Hogeschool Gent
Departement Toegepaste Ingenieurswetenschappen
Vakgroep Informatica
Academiejaar 2008-2009

Aanvullingen bij “Programmaontwerp en -realisatie”

Jan Cnops
12 oktober 2008

16.4 PRAKTISCHE INDELING

Voordat taken kunnen uitgevoerd worden, of zelfs maar gepland worden, moet worden bepaald *wie* ze zal uitvoeren. Daarvoor moet een bedrijf, of meer specifiek de projectmanager, uiteraard weten wie welke taak kan uitvoeren. Bovendien moet hij voor elke werknemer weten wanneer hij beschikbaar is: dit houdt in dat eventuele vakanties moeten gekend zijn, en dat voor elke werknemer moet geweten zijn of hij vol- of deeltijds werkt. Bovendien moet de betrokkenheid bij andere projecten worden ingecalculeerd.

Sommige taken worden door één persoon uitgevoerd, maar dikwijls wordt een taak aan een groep toegewezen. In wat volgt hebben we het over programmeurteams, maar het meeste geldt ook voor de andere activiteiten van een project: als we het zullen hebben over egoloos programmeren, dan bestaat er ook een egoloze analyse, en zo verder. Eerst moet opgemerkt worden dat het werken in groepen niet altijd een versnelde afwerking tot gevolg heeft. Dit is bijna zeker het geval als aan een taak *terwijl ze bezig is* mensen worden toegevoegd: dit leidt bijna steeds tot vertraging. Dit is de zogenaamde wet van Brooks. De nieuweling moet zich eerst inwerken in de taak voor hij productief wordt, en daarbij zal hij de anderen van hun werk afleiden.

De typische grootte van een groep is vijf personen, maar ze kan variëren van twee tot hoogstens acht, uitzonderlijk tien personen. Bij grotere groepen gaat er bijna automatisch een opsplitsing in deelgroepen ontstaan. Er zijn twee basismodellen voor groepsorganisatie, het democratische team en het hoofdprogrammeursteam.

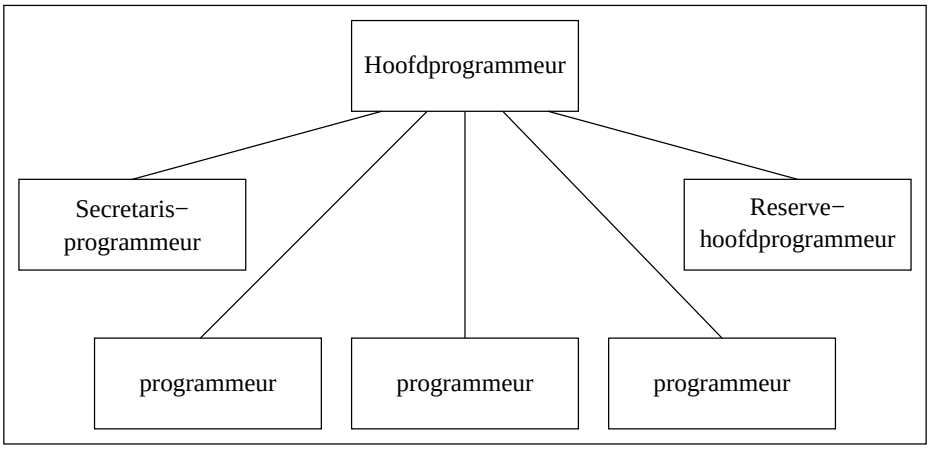
Bij het democratische team gaat men meestal uit van de noodzaak tot egoloos programmeren. Er is vastgesteld dat programmeurs die op hun eentje een stuk code hebben geschreven, daar zeer aan gehecht geraken. De code is als het ware een stukje van hun eigen ego geworden. Ze beschouwen dan ook elke kritiek op die code als kritiek op henzelf. Dit maakt het moeilijk om fouten verbeterd te krijgen: ofwel vindt de programmeur dat de fout niet bij ‘zijn’ code ligt, maar ergens anders, ofwel voelt hij zich gekwetst. Op deze manier wordt het moeilijk om vlot fouten te vinden en aanpassingen te laten doen. Bij egoloos programmeren wordt elke programmeur verondersteld de andere leden van het team aan te moedigen om fouten te vinden, en om dergelijke rapportering als een normaal, en zelfs positief onderdeel van het ontwikkelingsproces te bekijken. Merk op dat eXtreme Programming de ultieme consequentie trekt uit de idee van egoloos programmeren, en alle code eigendom maakt van de hele groep. Formeel is er in zo’n organisatie geen hiërarchie binnen de groep: beslissingen met betrekking tot interne organisatie, zoals taakverdeling, worden democratisch genomen. Wel is er natuurlijk buiten de groep een management dat de groep stuurt.

Het organiseren van het ontstaan van een democratisch team is vrij moeilijk. Meestal ontstaan ze vrij spontaan omdat de leden van een groep zich als gelijkwaardig be-

schouwen: als er duidelijke verschillen zijn in kennis en ervaring enerzijds, of in geldingsdrang anderzijds, dan ontstaan onevenwichten waarbij leiders naar voor komen en het democratisch karakter van de groep verdwijnt. Vandaar dat weinig bedrijven op een democratisch team mikken: het ergste voor de groepswerking zou zijn dat een persoon met weinig bekwaamheid en ervaring, maar veel geldingsdrang, het laken naar zich zou toe trekken. Dikwijls hebben democratische teams dus een formele leider aangeduid door het management. Dit kan tot problemen leiden. Zo is er het geval bekend van een democratisch team dat een zeer goed product afleverde. Het management besloot daarop de formele leider een geldige bonus toe te kennen. Deze weigerde de bonus voor hemzelf, met het argument dat het goede resultaat aan de groep lag, en dat de bonus over de leden moest verdeeld worden. Toen hij gedwongen werd de bonus toch te aanvaarden, verdeelde hij deze zelf onder de groepsleden. Uiteindelijk zette dit gebeuren zoveel kwaad bloed dat de hele groep ontslag nam en gezamenlijk bij de concurrentie ging werken.

Het blijkt dat democratische teams zeer productief zijn, vooral in omstandigheden waar er sprake is van veel innovatie en problemen die veel creativiteit vereisen. Bij meer klassiek werk is het hoofdprogrammeursteam meer aangewezen. Dit geldt ook als er een grote ongelijkheid is in ervaring en vaardigheid van de verschillende groepsleden, omdat de meest ervaren leden dan het meeste werk verzetten, en bij een democratische organisatie vinden dat de minder ervaren leden te veel eer krijgen voor het resultaat.

Zoals de naam het laat vermoeden is er bij een hoofdprogrammeursteam een hoofdprogrammeur. De volledige structuur is uitgetekend in Figuur 16.1. In de klassieke

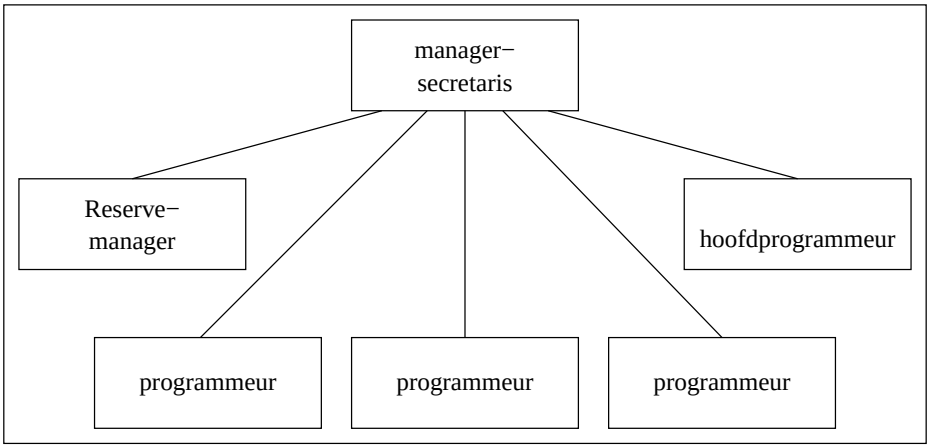


Figuur 16.1. Hoofdprogrammeursteam met zes leden.

vorm zijn er zes leden, waaronder drie gewone programmeurs. De functie van de hoofdprogrammeur is tweedelig. Ten eerste is hij de manager van de groep, en zorgt hij voor de praktische organisatie zoals taakverdeling. Maar hij dient ook een expert-programmeur te zijn, en van hem wordt verwacht dat hij leiding geeft bij het uittekenen van de software en moeilijke en belangrijke programmagedeeltes voor zijn rekening neemt. Omdat hij zo'n belangrijke plaats inneemt in het team, wordt een reserve voorzien. Deze vervangt de hoofdprogrammeur bij ziekte, afwezigheid wegens het volgen

van een cursus, en dergelijke meer. Daarnaast zorgt hij ook voor de speciale programmagedeelten, en neemt daardoor de hoofdprogrammeur werk uit handen. Een speciale rol is weggelegd voor de secretaris. Deze is verantwoordelijk voor de projectdocumentatie, en ook voor het beheer van de code (versiebeheer) en van het testen. Vroeger (het model van het hoofdprogrammeursteam werd eerst beschreven in 1971, toen er nog veel met ponskaarten gewerkt werd) zorgde hij ook voor compilatie van geschreven code; uiteraard is dat nu het werk van de afzonderlijke programmeurs. Merk op dat de job van secretaris ervaring en kennis vereist, en dat daarom de secretaris hoger staat in de hiërarchie dan een gewone programmeur.

Eén van de problemen met dit model is dat er wel erg veel verwacht wordt van de hoofdprogrammeur. Deze moet goede kwaliteiten bezitten op twee fronten: hoog niveau van programmeerkennis én hoog niveau van beheerskwaliteiten. Een mogelijke oplossing hiervoor is om de twee functies te scheiden. We krijgen dan een gewijzigd model, waarbij naast de hoofdprogrammeur, die verantwoordelijk is voor de technische leiding, een teammanager opduikt die de praktische zaken beheert. Nadeel van dit model is dat van de reserve: er is dan naast de reservehoofdprogrammeur een reserveteammanager nodig. Een mogelijke oplossing is dan om de teammanager ook secretaris te laten worden. Secretaris is een functie die dichter aansluit bij management dan bij programmeren. We krijgen dan het model van Figuur 16.2.



Figuur 16.2. Gewijzigd model, waar programmeur- en managertaken gescheiden zijn.

16.5 PLANNINGGEBASEERDE KOSTENSCHATTINGEN

Om het probleem van onnauwkeurige kostenschattingen te milderen maakt men vaak gebruik, voor één project, van verschillende ramingen. In totaal zijn er drie soorten kostenschattingen:

(1) Expertschattingen. Deze gaan uit, zoals vermeld, van opgedane ervaringen die men veralgemeent naar een nieuw project. Voordeel is dat ze goed zijn aangepast aan het bedrijf, nadeel is dat veralgemening naar een nieuw project altijd moeilijk is.

(2) Algoritmische schattingen. Zoals we gezien hebben zijn deze zonder aanpassing aan het bedrijf onbruikbaar. Wel bruikbaar zijn modellen, zoals COCOMO, die kunnen geijkt worden aan de voorgaande projecten.
(3) Planninggebaseerde schattingen, die we in deze paragraaf zullen bekijken.

Als men meervoudige kostenramingen gebruikt om de betrouwbaarheid te verhogen, dan kan het zijn dat men verschillende ramingen uit dezelfde categorie heeft (bijvoorbeeld verschillende experts die een schatting maken of verschillende algoritmes die elk een resultaat geven), of dat men verschillende categorieën mengt. In elk geval zal men de kostenramingen vergelijken: als deze goed parallel lopen gaat men ervan uit dat het resultaat betrouwbaar is. Als er grote discrepanties zijn gaat men niet direct een gemiddelde opmaken als schatting: men probeert de reden voor de verschillen te vinden, en te ontdekken of bepaalde schattingen niet kunnen verbeterd worden. Zo kan het zijn dat een expert een element dat in de planning voorkomt vergeten was en zo een te lage schatting geeft, maar het kan ook zijn dat hij een element dat in de planning vergeten was zelf wel meerekende, en dat dus de planning wordt aangepast.

Planninggebaseerde schattingen zijn in principe zeer eenvoudig: aan elke gedefinieerde taak moet men hulpbronnen toekennen, men bepaalt de prijs van deze bronnen, en telt alles op. Toch zijn er een paar addertjes onder het gras.

Ten eerste is het zo dat men, als men een risicoanalyse wil uitvoeren, moet weten welk gedeelte van de geschatte kosten reeds gemaakt is. Wat men moet weten is dus, en dit op elk moment:

(1) Welke kosten zijn er in realiteit reeds gemaakt? Dit bedrag is men kwijt, en moet men niet opnemen in de berekening van het risico.
(2) Welke kosten die er geschat waren moet men uit de schatting halen?
(3) Welke kosten uit de schatting blijven er zitten?

De bedragen uit (1) en (2) kunnen verschillen: zo is het best mogelijk dat een taak die uitgevoerd is geschat was op €750, maar uiteindelijk €1.000 gekost heeft. Het is belangrijk een zicht te hebben op deze verschillen. Ten eerste in het kader van kwaliteitscontrole, waarbij men de vraag stelt of men de schattingsmethode die de fout opleverde kan verbeteren. Maar zeer vaak duidt kostenoverschrijding erop dat de complexiteit van de taak onderschat was, en dit wijst er dan misschien op dat een taak die van deze eerste afhangt zelf ook onderschat is. Als bijvoorbeeld het ontwerp van een module uitloopt, is het waarschijnlijk zo dat ook het programmeren ervan langer zal duren, en dus duurder zal zijn, dan verwacht. Nemen we een numeriek voorbeeld. De oorspronkelijke schatting was €30.000, waarvan €10.000 voor ondertussen afgewerkte taken. In realiteit hebben deze echter €15.000 gekost. Dit leidt tot een herziening van de kostenschatting: €38.000, waarvan €23.000 nog moet besteed worden. Het is deze €23.000 die men gaat vergelijken met de verwachte opbrengst, en niet de totale kost van €38.000. Immers, stoppen met het project zou maar €23.000 besparen.

Men moet dus een idee hebben van *wanneer* kosten gemaakt worden. Als het gaat over de aankoop van een bepaald toestel, dan mag men (meestal) aannemen dat dit in het begin van een taak gebeurt. Inderdaad, zelfs als men het project halverwege de taak laat vallen, dan heeft men deze kosten gemaakt, en kan deze (waarschijnlijk) niet recupereren. Hetzelfde geldt dikwijls als men een taak uitbesteedt. Men sluit een contract af met een derde partij, en men kan daar niet op terug komen. Personeels-

kosten daarentegen zijn meestal evenredig verdeeld over een project. Verder kan het tijdstip belangrijk zijn omdat een bepaalde kost varieert in de tijd. Prijzen stijgen of dalen, en dus kan dezelfde hulpbron op verschillende ogenblikken verschillende prijzen hebben. Een typisch voorbeeld hiervan (alhoewel het niet echt veel uitmaakt bij softwareprojecten) is elektriciteit, met zijn nacht- en dagtarieven.

Een laatste belangrijke probleem is dat van de over- en onderbevraging van hulpbronnen. Bij overbevraging gaat men een hulpbron vragen meer werk te leveren dan ze aankan. Hardware is in dit opzicht weinig flexibel: als bijvoorbeeld een server te zwaar belast wordt zal hij snel ontoelaatbaar traag worden. Bij personen heeft overbevraging twee gevolgen. Het eerste is een toename van de prijs in vorm van overuren[1]. Het tweede is daling van de efficiëntie van het personeelslid door vermoeidheid en stress. Bij onderbevraging doet de hulpbron minder dan ze aankan, waardoor de prijs per geleverde prestatie toeneemt. Daar er vaak meerdere projecten tegelijk lopen, is het soms moeilijk om deze extra kosten te alloceren. Als een personeelslid 50% van de tijd met project 1 bezig is, en helemaal niet met project 2, dan is het de helft van de tijd technisch werkloos. Maar moet deze kost doorgerekend worden aan project 1 of aan project 2? Men kan dit oplossen door een *flexibiliteitscoëfficiënt* in te voeren. Projecten met een flexibele planning, waardoor taken toegewezen kunnen worden aan andere personen, of waar er veel speling is bij het wanneer dat een taak dient te worden uitgevoerd, zijn minder verantwoordelijk voor over- en onderbevraging dan andere. Daarom dienen ze minder bij te dragen aan de daardoor veroorzaakte kosten.

Tot slot van deze inleiding nog twee opmerkingen die het resultaat van een kostenraming kunnen beïnvloeden.

- *De wet van Parkinson.* Deze zegt dat werk altijd zover uitdeit dat het alle beschikbare tijd en werkkracht inneemt. Als een project geschat wordt op 12 mensmaanden, zal het nooit af geraken in minder dan 12 mensmaanden. Dit wordt soms gebruikt door projectmanagers die de kosten willen beperken. Hiertoe maken ze een “officiële” schatting die lager ligt dan de eigenlijke schatting, om iedereen ertoe aan te zetten een beetje harder te werken. Te ver gaan daarin leidt tot schattingen en tijdschema’s die duidelijk niet haalbaar zijn en demotiverend werken, wat een tegengesteld effect sorteert.
- *Pricing to win.* Met deze uitdrukking wordt bedoeld dat men een schatting maakt die het meeste winst oplevert. Men schat in wat de klant bereid is om te betalen, en maakt dit de geschatte kostprijs van het project. Men vind in de literatuur weinig informatie over hoe dit best gebeurt.

16.6 COCOMO

Zoals we gezien hebben is het grote probleem bij kostenschattingen dat algemene methodes onnauwkeurig zijn, met foutenmarges gaande tot een factor 8, en dat men bij schattingen gebaseerd op concrete ervaring binnen het bedrijf het probleem heeft dat een nieuw project uiteraard anders is dan alle voorgaande, en dus een onnauwkeurige

[1] In sommige firma’s is er een directe kostenbesparing omdat men verwacht dat het personeel onbetaalde overuren maakt. Hierdoor wordt het echter moeilijker personeel aan te trekken, wat dan weer leidt tot een verhoogde loonkost.

schatting oplevert. Men heeft gepoogd om de voordelen van beide methodes te behouden en de nadelen te vermijden door metrieken te *parametriseren*. Hierbij gebruikt men een bepaalde formule, maar deze formule heeft parameters. Door nu deze formule toe te passen op projecten uit het verleden, kan men zien of men de juiste waarden voor de parameters gebruikt, en deze dan eventueel aanpassen. De bekendste van deze modellen zijn de twee COCOMO's. COCOMO staat voor **CO**nstructive **CO**st **MO**del[2]. Het eerste model was COCOMO 81 (dit slaat op het jaartal), de nieuwere versie is COCOMO II, uit het jaar 2000. In wat volgt geven we bij alle formules de standaardwaarden voor de parameters zoals die gegeven worden door het COCOMO. Dit zijn gemiddelden die gehaald zijn uit de gegevens van enkele tientallen tot enkele honderden softwareprojecten. Deze geven idee van welke grootteorde de parameters zijn, en kunnen door beginnende organisaties gebruikt worden; het resultaat is dan echter zeer onnauwkeurig. Bedoeling van de modellen is van deze toe te passen op voorafgaande projecten, en de parameters aan te passen tot overeenstemming is bereikt.

16.6.1 COCOMO 81

COCOMO 81 is, zeker in een moderne ontwikkelomgeving, gewoonweg niet bruikbaar, maar staat wel toe om een aantal fundamentele eigenschappen uit te leggen. Er zijn drie varianten, en alledrie maken ze gebruik van dezelfde metriek: KDSI, wat staat voor duizend lijnen code (Kilo Delivered Source code Instructions). De basisformule heeft altijd dezelfde vorm, en het doel is steeds een schatting te geven van het aantal mensmaanden dat nodig is om het project uit te voeren. Schattingen van andere kosten kunnen met dit model niet gemaakt worden. De invulling van de parameters van de basisformule en de aanpassingen van het resultaat zijn verschillend in de drie varianten. De varianten zijn

1. Basis-COCOMO (basic COCOMO). De gebruikte formule is van de vorm (MM staat voor mensmaanden).

   $$\text{MM} = \text{a} \times \text{KDSI}^k.$$

   Waar a en k afhangen van het soort project (zie verder). Basis-COCOMO heeft dus niet veel nut.
2. Tussen-COCOMO (intermediate COCOMO). De basisformule is dezelfde met, eigenaardig genoeg andere waarden voor a en k. Het resultaat van deze formule wordt dan nog vermenigvuldigd met zogenaamde *cost drivers* (zie verder).
2. Gevorderde COCOMO (advanced COCOMO). Ook hier worden cost drivers gebruikt, maar op een meer gesofisticeerde manier.

De parameters a en k hangen niet alleen van de gebruikte variant af, maar ook van de zogenaamde *ontwikkelingsmodus*. Verschillende projecten hebben verschillende karakteristieken, en die maken dat dezelfde formule niet altijd bruikbaar is. Er zijn drie verschillende ontwikkelingsmodi:

- Organische modus. Hier werken vrij kleine teams in een vertrouwde omgeving aan steeds gelijksoortige toepassingen. Hierdoor zijn er weinig communicatieproblemen en kan het personeel vlot zijn werk uitvoeren.

[2] Het is dus verkeerd om van het COCOMO-model te spreken; dat zou het constructive cost model model zijn.

- Halfopen[3] modus. Zit tussen de twee andere modi in. Wordt getypeerd door een mix van ervaren en onervaren personeel. Dit slaat zowel op ervaring in het algemeen, als ervaring met de specifiek te ontwikkelen toepassingen.
- Ingebedde modus. De projecten zijn ingebed in een complex geheel van bestaande software en van hardware, met strenge regulaties. Hierdoor vraagt ontwikkeling grondige kennis van dit geheel, en is er gevaar dat de resultaten van het project niet aan de strenge vereisten voldoen.

Welke modus toepasbaar is wordt bepaald aan de hand van een aantal criteria:

	Grootte	Innovatie	Condities	Omgeving
Organisch	klein	weinig	niet streng	stabiel
halfopen	gemiddeld	gemiddeld	gemiddeld	gemiddeld
ingebed	groot	groot	streng	complexe interfaces

Met condities worden randvoorwaarden bedoeld die aan de software worden opgelegd, zoals tijds- of geheugengebruik. Hiermee worden de parameters

	Basis-COCOMO	tussen-COCOMO
Organisch	$2{,}4 \times \text{KDSI}^{1{,}05}$	$3{,}2 \times \text{KDSI}^{1{,}05}$
half-onafhankelijk	$3{,}0 \times \text{KDSI}^{1{,}12}$	$3{,}0 \times \text{KDSI}^{1{,}12}$
ingebed	$3{,}6 \times \text{KDSI}^{1{,}20}$	$2{,}8 \times \text{KDSI}^{1{,}20}$

De cost drivers zijn factoren waarmee het resultaat in tussen-COCOMO moet worden vermenigvuldigd. Er zijn verschillende drivers die verschillende aspecten van een project en van de ontwikkelomgeving beschrijven. Voor elke driver geeft COCOMO een minimum- en een maximumwaarde. Nemen we bijvoorbeeld de driver 'ervaring met de programmeertaal'. Deze driver heeft een maximumwaarde van 1,14 en een minimumwaarde van 0,95. Als de programmeurs zeer goed vertrouwd zijn met de programmeertaal waarin ze werken kiest men de *kleinste* waarde, als ze de taal niet kennen de grootste. De spreidingsfactor is het quotiënt van maximum en minimum, in dit geval dus 1,14/0,95 = 1,20. Deze geeft aan hoeveel de cost driver de kost kan beïnvloeden. Voorbeelden van andere drivers, met hun spreidingsfactoren zijn:

- Complexiteit van het project. Spreidingsfactor 2,36.
- Bekwaamheid analisten. Spreidingsfactor 2,06.
- Bekwaamheid programmeurs. Spreidingsfactor 2,03.
- Vereiste betrouwbaarheid. Spreidingsfactor 1,87.
- Gebruik moderne programmeertechnieken. Spreidingsfactor 1,51.

De juiste waarden die aan de drivers moeten worden toegekend worden bepaald door experts. Het is geen toeval dat persoonlijke bekwaamheden een belangrijke rol spelen: deze vormen de belangrijkste bron van variabiliteit in de kosten. Men kan twee soorten drivers onderscheiden: deze voor omgevingsfactoren en deze voor projectfactoren. Bij omgevingsfactoren, zoals de bekwaamheid van personeel, veranderen de drivers niet (of niet veel) van het ene project naar het andere, bij projectfactoren is dit wel het geval. Deze laatste moeten dus elke keer weer opnieuw worden ingeschat door experts. Hierbij probeert men gebruik te maken van de ervaringen uit het verleden: men gaat bijvoorbeeld voor de factor 'complexiteit van het project' kijken naar een analoog

[3] Halfopen (semi-detached) zoals in halfopen bebouwing.

project, en neemt daarvan de driver over. Als het project duidelijk complexer is als alle voorgaande projecten, dan neemt men een grotere driver. Voor omgevingsfactoren gaat men gewoon de waarden uit vorige projecten overnemen.

Bij gevorderde COCOMO wordt de werkbelasting geschat, niet over het hele project, maar fase per fase in de ontwikkeling. Hierbij wordt zeer expliciet gebruik gemaakt van een cascadeontwikkelingsmodel. Wel werkt COCOMO hier met vier fasen in plaats van vijf. In de termen van COCOMO zijn de vier fasen dan

1. Behoefteplanning en productontwerp.
2. Gedetailleerd ontwerp.
3. Coderen en testen van modules.
4. Integratie en testen.

Voor elke driver en voor elke fase kan men een verschillende waarde invullen. Om een voorbeeld te geven: een van de drivers steunt op de vaardigheid van de analisten. Als deze zeer groot is levert dit een besparing op. Deze besparing is zeer groot bij de eerste fase (driver: 0,55), en kleiner bij de volgende fasen (driver: 0,75). Immers, ervaren analisten werken niet alleen beter maar ook sneller. In de eerste fase hebben beide aspecten invloed; in latere fase spelen alleen de voordelen van het werken op een betere analyse een rol.

Het gebruik van COCOMO vereist natuurlijk dat men de driverwaarden kent. Hierbij steunt men dan vaak op de driverwaarden voor projecten uit het verleden. Men gaat dus COCOMO toepassen op reeds afgewerkte projecten, en de parameters aanpassen tot de resultaten kloppen. Dit is een zeer fragiel proces, omdat er veel parameters zijn, en er een zekere ruis zit op de meetwaarden: zelfs projecten met zeer gelijkaardige factoren kunnen toch verschillende werkingskost hebben. Zeker als het aantal projecten uit het verleden klein is, is het probleem van het bepalen van de drivers niet eenduidig. Het gebruik van COCOMO 81 als standaard leverde echter informatie op uit een groot aantal bedrijven met veel projecten, zodat men criteria kon bepalen om de drivers te schatten. Zo kan men dan de driver voor bekwaamheid van de analisten gaan bepalen op basis van hun opleiding en hun werkervaring. Dit werkte vrij goed in de tijd dat er in de meeste bedrijven een zeer gelijkaardige omgeving was (administratieve projecten, COBOL als programmeertaal, nauwelijks CASE-tools, enzovoorts).

De voordelen van dit model zijn dat het eenvoudig is om te zien wat er gebeurt. Bovendien is het gebruik van cost drivers een goede hulp: wie dit model gebruikt krijgt meteen een lijst van factoren die de kosten kunnen beïnvloeden, samen met een schatting hoe zwaar elke factor kan doorwegen. De nadelen zijn echter veelvuldig:

- De kostenschatting steunt uiteindelijk op de schatting van KDSI. Dit schuift het probleem op: in plaats van de kosten te schatten met de natte vinger, moet men nu KDSI schatten met de natte vinger.
- Er zijn drie ontwikkelingsmodi, en het resultaat is zeer sterk afhankelijk van de ontwikkelingsmodus. Hierbij wordt de kostenraming *groter* als de ontwikkelingsmethode *meer* gesofisticeerd is. Zo kost een project met 10 KDSI volgens basis-COCOMO in organische modus 27 mensmaanden en in ingebedde modus 57, meer dan het dubbel. Dit is niet geheel onzinnig, omdat basis-COCOMO geen rekening houdt met de complexiteit van het project. Complexe projecten komen meer voor in gesofisticeerde ontwikkelingsomgevingen, en dus veronder-

stelt basis-COCOMO dat, als er een gesofisticeerde ontwikkelingsomgeving is, het project wel complex moet zijn. Het geeft echter meteen aan dat het basismodel slechts een zeer ruwe schatting geeft.

Merken we op dat COCOMO 81 ook een schatting geeft van de totale duur van een project. De formule voor de tijdsduur in maanden is

$$2,5 \times \mathrm{MM}^{a},$$

met a gaande van 0,38 (organisch) tot 0,32 (ingebed). Dit betekent dat als MM met een factor 8 toeneemt de tijdsduur verdubbelt (en er natuurlijk vier keer zoveel personeel moet worden ingezet. Deze formule is duidelijk niet geldig voor kleine waarden van MM (vul zelf maar eens MM=1 in). Om een idee te geven: een project van 100 mensmaanden zal iets minder dan een jaar duren, en dus gemiddeld 9 mensen vergen. Zoals vermeld is dit minder bij begin en einde, en meer in het midden.

16.6.2 COCOMO II

COCOMO 81 was bruikbaar in de homogene omgeving waarvoor het werd ontworpen, maar de ontwikkelingen in de informatica zorgden ervoor dat het model minder en minder nut had. Een aantal factoren:

- Het opduiken van niet-lineaire ontwikkelingsmodellen. Hierdoor vermindert de invloed van de complexiteit op de kosten.
- Het toenemen van de mate van hergebruik. Hergebruik van code geeft een heel andere kostenstructuur dan zelf schrijven van code.
- Door het invoeren van standaarden als CMM en ISO 9000 worden zeer grote besparingen verkregen, die onvoldoende worden doorgerekend in COCOMO 81.
- Objectgericht ontwerp en objectgerichte programmatie veranderen ook de kostenstructuur, met bijvoorbeeld minder kosten voor testen.
- Het gebruik van meer gesofisticeerde ontwikkelomgevingen, met IDE's en CASE-tools moet worden verrekend.
- Softwareprojecten zijn meer verscheiden geworden. Bijvoorbeeld is er het toegenomen belang van GUI's en webapplicaties. Deze hebben een heel eigen kostenstructuur.
- Toename van automatische codegeneratie, zoals bijvoorbeeld bij het maken van gebruikersschermen. Hierdoor wordt de waarde van KDSI als maat zeer twijfelachtig.

Een aantal van de bezwaren tegen COCOMO 81 in de nieuwe omstandigheden zijn opgevangen in COCOMO II.

Het eerste probleem dat COCOMO II aanpakte is dat van de basismaat. COCOMO 81 steunde op het aantal lijnen code, maar dit wordt minder en minder relevant. Meer en meer wordt code hergebruikt, en bovendien maakt men gebruik van automatische codegeneratoren. Moderne programmeertalen zijn krachtiger, maar daardoor ook moeilijker, dan oudere. Om een klein voorbeeld te geven nemen we twee C++-programmeurs waarvan er een de Standard Library gebruikt en de tweede niet. De tweede genereert meer code dan de eerste in gelijkaardige omstandigheden. Bovendien zal de eerste voor een gemiddelde lijn code meer tijd nodig hebben dan de tweede. Immers, wie de STL gebruikt, zal regelmatig moeten gaan opzoeken hoe de interface

daarvan in mekaar zit, terwijl de tweede programmeur meer elementaire opdrachten gebruikt. Maar zonder twijfel is de eerste programmeur productiever dan de tweede. Als we het werk van de analist, dat niet afhangt van het gebruik van de STL, moeten afmeten aan het aantal lijnen code dan krijgen we problemen. Bovendien zitten we nog altijd met het probleem dat we het aantal lijnen code niet exact weten voor de code geschreven is, en dat we dus dit aantal moeten schatten de hand van een aantal factoren.

