $\forall_{p\in\tau}\exists_{a\in\mathcal{A}}\exists_{k\in\mathfrak{K}}\left[\mathbf{co}(\tau,p)\in\mathbf{Pred}(a,k)\wedge\exists_{\sigma\subseteq P}\left[\begin{array}{l}\mathbf{Pred}(a,k)\subseteq\sigma\wedge\forall_{p,q\in\sigma}\left[\phi_1^{-1}(p)=\phi_2^{-1}(q)\right]\wedge\\ \mathbf{unique}(\sigma)\in\mathfrak{R}\wedge\mathbf{total}(\sigma)\in\mathfrak{R}\end{array}\right]\right]$

   Iedere object instantie, behorend tot *k*, moet precies één waarde hebben voor de attributen, die een rol spelen in de bij τ behorende instantie associaties. Daarom moet iedere copredicator deel uitmaken van een verzameling predicatoren σ, die zowel een uniqueness constraint als een total role constraint is. Alle predicatoren, die element zijn van σ, moeten bovendien op dezelfde predicator van de gemeenschappelijke pater familias van hun instantie associaties kunnen worden afgebeeld. ϕ_1 en ϕ_2 zijn de uit de objectstructuur afleidbare bijecties van een instantie associatie naar zijn pater familias.

4. $\forall_{p\in\tau}\left[\mathbf{Identifiable}\left(\mathbf{Base}(p)\right)\right]$

   De bases van de predicatoren in τ moeten identificeerbaar zijn.

Een objectklasse kan meerdere verzamelingen τ hebben, zodanig dat $\mathbf{Identification}(k,\tau)$. Om de notatie van object instanties te vergemakkelijken moet de keuze voor een verzameling attributen als identificatie van een objectklasse in de objectstructuur worden vastgelegd. Daar in feite niet de verzameling predicatoren τ zorgt voor de identificatie van objectklasse *k*, maar de bij de predicatoren behorende attributen daarvoor verantwoordelijk zijn, leggen wij niet de verzameling predicatoren zelf vast, maar de bijbehorende verzameling attributen. Om de notatie nog gemakkelijker te maken moet ook de volgorde van de attributen in die verzameling worden vastgelegd.

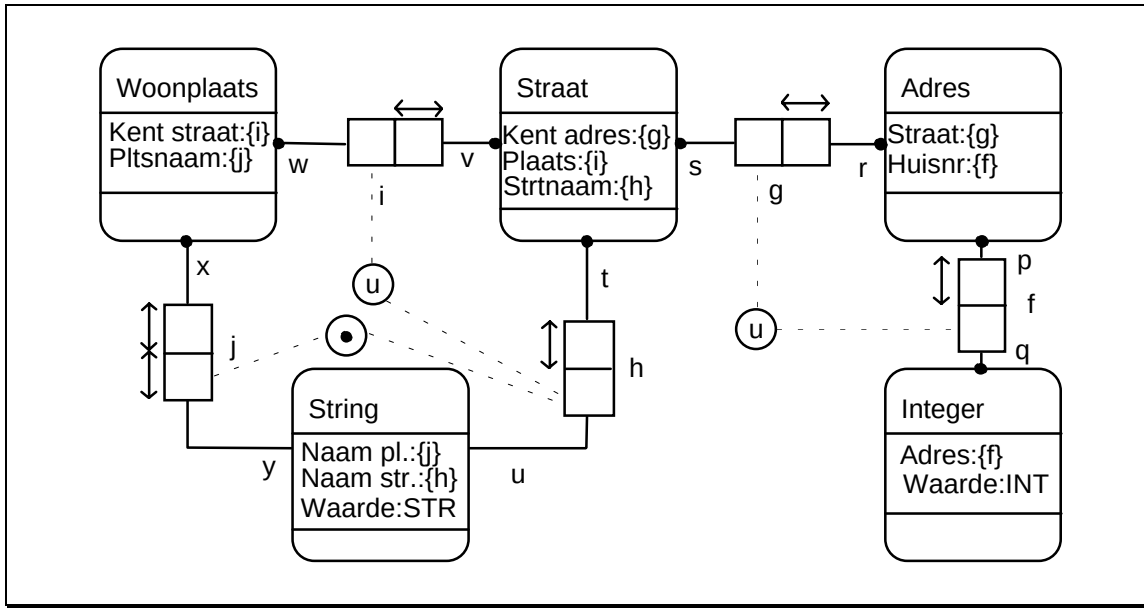
De partiële functie $\mathbf{Ident}:\mathcal{AK}\rightarrow\mathcal{A}^{+}$, die van nu af deel uitmaakt van ieder structureel identificeerbaar schema Σ, legt voor iedere abstracte objectklasse *k* de keuze van τ vast en legt bovendien de volgorde van de tot τ behorende attributen vast. Deze functie moet zodanig gedefinieerd zijn, dat bij de identificatie van iedere abstracte objectklasse *k* de verzameling

$$\left\{p\,|\exists_{i\in\{1,\ldots,|\mathbf{Ident}(k)|\}}\exists_{q\in\mathbf{Pred}\left(\mathbf{Ident}(k)[i],k\right)}\left[\mathbf{Fact}(p)=\mathbf{Fact}(q)\wedge p\neq q\right]\right\}$$

kan worden gekozen voor τ in regel **[ID2]**.

Voorbeeld 8.4.1

Figuur 8.7 is een voorbeeld van een structureel identificeerbaar schema. Alle concrete objectklassen (String en Integer) zijn betrokken in een total role constraint. De objectklasse Woonplaats kan worden geïdentificeerd aan de hand van $\tau=\{y\}$*, want y is uniek,* $\mathbf{co}(\tau,y)=x$*, x is uniek en totaal en de basis van y (String) is een concrete objectklasse en is dus identificeerbaar. De identificatie van Straat verloopt via* $\tau=\{u,w\}$*. Wederom geldt* $\mathbf{unique}(\tau)$*,* $\mathbf{co}(\tau,u)=t$ *en* $\mathbf{co}(\tau,w)=v$ *en zijn zowel t als v uniek en totaal, terwijl de bases van u (String) en w (Woonplaats) identificeerbaar zijn, zoals wij reeds hebben gezien. Op eenzelfde manier is de identificeerbaarheid van Adres af te leiden via* $\tau=\{q,s\}$.



