

Klasse-interferentie

Rede
in verkorte vorm uitgesproken
bij de openbare aanvaarding van het ambt
van hoogleraar Onderwijs en softwareconstructie
aan de Open Universiteit
op maandag 7 juni 2010
door prof. dr. A. Bijlsma



Open Universiteit
www.ou.nl

Inleiding

Mijnheer de Rector Magnificus,
Beste collega's, familieleden en vrienden,

Een paar maanden geleden bracht een delegatie van de Open Universiteit een werkbezoek aan de Fernuniversität Hagen. Vrijwel meteen na aankomst stelde een medewerker van de ontvangende organisatie mij de vraag 'Are you a computer scientist?' Ik raakte door deze vraag enigszins in verwarring, niet zozeer omdat de voertaal ten oosten van Heerlen Engels bleek te zijn, maar vooral omdat een eenvoudig antwoord in geen enkele taal te geven is. Wat ik wel zeker weet, is dat ik een wiskundige ben. Bij onzekerheid dienaangaande valt het na te lezen op verschillende diploma's die ik desgewenst kan laten zien. Maar of ik, als wiskundige met belangstelling voor het totstandkomen van computerprogrammatuur, ook een *computer scientist* mag heten, hangt sterk af van de vraag wat men met die term wel zou kunnen bedoelen.

Ik kan u namelijk verklappen dat er helemaal niet een wetenschap met de naam *computer science* bestaat, althans niet in de zin van een kennisgebied dat door de coherentie van de gebruikte onderzoeksmethoden is te onderscheiden. In de tijdschriften en conferenties die aan het onderwerp worden gewijd, worden allerlei problemen die een relatie met computers hebben besproken en benaderd met methoden die ontleend zijn aan psychologie, sociologie, wiskunde, elektrotechniek, bedrijfskunde en nog wel andere disciplines. Wanneer men de vrije keuze heeft tussen het afnemen van een enquête, het bouwen van een prototype, het schrijven van een formeel bewijs of participerende observatie, is er dan nog sprake van een enkele wetenschap? Is het werkelijk zinvoller de verschillende probleemgebieden die met de introductie van de computer samenhangen onder een enkele benaming samen te nemen dan het een eeuw geleden geweest zou zijn de sociale, literaire, technische en organisatorische aspecten van de op-

komst van het telefoneren te bestuderen in het kader van een faculteit telefoonkunde?

In wat volgt zal ik dus bewust een wat eenzijdig beeld van de taak van de informatica schetsen, namelijk het perspectief zoals dat wordt waargenomen vanuit mijn eigen wiskundige achtergrond. De elektrotechnicus zal aandacht voor de onderliggende flip-flops missen, de informatiekundige zal zich afvragen of de exacte specificaties vanwaaruit ik meen te kunnen vertrekken wel overeenkomen met de ware behoeften der gebruikers. Ik wil niet beweren dat zulke kwesties niet belangrijk of belangwekkend zijn, maar wel dat ik er niets over mee te delen heb. Ik heb nu eenmaal alleen een hamer, en zal me dus beperken tot de wereld voor zover die uit spijkers bestaat.

Doelstelling

Softwareconstructie

Computerprogrammatuur vormt het ingewikkeldste product dat de mens tot op heden heeft uitgevonden. Niet alleen ingewikkeld door de grote omvang en mate van detaillering van de programmacode, maar vooral door de onoverzichtelijke veelheid aan scenario's die zich kunnen afspelen als de code wordt uitgevoerd. Vanzelfsprekend maakt dit de productie van foutloze programmatuur een uitermate lastige opgave, waarvoor maar zeer ten dele richtlijnen te geven zijn. De wijze waarop programma's in de praktijk tot stand komen, is dan ook weinig systematisch. In een klassieke beschrijving van Weizenbaum [43]:

The compulsive programmer spends all the time he can working on one of his big projects. 'Working' is not the word he uses; he calls what he does 'hacking.' To hack is, according to the dictionary, 'to cut irregularly, without skill or definite purpose; to mangle by or as if by repeated strokes of a cutting instrument.' I have already said that the compulsive programmer, or hacker as he calls himself, is usually a superb technician. It seems therefore that he is not 'without skill' as the definition would have it. But the definition fits in the deeper sense that the hacker is 'without definite purpose': he cannot set before himself a clearly defined long-term goal and a plan for achieving it, for he has only technique, not knowledge. He has nothing he can analyze or synthesize; in short, he has nothing to form theories about. His skill is therefore aimless, even disembodied. It is simply not connected with anything other than the instrument on which it may be exercised. His skill is like that of a monastic copyist who, though illiterate, is a first-rate calligrapher.

De geesteshouding die hier wordt beschreven leren we uit de eerste hand kennen in een literair werk (het woord 'roman' is niet geheel van toepassing) van Gerrit Krol [23]:

In twee dagen had ik mijn nieuwe programma klaar [...] Bosma moest weten hoe het werkte. 'Heel eenvoudig' zei ik. Ik legde het hem uit, met mijn handen tekeningen in de lucht makend. Hij begreep het niet. Ik ook niet. Ik had met de machine gepraat en ik kon het niet vertalen. Ik nam Bosma tenslotte naar de machine, liet het programma snorren, de beeldbuizen flikkerden en daarin kon hij zien gebeuren wat ik hem niet meedelen kon.

De missie van de informatica als wetenschap zou, denk ik, moeten zijn deze sterk intuïtieve benadering te vervangen door een systematische, controleerbare en overdraagbare werkwijze. Een ingenieur die een brug ontwerpt doet dat niet door koortsachtig net zo lang balken aan elkaar te nagelen totdat bij hem het gevoel ontstaat dat het bouwsel wel een redelijk gewicht zou kunnen dragen. Nee, hij maakt vooraf een berekening op grond van door meting bekende eigenschappen van de materialen waaruit blijkt welke vorm en afmetingen aan de gewenste eigenschappen voldoen. De wens een vergelijkbare betrouwbaarheid aan de totstandkoming van programmatuur te kunnen hechten, is de oorsprong van de term *software engineering*.

Een specifieke technische benadering van dit probleem bestaat erin dat men de taak van elke te ontwikkelen module vastlegt door middel van logische formules en daaruit door stapsgewijze transformatie volgens bekende regels een programmatekst afleidt. Deze handelwijze heeft in verschillende varianten bekend gestaan als *structured programming* [7], *stepwise refinement* [44] en *program derivation* [10]. Zelf heb ik, misschien in navolging van Meyer [32], een voorkeur voor de term *softwareconstructie*, vooral vanwege de gevoelswaarde van het woord constructie, dat enerzijds de stevigheid van gewapend beton oproept en anderzijds doet denken aan de procedurele nauwkeurigheid van de constructies met passer en liniaal uit de vlakke meetkunde.

Afstandsonderwijs

Een typerende eigenschap van materialen voor afstandsonderwijs is dat men bij de uitleg van begrippen en oplossingsmethoden in extreme mate

expliciet moet zijn. Immers, de stof wordt door de student tot zich genomen zonder dat de docent daarbij aanwezig is en optredende misverstanden meteen kan corrigeren. Het zal duidelijk zijn dat er bij een programmeerstijl die zich vooral op inspiratie en intuïtie beroept bijzonder weinig expliciete aanwijzingen kunnen worden gegeven. In veel informaticaopleidingen is het formele deel van het onderwijs dan ook beperkt tot de syntactische kennis van programmeertalen, terwijl het eigenlijke programmeren in de vorm van praktische oefeningen wordt getraind zonder dat de lerende zich daarbij erg bewust is van wat hij doet en waarom. De bij deze training nodige feedback vereist het besteden van veel tijd per deelnemende student en is dus moeilijk schaalbaar naar grote aantallen deelnemers, terwijl de noodzaak tot synchronisatie tussen student en docent de vrijheid van tijd en tempo sterk beperkt. Naarmate er dus meer expliciet beschreven methodiek beschikbaar is, wordt het gemakkelijker de kennis daarvan in afstandsonderwijs over te dragen.

Knuth [21] heeft¹ voorgesteld de overwegingen die worden doorlopen tijdens de constructie van programmatuur volledig expliciet te maken en de uiteenzetting daarvan voorrang te geven boven het instrueren van de machine.

I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be *works of literature*. [...] Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.

Een uitleg zoals hier bedoeld is essentieel om programmeeronderwijs op afstand mogelijk te maken. De prominente rol van niet geheel bewuste processen is de reden dat afstandsonderwijs niet succesvol is als methode om te leren fietsen of zwemmen – aan te tonen is derhalve dat de kunst van het programmeren niet in deze categorie valt en meer gemeen heeft met de systematische procedures der exacte wetenschappen.