16.6.3 SLOC

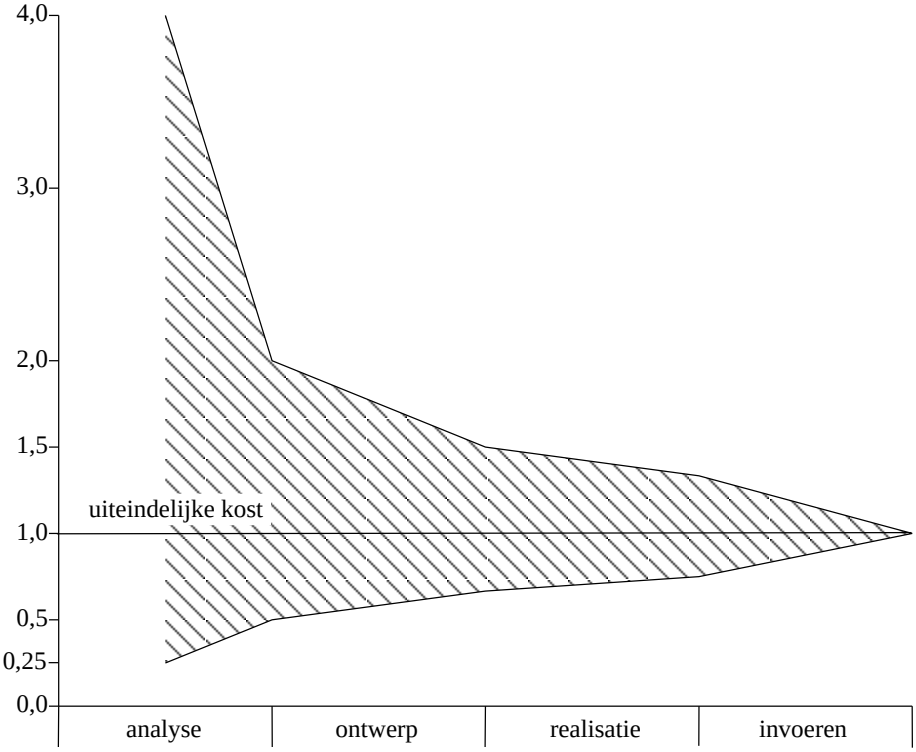
Toch gebruikt COCOMO II nog altijd lijnen code, maar in plaats van KDSI gaat het nu om KSLOC, Kilo Source LOgical Code. Men meet niet meer hoeveel lijnen een programma bevat, maar hoeveel logische eenheden. Zo is bijvoorbeeld een *if-then-else*-structuur 1 SLOC, zelfs al staat hij verspreid over verschillende lijnen. De reden voor deze beslissing is dat er verschillende methodieken zijn die leiden tot een schatting van het aantal SLOC. SLOC wordt op deze manier een abstracte eenheid: zo is het best mogelijk dat twee programmeertalen een verschillend aantal regels code opleveren voor een bepaald project, maar dat het aantal SLOC voor de twee talen hetzelfde is. Het is altijd mogelijk een of andere schattingsmethode te gebruiken om het aantal SLOC te berekenen, en dan COCOMO te gebruiken om een kostenschatting te bekomen. De meeste methodieken werken met een puntensysteem, zoals *functiepunten* (function points), *objectpunten* (object points) of *kenmerkpunten* (feature points). Objectpunten worden gegeven aan de hand van systeemvereisten, zoals het aantal gebruikersschermen, en kunnen dus zeer vroeg in het analyseproces al toegekend worden. Functiepunten en kenmerkpunten steunen gedeeltelijk op een aantal interne gegevens, zoals het aantal databestanden en het aantal gebruikte algoritmen. Deze kunnen pas toegekend worden nadat het ontwerp al gedeeltelijk gemaakt is. Hier komen we bij het tweede punt van COCOMO II. Naarmate het ontwikkelingsproces vordert, krijgen we meer gegevens binnen. Het spreekt vanzelf dat de laatst binnengekomen gegevens de meest nauwkeurige schattingen opleveren voor het vervolg. Volgens een studie neemt de betrouwbaarheid van kostenschattingen toe volgens Figuur 16.3. Het gearceerde gebied van deze figuur is het gebied waar kostenschattingen meestal in vallen.

Op het ogenblik dat de code geschreven is kunnen we ons baseren op KDSI, maar voor eerdere schattingen moeten we een aangepaste maat gebruiken. Op deze manier bestaat COCOMO II uiteindelijk uit drie algoritmes, die op verschillende momenten in de ontwikkeling gebruikt worden. Deze drie modellen zijn

1. het vroegeanalysemodel: te gebruiken in de allereerste fase van de ontwikkeling. Dit maakt gebruik van zeer algemene kenmerken van het systeem, in de vorm van zogenaamde *objectpunten* (zie verder).
2. Het vroegeontwerpmodel. De naam is duidelijk. Het model maakt gebruik van *functiepunten*.
3. Het post-architecturale model. Te gebruiken op het ogenblik dat de architectuur van de software volledig bekend is.

In het vroegeanalysemodel wordt gebruik gemaakt van de schaarse gegevens die beschikbaar zijn om een schatting te maken. Er zijn drie factoren die meetellen:

- Het aantal te verwachten schermen voor de gebruikersinterface.



Figuur 16.3. Mogelijke afwijking van schattingen.

- Het aantal te verwachten softwarerapporten.
- Het aantal te verwachten softwarecomponenten.

Elk van deze drie getallen worden vermenigvuldigd met een gewichtsfactor en dan opgeteld. Dit geeft het aantal objectpunten OP. Daarna houdt men rekening met hergebruik. Het aantal *nieuwe* objectpunten NOP wordt gedefinieerd door de hergebruiksfactor r, die aangeeft hoeveel OP er nieuw zijn:

$$\text{NOP} = \text{OP} - \text{r} \times \text{OP}.$$

Daarna bekijkt men de productiviteit van de organisatie. Deze wordt uitgedrukt in NOP/maand. De productiviteit wordt uitgedrukt in de vaardigheid en maturiteit van de organisatie (de C en M uit CMM) en de ervaring en vaardigheid van de ontwikkelaars. Men heeft vijf niveaus, net zoals bij CMM, en men kent deze de volgende productiviteit toe:

Niveau	productiviteit
Zeer laag	4
Laag	7
Gemiddeld	13
Hoog	25
Zeer hoog	50

.

Het quotiënt van NOP en de productiviteit levert dan de geschatte werklast op. Merk op dat de formule zeer eenvoudig is: er is geen sprake van exponenten of grote hoe-

veelheden cost drivers. Bovendien verhoogt de productiviteit per niveau met ongeveer een factor twee. Dit geeft aan dat het model niet de pretentie heeft een nauwkeurige schatting op te leveren: bij een bedrijf dat tussen twee niveaus zit, is de fout gemaakt door naar een niveau af te ronden bijna 50%.

Bij het vroegeontwerpmodel kan men reeds gebruik maken van een aantal interne factoren van het project, de zogenaamde functiepunten. Deze worden gemeten door een aantal kenmerken te tellen:

- Inputs. We hebben een input als we een elementair gegeven het systeem binnenkomt. Voorbeeld: het inscannen van een barcode, of het invullen van een formulier.
- Outputs. Analoog. Voorbeeld: een lijst van boeken van een auteur op het scherm zetten.
- Opvragingen. Dit is een geheel met input, opzoeken en output. Opzoeken gebeurt in databestanden of een databank.
- Interne databestanden. Een bestand bestaat uit logisch samenhangende gegevens. Een tabel van een databank is dan één bestand. Intern doelt op het feit dat de applicatie volledig verantwoordelijk is voor het onderhoud.
- Externe databestanden. De bestanden worden onderhouden door een van buitenaf; de applicatie zoekt er alleen in op.

Elk van deze getallen wordt vermenigvuldigd met een gewicht. Dit gewicht hangt af van de complexiteit van item, en van de aard van het kenmerk. Deze gewichten gaan van 3 (simpele input) tot 15 (complex databestand). De complexiteit wordt beoordeeld als laag, hoog of gemiddeld. Dit wordt bijvoorbeeld voor een transactie (input, output of opvraging) berekend met de volgende tabel, waarin FC de *file count* is die het aantal betrokken databestanden opgeeft, en DFC de *data field count* is die het aantal datavelden aanduidt

		DFC		
		$1-5$	$6-15$	>15
FC	$0-1$	laag	laag	midden
	$2-3$	laag	midden	hoog
	>3	midden	hoog	hoog

Alles wordt opgeteld om FP te krijgen, het aantal functiepunten. Dan geeft COCOMO II een tabel om FP om te rekenen naar KDSI, en gebruikt men de formule

$$\text{MM= 2,5} \times \text{m} \times \text{KDSI}^{1,15} + \text{AE}.$$

Hierin is m het product van de cost drivers, een systeem dat grosso modo overeenkomt met het systeem van COCOMO 81, alhoewel een aantal meer moderne factoren, zoals de mate van hergebruik, zijn opgenomen. Nieuw in deze formule is de term AE. AE staat voor *autoeffort* en geeft aan hoeveel werk de ontwikkelaars steken in het automatisch genereren van code, en aan de integratie van deze code in de andere code.

Het omrekenen van FP naar KDSI heeft een probleem: hoe hoger de programmeertaal, hoe minder lijnen code men nodig heeft om een bepaald functiepunt te realiseren. Aan de andere kant blijkt de efficiëntie van het productieproces, in lijnen code uitgedrukt, te *verminderen*: vermits een lijn code meer doet, is het moeilijker ze te ontwerpen. Waar hogere programmeertalen de productiviteit uitgedrukt in FP/mensmaand

doen stijgen, doen ze de productiviteit uitgedrukt in KDSI/mensmaand dalen. Vermits we uiteindelijk geïnteresseerd zijn in de hoeveelheid werk, en niet in de hoeveelheid code, moeten de COCOMO's dus voor elke programmeertaal herijkt worden.

Tenslotte is er het post-architecturale model. Ook hier gebruikt men een KDSI die afgeleid is uit de functiepunten. Er worden twee aanpassingen doorgevoerd aan het bekomen getal:

- Variabiliteit van vereisten. In het geval waar de vereisten gemakkelijk veranderen moet er vrij veel aanpassing gebeuren in latere fasen van het project. Dit leidt tot een hogere kost.
- De mate van hergebruik. Hergebruik veroorzaakt vrij hoge kosten in het begin van het ontwerp (kosten die op dit ogenblik al gemaakt zijn), maar besparingen komen pas later.

Het post-architecturale model gebruikt zeventien cost drivers, verdeeld in vier groepen:

- Producteigenschappen zoals complexiteit en vereiste betouwbaarheid.
- Hardwareverbonden eigenschappen, zoals vereisten qua snelheid en beperking van geheugengebruik.
- Personeelseigenschappen zoals ervaring.
- Projectomgeving, zoals gebruik van CASE.

Nieuw in deze fase is dat de exponent k zoals gebruikt in de factor KDSI^k niet vastligt. Deze wordt bepaald door vijf *schaalfactoren*. Deze naam volgt uit het feit dat k aangeeft in welke mate schaalvergroting tot meerwerk leidt. De vijf schaalfactoren zijn:

- Vertrouwdheid van de ontwikkelaars met dit soort project.
- Ontwikkelingsflexibiliteit. Een hoge flexibiliteit geeft aan dat de klant zich niet intensief bemoeit met het ontwikkelingsproces.
- Risicobeheer. Geeft aan hoeveel tijd moet besteed worden aan risicomanagement.
- Teamsamenhang.
- Maturiteit van de organisatie.

Elk van deze schaalfactoren kan een (gehele) waarde aannemen tussen 1 en 5 (grenzen inclusief). k wordt dan gegeven door

$$k = 0,91 + 0,01 \sum_{i=1}^{5} SF_i.$$

Waarin SF_i de schaalfactoren zijn. Wat opvalt is dat k een waarde *kleiner* dan 1 kan aannemen. Dit betekent dat er schaalvoordelen zijn: als project A 10 keer groter is dan project B en $k < 1$, dan zal A minder dan 10 keer de arbeidskracht van B vereisen. In COCOMO 81 was k altijd groter dan 1, omwille van de toenemende complexiteit bij grotere projecten. Nu echter kunnen schaalvoordelen optreden omdat bepaalde kosten minder snel toenemen dan de grootte. Zo is het, door betere analysemethodes, en vooral door betere documentatie, gemakkelijker geworden om de weg te vinden in een groot project. De tijd om vertrouwd te worden met het project neemt dus toe met de grootte, maar niet meer zo sterk dan vroeger. Bovendien zijn er opstartkosten, zoals

het volgen van een cursus bij het gebruik van een nieuwe tool. Dit soort kosten weegt minder zwaar door op een groot project dan op een klein.

16.7 HERGEBRUIK

Bij COCOMO 81 werd ervan uitgegaan dat alle code van een project binnenshuis geschreven code was, speciaal voor het project gemaakt. In de tijd dat dit model gemaakt is was dit een redelijke veronderstelling, maar dat is al lang niet meer het geval. Er worden verschillende categorieën van software onderscheiden:

1. Door een CASE-tool automatisch gegenereerde code.
2. Hergebruikte code. Bestaande code die als black box behandeld wordt en ongewijzigd in het product wordt opgenomen.
3. Aangepaste code. Beschouwd als white box en gewijzigd voor de noden van het project.
4. COTS (**C**ommercial **O**f **T**he **S**helf): geen broncode.

De omstandigheden waarin een CASE-tool code genereert verschillen zeer veel, zodat hier geen algemene richtlijn kan gegeven worden. Voor de twee volgende categorieën is er een gemeenschappelijke kostenfactor: AA (Assessment and Assimilation: beoordeling en invoeging), uitgedrukt in de kost per eenheid code. Deze houdt in het zoeken van de te hergebruiken code, en dus het beoordelen van mogelijke alternatieven, evenals de kosten voor het inpassen in het project. AA wordt o.a. beïnvloed door de kwaliteit van de documentatie.

Bij code die gewijzigd moet worden komen er nog twee kostenfactoren bij:

1. Het aanpassen van de code. Dit is proportioneel met de totale hoeveelheid code, en het aanpassingspercentage AAF. Dit aanpassingspercentage volgt dan weer uit drie deelfactoren:

   1. DM (Design Modification).
   2. CM (Code Modification).
   3. IM: kost om de software te integreren.

   We krijgen dan de formule

   $$\text{AAF} = (0.4 \times \text{DM}) + (0.3 \times \text{CM}) + (0.3 \times \text{IM}).$$

2. Het vertrouwd worden met de code. Dit hangt af van een factor UNFM, die aangeeft of de ontwerpers vertrouwd zijn met de in te passen software. UNFM=0,0 wil zeggen dat ze al vaak met deze software hebben gewerkt, UNFM=1,0 dat de software totaal onbekend is. Een tweede factor SU (Software Understanding increment), die aangeeft of de software gemakkelijk te begrijpen is. De software wordt beoordeeld op *structuur, duidelijkheid* van de applicatie, en *zelfbeschrijvendheid*. Software die zeer goed scoort op deze punten geeft een SU van 10%, zeer slechte haalt 50%. Dit laatste betekent dus dat men maar twee keer zoveel werk heeft om code zelf te maken dan om slechtgedocumenteerde code te begrijpen. De totale kost van dit punt is UNFM$\times$SU. Er is echter een maar: als de software veel moet aangepast worden, dan moet de code zeer grondig bestudeerd

worden; als er maar enkele aanpassingen nodig zijn moeten alleen kleine deeltjes van de software inwendig bekeken worden. Daarom is er een niet-lineaire formule: als er weinig aanpassingen zijn stijgt deze kost lineair met de proportie aanpassingen, als er zeer veel aanpassingen zijn moet men de hele code leren kennen. De grens wordt op 50% gelegd. Men gaat ervan uit dat de gehele code moet gekend zijn om 50% ervan te kunnen aanpassen. Als er dan nog meer aanpassingen moeten gebeuren is er geen toename van de leerkost meer. Het vereiste werk is dan

$$\text{VC} = \begin{cases} \dfrac{\text{AAF} \times 0.02 \times \text{SU} \times \text{UNFM}}{100}, & \text{AAF} \leq 50 \\ \dfrac{\text{SU} \times \text{UNFM}}{100}, & \text{AAF} > 50 \end{cases}$$

Typisch hebben we volgende waarden:

	hergebruikt CASE-uitvoer COTS	aangepast
DM	0%	0-100% Meestal >0%
CM	0%	0-100% Meestal >DM Zeker >0
IM	0%-100% meestal klein zelden 0%	0%-100% meestal middel soms > 100%
AA	0%-8%	0%-8%
SU	—	0%-50%
UNFM	—	0-1

Dit levert dan de *totale inspanning per eenheid code* op voor het hergebruik:

$$\text{AAM} = \frac{\text{AA} + \text{AAF} + \text{VC}}{100}.$$

Merk op dat AA, AAF en VC als percentages worden uitgedrukt, maar AAM als een fractie. De equivalente hoeveelheid KSLOC die overeenkomt met het werk gedaan om de hergebruikte code in te passen is dus

$$\text{equivalente KSLOC} = \text{aangepaste KSLOC} \times \text{AAM}.$$

Om nu de uiteindelijke formule voor de grootte van het project te krijgen, uitgedrukt in KSLOC, moeten we nog een factor toevoegen, REVL (Requirements Evolution and VoLatility). Bij projecten in een sterk veranderlijke omgeving worden vereisten soms tijdens de looptijd van het project veranderd. Dit levert extra werk op, en REVL geeft het percentage van dit extra werk. Uiteraard is REVL=0 als de vereisten niet veranderen. Zo krijgen we de formule voor de uiteindelijke grootte van het project:

$$\text{Grootte} = \left(1 + \frac{\text{REVL}}{100}\right) \times (\text{nieuwe KSLOC} + \text{equivalente KSLOC})$$

16.8 COCOTS

COTS is, volgens de COCOTS-definitie een commercieel softwareproduct met volgende eigenschappen:

- Verkocht of geleaset aan aangekondigde prijzen.
- Geen sourcecode beschikbaar.
- Meestal met periodieke updates (veroudering).
- Ontwikkeling niet onder controle van de ontwikkelaar.

In het laatste punt: we gaan ervan uit dat de producent van de COTS-software wel controle heeft over zijn product; het gaat hier over gebrek aan controle door de ontwikkelaar van het project waarin de COTS-componenten worden geïntegreerd. Deze heeft weinig of niets te zeggen over de functionaliteit van het COTS-product, noch over de evolutie. Vermits de ontwikkelaar, als er een nieuwe versie uitkomt, na een tijdje de oude versie niet meer onderhoudt (verwijderen van bugs), is de gebruiker van een COTS-component dikwijls *verplicht* te upgraden, zelfs al brengt dit voor hem alleen extra kosten mee. Als er bugs zijn, of gewenste wijzigingen, kan de software alleen door de producent aangepast worden. Bij open sourcesoftware moet men de aanpassing vaak zelf doen, maar men *kan* ze tenminste zelf doen. Bijgevolg moet bij COTS niet alleen de kwaliteit van de software, maar ook deze van de leverancier beoordeeld worden. Er dienen vier soorten werk in acht genomen te worden bij het gebruik van COTS:

- **Beoordeling**. Voor een pakket kan worden aangekocht, moet er een afweging worden gemaakt van de verschillende opties. Eventueel dienen verschillende COTS-pakketten tegen elkaar worden afgewogen, maar er kunnen ook alternatieven zijn zoals open sourcecode of zelf schrijven. Dit is de factor AA van hierboven.
- **Aanpassing**. Configuratie voor gebruik in eigen project.
- **Maken van lijmcode.** Lijmcode (*glue code*) is code die moet geschreven worden om een COTS-module te integreren. Deze twee punten vormen samen de factor IM uit de formule.
- **Vluchtigheid.** Werk veroorzaakt door de noodzaak om upgrades te doen.

Ook hier krijgen we een formule van de nu al vertrouwde vorm

$$\text{MM} = A \times \text{Grootte}^k \times \times_{i=1}^{14} EM_i.$$

Hierin zijn de factoren EM_i afkomstig van 14 cost drivers, en is k afhankelijk van een aantal schaalfactoren. Deze formule kan gebruikt worden op elk van de vier soorten werk apart.

16.9 RISICOANALYSE EN -BEHEER

De uitvoering van een project houdt een aantal risico's in. Er kan van alles fout gaan, en de gevolgen daarvan kunnen zeer klein zijn (een kleine meerkost bijvoorbeeld) maar ook zeer groot. Een strategie is natuurlijk om problemen pas op te lossen als ze zich voordoen. Maar in veel gevallen is het mogelijk om maatregelen te nemen om

problemen te voorkomen, en het is steeds belangrijk om enig zicht te hebben op de mogelijke extra kosten, al was het maar om bijvoorbeeld een offerteprijs te kunnen maken of om een planning op te maken. Goed georganiseerde bedrijven besteden dan ook veel zorg aan risicobeheer, en bij CMM is risicobeheer verplicht om een certificaat van niveau 3 te halen.

Waar kostenschattingen al vrij moeilijk zijn, is het zeer moeilijk om risico's in te schatten. Per definitie gaat het over gebeurtenissen die al dan niet kunnen plaatsvinden. Sommige risico's gaan over totaal onverwachte gebeurtenissen. Het spreekt vanzelf dat deze risico's per definitie niet te voorspellen zijn, laat staan beheerst kunnen worden. Bij de meeste softewareprojecten zijn er echter een aantal risico's verbonden met (vrij) vaak voorkomende gebeurtenissen, zodat men een idee heeft van de kans op zo'n gebeurtenis, en de impact ervan. Strikt genomen zijn er twee soorten risicobeheer verbonden aan een softwareproject:

- Softwareveiligheid. Wat kan er misgaan bij het gebruik van de afgeleverde software, en hoe zwaar zijn de gevolgen? De risico's kunnen gaan van miniem (een kleine, omzeilbare bug) tot enorm. Bijvoorbeeld bij medische software kan het leven van patiënten door fouten in gevaar gebracht worden. Dit is software die vaak niet door een groot bedrijf met hoog CMM-niveau gemaakt wordt. Ook computergestuurde productieprocessen kunnen gevaar opleveren.
- Procesveiligheid. Wat kan er misgaan bij het ontwikkelingsproces? Hier zijn de gevolgen meer overzienbaar. De meeste risico's leiden tot meerkost en/of vertraging van het project. Dit kan dan ook leiden tot een verminderde kwaliteit van de af te leveren software. Zowat het ergste wat er kan gebeuren is dat het project wordt afgeblazen, met verlies van alle geïnvesteerde middelen.

In deze paragraaf zullen we ons voornamelijk bezighouden met procesveiligheid. Fundamenteel zijn er twee verschillende soorten van aanpak:

- Een *passieve* aanpak. Risico's worden beschouwd als onveranderlijke zaken die het project kunnen overkomen. Al wat men kan doem is middelen (geld, tijd) voorzien om de gevolgen op te vangen.
- Een *actieve* aanpak. Men kan voorzorgen nemen om de waarschijnlijkheid van het risico te verminderen en om de gevolgen, als het zich voordoet, te milderen.

Nemen we als voorbeeld het risico dat de harde schijf met alle informatie van het project crasht. De passieve aanpak bestaat eruit om voldoende middelen vrij te maken om opnieuw te beginnen als het feit zich voordoet (eventueel gespreid over verschillende projecten: als de schijf gemiddeld een keer crasht op 10 projecten, gaat men 5% extra middelen voorzien om opnieuw te beginnen[4].) Bij een actieve aanpak gaat men proberen de waarschijnlijkheid te doen afnemen (door meer betrouwbare hardware aan te schaffen), en de gevolgen te verminderen (door back-ups te nemen).

In het voorbeeld is de actieve aanpak duidelijk veel beter dan de passieve aanpak, maar dit is niet altijd het geval. Veel risico's zijn niet te vermijden, en hun gevolgen liggen vast. Een wispelturige klant kan bijvoorbeeld extra kosten veroorzaken door de specificaties vaak te willen veranderen, maar daar is op voorhand weinig tegen te doen. Al wat men kan doen is de kostprijs op voorhand inschatten. Dergelijke kost-

[4] 5%, en geen 10%, want elke crash zal gemiddeld maar de helft van een project meenemen.

prijsberekening is ook een goede basis voor een actieve aanpak. Als men de kostprijs kan vergelijken met en zonder speciale maatregelen, dan heeft men een idee van het nut van die maatregelen. Als blijkt dat een bepaalde werkmethode die een bepaald risico vermijdt de kostprijs van het project met 10% doet toenemen, terwijl de gevolgen van het risico slechts 8% van de kostprijs bedragen, weet men dat de vermijdende maatregelen beter niet genomen worden.

Bij het berekenen van de kostprijs van risico's zijn er twee soorten aanpak:

- Men kan uitgaan van de *kenmerken* van een project. Dit is de *impliciete* methode, waarin men de risicobronnen niet expliciet gaat opsommen.
- Men kan risico per risico bekijken en daarvan de kostprijs berekenen. Dit is de *expliciete* methode.

Het spreekt vanzelf dat de tweede methode meer mogelijkheden biedt als basis voor een actieve aanpak, omdat een actieve aanpak vereist dat de risico's geïdentificeerd worden.

Bij de impliciete methode gaat men een inschatting maken van bepaalde kenmerken waarvan de ervaring zegt dat ze risico's opleveren. De kostenschatting wordt dan verhoogd naargelang deze eigenschappen. De impliciete methode is min of meer ingebouwd in het COCOMO. Zo is er bijvoorbeeld de factor REVL (Requirements Evolution and VoLatility). Deze geeft de mate aan waarin specificaties kunnen veranderen. Dit is een risico: de specificaties hoeven niet te veranderen, en de mate van verandering kan verschillen van de factor REVL. Ook cost drivers zoals complexiteit (in een complex project is de kans groot dat er stukken zijn waaraan meer tijd wordt besteed als eerst voorzien) en ervarenheid van personeel (van onervaren personeel is het moeilijk in te schatten hoe snel het zal werken) zijn typische voorbeelden van risicogerelateerde eigenschappen.

Een verwante methode wordt gebruikt bij functiepuntenanalyse. We hebben al gezien dat functiepunten een methode opleveren om de grootte van een project te schatten. Men kan nu het aantal functiepunten aanpassen door te gaan kijken naar een aantal systeemkarakteristieken. In totaal zijn er 14 zulke karakteristieken. Zo is er bijvoorbeeld de mate van herbruikbaarheid van de code, de vraag of gegevens is real time moeten aangepast worden, de mate van belasting van de hardware, enzovoorts. Elk van deze karakteristieken krijgt een DI-waarde DI_i die ligt tussen 0 en 5. Het aantal functiepunten wordt dan vermenigvuldigd met *VAF*, met

$$VAF = 0,65 + 0,1 \sum_{i=1}^{14} DI_i.$$

Als alle karakteristieken hun maximale waarde 5 bereiken, dan is $VAF = 1,35$, zodat het aantal functiepunten met 35% toeneemt.

De grens tussen impliciete en expliciete methode is niet altijd exact te trekken. Een aantal kenmerken van een project kan men niet beïnvloeden, maar andere wel. Bekijken we bijvoorbeeld hergebruik. Een risico hierbij is dat men de hergebruikte code moet aanpassen, en de leerkost hierbij was proportioneel met UNFM$\times$SU (graad van onvertrouwdheid maal ontoegankelijkheid van de code). Dit zijn kenmerken van het project, maar men kan deze veranderen door uit te gaan van andere te hergebruiken code, of men kan besluiten dat het risico te groot is (hergebruik van code *kan* duurder zijn dan zelf schrijven) en dat alles in huis moet geschreven worden.

Bij de expliciete methode is het eerste probleem om vast te stellen welke de risico's zijn. Hierbij gaat men uit van vragenlijsten, of van lijsten met mogelijke bronnen van risico's, zoals de onderstaande tabel:

Management	**Specificatie**	**Ontwerp**
Groepswerk	Duidelijkheid vereisten	Formeel proces
Planning	Formeel proces	Strengheid review
Opvolging	Inbreng klant	Mate hergebruik
Automatisering	Invloed op bedrijf	Ervaring personeel
Bekwaamheid management		Inbreng klant
		automatisering
Realisatie	**Testen**	**Omgeving**
Revisie code	Formele testomgeving	Nieuwheid technologie
Hergebruik code	Testplanning	Veranderlijkheid
Beschikbaarheid hardware	Ervaring personeel	Training personeel
Ervaring personeel	Inbreng klant	Stabiliteit bedrijf
Automatisatie		

Men overloopt dergelijke tabel en bepaalt de bijbehorende risico's. Zo is het voorbeeld uit vorige paragraaf onder te brengen bij Realisatie::hergebruik, en kan een gebrek aan stabiliteit bij het bedrijf leiden tot personeelsverloop met de bijbehorende kosten. Voor elk risico schat men tegelijkertijd mogelijke manieren om het risico te verminderen in. Zoals we gezien hebben zijn er twee mogelijke soorten acties: men kan de *kans* P van het risico verminderen, of men kan de gebeurlijke *kost* C verminderen. Beide hebben hun invloed op de zogenaamde *impact* I, gegeven door

$$I = C \times P.$$

De impact is de *verwachte gemiddelde kost* van het risico. We gaan er nu van uit dat we de totale impact van de risico's willen verminderen. Dat is, gegeven een aantal risico's genummerd van 1 tot n, dan willen we

$$I_{tot} = \sum_{i=1}^{n} I_i$$

zo klein mogelijk maken. Merk op dat I_{tot} een schatting is van de totale kost veroorzaakt door alle risico's samen. Nemen we nu, voor een bepaald risico, een verzachtende maatregel, zoals het nemen van back-ups in ons voorbeeld. Zulk een maatregel *veroorzaakt* zelf een kost, M (M staat voor *mitigation*). We veronderstellen dat M groter dan 0 is[5]. De maatregel kan zowel de waarschijnlijkheid als de kost van de gevolgen beïnvloeden, en dus verandert de impact van het risico:

$$I_{\text{voor}} = C_{\text{voor}} \times P_{\text{voor}} \rightarrow I_{\text{na}} = C_{\text{na}} \times P_{\text{na}}.$$

De efficiëntie van de investering M is wordt dan aangeduid door RRL (Risk Reduction Leverage), met

$$\text{RRL} = \frac{I_{\text{voor}} - I_{\text{na}}}{M}.$$

[5] Strikt genomen is dat niet altijd zo. Het kan zijn dat meer betrouwbare hardware goedkoper is dan andere. Vermits we in zo'n geval geen enkele reden hebben om de maatregel niet te nemen, kunnen we dit geval buiten beschouwing laten.

Hoe groter RRL is, hoe efficiënter de investering. Het is duidelijk dat we alleen geïnteresseerd zullen zijn in die investeringen waarvoor RRL > 1. Vermits er enige onzekerheid is over deze RRL, vermits deze berust op schattingen van de probabiliteiten, is het best mogelijk dat investeringen met een RRL-waarde nauwelijks groter dan 1 ook niet zullen gedaan worden. In elk geval zal men bij voorkeur investeren in maatregelen met de grootste RRL.

In deze analyse zijn we ervan uitgegaan dat het de bedoeling is om I_{tot} te minimaliseren. Dit is zo als er een groot aantal risico's is met een gelijkaardige risicokost. Indien er echter risico's zijn met een zeer hoge kost C gekoppeld aan een zeer lage probabiliteit P, dan gaat men dikwijls maatregelen nemen die de kost verminderen, zelfs indien de bijbehorende RRL kleiner is dan 1. Dergelijke maatregelen volgen het zgn. *verzekeringsprincipe*. Bij een verzekering draagt een verzekeringsnemer een onwaarschijnlijk risico met zeer hoge kost over aan een verzekeraar, en dit tegen betaling van een premie (met een lage kost). De waarschijnlijkheid dat de premie moet betaald worden is 1. De verzekeraar gaat ervan uit dat de impact van de verzekerde risico's kleiner zal zijn dan de som van de premies. Hieruit volgt dat voor de verzekerde de kost M (de premie) meestal groter is dan de impact van het oorspronkelijke risico. Als we aannemen dat I_{na} 0 is (en bij de meeste verzekerde risico's zijn er residuele kosten, zoals vrijstellingen), dan nog is volgens de bovenstaande formule RRL < 1.

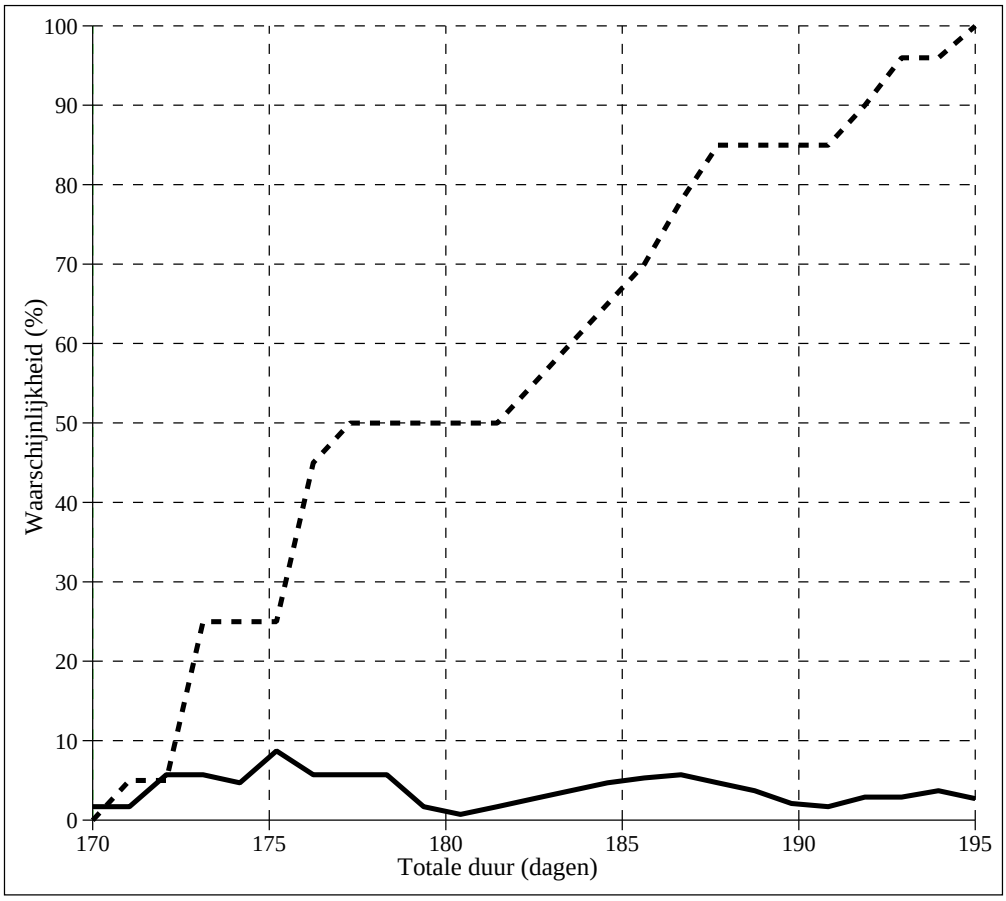
In de praktijk is het meestal onmogelijk om juiste bepalingen te geven van de kost en van de waarschijnlijkheid van een risico, zodat een berekening volgens bovenstaande formules meer nauwkeurigheid doet uitschijnen dan ze geeft. Daarom maakt men vaak gebruik van vereenvoudigde schattingen. Een voorbeeld hiervan is Tassc:Estimator. Hier wordt gebruikt gemaakt van vijf niveaus, zowel voor de kost als voor de waarschijnlijkheid. Deze worden als volgt getalsmatig omgezet:

kost					
	miniem	laag	hoog	zwaar	kritisch
	1%	2%	3%	4%	5%
kans					
	zeer klein	klein	50/50	groot	bijna zeker
	15%	35%	50%	65%	85%

Merk op dat de getalwaarden verschillen naargelang de grootte van een project: 5% van een groot project is zeker een zeer zware kost, terwijl bij kleinere projecten dit percentage slechts als 'zwaar' zou aangemerkt worden. Als men voor alle risico's de kost en de waarschijnlijkheid heeft vastgesteld, hetzij op de 'losse' manier, hetzij met de schattingsmethode zojuist gegeven, dan kan men *risicocurves* tekenen. In het voorbeeld van Figuur 16.4 hebben zien we hiervan een voorbeeld. Hier is de kost uitgedrukt in extra dagen loopduur van het project. Voor alle duidelijkheid heeft men hier niet de kost veroorzaakt door de risico's uitgetekend, maar heeft men hierbij de basisschatting van 170 dagen bijgevoegd.