Figuur 8.7: Een voorbeeld van een structureel identificeerbaar schema, ontleend aan [HOF93].

Voor dit schema kan de functie **Ident** *worden gedefinieerd als*

$\mathbf{Ident}(Adres) = \langle Straat, Huisnr \rangle$ $\mathbf{Ident}(Straat) = \langle Strtnaam, Plaats \rangle$

$\mathbf{Ident}(Woonplaats) = \langle Pltsnaam \rangle$

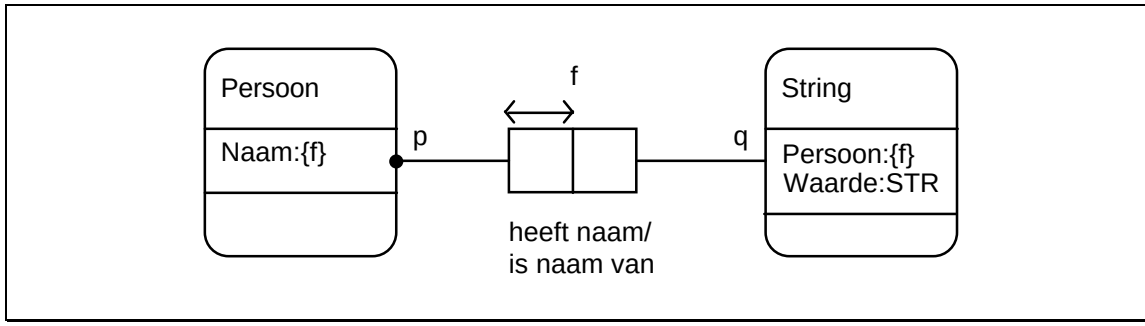
Hierdoor kunnen adressen worden genoteerd in de vorm $\langle String, String, Integer \rangle$*, waarbij de eerste String de woonplaats en de tweede String de straatnaam representeert.*

Om de denotatie van instantie associaties te vergemakkelijken, breiden we het domein van de functie **Ident** uit met de verzameling $\mathcal{T}$. Voor iedere instantie associatie geldt $\mathbf{set}(\mathbf{Ident}(f)) = f$. Zo kunnen instantie associaties worden genoteerd, zonder dat de predicatoren moeten worden vermeld, omdat die volgen uit de tekstuele volgorde.

Bij Conceptuele Modellering kan worden bewezen, dat uit de structurele identificeerbaarheid van een schema de zwakke identificeerbaarheid van een schema volgt [HOF93]. Bij Object-georiënteerde modellering is dat niet het geval. Daar object instanties van concrete objectklassen dezelfde waarde mogen hebben, zijn populaties mogelijk, waarin abstracte objectklassen, die zijn geassocieerd met verschillende object instanties, toch dezelfde eigenschappen hebben.

Voorbeeld 8.4.2

Het schema in figuur 8.8 is structureel identificeerbaar, met $\mathbf{Ident}(Persoon) = \langle Naam \rangle$*. Het schema sluit echter niet uit, dat wij twee personen invoeren , die zijn geassocieerd met verschillende object instanties, die behoren tot de objectklasse String, met een gelijke concrete waarde. Met andere woorden, twee object instanties 'a' en 'b', behorend tot de objectklasse Persoon, kunnen worden geïdentificeerd door twee verschillende object instanties 'x' en 'y', behorend tot de objectklasse String, maar toch beide de naam 'Jan' dragen. Structurele identificatie is daarom in de Object-georiënteerde benadering geen garantie voor zwakke identificatie. Als zwakke identificatie*



Figuur 8.8: Een voorbeeld van een structureel identificeerbaar schema.

vereist is, dan moet dit worden afgedwongen met behulp van extra controles op de waarden van de object instanties van concrete objectklassen, die voor die zwakke identificatie moeten zorgen.

In sommige gevallen is zwakke identificatie zelfs niet gewenst. In bijvoorbeeld PSM (zie [HOF93]) wordt een verzameling geïdentificeerd door zijn elementen. Twee verzamelingen met dezelfde elementen zijn dus per definitie dezelfde verzameling. Er kunnen zich echter situaties voordoen, waarin twee verzamelingen toevallig tijdelijk dezelfde elementen hebben, maar toch verschillende verzamelingen zijn.

Voorbeeld 8.4.3

Het bestuur van een studievereniging wordt beschouwd als een verzameling studenten. Ook practicumgroepen voor een bepaald vak worden beschouwd als verzamelingen studenten. Als het bestuur van een studievereniging besluit om samen een practicumgroep te vormen, dan is er naar onze mening geen sprake van één verzameling, die twee verschillende rollen speelt, maar eerder van twee verzamelingen, die tijdelijk dezelfde elementen hebben. Wanneer één van de studenten stopt met het vak, verdwijnt hij uit de practicumgroep, maar daaruit volgt niet, dat hij ook uit het bestuur van de studievereniging verdwijnt. Zwakke identificatie is naar onze mening in deze situatie niet gewenst.

Bij datamodellering wordt dit opgelost door een code te creëren, waardoor de verzamelingen worden geïdentificeerd. Object-Oriëntatie biedt een elegantere oplossing. Op machineniveau is sprake van twee verschillende verzamelingen, met verschillende object identiteiten, terwijl op gebruikersniveau geen onderscheid wordt gemaakt tussen de twee verzamelingen.