¹In een artikel waarin hij stelt dat het woord *web* een van de weinige drieletterige woorden in het Engels is die geen enkele relatie tot de informatica hebben. Dat dateert het artikel wel – het is van 1984.

Invarianten

Het kernbegrip in de methoden van softwareconstructie die zojuist zijn aangehaald is het begrip *invariant*. Dit stelt ons in staat het onvoorstelbaar grote aantal mogelijke executies van een programma buiten beschouwing te laten door ons te concentreren op de eigenschappen die al deze paden met elkaar gemeen hebben. Ik zal nu drie voorbeelden geven van schijnbaar lastige problemen waarbij de moeilijkheid door het introduceren van een geschikte invariant als sneeuw voor de zon verdwijnt.

Het spel en de knikkers

Beschouw eens het geval van de pot die zowel rode als blauwe knikkers bevat², een voorwerp dat naar verluidt vooral bij stochastici op de schoorsteenmantel wordt aangetroffen. Om precies te zijn bevat deze pot 2841 rode en 2690 blauwe knikkers. Wij spelen nu het volgende spel: pak blindelings twee willekeurige knikkers uit de pot. Als ze dezelfde kleur hebben, stop dan een nieuwe blauwe knikker in de pot. We nemen aan dat we ergens een voldoende groot reservoir van blauwe knikkers bij de hand hebben om dit mogelijk te maken. Hebben de twee knikkers een verschillende kleur, stop dan de rode terug in de pot. Herhaal dit zolang de pot nog ten minste twee knikkers bevat.

Ik geef onmiddellijk toe dat er boeiender bezigheden denkbaar zijn om de lange winteravonden te vullen, maar er rijzen wel enige interessante vragen. In de eerste plaats: eindigt dit spel op den duur? Het spel eindigt als het niet meer mogelijk is twee knikkers uit de pot te nemen, dus als het aantal knikkers tot 0 of 1 is teruggelopen. In feite is het heel eenvoudig te zien dat die toestand ooit wordt bereikt, omdat in elke iteratie twee knik-

²Dit voorbeeld verscheen voor het eerst in druk bij David Gries [16], die het aan Carel Scholten toeschrijft.

kers worden verwijderd en één teruggeplaatst, zodat het aantal knikkers in de pot met 1 daalt. We zien dus ook meteen dat de toestand waarin het spel eindigt die is waarin nog slechts één enkele knikker is overgebleven. En dan nu de hamvraag: welke kleur heeft die laatste knikker?

Het probleem is hier dat het aantal mogelijke manieren waarop het spel kan verlopen nogal groot is, een probleem dat sterk lijkt op het onoverzichtelijk grote aantal executiepaden van een computerprogramma, en dat in de informatica als de *combinatorische explosie* bekend staat. Immers, meteen al bij de eerste greep in de pot zijn er drie mogelijkheden: we pakken twee blauwe knikkers, twee rode of een van elk. Afhankelijk van wat er gepakt is wordt een rode of blauwe knikker teruggeplaatst, dus na afloop van de eerste zet kan het universum in drie verschillende toestanden verkeren, en in elk van die drie toestanden moet het vervolg van de gebeurtenissen afzonderlijk worden geanalyseerd. Omdat het hele spel precies $2841 + 2690 - 1 = 5530$ stappen duurt, is dit zacht gezegd onbegonnen werk.

Het beeld klaart echter aanzienlijk op als we ons concentreren op wat er gebeurt met het aantal rode knikkers in de pot. In elke stap waarin twee rode knikkers worden gepakt, wordt een blauwe teruggeplaatst en daalt het aantal rode knikkers dus netto met 2. In elke stap waarin twee blauwe knikkers worden gepakt, wordt ook een blauwe teruggeplaatst en verandert het aantal rode knikkers dus niet. En in elke stap waarin twee knikkers van een verschillende kleur worden gekozen, wordt de rode van die twee teruggeplaatst en verandert het aantal rode knikkers dus evenmin. Conclusie: het aantal rode knikkers kan alleen dalen met 2 tegelijk en behoudt dus zijn pariteit, zijn even- of onevenheid. Omdat in het begin de pot een oneven aantal rode knikkers bevatte (namelijk 2841), respecteert dit spel de invariant

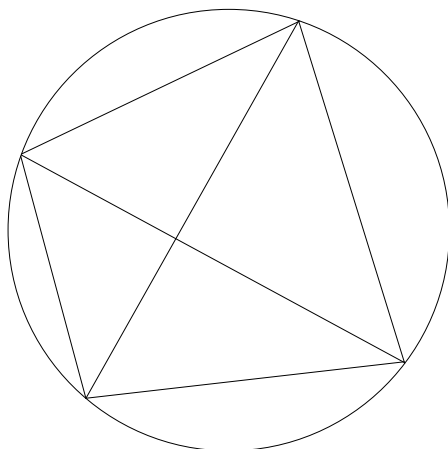
Het aantal rode knikkers in de pot is oneven.

In het bijzonder volgt hieruit dat het aantal rode knikkers niet 0 kan worden. In de eindtoestand, waarin de pot nog slechts één knikker bevat, is dat dus noodzakelijk een rode.

Pizza

Op de rand van een cirkelschijf kiezen we 15 punten en we trekken alle verbindende koorden – we doen dat zó dat er geen drie koorden door één punt

gaan³. De vraag is in hoeveel stukken de cirkelschijf dan wordt verdeeld.



In de praktijk zie je vaak dat wetenschappers, die toch beter zouden moeten weten, een probleem menen op te lossen door een paar eenvoudige gevallen te bestuderen en op basis daarvan een uitspraak met algemene geldigheid te formuleren. In het geval van dit voorbeeld gaat dat echter grandioos mis, kijk maar. Door voor kleine aantallen randpunten het experiment in feite uit te voeren en de ontstane stukken te tellen, komen we op een suggestieve tabel.

randpunten	stukken
1	1
2	2
3	4
4	8
5	16

Het vergt nu niet veel fantasie om het vermoeden te formuleren dat bij n punten op de rand van de cirkel er blijkbaar 2^{n-1} stukken ontstaan, dus bij 15 randpunten kunnen we rekenen op $2^{14} = 16384$ stukken. Probleem opgelost, zou je zeggen.

³Afkomstig uit EWD979 [11], waarin de oplossing wordt toegeschreven aan Aart Blokhuys.

Maar als we wat voorzichtiger geweest waren en ook nog het geval met 6 randpunten hadden nageteld, hadden we ontdekt dat daar niet de verwachte 32 stukken ontstaan, maar slechts 31. Een groot raadsel! Beoefenaars van een continue wetenschap kunnen in zo'n geval de schuld geven aan ruis of benaderingsfouten, maar in een discrete wetenschap als combinatoriek of informatica is die uitvlucht niet geldig. Het is echt 31 en geen 32. Laten we het dus over een andere boeg gooien en op een wat algemenere manier proberen te zien wat er gebeurt als we een koorde trekken.

Met k geven we het aantal koorden dat al getekend is weer. Dan is het duidelijk wat er met k gebeurt als we nog een koorde trekken: de waarde wordt 1 hoger. In de informatica schrijft men dat wel als

$$k := k + 1$$

Geef nu met s het aantal stukken aan waarin de cirkelschijf is verdeeld. De interessante vraag is wat de waarde van s doet als er een koorde bijkomt. Die waarde wordt zeker groter: als de nieuwe koorde onderweg geen andere snijdt, wordt s met 1 verhoogd. Maar als de nieuwe koorde onderweg een andere snijdt, worden beide stukken aan weerszijden van die andere doorsneden, zodat het aantal stukken nog extra toeneemt. Er geldt

$$s := s + x + 1$$

waar x het aantal bestaande koorden is dat door de nieuwe wordt gesneden. Op zichzelf hebben we daar nog niet veel aan, omdat we geen idee hebben hoe groot x is.

Maar kijk nu eens naar nog een coördinaat van het systeem, het aantal inwendige snijpunten van koorden. Als we dat i noemen, wordt de verandering ervan beschreven door

$$i := i + x$$

Kortom, s neemt evenveel toe als $k + i$, dus $s - k - i$ is constant. Initieel is $k = 0$, $i = 0$, $s = 1$. Het tekenen van koorden respecteert dus de invariant

$$s = k + i + 1$$

Laten we nu terug gaan naar de oorspronkelijke opgave: alle koorden die 15 randpunten verbinden. Elk van die koorden is bepaald door twee randpunten, dus het aantal koorden is $\binom{15}{2} = 105$. Een inwendig snijpunt

wordt bepaald door vier randpunten, want het is het snijpunt van de diagonalen van de daardoor bepaalde vierhoek. Dat zijn er dus $\binom{15}{4} = 1365$. Dankzij de invariant zien we in dat het aantal stukken waarin de schijf verdeeld wordt gelijk is aan $105 + 1365 + 1 = 1471$.