Er zijn twee risicocurves: een puntcurve en een cumulatieve curve.

De puntcurve geeft weer wat de kans is dat een bepaald project *exact met* een bepaalde tijdsduur beëindigt wordt; de cumulatieve curve geeft de kans dat de tijdsduur *hoogstens* een bepaalde tijdsduur heeft. Hoe berekenen we nu beide curves? Nemen we een willekeurige deelverzameling S van alle risico's (of om juist te zijn: een deelverzameling van indexen van risico's). De kans dat alle risico's van S zich zullen uiten,



Figuur 16.4. Risicocurve. Waarschijnlijkheid van de duur van het project (volle lijn) en cumulatieve waarschijnlijkheid (streepjeslijn).

terwijl geen enkel risico dat niet in zit S gerealiseerd wordt is

$$P(S) = (\times_{i \in S} P_i) \times (\times_{i \notin S} (1 - P_i)),$$

als we tenminste mogen veronderstellen dat de risico's niet afhankelijk zijn, zodat het voorkomen van één risico niet de kans van een ander risico beïnvloedt. De kost van S krijgen we door de kosten gewoon op te tellen:

$$C(S) = \sum_{i \in S} C_i.$$

Bijgevolg is de kans dat de totale kost exact een bepaalde waarde C aanneemt gelijk aan

$$P(C_{\text{tot}} = C) = \sum_{C(S)=C} P(S).$$

Als men dit echter gaat uittekenen op een grafiek is het zeer goed mogelijk dat men een curve krijgt die zeer veel schommelt. Als in het voorbeeld elk risico een even aantal dagen vertraging oplevert, dan is elk oneven aantal dagen onmogelijk, zodat we een

sterk getande curve krijgen. Dit lost men op door een gewogen gemiddelde te nemen van nabijgelegen waarden.

Naarmate het project evalueert kan de status van bepaalde risico's veranderen. Het risico kan zich hebben voorgedaan (waardoor zijn probabiliteit verhoogt tot 100%), maar het kan ook zijn dat het risico wegvalt, omdat het alleen in een bepaalde fase van het project kan voordoen. Software die risicobeheer begeleidt zal dus de mogelijkheid bieden de status van een risico te veranderen van actief (kan nog voorkomen) naar afgevuurd of inactief. Hierdoor veranderen ook de risicocurves.

We zijn reeds op verschillende plaatsen problemen tegengekomen die verband houden met het verschil tussen specificatie en implementatie. Grosso modo liggen deze problemen op twee terreinen: het *datagedeelte* en het *procedureel gedeelte*. Bij het datagedeelte hebben we gezien dat we beginnen met een abstracte beschrijving van modules, zoals klassen. Bij de implementatie duiken er dan problemen op. Eigenlijk zijn er al problemen bij het klassendiagram van UML. Hier kunnen we afgeleide attributen aangeven. Maar zoals we gezien hebben is het vaak (vrij) willekeurig welke attributen we als afgeleide, en welke we als echte attributen beschouwen. Problemen zijn er ook bij het proceduraal gedeelte. Hoe definiëren we pre- en postcondities? Het kan niet de bedoeling zijn dat we voor een methode van vier lijntjes code bladzijden en bladzijden specificaties moeten schrijven. Aan de andere kant moeten de specificaties ondubbelzinnig zijn. Het blijkt dat bij het debuggen veel fouten het gevolg zijn van verkeerd begrepen specificaties: module X verwacht van Y dat ze iets doet, maar in een speciaal geval doet Y net iets anders dan ze moet doen.

Het is instructief om hier een voorbeeld te geven van wat er mis kan gaan met een vage specificatie. We gaan uit van een klasse `P` (de naam ervan is met opzet nietszeggend gekozen). `P` bevat twee gegevensvelden, als volgt:

```
typedef unsigned int uint;
class P{
    ...
    uint n;
    uint* tab;
};
```

Hier is `n` de grootte van de tabel waar `tab` naar wijst. Merk op dat dit als specificatie al zeer vaag is: wijst `tab` naar een tabel die alleen door onze `P` gekend is en beheerd wordt (en vermoedelijk in de constructor gemaakt wordt en in de destructor vernietigd), of is het een pointer naar een tabel die buiten onze klasse om bestaat? Voor `P` schrijven we nu twee zoekfuncties die een element in de tabel opzoeken, en de index van dit element teruggeven. Wie denkt dat dit een duidelijke omschrijving is van de zoekfuncties:

```
int P::zoekA(uint a){
    int i=0;
    while ( i<n && tab[i]!=a)
        i++;
    return i;
}

int P::zoekB(uint a){
    int i=n-1;
```

```
    while ( i>=0 && tab[i]!=a)
        i--;
    return i;
}
```

Het is duidelijk dat `zoekA(a)` en `zoekB(a)` alleen het zelfde resultaat geven als `a` juist één keer voorkomt in de tabel:

- Als `a` niet voorkomt in de tabel geeft `zoekA` de waarde `n` terug, terwijl `zoekB` de waarde -1 teruggeeft.
- Als `a` meerdere keren voorkomt in de tabel, dan geeft `zoekA` de waarde van de *kleinste* index waarvoor `tab[i]=a` terug, terwijl `zoekB` de *grootste* index teruggeeft.

Ook bij wiskundige inspectie zitten we met hetzelfde probleem: we moeten op een vlotte én duidelijke manier de tussentoestanden kunnen beschrijven. Dit is ook belangrijk bij testen: hoe kunnen we een module testen, als we niet eens weten wat ze juist moet doen? Bovendien gaan we bij het ontwerp van blackboxtesten uit van de specificaties. Hierbij hebben we speciale aandacht voor rand- en overgangsgevallen. Ook hier geldt dus dat duidelijkheid van de specificaties vlot en efficiënt werk ten goede komt. Bovendien is het zo dat, als we de specificaties op de goede manier uitdrukken, het misschien mogelijk is om een gedeelte van onze tests automatisch kunnen laten verlopen. Als we namelijk onze specificaties geven in de vorm van een stuk *uitvoerbare code* –in de praktijk een functie die een logische waarde teruggeeft: waar als aan de specificatie voldaan is, en onwaar als dit niet het geval is– dan kan een geautomatiseerde testomgeving deze code laten lopen op het gepaste moment. Dit kan ook met de hand gebeuren: in een zeer informele vorm vinden we dit al terug bij het gebruik van de `assert`opdracht in C++ en Java. Deze laat toe om een controleopdracht in de code bij te schuiven. Bij het optreden van een bug kan de `assert`opdracht de fout detecteren en melden door het programma voortijdig te doen stoppen. Dit is uiteraard vrij primitief, maar het heeft het voordeel dat de fout in een vroeg stadium kan ontdekt worden. Zodoende is de afstand tussen optreden van de fout en ontdekking klein, en dit maakt het gemakkelijker de oorzaak van de fout te vinden.

Een hulpmiddel hiervoor is formele specificatietheorie, die probeert de overgang van abstracte definitie naar concrete implementatie zo vlot mogelijk te doen verlopen. Het is ook mogelijk de specificaties uit te drukken in een *formele taal*. Niet alleen opent dit de weg naar specificaties die exacter zijn dan specificaties in natuurlijke taal, het wordt ook mogelijk de specificaties geautomatiseerd te verwerken. Zo is het denkbaar dat een gedeelte van de code voor testen automatisch wordt gegenereerd op basis van de specificaties. Als voorbeeld zullen we hier de formele specificatietaal Z bespreken.

19.1 REPRESENTATIE- EN ABSTRACTE WAARDENVERZAMELING

We hebben reeds gezien dat een object een toestand heeft. De verzameling van mogelijke toestanden voor een gegeven klasse heet de *abstracte ruimte*. Deze ruimte wordt in het vervolg aangeduid door **Abs** of, als we expliciet willen aangeven over welke klasse het gaat, als **Abs**(`klassennaam`). **Abs** wordt gedefinieerd tijdens de modellering van

de *realiteit*, en komt dus voor elke implementatie. Als we het vijflagenschema voor data op pagina 110 nemen, dan leeft de ruimte **Abs** dus in de meest linkse laag. Soms is het zeer eenvoudig om de abstracte ruimte van een klasse te omschrijven. Bij een klasse `Punt2D` van punten in een vlak, waarbij de coördinaten alle mogelijke waarden kunnen aannemen, hoort als **Abs**(`Punt2D`) het vlak zelf. Als we een klasse `Persoon` hebben gedefinieerd door het klassendiagram

Persoon
naam:string grootte:int

dan bestaat de abstracte ruimte voor die klasse uit de combinatie van alle mogelijke strings en alle mogelijke gehele getallen die mogelijk zijn bij een persoon. We spreken van de *abstracte* ruimte omdat niet alle toestanden in die ruimte daadwerkelijk zullen voorkomen in ons systeem. Het is zeer onwaarschijnlijk dat er ooit een persoon zal zijn met de naam *Olleke Bolleke Riebesolleke*, maar we sluiten dit niet uit. Bij de twee gegeven voorbeelden is de abstracte ruimte zeer eenvoudig, maar het kan zijn dat er beperkingen worden opgelegd aan de waarde van de attributen. Bovendien moet rekening gehouden worden met de verbindingen met andere objecten.

De *representatieruimte* **Rep** is de verzameling van alle mogelijke toestanden van de implementatieobjecten. Ruwweg gezegd zijn dit de mogelijke combinaties van alle gegevensvelden, attributen zowel als wijzers naar andere objecten die verbindingen realiseren. Als er beperkingen zijn op de waarden van de gegevensvelden, van welke vorm ook, dan is niet elke *mogelijke* combinatie een *geldige* combinatie. **Rep** bevat zowel de geldige als de ongeldige combinaties. Het is de bedoeling dat een implementatieobject overeenkomt met een reëel object. Dit doet twee vragen rijzen:

- Welke waarden in de representatieruimte komen overeen met een toestand in de abstracte ruimte?
- Met welke abstracte toestand komt zo'n geldige waarde overeen?

Als we terug naar ons vijflagenmodel kijken, dan bevinden we ons nu in de meest rechtse laag, die van de programmaobjecten. De eerste vraag is de vraag wat de toestand kan en mag zijn van een object in deze laag; de tweede vraag is wat de relatie is tussen zo'n object en een object in de linkse laag. Het is duidelijk dat alleen een geldige toestand overeenkomt met een reëel object, of althans een mogelijk object uit de ruimte **Abs**. Merk op dat het antwoord op de eerste vraag niet altijd het antwoord op de tweede vraag geeft. Het is niet omdat we weten dat een implementatieobject overeenkomt met een element van **Abs**, dat we weten met *welk* object het overeenkomt. In ons voorbeeld van het `grootte`attribuut van de `Persoon`sklasse is het duidelijk dat, in principe, alle positieve waarden geldig zijn, maar we moeten ook weten of de lengte uitgedrukt wordt in millimeter of in centimeter, of in een andere maat.

Het antwoord op de eerste vraag kunnen we proberen in code te vatten. In abstracte zin bestaat er een functie R die logische waarden aanneemt:

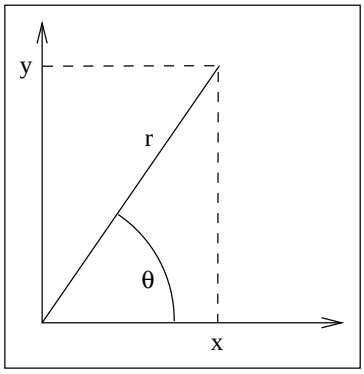
$$R : \textbf{Rep} \rightarrow \{\texttt{true}, \texttt{false}\}.$$

Als de waarden van een implementatieobject overeen komen met een toestand in de abstracte ruimte, dan zal R de waarde `true` teruggeven, anders de waarde `false`. Deze

functie wordt de *representatie-invariant* genoemd. Dit wordt vrij algemeen afgekort tot rep-invariant. Ook de tweede vraag kunnen we beantwoorden door een functie: de *abstractiefunctie* A,

$$A : \textbf{Rep} \rightarrow \textbf{Abs}.$$

Uiteraard bestaat $A(x)$ als en slechts als $R(x) = \texttt{true}$. De functie R kan in code worden uitgedrukt, en bijvoorbeeld geïmplementeerd worden als een lidfunctie van de betreffende klasse. Deze lidfunctie speelt geen rol bij het gewoon gebruik van de klasse; ze wordt enkel gebruikt in de testfase. Hoewel R formeel een navraag is die het object niet verandert, geeft het geen attribuut, zelfs geen afgeleid attribuut, van de klasse weer. Immers, voor elk object van de klasse moet R altijd `true` zijn. De functie A is uiteraard alleen gedefinieerd voor *geldige* waarden in **Rep**. Ze kan ook niet worden geïmplementeerd, vermits we geen computervoorstelling hebben van de abstracte ruimte (behalve dan **Rep** zelf natuurlijk). Ze speelt wel een belangrijke rol in het begrijpen van wat we doen. Ze kan wel omschreven worden en, als **Abs** wiskundig kan gedefinieerd worden, dan kan dit zeer exact gebeuren. Zo kunnen we het klassieke voorbeeld nemen van verschillende coördinatenstelsels op het vlak. We nemen weer de klasse `Punt2D`. Zoals bekend kunnen we de plaats van een punt bepalen met cartesiaanse coördinaten (x, y) of met poolcoördinaten (r, θ) zoals getekend op Figuur 19.1. We kunnen de klasse dan ook op verschillende manieren implementeren:



Figuur 19.1. Cartesiaanse en poolcoördinaten van een punt.

```
Punt2DC{
    double x, y;
}
```

```
Punt2DP{
    double r, theta;
}
```

Duidelijk is dat **Rep**(`Punt2DC`) = **Rep**(`Punt2DP`), vermits er beide keren twee `doubles` zijn. De rep-invarianten zij echter verschillend: in het eerste geval is elke combinatie van twee reële gevallen een geldig punt, in het tweede geval geldt als bijkomende voorwaarde dat $\texttt{r} \geq 0$. Nemen we aan dat de coördinaten (x, y) de 'natuurlijke coördinaten zijn die we gebruiken in **Abs**, dan krijgen we

$$\begin{aligned} A_{\texttt{Punt2DC}}(\texttt{c}) &= (\texttt{c.x}, \texttt{c.y}) \\ A_{\texttt{Punt2DP}}(\texttt{p}) &= (\texttt{p.r}\cos(\texttt{p.theta}), \texttt{p.r}\sin(\texttt{p.theta})). \end{aligned}$$

Merk op dat we in het tweede geval geen eenduidige voorstelling hebben van het punt $(0,0)$: elk object van `Punt2DP` waarvoor `r` $= 0$ wordt afgebeeld op dit punt. Bovendien, als we twee elementen `p` en `q` hebben waarbij `p.r` $=$ `q.r` en ook `p.theta` $=$ `q.theta` $+$ $2k\pi$, met k een willekeurig geheel getal, dan zal

$$A_{\texttt{Punt2DP}}(\texttt{p}) = A_{\texttt{Punt2DP}}(\texttt{q}).$$

In ons voorbeeld van de `Persoon`sklasse is de afbeelding A een eenvoudige een-op-een afbeelding. Bij meer complexe klassen en implementatie is dat niet meer het geval. Nemen we bijvoorbeeld een hashtabel met open adressering. Als we aan een lege hashtabel een element toevoegen en daarna terug verwijderen, dan hebben we terug een lege hashtabel. Alleen is er één vakje van de tabel dat niet meer als leeg staat aangeduid, maar als geschrapt. Na de bewerking hebben we dus een ander element van **Rep** dan daarvoor, want een aantal bits in het implementatieobject zijn veranderd. Beide elementen van **Rep** worden door de functie A echter afgebeeld op hetzelfde element van **Abs**, namelijk op de abstracte lege hashtabel.

Het is belangrijk om twee dingen niet te vergeten. De ruimte **Abs** wordt gedefinieerd bij de specificatie van de klasse. Het is hiermee bijvoorbeeld dat we te maken hebben bij het opmaken van het klassendiagram. Bij implementatie wordt de ruimte **Rep** *gekozen*. Het is dus niet zo dat **Rep** volledig bepaald wordt door **Abs**. We hebben reeds het voorbeeld van de klasse `Punt2D` gezien waar verschillende representatieruimten kunnen gekozen worden, en er zijn veel voorbeelden van hoe eenzelfde ruimte **Abs** op verschillende manieren kan geïmplementeerd worden. Uiteraard is de keuze van de ruimte **Rep** niet onbeperkt: zo moet er een zinnige interpretatie zijn van de functie A, maar voor de rest is er enige vrijheid.

Op een zeer elementair niveau is er de keuze van de manier waarop attributen worden opgeslagen: slaan we de *grootte* van een persoon als een geheel getal of als een vlottendekommagetal? Deze keuze heeft uiteraard geen invloed op **Abs**, vermits dit de verzameling is van alle personen Iets meer vrijheid hebben we als er afgeleide attributen zijn. Zeer vaak is het mogelijk om, uit de totale lijst van attributen de verzameling van afgeleide attributen op verschillende manieren te definiëren op zo'n manier dat het object bepaald wordt door de echte attributen zonder dat er (te veel) overlapping is. Maar er kunnen andere verschillen zijn, die veel dieper gaan. Dit komt vaak voor bij containerklassen. Nemen we het voorbeeld van een klasse `Intset` die verzamelingen van gehele getallen voorstelt. Er zijn verschillende manieren om zo'n verzameling te implementeren. Op de achtergrond kan er een gelinkte lijst zijn, maar ook een tabel, of een binaire boom. Dit wordt bepaald bij het ontwerp van de implementatie. Wat men niet mag vergeten is dat de keuze van **Rep** niet altijd de keuze van de abstractiefunctie, en samenhangend daarmee de rep-invariant R, helemaal bepaalt. Bij het `Persoon`svoorbeeld was de keuze van de abstractiefunctie, eenmaal we beslist hebben de klasse `Persoon` te implementeren met twee lidvelden, een `string` en een `int`, beperkt. Eén keer we **Rep** gekozen hadden, met een `int`veld voor de grootte, konden we nog kiezen of we de grootte uitdrukten in centimeter of in millimeter. Deze keuze heeft uiteraard geen invloed op **Abs**, vermits dit de verzameling is van alle personen, en ook niet op **Rep**. Het heeft ook geen invloed op de rep-invariant R. We eisen natuurlijk dat de lengte strikt positief is, maar vermits we geen echte grens hebben voor de grootte van de kleinst of grootst mogelijke mens, gaan we geen nauwere grenzen opleggen in

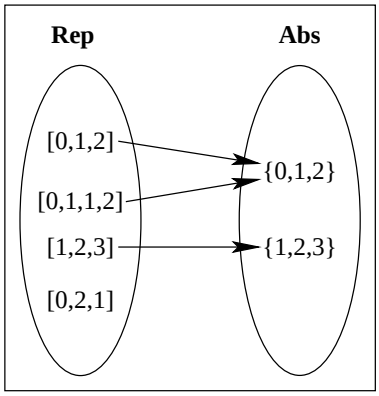
R. Uiteraard, of `grootte=180` een normale waarde is, hangt af van de gekozen A. Bij complexe klassen is dit de keuze niet meer triviaal.

Nemen we weer onze `Intset`, en veronderstellen we een implementatie van deze vorm:

```
class Intset{
    int* tabel;
    unsigned int maxgrootte;
    unsigned int aantal;
}
```

De bedoeling lijkt op het eerste zicht wel duidelijk. We hebben een pointer naar een tabel die `maxgrootte` elementen kan bevatten, en waarvan de eerste `aantal` vakjes zijn ingevuld. Maar is dit alles wel eenduidig? Om te beginnen zouden we kunnen eisen dat de tabel *gesorteerd* is. Dit maakt opzoeken gemakkelijker. Verder is er ook nog het probleem van de duplicaten. Is $[0, 1, 1]$ een geldige waarde? We kunnen eisen dat er geen duplicaten mogen voorkomen in de representatie en dat we niet sorteren: dit maakt toevoegen moeilijker. We moeten opzoeken of het element nog niet voorkomt, en dit maakt van toevoegen een $O(n)$-operatie. Als we wel duplicaten toestaan kunnen gewoon, zonder te kijken, elementen achteraan toevoegen. Maar dit heeft ook zijn nadelen: door duplicaten toe te staan kunnen er meer vakjes in de tabel bezet geraken, zodat opzoeken trager wordt. Bovendien moet men er bij verwijderen zorg voor dragen alle duplicaten te verwijderen. Als er tien exemplaren van het getal 1 in de representatie zitten, en we willen 1 verwijderen uit de verzameling, moeten alle tien duplicaten verdwijnen.

Merk op dat de combinatie duplicaten toestaan en sorteren vrij onzinnig is. Het is echter een toelaatbare keuze, en we krijgen dan de abstractiefunctie uit Figuur 19.2. Zoals we al hebben opgemerkt kunnen we proberen de functie R die aangeeft of een



Figuur 19.2. Abstractiefunctie.

representatie geldig is in code uit te schrijven. Bekijken we even een voorbeeld. Als verzameling **Abs** nemen we weer de verzameling van alle verzamelingen van gehele getallen, en voor **Rep** nemen we de verzameling dubbelgelinkte lijsten, met de volgende datavelden:

```
class Intlijst{
    Knoop* k;
};

class Knoop{
    int getal;
    Knoop* vorige;
    Intlijst volgende;
};
```

Vermits we duplicaten toestaan, is elke combinatie van getallen geldig. De controle van de rep-invariant beperkt zich dus tot de controle van de `Knoop`pointers. Een representatie is geldig als en slechts als:

(1) In de eerste knoop (als hij bestaat) `vorige` een nulpointer is.
(2) In de laatste knoop `volgend.k` een nulpointer is.
(3) Bij twee opeenvolgende knopen de twee tussenliggende pointers naar de andere knoop gericht zijn.

We kunnen dit ook in code schrijven:

```
bool Intlijst::R(){
    bool geldig=true;
    Knoop* nuknoop=this->k;
    while (nuknoop != 0 && geldig){
        geldig=(nuknoop->volgende.k==0 ||
                nuknoop->volgende.k->vorige==nuknoop);
//***************************(zie verder in de tekst)
        nuknoop=nuknoop->volgende.k;
    }
    return geldig;
```

Tot op zekere hoogte kan men zeggen dat daarmee R niet volledig omschreven is: in de code testen we niet expliciet of alle pointers wel geldige waarden aannemen. Het is wel zeer waarschijnlijk, maar niet helemaal zeker, dat de functie `R` zal crashen bij uitvoering als dit niet het geval is. Maar we zijn nog een belangrijker element vergeten. Dit is dat er altijd slechts exact één `Intlijst` naar een `Knoop` mag wijzen. Als er `Knoop` is waar geen `Intlijst` naar wijst dan kunnen we deze `Knoop` niet bereiken; als er meerdere `Intlijst`en naar wijzen levert dit problemen op, bijvoorbeeld bij destructie. We moeten dus nog een vierde voorwaarde toevoegen:

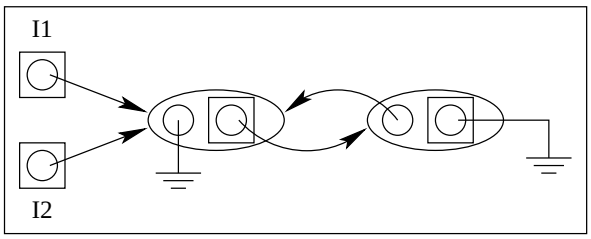
4 Er is een 1-1-relatie tussen knopen en Intlijsten met `k` verschillend van nul.

Deze voorwaarde is veel moeilijker te controleren als de andere, maar het is niet onmogelijk. Er moet het volgende gebeuren:

- In de klasse `Intlijst` wordt een lijst `allelijsten` bijgehouden van pointers naar bestaande `Intlijst`en. Deze lijst is een `static`variabele van de klasse. In de constructor wordt de nieuwe `Intlijst` geregistreerd in de lijst, bij de destructor wordt de referentie uit de lijst verwijderd.
- In de klasse `Knoop` wordt een analoge lijst van knopen bijgehouden.

- Bij controle worden de twee lijsten met elkaar vergeleken.

Merk op dat nu problematisch is om bij conflicten uit te maken *welke* `Intlijst` ongeldig is. Nemen we bijvoorbeeld Figuur 19.3. Het is duidelijk dat `I1` en `I2` naar dezelfde



Figuur 19.3. Verbroken voorwaarde met `Intlijst`en.

knoop wijzen. Als we een van de twee verwijderen of naar ergens anders laten wijzen is het probleem opgelost. We kunnen dus niet zeggen dat `I1` verkeerd is en `I2` juist, of omgekeerd. In de praktijk is dit geen probleem: het is de `Intlijst` die het laatst veranderd is die de problemen oplevert, en dus als fout moet gekenmerkt worden. Merk op dat het niet noodzakelijk bij de eerste knoop verkeerd gaat: we moeten voor elke knoop in de lijst nagaan of er geen `Intlijst` van buitenuit naar de knoop wijst. We moeten dus de met sterretjes gemarkeerde plaats in de code voor `Intlijst::R` vervangen door een controle. In pseudocode:

```
voor alle pointers p in Intlijst::allelijsten
    als (p->k==nuknoop
            && p wijst niet naar de voorliggende Intlijst)
        geldig=false;
```

19.2 **REP**, STATEN EN OVERERVING

Zoals we gezien hebben zijn er twee belangrijke soorten van overerving: restrictie en uitbreiding. Deze twee hebben verschillende eigenschappen ten aanzien van representaties. We gaan in de rest van deze paragraaf uit van een bovenklasse A en een onderklasse B.

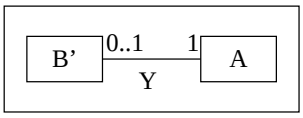
De klassieke vorm van overerving is overerving door uitbreiding. Hierbij heeft elk element van de klasse B alle eigenschappen uit A, maar er zijn ook bijkomende eigenschappen. Indien deze eigenschappen alleen nieuwe operaties zijn, dan verandert er niets aan de representatie: we hebben $\mathbf{Abs}(A) = \mathbf{Abs}(B)$, $\mathbf{Rep}(A) = \mathbf{Rep}(B)$, $R_A = R_B$ en $A_A = A_B$. Nu is het zo dat een operatie niet altijd zinvol hoeft te zijn: als een klasse verschillende staten heeft dan kan het zijn dat een operatie alleen zinvol is in bepaalde staten, en niet in andere. Maar het feit dat B tot stand is gekomen door uitbreiding impliceert dat de nieuwe operaties zinvol zijn voor elk[1] element van $\mathbf{Abs}(A)$. Veronderstel immers dat er elementen van $\mathbf{Abs}(A)$ zijn waarvoor geen enkele nieuwe operatie geldig is. Voor deze elementen is de uitbreiding van B zinloos. Dit legt een restrictievoorwaarde op: De klasse B beschrijft alleen elementen van A waarvoor de

[1] Strikt genomen: voor elk element van $\mathbf{Abs}(A)$ moet er minstens één nieuwe operatie zinvol zijn.

uitbreiding niet zinloos is. B is bijgevolg niet tot stand gekomen door uitbreiding, maar door restrictie, en dit geval behandelen we later.

Indien de uitbreiding dus alleen bestaat uit operaties, dan kan er ook geen sprake van zijn om bepaalde operaties te overschrijven, want dat breekt de consistentievoorwaarde. Immers, stel dat we een operatie `foo()` overschrijven. Er zijn dan twee methodes van de operatie `foo()`, `B::foo()` en `A::foo()`, die verschillende acties uitvoeren. Maar **Abs**$(A) =$ **Abs**(B). Nemen we daar een element uit, dan kunnen we dit representeren door een element b uit B, maar ook door een element a uit A. Het maakt nu een verschil of we `b.foo()` uitvoeren of `a.foo()`. De consistentievoorwaarde voor verschillende methodes bij eenzelfde operatie zegt echter dat de verschillende methodes gelijkaardige acties moeten uitvoeren. Voor één enkel object moet dit dezelfde actie zijn. Wat we dus hebben is, als er uitbreiding is maar geen nieuw gegevensveld, dat B gewoon meer eigenschappen van objecten uit de reële wereld beschrijft dan A, maar dat het wel om exact dezelfde objecten gaat. Bijgevolg is A eigenlijk overbodig: het gaat hier om een typisch geval van opportuniteitsovererving.

Bij echte uitbreiding wordt dus het datagedeelte uit A uitgebreid met een datagedeelte. We hebben dan een alternatief voor overerving, namelijk het schema uit Hoofdstuk 8, waarbij we in plaats van een object b uit de klasse B een object a uit de klasse A hebben, plus een element b' dat alle datavelden en operaties bevat die de uitbreiding uitmaken. In dit geval geldt dat **Abs**$(B) \subset$ **Abs**(A). Dit wordt in Figuur



Figuur 19.4. Datamodel van overerving door uitbreiding.

19.4 aangegeven door de multipliciteit aan de A-kant. De associatie hebben we Y genoemd. Met elke b' uit B' komt dan een $a = Y(b')$ uit A overeen.

Alleen elementen van A waarmee een element uit B' overeenkomt zijn lid van **Abs**B. Als we nu **Rep**(B) bekijken dan zien we dat elk element daaruit is samengesteld uit twee delen: er is het deel dat overeenkomt met de gegevensvelden uit A, en een deel dat overeenkomt met de gegevensvelden uit B'. We kunnen dus elk element $b \in$ **Rep**(B) schrijven als een koppel (a, b'), met $a \in$ **Rep**(A) en $b \in$ **Rep**(B'). Met andere symbolen:

$$\textbf{Rep}(B) = \textbf{Rep}(A) \times \textbf{Rep}(B').$$

Wat de functie R_B betreft, een willekeurig koppel (a, b') uit **Rep**$(A) \times$ **Rep**(B') moet aan *drie* voorwaarden voldoen:

- $R_A(a)$. Inderdaad, a moet een element van **Abs**(A) voorstellen.
- $R_{B'}(b')$. b' moet een geldige toevoeging aan een A-element zijn.
- $Y(b') = a$. De twee delen moeten bij elkaar horen.

Symbolisch wordt dit

$$R_B((a, b')) \equiv R_A(a) \wedge R_{B'}(b') \wedge Y(b') = a.$$

Bekijken we tenslotte nog de abstractiefunctie A_B die een element b uit **Rep**(B) waarvoor R_B waar is afbeeldt op het overeenkomstige element uit **Abs**(B). Nu is **Abs**$(B) \subset$ **Abs**(A), en hangt de afbeelding van (a, b') alleen af van het eerste element van het koppel:

$$A_B((a, b')) = A_A(a).$$

Tot zover uitbreiding.