Als zwakke identificatie op gebruikersniveau gewenst blijkt te zijn, dan is een extra controle noodzakelijk op de waarden van object instanties, behorend tot concrete objectklassen, die object instanties, die behoren tot dezelfde abstracte objectklasse, identificeren. Hiertoe kan bijvoorbeeld een extra uniqueness constraint $\mathbf{value\_unique}(\tau)$ worden geïntroduceerd. In deze scriptie zullen wij een dergelijke constraint niet in detail uitwerken.

Samenvatting en conclusies

In deze scriptie is een aantal aspecten van het begrip Object-Oriëntatie behandeld. In hoofdstuk 1 is de Object-georiënteerde benadering gedefinieerd als die benadering van het ontwikkelen van geautomatiseerde systemen, die wordt gekenmerkt door:

- de modellering van de werkelijkheid als een verzameling object instanties, die met elkaar kunnen communiceren door het sturen van boodschappen
- de inkapseling van data (attributen) en de daarop werkende operaties in object instanties
- de groepering van object instanties met vergelijkbare eigenschappen (attributen) en een vergelijkbaar gedrag (operaties) in objectklassen
- de aanwezigheid van overerving en polymorfisme

In het vervolg van deel I zijn de voor- en nadelen van Object-Oriëntatie behandeld, alsmede de gevaren, die een onderneming loopt, wanneer slordig met Object-georiënteerde technologie wordt omgesprongen, en is aandacht besteed aan managementtechnieken, Object-georiënteerde methoden, Object-georiënteerde programmeertalen, Object-georiënteerde database management systemen, Object-georiënteerde tools, de standaardisatie organisatie OMG en tot nu toe met Object-Oriëntatie behaalde resultaten.

In deel II is ingegaan op de invoering van Object-Oriëntatie in een onderneming. Daarbij zijn de kritieke succesfactoren behandeld en is een methode gepresenteerd om die kritieke succesfactoren in kaart te brengen. Daarnaast is een aantal richtlijnen gegeven voor de invoering van Object-Oriëntatie in een onderneming.

In deel III is een formalisatie van een onafhankelijk Object Model gegeven, alsmede een gedeeltelijke formalisatie van een Transactie Model, waarmee het Object Model kan worden gemanipuleerd.

Op basis van het in deze scriptie beschreven onderzoek menen wij te mogen concluderen, dat Object-georiënteerde systeemontwikkeling geen wondermiddel is, dat de software crisis zal oplossen. Ook is Object-Oriëntatie niets nieuws. Object-Oriëntatie combineert een aantal al lang bestaande concepten, te weten abstracte datatypen, afscherming van informatie en overerving van functionaliteit. Deze concepten worden al geruime tijd toegepast bij het ontwikkelen van geautomatiseerde systemen, maar dan wel vrijblijvend.

Object-Oriëntatie dwingt het gebruik van de bovenstaande concepten af. Wanneer Object-Oriëntatie goed wordt toegepast, zal dat leiden tot systemen met een hogere kwaliteit en met een hoger herbruikbaarheidsgehalte. Maar voor die kwaliteit en voor dat herbruikbaarheidsgehalte blijft uiteindelijk de systeemontwikkelaar verantwoordelijk.

Object-Oriëntatie is ook niet een benadering, die achteloos kan worden ingevoerd. De invoering van Object-Oriëntatie gaat gepaard met hoge kosten en vereist veel veranderingen in de organisatie van de automatisering en moet daarom goed worden voorbereid. Object-Oriëntatie kan ook niet in iedere onderneming zomaar worden ingevoerd. De slaagkans van de invoering van Object-Oriëntatie is afhankelijk van een aantal factoren, waaraan niet iedere onderneming zal voldoen. Als met dit alles rekening wordt gehouden, dan kan Object-Oriëntatie een goede stap in de richting van het einde van de software crisis zijn.

Daarnaast is Object-Oriëntatie nog niet volwassen. De problemen, die wij hebben ondervonden bij het opstellen van een formeel objectmodel en een formeel dynamisch model, tonen dat overduidelijk aan. Er bestaan nog teveel verschillende meningen over de betekenis, die aan een aantal begrippen moet worden toegekend, en er bestaan nog teveel onduidelijkheden met betrekking tot de werking van een aantal mechanismes. Wij hopen, dat het door ons opgestelde model bijdraagt aan de oplossing van een aantal van

deze problemen. Wij beseffen echter terdege, dat wij nog lang niet alle problemen hebben onderkend, laat staan opgelost.

Het is ons helaas niet gelukt om uitspraken te doen over de betere aansluiting van Object-Oriëntatie op de wijze, waarop mensen denken. Dit gegeven wordt door veel auteurs opgemerkt, maar wordt door geen enkele auteur onderbouwd. Pogingen van ons om een dergelijke stelling te onderbouwen of te weerleggen hebben niet geleid tot aansprekende resultaten. Er bestaan raakvlakken tussen Object-Oriëntatie en een aantal binnen de psychologie gehanteerde modellen, zoals semantische netwerken en frame-based modellen, maar er zijn ook grote verschillen. Met name willen wij in dit verband het ontbreken van enige vorm van dynamiek in cognitieve modellen noemen. Maar al waren dergelijke raakvlakken wel nadrukkelijk aanwezig geweest, dan nog was daarmee geen bewijs geleverd voor de stelling, dat Object-Oriëntatie beter aansluit op de manier waarop mensen de wereld rondom hen beschouwen, laat staan voor de stelling, dat die betere aansluiting zou leiden tot een beter ontwikkelproces. Hopelijk kunnen anderen in de toekomst wel op wetenschappelijk onderzoek gebaseerde uitspraken over deze stellingen doen.

Bijlage A Verklaring van de wiskundige en grafische notaties

A.1 Verklaring van de wiskundige notaties

In dit deel van de bijlage wordt een aantal wiskundige notaties, die in deze scriptie worden gebruikt, kort uitgelegd. Dit deel van de bijlage is overgenomen uit [HOF93].