Denk zwart-wit

We denken ons het platte vlak betegeld met eenheidsvierkanten, als een oneindig groot schaakbord⁴. Maar de kleurenverdeling is anders dan die bij een schaakbord: initieel zijn er precies negen zwarte tegels, alle andere zijn wit.

Die kleuring gaan we nu als volgt veranderen: we kiezen een witte tegel met ten minste twee zwarte burens, en kleuren die zwart. Dit herhalen we zolang er zulke tegels te vinden zijn. De vraag is nu: is het mogelijk dat het zwarte gebied op een gegeven moment bestaat uit een vierkant met zijde 10? Tussen twee haakjes: als die toestand zich ooit voordoet, is het spel meteen afgelopen, want dan heeft een witte tegel één zwarte buur als die aan het zwarte gebied grenst, en geen enkele anders.

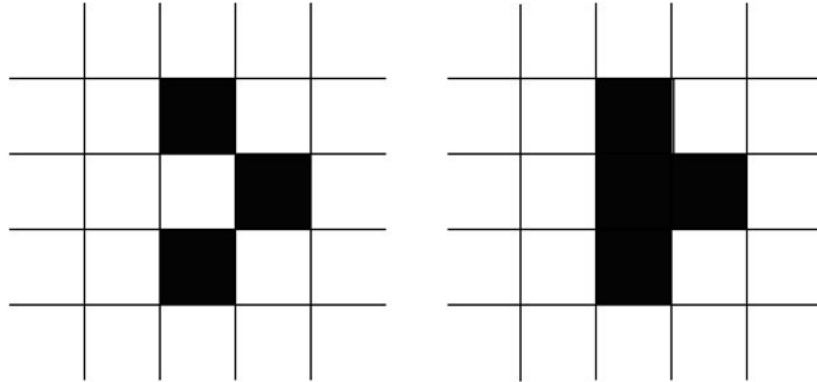
Het is duidelijk dat we dit probleem niet kunnen oplossen door alle manieren te inventariseren waarop de negen zwarte tegels oorspronkelijk in het platte vlak lagen – dat zijn er oneindig veel.

Het probleem wordt pas verhelderd als we ons concentreren op de *omtrek* van het zwarte gebied, dat wil zeggen het aantal zijden met aan de ene kant een witte en aan de andere kant een zwarte tegel. Laten we zo'n zijde voor het gemak even een *grens* noemen. Elke grens is een zijde van een zwarte tegel, en er zijn aan het begin negen zwarte tegels met elk vier zijden, dus het aantal grenzen is initieel ten hoogste 36.

Wat gebeurt er nu als we een tegel zwart kleuren? Bekijk de figuur die wordt gevormd door die tegel met zijn vier burens, waarvan er volgens de spelregels ten minste twee zwart zijn. Als we de centrale tegel zwart kleuren, verdwijnen er grenzen bij zijn zwarte burens, maar er komen nieuwe grenzen bij zijn witte burens.

Als alle vier de burens zwart zijn, gaan er bij het zwart kleuren van de centrale tegel vier grenzen verloren en er ontstaat geen nieuwe, zodat het aantal grenzen netto met 4 daalt.

⁴Dit voorbeeld staat in EWD1212 [12], waar het wordt toegeschreven aan Richard Bird.



Als drie van de burens zwart zijn, gaan er bij het zwart kleuren van de centrale tegel drie grenzen verloren, maar er ontstaat ook een nieuwe, grenzend aan de witte buur. Het aantal grenzen daalt dan dus netto met 2.

Als twee van de burens zwart zijn, gaan er twee grenzen verloren en ontstaan er twee nieuwe, zodat het aantal grenzen in dat geval niet verandert. Het spel respecteert derhalve de invariant

De lengte van de omtrek van het zwarte gebied is ten hoogste 36.

Kortom, in geen van de gevallen neemt het aantal grenzen toe, dus de omtrek, die initieel hooguit 36 lang was, wordt op den duur alleen maar korter. Kunnen we dus langs deze weg een vierkant met zijde 10 zwart kleuren? Nee, want de omtrek daarvan heeft lengte 40, een onbereikbaar hoge waarde.

Objecten

Object-oriëntatie

Niet alleen de toestandsruimte die tijdens de uitvoering van een programma wordt doorlopen is erg groot, in moderne systemen is ook de programmatekst als artefact enorm. De eerste pc die ik in huis had, ruim 25 jaar geleden, bezat geen harde schijf maar slechts twee diskettstations voor 5,25 inch floppy disks met in de uitvoering 'Double Side, Double Density' een capaciteit van 360 kB. Het operating system moest eerst worden geladen, en daarna diende station A: om de programmatuur te herbergen en B: voor de geproduceerde eigen bestanden. En dat paste! Ter vergelijking: het mapje met de programmatuur voor Microsoft Office is op mijn computer 736 MB groot, dat scheelt dus ongeveer een factor 2000. Dat is niet eens indrukwekkend in vergelijking met de ontwikkeling die de hardware heeft doorgemaakt (in mijn studententijd was een computer een zaal vol stalen kasten waar je tussendoor kon lopen, nu past dezelfde rekencapaciteit in je binnenzak), maar het neemt niet weg dat ook software enorm veel groter en complexer is geworden.

Alle programmatuur moet worden onderhouden. Niet dat deze aan slijtage onderhevig is: onderhoud slaat bij software op de aanpassing aan nieuwe externe omstandigheden (andere hardware, andere connectiviteit) en op het verwijderen van na ingebruikname alsnog ontdekte fouten. Programmatuur die zo groot is, kan niet worden onderhouden als daarin niet een zekere substructuur wordt aangebracht die enerzijds aanwijzingen geeft in welk deel van de programmacode gewenste functionaliteit kan worden gezocht, anderzijds het mogelijk maakt lokaal wijzigingen aan te brengen zonder dat dit tot een cascade van andere noodzakelijke aanpassingen in ver-verwijderde delen van de code leidt.

De dominante wijze om programmacode van een structuur te voorzien,

is op dit moment *object-oriëntatie*, een principe dat ten grondslag ligt aan veelgebruikte programmeertalen zoals Java en C#. Binnen dit paradigma kan de uitvoering van een programma worden gezien als de interactie tussen een groot aantal entiteiten, objecten genaamd. Een object heeft een eigen identiteit en beheert een zekere hoeveelheid informatie, vastgelegd in de waarde van *attributen*. De waarde van de attributen wijzigt tijdens de uitvoering, zodat we van de toestand van het object op elk moment kunnen spreken. De vele dimensies die de toestandruimte van het totale programma opspannen kunnen zo worden verdeeld over de veel beperktere toestandruimten van de afzonderlijke objecten. Dat scheelt meteen enorm aan complexiteit, als tenminste de afzonderlijke objecten in relatief isolement kunnen worden geanalyseerd en ontworpen – een cruciaal voorbehoud, zoals we zullen zien.

Bovendien is de wijze waarop de toestand van de objecten kan veranderen aan strenge restricties onderhevig. Elk object – meer nauwkeurig gezegd elke klasse, waarbij een klasse het type van objecten met identiek gedrag uitdrukt – beschikt over een aantal methoden, uitvoerbare stukken code, die samen de interface van het object naar de buitenwereld vormen. Uitvoering van een methode is volgens dit programmeermodel de enige manier waarop de toestand van een object kan veranderen en de enige manier waarop informatie over de toestand van het object kan worden opgevraagd. Rechtstreeks bekijken en wijzigen van de inhoud van een object is niet de bedoeling: de objecten zijn hermetisch gesloten zwarte dozen met een beperkt aantal draaiknoppen en wijzerplaten. Uitvoering van het programma komt er nu op neer dat er objecten worden gecreëerd die elkaar vervolgens vragen stellen en opdrachten geven.

De reden dat dit zo handig is, is dat we bij het ontwerpen van programma's de objecten zo kunnen kiezen dat de object-interactie een simulatie vormt van een of andere situatie in de werkelijkheid die door de software gemodelleerd wordt. De objecten corresponderen met 'dingen', fysiek of conceptueel, die wij in de werkelijkheid of in onze gedachten daarover waarnemen, en de klassen dus met 'soorten dingen'. In de programmacode voor een tekstverwerker zijn woord, alinea en lettertype voor de hand liggende klassen. De methoden van die klassen stemmen overeen met gebeurtenissen die zich tijdens de uitvoering van het programma voordoen: zo kan een woord worden verwijderd, cursief gezet of verplaatst, en al die acties zijn goede kandidaten om als methode aan deze klasse te verbinden.