Bij zuivere restrictie, zoals we gezien hebben, gaat het erom om extra voorwaarden op te leggen aan objecten uit de bovenklasse. Als voorbeeld kunnen we als klasse A de verzameling van alle ellipsen nemen, en voor B de verzameling cirkels. Op het niveau van **Abs** betekent dit dat elk element van **Abs**(B) ook in **Abs**(A) zit, **Abs**$(B) \subset$ **Abs**(A): elke cirkel is een ellips. In principe zijn er geen extra datavelden nodig[2] bij de implementatie van B, zodat **Rep**$(B) =$ **Rep**(A). Het verschil zit hem in de invarianten R_A en R_B. Beide functies werken op elementen van dezelfde verzameling, maar duidelijk is de rep-invariant R_B van B sterker dan R_A, vermits hij ook nog de restrictievoorwaarde omvat. Er geldt:

$$\forall\, x \in \textbf{Rep}(B) : R_B(x) \equiv (R_A(x) \wedge \text{restrictievoorwaarde}(x)).$$

Zodoende is elke x waarvoor $R_B(x)$ voldaan is ook een element van **Rep**(A) waarvoor $R_A(x)$ geldt, en bovendien is

$$\forall\, x \in \textbf{Rep}(B) : R_B(x) \Rightarrow A_B(x) = A_A(x).$$

Zoals bekend is er een grote gelijkenis tussen overerving met restrictie en het gebruik van verschillende staten. In ons voorbeeld van cirkels en ellipsen zouden we, in plaats van een deelklasse `Cirkel` in te voeren ook een staat `cirkel` van de klasse `Ellips` kunnen definiëren. Het voornaamste verschil tussen de twee technieken is dat, als we met een staat werken, we cirkels kunnen vervormen tot ellipsen die niet meer cirkelvormig zijn. Dit lukt niet als we met een klasse werken: dan zou het object van de klasse `Cirkel` overgaan naar de klasse `Ellips`, en dit is niet voorzien. We zouden natuurlijk de cirkel kunnen deleten en een ellips creëren, maar dit is een actie die we in programma's niet willen ondernemen, omdat dan alle verwijzingen naar het oorspronkelijke object ongeldig worden.

Gaan we nu uit van een klasse A met n verschillende staten, S_1, S_2, ..., S_n. We kunnen dan de ruimte **Abs**(A) indelen in verschillende deelruimten. We noemen **Abs**(A, S_i) de verzameling van alle objecten die zich in staat S_i bevinden. Het is duidelijk dat elk object in **Abs**(A) zich in een of andere staat bevindt, en dus is

$$\textbf{Abs}(A) = \bigcup_{i=1}^{n} \textbf{Abs}(A, S_i).$$

Als documentatie bij de staten hoort uiteraard een beschrijving van elke staat. Om te testen en debuggen is het belangrijk deze te implementeren op dezelfde manier als de rep-invariant. We creëren daarom de *staatinvarianten*

$$R_{S_i} : \textbf{Rep} \rightarrow \{\texttt{true}, \texttt{false}\} : r_{s_i}(a) \equiv R_A(a) \wedge A(a) \in \textbf{Abs}(A, S_i).$$

[2] Als er extra gegevensvelden zijn dan kunnen we dit beschouwen als de opeenvolging van twee keer overerving. Eerst is er een overerving van A naar een impliciete tussenklasse, met restrictie, en daarna is er overerving met uitbreiding van deze impliciete klasse naar B.

in woorden: $R_{S_i}(a)$ is waar als en slechts als de rep-invariant waar is voor a, en als de abstractiefunctie a afbeeldt op een reëel object in staat S_i.

19.3 HET GEBRUIK VAN R BIJ SPECIFICATIE EN V& V

Het definiëren van de rep-invariant en de eventuele staatinvarianten is van het grootste belang bij het testen van code. Nadat hij gedefinieerd is moet hij geïmplementeerd worden, bijvoorbeeld in een lidfunctie `bool repOK()`. Noteer dat de functie zeker geen argumenten meekrijgt: ze moet controleren of `*this` voldoet aan de rep-invariant, en dit hangt natuurlijk niet af van uitwendige parameters. Het definiëren moet zorgvuldig gebeuren: het is vooral van belang dat `repOK()` nooit verkeerdelijk `true` teruggeeft. Dit is een algemene regel bij testen: beter een vals alarm zo nu en dan, dan een aantal fouten die niet opgespoord worden. Als de voorwaarden voor de rep-invariant te streng zijn opgesteld, dan zal dit snel blijken bij het testen, en dan kan het probleem verholpen worden. Hier zijn een aantal richtlijnen om een goede rep-invariant te vinden:

- Zoek naar zoveel mogelijk elementen uit **Rep** waarvoor de abstractiefunctie geen zin heeft. Bijvoorbeeld: personen hebben nooit een negatieve lengte. Als we in onze `Persoon`sklasse een object hebben met negatief `grootte`veld, dan is dit dus verkeerd.
- Let op de voorwaarden die horen bij algoritmen die je gebruikt. Als je bijvoorbeeld een zoekfunctie hebt bij `Intset` die ervan uitgaat dat de `Intset` minstens één element bevat, dan moet deze voorwaarde in de rep-invariant zitten (ofwel is de zoekfunctie verkeerd geïmplementeerd natuurlijk).
- Zoek naar gegevens die synchroon moeten veranderen. Veronderstellen we bijvoorbeeld dat een gelinkte lijst buiten zijn knopen nog een veld `aantalknopen` bijhoudt met daarin het aantal knopen. Als we een knoop toevoegen of verwijderen dan moeten we tegelijkertijd dit veld aanpassen. Het is dan evident dat de rep-invariant zal eisen dat dit veld wel degelijk het aantal knopen bevat. Ga, om synchroniciteit te vinden, alle gegevensvelden af, en kijk of je hun waarde kan veranderen zonder aanpassingen aan de andere gegevens.
- Zoek naar voorwaarden in de ruimte **Abs**. Zo zal bijvoorbeeld bij een lessenrooster er geen overlapping mogen zijn van de lesuren van docenten en studenten; op een schaakbord staan nooit meer dan 64 stukken, en zo meer. Dit beperkt de mogelijke toestanden, en bijgevolg de rep-invariant.
- Kijk in de code naar situaties die de code zouden doen crashen of een exceptie opgooien. Een typisch voorbeeld zijn nulpointers. Het klassendiagram biedt hier natuurlijk informatie: als een bepaalde pointer een associatie met multipliciteit $0 \ldots 1$ implementeert, mag dit wel een nulpointer zijn. Andere voorbeelden zijn delingen: als je door een getal deelt mag dit niet nul zijn. Dit moet uit de rep-invariant volgen. Ook buiten de grenzen van een tabel gaan moet voorkomen worden door de rep-invariant.

Als we een procedure uitschrijven in code kunnen we niet verwachten dat de rep-invariant altijd voldaan blijft. We kunnen in een opdracht slechts één wijziging tegelijkertijd doorvoeren, en als de rep-invariant een relatie tussen verschillende delen aangeeft, wordt deze relatie hoe dan ook tijdelijk doorbroken. Nemen we het voor-

beeld van onze `Intlijst`, en veronderstel dat we een `Knoop` moeten overhevelen van een lijst `I1` naar een lijst `I2`. We kunnen kiezen: ofwel laten wijzigen we eerst `I1` zodat deze niet meer naar de knoop wijst (maar dan wijst geen enkele `Intlijst` naar de knoop), ofwel wijzigen we eerst `I2` (maar dan wijzen er twee `Intlijsten` naar de knoop): in elk geval is aan de rep-invariant niet meer voldaan.

Nu hebben we gezien dat we, zowel bij het ontwerpen van een procedure als bij mathematische inspectie, gebruik kunnen maken van tussentoestanden. Het is duidelijk dat, als we op een bepaald moment een rep-invariant doorbreken, we deze zo snel mogelijk moeten herstellen. In ons voorbeeld gaan we `I1` en `I2` dus wijzigen in twee opeenvolgende programmaopdrachten. Als het mogelijk is gaan we dan ook tussentoestanden definiëren op zo'n manier dat bij elke tussentoestand aan de rep-invariant voldaan is. Merk op dat dit geen wet van Meden en Perzen is. Soms is het bereiken of herstellen van de rep-invariant zo'n ingewikkelde zaak dat het niet efficiënt is om dit tussen twee opeenvolgende tussentoestanden in te klemmen. Het kan zelfs zijn dat de inbreuk op de rep-invariant wordt doorgegeven *tussen* verschillende methodes van het object. Dit is vaak het geval bij complexe datastructuren, waarbij er een structuurvoorwaarde is. Nemen we het voorbeeld van een heap: een volledige binaire boom waarbij de sleutel van elke knoop, buiten de wortel, niet groter is dan die van zijn ouder. Als we daarin de wortel verwijderen dan moet deze vervangen worden, en dit vereist een vrij complexe herordening als de heap geïmplementeerd is met een tabel. Het is dan best mogelijk dat er operaties zijn waarbij de rep-invariant *niet* in de precondities staat. Het kan zelfs zijn dat de uitvoering van een operatie zinloos, of zelfs verboden, is als de rep-invariant wel voldaan is.

Het spreekt vanzelf dat de invariant alleen verbroken mag zijn op ogenblikken dat het object er controle over heeft dat hij hersteld wordt. Bijgevolg hoort de rep-invariant bij de pre- en postcondities van elke publieke operatie, en moet hij daar niet expliciet vermeld worden. Als er private methodes zijn waarbij de rep-invariant in pre- of postcondities niet noodzakelijk voldaan is, dan dient dit dan ook duidelijk te worden opgegeven.

19.4 BLOOTSTELLING VAN REP

In dit verband is het belangrijk te wijzen op het gevaar van wat bekend staat als *blootstelling van rep* (Eng: *rep exposure*). Een object mag de rep-invariant tijdelijk doorbreken bij de uitvoering van een operatie als het ervoor zorgt dat het deze herstelt. Het moet er ook voor zorgen dat er geen andere objecten zijn die de rep-invariant kunnen doorbreken. Indien een ander object een wijziging kan aanbrengen die de rep-invariant doorbreekt, dan zeggen we dat hij blootgesteld is aan wijziging. Er zijn twee belangrijke bronnen van zo'n blootstelling:

- Het publiek maken van datavelden. Hierdoor kunnen andere objecten deze velden wijzigen. Dit is een elementaire fout die zelden voorkomt, behalve om efficiëntieredenen.
- Een ander object heeft een *pointer* naar een dataveld of deelobject waardoor het dit kan veranderen.

De tweede fout is veel subtieler en wordt niet gemakkelijk opgespoord. Nemen we als

voorbeeld een datastructuur waar sorteerbare sleutels met bijbehorende data worden opgeslagen. We definiëren een (sleutel,data)-paar `Sleuteldata` en een `Container` die 200 (sleutel,data)-paren kan bevatten. Als implementatie kiezen we een gesorteerde `SleutelData`tabel:

```
class Container{
public:
    class SleutelData{
        public:
            SleutelData():sleutel(0),data(0){}
            bool operator<(const SleutelData& sd){
                return ((sleutel!=0 && sd.sleutel==0) ||
                    (sleutel!=0 && sd.sleutel!=0
                        && *sleutel < *sd.sleutel));
            }
            void zetPointers(Sleutel* sl ,Data* d);
        //vereist: beide pointers nul,
        //         of beide wijzend naar een geldig object.
            Sleutel* getSleutel() const{
                return sleutel;
            }
            Data* getData() const{
                return data;
            }
        private:
            Sleutel* sleutel;
            Data* data;
        };
    int zoekindex(const Sleutel&);

    SleutelData tabel[200];
public:
    Sleutel* zoek1(const Sleutel& sl){
        return tabel[zoekindex(sl)].getSleutel();
    }
    const Sleutel* zoek2(const Sleutel& sl){
        return tabel[zoekindex(sl)].getSleutel();
    }
    SleutelData* zoek3(const Sleutel& sl){
        return &tabel[zoekindex(sl)];
    }
    const SleutelData* zoek4(const Sleutel& sl){
        return &tabel[zoekindex(sl)];
    };
    Data* zoek5(const Sleutel& sl){
        return tabel[zoekindex(sl)].getData();
    }
```

```
};
```

Het heeft geen zin om data zonder sleutel te hebben, of een sleutel zonder data. Daarom eist de rep-invariant van `SleutelData` dat beide bestaan of niet bestaan. Omdat de rep-invariant moet voldaan zijn voor en na de uitvoering van elke publieke operatie is alleen een operatie voorzien die *beide* pointers wijzigt: het is onmogelijk om, als beide pointers nul zijn, een van de pointers te wijzigen zonder de rep-invariant te doorbreken. Voor `Container` zegt de rep-invariant dat de tabel gesorteerd moet zijn. Merk op dat dit inhoudt dat de niet ingevulde `SleutelData`objecten, die nulpointers bevatten, achteraan staan.

`zoek1` geeft een pointer terug naar een `Sleutel` uit de tabel. Dit is blootstelling: immers het cliëntobject kan nu de sleutel wijzigen, waardoor de sorteervoorwaarde doorbroken wordt. Wel veilig is `zoek2`: de cliënt kan het object bekijken maar niet wijzigen. `zoek3` is zo mogelijk nog onveiliger dan `zoek1`: de cliënt kan nu ook nog de `Sleuteldata` wijzigen, en bijvoorbeeld naar een andere sleutel laten wijzen. Enigszins verrassend is dat `zoek4` ook blootstelling geeft. Immers: de cliënt kan aan het constante `SleutelData`object het adres van de `Sleutel` vragen, en dit is *geen* `const`-pointer! De `const`voorwaarde op `SleutelData` verbiedt om datavelden van dit object te wijzigen, maar niet om de objecten waar ze naar wijzen te wijzigen. Een opdracht als

```
Sleutel a,b;
*(c.zoek4(a)->getSleutel())=b;
```

is dus geldig, en vervangt de `a`-waarde in de Container door de waarde van `b`. `Zoek5` tenslotte stelt de rep-invariant niet bloot aan verbreking. Deze functie laat toe dat een cliënt het datagedeelte verandert, maar dit beïnvloedt de rep-invariant niet.

Tot nu toe hebben we voorbeelden gezien waarbij een cliënt het adres krijgt van een item dat hij niet mag wijzigen: de blootstelling werd *geëxporteerd* naar de cliënt. Het is echter ook mogelijk om rep-blootstelling te *importeren*. Dit gebeurt als een bestaand object wordt ingecorporeerd in de rep-invariant. Zouden we bijvoorbeeld in de toevoegfunctie van `Container` code schrijven van de vorm

```
void Container::voegtoe(Sleutel* sl, Data* d){
    int plaats=...;\\plaats duidt aan waar moet toegevoegd
    tabel[plaats].zetpointers(sl,d);
}
```

dan is er ook blootstelling van rep. Immers, de sleutel waar `sl` naar wijst bestond reeds, en is gekend door de cliënt. Bijgevolg kan hij deze sleutel wijzigen buiten de controle van de `Container` om. Zo leidt bijvoorbeeld de code

```
Sleutel sl;
Data d;
c.voegtoe(&sl,&d);
```

tot problemen. Er is hier blootstelling van rep, en als de methode waar de code toe behoort eindigt worden `sl` en `d`, die statisch zijn aangemaakt, vernietigd.

19.5 FORMELE SPECIFICATIE

We hebben gezien dat het specificeren van procedures (inclusief lidfuncties/methodes) twee dingen inhoudt:

- De preconditie die aangeeft wat de procedure van de buitenwereld mag verwachten.
- De postconditie die aangeeft wat de buitenwereld van de procedure mag verwachten.

De preconditie geeft informatie over toegelaten waarden van parameters en de toegelaten staat van het `*this`object evenals toegestane hulpmiddelen (tussenbestanden, ...). Het is uiteraard de bedoeling dat de condities zo duidelijk mogelijk worden uitgedrukt. Daarom is het nodig om afspraken te maken over wat er *niet* vermeld wordt in de condities. Hierbij komt de rep-invariant te hulp. Bij zowel pre- als postcondities mag men veronderstellen dat de rep-invariant voldaan is voor alle objecten die bij de procedure betrokken zijn. Opmerkingen hierbij:

- Bij de preconditie van een constructor hoort dat het `*this`object nog niet bestaat. Het moet dus ook niet aan de rep-invariant voldoen.
- Bij de postconditie van een destructor bestaat het object niet meer, zodat ook hier niet meer aan de rep-invariant moet voldaan zijn.

Als het gaat over een eenvoudige procedure (die geen in- of uitvoer van het programma veroorzaakt) is de lijst van toegestane hulpmiddelen leeg (buiten een redelijk gebruik van geheugenallocatie). Vaak is het nuttig om in één oogopslag te kunnen zien welke objecten zulk een procedure kan wijzigen. Als het gaat over een lidfunctie die een waarde teruggeeft moet men ook snel kunnen zien wat voor waarde ze teruggeeft. Vaak is dit min of meer duidelijk uit de naam, maar uitzonderlijke gevallen moeten nader gespecificeerd worden. Op deze manier krijgen we *vier* mogelijke delen aan de specificatie. In de onderstaande tabel worden deze weergegeven, met hun Engelse kenwoord, de Nederlandse versie, en de defaultwaarde als ze niet expliciet vermeld worden:

Deel	Engels	Nederlands	default
Preconditie	requires	vereist	rep-invariant
Kader	modifies	wijzigt	(leeg)
Postconditie	effects	zorgt voor	rep-invariant
teruggeefwaarde	returns	geeft terug	(geen waarde)

We overlopen deze tabel. Bij de precondities wordt vermeld wat de mogelijke waarden zijn van parameters en wat de staat is van alle betrokken objecten. Als er geen preconditie vermeld is zijn er geen voorwaarden (behalve de rep-invarianten van alle objecten). Bijgevolg zijn alle mogelijke staten en waarde toegestaan, en is aan de preconditie altijd voldaan. Het *kader* (Engels: *frame*) geeft aan welke deelobjecten (en datavelden van `*this`) kunnen gewijzigd worden. Dit is een deel van de postconditie, vermits het aangeeft wat de procedure *niet* mag wijzigen: wat niet in de wijzigtlijst staat mag niet gewijzigd worden. Als de lijst ontbreekt mag de procedure niets wijzigen. Welke wijzigingen de procedure kan aanbrengen wordt aangegeven in het zorgt-voorgedeelte. Let er hierbij voor op dat dit volledig moet zijn: elke wijziging

die alleen dingen uit de wijzigtlijst wijzigt en voldoet aan de zorgt-voorvoorwaarden is correct. Als bijvoorbeeld in de wijzigtlijst staat dat `x` en `y` mogen wijzigen, en in de zorgt-voorsectie staat niets over `x`, dan betekent dit dat `x` een willekeurige waarde mag aannemen. Uiteraard ontbreekt dit deel wanneer het kader leeg is. Bovendien wordt in de zorgt-voorsectie aangegeven welke berichten naar andere objecten door de operatie verstuurd kunnen worden. De teruggeefwaarde specificeert wat de procedure moet teruggeven, en ook deze kan ontbreken. Merk op dat een procedure met een leeg kader en zonder teruggeefwaarde bijna altijd zinloos is. In sommige gevallen kan dit echter zijn nut hebben. Als we bijvoorbeeld een bovenklasse hebben met een functie die een wijziging aanbrengt, dan kan het nuttig zijn deze te overschrijven met een functie die niets doet. Denken we bijvoorbeeld aan een klasse `GeometrischeFiguur` waar een `roteer`functie op gedefinieerd is. Creëren we nu een deelklasse `Cirkel`, dan moet deze overschreven worden, en `Cirkel::roteer(double hoek)` wordt een functie die niets doet. Omdat dit echter zo zeldzaam is, is het nodig om dit expliciet te vermelden. Men gaat dan niet kader, postconditie en teruggeefwaarde weglaten uit de specificatie, maar expliciet opgeven dat (en waarom) ze ontbreken.

Het is gebruikelijk de operaties van een klasse onder te verdelen in vier verschillende soorten:

- *Creatoren*. Een creator creëert een nieuw object van de klasse. Er kunnen parameters zijn, en dus verwijzingen naar andere objecten, maar daar mogen geen objecten van de eigen klasse bij zijn.
- *Producenten*. Een producent creëert een nieuw object uitgaande van een of meer objecten van de eigen klasse. Een copy constructor is dus geen creator, maar een producent. Een producent kan ook verschillende objecten gebruiken: als we onze klasse `Intset` bekijken, dan kan er een producent zijn die uit twee `Intsets` de unie produceert, en dit is weer een `Intset`.
- *Modificatoren*. Deze veranderen het `*this`object.
- *Kijkoperaties*. Deze geven informatie over het object terug. Het begrip is iets ruimer dan het begrip query. Bij een query denken we aan eenvoudige informatie, maar een kijkoperatie mag ook echte objecten teruggeven. Hebben we bijvoorbeeld een factory voor een of andere klasse, dan is een operatie die aan de factory vraagt om een object aan te maken een kijkoperatie van de factory, omdat het resultaat een beeld geeft van de toestand ervan, en de toestand van de factory niet wijzigt. Voor de klasse van de objecten die gemaakt worden is deze operatie uiteraard een creator.

Een specificatie kan *ondergedetermineerd* zijn. Dit betekent dat ze aan de programmeur enige vrijheid laat inzake implementatie. Nemen we de volgende specificatie van een functie:

```
int P::zoek(uint a)
// vereist:      a komt voor in de tabel
// geeft terug:  index zo dat tab[index]=a
```

De specificatie laat de mogelijkheid open dat `a` meer dan een keer in de tabel voorkomt. In dit geval zijn er verschillende geldige waarden die kunnen teruggegeven worden. Ondergedetermineerde specificaties kunnen een voordeel zijn. Door de keuze onnodig te beperken, kan men de programmeur verplichten een minder efficiënte

methode te gebruiken (naarmate een methode complexer is, kost het meer tijd ze te implementeren en is er meer kans op fouten. Dit is een vorm van inefficiëntie die soms veel belangrijker is dan inname van computertijd of -geheugen). Zo is een specificatie van een sorteerprocedure die alleen stipuleert dat het resultaat gesorteerd moet zijn ondergedetermineerd als er even grote, maar toch verschillende elementen kunnen zijn. Dit kan gebeuren bij namen (waar blanco's uit verwijderd worden bij sorteren, zodat "Vanbeke" even groot is als "Van Beke"), of bij een associatieve container waar de een sleutel twee keer kan voorkomen met verschillende data, en alleen de sleutel wordt gebruikt om de ordening te bepalen. Men kan de ondergedetermineerdheid verwijderen door stabiliteit te eisen (even grote elementen hebben dan dezelfde onderlinge positie als voor het sorteren), maar dan kan een algoritme zoals quicksort niet meer gebruikt worden.

Ondergedetermineerde specificaties houden echter een gevaar in. Cliënten van de code gaan vaak onterecht uit van een bepaalde interpretatie van ondergedetermineerde specificaties, en dit leidt tot fouten. Bovendien zijn ondergedetermineerde specificaties vaak het gevolg van slordigheid. Bijgevolg is het een goede praktijk om de speling in de specificaties *expliciet* weer te geven. Dit duidt aan dat de ongedetermineerdheid geen gevolg is van slordigheid, en maant de gebruiker aan tot voorzichtigheid. Bovenstaande specificatie wordt dus beter herschreven als

```
int P::zoek(uint a);
// vereist:      a komt voor in de tabel
// geeft terug:  de index,
//               of een van de indexen waarvoor tab[index]=a
```

In plaats ondergedetermineerd noemt men dergelijke specificaties ook *niet-deterministisch*. Deze term is enigszins misleidend: niet-deterministische code is code die niet altijd hetzelfde doet. Het is wel zo dat ondergedetermineerde specificaties kunnen voldaan worden door niet-deterministische code. Zo voldoet volgende code aan de specificatie die we zojuist gegeven hebben:

```
int P::zoekC(uint a){
    if (rand()%2 == 0)
        return zoekA(a);
    else
        return zoekB(a);
}
```

Bij het testen op ondergedetermineerde specificaties treedt een probleem op, omdat er niet meer sprake is van *het* juiste resultaat, maar van *een* juist resultaat. Men moet de dus de eigenschappen van het resultaat nagaan. Een orakel kan niet gebruikt worden omdat verschillende resultaten niet noodzakelijk wijzen op fouten.

Bij formele beschrijving van specificaties kan men uitgaan van twee benaderingen. In de eerste benadering wordt gebruik gemaakt van *gebeurtenissen*. Objecten worden beschouwd als iets waar we de binnenkant niet van kennen. Al wat we weten is welke operaties er (kunnen) worden uitgevoerd en wat deze opleveren. De meestgebruikte talen in deze benadering maken gebruik van *procesalgebras*. De tweede steunt op een model van de toestand van het object. Men spreekt soms van op toestand gebaseerde

talen, maar ook van modeltalen. In zo'n model worden meestal een aantal attributen van de klasse gedefinieerd. De specificaties voor de operaties worden dan uitgedrukt in termen van deze attributen.

19.6 ALGEBRAÏSCHE SPECIFICATIE

Bij algebraïsche specificatie gaat men een klasse specificeren aan de hand van haar operaties. Buiten de opsomming van de operaties zelf horen daarbij ook een aantal *axioma's*. Een axioma is een uitspraak die altijd waar moet zijn, en die bestaat uit de vergelijking van twee waarden. Als men zeer strikt is gaat men uit van een aantal basistypes (zoals gehele getallen of strings) die men kan vergelijken. Elk axioma vergelijkt dan twee entiteiten van een of ander basistypes. Vaak gaat men echter objecten van een bepaalde klasse vergelijken: men veronderstelt dat de nodige *comparator* (zoals . . . is gelijk aan . . . , of . . . is kleiner dan . . .) zinvol gedefinieerd is op deze klasse. Elke operatie heeft een resultaat. Als we gaan kijken naar de classificatie in vier soorten van operaties (creatoren, producenten, modificatoren en kijkoperaties) voor een gegeven klasse `P` dan gaat men ervan uit dat het resultaat van de eerste drie soorten steeds een `P` is. Het resultaat van een kijkoperatie kan elke mogelijk soort van klasse of type zijn.

Het `*this`argument van de operatie van een klasse gaat men expliciet als eerste parameter van een operatie schrijven. Dit komt overeen met het werk van een preprocessor die een C++-programma, met klassen, omzet naar een C-programma met `structs`. C-`structs` hebben wel gegevensvelden maar geen operatoren. Als we in C++ bijvoorbeeld schrijven `p.foo(5)`, waarbij p van de klasse `Bar` is, dan wordt dit in C omgezet naar `Bar_foo(p,5)`. Analoog gaat men bij algebraïsche specificatie te werk. Er is geen verschil te merken tussen modificatoren en producenten: beide hebben een of meer argumenten van de klasse, en geven als resultaat een argument van de klasse terug. Nemen we een voorbeeld: de specificatie van een gelinkte lijst, waarin achteraan wordt toegevoegd. Algebraïsche specificatie begint met de naam van de klasse, dan is er een lijst van operaties met hun signatuur. Er wordt aangegeven welke de parameters zijn en wat het resultaat is, en daarna is er een lijst van specificaties in de vorm van axioma's.

klasse Lijst<class T>
creëer()→Lijst voegtoe(Lijst, T)→Lijst staart(Lijst)→Lijst eerste(Lijst)→T aantalElementen(Lijst)→int
(1)eerste(creëer())=exceptie (2)eerste(voegtoe(L,e))= **if** (L==creëer()) **then** e **else** eerste(L) (3)aantalElementen(creëer())=0 (4)aantalElementen(voegtoe(L,e))=aantalElementen(L)+1 (5)staart(creëer())=creëer() (6)staart(voegtoe(L,e))= **if** (L==creëer()) **then** creëer() **else** voegtoe(staart(L),e)

Voor de eenvoud veronderstellen we hier dat `exceptie` zinvol kan vergeleken worden met een element van een willekeurige klasse.

In onze lijst wordt er wel toegevoegd, maar niet verwijderd. Se hebben een creator, `creëer`, die een lege lijst creëert, en een modificator, `voegtoe`. Er is een producent, `staart`. Deze geeft een lijst terug, gelijk aan het argument, behalve dat het eerste element verwijderd is. Merk op dat het geen belang heeft, voor de formele specificatie, of de teruggavewaarde een gewijzigde versie is van `*this`, of dat `*this` ongewijzigd blijft bestaan en er een nieuwe, kortere lijst wordt aangemaakt. We zouden `staart` dus ook als een modificator kunnen beschouwen. Moesten we een model hebben van de interne structuur zouden we in de specificaties expliciet kunnen vermelden of toevoegen vooraan of achteraan gebeurt. Nu wordt dit impliciet aangegeven door axioma (2).

Eén van de aspecten van dit soort specificaties is de vraag van *volledigheid*: is het zo dat we voor een willekeurige reeks van operaties kunnen afleiden wat het resultaat is? Kunnen we bijvoorbeeld uit de bovenstaande axioma's afleiden dat

`aantalElementen(staart(voegtoe(L)))=AantalElementen(L)`?

Onvolledigheid, of incompleetheid, houdt verband met het ondergedetermineerd zijn van bepaalde operaties. Als we een formele methode hebben om te zien of ons axiomastelsel volledig is, kunnen we dus ondergedetermineerdheid opsporen en zo nodig verhelpen. We moeten dus kunnen onderzoeken of we voor alle zinvolle reeksen van operaties die een resultaat opleveren dit resultaat kunnen afleiden uit de axioma's. Om te beginnen is het zo dat om zinvol te zijn we een reeks operaties moeten hebben die allemaal een `Lijst` opleveren, behalve de laatste die een kijkoperatie moet zijn. Dergelijke reeks van operaties noemen we een *kijkreeks*. Nemen we het voorbeeld van een `Lijst` die we creëren, waar we twee elementen `e1` en `e2` aan toevoegen, waarna we de `staart` ervan nemen (dit is equivalent met het eerste element verwijderen), om tenslotte het eerste overblijvende element te bekijken. Dit zou dan weer het element `e2` moeten opleveren. Men kan dit formeel uitdrukken: `eerste(staart(voegtoe(voegtoe(creëer(),e1),e2)))=e2`. Kunnen we dit afleiden uit de axioma's? We verwisselen eerst met behulp van axioma (6) (tweede en

eerste geval) de `staartoperatie` met de
voegtoeoperaties, en krijgen in twee stappen

```
eerste(voegtoe(staart(voegtoe(creëer(),e1)),e2))
eerste(voegtoe(creëer(),e2)).
```

Dit is, volgens axioma (2), inderdaad `e2`.

Om in te zien dat we inderdaad compleetheid hebben moeten we de operaties in twee groepen indelen: de kijkoperaties en de andere, die een `Lijst` opleveren. Als we producenten die als argument meer dan een `lijst` buiten beschouwing laten, hebben we alleen creatoren, modificatoren en kijkoperaties. Een zinvolle reeks van operaties start dus (noteer dat de start van een rij operaties rechts staat, en niet links) met een creator. Deze wordt gevolgd door nul of meer modificatoren, en afgesloten met een kijkoperatie. Willen we bewijzen dat de reeks axioma's compleet is dan moeten we bewijzen dat we het resultaat van zulk een rij kunnen afleiden uit het resultaat van een kortere rij, of rechtstreeks uit de axioma's: een bewijs door inductie. Bewijzen we dat de specificatie van `Lijst` compleet is. Zij Q een zinvolle reeks operaties, die m modificatoren bevat. Vermits er maar een creator is, begint Q met `creëer`. De kijkoperatie in Q is ofwel `eerste` of wel `aantalElementen`. Als m gelijk is aan nul wordt het resultaat gegeven door axioma (1) of (3). Zij nu $m > 0$. Er zijn drie mogelijkheden:

(1) De laatste modificator is `voegtoe`. We kunnen het resultaat afleiden uit axioma (2) of (4), en uit het resultaat voor een kortere reeks operaties.
(2) Alle modificatoren zijn `staart`operaties. Toepassing van axioma (6), met de vorm `staart(creëer())=creëer()` doet een `staart`operatie verdwijnen, en levert een kortere reeks op. Uiteindelijk verdwijnen alle `staart`operaties, en wordt het argument van de kijkoperatie `creëer()`.
(3) De laatste modificator is `staart`, maar er komt een `voegtoe` voor in de reeks modificatoren. Door axioma (6) toe te passen, eventueel meerdere keren, kunnen we ofwel de `voegtoe` naar links bewegen, of doen verdwijnen. Als we een `voegtoe` tot meest linkse modificator kunnen brengen zitten we in geval (1), anders verdwijnen alle `voegtoe`s en zitten we in geval (2).

De strategie is steeds dezelfde: we moeten een lange kijkreeks reduceren tot een kortere. Men probeert de axioma's te standaardiseren zodat dit op een systematische manier kan gebeuren. Als de axioma's deze standaardvorm hebben, dan kunnen we de modificatoren indelen in *reduceerbare* en *niet-reduceerbare* modificatoren. Een modificator `mod` is reduceerbaar als twee voorwaarden voldaan is:

(a) Het resultaat van `mod` toegepast op een willekeurige creator kan uitgedrukt worden als het resultaat van een creator. In ons voorbeeld: `staart(creëer())=creëer()`.
(b) Als we een rij modificatoren hebben, die toegepast is op een creator, en de laatste operator is `mod`, dan kunnen we deze steeds vervangen door een rij die niet langer is en waarbij `mod` niet achteraan staat. In ons voorbeeld wordt dit gerealiseerd door axioma (6), dat toelaat `staart()` te verwijderen als laatste element.

In ons voorbeeld is `staart()` een reduceerbare operator, terwijl `voegtoe()` niet-reduceerbaar() is. Merk op dat het voor (b) voldoende is als we voor elke niet-reduceerbare operator `nietreduc` we elke uitdrukking van de vorm `mod(nietreduc(X)`, waarbij

`X` een willekeurige uitdrukking is, kunnen omvormen naar een uitdrukking van de vorm `op1(op2(X))`, waarbij `op1` niet-reduceerbaar is, of een uitdrukking van de vorm `op2(X)`, waarbij `op2` al dan niet reduceerbaar mag zijn.