A.1.1 Functies

Een partiële functie f van A naar B is gedefinieerd als $f : A \mapsto B$. Formeel geldt $f \subseteq A \times B$, zodanig dat $\langle a,b\rangle \in f \wedge \langle a,c\rangle \in f \Rightarrow b = c$. De volgende afkortingen worden gebruikt.

$$\begin{aligned} f(a)\downarrow = &\quad \exists_{b\in B}\left[f(a) = b\right] \\ f(a)\uparrow = &\quad \neg f(a)\downarrow \\ \mathbf{dom}(f) = &\quad \left\{ a \in A \mid f(a)\downarrow \right\} \\ \mathbf{ran}(f) = &\quad \left\{ b \in B \mid \exists_{a\in A}\left[f(a) = b\right] \right\} \end{aligned}$$

Als f een partiële functie is, dan is $f \oplus \{a{:}b\}$ ook een partiële functie, die is gedefinieerd als

$$f \oplus \{a{:}b\} = \left\{ \langle a,b\rangle \right\} \cup \left\{ \langle x,y\rangle \in f \mid x \neq a \right\}.$$

De functie $f \oplus \{a{:}b\}$ gedraagt zich dus hetzelfde als de functie f. De enige uitzondering is a, waar de waarde b is.

Een totale functie f van A naar B is gedefinieerd als $f : A \to B$. Formeel is f een partiële functie van A naar B ($f : A \mapsto B$), zodanig dat f voor ieder element van A is gedefinieerd ($\mathbf{dom}(f) = A$).

Om veelvuldig gebruik van open- en sluithaken te voorkomen kan functiecompositie toegepast worden. De functie $f \cdot g$ is gedefinieerd als

$$f \circ g(x) = f\left(g(x)\right).$$

Het is natuurlijk wel vereist, dat $\mathbf{ran}(g) \subseteq \mathbf{dom}(f)$.

De functie $f\left[A'\right]$ is de functie f, beperkt tot subdomein $A' \subseteq \mathbf{dom}(f)$. Deze functie is gedefinieerd als

$$f\left[A'\right] = \left\{ \langle a,b\rangle \in f \mid a \in A' \right\}.$$

Om veelvuldig gebruik van tupelhaken ($\langle\ \rangle$) te voorkomen kan een afkorting voor het noteren van functies worden gebruikt. De functie f, gedefinieerd als $f = \left\{ \langle p,a\rangle, \langle q,b\rangle \right\}$, kan bijvoorbeeld worden genoteerd als $\{p{:}a, q{:}b\}$.

A.1.2 Verzamelingen en reeksen

De machtsverzameling van een verzameling A is de verzameling van alle deelverzamelingen van A en wordt genoteerd als $\wp(A)$. De verzameling van alle eindige reeksen met elementen uit A wordt genoteerd

als A^*, terwijl de verzameling van alle niet-lege eindige reeksen met elementen uit *A* wordt genoteerd als A^+.

Het *i*-de element uit een reeks $\langle a_1,..,a_i,..,a_n\rangle$, ofwel a_i, kan worden gevonden met behulp van de projectie.

$$\langle a_1,..,a_i,..,a_n\rangle[i] = a_i$$

De lengte van een reeks, ofwel het aantal elementen, kan op de volgende manier worden gevonden.

$$|\langle a_1,..,a_n\rangle| = n$$

Het hoofd van een niet-lege reeks $\langle a_1,..,a_n\rangle$, genoteerd als $\mathbf{head}(\langle a_1,..,a_n\rangle)$, is het eerste element van die reeks, ofwel a_i. De staart van een niet-lege reeks $\langle a_1,..,a_n\rangle$, genoteerd als $\mathbf{tail}(\langle a_1,..,a_n\rangle)$, is de reeks zonder haar hoofd, ofwel $\langle a_2,..,a_n\rangle$. Een reeks kan worden uitgebreid met een nieuw laatste element met behulp van de operator *.

$$\langle a_1,..,a_n\rangle * \langle a_{n+1}\rangle = \langle a_1,..,a_n,a_{n+1}\rangle$$

Een reeks kan met behulp van de functie **set** naar een verzameling worden geconverteerd.

$$\mathbf{set}(\langle a_1,..,a_n\rangle) = \{a_1,..,a_n\}$$

A is een echte deelverzameling van *B*, dan en slechts dan als *A* ongelijk is aan *B* en *A* een deelverzameling is van *B*.

$$A \subset B = A \subseteq B \wedge A \neq B$$

Het verschil tussen twee verzamelingen *A* en *B*, genoteerd als $A \setminus B$, is die verzameling, die de elementen van *A* bevat, die niet voorkomen in B.

$$A \setminus B = \{a \in A | a \notin B\}$$

A.1.3 Relationele algebra

Als *r* en *s* compatibele relationele expressies zijn ($\mathbf{Schema}(r) = \mathbf{Schema}(s)$), dan zijn de vereniging $r \cup s$ en het verschil $r \setminus s$ beide relationele expressies met $\mathbf{Schema}(r)$ en met de volgende populaties:

- $\mathbf{Val}[r \cup s](\mathbf{Pop}) = \mathbf{Val}[r](\mathbf{Pop}) \cup \mathbf{Val}[s](\mathbf{Pop})$
- $\mathbf{Val}[r \setminus s](\mathbf{Pop}) = \mathbf{Val}[r](\mathbf{Pop}) \setminus \mathbf{Val}[s](\mathbf{Pop})$

Als *r* en *s* relationele expressies zijn, dan is de join $r \triangleright\triangleleft s$ een relationele expressie, die is gedefinieerd als

1. $\mathbf{Schema}(r \triangleright\triangleleft s) = \mathbf{Schema}(r) \cup \mathbf{Schema}(s)$
2. $\mathbf{Val}[r \triangleright\triangleleft s](\mathbf{Pop}) = \{t | t[\mathbf{Schema}(r) \in \mathbf{Val}[r]](\mathbf{Pop}) \wedge t[\mathbf{Schema}(s) \in \mathbf{Val}[s]](\mathbf{Pop})\}$