Het wekt dan ook geen verbazing dat het object-georiënteerde para-

digma voor het eerst werd gebruikt in SIMULA 67 [8], een programmeertaal die uitdrukkelijk voor het uitvoeren van simulaties was bedoeld.

Klasse-invarianten

Het begrip klasse-invariant is geïntroduceerd door Meyer [31], als verdere ontwikkeling van het werk van Hoare over invarianten van datatypen [17], en vormt de kern van het begrip 'Design by Contract'. In deze discipline is elke operatie voorzien van een contract, dat vastlegt in welke omstandigheden de operatie zinvol kan worden uitgevoerd (de preconditionie), en garandeert op welke resultaten dan na afloop mag worden gerekend (de postconditie). Een klasse-invariant is een relatie tussen de waarden van de attributen van de objecten die tot de klasse behoren. Anders dan de naam wellicht suggereert, behoeft een klasse-invariant niet op elk moment tijdens de executie van een klasse te gelden. Operaties die een bepaald effect willen bewerkstelligen, zullen als neveneffect daarvan mogelijk een schending van de invariant teweegbrengen. Om Meyer [32] te citeren:

As in human affairs, trying to do something useful may disrupt the established order of things.

In zulke gevallen moet de schadelijke actie echter altijd worden gevolgd door opruimwerkzaamheden die de klasse-invariant herstellen; pas nadat dit herstel is voltooid, kan de operatie worden beëindigd.

Meer formeel kan Meyers regel voor een klasse-invariant P bij een klasse C als volgt worden samengevat [32, blz. 366]:

- Elke constructor van C , toegepast op argumenten die aan de preconditionie voldoen, levert een object op dat in een toestand verkeert die aan P voldoet,
- Elke publieke operatie van C , toegepast op argumenten en in een toestand die aan zowel P als de preconditionie voldoen, eindigt in een toestand die weer aan P voldoet.

In grote lijnen kunnen we zeggen dat een klasse-invariant dus een bewerking is die aan zowel de preconditionie als de postconditie van elke publieke operatie van de klasse wordt toegevoegd. Eigenlijk is de eis nog iets sterker, want dit geldt niet alleen voor de operaties van de bestaande

klasse, maar ook voor in de toekomst te ontwikkelen operaties van nu nog onvoorziene extensies.

Zulke klasse-invarianten kunnen dus dienen als gids bij het ontwikkelen van de software, en voorkomen dat bij de constructie van nieuwe operaties veronderstellingen worden gemaakt over de toestand van objecten die in feite niet vervuld zijn. Velen zijn er ook voorstander van tijdens de uitvoering van een programma op alle passende momenten de juistheid van invarianten en andere asserties te blijven controleren. Bij het testen van nieuwe software is dit vrij gebruikelijk, bij software die in gebruik is genomen minder, om redenen van snelheid. Maar, in de woorden van Tony Hoare [18]:

It is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea?

Veel programmeertalen zijn tegenwoordig voorzien van faciliteiten die het controleren van asserties tijdens de programma-uitvoering faciliteren. De door Meyer ontwikkelde programmeertaal Eiffel [32] beschikte van meet af aan over constructies als `require` (voor precondities), `ensure` (voor postcondities) en `invariant` (voor klasse-invarianten). Maar ook een populaire taal als Java beschikt tegenwoordig, bij nader inzien, over een `assert`-constructie waarmee, zij het met meer syntactische moeite, voor controle tijdens de executie kan worden gezorgd. Ook zijn er aanvullingen op de taal, zoals Contract4J [22], die de ondersteuning uitbreiden tot het niveau dat Eiffel van meet af aan bezat.

Niettemin zijn er aan het begrip klasse-invariant verschillende moeilijkheden verbonden. We zijn nu aangekomen bij de kern van mijn betoog. Ik zal u aan de hand van een aantal voorbeelden demonstreren waar deze moeilijkheden optreden en vervolgens welke oplossingen daarvoor in de loop van de afgelopen tien jaar zijn gesuggereerd.

Abstractie

Een fundamenteel probleem bij het karakteriseren van precondities, postcondities en invarianten door middel van de waarden van object-attributen

is dat daarmee de *encapsulatie* wordt doorbroken. Met dit begrip bedoelt men te zeggen dat de interne structuur en werking van objecten voor de buitenwereld onzichtbaar moet zijn, onder andere omdat dit het mogelijk maakt die werking in een later stadium nog te wijzigen zonder dat dit voor code buiten de object-implementatie enig gevolg heeft. Zo zullen lokale verbeteringen in een groot softwaresysteem mogelijk blijven zonder dat de repercussies aanpassingen van verafgelegen onderdelen van de programmatuur nodig maakt.

Als echter de gebruiker van een klasse moet afgaan op specificaties die precies vertellen wat het effect van methoden op de attributen zijn, wordt daarmee ook voor de buitenwereld vastgelegd welke attributen er eigenlijk zijn. De oplossing voor dit probleem is de klasse niet te specificeren in termen van attribuutwaarden, maar in termen van een abstract *model*. De operaties op een klasse waarvan de objecten collecties dingen voorstellen, kunnen bijvoorbeeld worden gespecificeerd met behulp van het begrip *verzameling* uit de wiskunde, ongeacht de vraag of de verzamelingen intern zijn gerepresenteerd als binaire boom, bitarray of hashtable.

Natuurlijk moet wel ergens worden vastgelegd wat het verband is tussen het model en de interne implementatie. De traditionele visie [17, 29] is het gebruikmaken van een *abstractiefunctie* die de toestand van de implementatie afbeeldt op een toestand van het abstracte model. Maar omdat er voorbeelden zijn te bedenken [15] waar in het model gemaakt onderscheid irrelevant is voor de implementatie en dus slechts als gedachtevariabele een rol speelt, is het algemener te spreken van een *implementatie-invariant* [4]. Het punt is hier dat die implementatie-invariant alleen lokaal nodig is om te bewijzen dat de interne implementatie van de klasse correct is als representatie van de abstractie, een taak voor de implementator van de klasse waar de cliënten zich geen zorgen over hoeven te maken.

Een andere benadering wordt gevolgd door Leino en Müller [28], die wijzigingen in de toestand van de abstractie niet automatisch laten plaatsvinden als dat door de implementatie-invariant uit een toestandswijziging van de implementatie voortvloeit, maar dit effect toewijzen aan een expliciete opdracht *pack*, waarvan de programmeur het tijdstip van uitvoeren moet bepalen. Het nadeel is dat hier een extra last op de programmeur wordt gelegd, het voordeel dat ook tussentoestanden mogelijk worden die niet met een mogelijke waarde van de abstractie corresponderen.

Problemen

The Dependent Delegate Dilemma

De eis dat een invariant op elk willekeurig moment moet gelden, zal in het algemeen ondoenlijk zwaar zijn. Allerlei operaties kunnen de invariant verstoren en het is de taak van de programmeur ervoor te zorgen dat die is hersteld op het eerstvolgende moment dat de geldigheid ervan weer nodig is omdat andere onderdelen van de software daarop vertrouwen.

In het oorspronkelijke idee van de klasse-invariant wordt de invariant geacht te gelden als het object voor de buitenwereld benaderbaar is, dat wil zeggen gereed om op verzoek van een cliënt een van zijn methoden uit te voeren. In toestanden waar het object al bezig is met de uitvoering van een methode, is het niet bereid verzoeken uit de buitenwereld te beantwoorden en dus kan het geen kwaad als het tijdelijk de klasse-invariant schendt: de tijdelijk ongeoorloofde toestand is niet openbaar zichtbaar. Wie er bezwaar tegen heeft zich aan de wereld te vertonen zonder stropdas, boerka of contactlenzen kan deze attributen in de beslotenheid van het eigen huis zonder bezwaar verwijderen, zolang ze maar weer ijlings worden teruggeplaatst op het moment dat de bel gaat.

Het Dependent Delegate Dilemma – een klinkende naam die door Bertrand Meyer [33] is voorgesteld voor wat eerder [40] *re-entrancy* werd genoemd – treedt op als bij de acties die een object onderneemt om een verstoorte invariant te herstellen gebruik gemaakt wordt van de diensten van hetzelfde object of een ander object dat, rechtstreeks of indirect, in een cliëntverhouding tot het oorspronkelijke object staat.