Als de axioma's gestandaardiseerd zijn, dan hebben we ook voor elke kijkoperatie `kijk1` en voor elke niet-reduceerbare operator `nietred` een axioma dat toelaat het resultaat elke uitdrukking van de vorm `kijk1(nietreduc(X))` uit te drukken in termen van een uitdrukking `kijk2(X)`, waarbij `kijk2` een kijkoperatie is. Meestal is `kijk1=kijk2`, maar dit hoeft niet zo te zijn. In ons voorbeeld is het wel zo: we hebben

`aantalElementen(voegtoe(L,e))=aantalElementen(L)+1.`

Bovendien moet uiteraard het resultaat van elke kijkoperatie toegepast op een willekeurige creator gedefinieerd zijn.

Als de axioma's gestandaardiseerd zijn, dan is het gemakkelijk in te zien hoeveel axioma's er moeten zijn. Als c het aantal creatoren is, r het aantal reduceerbare modificatoren, n het aantal niet-reduceerbare modificatoren en k het aantal kijkoperaties, dan is het aantal axioma's nodig om compleetheid te krijgen gelijk aan

$$(r+k)(c+n).$$

Immers, we hebben voor elke reduceerbare modificator `mod` c axioma's nodig om de actie op creatoren uit te drukken en n axioma's die uitdrukkingen van de vorm `mod(nietreduc(X))` te reduceren. Voor de r reduceerbare operatoren geeft dit $r(c+n)$ axioma's. Ook voor elk van de k kijkoperatoren hebben we $c+n$ axioma's nodig.

Axioma's kunnen zelf ook parameters bevatten. Nemen we het voorbeeld van een koffieautomaat. Deze heeft twee operaties: een koffie nemen, en een bijvuloperatie. Deze laatste heeft een parameter: het aantal bekers koffie dat de automaat zal bevatten na de operatie (dit vergemakkelijkt ons model: het aantal bijkomende bekers zou iets ingewikkelder uit te drukken zijn). Een algebraïsche specificatie van zo'n automaat ziet er dan als volgt uit:

`klasse Automaat`

`creëer()`$\rightarrow$ `Automaat`
`vulbij(Automaat,int)`$\rightarrow$ `Automaat`
`neem(Automaat)`$\rightarrow$`(resultaat,Automaat)`

`neem((`**P2** `neem)`$^{\mathtt{n}}$`(vulbij(Automaat,p)))=`**if** `(n`$\leq$`p)`
then `(koffie, Automaat)`
else `("leeg", Automaat)`
`neem(creëer())=("leeg",creëer())`

We hebben hier een aantal nieuwe notaties ingevoerd. **P** is de projectieoperator, en **P2** is de projectieoperator op het tweede element: als (a,b) een paar is (a en b elementen van willekeurige klassen), dan is **P2**$(a,b)=b$. Analoog is **P1** de projectie op het eerste element. Op deze manier word **P1**`neem` een kijkoperatie, die een `resultaat` teruggeeft, en **P2**`neem` is een zuivere modificator die een `Automaat` teruggeeft. De operatie **P2**`neem` beeldt een `Automaat` af op een `Automaat`. We kunnen ze dus herhaalde malen toepassen, wat wordt weergegeven door de macht n. Merk op dat we de axioma's van de Automaat niet gestandaardiseerd hebben. Om ze te standaardiseren moeten we de axiomaparameters verwijderen. Om dit te doen splitsen we de `vulbij`operatie:

$$\texttt{vulbij(A,p)} \equiv \texttt{vuleenbij}^{\texttt{p}}\texttt{(maakleeg(A))}$$

waarin `maakleeg()` een nieuwe operatie is. Bovendien moeten we `neem` splitsen in de kijkoperatie **P1**`neem` en de modificator **P2**`neem`.

Het is duidelijk dat er één creator is, namelijk `creëer`. We kiezen nu welke modificatoren we reduceerbaar maken, en welke niet-reduceerbaar. `vulbij` wordt gedefinieerd in termen van de andere modificatoren. Hiervoor voeren we een nulde axioma in, en verder moeten we ons niet meer met `vulbij` bezighouden, maar alleen met de drie modificatoren `maakleeg`, `vuleenbij` en **P2**`neem`. `maakleeg()` is duidelijk reduceerbaar, vermits het altijd een lege automaat oplevert, hetgeen hetzelfde is als een nieuwe automaat. `vuleenbij()` is niet-reduceerbaar, want `vuleenbij(creëer())` is een automaat die niet uit te drukken is in termen van een creator. Kunnen we **P2**`neem` reduceren? We weten wat het resultaat is voor **P2**`neem`(creëer()), dus moeten we het reduceren tegenover het niet-reduceerbare `vuleenbij`, wat gemakkelijk is: de twee modificatoren heffen elkaar op. Verder is er één kijkoperatie. In onze formule voor het aantal axioma's is $r = 2$, $k = 1$, $n = 1$ en $c = 1$, wat dus 6 axioma's oplevert. Hier hoort het nulde axioma dat `vulbij` definieert niet bij. `maakleeg` is zo eenvoudig dat we de twee axioma's (een dat `maakleeg(creëer())` definieert en het andere dat `maakleeg()` reduceert tegenover `vuleenbij` op één lijn schrijven. We krijgen de gestandaardiseerde vorm:

`klasse Automaat`

`creëer()`→ `Automaat`
`vulbij(Automaat,int)`→ `Automaat`
`vuleenbij(Automaat)`→ `Automaat`
`maakleeg(Automaat)`→ `Automaat`
`neem(Automaat)`→ `(resultaat,Automaat)`

`(0) vulbij(A,p)` ≡ `vuleenbij`$^{\texttt{p}}$`(maakleeg(A))`
`(1-2) maakleeg(A)=creëer()`
`(3)`**P2**`neem(creëer())=creëer()`
`(4)`**P2**`neem(vuleenbij(A))=A`
`(5)`**P1**`neem(creëer())="leeg"`
`(6)`**P1**`neem(vuleenbij(A))=koffie`

Alhoewel algebraïsche specificatie geen expliciete attributen kent, kan het begrip toestand wel zinvol gedefinieerd worden, en kunnen we soms het bestaan van bepaalde attributen van een klasse bewijzen. Merk op dat dit *niet* impliceert dat deze attributen als zodanig moeten geïmplementeerd worden: wel is het zo dat deze attributen minstens als afleidbare attributen moeten kunnen berekend worden uit de echte attributen. Omdat we vaak gaan werken met sequenties van operaties, eerst enige notaties. Deze zijn gebaseerd op reguliere uitdrukkingen. We definiëren de verzamelingen **Kijk**, **Mod** en **Crea** als de verzamelingen van kijkoperaties, modificatoren en creatoren van de gegeven klasse. In een reguliere operatieuitdrukking (REGOU) staat `kijk` voor een willekeurige kijkoperatie. `mod` en `crea` worden analoog gedefinieerd waarbij er wordt verondersteld dat alle parameters ingevuld zijn. Deze drie symbolen worden atomen genoemd. Een sterretje * na een uitdrukking betekent 0 of meer van dergelijke operaties na elkaar. Zo is bijvoorbeeld

```
kijk mod* crea
```

de REGOU die hoort bij alle kijkreeksen waarvoor een resultaat moet kunnen gegeven worden. We vereenvoudigen hier een beetje: het is immers mogelijk dat een bepaalde sequentie van operaties een ongeldig object oplevert. Dit lossen we als volgt op: we hebben reeds `exceptie` gebruikt bij de specificatie van een gelinkte lijst. We zeggen nu dat een willekeurige operatie kan toegepast worden op `exceptie` en dan weer `exceptie` oplevert. Merk overigens op dat de sequenties van rechts naar links gelezen worden, wat volgt uit onze functienotatie in axioma's, waar we bijvoorbeeld schrijven `neem(creëer())`, met de creator op de meest rechtse positie.

Vervangen we alle atomen in een REGOU door de verzamelingnamen, dan krijgen we de verzameling van alle operatiesequenties die aan de REGOU voldoen. Zo is **K=Kijk Mod* Crea** de verzameling van alle kijkreeksen. De eis dat de axioma's een complete specificatie geven kan dus zo geformuleerd worden: de functie $W : \mathbf{K} \rightarrow$ **Waarden** moet totaal zijn, dus op elk element van **K** gedefinieerd zijn, Hier is W de waardefunctie en **Waarden** de verzameling van alle mogelijke waarden.

Om het begrip toestand zinvol te maken moeten we eerst het begrip *levensloop* van een object definiëren. Een object wordt gecreëerd, en ondergaat daarna eventueel een aantal modificaties. Al wat we hebben bij algebraïsche specificatie is een beschrijving van deze gebeurtenissen. Een levensloop is dan een creator gevolgd door nul of meer modificatoren. Merk op dat als er parameters zijn deze zijn ingevuld bij een levensloop, en dat verschillende waarden van parameters verschillende levenslopen definiëren. Zo zijn bij de `Lijst`klasse de twee levenslopen `voegtoe(creëer(),e1)` en `voegtoe(creëer(),e2)` verschillende levenslopen. De verzameling **LL** van levenslopen is dus

$$\mathbf{LL} = \mathbf{Mod*\ Crea}.$$

Intuïtief zeggen we dat twee objecten dezelfde toestand hebben als ze hetzelfde (potentieel) gedrag in de toekomst vertonen. Hiertoe is niet voldoende dat ze voor alle kijkoperaties hetzelfde resultaat geven. Nemen we het voorbeeld van onze koffieautomaat. We nemen informeel aan dat de toestand beschreven wordt door het aantal koppen koffie dat er nog inzit (en we gaan zo meteen bewijzen dat dat ook zo is): een automaat met 1 nog één kop koffie is bijna leeg, een met honderd koppen niet. Het verschil merken we als we twee keer proberen een kop koffie te nemen. We moeten dus niet alleen de kijkoperaties nemen, maar de kijkoperaties voorafgegaan door willekeurige reeksen modificatoren. We zeggen dat twee levenslopen equivalent zijn als ze beide, voor zo'n willekeurige reeks, dezelfde waarde opleveren:

$$x, y \in \mathbf{LL} : x \simeq y \Leftrightarrow \forall\, u \in \mathbf{Kijk\ Mod*}\, W(ux) = W(uy).$$

We zijn er hier van uitgegaan dat geen enkele operatie ondergedetermineerd is. Immers, als er sprake is van ondergedetermineerdheid, dan kan *dezelfde* levensloop in verschillende gevallen verschillende antwoorden geven, zodat hij niet equivalent is met zichzelf. Een toestand is nu een verzameling die alle levenslopen bevat die equivalent zijn met elkaar. Twee equivalente levenslopen beschrijven dus verschillende manieren om tot dezelfde toestand te komen. In het voorbeeld van onze koffieautomaat nemen we nu levenslopen van de vorm

$$\ell_n = \texttt{vuleenbij}^{\texttt{n}}(\texttt{creëer}()).$$

Het is gemakkelijk om aan te dat ℓ_p en ℓ_n equivalent zijn als en slechts als `n` $=$ `p`. Is bijvoorbeeld `n` $<$ `p` dan levert de sequentie $u =$ **P1**`neem`**P2**`neem`^{n} op dat $W(u\ell_n) =$ `"leeg"` en $W(u\ell_p) =$ `koffie`. Aan de andere kant is het zo dat *elke levensloop* equivalent is met een of andere ℓ_n. Dit is ook zeer gemakkelijk te bewijzen. Immers, in het algemeen, als we een levensloop hebben die een reduceerbare modificator bevat, dan kunnen we door reductie bewijzen dat de levensloop equivalent is met een kortere levensloop. Door dit zo nodig meerdere malen te doen houden we geen reduceerbare modificatoren meer over. Bijgevolg is elke levensloop equivalent met een levensloop die alleen maar niet-reduceerbare modificatoren bevat. Maar voor onze automaat is er slechts één creator, en slechts een niet-reduceerbare modificator, en dus is elke levensloop zonder reduceerbare modificatoren van de vorm ℓ_n. We kunnen dus concluderen dat er met elke mogelijke toestand juist één levensloop ℓ_n overeenkomt. We zouden dus ook de toestand kunnen beschrijven door juist de waarde van n op te geven. Met andere woorden: door te zeggen dat de toestand beschreven wordt door een attribuut van type `unsigned int`.

Dergelijke attributen die afgeleid kunnen worden uit algebraïsche specificaties komen vrij veel voor. In extreme gevallen hebben we een klasse met een creator `creëer`, die een aantal argumenten heeft, en kan men bewijzen dat elke toestand equivalent is met juist één levensloop van de vorm `creëer(p1, p2, ...)`. Het is dan duidelijk dat men dan de toestand kan beschrijven aan de hand van de argumentwaarden van `creëer`. Dit is bijvoorbeeld het geval als een klasse geen mutatoren heeft, en we kunnen bewijzen dat geen twee verschillende sets van argumentwaarden dezelfde toestand geven.

Er is nog een andere manier om aan attributen te komen, en dat is om niet uit te gaan van de levenslopen, maar van de vragen die we aan een levensloop kunnen stellen. Bekijken we de verzameling $\mathbf{V} =$ **Kijk Mod***, die we al gebruikt hadden om equivalente levenslopen te definiëren. Dit is de verzameling vragen die we aan een levensloop kunnen stellen. Als v een vraag is en ℓ is een levensloop, dan is $W(v\ell)$ het *antwoord* op vraag v gesteld aan levensloop ℓ. Twee vragen zijn equivalent als het antwoord op beide vragen voor elke levensloop hetzelfde is. Formeel:

$$u, v \in \mathbf{V} : u \simeq v \Leftrightarrow \forall\, x \in \mathbf{LL} : W(ux) = W(vx).$$

Een vraag is triviaal indien elke levensloop hetzelfde antwoord geeft. Formeel:

$$v \in \mathbf{V} : \mathbf{triviaal}(v) \Leftrightarrow \forall\, x, y \in \mathbf{LL} : W(vx) = W(vy).$$

Een attribuut wordt nu gedefinieerd als een equivalentieklasse van niet-triviale vragen. Het type van het attribuut wordt gegeven door het returntype van de kijkoperatie van een van de equivalente vragen: uiteraard is dit returntype voor al deze vragen hetzelfde[3]. Deze definitie garandeert ons dat we alle relevante attributen vinden. Het is echter wel zo dat de vorm die de attributen aannemen vreemd kan zijn, en dat we ook alle mogelijke afgeleide attributen zullen vinden. Nemen we het voorbeeld van onze automaat. Het is duidelijk dat elke vraag die de operatie `maakleeg` bevat een triviale vraag is, vermits de automaat hierdoor herleid wordt tot de vorm `creëer()`. Merk op dat

[3] Dit is weer een vereenvoudiging: zo zou een van de vragen een `unsigned int` als type kunnen hebben, en een andere een `int`, waarbij de mogelijke waarden die het antwoord aanneemt in de doorsnede van de twee types ligt.

in het algemene geval we de niet-reduceerbare modificatoren niet kunnen verwijderen, omdat een vraag geen creator bevat; wel kunnen we ze naar rechts verplaatsen. Hier is het echter wel het geval. Elke vraag heeft als kijkoperatie **P1**`neem`, en als modificatorenreeks combinaties van **P2**`neem` en `vuleenbij`. Als er in de reeks ergens een `vuleenbij` voorkomt *rechts* van een **P2**`neem`, dan kunnen we deze volgens axioma (4) tegen elkaar schrappen. Staat er echter een `vuleenbij` helemaal links in de rij, dan is het antwoord op de vraag volgens axioma (6) altijd `koffie`, en dus is de vraag triviaal. Elke niet-triviale vraag is dus equivalent met een vraag van de vorm

$$v_n = \textbf{P1}\texttt{neem}(\textbf{P2}\texttt{neem})^{\texttt{n}}.$$

Het is gemakkelijk in te zien dat v_n niet equivalent is met v_p als $n \neq p$, omdat de antwoorden die door de levenslopen ℓ_n gegeven worden gelijk zijn aan

$$W(v_n\ell_p) = \begin{cases} \texttt{koffie} & n < p \\ \texttt{leeg} & n \geq p \end{cases}$$

We hebben dus oneindig veel attributen. Vermits hun type twee waarden kan aannemen kunnen we ze omcasten naar `bool`, met waarde `true` als het antwoord koffie is. Dit lijkt iets heel anders dan het ene `int`-attribuut dat we daarstraks berekend hadden, maar is toch hetzelfde. Immers, er zijn verborgen afhankelijkheden: als $W(v_n\ell_p) =$ `"leeg"`, dan is ook $W(v_{n+1}\ell_p) =$ `"leeg"`. In termen van de booleaanse waarden: de waarden van de attributen vormen een rij die begint met een aantal (eventueel nul) keer `true`, gevolgd door allemaal `false`n. Dit begint heel sterk te lijken op een rep-invariant, en inderdaad is dit de rep-invariant van een, waarschijnlijk zeer inefficiënte, implementatie waarbij een geheel getal wordt voorgesteld met oneindig veel `bool`s.

We kunnen dezelfde techniek ook toepassen op ons eerste voorbeeld van algebraïsche specificatie, dit van de klasse `Lijst<T>`. Deze klasse had één creator zonder parameters, `creëer()`, en één niet-reduceerbare modificator met een parameter, `voegtoe()`. Bijgevolg zijn alle levenslopen equivalent met een levensloop van de vorm
voegtoe(voegtoe...(creëer(),e1)...en).
Voeren we hiervoor de verkorte notatie `(e1, ..., en)` in. Het is gemakkelijk in te zien dat de vraag `eerste staart`$^{k-1}$ gesteld aan `(e1, ..., en)` als antwoord `ek` heeft als $k \leq n$, en `exceptie` als $k > n$. Bijgevolg zijn geen twee levenslopen aangegeven door een verschillende rij van `T`s equivalent, en is de toestandsruimte gelijk aan de ruimte van zulke rijen. Merk op dat we niet-triviale vragen kunnen bedenken die niet equivalent zijn met een vraag van het type `eerste staart`$^{k-1}$, zoals bijvoorbeeld `eerste staart`$^{k-1}$`voegtoe(.,a)`.

Eerder hebben we al de ruimtes **Rep** en **Abs** ingevoerd. Noemen we nu **T** de verzameling van toestanden van een klasse, zoals gedefinieerd met algebraïsche specificatie. Het is duidelijk dat **T** zo weinig mogelijk verband houdt met **Rep**: **Rep** heeft te maken met implementatie, en we proberen juist zoveel mogelijk onafhankelijk te werken van implementatie. Anders zit het met de ruimte **Abs**. We hebben het element `exceptie` ingevoerd om ongeldige sequenties van operaties op te vangen. Het is duidelijk dat alle levenslopen die leiden tot de exceptie equivalent zijn, en dus kunnen we stellen dat `exceptie` een toestand is, `exceptie`$\in$**T**. Nemen we nu een levensloop die niet tot `exceptie` voert. Het is duidelijk dat het de bedoeling is dat dit met een

of ander element van **Abs** overeenkomt. Nu was de overeenkomst tussen **Rep** en **Abs** niet eenduidig bepaald (denk aan de lengte van de `Persoon`klasse), maar nu is er wel eenduidigheid. Immers, een element van **Abs** wordt bepaald door zijn (potentieel) gedrag, en dit wordt gemodelleerd door te kijken wat het resultaat is van toepassen van **Kijk Mod*** op een levensloop. Verschillende elementen van **Abs** hebben verschillend gedrag, en kunnen dus niet met dezelfde levensloop overeenkomen. Omgekeerd: als twee levenslopen equivalent zijn hebben ze hetzelfde potentieel gedrag, en modelleren ze dus hetzelfde element van **Abs**. Bijgevolg, met elke toestand (behalve `exceptie`) komt juist één element van **Abs** overeen. Anderzijds, als er elementen van **Abs** zijn waarmee geen toestand overeenkomt, dan zitten we ook met een probleem: als we de specificaties implementeren, dan kunnen we geen informatie verwerken over die extra elementen van **Abs**. De conclusie is dat, bij een goed ontwerp met gedetermineerde specificaties er een 1-1-relatie bestaat tussen **T** en **Abs**. Dit houdt in dat, als we een attribuut kunnen geven dat natuurlijk lijkt voor **Abs**, dit attribuut moet af te leiden zijn uit de levenslopen van de klassenspecificatie. Nemen we bijvoorbeeld een klasse `Persoon` met een kijkoperatie `geefNaam()`, dan moeten we op een of andere manier uitdrukken dat met een object uit de `Persoon`klasse een reëel persoon overeenkomt, en dat `geefNaam()` de naam van die persoon moet teruggeven. Expliciet werken met attributen houdt echter een gevaar in: het is moeilijk formeel zeker te bepalen dat de attributen die we zo krijgen allemaal onafhankelijk zijn (zijn er geen afgeleide attributen bij?) en ook is het moeilijk om te zien of we wel *alle* relevante attributen bepaald hebben.

19.7 ALGEBRAÏSCHE SPECIFICATIE EN GEPARAMETRISEERDE KLASSEN

In de vorige paragraaf hebben we een geparametriseerde klasse `Lijst<class T>` ingevoerd. Als we een formele specificatie van zo een klasse willen geven, dan moeten we daarbij ook rekening houden met de eigenschappen van de parameter. Hebben we bijvoorbeeld een klasse van containers die elementen van een of andere `T` kan bevatten, en willen we dat we kunnen opzoeken in $O(\log n)$, waarin n het aantal elementen in de container is, dan moeten we eisen dat de elementen van `T` sorteerbaar zijn. Vaak wordt in specificatie bij de klasse aangegeven wat de klasse verwacht van haar parameter. Dit is echter geen goed idee. Nemen we de klasse `Lijst<class T>`, en breiden we deze uit met een kijkoperatie `geefKleinsteGrotere(T)`, die informeel gezegd het kleinste element in de lijst teruggeeft dat groter is dan zijn argument. Maar uiteraard is dit alleen zinvol als objecten van de klasse `T` een volgorde hebben. Herinneren we nu even aan het mechanisme dat een C++-compiler gebruikt bij geparametriseerde code. Als de compiler zulke code tegenkomt dan parset hij ze. Als er daarna een verwijzing met ingevulde parameters is, dan compileert de compiler de geparametriseerde code, maar alleen die code die *nodig* is. Gebruiken we dus een `Lijst` met ingevulde parameter, maar niet de lidfunctie `geefKleinsteGrotere()`, ook niet onrechtstreeks, dan zal de compiler de code voor deze lidfunctie niet compileren. Er is dan ook geen enkel probleem als we als parameter een klasse opgeven die niet geordend kan worden.

We moeten dus opgeven dat we, als we `geefKleinsteGrotere()` willen gebruiken, er moeten op letten dat de parameter `T` aan bepaalde eisen voldoet. De meeste van deze eisen worden uitgedrukt door middel van *diensten*. Een dienst is nog

het best te vergelijken met de specificatie van een Java-interface: het is een geheel van operaties voor een klasse, met hun specificaties. Bijvoorbeeld, voor de operatie `geefKleinsteGrotere()` moeten we een kleiner-dan dienst hebben:

`dienst kleiner-dan<class T>`
`operator<(T,T)→bool`

`(1)a<b⇒!b<a.`
`(2)!(a<b || b<a)⇔a==b.`
`(3) (a<b && b<c)⇒ a<c.`

Met deze dienst, die we nogal informeel gespecificeerd hebben omdat we niet gedefinieerd hebben hoe we de resultaten van twee of meer kijkoperaties formeel kunnen vergelijken, kunnen we dan ineens een voorbeeld geven van overerving:

`klasse KGLijst<class T>` **erft over van** `Lijst<class T>`
`geefKleinsteGrotere(T) → T:`**gebruikt** `T::kleiner-dan`
`(1)...`

Merk op dat we slechts een zeer kleine deelverzameling van een volledige algebraische specificatietaal gegeven hebben. In het laatste voorbeeld hebben we dan ook de axioma's voor `geefKleinsteGrotere()` niet gegeven. Dit is dan ook geen eenvoudig probleem: probeer je maar eens in te beelden wat je moet doen als je zo een functie moet testen. Je mag hierbij geen gebruik maken van de interne structuur van de gelinkte lijst, want als er een fout zit in de implementatie van je lijst is die interne structuur niet betrouwbaar. Dus mag je alleen de levensloop gebruiken. De testcode die je dan schrijft is inherent even ingewikkeld als de axioma's van `KGLijst`.

19.8 MODELSPECIFICATIE MET Z

In deze paragraaf bekijken we het alternatief van *modeltalen*. Deze geven een specificatie van een systeem op basis van de *toestand* ervan. Deze manier van werken sluit dicht aan bij de klassieke manier van specificeren. De toestand wordt beschreven aan de hand van een aantal attributen, en operaties worden beschreven door de wijziging in de waarden van de attributen (voor modificatoren), en door de functionele relatie tussen attributen en teruggeefwaarde (voor kijkoperaties).

De uiteenzetting die we hier geven is gebaseerd op de specificatietaal Z[4]. Een aspect van Z dat nogal wat problemen oplevert is het uitgebreid gebruik van nogal vreemde symbolen, zoals $\rhd$, $\succ\!\!\succ$, $\Leftrightarrow$, $\frac{0}{9}/$, enzovoorts. Een eerste probleem is dat veel tekstverwerkers niet vlot in staat zijn om deze symbolen weer te geven, zodat men Z-specificaties niet in documenten kan opnemen. Het tweede probleem is dat deze symbolen niet echt duidelijk zijn voor niet-specialisten, wat Z moeilijk te leren maakt. Zo betekent bijvoorbeeld $\Box(i = 1)$ dat de variabele i altijd de waarde 1 heeft, maar dit is niet echt een sprekende notatie. De reden hiervoor is te vinden in het pakket LaTeX, dat gebruikt werd (en wordt) door de ontwerpers van Z. In LaTeX is het niet moeilijk om

[4] Niet te verwarren met de programmeertaal Z. Gelukkig is Z (specificatie) ontworpen door Engelsen, en Z (programmatie) door Amerikanen. De uitspraak voor de eerste Z is dan ook [zed], die voor de tweede [zi:] (zoals 'zie' in het Nederlands).

een nieuwe symbool te definiëren door het definiëren van een macro, en de ontwerpers van Z hebben daar zeer veel gebruik van gemaakt. De reden hiervoor is dat degene die Z-specificaties schrijft in LaTeX op deze manier verplicht is om de macronamen te gebruiken. Nu bestaat er software die een LaTeX-document filtert op Z-specificaties, en deze dan verder verwerkt om ze bijvoorbeeld te valideren of op consistentie te controleren. Door het gebruik van macro's wordt dit parsen gemakkelijk. Dikwijls is de naam van de macronaam sprekender dan het symbool: $\Box(i = 1)$ wordt in LaTeX geschreven als `\always(i=1)`. We zullen dan ook in wat volgt regelmatig het Z-symbool vervangen door zijn macronaam indien dit duidelijker is.

De oorspronkelijke definitie van Z werkte niet objectgericht: schema's werkten met losse groepen gegevens. Later is hieruit Object-Z gegroeid, ongeveer zoals C++ geënt is op het objectloze C. Trouwens: net zoals een geldig C-programma in principe een geldig C++-programma is, is een Z-specificatie een Object-Z-specificatie waarin geen klassen voorkomen. Het zal handig zijn om eerst een introductie te geven tot Z, en daarna tot Object-Z.

Z werkt met *schema's*. Een schema bestaat uit drie delen: de naam van het schema, een *signatuur* en een *predicaat*. De signatuur bestaat in principe uit een lijst van entiteiten die deel uitmaken van het schema, aangeduid met hun naam en type. Het predicaat is een uitspraak waarvan wordt aangenomen dat hij steeds waar is. Deze uitspraak geeft de relaties tussen de entiteiten uit de signatuur. Hieronder vinden we een schema voor een fluit, dat aangeeft dat een fluit lawaai maakt als de druk hoog genoeg is:

fluit

$$
\begin{array}{|l}
geluid : (luid, stil) \\
druk : \mathbb{R} \\
mindruk : \mathbb{R} \\
\hline
geluid = luid \Leftrightarrow druk > mindruk \\
\hline
\end{array}
$$

Een tweede schema definieert een waterketel. Ook hier is een druk, die afhankelijk is van de temperatuur:

ketel

$$
\begin{array}{|l}
druk : \mathbb{R} \\
temperatuur : \mathbb{R} \\
maxtemp : \mathbb{R} \\
\hline
druk = temperatuur / 100 \\
maxtemp = 101 \\
\hline
\end{array}
$$

We hebben hier een constante, $maxtemp$, waarvan de waarde gegeven wordt. Een schema kan ook andere schema's vermelden in zijn signatuur. De entiteiten en het predicaat van de andere schema's worden dan overgenomen. Het predicaat kan dan uitspraken bevatten die relaties van entiteiten uit verschillende schema's beschrijven. Zo definieert het schema ***fluitketel*** een ketel die fluit bij een temperatuur van meer dan honderd graden:

fluitketel

$$\begin{array}{|l}
\hline
\mathit{fluit} \\
\mathit{ketel} \\
\hline
\mathit{mindruk} = 1 \\
\hline
\end{array}$$

Let hierbij op dat er geen aggregatie is van een fluit en een ketel, maar een versmelting van twee *schema's*. Het nieuwe schema bevat als signatuur de unie van de signaturen van de deelschema's: de entiteit $druk$, die zowel in $fluit$ als in $fluitketel$ voorkomt, staat dan ook maar één keer in het $fluitketel$schema, en zorgt voor de verbinding tussen de fluit en de ketel. Het $fluitketel$schema kan dan ook geschreven worden zonder deelschema's:

fluitketel

$$\begin{array}{|l}
\hline
\mathit{geluid} : (\mathit{luid}, \mathit{stil}) \\
\mathit{druk} : \mathbb{R} \\
\mathit{mindruk} : \mathbb{R} \\
\mathit{temperatuur} : \mathbb{R} \\
\mathit{maxtemp} : \mathbb{R} \\
\hline
\mathit{geluid} = \mathit{luid} \Leftrightarrow \mathit{druk} > \mathit{mindrukdruk} : \mathbb{R} \\
\mathit{druk} = \mathit{temperatuur}/100 \\
\mathit{maxtemp} = 101 \\
\mathit{mindruk} = 1 \\
\hline
\end{array}$$

Een belangrijk schema om verandering aan te geven is het zogenaamde *Deltaschema*. Een Deltaschema wordt afgeleid van een gewoon schema als volgt: voor elke entiteit in het oorspronkelijke schema bevat het Deltaschema twee entiteiten: een met dezelfde naam, en een met de naam met een accent op het einde. De eerste entiteit geeft de oorspronkelijke entiteit vóór een actie weer, de tweede erna. Ook het predicaat wordt verdubbeld. Dit geeft aan dat zowel vóór als na de actie aan het predicaat van het oorspronkelijke schema voldaan moet zijn. Hoewel het Deltaschema uit het oorspronkelijke schema volgt, geven we toch het schema Δketel weer:

$\Delta ketel$

$$\begin{array}{|l}
\hline
\mathit{druk} : \mathbb{R} \\
\mathit{temperatuur} : \mathbb{R} \\
\mathit{druk}' : \mathbb{R} \\
\mathit{temperatuur}' : \mathbb{R} \\
\hline
\mathit{druk} = \mathit{temperatuur}/100 \\
\mathit{druk}' = \mathit{temperatuur}'/100 \\
\hline
\end{array}$$

Om nu een operatie (in Z is dit dus een procedure die werkt met globale variabelen) te specificeren hebben we nog twee dingen nodig: parameters en teruggeefwaarden. In Z wordt een parameter aangegeven als een entiteit met een naam die eindigt op een vraagteken; een teruggeefwaarde heeft een naam eindigend op een uitroepteken. We kunnen nu een opwarmschema geven voor onze fluitketel

$$
\begin{array}{|l}
\hline
warmop \\
\Delta \textit{fluitketel} \\
verschil? : \mathbb{R} \\
\hline
temperatuur + verschil? < maxtemp \\
temperatuur' = temperatuur + verschil? \\
mindruk = mindruk' \\
\hline
\end{array}
$$

Merk op dat de wijziging van de druk, en de mogelijke wijziging van het geluid, niet expliciet is aangegeven in het schema. Immers, deze volgen uit de predicaten van de ingesloten schema's. Hieruit kan bijvoorbeeld worden afgeleid dat $druk' = druk + verschil/100$. Opvallend zijn de twee extra elementen uit het predicaat van $warmop$. Ten eerste hebben we de uitspraak $temperatuur + verschil < maxtemp$. Vermits deze een relatie aangeeft van entiteiten zonder accent, gaat het om een deel van de preconditie, dat toegevoegd wordt aan het predicaat van ***fluitketel***. Ten tweede is er $mindruk = mindruk'$. Dit geeft geen wijziging aan, maar is toch noodzakelijk. Immers, specificaties kunnen en mogen ondergedetermineerd zijn. Elke entiteit die door een accent in de signatuur vrijgegeven wordt mag veranderen, en voor zover de exacte waarde na verandering niet kan afgeleid worden uit het predicaat, is de waarde onbepaald. De uitdrukking $mindruk = mindruk'$ is dus nodig om ervoor te zorgen dat $mindruk$ niet verandert. Merk overigens op dat $=$ hier geen toekenningsoperator is, vandaar dat het accent rechts mag staan. In het schema $warmop$ wordt niet beschreven wat er gebeurt als aan de voorwaarde niet voldaan is. Nemen we nu aan dat er dan een beveiliging in werking treedt die de zorgt dat er niets gebeurt (realistischer zou zijn om de temperatuur te beperken tot 101 graden, maar dit is minder interessant als model). Voor dit soort gevallen heeft Z het zogenaamde Ksischema[5] Het werkt analoog als een Deltaschema, maar aan het predicaat wordt toegevoegd dat alle entiteiten constant blijven. Zo is bijvoorbeeld het schema $\Xi ketel$ gegeven door

$$
\begin{array}{|l}
\hline
\Xi ketel \\
druk : \mathbb{R} \\
temperatuur : \mathbb{R} \\
druk' : \mathbb{R} \\
temperatuur' : \mathbb{R} \\
\hline
druk' = druk \\
temperatuur' = temperatuur \\
\hline
\end{array}
$$

Het is overbodig om het originele predicaat van $ketel$ nog te vermelden: als dit relevant is zal er altijd wel een schema zijn dat dit toch al bevat. Met een Ksischema is ***beveiliging*** nu snel geschreven:

$$
\begin{array}{|l}
\hline
beveiliging \\
\Xi \textit{fluitketel} \\
verschil? \\
\hline
temperatuur + verschil \geq 101 \\
\hline
\end{array}
$$

[5] Ξ, uitgesproken ksi, is een Griekse hoofdletter.