Als *r* een relationele expressie is, $p_1,..,p_n$ en $q_1,..,q_n$ zijn predicatoren, alle $q_1,..,q_n$ zijn verschillend en $p_1,..,p_n \in \mathbf{Schema}(r)$, dan is de projectie $\pi_{q_1:p_1,..,q_n:p_n}(r)$ een relationele expressie, die is gedefinieerd als

1. $\mathbf{Schema}\left(\pi_{q_1:p_1,..,q_n:p_n}(r)\right) = \{q_1,..,q_n\}$
2. $\mathbf{Val}\left[\left[\pi_{q_1:p_1,..,q_n:p_n}(r)\right]\right](\mathbf{Pop}) = \left\{t \mid \exists_{s \in \mathbf{Val}[[r]](\mathbf{Pop})} \forall_{1 \le i \le n}\left[t(q_i) = s(p_1)\right]\right\}$

Voor de selectieoperator moeten de syntax en de semantiek van selectieformules worden gedefinieerd.

Definitie A.1.1

De verzameling Ψ bevat de selectieformules en is inductief gedefinieerd als de kleinste verzameling die voldoet aan:

1. $p,q \in \mathcal{P}' \Rightarrow p = q \in \Psi$
2. $p \in \mathcal{P}', c \in \bigcup D \Rightarrow p = c \in \Psi$
3. $f_1, f_2 \in \Psi \Rightarrow f_1 \wedge f_2 \in \Psi$
4. $f_1 \in \Psi \Rightarrow \neg f_1 \in \Psi$

Hierin is $\mathcal{P}'$ de verzameling van alle predicatoren, die kunnen voorkomen in objectstructuren. •

De volgende definitie definieert, wanneer een paar, dat voorkomt in de populatie van een relationele expressie, voldoet aan een selectieformule. Deze definitie maakt gebruik van de inductieve definitie van Ψ.

Definitie A.1.2

Als f een partiële functie is van $\mathcal{P}'$ naar Θ, dan geldt

1. $t \models p = q$ dan en slechts dan als $t(p) = t(q)$ en $t(p)\downarrow$ en $t(q)\downarrow$
2. $t \models p = c$ dan en slechts dan als $t(p) = c$ en $t(p)\downarrow$
3. $t \models f_1 \wedge f_2$ dan en slechts dan als $t \models f_1$ en $t \models f_2$
4. $t \models \neg f_1$ dan en slechts dan als niet $t \models f_1$ •

Als *r* een relationele expressie is en *F* is een selectieformule ($F \in \Psi$), dan is de selectie $\sigma_F(r)$ een relationele expressie, die is gedefinieerd als

1. $\mathbf{Schema}\left(\sigma_F(r)\right) = \mathbf{Schema}(r)$
2. $\mathbf{Val}\left[\left[\sigma_F(r)\right]\right](\mathbf{Pop}) = \left\{t \in \mathbf{Val}\left[\left[r\right]\right](\mathbf{Pop}) \mid t \models F\right\}$

De extensieoperator χ breidt een relationele expressie *r* uit met een nieuwe predicator *a* ($a \notin \mathbf{Schema}(r)$) en telt het aantal paren met een gelijke vulling van τ ($\tau \subseteq \mathbf{Schema}(r)$) op de volgende manier:

1. $\mathbf{Schema}\left(\chi(r,\tau,a)\right) = \mathbf{Schema}(r) \cup \{a\}$
2. $\mathbf{Val}\left[\left[\chi(r,\tau,a)\right]\right](\mathbf{Pop}) =$
   $\left\{t \mid t\left[\mathbf{Schema}(r)\right] \in \mathbf{Val}\left[\left[r\right]\right](\mathbf{Pop}) \wedge t(a) = \left|\left\{s \in \mathbf{Val}\left[\left[r\right]\right](\mathbf{Pop}) \mid t[\tau] = s[\tau]\right\}\right|\right\}$

De test **IsEmpty** geeft aan, of een relationele expressie een lege populatie heeft.