Om te voorkomen dat mijn verhaal door de voortdurende verwijzing naar het oorspronkelijke object en het daardoor te hulp geroepen tweede object te ondoordringelijk wordt, wil ik een concreet voorbeeld geven. Dit is typisch wat men een speelgoedvoorbeeld noemt, omdat het probleem

tot de allereenvoudigste vorm is geabstraheerd en hier zo onverhuld naar voren komt dat niemand zich zal kunnen voorstellen dat hier werkelijk iets fout gaat. In de praktijk zijn de situaties veel ingewikkelder: Szyperski [41] geeft een voorbeeld dat zich afspeelt in een realistisch tekstverwerkings-systeem; de uitleg neemt vele bladzijden in beslag.

In ons simpele voorbeeld gaat het om een object dat een lijst $a[0..)$ van logische waarden en een natuurlijk getal n bijhoudt, met als klasse-invariant

$$P: (\forall i: 0 \leq i \leq n: a[i])$$

De klasse heeft een methode $set(int\ k)$ met postconditie $a[k]$. Als in een andere methode nu n met 1 verhoogd wordt, geldt P niet meer omdat $a[n]$ mogelijk nog niet de juiste waarde (namelijk $true$) heeft. Dat kunnen we oplossen door vervolgens $set(n)$ aan te roepen.

Als de implementatie van $set(int\ k)$, zoals te verwachten is, bestaat uit de toekenning $a[k] := true$, gaat dit helemaal goed. Maar als we in plaats daarvan deze methode implementeren door $a[k] := a[n]$, wat op grond van P correct is, wordt het gewenste effect niet bereikt. Dat komt omdat de aanroep van $set(n)$ plaatsvindt in een toestand waarin de invariant verstoord is, zodat het niet verantwoord is P te gebruiken.

In dit voorbeeld treedt het misverstand helemaal op binnen een enkel object en kan het door de programmeur van de desbetreffende klasse dus gemakkelijk worden opgespoord. In de praktijk worden om de invariant te herstellen echter vaak methoden van andere klassen aangeroepen die, na een lange keten van aanroepen, uiteindelijk terugkeren bij het oorspronkelijke object en dat onbedoeld aantreffen in een toestand waarin de invariant verstoord is. Omdat de vorm van dergelijke ketens van aanroepen afhangt van de dynamische waarde van objectreferenties, is in het algemeen niet op grond van simpele inspectie van de programmatekst vast te stellen of dit gevaar werkelijk aanwezig is.

Het Indirecte Invariant Effect

Het Indirecte Invariant Effect treedt op als in de invariant van een klasse attributen van een ander object dan `this` worden genoemd. Het probleem wordt veroorzaakt doordat genoemde attributen buiten de operaties op `this` om gewijzigd kunnen worden. Met een iets minder klinkende naam

dan het vorige probleem wordt dit door Meyer [33] wel The Indelicate Delegate Problem genoemd.

Beschouw twee klassen *Persoon* en *Huis*, met daartussen de relatie dat een persoon bewoner kan zijn van een huis. Om dit weer te geven, nemen we in de klasse *Persoon* een attribuut *woning* van type *Huis* op. Dat in de harde werkelijkheid personen ook dakloos kunnen zijn, terwijl anderen over meer dan één woning beschikken, laten we in dit model maar even buiten beschouwing. Wel is het uitdrukkelijk mogelijk dat meer dan één persoon hetzelfde huis bewoont: in dat geval hebben de *woning*-attributen dezelfde waarde.

Hoewel een huis dus meer dan één bewoner kan tellen, kan er volgens de belastingwetgeving maar een enkele hoofdbewoner zijn, die verantwoordelijk is voor de betaling van diverse aan het huis gerelateerde lasten. Om dit te modelleren, geven we de klasse *Huis* een attribuut *hoofdbewoner*. We willen nu eisen dat elk huis dat bewoners heeft ook een hoofdbewoner heeft, die zelf een van die bewoners is. Dit kunnen we uitdrukken door de volgende klasse-invariant toe te voegen aan de klasse *Huis*:

$$(\exists p :: p.woning = \text{this}) \Rightarrow \text{hoofdbewoner}.woning = \text{this}$$

Uiteraard willen we de mogelijkheid behouden dat een huis een nieuwe hoofdbewoner krijgt, en daartoe een methode *overdraag*(*Persoon* p) toevoegen met postconditie *hoofdbewoner* = p. Een simpele toekenning

$$\text{hoofdbewoner} := p$$

bewerkstelt wel de postconditie, maar schendt mogelijk de invariant. We kunnen die herstellen door ervoor te zorgen dat

$$p.woning = \text{this} \tag{1}$$

gaat gelden. Het is dus blijkbaar nodig dat *Persoon* ook een methode *verhuis*(*Huis* h) krijgt met postconditie *woning* = h. Door nu in de implementatie van *overdraag*(p) een aanroep van *p.verhuis*(*this*) te plaatsen kunnen we inderdaad (1) bewerkstelligen.

Ogenschijnlijk is daarmee een goede implementatie van *overdraag*(p) gevonden, maar helaas blijkt deze niet correct te zijn. Immers, beschouw het scenario

```
h1.overdraag(p1);
h2.overdraag(p1);
p2.verhuis(h1);
```

Na afloop hiervan geldt $p2.woning = h1$ en $h1.hoofdbewoner.woning = h2$, zodat het object $h1$ niet aan de klasse-invariant voldoet. Dit komt doordat de operatie $p2.verhuis(h1)$, die plaatsvindt buiten de context van een *overdraag*-operatie, geen rekening houdt met de invariant van *Huis*.

Men zou kunnen zeggen dat het probleem met de klassieke klasse-invariant is dat aan objecten een strikt egoïstisch gedrag wordt toegeschreven: het object heeft de verplichting zijn eigen invariant aan het eind van de methode-executie te hebben hersteld, maar er wordt niet geëist dat het daarbij de invarianten van andere objecten moet respecteren.

Remedies

Versterking van het begrip invariant

De gedachte achter klasse-invarianten is dat deze de toestanden van het object karakteriseren die consistent zijn met de werkelijkheid die in het objectgeoriënteerde model wordt gemodelleerd. Toestanden waarin de invariant tijdelijk verstoord is, zijn corrupt of illegaal, hebben geen semantiek en mogen niet aan de buitenwereld worden gecommuniceerd.

Het is daarom niet voldoende te eisen dat de invariant is hersteld aan het eind van een operatie. Weliswaar is de observatie juist dat het object niet gereed is verzoeken van de buitenwereld in behandeling te nemen totdat de lopende operatie is voltooid, maar verzoeken van de buitenwereld zijn niet de enige gebeurtenissen die ertoe leiden dat code van buiten het object gaat worden uitgevoerd. Dit verschijnsel treedt ook op als het object in een poging de gevraagde postconditie te bereiken of de eigen invariant te herstellen daarbij een verzoek uitstuurt dat via een reeks van aanroepen leidt tot het starten van een nieuwe operatie op hetzelfde object. Die laatste zal volgens het geschetste bewijssysteem dan ten onrechte aannemen dat de invariant reeds hersteld is, waarmee het Dependent Delegate Dilemma zich voltrekt.

Een manier om dit probleem te vermijden, is eisen dat de invariant reeds hersteld is op het moment dat de controle uit handen wordt gegeven: met andere woorden, de invariant wordt als assertie toegevoegd niet alleen aan het begin en eind van elke implementatie, maar ook voorafgaand aan elke methode-aanroep in de code.

Of we na afloop van zo'n aanroep nog kunnen vertrouwen dat de al herstelde invariant niet opnieuw verstoord is, ligt aan de mogelijkheid van het optreden van het Indirecte Invariant Effect. Dit kan alleen worden uitgesloten als we in de bewijsverplichtingen niet alleen eisen dat de invariant van

het actieve object zelf hersteld moet worden, maar die eis ook stellen voor de invarianten van alle andere objecten die mogelijk door acties daarvan beïnvloed zouden kunnen zijn. In het voorbeeld van huizen en bewoners dat we eerder beschouwden is het dus niet genoeg dat de operaties op elk huis zijn eigen invariant herstellen, maar ze moeten dat ook doen voor alle andere huizen.

Als we die eis ook stellen, is er verder geen probleem: Huizing et al. [19] hebben bewezen dat een bewijssysteem waar de verplichtingen op de hier geschetste wijze versterkt zijn, inderdaad in logische zin *sound* is, dus niet tot ongerijmdheden kan leiden.

Het probleem met deze benadering is echter dat deze eisen draconisch zijn. Eisen dat de klasse-invarianten van alle relevante objecten hersteld moeten zijn voordat de eerste aanroep naar buiten plaatsvindt, miskent dat deze aanroepen gewoonlijk juist nodig zijn om het herstel van de invariant mogelijk te maken. In het huizenvoorbeeld moesten we $p.woning = \text{this}$ totstandbrengen, en daarvoor is een aanroep van een operatie op p werkelijk onontbeerlijk. Aan de versterkte invariantie-eis, hoeveel betrouwbaarheid die ook met zich meebrengt, is dus onmogelijk te voldoen. Je kunt ook te voorzichtig zijn.