Een schema kan nu $beveiliging$ en $warmop$ samenvoegen:

$beveiligdeOpwarming$

$$beveiliging \lor warmop$$

het $\lor$-symbool (LaTeX-macro `\zor`) geeft aan dat de signatuur van ***beveiligdeOpwarming*** alle entiteiten uit de twee andere signaturen in zich heeft, en dat het predicaat van $beveiligdeOpwarming$ waar is als een van de twee andere predicaten waar is. In dit voorbeeld kunnen we zien dat het op deze manier samennemen met de nodige voorzichtigheid moet gebeuren. Veronderstellen we even dat we in $beveiliging$ het Ksischema hadden weggelaten. Dan zouden, als $temperatuur + verschil > 101$, alle waarden in de fluitketel willekeurig mogen veranderen. Immers, de entiteiten met accent worden vrijgegeven door het Deltaschema in $warmop$, en aan het predicaat is voldaan zonder dat dit beperkingen oplegt aan deze entiteiten. Zetten we het Ksischema terug en verwijderen we de voorwaarde $temperatuur + verschil \geq 101$ uit $beveiliging$, dan zijn de specificaties van $beveiligdeOpwarming$ ondergedetermineerd. Immers, als $temperatuur + verschil < 101$dan kan $beveiligdeOpwarming$ kiezen voor het predicaat van $warmop$, en de temperatuur veranderen, of voor het predicaat van $beveiliging$, en niets doen.

Tenslotte geven we nog een eenvoudig voorbeeld van een schema met een teruggeefwaarde, en een schema om onze fluitketel te initialiseren:

$geefDruk$

$$fluitketel$$
$$uit!$$
$$\vdash$$
$$uit! = druk$$

$init$

$$\Delta fluitketel$$
$$temperatuur?$$
$$\vdash$$
$$temperatuur' = temperatuur?$$

Een van de hulpmiddelen die zeer vaak gebruikt worden in Z is dat van functiedefinities[6]. Functies zoals wij die kennen (zoals $\cos(x)$) zijn functies in de zin van Z, maar de bedoeling van Z-functies is voornamelijk om abstracte datastructuren die `(sleutel,data)`-paren kunnen opslaan te kunnen specificeren, zonder de onderliggende concrete datastructuur (die een implementatiekeuze is) te moeten aangeven. Nemen we het voorbeeld van het volgende schema:

$telefoonboek$

$$nr : \text{seq}\, char \rightarrow \text{seq}\, cijfer$$

[6] In Z zijn er dan ook een aantal pijlen voor dit doel voorbehouden, namelijk $\mapsto$, $\leftrightarrow$, $\rightarrow$, $\rightarrowtail$, $\twoheadrightarrow$, $\nrightarrow$, $\rightarrowtail\!\!\!\!\!\!\rightarrow$, $\twoheadrightarrow\!\!\!\!\!\!\rightarrow$, $\nrightarrow\!\!\!\!\!\!\rightarrow$, $\rightarrowtail\!\!\!\!\!\!\rightarrow$, en $\twoheadrightarrow\!\!\!\!\!\!\rightarrow$. Deze hebben allemaal een verschillende betekenis naargelang het soort functie (1-1, surjectie, ...). We gebruiken hier de pijl $\rightarrow$, die in Z de algemene functiepijl is.

dit definieert een telefoonboek als een functie die letterreeksen (d.w.z. strings) afbeeldt op cijferreeksen. Dat deze afbeelding namen afbeeldt op telefoonnummers wordt niet duidelijk uit dit schema, maar uit de volgende kijkoperatie:

$$
\begin{array}{|l}
\hline
geefnummer \\
telefoonboek \\
naam? \\
telefoonnummer! \\
\hline
(naam? \in \mathrm{dom}\, nr \land telefoonnnummer! = nr(naam?)) \lor \\
(naam? \notin \mathrm{dom}\, nr \land telefoonnr! =) \\
\hline
\end{array}
$$

Merk op dat we het tweede alternatief, waardoor $geefnummer$ een lege reeks teruggeeft als de naam niet in het telefoonboek voorkomt, niet geven met een `if-then-else`-structuur. Het predicaat moet waar zijn, en dus moet een van de twee alternatieven van de offunctie waar zijn. (aangeduid met $\lor$) Met deze twee schema's wordt niet aangegeven in welke vorm de functie nr wordt geïmplementeerd (alhoewel: een hashtabel zal het wel niet zijn, want er is nergens een $hash$-schema gegeven, en dus kunnen we geen hashwaarden uit namen afleiden). Het grote verschil tussen Z-functies en wiskundige functies is dat we Z-functies kunnen veranderen: zo kunnen we aan een telefoonboek een naam-nummerpaar toevoegen, of er een uit verwijderen. Ook hiervoor heeft Z een reeks symbolen, zoals $\lhd$, $\lhd\!\!\!-$, $\rhd$ en $\rhd\!\!\!-$. We geven als voorbeeld het schema om aan het telefoonboek een toevoeging te doen.

$$
\begin{array}{|l}
\hline
voegnrtoe \\
\Delta telefoonboek \\
naam? \\
nummer? \\
\hline
nr' = nr \oplus (naam? \mapsto nummer?) \\
\hline
\end{array}
$$

De makers van Z keken naar een functie zoals een wiskundige dat doet: als naar een verzameling van paren. In plaats van te schrijven dat $nr(\text{"}Will\text{"}) = 797204$, kan men ook stellen dat $(\text{"}Will\text{"}, 797204) \in nr$. Vermits een functie een verzameling is, is het logisch dat men aan deze verzameling elementen kan toevoegen, of er elementen kan uit verwijderen.

19.9 MODELSPECIFICATIE MET OBJECT-Z

Z zoals hierboven beschreven gaat al een eind in de richting van objectgericht ontwerp. Immers, wat we beschreven hebben met de schema's is een fluitketel, en dat is een object. We hadden een creator ($init$), modificatoren ($warmop$, ...) en een kijkoperatie ($geefDruk$) Er is een probleem echter, en dit is een probleem dat algemeen aanwezig is in niet-objectgerichte programmeertalen. We kunnen wel één fluitketel beschrijven, maar als we een systeem hebben met meer fluitketels moeten we onze beschrijving herhalen voor elke nieuwe fluitketel. Er is immers maar één entiteit $druk$ in ons systeem, een entiteit $temperatuur$, enzovoorts. Wat we willen is een systeem waarin we

kunnen zeggen dat er twee entiteiten $f1$ en $f2$ zijn die elk fluitketels zijn, met elk hun eigen druk, $f1.druk$ en $f2.druk$, en zo verder. Wat we doen is dat we alle elementen van Z, die globaal waren en golden voor een heel systeem, gaan inperken zodat ze gelden binnen één object van de klasse. Een klassendefinitie in Object-Z bestaat dus uit een aantal Z-schema's. Zo kunnen we bijvoorbeeld een klasse $Ketel$ definiëren

Ketel

$maxtemp : \mathbb{R}$

Een ketel is een instrument om vloeistof op te warmen.

$druk : \mathbb{R}$
$temperatuur : \mathbb{R}$

$druk = temperatuur/(maxtemp - 1)$
$temperatuur < maxtemp$

INIT

$temperatuur?$
$maxtemp?$

$temperatuur = temperatuur?$
$maxtemp = maxtemp?$

warmop

$\Delta(temperatuur, druk)$
$verschil? : \mathbb{R}$

$temperatuur' = temperatuur + verschil$

Om te beginnen hebben we een schema dat een lijst geeft van constanten. Merk op dat dit geen klassenconstanten zijn: elke ketel heeft zijn eigen constante. Dit is zeer handig als we een ketel met alcohol ($maxtemp = 79,5$) of zwavel ($maxtemp = 446$) hebben. De constanten kunnen natuurlijk niet gewijzigd worden gedurende de levensduur van een object. Als we een klassenconstante willen definiëren dan moeten we een predicaat opgeven bij de constantenlijst: als we alleen waterketels hebben dan schrijven we

$maxtemp : \mathbb{R}$

$maxtemp = 101$

Daaronder hebben we een informele beschrijving van de klasse, en daaronder een schema met attributen en onderdelen: het zogenaamde *toestandsschema*. Object-Z ondersteunt aggregatie, zodat objecten van een andere klasse deel kunnen uitmaken van een object. Noteer overigens dat in de terminologie van Object-Z deze deelobjecten ook attributen zijn: we houden ons echter aan de UML-naamgeving die attributen beperkt tot elementaire types.

Vermits de toestand van een object beschreven wordt met attributen, kan het zijn dat de waarden van attributen aan voorwaarden moet voldoen. Vergelijk dit met de

rep-invariant: alleen hebben we hier geen implementatieattributen maar specificatieattributen. Omdat deze niet uniek bepaald zijn door de ruimte **Abs**, is er ook weer sprake van een keuze, en heeft het zin om begrippen zoals rep-invariant en abstractiefunctie ook hier te gebruiken. De toestand van een object wordt beschreven aan de hand van de waarden van de attributen en de toestanden van eventuele deelobjecten. Merk op dat we ook bij een Z-specificatie geen garantie hebben dat twee verschillende toestanden door de abstractiefunctie worden afgebeeld op verschillende objecten in **Abs**.

Na het toestandsschema volgen de operatieschema's. Het gaat om specificatie, en dus zijn alle operaties publiek. De rep-invariant maakt dus deel uit van de preconditie (de versie zonder accenten) en de postconditie (de versie met accenten) van alle operaties. Omdat we nu binnen een klasse werken moeten we de rep-invariant niet expliciet aangeven.

Het eerste schema is hier verplicht het INITschema. Er is slechts één INITschema voorzien. Dit komt omdat we de $\vee$-operator kunnen gebruiken om verschillende schema's samen te voegen: als er verschillende constructoren zijn kunnen we die op deze manier tot één INITschema maken. Merk op dat hier geen accentvariabelen zijn: er is alleen een toestand op het einde van INIT, niet in het begin. Voor de andere operaties is er een nieuw element: de Deltalijst, die begint met Δ, en aangeeft welke attributen en onderdelen kunnen veranderen. Wat niet in de lijst staat moet onveranderd blijven. Dit komt overeen met de `Wijzigt`clausule in een meer informele specificatie: wat niet in die clausule staat moet ook onveranderd blijven. Merk op dat pre- en postcondities niet zeer duidelijk te onderscheiden zijn in het predicaat van een operatie. Zo is er in het predicaat van $warmop$ geen expliciete vermelding van de preconditie $temperatuur + verschil < maxtemp$. Nochtans, zonder deze conditie is het onmogelijk om aan het predicaat te voldoen.

Bij algebraïsche specificatie hadden we de levensloop. Bij de theorie van Object-Z kunnen we iets analoogs doen: in dit geval spreken we van de *geschiedenis* van een object. De levensloop bestond uit de opeenvolging van operaties die op het object hadden ingewerkt, te beginnen met de creatie. Bij de geschiedenis in de zin van Z hebben we een sequentie, in dewelke toestanden en operaties elkaar afwisselen. De eerste toestand is de initiële toestand. Dit is de toestand van het object nadat het gemaakt is door de operatie INIT. Vermits INIT parameters kan hebben, is niet noodzakelijk de begintoestand van alle objecten hetzelfde. Daarna volgen alle toestanden die het object doorlopen heeft, en tussen elke paar opeenvolgende toestanden komt de operatie (met ingevulde parameters) die de overgang van de ene naar de andere toestand heeft tot stand gebracht.

Merk op dat er een verschil is tussen de toestand zoals die hier beschreven wordt en deze zoals gedefinieerd bij algebraïsche specificatie. Laat ons de twee begrippen Z-toestand en A-toestand (A voor algebraïsch) noemen. Niet alleen is de manier van definiëren anders: bij de Z-toestand wordt gebruik gemaakt van attributen, terwijl bij de A-toestand wordt uitgegaan van antwoorden op niet-triviale vragen. Er is ook geen 1-1 relatie tussen Z-toestanden en A-toestanden. Nemen we twee geschiedenissen die leiden tot dezelfde Z-toestand, met dus hetzelfde waardepatroon voor de attributen. Het is duidelijk dat de levenslopen (dit zijn de geschiedenissen waaruit de toestanden geschrapt zijn) behoren tot dezelfde A-toestand: het antwoord op elke vraag zal voor de twee gevallen hetzelfde zijn. Maar het omgekeerde is niet waar. Het is best mogelijk dat er twee Z-toestanden zijn die wel onderscheiden kunnen worden omdat

een of ander attribuut verschillende waarden heeft, maar dat elke vraag toch voor beide Z-toestanden hetzelfde antwoord oplevert. Een voorbeeld kunnen we vinden bij `Punt2D`. Als we voor de Z-specificatie uitgaan poolcoördinaten (r, θ), dan zullen alle toestand met $r = 0$, wat ook de waarde van θ is, dezelfde antwoorden geven op dezelfde vraag. Twee Z-toestanden komen zeker overeen met dezelfde A-toestand als ze door de abstractiefunctie op hetzelfde element van **Abs** worden afgebeeld.

We hebben reeds terloops opgemerkt dat objecten deelobjecten kunnen hebben. Dergelijke objecten noemen we *samengestelde* objecten. Deze samenstelling kan op twee manieren gebeuren:

- Als elk object van klasse A een vast aantal objecten van klasse B bevat, dan kunnen we elk B-deelobject een naam geven. We spreken van *benoemde* aggregatie.
- Als het aantal B-objecten variabel is, dan moeten we met functies of met sequenties werken.

Deze indeling komt goed overeen met de manier waarop we in C++ aggregatie kunnen implementeren. Nemen we een klasse `Keuken`, waarbij elke keuken juist één koelkast bevat. Dit kunnen we implementeren met

```
class Keuken{
    ...
    Koelkast koelkast;
}
```

Als echter een keuken verschillende koelkasten kan bevatten, dan moeten we een of andere C++-container gebruiken om al de koelkasten in op te slaan. Eén van de mogelijkheden is om gebruik te maken van een tabel. In Object-Z is het equivalent van een tabel de sequentie, en alhoewel Object-Z een specifieke notatie heeft voor sequenties, kunnen we een sequentie toch beschouwen als een Z-functie. Immers, een tabel van koelkasten met grootte n kunnen we opvatten als een functie $tab : \mathbb{N} \rightarrow Koelkast$ met als speciale eigenschap dat $tab(i)$ gedefinieerd is als en slechts als $i < n$. Een Z-functie was een verzameling paren van objecten. Men gaat ervan uit dat de functie is samengesteld, in de zin van aggregatie uit die paren, en dat de paren zijn samengesteld uit de twee objecten[7]. We zien dus dat we met een functie alleen aggregatie kunnen modelleren. Later zullen we zien hoe gewone associaties gemodelleerd worden.

Aggregatie geeft meteen het grote verschil tussen algebraïsche specificatie en niet-objectgerichte modelspecificatie enerzijds en objectgerichte modelspecificatie anderzijds. De eerste twee laten niet toe om op verschillende niveaus van detaillering te werken. Bij algebraïsche modellering is dit een principepszaak: we moeten specificeren zonder naar het inwendige te kijken. Bij Z is dit gewoon een praktisch noodzaak: alhoewel we schema's kunnen vernestelen, is er geen duidelijk hiërarchisch principe in Z. In Object-Z is deze hiërarchie er wel: op elk niveau kunnen we een klasse definiëren die alle klassen van het diepere niveau omvat.

Nemen we het voorbeeld van het Pacmanproject op bladzijde 224. We kunnen elk van de vijf klassen die vermeld staan op het klassendiagram vatten in een Object-Z-specificatie. Maar dit geeft niet alle informatie over de klassen. Wat vermeld staat in deze specificatie is de beschrijving van de toestandsattributen en de operaties. De

[7] Voor zover het niet gaat over elementaire datatypes zoals `int` of `string`.

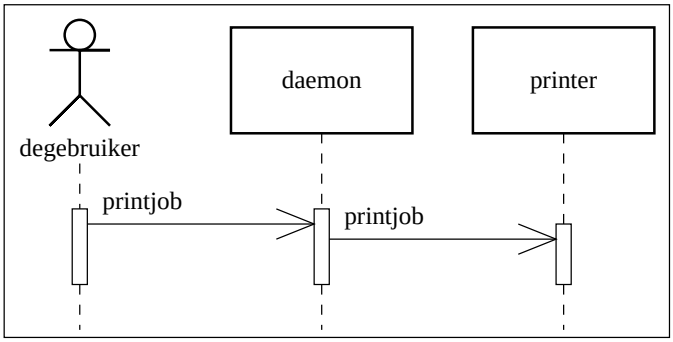
toestandsattributen vallen samen met de UML-attributen, vermits in geen van de vijf klassen samengestelde objecten zitten. Wat we niet in de Object-Z-specificaties van de klassen kunnen vatten, is de *samenwerking* tussen de verschillende objecten. Om deze te kunnen vatten moeten we een object invoeren op een hoger niveau, dat van het hele systeem. Dit object is een object van de klasse $Pacmansysteem$. In het schema beschrijven we wat er in het systeem met de objecten onderling gebeurt. In het toestandsschema kunnen we aangeven welke objecten er allemaal voorkomen. Merk op dat dit meer informatie is dan er in het UML-klassendiagram vermeld staat: in het $Pacmansysteem$schema wordt bijvoorbeeld zeer expliciet gesteld dat er één pacman is en vier spoken. Merk op dat, vermits er exact vier spoken zijn volgens de specificaties, we dit aantal zouden kunnen aanduiden door gewoon vier namen te geven. Het alternatief is te werken met een sequentie of een functie. Een sequentie is het meest logische, we geven ze hier in functievorm om niet de symbolen voor sequenties te moeten invoeren.

Ook alle verbindingen worden er vermeld, en dit in de vorm van functies. Voor elke associatie en voor elke richting is er één functie. In het signatuurgedeelte worden de associaties opgesomd, in het predicaatgedeelte worden de eigenschappen gedeclareerd. Men kan hier onder andere de multipliciteit van de verbindingen aangeven. In ons voorbeeld hebben we gespecificeerd dat elk spook de pacman kent. Merk op dat we enkel de gewone associaties op deze manier beschrijven. Aggregaties vormen een speciaal probleem, waar we later nog op terug komen. Door met functies te werken hebben we een extra probleem. Wat gebeurt er als we bijvoorbeeld spook 3 willen wijzigen? We hebben een Deltalijst, maar als we de functie spook in deze lijst opnemen, dan mogen we alles wijzigen. Omdat aangeven dat we maar één spook willen wijzigen een beetje omslachtig is, zetten we dit eens en voor altijd in een apart schema, een zogenaamd raamschema (Eng: *framing scheme*), dat aangeduid wordt met Φ, de Griekse hoofdletter fi. In het voorbeeld hebben we zo het schema $\Phi wijzigspook$. $n \lhd spook$ betekent de functie $spook$ maar met het element n uit het domein verwijderd. Raamschema's duiden zelf geen operatie aan, maar kunnen wel opgenomen worden in het schema voor een operatie

$$
\begin{array}{|l}
\hline
Pacmansysteem \\
\hline
aantspook : \mathbb{N} \\
\hline
aantspook = 4 \\
\hline
\\
\hline
pacman : Pacman \\
spook : \mathbb{N} \rightarrow Spook \\
SpookKentPacman : Spook \rightarrow Pacman \\
\ldots \\
\hline
n \in \mathrm{dom}\, spook \Leftrightarrow n < aantspook \\
n \in \mathrm{dom}\, spook \Leftrightarrow \{spook(i) \mapsto pacman\} \in SpookKentPacman \\
\hline
\\
\hline
\Phi wijzigspook \\
\hline
\Delta(spook) \\
n \\
\hline
n \lhd spook' = n \lhd spook. \\
\hline
\hline
\end{array}
$$

Hiermee hebben we de statische relaties tussen objecten van een systeem beschreven. Hoe zit het nu met het dynamische model? In dit model wordt beschreven hoe objecten onderling berichten uitwisselen. Hiervoor heeft men in Object-Z het $\|$-symbool (LaTeX-macro:`\parallel`). Om de werking van dit symbool te beschrijven moeten we even terug naar sequentiediagrammen. Zo een diagram beschrijft de uitwerking van een taak uit het takendiagram. We hebben dus een bericht naar het systeem (als we het systeem als object beschouwen zet dit een operatie van het systeem in werking), en naar aanleiding van dit bericht sturen deelobjecten van het systeem berichten naar elkaar. Maar binnen het systeem wordt het bericht van buitenaf opgevangen door een object: de operatie van het systeem wordt dus uiteindelijk uitgevoerd door een operatie van dit object. Dit kan operaties oproepen van andere objecten, en zo verder. Nemen we het simpelst mogelijke voorbeeld van een printsysteem bestaande uit twee objecten: een printerdaemon en een printer. Als een printopdracht het systeem binnenkomt wordt dit opgevangen door de daemon en doorgestuurd naar de printer. We laten hier alle mogelijke complicaties, zoals een printer die nog bezig is of off-line staat, weg en krijgen Figuur 19.5 We moeten twee dingen specificeren. Ten eerste is er de operatie van het systeem (in ons voorbeeld: $print$). Er moet worden aangegeven met welke operatie van welk object ze overeenkomt. Ten tweede is er de communicatie tussen objecten (in ons voorbeeld: het sturen van de printjob van daemon naar printer).

Voor het eerste voorziet Object-Z het $\widehat{=}$-symbool (LaTeX-macro: `\sdef`, wat staat voor *service definition*). Dit identificeert een operatie van een object met de operatie van een deelobject. Het $\|$-symbool (LaTeX-macro `\parallel`; het is geen Object-Z specifike macro) geeft dan weer waar resultaten van een operatie naartoe gestuurd worden. In het voorbeeld hieronder wordt de operatie $print$ gedefinieerd als zijnde hetzelfde als $daemon.print$. Het resultaat hiervan wordt doorgestuurd naar $printer.print$. Opmerkelijk is dat niet expliciet wordt vermeld *wat* er juist wordt doorgestuurd naar de volgende operatie. In Object-Z gelden hiervoor de volgende regels:



Figuur 19.5. Een simpel sequentiediagram.

- Voor het $\widehat{=}$-symbool geldt dat het linkerlid de invoerparameters overneemt van het rechterlid. Bijgevolg heeft de $print$operatie van $Printsysteem$ een invoerparameter, namelijk $infile$?.
- Voor het $\|$-symbool geldt dat, als een uitvoerparameter aan de linkerkant dezelfde naam heeft als een invoerparameter aan de rechterkant (op ! en ? na), dat deze dan geïdentificeerd worden. In ons voorbeeld hebben we links de uitvoerparameters $uitfile$! en $datum$! en rechts de invoerparameter $uitfile$?. Bijgevolg wordt $uitfile? = uitfile!$, en wordt $datum!$ niet gebruikt.

Printsysteem

$daemon : Daemon$
$printer : Printer$

INIT

$\ldots$

$print \mathrel{\widehat{=}} daemon.print \parallel printer.print$

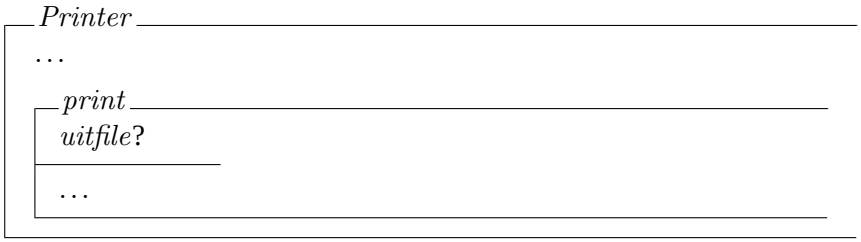
Daemon

$\ldots$

print

$infile?$
$uitfile!$
$datum!$

$\ldots$



Merk op dat we bij grotere systemen meer dan een niveau boven het klassenniveau kunnen uitgaan: zoals we gezien hebben zijn er in UML deelsystemen en componenten voorzien, waarbij componenten instantiaties zijn van deelsystemen. Bijgevolg kunnen we een deelsysteem beschrijven met een klassenschema van Object-Z. Merk daarbij op dat aggregatie een relatie is op objectniveau: een object is deelobject van een ander object. Dit heeft niets te maken met overerving, waar een klasse een deelklasse kan zijn van een andere. In feite is het omgekeerd: als B een deelklasse is van A, dan wil dit vreemd genoeg zeggen dat de *beschrijving* A (en we hebben een klasse gedefinieerd als een beschrijving, niet als een verzameling objecten) een deel is van de *beschrijving B*, en inderdaad, als we deze relatie willen specificeren in Object-Z, dan moeten we het schema A opnemen als deelschema van B. Deze verwarring tussen deelklassen en deelobjecten vinden we ook terug in UML. Objecten kunnen maar van één object deel uitmaken: dat is logisch, want het gaat over aggregatie. Maar deelsystemen kunnen ook maar deel uitmaken van één deelsysteem, en dit is onlogisch, want deelsystemen zijn beschrijvingen en geen instantiaties. Op deze manier werkt UML hergebruik tegen. Hebben we immers een klasse die we in verschillende deelsystemen willen gebruiken, dan mag dit gewoonweg niet.

Analoog hebben we een probleem bij aggregatie. In een UML-klassendiagram kunnen we aggregatieassociaties tekenen. (merk op dat, in tegenstelling tot wat bij deelsystemen de regel is, objecten van een klasse A in UML wél kunnen deel uitmaken van objecten van verschillende andere klassen). Waar een deelsysteem A dat deel uitmaakt van een ander deelsysteem B in UML getekend wordt *binnen* A, en dus op een niveau lager, wordt een deelklasse A waarvan de objecten deel uitmaken van objecten van een klasse B getekend in hetzelfde klassendiagram als B, en dus op hetzelfde niveau. Object-Z is in dit opzicht zuiverder: schema A staat op hetzelfde niveau als schema B, maar binnen schema B wordt aangegeven dat zijn instantiaties objecten gedefinieerd door A omvatten.

Merken we tenslotte nog op dat de axiomastructuur de mogelijkheid opent tot het (gedeeltelijk) automatisch bewijzen van de correctheid van implementatie van aggregatie. Nemen we een eenvoudig voorbeeld: we hebben een bedrijf met verschillende departementen, met een intern telefoonsysteem. Elk departement heeft een prefixcode: wie van een departement naar iemand van een ander departement wil bellen moet een nul draaien, dan het prefix van het departement, en dan het nummer van de persoon (dat nooit met nul begint). Binnen elk departement bestaat er reeds een telefoonboek, geïmplementeerd in de klasse `DepTelefoonboek`. Voor de eenvoud gaan we ervan uit dat elke onbekende naam geassocieerd wordt met een leeg telefoonnummer. Merk op dat daardoor de operatie $geefnummer$ eenvoudiger wordt dan deze die we bij het niet-objectgerichte telefoonboek gezien hadden.

DepTelefoonboek

$prefix$: seq $cijfer$

nr : seq $char \rightarrow$ seq $cijfer$

INIT

$prefix?$

$\forall\, naam \in \text{seq}\, char : nr(naam) = ""$
$prefix = prefix?$

geefNummer

$telefoonboek$
$naam?$
$telefoonnummer!$

$telefonnnummer = nr(naam?)$

...

Er is een operatie `geefNummer(string)` die het nummer teruggeeft als men een naam opgeeft (met een lege reeks als de naam niet gevonden is). Het teruggegeven nummer bevat het departementsprefix niet. We willen nu een klasse `BedrijfsTelefoonboek`, die analoog werkt. Alleen hebben we nu `geefNummer(string, cijferreeks)`. Men moet niet alleen de naam opgeven van de bestemmeling, maar ook het prefix van het eigen departement. Deze operatie geeft het nummer terug dat men moet ingeven, dus *zonder* prefix als de bestemmeling tot het eigen departement behoort, *met* nul en prefix voor een ander departement. We doen dit in twee stappen. Eerst definieren we een $prefixDepTelefoonboek$ dat een $DepTelefoonboek$ bevat. Merk op dat we voor de concatenatie van cijferreeksen de $+$-notatie gebruiken, die meer vertrouwd is dan de Z-notatie.

prefixDepTelefoonboek

$boek : DepTelefoonboek$

INIT

$prefix?$

$boek.init(prefix?)$

gevondenAnderePrefix

$naam?$
$prefix?$
$telefoonnummer!$

$boek.geefNummer(naam) \neq \text{""} \wedge prefix? \neq boek.prefix$
$telefoonnummer = \text{"0"} + boek.prefix$
$\quad + boek.geefNummer(naam)$

gewoonnummer

$naam?$
$prefix?$
$telefoonnummer!$

$boek.geefNummer(naam) == \text{""} \vee prefix? == boek.prefix$
$telefoonnummer = boek.geefNummer(naam)$

geefPrefixEnNummer

$gevondenAnderePrefix \vee gewoonnummer$

en uiteindelijk de klasse Telefoonboek:

DepTelefoonboek

$boeken$: seq $prefixDepTelefoonboek$

INIT

...

nietgevonden

$naam?$
$prefix?$
$telefoonnummer!$

$\forall i \in \text{dom}\, boek : boek(i).geefPrefixEnNummer(naam?, prefix?) = ""$
$telefoonnummer! = ""$

welgevonden

$naam?$
$prefix?$
$telefoonnummer!$

$\exists i \in \text{dom}\, boek : boek(i).geefPrefixEnNummer(naam?, prefix?) \neq "" \wedge$
$telefoonnummer! = boek(i).geefPrefixEnNummer(naam?, prefix?)$

geefPrefixEnNummer

$welgevonden \vee nietgevonden$

De operatie $geefPrefixEnNummer$ van $Telefoonboek$ is gedefinieerd met de ofoperator $\vee$. De predicaten van twee schema's die hierdoor verbonden worden sluiten elkaar onderling uit, zodat weer een gedetermineerd schema bekomen wordt.

In het bovengaande voorbeeld hebben we $PrefixDepTelefoonboek$ geconstrueerd op basis van $DepTelefoonboek$. We hebben hiervoor aggregatie gebruikt. Het was uiteraard ook mogelijk geweest om met overerving te werken. Dit was echter een fout geweest. Polymorfisme eist dat we een element van een deelklasse kunnen gebruiken als een element van de bovenklasse. Het is echter niet de bedoeling dat $prefixDepTelefoonboek$ een operatie $geefNummer$ ondersteunt.

Uiteraard wordt overerving in Object-Z wel ondersteund. Hierbij dienen twee zeer specifieke eigenschappen vermeld. Ten eerste is het mogelijk om attributen, operaties en parameters van operaties te *hernoemen*. Als we bijvoorbeeld een klasse van cirkels afleiden uit een klasse ellips, dan kunnen we besluiten om het attribuut $groteAs$ te hernoemen naar $diameter$, omdat dit een meer logische naam oplevert voor dit attribuut, en dus de leesbaarheid bevordert. Ten tweede is er een eigenaardigheid bij het overschrijven van operaties. De algemene regel bij Object-Z is dat we, als we twee schema's samennemen, een schema bekomen waarin alle elementen van zowel het ene als het andere schema zijn opgenomen. Bij overerving wordt het schema van de bovenklasse opgenomen in het schema van de deelklasse. Bijgevolg moeten we in de deelklasse alleen bijkomende elementen vermelden. Dit is gedeeltelijk ook zo bij de meeste programmeertalen: bijkomende attributen moeten vermeld worden, de oude

attributen worden zonder vermelding overgenomen uit de bovenklasse. Object-Z gaat echter verder dan de attributenlijst, en doet dit ook op het niveau van de operaties. Als we dus in de deelklasse een schema geven voor een operatie, dan is dit niet een volledig nieuw schema, maar een *aanvulling* op het oude schema. Een voorbeeldje: nemen we aan dat we van $prefixDepTelefoonboek$ willen overerven, en een nieuw telefoonboek maken, $LoginDepTelefoonboek$. Dit heeft ook een lijst van personen die ingelogd zijn op het netwerk (en dus aanwezig zijn op kantoor). Een specificatie hiervoor zou dan zijn:

LoginDepTelefoonboek

$DepTelefoonboek[geefLoginEnNummer/geefNummer]$

$ingelogd : \text{seq}\, char \rightarrow \{true, false\}$

...

geefLoginEnNummer

$ingelogd!$

$ingelogd! = ingelogd(naam?)$

Hier zien we het mechanisme om overerving aan te duiden. Na de naam van de bovenklasse (er mogen meerdere bovenklassen zijn) volgt een lijst met substituties. Bij het schema voor de operatie $geefLoginEnNummer$ moeten we alleen de nieuwe elementen opgeven: automatisch worden de elementen uit het overeenkomstige schema van de bovenklasse toegevoegd. Als we een operatie werkelijk willen overschrijven zonder dat het oude schema wordt ingevoegd dan moet dit bij de substitutielijst worden opgegeven. Zo zouden we bij een cirkel, waar we de $roteer$operatie overschrijven in plaats van aan te vullen, de overerving aangeven door

Cirkel

$Ellips[diameter/groteAs,$ **redef** $roteer]$

$kleineAs = diameter$

...