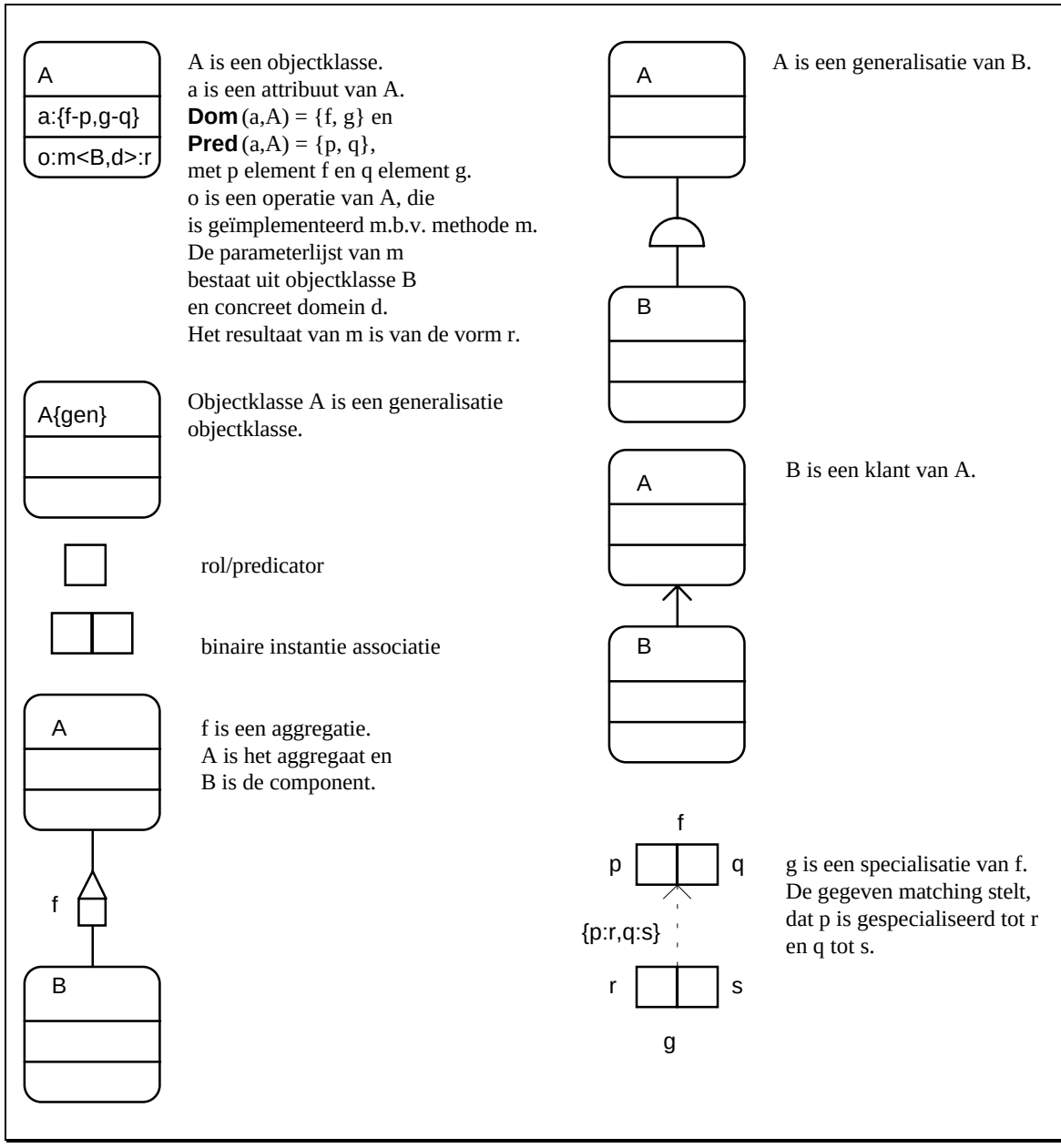
$$\mathbf{Pop} \models \mathbf{IsEmpty}(r) \text{ dan en slechts dan als } \mathbf{Val}\left[\left[r\right]\right](\mathbf{Pop}) = \varnothing$$

Als *p* een predicator is, dan converteert de bereikoperator θ_p de populatie van een relationele expressie *r* met $\mathbf{Schema}(r) = \{p\}$ naar de verzameling van waarden, die worden aangenomen door deze predicator.

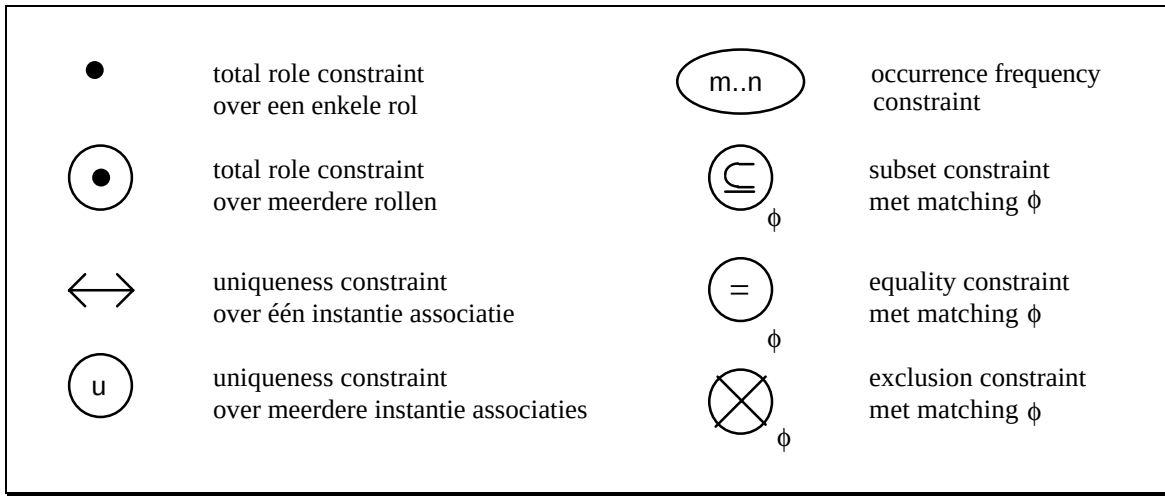
$$\mathbf{Val}\left[\left[\theta_p(r)\right]\right](\mathbf{Pop}) = \left\{t(p) \mid t \in \mathbf{Val}\left[\left[r\right]\right](\mathbf{Pop})\right\}$$

A.2 Verklaring van de grafische notatie

Dit deel van de bijlage bevat een overzicht van de grafische conventies van de in deze scriptie gebruikte notatie.



Figuur A.1: Constructiemechanismes voor Object Modellen.



Figuur A.2: Grafische constraints in Object Modellen.

Bibliografie

[AW91] D.E. Avison en A.T. Wood-Harper. "Information Systems Development Research: An Exploration of Ideas in Practice." *Computer Journal* 34(2):98-112, April 1991

[BC89] K. Beck en W. Cunningham. "A laboratory for teaching object-oriented thinking." *OOPSLA '89 Conference Proceedings (New Orleans, Louisiana, Oktober 1989), ACM SIGPLAN Notices* 24(10):1-6, Oktober 1989

[BEA93] K. Beam, Redactie. "Object Technology: Object-Oriented Programming." *IS/Analyzer* 31(12), Themanummer, December 1993

[BEM91] T.M.A. Bemelmans. *Bestuurlijke informatiesystemen en automatisering. Vierde, herziene druk,* Kluwer Bedrijfswetenschappen/Stenfert Kroese, Leiden, 1991

[BG90] D.M. Balda en D.A. Gustafson. "Cost estimation models for the reuse and prototype software development life-cycles." *ACM SIGSOFT Software Engineering Notes* 15(3):42-50, Juli 1990