Wat hier aan de hand is, is dat het probleem weggedefinieerd is doordat alle programma's die problemen kunnen opleveren buiten de orde zijn verklaard, alsof je de werkloosheidscijfers laat dalen door een engere definitie van werkloosheid te kiezen.

Verzwakking van het begrip invariant

Er blijkt dus dat de bewijsverplichtingen die nodig zijn om veilig op het gelden van klasse-invarianten te kunnen vertrouwen, onwerkbaar zwaar zijn. Een mogelijke uitweg is: niet meer de pretentie hebben dat op die geldigheid vertrouwd kan worden aan het begin van elke methode-implementatie. Als we dan nog wel de moeite nemen de invariant te herstellen aan het eind van de implementatie, kunnen we in elk geval nog concluderen dat de invariant geldt als het systeem geheel in rust is, dat wil zeggen geen operatie op enig object in uitvoering is.

Op zichzelf is zo'n eis niet onrealistisch. Beschouw bijvoorbeeld een spreadsheet – in correct maar ongebruikelijk Nederlands een rekenblad. Dan gelden daarin zekere sterke invarianten in de zin van de vorige paragraaf, zoals de eis dat de cellen een waarde van een bepaald type bezit-

ten en een formule waarmee een dergelijke waarde kan worden berekend. Maar aan de eis dat de getoonde waarde consistent is met de uitkomst van de formule is alleen voldaan als er op dat moment niet een berekening plaatsvindt. Typen we een waarde in een cel van het blad, dan beïnvloedt dat de waarde van alle cellen waarvan de formule naar deze cel verwijst. Die wijzigingen veroorzaken op hun beurt weer andere, en zo komt een cascade van opeenvolgende veranderingen op gang. Gedurende deze hele berekening zijn de waarden van sommige cellen inconsistent, en pas als de aanpassing geheel voltooid is, geldt de genoemde invariant weer. Dit is een typisch voorbeeld van een zwakke invariant, die alleen aan de postconditie van de operaties wordt toegevoegd en niet op andere plaatsen.

De Eindhovense werkgroep SOOP [40] gebruikte het begrip zwakke invariant om een compacte specificatie te geven van het ontwerppatroon Observer [14].

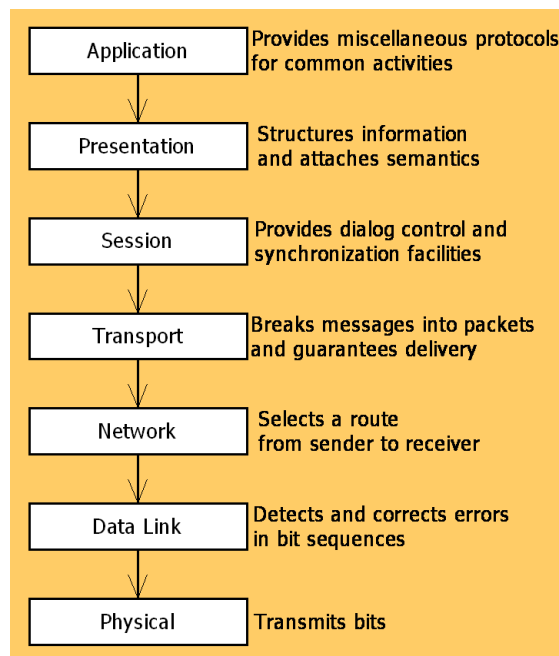
Meyer [33] stelt een iets ingewikkelder variant van deze benadering voor. In zijn visie dienen we methodes op twee verschillende wijzen te analyseren. Enerzijds moeten geëxporteerde methodes voldoen aan de oorspronkelijke eis van behoud van de klasse-invariant, maar daarnaast is een tweede analyse nodig waarbij de klasse-invariant wordt uitgeschakeld bij methode-aanroepen die afkomstig zijn uit het object `this` zelf of uit een object waarmee `this` een cliëntrelatie onderhoudt. In zulke gevallen zouden alleen de preconditione en de postconditie van de methode in de analyse moeten worden betrokken, en niet de klasse-invariant. Deze benadering leidt echter tot een probleem: ze heeft betrekking op de identiteit van afzonderlijke objecten, die door het effect van indirecte adressering dynamisch kan veranderen en niet goed vooraf te analyseren is. In dat geval kunnen we dus niet op grond van studie van de programmatekst vaststellen dat het programma correct zal werken en is er geen betere oplossing dan in de compiler faciliteiten in te bouwen die tijdens de executie alarm slaan als een klasse-invariant op een cruciaal moment geschonden blijkt. Sinds bij de eerste maanlanding, de expeditie van de Apollo-11, de boordcomputer op enige meters boven het maanoppervlak zo frequent foutmeldingen produceerde dat de laatste fase van de daling met handbesturing moest worden uitgevoerd [13], zijn programmeurs ervan doordrongen dat statische controles, die dus plaatsvinden bij het invoeren en vertalen van de programmatekst, verre te prefereren zijn boven de productie van run-time alarmeringen⁵.

⁵Gelukkig werd al voor de lancering ontdekt dat de zwaartekracht van de maan abusievelijk als afstotend in plaats van aantrekkend was geprogrammeerd [38].

Een zeer flexibele aanpak werd voorgesteld door Barnett et al. [2]: zij introduceren per object een expliciete publieke specificatievariabele die de waarde *Invalid* of *Valid* kan hebben en die in elke toestand aangeeft of de klasse-invariant daar wel of niet geldt. Aanroepen van methoden die het gelden van die invariant veronderstellen legt aan de cliënt de verplichting op de waarde van deze specificatievariabele te controleren. Rechtstreeks controleren van de invariant zelf is voor de cliënt in het algemeen niet mogelijk omdat daarin afgeschermden privévariabelen en willekeurige wiskundige expressies kunnen voorkomen.

Eigendom: 9/10 van de wet?

Müller et al. [37] geven een andere uitweg uit de moeilijkheid dat de versterkte invarianten van Huizing et al. [19] een te zware eis opleggen. Zij beschouwen een gelaagde architectuur [6, 9]; dat wil zeggen dat de klassen in het systeem zijn verdeeld in hiërarchisch geordende groepen, lagen genaamd. Daarbij geldt de regel dat vanuit elke laag alleen methoden van klassen in de onderliggende lagen mogen worden aangeroepen. Het ISO-OSI referentiemodel [42] is een klassiek voorbeeld.



In een gelaagde architectuur kan de benadering met sterke invarianten niet toegepast worden. Deze zou immers veronderstellen dat elk object, alvorens de controle uit handen te geven, de invarianten van alle mogelijk beïnvloede objecten heeft hersteld. Maar dat zou vereisen dat deze taak ook wordt verricht voor de hoger gelegen lagen, waartoe het object geen toegang heeft en waar het zelfs geen informatie over bezit. Het voorstel van Müller et al., dat voorkomt uit Müllers promotie-onderzoek aan de Fernuniversität Hagen [36], komt er nu op neer die laatste verplichting weg te nemen: een klasse is alleen verantwoordelijk voor het herstel van de invarianten in dezelfde laag. Immers, de aanroep van een methode uit een diepere laag kan door de beperkingen van de architectuur niet resulteren in terugkeer naar het oorspronkelijke object: cyclische ketens van aanroepen zijn uitgesloten en het Dependent Delegate Dilemma kan dan niet optreden.

Deze benadering is een voorbeeld van een toepassing van het eigendomsbegrip in objectgeoriënteerd ontwerp. Het idee is aan te nemen dat elk object ten hoogste één eigenaar heeft, waarbij de eigendomsrelatie een acyclische gerichte graaf vormt, als u mij het wiskundejargon excuseert. Een context is de verzameling van objecten met dezelfde eigenaar. Een object mag alleen verwijzingen bezitten naar objecten in dezelfde context of naar objecten waarvan het de eigenaar is.

Leino en Müller [27] gebruiken ook het eigendomsbegrip, maar passen dit toe in een algemenere context dan die van de gelaagde architectuur. Voortbouwend op de techniek van Barnett et al. [2], die al een expliciete publieke specificatievariabele introduceerden om het gelden van de klasse-invariant aan te geven, voegen zij een soortgelijke specificatievariabele toe die de eigenaar van het object weergeeft. Het grote voordeel tegenover de methode van Müller et al. [37] is dat deze eigendomsrelatie dynamisch kan variëren: objecten kunnen in de loop der tijd van eigenaar veranderen. Het nadeel is dat de specificaties een orde van grootte ingewikkelder worden en dus ook meer werk kosten om te implementeren en te gebruiken.