19.10 VERGELIJKING

Beide benaderingen hebben hun voor- en nadelen. Het belangrijkste nadeel van gebeurtenistalen is dat de specificaties zeer eigenaardig aandoen, en vrij ingewikkeld zijn. Het is namelijk niet mogelijk om een specificatie van een operatie op zich te geven; men moet combinaties van operaties definiëren. Voordeel van deze manier van werken is dat ze nauw kan aansluiten bij de testpraktijk. Als wordt aangegeven wat de uitvoer moet zijn van een sequentie van operaties, dan vertaalt zich dit onmiddellijk in een test: voer de sequentie uit, en kijk naar het resultaat. Een volledige testsuite dient natuurlijk rekening te houden met verschillende waarden voor de parameters.

Modeltalen sluiten beter aan bij de klassieke praktijk van het specificeren van operaties met pre- en postcondities. Ze maken het mogelijk de operaties een voor een te bekijken. Maar er zijn ook nadelen aan verbonden. Eén ervan is de noodzaak om de toestand te beschrijven. Merk op dat er meestal gebruikt gemaakt wordt van een combinatie van de attributen zoals gebruikt in UML, en de deelobjecten die in UML met een aggregatieassociatie worden voorgesteld. In UML is dan ook duidelijk gekozen voor de aanpak met toestanden. Bij de bespreking van UML hebben we al gezien dat dit werken met attributen problemen oplevert. Zo is er het probleem van afgeleide attributen: het is niet duidelijk welke attributen als originele attributen, en welke als afgeleid moeten beschouwd worden. Bovendien is er het probleem dat het werken met expliciete toestanden de implementatie stuurt. Nemen we het voorbeeld van de twee implementaties van `Punt2D`. We hebben twee mogelijke implementaties gegeven: deze met cartesiaanse coördinaten, en deze die werkt met poolcoördinaten. In een modeltaal zijn we verplicht een van deze twee (of een andere) als model te kiezen. Bijgevolg zal een programmeur geneigd zijn deze keuze ook te implementeren, zelfs al is zij niet optimaal. Voor het geval van `Punt2D` is de keuze nog eenvoudig en duidelijk, maar dat verandert wanneer we moeten werken met ingewikkelde datastructuren. Het kan een belangrijke beslissing zijn welke structuur men kiest, en deze keuze kan men pas maken op het ogenblik dat men goed weet welke eigenschappen van de structuur belangrijk zijn, dus nadat men de specificatie heeft gemaakt. In principe is het onmogelijk om met algebraïsche specificatie uit te drukken dat een object een binaire zoekboom is. Dit is dan ook de bedoeling: op deze manier is men verplicht te specificeren zonder datastructuur in gedachten.

Het vermelden van attributen (en dit geldt zowel voor een modelleertaal zoals UML als voor een specificatietaal zoals Object-Z) doorbreekt uiteindelijk de modulariteitseis dat de grens van een module moet beschreven worden onafhankelijk van het inwendige. Gebeurtenistalen zijn daarom zuiverder dan modeltalen. Deze zuiverheid heeft ook zijn nadelen. Bij het beschrijven van een systeem delen we dit systeem op: zo hebben we bijvoorbeeld een klassendiagram dat een inventaris geeft van alle mogelijke objecten van het systeem. Nu is het doenbaar elke klasse binnen een systeem met algebraïsche specificatie te beschrijven, als de klassen tenminste niet te ingewikkeld zijn. Maar als we het hele systeem zo willen beschrijven, dan krijgen we een enorm complex geheel. Volgens de filosofie van algebraïsche specificatie mogen we geen interne delen onderscheiden, en dus ook geen gebruik maken van een klassendiagram: alles moet afgehandeld worden op het niveau van de operaties van het systeem. Dit geeft een enorm kluwen. Zoals we gezien hebben kan men bij modeltalen gebruik maken van aggregatie: bij algebraïsche specificatie is dit principieel niet mogelijk.

HOOFDSTUK 20
REFACTORING

20.1 DE GESCHIEDENIS VAN NETSCAPE

De eerste veelgebruikte grafische webbrowser was Mosaic, gebouwd door een team van de universiteit van Illinois. In 1994 bouwden ze de opvolger Netscape, met versies voor verschillende platforms (Windows, Unix, Mac). Microsoft bracht de eerste versie van Internet Explorer uit halverwege 1995, maar tot ongeveer 1997 bleef Netscape met voorsprong de meest populaire internetbrowser. Het is pas met IE 3.0 dat Microsoft de kloof kon dichten. Daarna ging het met Netscape snel bergaf. Een geheel vernieuwde versie, Communicator 6.0, werd zelfs nooit afgewerkt, en er werd teruggegrepen naar een oude versie 4.0 met een poging om dit op te knappen. Veel is het nooit geworden, en uiteindelijk werd Netscape opgekocht door AOL.

Het probleem was onzorgvuldig design. De eerste versie van Netscape was een *quick and dirty* product: Het werkte, maar de code was slecht gestructureerd en dus moeilijk te onderhouden en uit te breiden. Naarmate de browser groter werd en het internet ingewikkelder (allerhande soorten plug-ins, nieuwe standaarden, animatie,...) stegen de kosten onevenredig veel: Mosaic was gemaakt door een team van 10 ontwikkelaars, Navigator 4.0 door 120 mensen. IE 1.0 was ook niet goed gestructureerd, maar Microsoft onderkende het probleem tijdig en herzag de code grondig voor IE 3.0.

Internetbrowsers bestaan in een snel evoluerende omgeving, zodat code dikwijls grondig moet herzien worden. Hierdoor degradeert de kwaliteit: rond 1996 crashte zowat elke browser regelmatig. Bovendien wordt de code moeilijker en moeilijker aanpasbaar. Vandaar de beslissing van Netscape in 1998 om van de grond af aan opnieuw te beginnen. Dit is uiteraard een zeer duur proces, en men is op zoek gegaan naar methodes om bestaande code te verbeteren. Dit heeft geleid tot het begrip *refactoring*: het *herwerken* of herschikken van code. Microsoft heeft de code voor IE succesvol herwerkt voor versie 3.0, Netscape probeerde eerst om vanaf nul te herbeginnen (versie 6.0), en herwerkte dan de bestaande code zonder veel succes. Dit lag o.a. aan het feit dat er voor de herwerking veel te weinig tijd voorzien was.

20.2 TOEPASBAARHEID VAN HERWERKEN

Bij het schrijven, uitbreiden en wijzigen van software is het altijd noodzakelijk om de organisatie van de software aan te passen. Eén van de basisprincipes van herwerken is om de twee processen te scheiden: men maakt eerst de bestaande code klaar om wijzigingen in aan te brengen, en *daarna* wijzigt men de functionaliteit. Herwerken is dus de *reorganisatie* van code *zonder* dat de functionaliteit wijzigt: de resulterende code dient op exact dezelfde manier te werken als de oorspronkelijke. De bedoeling hiervan is tweevoudig. Het eerste doel is om de code zo duidelijk en begrijpbaar mogelijk te maken; het tweede doel is om de software flexibel te maken. Het is duidelijk dat her-

werken geen nut heeft tenzij de code nog moet gewijzigd worden. Er zijn verschillende situaties waarbij herwerken aangewezen is:

- Bij het gebruik van *incrementele* ontwikkelingsmethodes. Hierbij kan het zowel gaan om een voorafgaand geplande incrementele methode met duidelijk afgelijnde builds, of, zoals in het geval van de browser, van een uitbreiding van bestaande software omwille van een wijziging in de omgeving.
- Bij het *hergebruiken* van code. Indien de code ongewijzigd kan overgenomen worden heeft herwerken niet veel zin, maar dikwijls wordt overgenomen code wel gewijzigd. Herwerken maakt dan deel uit van het leren kennen van de code. In plaats van vreemd uitziende constructies in de overgenomen code goed te leren kennen verwijdert men ze gewoon.
- Bij het ontwikkelingsproces. Vaak blijkt dat het ontwerp van de code gebreken vertoont en dat reorganisatie zich opdringt. Dit kan dan best zo snel mogelijk gebeuren, met het oog op testen en debuggen.

Reorganisatie van code is een proces dat uitgaat van de code zelf, en niet van een analyse- of ontwerpdocument. Het gaat dan ook uit van lokale mankementen van de code: elke keer dat code moeilijk te begrijpen is, of niet flexibel, zal men proberen te herstructureren. Het mankement wordt ook zo lokaal mogelijk opgelost: dikwijls binnen een methode. Slechts als men de noodzaak voelt om hogerop te gaan, zal dit gebeuren, en men zal op deze manier nooit verder geraken dan het niveau van ofwel een hiërarchie van van elkaar overervende klassen, ofwel een kleine groep van samenwerkende klassen.

De reorganisatie berust op een relatief klein aantal, welomschreven acties. Vermits het de bedoeling is om de functionaliteit van de software te behouden, dient gegarandeerd te worden dat dit inderdaad zo is. Het is dus uit den boze om de fundamentele structuur van de software te hertekenen: wie veel wijzigingen tegelijkertijd doorvoert loopt een groot risico op verandering van de functionaliteit. Ook is het niet de bedoeling om iets nieuws te proberen: men gebruikt best alleen acties uit de standaardlijst. Elk van deze acties heeft dan ook een vaste *naam*, zodanig dat het gemakkelijk is om ernaar te verwijzen. Als zulk een actie correct wordt uitgevoerd, dan is er gegarandeerd dat er niets verandert aan de functionaliteit. De acties zijn echter redelijk delicaat, en er kunnen onverwachte problemen opduiken. Nemen we als voorbeeld een van de eenvoudigste acties: het hernoemen van een methode van een klasse (de naam van deze methode is, zeer toepasselijk, *hernoemen van methode*[1]. Op het eerste gezicht is dit zeer eenvoudig: Als de oude naam `foo` was, en de nieuwe `bar`, verandert men gewoon overal in de code `foo` in `bar`. Hiermee kunnen verschillende problemen opduiken. Ten eerste is het best mogelijk dat er andere entiteiten zijn die ook `foo` heten, en waarvan het niet de bedoeling is dat ze van naam veranderen. Maar een nog groter probleem duikt op als de naam `bar` reeds gebruikt wordt voor andere entiteiten. In de meeste gevallen levert dit enkel compileerfouten op, maar er is een geval waar de fout subtieler kan zijn, en dat is wanneer de naam `bar` ook al bestaat als de naam van de functie met dezelfde signatuur uit een of andere bovenklasse. In dit geval overschrijft de hernoem-

[1] In de terminologie die we vroeger gebruikten is een methode de implementatie van een operatie, en is het de operatie die een naam heeft. Zo is er in een hiërarchie van klassen bijvoorbeeld een *operatie* `foo`, waarbij de methode kan overschreven worden. Vanuit dit standpunt zou de naam *hernoem operatie* moeten zijn.

de operatie de andere `bar`operatie, en zijn de gevolgen onvoorspelbaar. Overigens kan het ook zijn dat `foo` een overschrijving was van een operatie die hogerop gedefinieerd was, en ook dan kunnen de gevolgen van hernoemen zeer vreemd zijn.

Herwerken steunt dan ook op het zeer dikwijls testen van de herschreven code. Het is volstrekt noodzakelijk om te werken met een automatische testomgeving, zodanig dat de wijzigende software zeer dikwijls kan onderworpen aan rigoureuze tests. Het veelvuldig laten lopen van de tests zorgt ervoor dat de oorzaak van bugs snel kan worden opgespoord. Immers, als na een kleine wijziging een bug wordt gesignaliseerd door de tests, dan is het duidelijk dat de bug te maken heeft met die kleine wijziging. Misschien is ze er niet door veroorzaakt, maar ze is dan wel op de voorgrond gekomen door die wijziging. Als men pas test na tientallen veranderingen te hebben aangebracht, moet men uitzoeken bij welke van die veranderingen die bug hoort.

Zoals reeds blijkt uit het voorbeeld van een methode met een nieuwe naam, is het met de hand uitvoeren van een herwerkingsoperatie delicaat en vervelend werk, en dus typisch iets dat we door een programma willen laten doen. Zo'n programma's kunnen bestaan, juist omdat het herwerkingsproces is opgebouwd uit welomschreven acties. Dergelijke programma's bestaan dan ook, en ze versnellen het werk ten zeerste. Merk op dat zulke programma's vrij ingewikkeld zijn, omdat ze niet kunnen volstaan met een oppervlakkige analyse van de code. Als, met bovenstaand voorbeeld, zo een programma ergens de uitdrukking `p.foo()` ziet staan, moet het weten of `p` tot de klasse behoort waarbij `foo` moet veranderd worden in `bar` of niet. Dergelijke programma's zijn dus niet veel eenvoudiger dan een compiler voor de betreffende taal. Niet alleen is zo een programma zeer nuttig om de actie zelf uit te voeren (waarbij er niet alleen tijdwinst is, maar waarbij de kans op fouten ook ten zeerste vermindert), ook kan zulk een programma controleren of de actie wel toelaatbaar is. Nemen we het voorbeeld van een andere actie: *extractie van code* uit een methode. Bij deze actie wordt een gedeelte van de code in een methode aangeduid. Deze wordt dan ondergebracht in een nieuwe, aparte methode. Op de oorspronkelijke plaats komt dan een oproep te staan naar de nieuwe methode. Hierbij moeten verschillende controles gebeuren op lokale variabelen. Indien een lokale variabele die gedeclareerd wordt binnen het aangeduid stuk code buiten het stuk nog gebruikt worden, dient men na te gaan of de waarde van die variabele met een uitvoerparameter moet terug geven. Als het uit te knippen stuk zelf lokale variabelen gebruikt die vóór het stuk werden geïnitialiseerd, dan moet men ook nagaan of de *waarde* gebruikt wordt, en eventueel een parameter voorzien in de oproep van de methode. Merk op dat men wel degelijk moet nagaan of de *waarde* belangrijk is, omdat domme variabelen vaak hergebruikt wordt. Nemen we bijvoorbeeld

```
int tmp;
tmp=a;a=b;b=tmp;
...
tmp=0;
for (int i=0;i<aantal;i++)
    tmp+=tab[i];
```

Hier wordt dezelfde variabele op twee plaatsen gebruikt, maar het is duidelijk niet nodig om de waarde van `tmp` mee te geven als parameter indien het tweede stuk apart wordt gezet: dit levert een volstrekt overbodige parameter op, en dus zinloze koppeling. Er is overigens een herwerkingsactie waarbij de `tmp`-variabele in het tweede stuk

code vervangen wordt door een nieuwe variabele.

Door het bestaan van herwerkingstools is refactoring veel goedkoper en sneller geworden. Op deze manier worden herwerkingsacties meer en meer verweven in het proces van het maken of wijzigen van software. Men gaat niet meer eerst herwerken en daarna de aanpassingen doen, maar men voert die herwerkingsacties uit die nodig zijn voor een bepaalde aanpassing. Dit heeft als voordeel dat men code die men niet moet aanpassen niet eens moet bekijken. Nadeel is wel dat men op deze manier minder goed gestructureerde code krijgt: als een organisatiefout die wijziging moeilijk maakt niet direct in de buurt van het probleem ligt, dan zal men de refactoringsactie niet uitvoeren. Nemen we het voorbeeld van een operatie `foo()` die is ondergebracht bij een klasse A maar eigenlijk hoort bij klasse B. Als men nu de functionaliteit van B wil uitbreiden met een nieuwe methode en daarvoor `foo()` nodig heeft, dan zal men deze niet vinden, en bijvoorbeeld zelf een nieuwe `fookloon()` schrijven, eventueel inline in de nieuwe methode van B. Wie echter systematisch de code doorloopt zal bij A zien dat de methode `foo()` niet op zijn plaats staat, en deze overbrengen naar B. Welke methode het beste is: herwerken en wijzigen van code zeer duidelijk scheiden, of een vermenging van de twee, zal afhangen van een aantal factoren. Zo is herschikken zonder wijziging een goede techniek om code voor hergebruik te leren kennen, en zijn aanpassingen pas mogelijk als men de code enigszins kent. Als men verwacht dat men veel aanpassingen zal moeten doen is het dan ook interessant om eerst alle code te herschikken. Als men slechts enkele kleine wijzigingen verwacht moet men niet alle code leren kennen, en kan men gericht herschikken tot de wijzigingen kunnen worden aangebracht. Merken we tenslotte nog op dat het systeem van scheiden van herwerken en wijzigen niet altijd op veel bijval kan rekenen van het management, zeker niet wanneer er een strikt tijdschema te halen valt. De doelstellingen van herwerken, duidelijkheid en flexibiliteit worden dan vaak gezien als eigenschappen met weinig belang. Een periode uittrekken om code te veranderen in code die juist hetzelfde doet lijkt dan, speciaal voor mensen die niet rechtstreeks met het programmeerproces verbonden zijn, volledig nutteloos.

Herschikking van code maakt het tot op zekere hoogte mogelijk om fouten in het ontwerp van code op te vangen. Dit kan echter geen excuus zijn om onzorgvuldig te gaan ontwerpen. Zoals gezegd is refactoring een bottom-upproces: uitgaande van lokale eigenschappen van code wordt geherstructureerd. Hierdoor worden kleine ontwerpfouten gemakkelijk afgevlakt. Naarmate de fouten echter op een hoger niveau liggen komen er echter problemen. Ten eerste heeft men, door het werken vanop het niveau van code, niet veel overzicht: het is daarvoor dat UML-diagrammen dienen. Ten tweede is er op den duur veel code betrokken bij het herwerken, zodat het proces ingewikkeld en traag wordt.

20.3 UITWERKING

De volgende stappen worden ondernomen bij herschikking:

- Het overlopen van de code en het ontdekken van problemen.
- Het bepalen van de juiste actie of acties om te herschikken.
- Het uitvoeren van de acties en het testen van het resultaat.

Het overlopen van de code kan, zoals gezegd, gebeuren naar aanleiding van de adoptie

van code voor hergebruik, of omwille van de noodzaak tot aanpassing. Bij nieuwe code is het ook nuttig aan herschikking te denken bij de acceptatie van code. Als er een review van code gebeurt door een groep bestaande uit de programmeur en anderen betrokken bij het project dan is dit het ogenblik om na te gaan of de code door degenen die ze niet geschreven hebben als duidelijk en natuurlijk ervaren wordt: zo niet dient herschikking overwogen te worden. Immers, op dit ogenblik is het waarschijnlijk dat er aan de code nog wijzigingen worden aangebracht, omdat er nog getest moet worden (eventueel alleen integratietests). Bijgevolg is verbetering van leesbaarheid en van flexibiliteit belangrijk. Merk op dat het hier misschien alleen gaat over flexibiliteit die te maken heeft met verbetering van fouten, en niet over flexibiliteit in verband met uitbreiding van de mogelijkheden. Zoals we zullen zien zijn sommige herschikkingsacties toegespitst op het laatste soort flexibiliteit. In een omgeving met een lage graad van volatiliteit van vereisten zullen deze acties een lage prioriteit krijgen, vooral omdat ze kunnen worden uitgesteld tot het ogenblik dat de noodzaak zich voordoet.

Een belangrijk aspect van de kennis van herschikkingstechnieken is dat ze de nadruk leggen op een goede lokale organisatie van code. Het vinden van potentiële probleempunten is van cruciaal belang. Vandaar dat het nuttig is om een aantal indicatoren van problemen in meer detail te bekijken, zelfs voor programmeurs die nauwelijks in contact zullen komen met herwerkingstechnieken. Immers, het vermijden van zulke probleempunten leidt tot goede code op zich. Een gids voor refactoring kan dan gelezen worden als een gids voor stijlvol programmeren, maar dan op een hoger niveau: waar een stijlgids het heeft over de code *binnen* een methode, gaat refactoring voornamelijk over de organisatie van code zoals ze verdeeld wordt over de operaties van een klasse, en over verschillende klassen.

Bij een goed ontwerp en uitwerking is herwerken van code niet zo dikwijls nodig. Soms worden de gevolgen van een aantal ontwerpbeslissingen pas duidelijk bij programmeren, en dan dient het ontwerp aangepast: de herwerkingsactie op codeniveau weerspiegelt zich dan in een wijziging in het ontwerp. Toch is dit soort aanpassingen vrij klein als het gaat om code die afgeleid wordt uit een ontwerp gebaseerd op stabiele vereisten. Het is pas als die vereisten volatiel zijn dat code vaak wordt herwerkt en zijn flexibiliteit verliest: bij Netscape was dat het geval na ettelijke versies. Dan is refactoring een hoge noodzaak.

Het is echter niet gemakkelijk om de symptomen van probleempunten zeer duidelijk te omschrijven. Er is een beperkt succes met het gebruik van softwaremetrieken. Zo kan het zijn dat een methode van een klasse vaak verwijst naar attributen en operaties van objecten in een bepaalde andere klasse. Dan kan men de vraag stellen of deze methode niet thuishoort in die andere klasse. Nu kan men het aantal van dergelijke verwijzingen in principe tellen met een programma (het is m.a.w. een softwaremetriek), en dus is het mogelijk een programma te schrijven dat dergelijke herwerkingsactie aanbeveelt. Er kunnen echter goede redenen zijn waarom de actie niet gewenst is. Herwerken van code is dus voornamelijk werk voor een ervaren programmeur. Het valt trouwens op dat bijna alle herwerkingsacties een tegengestelde actie hebben. Dit kan een gelijksoortige actie zijn: een methode hernoemen wordt teniet gedaan door de methode terug de oude naam te geven, maar soms komen acties in paren. We hebben gezien dat we bij extractie van code een nieuwe methode creëren waarin code uit een lange methode wordt ondergebracht. Deze actie heeft zijn tegenpool: *inline brengen van methode*, waarbij een methode wordt verwijderd, terwijl haar code wordt onder-

gebracht in een oproepende methode. Bijgevolg is het niet zo dat men moet zoeken naar zoveel mogelijk gelegenheden om een herwerkingsactie toe te passen: meestal is het de kunst om de voor- en nadelen tegen elkaar af te wegen. Het is in verband met herwerken van code gebruikelijk om te spreken van de geur (Eng: *smell*) van probleemcode. Een ervaren programmeur is veel meer gevoelig voor dergelijke geur dan iemand die nieuw is in het vak. Toch is het mogelijk een aantal herkenningspunten op te geven.

We kunnen drie grote niveaus onderscheiden waarop kan gewerkt worden:

- Binnen een enkele klasse.
- Samenwerking tussen twee klassen, apart of binnen een hiërarchie van overerving.
- Tussen verschillende hiërarchieën van klassen.

Verder zijn er twee criteria die gebruikt worden:

- De begrijpelijkheid van code.
- De flexibiliteit van code.

Intuïtief kan men het tweede criterium als volgt beschrijven: code is flexibel als een wijziging in de functionaliteit weinig werk vraagt. Een van de principes hierbij is *eenheid van plaats*: als om een functie te veranderen code moet worden aangepast op verschillende plaatsen, dan is dit veel werk (al deze plaatsen moeten gezocht en gevonden worden). Bovendien is er veel kans op fouten: een plaats die gewijzigd moet worden kan vergeten worden. Bovendien is er een grote koppeling: voor elke te wijzigen plaats moet worden nagegaan of de wijziging de overige werking van het systeem niet stoort. Een tweede principe is *standaardisatie* van de aanpassing. Zo mogelijk moet de verandering gebeuren op een manier die *geen* inventiviteit van de programmeur vraagt. Een standaardmethode is gemakkelijker om juist uit te voeren en levert dus minder fouten op; bovendien is ze gemakkelijker te begrijpen als de code achteraf weer moet gewijzigd worden. Sommige herwerkingsacties kunnen verschillende soorten problemen oplossen. Het zal het nuttigste zijn om herwerken meer in detail te bespreken aan de hand van symptomen die op problemen wijzen: van daaruit kunnen we dan een aantal herwerkingsacties nader belichten. Het is belangrijk om te realiseren dat veel van de indicatoren van probleemcode al kunnen herkend worden bij het ontwerpproces. Dat geldt dan niet zozeer voor problemen binnen een klasse, maar wel voor klassenoverschrijdende problemen.

20.4 PROBLEMEN BINNEN EEN KLASSE

Veel van de problemen die optreden binnen een enkele klasse kunnen ook optreden met code die verspreid is tussen verschillende klassen. De eenvoudigste herwerkingsoperatie is het hernoemen van elementen: operaties, attributen, associaties, klassen. Dit verandert niets aan de functionaliteit en is dus puur gericht op betere begrijpelijkheid. Toch moet de nodige voorzichtigheid in acht worden genomen.

In totaal onderscheiden we volgende probleempunten:

(1) Onduidelijke naamgeving.
(2) Onjuiste indeling private operaties.

(3) Overmaat aan conditionele uitdrukkingen.
(4) Onechte gegevensvelden/deelobjecten.
(5) Te grote klasse.
(6) Overdaad aan parameters.
(7) Intelligente attributen.

Waarvan we het eerste reeds behandeld hebben.

Een tweede probleempunt heeft te maken met private hulpoperaties. Bij het ontwerp van code hebben we gezien dat er twee redenen zijn om private operaties in een klasse aan te brengen. Eén ervan was het bestaan van gedupliceerde code: dezelfde handeling wordt in twee of meer operaties, of binnen een operatie, verscheidene malen toegepast. Dit is vaak iets wat op ontwerpniveau over het hoofd wordt gezien: pas bij het voor de tweede maal coderen heeft de programmeur het gevoel dat "hij die code al eens eerder heeft gezien". Dit is het ogenblik om te besluiten om de duplicaatcode onder te brengen in een aparte methode[2]. Wordt duplicaatcode gevonden in bestaande code dan wordt de methode *extraheer methode* gebruikt, en worden de verschillende kopies van de code allemaal vervangen door een oproep naar de nieuwe operatie. De tweede reden om private operaties te creëren is dat een grote methode kan ingedeeld worden in verschillende deeloperaties. Soms is het mogelijk een deel van de code zinvol onder te brengen in een aparte operatie. Niet alle specialisten zijn enthousiast over zo'n maatregel. Onder het motto dat encapsulatie op alle niveaus maximaal moet zijn, eist men dat de code, vermits ze een deel is van die ene grote methode, binnen deze methode verborgen blijft. De maatregel om code te extraheren maakt de code van de grote methode leesbaarder, maar ze verdringt de andere (private) methodes uit de aandacht. In elk geval is hij niet gewenst als de lange code werkelijk samenhangend is, en bijvoorbeeld een enkel algoritme beschrijft, waarvan de losse onderdelen geen zin hebben. Vandaar er naast *extraheer methode* ook een tegengestelde actie is, *inline brengen van methode*.

Een derde probleempunt heeft te maken met het voorkomen van conditionele uitdrukkingen in de vorm van `if`- of `switch`opdrachten. Uiteraard is het niet de bedoeling om alle zulke opdrachten te verwijderen, maar ze kunnen een symptoom zijn van verschillende fouten. Conditionele uitdrukkingen zorgen ervoor dat één stuk code verschillende zaken doet, en men moet de vraag stellen of deze verschillende zaken niet moeten gescheiden worden. Het probleem kan zich voordoen op verschillende niveaus. Bij een ervan is er sprake van een methode die twee of meer geheel verschillende zaken doet. Geven we een (extreem) voorbeeld:

```
void zetAfmeting(int a, bool opzij){
    if (opzij)
        breedte=a;
    else
        lengte=a;
}
```

Deze code kan gemakkelijk vervangen worden door *twee* methodes; de herwerkoperatie heet dan ook *vervang parameter door expliciete methodes*:

[2] In de praktijk blijkt dat programmeurs dit meestal pas doen als ze de code voor de *derde* maal intikken.

```
void zetBreedte(int a){
    breedte=a;
}
void zetHoogte(int a){
    hoogte=a;
}
```

Het alternatief is veel eenvoudiger om te begrijpen: het gedrag wordt onmiddellijk afgeleid uit de procedurenaam. In het eerste geval moet de oproeper ook expliciet kiezen voor lengte of breedte door de `opzij`parameter in te vullen. Bovendien moet hij in de code kijken om te zien wat er juist gebeurt.

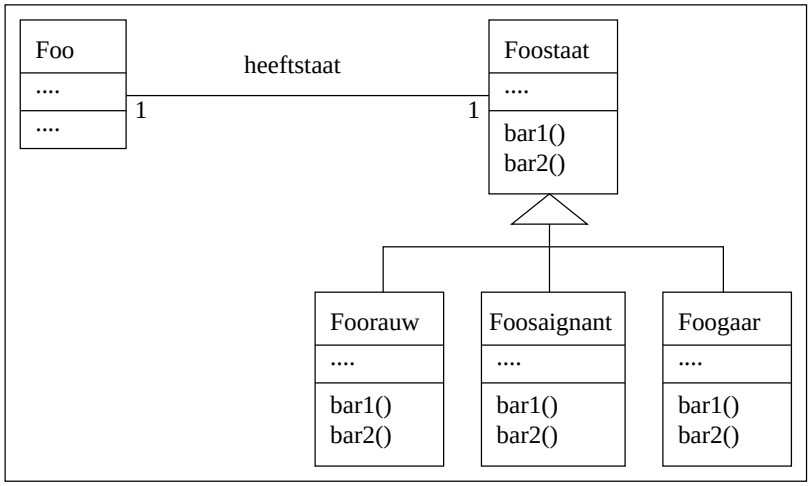
Een tweede probleem dat gekenmerkt wordt door conditionele uitdrukkingen is het bestaan van aparte staten en/of deelklassen. Zoals we weten is er een nauw verband tussen de twee: overerving met restrictie kan gemodelleerd worden met verschillende staten. Er zijn verschillende mogelijkheden:

- De conditie hangt niet af van de toestand van het object. Meestal is er dan geen probleem, maar als de conditie afhangt van de toestand van een ander object, kan het zijn dat in de andere klasse er staten en/of deelklassen zijn.
- De conditie hangt af van de toestand van het object, maar de toestand kan niet zo veranderen dat een ander alternatief gekozen wordt. Als er bijvoorbeeld twee alternatieven A en B zijn, dan zijn er objecten die altijd A kiezen en objecten die altijd B kiezen: welk van de twee het wordt is bepaald door de constructor. In dit geval zijn er eigenlijk twee klassen. Ofwel dient de ene een deelklasse te worden van de andere (en wordt de problematische methode overschreven), ofwel moeten ze beide deelklasse worden van een gemeenschappelijke bovenklasse. Waarschijnlijk is deze bovenklasse een abstracte klasse.
- De conditie hangt af van de toestand van het object, maar een object kan soms het ene alternatief en soms een ander kiezen. In dit geval zijn er staten.

Indien er duidelijk verschillende staten zijn, waarin verschillende methodes een andere reactie vertonen, dan kan het nuttig zijn om te werken met het *state pattern*. Hierbij worden de conditionele methodes ondergebracht in een hiërarchie van statenklassen. Veronderstellen we dat we een klasse `Foo` hebben met een `enum`veld met mogelijke waarden `rauw, gaar, saignant`. Er zijn twee methodes met condities:

```
void bar1(){
    switch(staat){
        case rauw:      actieA1();break;
        case saignant:  actieB1();break;
        case gaar:      actieC1();
    }
void bar2(){
    switch(staat){
        case rauw:      actieA2();break;
        case saignant:  actieB2();break;
        case gaar:      actieC2();
    }
```

Duidelijk hebben we hier drie staten. Met het statenpatroon verwijderen we nu de operaties `bar1()` en `bar2()` uit de klasse `Foo` en brengen ze onder in een abstracte klasse `Foostaat`. Er komen voor elke operatie drie methodes, die zijn ondergebracht in drie deelklassen van `Foostaat`. Het resultaat is te zien op Figuur 20.1. Merk op dat `Foostaat` en zijn deelklassen *technische* klassen zijn, in de zin dat er geen objecten uit de realiteit mee overeenkomen. Met een reëel object komt een gecombineerd paar van een `Foo` en een `Foostaat` overeen. Tenslotte moet vermeld dat het attribuut *staat* uit de klasse `Foo` zinloos is geworden, en dus ook verdwijnt. In plaats van dit attribuut te wijzigen wordt het `Foostaat`object vervangen.