[BOE81] B.W. Boehm. *Software Engineering Economics,* Prentice Hall, Englewood Cliffs, New Jersey, 1981

[BOO86] G. Booch. "Object-Oriented Development." *IEEE Transactions on Software Engineering* 12(2):211-221, Februari 1986

[BOO91] G. Booch. *Object-Oriented Design with Applications,* Benjamin-Cummings, Redwood City, California, 1991

[BOO94] G. Booch. *Object-Oriented Analysis and Design With Applications, Second Edition,* Benjamin-Cummings, Redwood City, California, 1994

[BRO93] G.H.W.M. Bronts. *Formalization of an Object Model,* Katholieke Universiteit Nijmegen, Faculteit der Wiskunde en Informatica, Afdeling Informatiesystemen, Afstudeerscriptie 283, Augustus 1993

[BS87] D. Benyon en S. Skidmore. "Towards a Tool Kit for the Systems Analyst." *Computer Journal* 30(1):2-7, 1987

[BS93] G.M. Barnes en B.R. Swim. "Inheriting software metrics." *Journal of Object-Oriented Programming* 6(7):27-34, November/December 1993

[CAB+93] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist en P. Jeremaes. *O-O Development: The FUSION Method,* Prentice Hall, Englewood Cliffs, New Jersey, 1993

[CCH+94] N.P. Capper, R.J. Colgate, J.C. Hunter en M.F. James. "The impact of object-oriented technology on software quality: Three case histories." *IBM Systems Journal* 33(1):131-157, 1994

[CF92] D. de Champeaux en P. Faure. "A comparative study of object-oriented analysis methods." *Journal of Object-Oriented Programming* 5(1):21-33, Maart 1992

[CN93] P. Coad en J. Nicola. *Object-Oriented Programming,* Prentice Hall, Englewood Cliffs, New Jersey, 1993

[COC93] A.A.R. Cockburn. "The impact of object-orientation on application development." *IBM Systems Journal* 32(3):420-444, 1993

[COM93] "Het praktisch nut van Object-Orientatie. Acht teams testen ontwikkelomgevingen uit tijdens evaluatieproject." *Computable* 26(27):11-13, 9 Juli 1993

[CY91a] P. Coad en E. Yourdon. *Object-Oriented Analysis, second Edition,* Prentice Hall, Englewood Cliffs, New Jersey, 1991

[CY91b] P. Coad en E. Yourdon. *Object-Oriented Design,* Prentice Hall, Englewood Cliffs, New Jersey, 1991

[DM93] J. Davis en T. Morgan. "Object-Oriented Development at Brooklyn Union Gas." *IEEE Software* 10(1):67-75, Januari 1993

[EKW92] D.W. Embley, B.D. Kurtz en S.N. Woodfield. *Object-Oriented Systems Analysis and Specification: A Model-Driven Approach,* Prentice Hall, Englewood Cliffs, New Jersey, 1992

[ES90] M.A. Ellis en B. Stroustrup. *The Annotated C++ Reference Manual,* Addison-Wesley, Reading, Massachusetts, 1990

[FHR+93] M.E. Fayad, L.J. Hawn, M.A. Roberts en J.R. Klatt. "Using the Shlaer-Mellor Object-Oriented Analysis Method." *IEEE Software* 10(2):43-52, Maart 1993

[GR89] A. Goldberg en D. Robson. *Smalltalk-80: The Language,* Addison-Wesley, Reading, Massachusetts, 1989

[HEN93] B. Henderson-Sellers. "The economics of reusing library classes." *Journal of Object-Oriented Programming* 6(4):43-50, Juli/Augustus 1993

[HH93] S. Henry en M. Humphrey. "Object-oriented vs. procedural languages: Effectiveness in program maintenance." *Journal of Object-Oriented Programming* 6(3):41-49, Juni 1993

[HK87] R. Hull en R. King. "Semantic Database Modelling: Survey, Applications and Research Issues." *Computing Surveys* 19(3):201-260, September 1987

[HOF93] A.H.M. ter Hofstede. *Information modelling in data intensive domains,* Academisch Proefschrift, Katholieke Universiteit Nijmegen, 1993

[HUM89] W.S. Humphrey. *Managing the Software Process,* Addison-Wesley, Reading, Massachusetts, 1989

[JCJ+92] I. Jacobson, M. Christerson, P. Jonsson en G. Øvergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach,* ACM Press/Addison-Wesley, Wokingham, England, 1992

[JON91] K. Jones. *The Benefits of CASE: Myths and Reality,* PEP Paper 20, Butler Cox, London, England, November 1991

[KG89] S.E. Keene en D. Gerson. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS,* Addison-Wesley, Reading, Massachusetts, 1989

[KM90] T. Korson en J.D. McGregor. "Understanding object-oriented: A unifying paradigm." *Communications of the ACM* 33(9):40-60, September 1990

[KRI93] G. Kristen. *KISS-methode voor Object Oriëntatie,* Academic Service, Schoonhoven, 1993

[LP90] W.R. LaLonde en J.R. Pugh. *Inside Smalltalk volume one,* Prentice Hall, Englewood Cliffs, New Jersey, 1990

[LP91] W.R. LaLonde en J.R. Pugh. *Inside Smalltalk volume two,* Prentice Hall, Englewood Cliffs, New Jersey, 1991

[LP93] W.R. LaLonde en J.R. Pugh. *Smalltalk in Action,* Prentice Hall, Englewood Cliffs, New Jersey, 1993

[LZ74] B. Liskov en S. Zilles. "Programming with abstract data types." *ACM SIGPLAN Notices* 9(4):50-59, April 1974

[MAR87] J. Martin. *Recommended Diagramming Standards for Analysts and Programmers: A Basis for Automation,* Prentice Hall, Englewood Cliffs, New Jersey, 1987

[MAR90a] J. Martin. *Information Engineering. Book I: Introduction,* Prentice Hall, Englewood Cliffs, New Jersey, 1990