Er is nog een tweede nadeel: er zijn situaties waar een hiërarchie zoals bedoeld in het eigendomsbegrip redelijkerwijs niet verondersteld kan worden. Barnett en Naumann [3] bouwen voort op Leino en Müller [27], maar breiden die uit met een mechanisme waarbij een klasse aan een *bevriende* klasse het recht kan verlenen een invariant op te voeren die afhangt van attributen van de eerstgenoemde. Deze actie legt aan beide klassen beperkingen op: de bevriende klasse kan restricties formuleren waaraan de oorspronkelijke zich bij het wijzigen van attribuutwaarden moet houden,

maar is daartegenover verplicht alle instanties zich te laten aanmelden. Het resulterende bewijssysteem, ontwikkeld bij Microsoft, staat bekend als de *Boogie*-methodologie. Hoewel deze een veel grotere klasse programma's toelaat dan de eerder genoemde benaderingen, waaronder nu ook praktisch bruikbare voorbeelden, vormen deze nog steeds maar een deel van de programma's die men in de praktijk kan tegenkomen, en de formele verplichtingen nemen verder toe.

Middelkoop et al. [34, 35] verfijnen het onderscheid verder door specificaties uit te breiden met twee nieuwe constructies: *inc* (voor inconsistent) geeft aan dat een methode niet het gelden van zekere klasse-invarianten vereist, en *coop* dat toekenningen aan een attribuut zekere klasse-invarianten ongeldig kunnen maken. Daarmee wordt weer een grotere klasse van programma's verifieerbaar, ten koste van de verdere uitbreiding van de specificatie- en bewijsverplichtingen. Ook hiermee kan echter nog niet alles wat we zouden willen: in het voorbeeld van de spreadsheet hebben de cellen waarvan de invariant geschonden wordt een verwijzing naar de cel waarvan de waarde wijzigt, maar niet andersom.

Alternatief

Evaluatie

De opeenvolgende aanpassingen van de methodologie van klasse-invarianten hebben als doel deze bruikbaar te maken voor een ruimere collectie programma's. De gerealiseerde verbeteringen in deze richting zetten echter geleidelijk kleinere stappen in de richting van het gewenste resultaat, toepasbaarheid voor alle programma's die men in de praktijk zou willen schrijven, en doen dit ten koste van steeds toenemende formele verplichtingen en restricties. Daardoor vertoont deze onderzoekslijn veel van de eigenschappen van wat wetenschapsfilosoof Imre Lakatos, sprekend over empirische wetenschappen, een *degenererend onderzoekprogramma* heeft genoemd [25]:

I define a research programme as degenerating even if it anticipates novel facts but does so in a patched-up development rather than by a coherent, pre-planned positive heuristic.

In de informatica gaat het niet zozeer om het voorspellen van nieuwe waarnemingen als wel om het toepassen van een constructiemethodologie om op een praktisch hanteerbare wijze een grotere groep programma's te kunnen bouwen. Maar *mutatis mutandis* is de karakterisering hier toepasselijk. In de woorden van Parkinson [39]:

The class invariant can only reason about a single object. Using ownership based methodologies [...] we can extend the class invariant to some aggregate structures, and allow an object's invariant to depend on objects it owns completely. However, in more complex examples the ownership is less clear cut. Consider two collaborating classes: neither owns the other and each

has an invariant depending on the state of the other. Any update to one object will potentially invalidate the invariant of the other object. So how can we update this co-dependent structure? Ideas such as peer invariants [27], friends and update guards [3], and history properties [30], have been used to extend the idea of a class invariant, so that it can depend, soundly, on other objects. But is the complexity of these proposals a sign that the class invariant is not the correct foundation?

Het alternatieve voorstel van Parkinson [39] is om invarianten te definiëren over *aggregate structures*, bijvoorbeeld het *Observer*-patroon. Dit laat echter de vraag onbeantwoord wat precies telt als een dergelijke structuur, hoe we ze in de programmacode kunnen herkennen, en hoe we moeten omgaan met objecten die van meer dan een structuur deel uitmaken.

Aspecten

Een mogelijk antwoord is te vinden in de ontwikkeling van *aspect-georiënteerd* programmeren. Aspecten zijn programmeertaalconstructies die onafhankelijk van, en in zekere zin loodrecht op, de indeling van het programma in klassen bestaan. Een aspect houdt zich bezig met de implementatie van één bepaald kwaliteitskenmerk van het programma, door daaraan codefragmenten toe te voegen die in de juiste omstandigheden worden afgevuurd. Zo kan men bijvoorbeeld de beveiliging van een programma verbeteren door daaraan een aspect toe te voegen dat, telkens wanneer gevoelige informatie wordt benaderd, de identiteit en rechten van de gebruiker verifieert. Waar een object een representatie in de software is van een entiteit die daarin gemodelleerd wordt, correspondeert een aspect met een voorwaarde uit het program van eisen en dus uiteindelijk met een wens van de gebruiker. Het programmeren van aspecten vereist een speciale aspect-georiënteerde programmeertaal: het bekendste voorbeeld daarvan is AspectJ [20, 24], een uitbreiding van Java.

Lam, Kuncak en Rinard [26] stellen voor invarianten te hechten aan aspecten in plaats van aan klassen. Het voordeel daarvan is dat een aspect zich met een enkele welomschreven eigenschap bezighoudt en alle daarvoor relevante code bevat; een invariant heeft eveneens ten doel een bepaalde eigenschap te realiseren en past daarom in reikwijdte goed bij een aspect. Deze cohesie van aspecten draagt er ook aan bij dat zij kunnen worden ontworpen zonder van de diensten van andere aspecten gebruik

te maken: in feite is interferentie tussen aspecten een moeilijk beheersbaar effect, dat men bij het ontwerp bewust uitsluit. De auteurs [26] vergelijken hun benadering expliciet met een aanpak die op eigendomsrelaties is gebaseerd: waar die laatste specifiek zijn gericht op gevallen waarbij de aanroephiërarchie overeenstemt met de eigendomshiërarchie, is de aspectbenadering geschikt voor het verifiëren van condities die geheel los staan van de oorspronkelijke programmastructuur.

Dat aspecten onafhankelijk van elkaar ontworpen kunnen worden, wil niet zeggen dat ze onafhankelijk van de klassen bestaan. Integendeel: aspect-georiënteerd programmeren zoals het oorspronkelijk is bedacht is een techniek die zich volstrekt parasitair aan een gegeven object-georiënteerd programma hecht. De omstandigheden waaronder bepaalde code uit het aspect moet worden uitgevoerd, worden gekarakteriseerd aan de hand van de namen van attributen en methoden uit de objecten. Grotere groepen van dergelijke eenheden kunnen alleen worden benaderd via systematische naamgeving en het gebruik van *wildcards*. Niet alleen vereist dit grote discipline in de codering van de objecten, met een schuin oog naar de vereisten van de toe te voegen aspecten, het veroorzaakt ook dat eenmaal geprogrammeerde aspecten volledig specifiek zijn afgestemd op de objecten waarmee ze werken en niet voor hergebruik in andere programma's beschikbaar zijn.

De situatie is sterk verbeterd doordat bij de introductie van versie 5 van Java in september 2004 zogenaamde *annotaties* zijn ingevoerd [1]. Deze stellen de programmeur in staat codefragmenten op een systematische manier te markeren, onafhankelijk van de gekozen namen in het programma. Men kan bijvoorbeeld nog te verfijnen code markeren met `@preliminary` of objecten die moeten worden opgeslagen met `@persistent`. Aspecten kunnen behalve op de namen van programma-elementen nu ook selecteren op dergelijke annotaties. Als de annotaties in verschillende programma's op dezelfde wijze zijn gebruikt, komen aspecten daardoor voor hergebruik in aanmerking. In versie 3 van de Enterprise Javabeans [5] is hiervan een dankbaar gebruik gemaakt om de taken voor de programmeur te vereenvoudigen. Voor algemene toepassing van deze techniek ontbreekt echter nog een universele conventie voor syntax en semantiek van dergelijke markerings. Hiermee zij een bescheiden suggestie voor toekomstig nuttig werk gedaan.

Dankwoord

Aan het slot van deze rede is een woord van dank op zijn plaats.

Allereerst wil ik het College van Bestuur dankzeggen voor het in mij gestelde vertrouwen.