Figuur 20.1. Een hiërarchie van statenklassen

Probleem vier is dat van onechte attributen en deelobjecten. Sommige lidvelden van een klasse hebben geen betekenis als een object “in rust” is, d.w.z. als er geen enkele operatie in uitvoering is. Dit wijst op een grote ontwerpfout: gegevensvelden moeten de toestand aanduiden, en een gegevensveld dat alleen zin heeft als een of andere operatie bezig is dient binnen die operatie te blijven. Een klassiek voorbeeld van zo’n ontwerpfout is een `Graaf`klasse die als deelobject een queue van knopen bevat die gebruikt wordt bij breedte-eerst zoeken. Deze queue is zinloos als er geen breedte-eerst zoeken aan de gang is, mag dus geen deel uitmaken van het `Graag`object.

Soms is het echter zo dat er een samenhangende groep van operaties is, meestal bestaande uit een publieke operatie en een aantal private operaties, die alleen door de publieke methode gebruikt worden. Vaak wisselen deze ook veel gegevens uit, zodat er óf veel parameters in de hoofdingen staan, óf lidvelden van de klasse gebruikt worden om de gegevens door te geven. Vanuit het standpunt van modulair ontwerp is dit geheel een module met goede afscheiding: het is dan ook nodig om dit in een aparte structuur onder te brengen. In dit geval verdient het aanbeveling al deze methodes samen met de gegevensvelden onder te brengen in een aparte klasse. Dergelijke klasse wordt een *methodeklasse* genoemd. Strikt genomen is het geen klasse: vaak is de enige publieke operatie de constructor: deze voert de operatie uit, en daarna kan het object verdwijnen.

Probleem vijf doet zich voor als we een grote klasse hebben, d.w.z. een met veel gegevensvelden en/of operaties. Het is dan bijna steeds mogelijk om deze op te split-

sen. Herinner je dat een vaste regel is dat op elk niveau van modulaire indeling er hoogstens vijf à tien verschillende deelmodules mogen zijn. Het kan zijn dat we een methodeobject kunnen maken om een deel af te splitsen; ook een gewone klasse kan soms gebruikt worden.

Zesde probleem is dat van lange lijsten parameters. Er is bijna steeds iets mis met een operatie die veel parameters heeft. De beschrijving van een grens van een module moet steeds zo klein mogelijk zijn, en een lange parameterlijst is een lange beschrijving. In veel gevallen zijn alle (of bijna alle) parameters gegevensvelden van één object. In dit geval kan men een referentie naar het object doorgeven in plaats van alle parameters. Merk op dat dit de code ook flexibeler maakt: als een operatie veel informatie nodig heeft van een bepaald object, is er veel kans dat een licht gewijzigde versie van de operatie ook nog andere informatie nodig heeft van dat object. Met de lange parameterlijst is dat niet mogelijk, met een referentie naar het object wel. Als er geen object is dat de gegevens samenbrengt, kan men zich afvragen of er geen zou moeten zijn: dit lijkt sterk op het methodeobject dat we bij het vorige probleem hebben ingevoerd.

Tenslotte is er het probleem van operaties die toegespitst zijn op een gegevensveld. Hierdoor heeft dit gegevensveld extra functionaliteit: het is een *intelligent attribuut* geworden. Nemen we het voorbeeld van een klasse `Persoon` met een attribuut `telefoonnummer` dat gedefinieerd staat als een element van de klasse string, en waarbij `Persoon` een operatie heeft om een zonenummer uit het telefoonnummer te isoleren. Alhoewel het hier alleen maar gaat over één attribuut met één operatie, is het toch beter deze apart te zetten. Immers, blijkbaar is een `telefoonnummer` een speciaal soort string, waaruit een zonenummer te halen valt. Deze operatie hoort duidelijk niet bij `Persoon`. Het is beter in dit geval een klasse `Telefoonnummer` te creëren en het zoeken naar het zonenummer daarin op te nemen. Merk op dat `Telefoonnummer` een deelklasse wordt van `string`. Een ander voorbeeld is dat van een geografische positie. Aanvankelijk zal die, waarschijnlijk in de vorm van twee getallen, als attributen worden geïmplementeerd. Als blijkt dat er verschillende operaties mee gebeuren (zoals overgaan van een coördinatenstelsel naar een ander, bijvoorbeeld van lengte- en breedtegraad naar plaats op een landkaart), dient dit overgeheveld te worden naar een aparte klasse.

20.5 PROBLEMEN TUSSEN TWEE KLASSEN

Veel van de problemen die optreden binnen een klasse kunnen ook voorkomen tussen twee klassen. De meeste problemen hebben echter te maken met een te grote koppeling tussen twee klassen. We onderscheiden volgende problemen:

(1) Duplicaatcode.
(2) Veel gebruik van gegevensvelden van de andere klasse.
(3) Onvolledige klassen.
(4) Onafhankelijke klassen met gelijkaardige functionaliteit.
(5) Problemen met delegatie.
(6) Het hagelverwijderensymptoom.
(7) Divergente verandering.

het eerste probleem is het bestaan van duplicaatcode. Het bestaan van duplicaten in verschillende klassen is niet zo wijd verspreid als duplicaten binnen een klasse, maar de problemen zijn hetzelfde. Het is mogelijk dat het dupliceren van code niet eenvoudig te vermijden is. Een goed criterium is hierbij of bij wijziging beide duplicaten moeten veranderd worden of niet. Als bijvoorbeeld twee klassen gebruik maken van binair zoeken in een gesorteerde tabel is er weinig reden om zich zorgen te maken: binair zoeken is een algoritme dat niet afhangt van de omstandigheden, alleen van het bestaan van een gesorteerde tabel. Als deze tabel in een van de klassen verdwijnt moet de code in de andere klasse niet aangepast worden. Als daarentegen twee klassen dezelfde code gebruiken om resultaten uit te schrijven, dan is het nodig om naar een oplossing te zoeken. Immers: als de uitschrijfmethode verandert (bijvoorbeeld niet meer naar het scherm maar naar een uit te printen rapport) dan moet de code op de twee plaatsen veranderd worden: dubbel werk, en dus gevoelig voor fouten. Voor het probleem zijn er twee, fundamenteel verschillende oplossingen. In beide gevallen gaat men zonodig eerst de gemeenschappelijke code in beide klassen in een aparte methode steken met *extraheer methode*, daarna gaat men proberen de twee methodes samen te brengen in een welbepaalde klasse.

- Als de twee klassen een gemeenschappelijke, vrij dichtbij liggende voorouder hebben, dan kan het mogelijk zijn om de code bij deze gemeenschappelijke voorouder te plaatsen. Doe dit alleen als de methode dan ook zinvol is voor de andere deelklassen (zelfs als ze niet wordt gebruikt). Uiteraard hebben in Java alle klassen een gemeenschappelijke voorouder, namelijk de klasse `Object`. Ga het dus niet te ver zoeken.
- Het is ook mogelijk dat de bovenklasse niet bestaat, maar dat het zinvol is ze te creëren. Vermits de bedoeling van herwerken is dat de functionaliteit van de code zal dit een abstracte klasse zijn: immers, instantiaties van deze bovenklasse zouden zich anders gedragen als een object dat beschreven werd door de oude code.
- Als de twee klassen niet nauw verwant zijn is het mogelijk dat ze beide code hebben die aan een derde klasse toebehoort. In ons voorbeeld van de uitschrijfoperatie is het logisch de gemeenschappelijk code onder te brengen in een of andere uitschrijfklasse: de manier van uitschrijven hangt niet zozeer af van de gegevens, maar wel van het uitschrijfmedium. Op deze manier worden uitvoeroperaties gegroepeerd. Elk soort uitvoermedium bepaalt hoe de gegevens vorm krijgen: de klassen die uitvoer leveren dienen alleen te weten dat ze hun data naar een uitvoerobject moeten sturen.

Een tweede probleem is dat van een operatie die veel kijkt naar een ander object dan `*this`. Naar dit object kan verwezen worden door een gegevensveld van de eigen klasse of door een parameter. Nemen we het voorbeeld van een klasse *Factuur*. In deze klasse is een operatie `berekenprijs(const Artikel&)` die de prijs van een item op de factuur berekent. Deze operatie kijkt uitgebreid naar een object van de klasse `Artikel`. Dit is slechte modulaire scheiding, en waarschijnlijk moet de methode worden ondergebracht bij de klasse `Artikel`.

Als er meerdere operaties zijn van klasse A die veel naar objecten van klasse B kijken en omgekeerd, dan is er iets grondig mis, maar de oplossing van het probleem is niet eenduidig. Soms is het nodig om de twee klassen samen te voegen. Dit is

bijna zeker het geval als er een 1-1 associatie tussen de twee klassen bestaat. Soms is het nodig ettelijke methodes en/of gegevensvelden van de ene naar de andere klasse te verhuizen zodat twee minder gekoppelde klassen ontstaan, en wordt het probleem daarmee opgelost.

Een derde probleem is dat sommige klassen de indruk geven niet volledig te zijn: klassen met zeer weinig methodes en/of gegevensvelden. Als er geen andere methodes zijn dan get- en setoperaties dan is er een andere klasse, of zijn er meerdere, die zorgen voor de operaties. Er zijn fundamenteel twee verschillende oplossingen voor dit probleem: ofwel verhuizen we code uit de manipulerende klasse(n) naar de probleemklasse, ofwel schaffen we de probleemklasse af. Bij de eerste oplossing kan het zijn dat een methode volledig wordt overgebracht, of dat gedeelten uit methodes eerst worden geëxtraheerd, waarna de nieuwe methodes worden overgebracht. Zo wordt de probleemklasse een volwaardige klasse. Dit zal vooral het geval zijn als er meer dan een manipulerende klasse is: duidelijk is er een gemeenschappelijke factor in de vorm van de gegevensvelden. Als er maar een manipulerende klasse is dan kan het zijn dat deze oplossing gekozen wordt, maar het kan ook zijn dat de probleemklasse wordt opgeslorpt door de manipulerende klasse. Dit is niet altijd mogelijk: als er een 1-n associatie is tussen manipulerende klasse enerzijds en dataklasse anderzijds, dan is dit opslorpen niet mogelijk, en trouwens ook niet wenselijk. Zeer dikwijls gaat het dan om een associatieklasse. Zoals we gezien hebben wordt een associatieklasse niet verondersteld om echte operaties te hebben. Bovendien bevat ze informatie die essentieel niet hoort bij een object op zich, maar bij twee objecten. Denken we hierbij maar aan de punten die bij een student-vakpaar horen. Zelfs al zouden we in zo'n geval erin slagen om de punten bij een van de twee deelnemende klassen te zetten (als er bijvoorbeeld voor elk vak maar één student is) dan zou dit nog aan de duidelijkheid schaden, en bovendien de flexibiliteit verminderen. Dit is het tegengestelde van de doelstellingen van het herwerken van code.

Bij probleem vier hebben we twee onafhankelijke klassen die gelijkaardige functies vertolken. Met onafhankelijk bedoelen we dat ze niet nauw verwant zijn door overerving. In dit geval is het de zaak om na te gaan of ze niet op een of andere manier samenhoren. Let er eerst op dat operaties van de twee klassen die hetzelfde doen dezelfde naam moeten hebben: zo valt de gelijkenis meer op. Daarna kan men proberen ze in een hiërarchie te krijgen. Het is mogelijk dat de ene klasse een uitbreiding is van de andere, en dus meer eigenschappen heeft. In dat geval wordt ze de deelklasse, en de andere de bovenklasse. Het kan echter ook zijn dat beide klassen een gedeelte hebben dat niet gemeenschappelijk is. In dit geval wordt het gemeenschappelijke stuk ondergebracht in een gemeenschappelijke bovenklasse.

de vijfde groep problemen heeft te maken met delegatie. Dit is een van de fundamentele objectgerichte programmeertechnieken: een object kan verwerking van een bericht delegeren naar een ander object. We spreken dan van een *delegerend* object en een *uitvoerend* object. Zo kan een delegerend object een aanvraag voor informatie doorspelen naar een ander object dat de informatie heeft, en het resultaat teruggeven aan de oorspronkelijke vrager. Dit is een goed mechanisme, maar het mag niet misbruikt worden. Soms zijn er klassen bij wie zowat alle methodes alleen maar delegeren en die zelf niets doen. Men kan dan de vraag stellen wat de zin is van deze klassen. Al wat ze doen is de berichtenketting langer maken.

Er dient opgemerkt te worden dat het gebeurt dat er klassen zijn die alleen maar

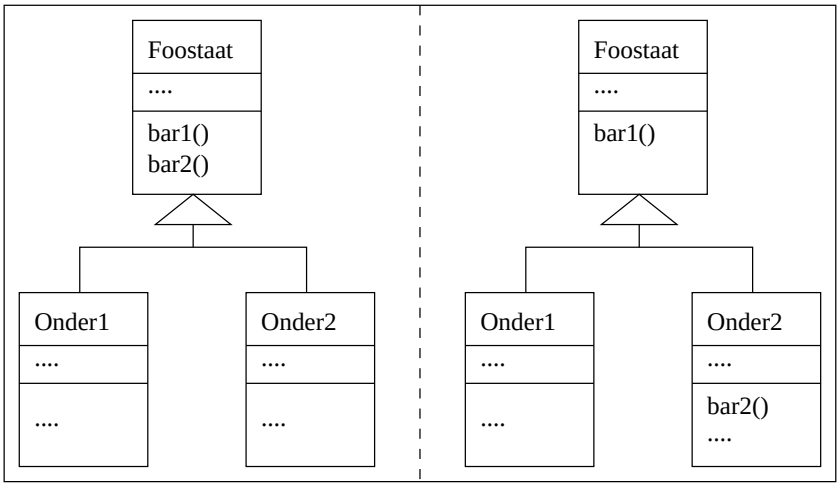
delegeren, maar dan wel naar verschillende andere klassen. Dit gebeurt bij het *façade-patroon*. Dit wordt gebruikt om de modulariteit van een geheel te verbeteren. Er wordt een deelsysteem afgescheiden, dat uit een vrij groot aantal klassen kan bestaan. Om hun onderlinge interacties te beperken zet men een façade op, die bestaat uit een of enkele klassen. Hierin staan enkel die operaties van het deelsysteem publiek die belangrijk zijn voor gebruikers *buiten* het deelsysteem, terwijl de klassen van het deelsysteem publieke operaties kunnen hebben die alleen belangrijk zijn *binnen* het deelsysteem. De façade scheidt het deelsysteem af: dit is een vorm van inkapseling, en het is uiteraard niet de bedoeling van deze op te heffen.

Behalve bij een façade is het echter meestal niet de bedoeling om objecten te hebben die niets anders doen dan delegeren. Daarbij moet men rekening houden met wat de delegerende klasse toevoegt aan de delegatie: zijn er niet-delegerende methodes? Zo niet, dan is het waarschijnlijk beter om de klasse te schrappen. Dit houdt in dat in al de code moet gezocht worden naar oproepen naar methodes van de klasse, en dat deze moet omgeleid worden naar het uitvoerende object.

Als de klasse wel een meerwaarde heeft, dan zijn er twee standaardvormen om de delegatie te realiseren. De ene is om de klasse een deelklasse te maken van de uitvoerende klasse. De originele methodes zijn dan onmiddellijk beschikbaar. De tweede methode is om de originele klasse, of liever zijn objecten te *verpakken* (in het Engels spreekt men van een *wrapper class*), een omslagklasse. We krijgen dan een klasse waarin elk object met aggregatie een object van de oorspronkelijke klasse bevat. Het speelt de meeste operaties door aan dit object. De tweede vorm wordt normaal gebruikt indien het gedrag van de oorspronkelijke klasse enigszins gewijzigd moet worden. We kunnen bijvoorbeeld een klasse `Telefoonboek` hebben die telefoonnummers teruggeeft. Als we nu object nodig hebben dat deze telefoonnummers in een iets andere vorm teruggeeft, dan gebruiken we een wrapper. Een tweede mogelijke reden om een wrapper te gebruiken in plaats van overerving is dat we niet alle publieke operaties van de uitvoerende klasse publiek willen maken. Als we een wrapper gebruiken moeten we immers de hele publieke interface expliciet definiëren: operaties worden overgenomen uit een bovenklasse, maar niet van een deelobject.

Als we dus een deelklasse hebben die maar een gedeelte van het publieke gedrag van een bovenklasse overneemt, dan kan het een goed idee zijn deze deelklasse te vervangen door een wrapperklasse. Maar het kan ook zijn dat overerving een goede oplossing is, maar dat er iets mis is met de plaats waar bepaalde eigenschappen gedefinieerd zijn. Hiervoor verwijzen we naar de diagrammen op Figuur 20.2. De beginsituatie zien we aan de linkerkant. We hebben de klasse `Boven` met daarin twee operaties gedefinieerd. Er zijn twee deelklassen. De tweede deelklasse, `Onder2`, gebruikt beide operaties gedefinieerd in `Boven`, maar de deelklasse `Onder1` gebruikt de operatie `bar2` niet. Nu maakt `bar2` wel deel uit van de gedefinieerde interface van `Onder1`, waardoor deze nodeloos ingewikkeld is. De oplossing is om `bar2` naar beneden te duwen, naar de klasse `Onder2`: op deze manier wordt ze uit `Onder1` verwijderd, en staat ze op een veel logischer plaats.

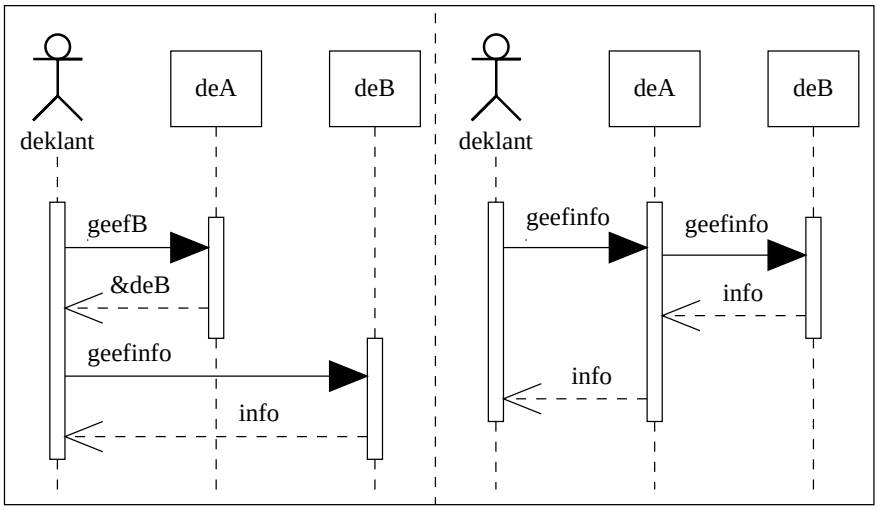
Wrappers en overerving zijn enkel geschikt om delegatie te implementeren op klassenniveau, waarbij een klasse eigenschappen overneemt van een andere, en niet op objectniveau. Als klasse A een deelklasse B heeft, en b is een object van B, dan heeft b alle operaties gedefinieerd in klasse A, maar er is geen *object* van A buiten b om waar b berichten naar moet sturen om die operaties uit te voeren. Bij een wrapper is



Figuur 20.2. Voor en na het naar beneden duwen van een operatie

zo'n A-object er wel, maar het bestaat niet buiten het omhullende b-object: zowel bij overerving als bij een wrapper is er voor de buitenwereld maar één object.

Er zijn echter gevallen waarin de delegatie echt is, in de zin dat het delegerende object en het uitvoerende object duidelijk verschillend van aard zijn. Dit is zeker het geval als de associatie tussen de twee een multipliciteit heeft die verschilt van 1-1. In zo'n geval is het zo goed als onmogelijk om wrapper- en overervingstechnieken te gebruiken, en is het meestal de beste strategie om de uitvoerder te verbergen. Bekijken we hiervoor de twee sequentiediagrammen in Figuur 20.3. Aan de linkerkant zien



Figuur 20.3. Het verbergen van de uitvoerder

we hoe een cliënt eerst aan `deA` het adres vraagt van een object `deB`, en daarna aan dit object informatie gaat vragen. Dit vereist veel kennis van de cliënt: hij moet een inzicht hebben in het verband tussen `deA`, `deB` en de info die hij wil. In het tweede

diagram is er veel meer ontkoppeling: al wat de cliënt moet weten is dat hij langs `deA` moet passeren om aan zijn informatie te geraken. In bijna alle gevallen is het rechtse schema te verkiezen boven het linkse.

In sommige gevallen is er een *ketting van berichten*. De cliënt vraagt `deA` het adres van `deB`, en vraagt dan aan `deB` het adres van `deC`, enzovoorts. Het kan zijn dat het verbergen van de uitvoerder voor elke stap de oplossing is, maar als de keten erg lang is dient men zich af te vragen of dit wel correct is. Duidelijk wordt van de cliënt veel kennis verondersteld over de structuur van de verschillende objecten. Uiteindelijk is er aan het einde van de ketting een uitvoerend object. De cliënt kent diens klasse (want hij spreekt dit object direct aan), en meestal is het in dit geval beter om dan de afgeleide verbinding van cliënt naar uitvoerend object een directe verbinding te maken, zodat de hele ketting overbodig wordt. Merk op dat dit de tegengestelde actie is van het verbergen van de uitvoerder.

Ten slotte zijn er nog twee symptomen die voornamelijk opduiken als men reeds bezig is met de software te veranderen. Het eerste symptoom heet hagel verwijderen (shotgun surgery). Dit refereert naar het verschijnsel dat men bij iemand die getroffen is door een schot hagel op verschillende plaatsen loodbolletjes moet verwijderen. Als men een bepaalde aanpassing doet (zoals overschakelen op een nieuwe soort databank), en men moet op veel verschillende plaatsen veranderingen doorvoeren, dan is dit het teken dat het bepaalde aspect dat men moet veranderen verspreid is over de code. Men gaat dan proberen methodes of datavelden te verplaatsen, en eventueel een klasse laten opslorpen door een andere om dit fenomeen te bedwingen. Nauw verwant is *divergente verandering*. Wanneer men een bepaalde klasse of operatie moet aanpassen bij een aantal niet-verwante veranderingen van de software, dan is het duidelijk dat dit item verschillende taken vervult. Waarschijnlijk is het dan beter om het item (meestal een klasse) op te splitsen in kleinere eenheden.

Veel van de herwerkingsacties zijn erop gericht om de flexibiliteit te verhogen. Hiervoor worden een aantal technieken gebruikt die het aantal elementen doen toenemen, zoals het gebruik van statenklassen. Deze zijn zeer nuttig als ze gebruikt worden: ze maken dan duidelijk welk soort flexibiliteit nodig was. Maar het toenemende aantal elementen maakt de software ingewikkelder dan voorheen. In een aantal gevallen zijn de technieken overbodig omdat ze flexibiliteit leveren op een punt waar ze niet gebruikt wordt. We krijgen dan bijvoorbeeld een hele hiërarchie van klassen die van elkaar overerven, maar waarvan er uiteindelijk maar enkele objecten voortbrengen. In dat geval kan het beter zijn om de structuur te vereenvoudigen, zelfs al wordt die daardoor meer rigide.

Geven we een voorbeeld van een herschikking van code die wel volgens de regels gebeurt, en die ook betere code oplevert, maar toch nog een loodzware structuur oplevert. Het voorbeeld komt uit het eerste lange werk over refactoring, en is een beetje aangepast. Het gaat over een menuprogramma. Een gebruiker geeft op welke dagen van de volgende week hij in de bedrijfscafetaria gaat eten, en krijgt dan het menu voor die dagen op het scherm. De interface met de gebruiker is met opzet zo eenvoudig mogelijk gehouden, omdat hij niet essentieel is voor de herwerking van code. De gebruiker schrijft gewoon een programma zoals dit:

```
#include "dagmenu.h"
int main(){
```

```
    menu.voegtoe(maandag,28);
    menu.voegtoe(woensdag,30);
    menu.print();
}
```

Als we dan in `dagmenu.h` gaan kijken dan vinden we de volgende code

```
enum dag={maandag,dinsdag,woensdag,donderdag,vrijdag,zaterdag};
class Menudag{
protected:
    int dagvanmaand;
    dag dagvanweek;
public:
    Menudag(dag d, int g):dagvanweek(d),dagvanmaand(g){};
    void printmenu(){
        cout<<dagvanmaand<<": ";
        if (dagvanweek==maandag)
            cout<<"Varkensgebraad met spruitjes"<<endl;
        else if (dagvanweek==vrijdag)
            cout<<"Gebakken tong met groentenkrans"<<endl;
        else
            cout<<"Friet met biefstuk"<<endl;
    }
};
class Menu{
protected:
    vector<Menudag> dagen;
    void voegtoe(dag d){
        dagen.push_back(Menudag(d));
    }
    void print(){
        for (vector<Menudag>::iterator it=dagen.begin();
                it < dagen.end();it++)
            it->printmenu();
    }
};
```

De opvallen de geur in dit voorbeeld wordt veroorzaakt door de conditionele uitdrukkingen in `Menudag::printmenu()`. Gaat het hier om staten of deelklassen? het laatste, want de toestand van een `Menudag` kan niet veranderen zodanig dat de uitkomst van de condities verandert (de toestand kan helemaal niet veranderen). Er is geen preferente dag die een bovenklasse zou kunnen worden. We maken dus een abstracte bovenklasse met zes deelklassen, een voor elke weekdag. Wat komt er in de bovenklasse? Het gegevensveld `dagvanweek` is overbodig geworden: voor elk van de zes deelklassen ligt de waarde ervan vast. `dagvanmaand` daarentegen blijft gemeenschappelijk. De `printmenu()`-operatie verschilt van deelklasse tot deelklasse: we maken ze dus virtueel, en definiëren ze pas in de deelklasse. We krijgen dan de volgende structuur:

```
class Menudag{
protected:
    int dagvanmaand;
public:
    Menudag(int g):dagvanmaand(g);
    virtual void printmenu()=0;
};
class Maandag:public Menudag{
public:
    Maandag(int g):Menudag(int g);
    void printmenu(){
        cout<<dagvanmaand<<": "
            <<"Varkensgebraad met spruitjes"<<endl;
    }
};
```

waarbij de dagen dinsdag tot en met zaterdag op analoge manier geprogrammeerd worden als maandag. Merk op dat we de `Menu` klasse moeten aanpassen. We kunnen geen vector van `Menudag`en meer gebruiken, maar moeten een vector van `Menudag*`-pointers gebruiken. Dit houdt in dat we bijvoorbeeld een destructor moeten schrijven:

```
    ~Menu(){
        for (vector<Menudag*>::iterator it=dagen.begin();
                it < dagen.end();it++)
            delete *it;
    }
```

Zijn we nu tevreden? Nog niet: in de deelklassen zit, in de functie `printmenu()` heel wat duplicaatcode. Veronderstel dat we de code willen veranderen om alles naar een HTML-pagina te schrijven: dan moeten we in zes klassen code veranderen. Dit is een typisch voorbeeld van hagel verwijderen. We brengen deze code dus samen op een plaats door een methode te extraheren met gemeenschappelijke code. Vermits de klassen met duplicaatcode allemaal een dichte voorouder hebben, `Menudag`, gaan we deze methode in de bovenklasse onderbrengen. We krijgen dus uiteindelijk

```
class Menudag{
protected:
    int dagvanmaand;
    virtual char* gerecht()=0;
public:
    Menudag(int g):dagvanmaand(g);
    void printmenu(){
        cout<<dagvanmaand<<": "<<gerecht()<<endl;
};
class Maandag:public Menudag{
public:
    Maandag(int g):Menudag(int g);
protected:
```

```
    char* gerecht(){
        return "Varkensgebraad met spruitjes";
    }
};
```

Indien we nu de code willen aanpassen om HTML-uitvoer te verkrijgen dan moeten we nog slechts de klasse `Menudag` aanpassen.

Alhoewel er een verbetering is van de kwaliteit van de code, hebben we toch een loodzware structuur gedefinieerd: een klasse met zes deelklassen, een voor elke dag van de week. Uiteindelijk hebben we daarmee ook nog een massa duplicaatcode: de code van de klasse `Maandag` beslaat 8 lijnen, en het enige verschil met de code voor `Dinsdag` is de naam van de klasse, en het gerecht van de dag. Bovendien, als ooit het menu verandert, een niet zo vreemde veronderstelling, moeten we elk van de zes klasse veranderen: duidelijk een geval van hagel verwijderen. In dit geval, hoewel de geur van slechte code zeer duidelijk aanwezig is, is er geen standaard herwerkingsactie die deze geur wegwerkt. De dagschotels voor de zes dagen moeten duidelijk samengebracht worden (hagel verwijderen), en de zes deelklassen zo mogelijk afgeschaft (duplicaatcode verwijderen). Een meer elegante oplossing van het probleem ziet er dan zo uit:

```
typedef int dagvanweek;
const dagvanweek MAANDAG=0;
...
const dagvanweek ZATERDAG=5;
class Menudag{
protected:
    dagvanweek dag;
    int dagvanmaand;
const char* gerecht[5];
public:
    Menudag(dagvanweek _dag, int g):dag(_dag)dagvanmaand(g);
    void printmenu(){
        cout<<dagvanmaand<<": "<<gerecht[dag]<<endl;
};
const char* Menudag::gerecht[5]={"Varkensgebraad met spruitjes",
        "Friet met biefstuk", ...
```

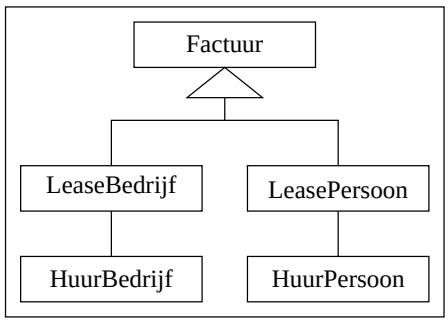
En uiteraard zal onze main ook weer moeten aangepast worden: we krijgen terug de oorspronkelijke vorm, behalve dat de dagen van de week nu met hoofdletters geschreven worden.

20.6 PROBLEMEN TUSSEN HIËRARCHIEËN

In de vorige paragrafen bekeken we problemen waarbij hoogstens één hiërarchie van klassen een rol speelde, in deze paragraaf gaan we nog een stapje verder en bekijken we een probleem waarbij de oplossing twee hiërarchieën bevat. De actie heet *het ontwarren van hiërarchieën*, en het symptoom dat aanwijst dat dit nodig is, is dat van *parallelle hiërarchieën*. Nemen we een voorbeeld van een bedrijf dat auto's verhuurt

op korte en lange termijn. Voor de facturatie heeft het bedrijf de hiërarchie van facturen die getoond wordt in Figuur 20.4. Er is een verschil in facturen naargelang de factuur wordt opgemaakt op naam van een bedrijf of op naam van een privépersoon, en naargelang het gaat over een lease- of over een huurcontract.

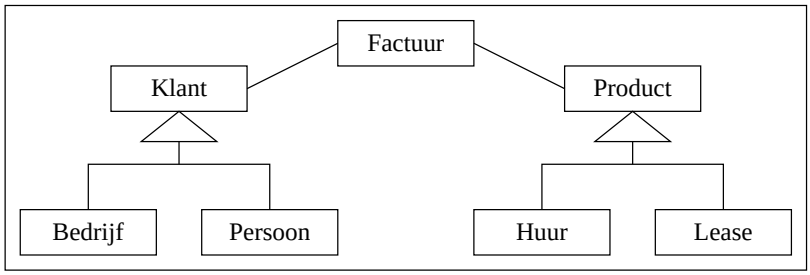
Opvallend aan deze hiërarchie is dat we er niet zinvol een klasse aan kunnen toe-



Figuur 20.4. Verstrengelde hiërarchieën

voegen: we moeten er altijd verschillende bijmaken, een geval van hagelverwijderen op hoog niveau. Nemen we bijvoorbeeld aan dat het bedrijf ook auto's gaat verkopen: dan moet het zowel een klasse `VerkoopBedrijf` als een klasse `VerkoopPersoon` maken. Als er een derde soort klant komt, bijvoorbeeld een vzw. met een ander BTW-stelsel als een gewoon bedrijf, dan moeten we onder `Factuur` een derde tak maken, met als klassen `LeaseVZW` en `HuurVZW`. We hebben hier twee hiërarchieën van klassen die in elkaar verstrengeld zijn geraakt. Merk op dat de ordening van de klassen in de hiërarchie die we hebben onnatuurlijk is, en dat de vier deelklassen dan ook anders zouden kunnen geordend zijn: bijvoorbeeld `HuurBedrijf` en `LeaseBedrijf` naast elkaar, eventueel als onderklassen van een klasse `Bedrijf`.

De oplossing is in dit geval duidelijk. Blijkbaar hangt de facturatie af van twee onafhankelijke gegevens: die van de klant (zoals de aanrekening van BTW, of misschien kortingen voor een categorie) en die van het product. Bijgevolg moeten er *twee* hiërarchieën zijn, een voor elke factor. Het resultaat ziet er dan uit zoals in Figuur 20.5 Merk op dat het mogelijk is dat deze oplossing niet echt voldoet, en dat we de



Figuur 20.5. Ontwarde hiërarchieën

verstrengelde oplossing moeten houden. Dit kan bijvoorbeeld het geval zijn als de prijzen voor producten voor privépersonen en voor bedrijven onafhankelijk van elkaar

kunnen veranderen. In dit geval is er essentieel een sterke koppeling tussen klanten en producten in de buitenwereld. Deze koppeling weerspiegelt zich dan in het model van Figuur 20.4.