[MAR90b] J. Martin. *Information Engineering. Book II: Planning and Analysis,* Prentice Hall, Englewood Cliffs, New Jersey, 1990

[MAR92] J. Martin. *Principles of Object-Oriented Analysis and Design,* Prentice Hall, Englewood Cliffs, New Jersey, 1992

[MEY88] B. Meyer. *Object-Oriented Software Construction,* Prentice Hall, Englewood Cliffs, New Jersey, 1988

[MEY91] B. Meyer. *Eiffel: The Language,* Prentice Hall, Englewood Cliffs, New Jersey, 1991

[MEY92] B. Meyer. "Applying 'Design by Contract'." *IEEE Computer* 25(10):40-51, Oktober 1992

[MO92] J. Martin en J.J. Odell. *Object-Oriented Analysis and Design,* Prentice Hall, Englewood Cliffs, New Jersey, 1992

[MP92] D.E. Monarchi en G.I. Puhr. "A Research Typology for Object-Oriented Analysis and Design." *Communications of the ACM* 35(9):35-47, September 1992

[NG74] R.L. Nolan en C.F. Gibson. "Managing the four stages of EDP growth." *Harvard Business Review*, Januari/Februari 1974, pp. 76-88

[NOL79] R.L. Nolan. "Managing the crisis in dataprocessing." *Harvard Business Review*, Maart/April 1979

[PAR72] D.L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM* 15(12):1053-1058, December 1972

[PAR76] D.L. Parnas. "On the Design and Development of Program Families." *IEEE Transactions on Software Engineering* 2(3):1-9, Maart 1976

[PIT93] M. Pittman. "Lessons learned in Managing Object-Oriented Development." *IEEE Software* 10(1):43-53, Januari 1993

[PRO94] C. Pronk. "Objectgeoriënteerde ontwerpmethoden en -technieken." In *Object Oriëntatie kritisch bekeken*, ASI Seminar (Amersfoort, juni 1994), Uitgeverij J.P. Schellekens, Rijswijk, 1994

[RBP+91] J.E. Rumbaugh, M.R. Blaha, W.J. Premerlani, F. Eddy en W. Lorensen. *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991

[RID94] Th. de Ridder. "Object Oriëntatie concepten in perspectief". In *Object Oriëntatie kritisch bekeken*, ASI Seminar (Amersfoort, juni 1994), Uitgeverij J.P. Schellekens, Rijswijk, 1994

[RUB91] H.A. Rubin. "Using 'readiness' to guide CASE implementation." *CASE Trends*, Januari/Februari 1991, pp. 37-41

[SAU89] J.H. Saunders. "A Survey of Object-Oriented Programming Languages." *Journal of Object-Oriented Programming* 1(6):5-11, Maart/April 1989

[SD93] K. Short en J. Dodd. *Information Engineering \with Objects, Issue 1.3. Texas Instruments Technical Paper*, Texas Instruments CASE Research Laboratory, JMA Information Engineering Ltd., Ashford, England, Juli 1993

[SIG92] "SIGS Publications. Happy 25th Anniversary Objects!" *Journal of Object-Oriented Programming* 5(6), Supplement, Oktober 1992. Met bijdragen van R. Kerr, K. Nygaard, B. Stroustrup, A. Kay en B. Meyer

[SM88] S. Shlaer en S.J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice Hall, Englewood Cliffs, New Jersey, 1988

[SM92] S. Shlaer en S.J. Mellor. *Object Lifecycles: Modeling the World in States*, Prentice Hall, Englewood Cliffs, New Jersey, 1992

[SM93] Sally Shlaer en Stephen J. Mellor. "A deeper look at managing the work on a project." *Journal of Object-Oriented Programming* 6(7):18-26, November/December 1993

[SNY93] A. Snyder. "The Essence of Objects: Concepts and Terms." *IEEE Software* 10(1):31-42, Januari 1993

[SOM89] I. Sommerville. *Software Engineering. Third Edition,* Addison-Wesley, Wokingham, England, 1989

[STR91] B. Stroustrup. *The C++ Programming Language. Second Edition,* Addison-Wesley, Reading, Massachusetts, 1991

[SZ92] J.F. Sowa en J.A. Zachman. "Extending and formalizing the framework for information systems architecture." *IBM Systems Journal* 31(3):590-616, 1992

[TAY92] D.A. Taylor. *Object-Oriented Information Systems: Planning and Implementation.* John Wiley & Sons, New York, New York, 1992

[VER93] Th.F. Verhoef. *Effective Information Modelling Support,* Academisch Proefschrift, Technische Universiteit Delft, 1993

[WES93] M. West. "Object-oriented Development: Strategies, Methodologies and Tools." Gartner Group Conference Presentation. In: *Sixth Annual Applications Development & Management Strategies Conference (Miami, Florida, 9-10 Juni 1993), Conference Book,* 1993

[WIL94] K.J. Williams. "Book Review: Object-Oriented Analysis and Design With Applications, 2nd Ed." *Journal of Object-Oriented Programming* 5(9):75,88, Februari 1994

[WJ92] R. Wieringa en W. de Jonge. *The identification of objects and roles - Object identifiers revisited,* Technisch Rapport IR-267 (versie 2), Faculteit der Wiskunde en Informatica, Vrije Universiteit, Amsterdam, Maart 1992

[WMH93] N. Wilde, P. Matthews en R. Huitt. "Maintaining Object-Oriented Software." *IEEE Software* 10(1):75-80, Januari 1993

[WWW90] R.J. Wirfs-Brock, B. Wilkerson en L. Wiener. *Designing Object-Oriented Software,* Prentice Hall, Englewood Cliffs, New Jersey, 1990

[YOU90] E. Yourdon. "Auld Lang Syne." *Byte,* Oktober 1990, pp. 257-264

[YOU94] E. Yourdon. *Object-Oriented Systems Design: An Integrated Approach,* Yourdon Press, 1994

Auteursindex

Index