De medewerkers van de faculteit Informatica wil ik danken voor hun bereidheid mij met grote hartelijkheid in hun midden op te nemen en zelfs naar me te luisteren, ook als ik ideeën opperde die in strijd waren met lange tradities en eerder bedacht beleid. De laatste paar jaar is door iedereen in de faculteit ontzettend hard gewerkt aan het realiseren van nieuwe activiteiten, zoals

- de bacheloropleiding Informatiekunde,
- de masteropleiding Software Engineering,
- de opleiding IT Governance,
- de internationale Free Technology Academy,
- de Netwerk Open Hogeschool Informatica,
- het Europese ontwikkelproject Math-Bridge,
- de pilot Strategie Open Educational Resources,
- de voortentamens en voorbereidingscursussen wiskunde,
- het Nederlands Informatica-Onderwijscongres 2011

en zo kan ik nog geruime tijd doorgaan. Dat dit alles is ingepast zonder uitbreiding van de personeelsformatie was alleen mogelijk doordat allen zich tot het uiterste hebben ingezet, een resultaat dat mede te danken is aan de bijzonder constructieve sfeer en samenwerkingsbereidheid binnen deze faculteit.

Ook de decanen van de andere faculteiten wil ik bedanken voor de inspanning die zij zich getroost hebben mij als nieuwkomer wegwijs te maken binnen de Open Universiteit en mij de fijne kneepjes bij te brengen

die men niet uit het jaarverslag kan leren. Ook de voormalig hoogleraar-directeur van het Directoraat Onderwijs (als ik de antieke terminologie goed weergeef), Alexander Udink ten Cate, heeft geheel spontaan en vrijwillig een bijdrage aan mijn opvoeding geleverd. Hij schatte daarbij overigens dat ik het in deze functie maximaal vier jaar zou uithouden, een reden te meer dat het nu maar eens van een oratie moest komen.

Tot slot een woord van dank aan allen die mij gemaakt hebben wie ik nu ben: mijn ouders, die hier helaas niet meer bij kunnen zijn; de vele onderwijsgeevenden die mij kennis bijbrachten die ik grotendeels weer vergeten ben en inspiratie gaven die gebleven is; mijn promotor Henk Jager; coreferent en latere leidinggevende Piet Cijssouw. Het produceren van een goedlopende Nederlandse zin is me geleerd door de redactie van *Propria Cures*. In de informatica ben ik binnengevoerd door Wim Feijen en Anne Kaldewaij. Belangrijk voor de ontwikkeling van mijn stijl en smaak in dat vak zijn vier personen die helaas het afgelopen decennium allen zijn overleden, namelijk Edsger Dijkstra, Netty van Gasteren, Martin Rem en Carel Scholten. De weg naar een onderwijsmanagementfunctie heeft Doaitse Swierstra mij gewezen. Dat ik daarop niet verdwaald ben, is te danken aan de opleidingsmanagers Jeroen Fokker, Cilia Witteman, Lennart Herlaar, Hans Voorbij, Anda Counotte en de opleidingsdirecteur Frank Wester.

En tenslotte, dicht bij huis en dicht bij hart, in chronologische volgorde: Helen, Dennis, Timon, Ineke en toen warempel nogmaals Ineke. Dankzij jullie heeft het leven mij veel meer gebracht dan ik als principeel en systematisch pessimist ooit voor mogelijk had gehouden. Na 60 jaar heb ik de smaak echt te pakken en ik ga graag nog een tijdje door, maar voor nu

heb ik gezegd.

Bibliografie

- [1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison–Wesley, Reading, Massachusetts, fourth edition, 2005.
- [2] Mike Barnett, Robert deLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. http://www.jot.fm/issues/issue_2004_06/article2.
- [3] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In D. Kozen, editor, *Mathematics of Program Construction (MPC 2004)*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84, Berlin, 2004. Springer-Verlag.
- [4] Lex Bijlsma. Model-based specification. *Information Processing Letters*, 77:77–84, 2001.
- [5] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0*. O’Reilly Media, Sebastopol, California, fifth edition, 2006.
- [6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: a System of Patterns*. John Wiley & Sons, Chichester, 1996.
- [7] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors. *Structured Programming*. Academic Press, London, 1972.
- [8] Ole-Johan Dahl. *SIMULA 67 Common Base Language*. Norwegian Computing Center, 1968.
- [9] Edsger W. Dijkstra. The structure of the ‘THE’ multi-programming system. *Communications of the ACM*, 11:341–346, 1968.

- [10] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
- [11] Edsger W. Dijkstra. A solution designed by A. Blokhuis. <http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD979.PDF>, 1986.
- [12] Edsger W. Dijkstra. For the record: Painting the squared plane. <http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1212.PDF>, 1995.
- [13] Don Eyles. Tales from the lunar module guidance computer. *Advances in the Astronautical Sciences*, 118:455–473, 2004. <http://www.doneyles.com/LM/Tales.html>.
- [14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.
- [15] A.J.M. van Gasteren and A. Bijlsma. An extension of the program derivation format. In David Gries and Willem-Paul de Roever, editors, *Programming Concepts and Methods (PROCOMET98)*, pages 167–185, London, 1998. Chapman and Hall.
- [16] David Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.
- [17] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
- [18] C.A.R. Hoare. Hints on programming language design. Technical Report AIM-224/STAN-CS-73-403, Stanford University, 1973.
- [19] Kees Huizing, Ruurd Kuiper, and SOOP. Verification of object oriented programs using class invariants. In Tom Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE2000)*, volume 1783 of *Lecture Notes in Computer Science*, pages 208–221, Berlin, 2000. Springer-Verlag.
- [20] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *European Conference on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354, Berlin, 2004. Springer-Verlag.

- [21] Donald E. Knuth. Literate programming. *The Computer Journal*, 27:97–111, 1984.
- [22] Ivan A. Krizsan. Getting started with Contract4J. http://www.polyglotprogramming.com/papers/Getting_Started_with_Contract4J.pdf, 2008.
- [23] Gerrit Krol. *Het gemillimeterde hoofd: Schrijven met sommen*. Em. Querido, Amsterdam, 1967.
- [24] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, Greenwich, Connecticut, 2003.
- [25] Imre Lakatos. History of science and its rational reconstructions. In Colin Howson, editor, *Method and Appraisal in the Physical Sciences*, pages 1–40. Cambridge University Press, 1976.
- [26] Patrick Lam, Viktor Kuncak, and Martin Rinard. Crosscutting techniques in program specification and analysis. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 05)*, pages 169–180, New York, 2005. ACM Press.
- [27] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP 2004)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516, Berlin, 2004. Springer-Verlag.
- [28] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *Programming Languages and Systems: 15th European Symposium on Programming (ESOP 2006)*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130, Berlin, 2006. Springer-Verlag.
- [29] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24:491–553, 2002.
- [30] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In Rocco De Nicola, editor, *Programming Languages and Systems: 16th European Symposium on Programming (ESOP 2007)*, volume 4421 of *Lecture Notes in Computer Science*, pages 80–94, Berlin, 2007. Springer-Verlag.

- [31] Bertrand Meyer. Design by contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1987.
- [32] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice–Hall PTR, Upper Saddle River, New Jersey, second edition, 1997.
- [33] Bertrand Meyer. The dependent delegate dilemma. In Manfred Broy, J. Gruenbauer, David Harel, and C.A.R. Hoare, editors, *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series II: Mathematics, Physics and Chemistry*, pages 105–118, Berlin, 2005. Springer-Verlag.
- [34] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik Luit. Cooperation-based invariants for OO languages. *Electronic Notes in Theoretical Computer Science*, 160:225–237, 2005.
- [35] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. Invariants for non-hierarchical object structures. *Electronic Notes in Theoretical Computer Science*, 195:211–229, 2008.
- [36] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2002.
- [37] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
- [38] Peter G. Neumann. *Computer-Related Risks*. ACM Press, New York, 1995.
- [39] Matthew Parkinson. Class invariants: The end of the road? In Tobias Wrigstad, editor, *3rd International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO 2007)*, 2007. <http://www.cs.purdue.edu/homes/wrigstad/iwaco/p3-parkinson.pdf>.
- [40] SOOP Working Group. Model-based specification of design patterns. *Nieuwsbrief NVTI*, pages 9–14, 1999. <http://igitur-archive.library.uu.nl/math/2007-0206-202949/UUindex.html>.
- [41] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison–Wesley, Reading, Massachusetts, second edition, 2002.

- [42] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall PTR, Upper Saddle River, New Jersey, fourth edition, 2003.
- [43] Joseph Weizenbaum. *Computer Power and Human Reason: from Judgment to Calculation*. W.H. Freeman, San Francisco, 1976.
- [44] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, 1971.